# The Racket Foreign Interface

Version 9.4.0.3

Eli Barzilay

November 2, 2026

```
(require ffi/unsafe)
```
package: `base`

The `ffi/unsafe` library enables the direct use of C-based APIs within Racket programs—without writing any new C code. From the Racket perspective, functions and data with a C-based API are *foreign*, hence the term *foreign interface*. Furthermore, since most APIs consist mostly of functions, the foreign interface is sometimes called a *foreign function interface*, abbreviated *FFI*.

# Contents


**1 Overview** **6**

1.1 Libraries, C Types, and Objects . . . . . 6

1.2 Function-Type Bells and Whistles . . . . . 7

1.3 By-Reference Arguments . . . . . 8

1.4 C Structs . . . . . 9

1.5 Pointers and Manual Allocation . . . . . 10

1.6 Pointers and GC-Managed Allocation . . . . . 11

1.7 Reliable Release of Resources . . . . . 12

1.8 Threads and Places . . . . . 13

1.9 More Examples . . . . . 14

**2 Loading Foreign Libraries** **15**

**3 C Types** **19**

3.1 Type Constructors . . . . . 19

3.2 Numeric Types . . . . . 21

3.3 Other Atomic Types . . . . . 23

3.4 String Types . . . . . 24

3.4.1 Primitive String Types . . . . . 24

3.4.2 Fixed Auto-Converting String Types . . . . . 25

3.4.3 Variable Auto-Converting String Type . . . . . 25

3.4.4 Other String Types . . . . . 26

3.5 Pointer Types . . . . . 26

3.6 Function Types . . . . . 28

3.6.1 Custom Function Types . . . . . 36

3.7 C Struct Types . . . . . 41
3.8 C Array Types . . . . . 48
3.9 C Union Types . . . . . 50
3.10 Enumerations and Masks . . . . . 52

**4 Pointer Functions** **55**

4.1 Pointer Dereferencing . . . . . 56
4.2 Memory Management . . . . . 59
4.3 Pointer Structure Property . . . . . 64

**5 Derived Utilities** **65**

5.1 Safe Homogenous Vectors . . . . . 65
5.2 Safe C Vectors . . . . . 72
5.3 Tagged C Pointer Types . . . . . 74
5.4 Serializable C Struct Types . . . . . 77
5.5 Static Callout and Callback Cores . . . . . 80
5.6 Defining Bindings . . . . . 80
5.6.1 FFI Identifier Conventions . . . . . 83
5.7 Allocation and Finalization . . . . . 84
5.8 Custodian Shutdown Registration . . . . . 86
5.9 Atomic Execution . . . . . 88
5.10 Speculatively Atomic Execution . . . . . 92
5.11 Thread Scheduling . . . . . 93
5.12 Ports . . . . . 95
5.13 Process-Wide and Place-Wide Registration . . . . . 98
5.14 Operating System Threads . . . . . 98

5.14.1 Operating System Asynchronous Channels . . . . . 99
5.15 Garbage Collection Callbacks . . . . . 100
5.16 Objective-C FFI . . . . . 102
5.16.1 FFI Types and Constants . . . . . 102
5.16.2 Syntactic Forms and Procedures . . . . . 103
5.16.3 Raw Runtime Functions . . . . . 109
5.16.4 Legacy Library . . . . . 112
5.17 Cocoa Foundation . . . . . 112
5.17.1 Strings . . . . . 112
5.17.2 Allocation Pools . . . . . 112
5.18 COM (Common Object Model) . . . . . 113
5.18.1 COM Automation . . . . . 114
5.18.2 COM Classes and Interfaces . . . . . 124
5.18.3 ActiveX Controls . . . . . 131
5.19 File Security-Guard Checks . . . . . 132
5.20 Windows API Helpers . . . . . 133
5.21 Windows VCRuntime Librraies . . . . . 134
5.22 Virtual Machine Primitives . . . . . 134
5.23 Declaring Foreign Libraries Needed at Run Time . . . . . 136

**6 Miscellaneous Support** **138**

**7 Unexported Primitive Functions** **141**

**Bibliography** **144**

**Index** **145**

**Index** **145**

# 1 Overview

Although using the FFI requires writing no new C code, it provides very little insulation against the issues that C programmers face related to safety and memory management. An FFI programmer must be particularly aware of memory management issues for data that spans the Racket–C divide. Thus, this manual relies in many ways on the information in *Inside: Racket C API*, which defines how Racket interacts with C APIs in general.

Since using the FFI entails many safety concerns that Racket programmers can normally ignore, the library name includes `unsafe`. Importing the library macro should be considered as a declaration that your code is itself unsafe, therefore can lead to serious problems in case of bugs: it is your responsibility to provide a safe interface. If your library provides an unsafe interface, then it should have `unsafe` in its name, too.

For more information on the motivation and design of the Racket FFI, see [Barzilay04].

## 1.1 Libraries, C Types, and Objects

To use the FFI, you must have in mind

- a particular library from which you want to access a function or value,
- a particular symbol exported by the file, and
- the C-level type (typically a function type) of the exported symbol.

The library corresponds to a file with a suffix such as `".dll"`, `".so"`, or `".dylib"` (depending on the platform), or it might be a library within a `".framework"` directory on Mac OS.

Knowing the library's name and/or path is often the trickiest part of using the FFI. Sometimes, when using a library name without a path prefix or file suffix, the library file can be located automatically, especially on Unix. See `ffi-lib` for advice.

The `ffi-lib` function gets a handle to a library. To extract exports of the library, it's simplest to use `define-ffi-definer` from the `ffi/unsafe/define` library:

```
#lang racket/base
(require ffi/unsafe
         ffi/unsafe/define)

(define-ffi-definer define-curses (ffi-lib "libcurses"))
```

This `define-ffi-definer` declaration introduces a `define-curses` form for binding

a Racket name to a value extracted from `"libcurses"`—which might be located at `"/usr/lib/libcurses.so"`, depending on the platform.

To use `define-curses`, we need the names and C types of functions from `"libcurses"`. We'll start by using the following functions:

```
WINDOW* initscr(void);
int waddstr(WINDOW *win, char *str);
int wrefresh(WINDOW *win);
int endwin(void);
```

We make these functions callable from Racket as follows:

```
(define _WINDOW-pointer (_cpointer 'WINDOW))

(define-curses initscr (_fun -> _WINDOW-pointer))
(define-curses waddstr (_fun _WINDOW-pointer _string -> _int))
(define-curses wrefresh (_fun _WINDOW-pointer -> _int))
(define-curses endwin (_fun -> _int))
```

By convention, an underscore prefix indicates a representation of a C type (such as `_int`) or a constructor of such representations (such as `_cpointer`).

The definition of `_WINDOW-pointer` creates a Racket value that reflects a C type via `_cpointer`, which creates a type representation for a pointer type—usually one that is opaque. The `'WINDOW` argument could have been any value, but by convention, we use a symbol matching the C base type.

Each `define-curses` form uses the given identifier as both the name of the library export and the Racket identifier to bind. The `(_fun ... -> ...)` part of each definition describes the C type of the exported function, since the library file does not encode that information for its exports. The types listed to the left of `->` are the argument types, while the type to the right of `->` is the result type. The pre-defined `_int` type naturally corresponds to the `int` C type, while `_string` corresponds to the `char*` type when it is intended as a string to read.

An optional `#:c-id` clause for `define-curses` can specify a name for the library export that is different from the Racket identifier to bind.

At this point, `initscr`, `waddstr`, `wrefresh`, and `endwin` are normal Racket bindings to Racket functions (that happen to call C functions), and so they can be exported from the defining module or called directly:

```
(define win (initscr))
(void (waddstr win "Hello"))
(void (wrefresh win))
(sleep 1)
(void (endwin))
```

## 1.2 Function-Type Bells and Whistles

Our initial use of functions like `waddstr` is sloppy, because we ignore return codes. C functions often return error codes, and checking them is a pain. A better approach is to build

the check into the `waddstr` binding and raise an exception when the code is non-zero.

The `_fun` function-type constructor includes many options to help convert C functions to nicer Racket functions. We can use some of those features to convert return codes into either `#<void>` or an exception:

```
(define (check v who)
  (unless (zero? v)
    (error who "failed: ~a" v)))

(define-curses initscr (_fun -> _WINDOW-pointer))
(define-curses waddstr (_fun _WINDOW-pointer _string -> (r : _int)
                             -> (check r 'waddstr)))
(define-curses wrefresh (_fun _WINDOW-pointer -> (r : _int)
                              -> (check r 'wrefresh)))
(define-curses endwin (_fun -> (r : _int)
                            -> (check r 'endwin)))
```

Using `(r : _int)` as a result type gives the local name `r` to the C function's result. This name is then used in the result post-processing expression that is specified after a second `->` in the `_fun` form.

## 1.3 By-Reference Arguments

To get mouse events from `"libcurses"`, we must explicitly enable them through the `mousemask` function:

```
typedef unsigned long mmask_t;
#define BUTTON1_CLICKED 004L

mmask_t mousemask(mmask_t newmask, mmask_t *oldmask);
```

Setting `BUTTON1_CLICKED` in the mask enables button-click events. At the same time, `mousemask` returns the current mask by installing it into the pointer provided as its second argument.

Since these kinds of call-by-reference interfaces are common in C, `_fun` cooperates with a `_ptr` form to automatically allocate space for a by-reference argument and extract the value put there by the C function. Give the extracted value name to use in the post-processing expression. The post-processing expression can combine the by-reference result with the function's direct result (which, in this case, reports a subset of the given mask that is actually supported).

```
(define _mmask_t _ulong)
(define-curses mousemask (_fun _mmask_t (o : (_ptr o _mmask_t))
```

```
                                -> (r : _mmask_t)
                                -> (values o r)))
(define BUTTON1_CLICKED 4)

(define-values (old supported) (mousemask BUTTON1_CLICKED))
```

## 1.4 C Structs

Assuming that mouse events are supported, the `"libcurses"` library reports them via `getmouse`, which accepts a pointer to a `MEVENT` struct to fill with mouse-event information:

```
typedef struct {
   short id;
   int x, y, z;
   mmask_t bstate;
} MEVENT;

int getmouse(MEVENT *event);
```

To work with `MEVENT` values, we use `define-cstruct`:

```
(define-cstruct _MEVENT ([id _short]
                         [x _int]
                         [y _int]
                         [z _int]
                         [bstate _mmask_t]))
```

This definition binds many names in the same way that `define-struct` binds many names: `_MEVENT` is a C type representing the struct type, `_MEVENT-pointer` is a C type representing a pointer to a `_MEVENT`, `make-MEVENT` constructs a `_MEVENT` value, `MEVENT-x` extracts the `x` fields from an `_MEVENT` value, and so on.

With this C struct declaration, we can define the function type for `getmouse`. The simplest approach is to define `getmouse` to accept an `_MEVENT-pointer`, and then explicitly allocate the `_MEVENT` value before calling `getmouse`:

```
(define-curses getmouse (_fun _MEVENT-pointer -> _int))

(define m (make-MEVENT 0 0 0 0 0))
(when (zero? (getmouse m))
  ; use m...
  ....)
```

For a more Racket-like function, use `(_ptr o _MEVENT)` and a post-processing expression:

```
(define-curses getmouse (_fun (m : (_ptr o _MEVENT))
                              -> (r : _int)
                              -> (and (zero? r) m)))

(waddstr win (format "click me fast..."))
(wrefresh win)
(sleep 1)

(define m (getmouse))
(when m
  (waddstr win (format "at ~a,~a"
                       (MEVENT-x m)
                       (MEVENT-y m)))
  (wrefresh win)
  (sleep 1))

(endwin)
```

The difference between `_MEVENT-pointer` and `_MEVENT` is crucial. Using `(_ptr o _MEVENT-pointer)` would allocate only enough space for a pointer to an `MEVENT` struct, which is not enough space for an `MEVENT` struct.

## 1.5 Pointers and Manual Allocation

To get text from the user instead of a mouse click, `"libcurses"` provides `wgetnstr`:

```
int wgetnstr(WINDOW *win, char *str, int n);
```

While the `char*` argument to `waddstr` is treated as a nul-terminated string, the `char*` argument to `wgetnstr` is treated as a buffer whose size is indicated by the final `int` argument. The C type `_string` does not work for such buffers.

One way to approach this function from Racket is to describe the arguments in their rawest form, using plain `_pointer` for the second argument to `wgetnstr`:

```
(define-curses wgetnstr (_fun _WINDOW-pointer _pointer _int -> _int))
```

To call this raw version of `wgetnstr`, allocate memory, zero it, and pass the size minus one (to leave room a nul terminator) to `wgetnstr`:

```
(define SIZE 256)
(define buffer (malloc 'raw SIZE))
(memset buffer 0 SIZE)

(void (wgetnstr win buffer (sub1 SIZE)))
```

When `wgetnstr` returns, it has written bytes to `buffer`. At that point, we can use `cast` to convert the value from a raw pointer to a string:

```
(cast buffer _pointer _string)
```

Conversion via the `_string` type causes the data referenced by the original pointer to be copied (and UTF-8 decoded), so the memory referenced by `buffer` is no longer needed. Memory allocated with `(malloc 'raw ...)` must be released with `free`:

```
(free buffer)
```

## 1.6 Pointers and GC-Managed Allocation

Instead of allocating `buffer` with `(malloc 'raw ...)`, we could have allocated it with `(malloc 'atomic ...)`:

```
(define buffer (malloc 'atomic SIZE))
```

Memory allocated with `'atomic` is managed by the garbage collector, so `free` is neither necessary nor allowed when the memory referenced by `buffer` is no longer needed. Instead, when `buffer` becomes inaccessible, the allocated memory will be reclaimed automatically.

Allowing the garbage collector (GC) to manage memory is usually preferable. It's easy to forget to call `free`, and exceptions or thread termination can easily skip a `free`.

At the same time, using GC-managed memory adds a different burden on the programmer: data managed by the GC may be moved to a new address as the GC compacts allocated objects to avoid fragmentation. C functions, meanwhile, expect to receive pointers to objects that will stay put.

Fortunately, unless a C function calls back into the Racket runtime system (perhaps through a function that is provided as an argument), no garbage collection will happen between the time that a C function is called and the time that the function returns.

Let's look a few possibilities related to allocation and pointers:

- Ok:

  ```
  (define p (malloc 'atomic SIZE))
  (wgetnstr win p (sub1 SIZE))
  ```

  Although the data allocated by `malloc` can move around, `p` will always point to it, and no garbage collection will happen between the time that the address is extracted from `p` to pass to `wgetnstr` and the time that `wgetnstr` returns.

- Bad:

```
(define p (malloc 'atomic SIZE))
(define i (cast p _pointer _intptr))
(wgetnstr win (cast i _intptr _pointer) (sub1 SIZE))
```

  The data referenced by `p` can move after the address is converted to an integer, in which case `i` cast back to a pointer will be the wrong address.

  Obviously, casting a pointer to an integer is generally a bad idea, but the cast simulates another possibility, which is passing the pointer to a C function that retains the pointer in its own private store for later use. Such private storage is invisible to the Racket GC, so it has the same effect as casting the pointer to an integer.

- Ok:

```
(define p (malloc 'atomic SIZE))
(define p2 (ptr-add p 4))
(wgetnstr win p2 (- SIZE 5))
```

  The pointer `p2` retains the original reference and only adds the `4` at the last minute before calling `wgetnstr` (i.e., after the point that garbage collection is allowed).

- Ok:

```
(define p (malloc 'atomic-interior SIZE))
(define i (cast p _pointer _intptr))
(wgetnstr win (cast i _intptr _pointer) (sub1 SIZE))
```

  This is ok assuming that `p` itself stays accessible, so that the data it references isn't reclaimed. Allocating with `'atomic-interior` puts data at a particular address and keeps it there. A garbage collection will not change the address in `p`, and so `i` (cast back to a pointer) will always refer to the data.

Keep in mind that C struct constructors like `make-MEVENT` are effectively the same as `(malloc 'atomic ...)`; the result values can move in memory during a garbage collection. The same is true of byte strings allocated with `make-bytes`, which (as a convenience) can be used directly as a pointer value (unlike character strings, which are always copied for UTF-8 encoding or decoding).

For more information about memory management and garbage collection, see §1.2 "Racket CS Memory Management" and §12 "Memory Allocation (BC)" in *Inside: Racket C API*.

## 1.7 Reliable Release of Resources

Using GC-managed memory saves you from manual `free`s for plain memory blocks, but C libraries often allocate resources and require a matching call to a function that releases the

resources. For example, "libcurses" supports windows on the screen that are created with newwin and released with delwin:

```
WINDOW *newwin(int lines, int ncols, int y, int x);
int delwin(WINDOW *win);
```

In a sufficiently complex program, ensuring that every newwin is paired with delwin can be challenging, especially if the functions are wrapped by otherwise safe functions that are provided from a library. A library that is intended to be safe for use in a sandbox, say, must protect against resource leaks within the Racket process as a whole when a sandboxed program misbehaves or is terminated.

The ffi/unsafe/alloc library provides functions to connect resource-allocating functions and resource-releasing functions. The library then arranges for finalization to release a resource if it becomes inaccessible (according to the GC) before it is explicitly released. At the same time, the library handles tricky atomicity requirements to ensure that the finalization is properly registered and never run multiple times.

Using ffi/unsafe/alloc, the newwin and delwin functions can be imported with allocator and deallocator wrappers, respectively:

```
(require ffi/unsafe/alloc)

(define-curses delwin (_fun _WINDOW-pointer -> _int)
  #:wrap (deallocator))

(define-curses newwin (_fun _int _int _int _int
                            -> _WINDOW-pointer)
  #:wrap (allocator delwin))
```

A deallocator wrapper makes a function cancel any existing finalizer for the function's argument. An allocator wrapper refers to the deallocator, so that the deallocator can be run if necessary by a finalizer.

If a resource is scarce or visible to end users, then custodian management is more appropriate than mere finalization as implemented by allocator. See the ffi/unsafe/custodian library.

## 1.8 Threads and Places

Although older versions of "libcurses" are not thread-safe, Racket threads do not correspond to OS-level threads, so using Racket threads to call "libcurses" functions creates no particular problems.

Racket places, however, correspond to OS-level threads. Using a foreign library from mul-

tiple places works when the library is thread-safe. Calling a non-thread-safe library from multiple places requires more care.

The simplest way to use a non-thread-safe library from multiple places is to specify the `#:in-original-place? #t` option of `_fun`, which routes every call to the function through the original Racket place instead of the calling place. Most of the functions that we initially used from `"libcurses"` can be made thread-safe simply:

```
(define-curses initscr
  (_fun #:in-original-place? #t -> _WINDOW-pointer))
(define-curses wrefresh
  (_fun #:in-original-place? #t _WINDOW-pointer -> _int))
(define-curses endwin
  (_fun #:in-original-place? #t -> _int))
```

The `waddstr` function is not quite as straightforward. The problem with

```
(define-curses waddstr
  (_fun #:in-original-place? #t _WINDOW-pointer _string -> _int))
```

is that the string argument to `waddstr` might move in the calling place before the `waddstr` call completes in the original place. To safely call `waddstr`, we can use a `_string/utf-8/immobile` type that allocates bytes for the string argument with `'atomic-interior`:

```
(define-curses waddstr
  (_fun #:in-original-place? #t _WINDOW-pointer _string/utf-
8/immobile -> _int))
```

Beware that passing memory allocated with `'interior` (as opposed to `'atomic-interior`) is safe only for functions that read the memory without writing. A foreign function must not write to `'interior`-allocated memory from a place other than the one where the memory is allocated, due a place-specific treatment of writes to implement generational garbage collection.

## 1.9 More Examples

For more examples of common FFI patterns, see the defined interfaces in the `"ffi/examples"` collection. See also [Barzilay04].

# 2 Loading Foreign Libraries

The FFI is normally used by extracting functions and other objects from shared objects (a.k.a. *shared libraries* or *dynamically loaded libraries*). The `ffi-lib` function loads a shared object.

```
(ffi-lib? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a *foreign-library value*, `#f` otherwise.

```
(ffi-lib  path
          [version
           #:get-lib-dirs get-lib-dirs
           #:fail fail
           #:global? global?
           #:custodian custodian])      → any
  path : (or/c path-string? #f)
  version : (or/c string? (listof (or/c string? #f)) #f) = #f
  get-lib-dirs : (-> (listof path?)) = get-lib-search-dirs
  fail : (or/c #f (-> any)) = #f
  global? : any/c = (eq? 'global (system-type 'so-mode))
  custodian : (or/c 'place custodian? #f) = #f
```

Returns a foreign-library value or the result of *fail*. Normally,

- *path* is a path without a version or suffix (i.e., without `".dll"`, `".so"`, or `".dylib"`); and
- *version* is a list of versions to try in order with `#f` (i.e., no version) as the last element of the list; for example, `'("2" #f)` indicates version 2 with a fallback to a versionless library.

  When the library suffix as reported by `(system-type 'so-suffix)` is `".dylib"`, then a version is added to *path* after a `"."` and before the `".dylib"` suffix. When the library suffix is `".dll"`, then a version is added to *path* after a `"-"` and before the `".dll"` suffix. For any other suffix, the version number is added after the suffix plus `"."`.

A string or `#f` *version* is equivalent to a list containing just the string or `#f`, and an empty string (by itself or in a list) is equivalent to `#f`.

Beware of relying on versionless library names. On some platforms, versionless library names are provided only by development packages. At the same time, other platforms may

require a versionless fallback. A list of version strings followed by `#f` is typically best for *version*.

Assuming that *path* is not `#f`, the result from `ffi-lib` represents the library found by the following search process:

- If *path* is not an absolute path, look in each directory reported by *get-lib-dirs*; the default list is the result of `(get-lib-search-dirs)`. In each directory, try *path* with the first version in *version*, adding a suitable suffix if *path* does not already end in the suffix, then try the second version in *version*, etc. (If *version* is an empty list, no paths are tried in this step.)
- Try the same filenames again, but without converting the path to an absolute path, which allows the operating system to use its own search paths. (If *version* is an empty list, no paths are tried in this step.)
- Try *path* without adding any version or suffix, and without converting to an absolute path.
- Try the version-adjusted filenames again, but relative to the current directory. (If *version* is an empty list, no paths are tried in this step.)
- Try *path* without adding any version or suffix, but converted to an absolute path relative to the current directory.

If none of the paths succeed and *fail* is a function, then *fail* is called in tail position. If *fail* is `#f`, an error is reported from trying the first path from the second bullet above or (if *version* is an empty list) from the third bullet above. A library file may exist but fail to load for some reason; the eventual error message will unfortunately name the fallback from the second or third bullet, since some operating systems offer no way to determine why a given library path failed.

If *path* is `#f`, then the resulting foreign-library value represents all libraries loaded in the current process, including libraries previously opened with `ffi-lib`. In particular, use `#f` to access C-level functionality exported by the run-time system (as described in *Inside: Racket C API*). The *version* argument is ignored when *path* is `#f`.

If *path* is not `#f`, *global?* is true, and the operating system supports opening a library in "global" mode so that the library's symbols are used for resolving references from libraries that are loaded later, then global mode is used to open the library. Otherwise, the library is opened in "local" mode, where the library's symbols are not made available for future resolution. This local-versus-global choice does not affect whether the library's symbols are available via `(ffi-lib #f)`.

If *custodian* is `'place` or a custodian, the library is unloaded when a custodian is shut down—either the given custodian or the place's main custodian if *custodian* is `'place`. When a library is unloaded, all references to the library become invalid. Supplying `'place`

for *custodian* is consistent with finalization via ffi/unsafe/alloc but will not, for example, unload the library when hitting in the Run button in DrRacket. Supplying (current-custodian) for *custodian* tends to unload the library for eagerly, but requires even more care to ensure that library references are not accessed after the library is unloaded.

If *custodian* is #f, the loaded library is associated with Racket (or DrRacket) for the duration of the process. Loading again with ffi-lib, will not force a re-load of the corresponding library.

When ffi-lib returns a reference to a library that was previously loaded within the current place, it increments a reference count on the loaded library rather than loading the library fresh. Unloading a library reference decrements the reference count and requests unloading at the operating-system level only if the reference count goes to zero.

The ffi-lib procedure logs (see §15.5 "Logging") on the topic 'ffi-lib. In particular, on failure it logs the paths attempted according to the rules above, but it cannot report the paths tried due to the operating system's library search path.

Changed in version 6.1.0.5 of package base: Changed the way a version number is added with a ".dll" suffix to place it before the suffix, instead of after.
Changed in version 7.3.0.3: Added logging.
Changed in version 7.4.0.7: Added the #:custodian argument.

```
(get-ffi-obj objname lib type [failure-thunk]) → any
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  type : ctype?
  failure-thunk : (or/c (-> any) #f) = #f
```

Looks for *objname* in *lib* library. If *lib* is not a foreign-library value it is converted to one by calling ffi-lib. If *objname* is found in *lib*, it is converted to Racket using the given *type*. Types are described in §3 "C Types"; in particular the get-ffi-obj procedure is most often used with function types created with _fun.

Keep in mind that get-ffi-obj is an unsafe procedure; see §1 "Overview" for details.

If the name is not found, and *failure-thunk* is provided, it is used to produce a return value. For example, a failure thunk can be provided to report a specific error if a name is not found:

```
(define foo
  (get-ffi-obj "foo" foolib (_fun _int -> _int)
    (lambda ()
      (error 'foolib
             "installed foolib does not provide \"foo\""))))
```

The default (also when *failure-thunk* is provided as #f) is to raise an exception.

```
(set-ffi-obj! objname lib type new) → void?
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  type : ctype?
  new : any/c
```

Looks for *objname* in *lib* similarly to `get-ffi-obj`, but then it stores the given *new* value into the library, converting it to a C value. This can be used for setting library customization variables that are part of its interface, including Racket callbacks.

```
(make-c-parameter  objname
                   lib
                   type
                  [failure-thunk])
 → (case-> (-> any)
           (any/c . -> . void?))
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  type : ctype?
  failure-thunk : (or/c (-> any) #f) = #f
```

Returns a parameter-like procedure that can either references the specified foreign value, or set it. The arguments are handled as in `get-ffi-obj`.

A parameter-like function is useful in case Racket code and library code interact through a library value. Although `make-c-parameter` can be used with any type, it is not recommended to use this for foreign functions, since each reference through the parameter will construct the low-level interface before the actual call.

Changed in version 8.4.0.5 of package `base`: Added *failure-thunk* argument.

```
(define-c id lib-expr type-expr)
```

Defines *id* behave like a Racket binding, but *id* is actually redirected through a parameter-like procedure created by `make-c-parameter`. The *id* is used both for the Racket binding and for the foreign name.

```
(ffi-obj-ref objname lib [failure-thunk]) → any
  objname : (or/c string? bytes? symbol?)
  lib : (or/c ffi-lib? path-string? #f)
  failure-thunk : (or/c (-> any) #f) = #f
```

Returns a pointer for the specified foreign name, calls *failure-thunk* if the name is not found, or raises an exception if *failure-thunk* is `#f`.

Normally, `get-ffi-obj` should be used, instead.

# 3 C Types

*C types* are the main concept of the FFI, either primitive types or user-defined types. The FFI deals with primitive types internally, converting them to and from C types. A user type is defined in terms of existing primitive and user types, along with conversion functions to and from the existing types.

## 3.1 Type Constructors

```
(make-ctype type racket-to-c c-to-racket) → ctype?
  type : ctype?
  racket-to-c : (or/c #f (any/c . -> . any))
  c-to-racket : (or/c #f (any/c . -> . any))
```

Creates a new C type value whose representation for foreign code is the same as *type*'s.

The given conversion functions convert to and from the Racket representation of the new type. Either conversion function can be `#f`, meaning that the conversion for the corresponding direction is the identity function. If both functions are `#f`, *type* is returned.

The *racket-to-c* function takes any value and, if it is a valid representation of the new type, converts it to a representation of *type*. The *c-to-racket* function takes a representation of *type* and produces a representation of the new type.

When the result type is used for an argument in a foreign call, beware that only the original argument value is specifically retained for the call, and not a result of *racket-to-c*. If the foreign call leads to a Racket callback, a garbage collection during the callback may move or reclaim an argument value that is otherwise unreferenced. Consider registering a mapping from the argument to result of *racket-to-c* in an ephemeron hash table so that the result remains reachable as long as the argument is reachable.

```
(ctype? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a C type, `#f` otherwise.

Examples:

```
> (ctype? _int)
#t
> (ctype? (_fun _int -> _int))
#t
> (ctype? #f)
#f
```

```
> (ctype? "foo")
#f
```

```
(ctype-sizeof type) → exact-nonnegative-integer?
  type : ctype?
(ctype-alignof type) → exact-nonnegative-integer?
  type : ctype?
```

Returns the size or alignment of a given *type* for the current platform.

Examples:

```
> (ctype-sizeof _int)
4
> (ctype-sizeof (_fun _int -> _int))
8
> (ctype-alignof _int)
4
> (ctype-alignof (_fun _int -> _int))
8
```

```
(ctype->layout type)
 → (flat-rec-contract rep symbol? (listof rep))
  type : ctype?
```

Returns a value to describe the eventual C representation of the type. It can be any of the following symbols:

```
'int8 'uint8 'int16 'uint16 'int32 'uint32 'int64 'uint64
'float 'double 'bool 'void 'pointer 'fpointer
'bytes 'string/ucs-4 'string/utf-16
```

The result can also be a list, which describes a C struct whose element representations are provided in order within the list. Finally, the result can be a vector of size 2 containing an element representation followed by an exact-integer count.

Examples:

```
> (ctype->layout _int)
'int32
> (ctype->layout _void)
'void
> (ctype->layout (_fun _int -> _int))
'fpointer
```

```
(compiler-sizeof sym) → exact-nonnegative-integer?
  sym : (or/c symbol? (listof symbol?))
```

Possible values for *sym* are `'int`, `'char`, `'wchar`, `'short`, `'long`, `'*`, `'void`, `'float`, `'double`, or lists of symbols, such as `'(long long)`. The result is the size of the corresponding type according to the C `sizeof` operator for the current platform. The `compiler-sizeof` operation should be used to gather information about the current platform, such as defining alias type like `_int` to a known type like `_int32`.

Examples:

```
> (compiler-sizeof 'int)
4
> (compiler-sizeof '(long long))
8
```

## 3.2 Numeric Types

```
_int8 : ctype?
_sint8 : ctype?
_uint8 : ctype?
_int16 : ctype?
_sint16 : ctype?
_uint16 : ctype?
_int32 : ctype?
_sint32 : ctype?
_uint32 : ctype?
_int64 : ctype?
_sint64 : ctype?
_uint64 : ctype?
```

The basic integer types at various sizes. The `s` or `u` prefix specifies a signed or an unsigned integer, respectively; the ones with no prefix are signed.

```
_byte : ctype?
_sbyte : ctype?
_ubyte : ctype?
```

The `_sbyte` and `_ubyte` types are aliases for `_sint8` and `_uint8`, respectively. The `_byte` type is like `_ubyte`, but adds 256 to a negative Racket value that would work as a `_sbyte` (i.e., it casts signed bytes to unsigned bytes).

```
_wchar : ctype?
```

The _wchar type is an alias for an unsigned integer type, such as _uint16 or _uint32, corresponding to the platform's wchar_t type.

Added in version 7.0.0.3 of package base.

```
_word : ctype?
_sword : ctype?
_uword : ctype?
```

The _sword and _uword types are aliases for _sint16 and _uint16, respectively. The _word type is like _uword, but coerces negative values in the same way as _byte.

```
_short : ctype?
_sshort : ctype?
_ushort : ctype?
_int : ctype?
_sint : ctype?
_uint : ctype?
_long : ctype?
_slong : ctype?
_ulong : ctype?
_llong : ctype?
_sllong : ctype?
_ullong : ctype?
_intptr : ctype?
_sintptr : ctype?
_uintptr : ctype?
```

Aliases for basic integer types. The _short aliases correspond to _int16. The _int aliases correspond to _int32. The _long aliases correspond to either _int32 or _int64, depending on the platform. Similarly, the _intptr aliases correspond to either _int32 or _int64, depending on the platform.

```
_size : ctype?
_ssize : ctype?
_ptrdiff : ctype?
_intmax : ctype?
_uintmax : ctype?
```

More aliases for basic integer types. The _size and _uintmax types are aliases for _uintptr, and the rest are aliases for _intptr.

```
_fixnum : ctype?
_ufixnum : ctype?
```

For cases where speed matters and where you know that the integer is small enough, the types _fixnum and _ufixnum are similar to _intptr and _uintptr but assume that the quantities fit in Racket's immediate integers (i.e., not bignums).

```
_fixint : ctype?
_ufixint : ctype?
```

Similar to _fixnum/_ufixnum, but based on _int/_uint instead of _intptr/_uintptr, and coercions from C are checked to be in range.

```
_float : ctype?
_double : ctype?
_double* : ctype?
```

The _float and _double types represent the corresponding C types. Both single- and double-precision Racket numbers are accepted for conversion via both _float and _double, while both _float and _double coerce C values to double-precision Racket numbers. The type _double* coerces any Racket real number to a C double.

```
_longdouble : ctype?
```

Represents the long double type on platforms where it is supported, in which case Racket extflonums convert to and from long double values.

## 3.3 Other Atomic Types

```
_stdbool : ctype?
```

The _stdbool type represents the C99 bool type from <stdbool.h>. Going from Racket to C, _stdbool translates #f to a 0 bool and any other value to a 1 bool. Going from C to Racket, _stdbool translates 0 to a #f and any other value to #t.

Added in version 6.0.0.6 of package base.

```
_bool : ctype?
```

Like _stdbool, but with an int representation on the C side, reflecting one of many traditional (i.e., pre-C99) encodings of booleans.

```
_void : ctype?
```

Indicates a Racket #<void> return value, and it cannot be used to translate values to C. This type cannot be used for function inputs.

## 3.4 String Types

### 3.4.1 Primitive String Types

See also `_bytes/nul-terminated` and `_bytes` for converting between byte strings and C's `char*` type.

`_string/ucs-4` : `ctype?`

A type for UCS-4 format strings that include a nul terminator. As usual, the type treats `#f` as `NULL` and vice versa.

For the CS implementation of Racket, the conversion of a Racket string for the foreign side is a copy of the Racket representation, where the copy is managed by the garbage collector.

For the BC implementation of Racket, the conversion of a Racket string for the foreign side shares memory with the Racket string representation, since UCS-4 is the native representation format for those variants. The foreign pointer corresponds to the `mzchar*` type in Racket's C API.

`_string/utf-16` : `ctype?`

Unicode strings in UTF-16 format that include a nul terminator. As usual, the types treat `#f` as `NULL` and vice versa.

The conversion of a Racket string for the foreign side is a copy of the Racket representation (reencoded), where the copy is managed by the garbage collector.

`_path` : `ctype?`

Simple `char*` strings that are nul terminated, corresponding to Racket's path or string. As usual, the type treats `#f` as `NULL` and vice versa.

For the BC implementation of Racket, the conversion of a Racket path for the foreign side shares memory with the Racket path representation. Otherwise (for the CS implementation or for Racket strings), conversion for the foreign side creates a copy that is managed by the garbage collector.

Beware that changing the current directory via `current-directory` does n<ot change the OS-level current directory as seen by foreign library functions. Paths normally should be converted to absolute form using `path->complete-path` (which uses the `current-directory` parameter) before passing them to a foreign function.

`_symbol` : `ctype?`

Simple `char*` strings as Racket symbols (encoded in UTF-8 and nul terminated), intended as read-only for the foreign side. Return values using this type are interned as symbols.

For the CS implementation of Racket, the conversion of a Racket symbol for the foreign side is a copy of the Racket representation, where the copy is managed by the garbage collector.

For the BC implementation of Racket, the conversion of a Racket symbol for the foreign side shares memory with the Racket symbol representation, but points to the middle of the symbol's allocated memory—so the string pointer must not be used across a garbage collection.

### 3.4.2 Fixed Auto-Converting String Types

```
_string/utf-8 : ctype?
_string/latin-1 : ctype?
_string/locale : ctype?
```

Types that correspond to (character) strings on the Racket side and `char*` strings on the C side. The bridge between the two requires a transformation on the content of the string. As usual, the types treat `#f` as `NULL` and vice versa.

```
_string/utf-8/immobile : ctype?
```

Like `_string/utf-8`, but the Racket-to-C conversion allocates the UTF-8 bytes and a nul terminator with `malloc` using `'atomic-interior`.

Added in version 9.4.0.2 of package `base`.

```
_string*/utf-8 : ctype?
_string*/latin-1 : ctype?
_string*/locale : ctype?
```

Similar to `_string/utf-8`, etc., but accepting a wider range of values: Racket byte strings are allowed and passed as is, and Racket paths are converted using `path->bytes`.

### 3.4.3 Variable Auto-Converting String Type

The `_string/ucs-4` type is rarely useful when interacting with foreign code, while using `_bytes/nul-terminated` is somewhat unnatural, since it forces Racket programmers to use byte strings. Using `_string/utf-8`, etc., meanwhile, may prematurely commit to a particular encoding of strings as bytes. The `_string` type supports conversion between Racket strings and `char*` strings using a parameter-determined conversion.

```
_string : ctype?
```

Expands to a use of the default-_string-type parameter. The parameter's value is consulted when _string is evaluated, so the parameter should be set before any interface definition that uses _string.

Don't use _string when you should use _path. Although C APIs typically represent paths as strings, and although the default _string (via default-_string-type) even implicitly converts Racket paths to strings, using _path ensures the proper encoding of strings as paths, which is not always UTF-8. See also _path for a caveat about relative paths.

```
(default-_string-type) → ctype?
(default-_string-type type) → void?
  type : ctype?
```

A parameter that determines the current meaning of _string. It is initially set to _string*/utf-8. If you change it, do so *before* interfaces are defined.

### 3.4.4 Other String Types

```
_file : ctype?
```

Like _path, but when values go from Racket to C, cleanse-path is used on the given value. As an output value, it is identical to _path.

```
_bytes/eof : ctype?
```

Similar to the _bytes type, except that a foreign return value of NULL is translated to a Racket eof value.

```
_string/eof : ctype?
```

Similar to the _string type, except that a foreign return value of NULL is translated to a Racket eof value.

## 3.5 Pointer Types

```
_pointer : ctype?
```

Corresponds to Racket *C pointer* values. These pointers can have an arbitrary Racket object attached as a type tag. The tag is ignored by built-in functionality; it is intended to be used by interfaces. See §5.3 "Tagged C Pointer Types" for creating pointer types that use these tags for safety. A #f value is converted to NULL and vice versa.

As a result type, the address referenced by a _pointer value must not refer to memory managed by the garbage collector (unless the address corresponds to a value that supports interior pointers and that is otherwise referenced to preserve the value from garbage collection). The reference is not traced or updated by the garbage collector. As an argument type, _pointer works for a reference to either GC-managed memory or not.

The equal? predicate equates C pointers (including pointers for _gcpointer and possibly containing an offset) when they refer to the same address—except for C pointers that are instances of structure types with the prop:cpointer property, in which case the equality rules of the relevant structure types apply.

```
_gcpointer : ctype?
```

The same as _pointer as an argument type, but as a result type, _gcpointer corresponds to a C pointer value that refers to memory managed by the garbage collector.

In the BC implementation of Racket, a _gcpointer result pointer can reference to memory that is not managed by the garbage collector, but beware of using an address that might eventually become managed by the garbage collector. For example, if a reference is created by malloc with 'raw and released by free, then the free may allow the memory formerly occupied by the reference to be used later by the garbage collector.

The cpointer-gcable? function returns #t for a cpointer generated via the _gcpointer result type. See cpointer-gcable? for more information.

```
_racket : ctype?
_scheme : ctype?
```

A type that can be used with any Racket object; it corresponds to the Scheme_Object* type of Racket's C API (see *Inside: Racket C API*). The _racket or _scheme type is useful only for libraries that are aware of Racket's C API.

As a result type with a function type, _racket or _scheme permits multiple values, but multiple values are not allowed in combination with a true value for #:in-original-place? or #:async-apply in _cprocedure or _fun.

```
_fpointer : ctype?
```

Similar to _pointer, except that when _fpointer is used as the type for get-ffi-obj or ffi-obj-ref, then a level of indirection is skipped. Furthermore, for a C pointer value from get-ffi-obj or ffi-obj-ref using _fpointer, ptr-ref on the pointer as a _fpointer simply returns the pointer instead of dereferencing it. Like _pointer, _fpointer treats #f as NULL and vice versa.

A type generated by _cprocedure or _fun builds on _fpointer, and normally _cprocedure or _fun should be used instead of _fpointer.

```
(_or-null ctype) → ctype?
  ctype : ctype?
```

Creates a type that is like *ctype*, but `#f` is converted to `NULL` and vice versa. The given *ctype* must have the same C representation as `_pointer`, `_gcpointer`, or `_fpointer`.

```
(_gcable ctype) → ctype?
  ctype : ctype?
```

Creates a type that is like *ctype*, but whose base representation is like `_gcpointer` instead of `_pointer`. The given *ctype* must have a base representation like `_pointer` or `_gcpointer` (and in the later case, the result is the *ctype*).

## 3.6 Function Types

```
(_cprocedure  input-types
              output-type
             [#:abi abi
              #:varargs-after varargs-after
              #:atomic? atomic?
              #:async-apply async-apply
              #:lock-name lock-name
              #:in-original-place? in-original-place?
              #:blocking? blocking?
              #:callback-exns? callback-exns?
              #:save-errno save-errno
              #:wrapper wrapper
              #:keep keep])                          → any
  input-types : (list ctype?)
  output-type : ctype?
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  varargs-after : (or/c #f positive-exact-integer?) = #f
  atomic? : any/c = #f
  async-apply : (or/c #f ((-> any/c) . -> . any/c) box?) = #f
  lock-name : (or/c string? #f) = #f
  in-original-place? : any/c = #f
  blocking? : any/c = #f
  callback-exns? : any/c = #f
  save-errno : (or/c #f 'posix 'windows) = #f
  wrapper : (or/c #f (procedure? . -> . procedure?)) = #f
  keep : (or/c boolean? box? (any/c . -> . any/c)) = #t
```

A type constructor that creates a new function type, which is specified by the given *input-types* list and *output-type*. Usually, the `_fun` syntax (described below) should be used

instead, since it manages a wide range of complicated cases and may enable static code generation.

The resulting type can be used to reference foreign functions (usually `ffi-obj`s, but any pointer object can be referenced with this type), generating a matching foreign *callout* object. Such objects are new primitive procedure objects that can be used like any other Racket procedure. As with other pointer types, `#f` is treated as a `NULL` function pointer and vice versa.

A type created with `_cprocedure` can also be used for passing Racket procedures to foreign functions, which will generate a foreign function pointer that calls to the given Racket *callback* procedure. There are no restrictions on the representation of the Racket procedure; in particular, the procedure can have free variables that refer to bindings in its environment. Callbacks are subject to run-time constraints, however, such as running in atomic mode or not raising exceptions; see more information on callbacks below.

The optional *abi* keyword argument determines the foreign ABI that is used. Supplying `#f` or `'default` indicates the platform-dependent default. The other possible values—`'stdcall` and `'sysv` (i.e., "cdecl")—are currently supported only for 32-bit Windows; using them on other platforms raises an exception. See also `ffi/winapi`.

The optional *varargs-after* argument indicates whether some function-type arguments should be considered "varargs," which are argument represented by an ellipsis `...` in the C declaration (but by explicit arguments in *input-types*). A `#f` value indicates that the C function type does not have varargs. If *varargs-after* is a number, then arguments after the first *varargs-after* arguments in *input-types* are varargs. Note that `#f` is different from `(length` *input-types*`)` on some platforms; the possibility of varargs for a function may imply a different calling convention even for non-vararg arguments. Note also that a non-`#f` *varargs-after* does *not* mean that you can supply any number of arguments to a callout or receive any number of arguments to a callback using the procedure type; to work with different argument counts and argument types, use `_cprocedure` (or `_fun`) separately for each combination.

For callouts to foreign functions with the generated type:

- If *save-errno* is `'posix`, then the value of `errno` is saved (specific to the current thread) immediately after a foreign function callout returns. The saved value is accessible through `saved-errno`. If *save-errno* is `'windows`, then the value of `GetLastError()` is saved for later use via `saved-errno`; the `'windows` option is available only on Windows (on other platforms `saved-errno` will return 0). If *save-errno* is `#f`, no error value is saved automatically.

  The error-recording support provided by *save-errno* is needed because the Racket runtime system may otherwise preempt the current Racket thread and itself call functions that set error values.

- If *wrapper* is not `#f`, it takes the callout that would otherwise be generated and returns

a replacement procedure. Thus, *wrapper* acts a hook to perform various argument manipulations before the true callout is invoked, and it can return different results (for example, grabbing a value stored in an "output" pointer and returning multiple values).

- If *lock-name* is not `#f`, then a process-wide lock with the given name is held during the foreign call. In a build that supports parallel places, *lock-name* is registered via `scheme_register_process_global`, so choose names that are suitably distinct.

- If *in-original-place?* is true, then when a foreign callout procedure with the generated type is called in a Racket place other than the original Racket place or in a Racket parallel thread, the procedure is called in the original Racket place in an unspecified coroutine thread. Use this mode for a foreign function that is not thread-safe at the C level, which means that it is not place-safe or parallel-thread-safe at the Racket level. Callbacks from place-unsafe code back into Racket at a non-original place typically will not work, since the place of the Racket code may have a different allocator than the original place.

- If *blocking?* is true, then a foreign callout deactivates tracking of the calling OS thread—to the degree supported by the Racket variant—during the foreign call. The value of *blocking?* affects only the CS implementation of Racket, where it enable activity such as garbage collection in other OS threads while the callout blocks. Since a garbage collection can happen during the foreign call, objects passed to the foreign call need to be immobile if they're managed by the garbage collector; in particular, any `_ptr` arguments should normally specify `'atomic-interior` allocation mode. If the blocking callout can invoke any callbacks back to Racket, those callbacks must be constructed with a non-`#f` value of *async-apply*, even if they are always applied in the OS thread used to run Racket.

- If *callback-exns?* is true, then a foreign callout allows an atomic callback during the foreign call to raise an exception that escapes from the foreign call. From the foreign library's perspective, the exception escapes via `longjmp`. Exception escapes are implemented through an exception handler that catches and reraises the exception.

  A callback that raises an exception must be an atomic callback in the BC implementation of Racket (and callbacks are always atomic in the CS implementation). Raising an exception is not allowed in a callback that has an *async-apply*, since the callback will run in an unspecified context. Raising an exception is also not allowed if the callout (that led to the callback) was created with *in-original-place?* as true and called in a non-original place.

- Values that are provided to a callout (i.e., the underlying callout, and not the replacement produced by a *wrapper*, if any) are always considered reachable by the garbage collector until the called foreign function returns. If the foreign function invokes Racket callbacks, however, beware that values managed by the Racket garbage collector might be moved in memory by the garbage collector. Also, beware that each argument is retained only as supplied, and not as potentially converted to a different representation based the argument's type (via layers of *racket-to-c* procedures for

`make-ctype`); a converter procedure associated with a type may need to create a reference connection between the original and converted values using an ephemeron hash table.

- A callout object is finalized internally. Beware of trying to use a callout object that is reachable only from a finalized object, since the two objects might be finalized in either order.

For callbacks to Racket functions with the generated type:

- The *keep* argument provides control over reachability by the garbage collector of the underlying value that foreign code see as a plain C function. Additional care must be taken in case the foreign code might retain the callback function, in which case the callback value must remain reachable or else the held callback will become invalid. The possible values of *keep* are as follows:
  - `#t` — the callback stays in memory as long as the converted Racket function is reachable. This mode is the default, as it is fine in most cases. Note that each Racket function can hold onto only one callback value through this mode, so it is not suitable for a function used multiple times as a reatined callback.
  - `#f` — the callback value is not held. This mode may be useful for a callback that is only used for the duration of the foreign call; for example, the comparison function argument to the standard C library `qsort` function is only used while `qsort` is working, and no additional references to the comparison function are kept. Use this option only in such cases, when no holding is necessary and you want to avoid the extra cost.
  - A box holding `#f` or any other non-list value — the callback value is stored in the box, overriding any non-list value that was in the box (making it useful for holding a single callback value). When you know that the callback is no longer needed, you can "release" the callback value by changing the box contents or by allowing the box itself to become unreachable. This mode can be useful if the box is held for a dynamic extent that corresponds to when the callback is needed; for example, you might encapsulate some foreign functionality in a Racket class or a unit, and keep the callback box as a field in new instances or instantiations of the unit.
  - A box holding `null` (or any list) — similar to a box holding a non-list value, except that new callback values are `cons`ed onto the contents of the box. This mode is therefore useful in cases when a Racket function is used in multiple callbacks (that is, sent to foreign code to hold onto multiple times) and all callbacks should be retained together.
  - A one-argument function — the function is invoked with the callback value when it is generated. This mode allows you to explicitly manage reachability of the generated callback closure.

- If *wrapper* is not #f, it takes the procedure to be converted into a callback and returns a replacement procedure to be invoked as the callback. Thus, *wrapper* acts a hook to perform various argument manipulations before a Racket callback function is called, and it can return different results to the foreign caller.

  The callback value's reachability (and its interaction with *keep*) is based on the original function for the callback, not the result of *wrapper*.

- If *atomic?* is true or when using the CS implementation of Racket, then when a Racket procedure is given this type and called as a callback from foreign code, then the Racket process is put into atomic mode while evaluating the Racket procedure body.

  In atomic mode, other Racket threads do not run, so the Racket code must not call any function that potentially blocks on synchronization with other threads, or else it may lead to deadlock. In addition, the Racket code must not perform any potentially blocking operation (such as I/O), it must not raise an uncaught exception unless called through a callout that supports exception (with #:callback-exns? #t), it must not perform any escaping continuation jumps, and (at least for the BC implementation) its non-tail recursion must be minimal to avoid C-level stack overflow; otherwise, the process may crash or misbehave.

  Callbacks are always atomic in the CS implementation of Racket. Even on the BC implementation of Racket, atomic mode is typically needed for callbacks, because capturing by copying a portion of the C stack is often incompatible with C libraries.

  If a callback in atomic mode sends a break to the current thread, then not only is the break delayed as usual for atomic mode, it delivery might be delayed further than return from a foreign call that led to the callback.

- If a *async-apply* is provided as a procedure or box, then a Racket callback procedure with the generated procedure type can be applied in a foreign thread (i.e., an OS-level thread other than the one used to run Racket).

  If *async-apply* is a procedure, the call in the foreign thread is transferred to the OS-level thread that runs Racket coroutine threads and to an unspecified coroutine thread; the job of the provided *async-apply* procedure is to arrange for the callback procedure to be run in a suitable Racket thread.

  The given *async-apply* procedure is applied to a thunk that encapsulates the specific callback invocation, and the foreign OS-level thread blocks until the thunk is called and completes; the thunk must be called exactly once, and the callback invocation must return normally. The given *async-apply* procedure itself is called in atomic mode.

  If the callback is known to complete quickly, requires no synchronization, and works independent of the Racket thread in which it runs, then it is safe for the given *async-apply* procedure to apply the thunk directly. Otherwise, the given *async-apply* procedure must arrange for the thunk to be applied in a suitable Racket thread sometime after the given *async-apply* procedure itself returns; if the thunk raises an exception or synchronizes within an unsuitable Racket-level thread, it can deadlock or otherwise damage the Racket process.

If *async-apply* is a box, then the value contained in the box is used as the result of the callback when it is called in a foreign thread; the *async-apply* value is converted to a foreign value at the time that _cprocedure is called. Using a boxed constant value for *async-apply* avoids the need to synchronize with the OS-level thread that runs Racket, but it effectively ignores the Racket procedure that is wrapped as callback when the callback is applied in a foreign thread.

Foreign-thread detection to trigger *async-apply* works only when Racket is compiled with OS-level thread support, which is the default for many platforms. If a callback with an *async-apply* is called from foreign code in the same OS-level thread that runs Racket, then *async-apply* is not used.

- A callback normally should not escape by raising an exception or invoking a continuation. An atomic callback can potentially raise an exception, but only if it is called during the invocation of a callout created with *callback-exns?* as true. A non-atomic callback must never raise an exception.

Changed in version 6.3 of package base: Added the #:lock-name argument.
Changed in version 6.12.0.2: Added the #:blocking? argument.
Changed in version 7.9.0.16: Added the #:varargs-after argument.
Changed in version 8.0.0.8: Added the #:callback-exns? argument.

```
(_fun fun-option ... maybe-args type-spec ... -> type-spec
      maybe-wrapper)

   fun-option = #:abi abi-expr
              | #:varargs-after varargs-after-expr
              | #:save-errno save-errno-expr
              | #:keep keep-expr
              | #:atomic? atomic?-expr
              | #:async-apply async-apply-expr
              | #:lock-name lock-name-expr
              | #:in-original-place? in-original-place?-expr
              | #:blocking? blocking?-expr
              | #:callback-exns? callback-exns?-expr
              | #:retry (retry-id [arg-id init-expr])

   maybe-args =
              | formals ::

    type-spec = type-expr
              | (id : type-expr)
              | (type-expr = value-expr)
              | (id : type-expr = value-expr)

maybe-wrapper =
              | -> output-expr
```

Creates a new function type. The `_fun` form is a convenient syntax for the `_cprocedure` type constructor, and it can enable more static generation of callout and callback code; see `_fun` from `ffi/unsafe/static` for more information.

In the simplest form of `_fun`, only the input *type-expr*s and the output *type-expr* are specified, and each types is a simple expression, which creates a straightforward function type. For example,

```
(_fun _string _int -> _int)
```

specifies a function that receives a string and an integer and returns an integer.

See `_cprocedure` for information about the `#:abi`, `#:varargs-after`, `#:save-errno`, `#:keep`, `#:atomic?`, `#:async-apply`, `#:in-original-place?`, `#:blocking`, and `#:callback-exns?` options.

In its full form, the `_fun` syntax provides an IDL-like language that creates a wrapper function around the primitive foreign function when the type is used for a callout. These wrappers can implement complex interfaces given simple specifications:

- The full form of each argument *type-spec* can include an optional label and an expression. A label *id* `:` makes the argument value accessible to later expressions using *id*. A `=` *value-expr* expression causes the wrapper function to calculates the argument for that position using *value-expr*, implying that the wrapper does not expect to be given an argument for that position.

  For example,

  ```
  (_fun (s : _string) (_int = (string-length s)) -> _int)
  ```

  produces a wrapper that takes a single string argument and calls a foreign function that takes a string and an integer; the string's length is provided as the integer argument.

- If the optional *output-expr* is specified, or if an expression is provided for the output type, then the expression specifies an expression that will be used as a return value for the function call, replacing the foreign function's result. The *output-expr* can use any of the previous labels, including a label given for the output to access the foreign function's return value.

  For example,

  ```
  (_fun _string (len : _int) -> (r : _int) -> (min r len))
  ```

  produces a wrapper that returns the minimum of the foreign function's result and the given integer argument.

- A `#:retry (retry-id [arg-id init-expr] ...)` specification binds *retry-id* for use in an *output-expr* for retrying the foreign call (normally in tail position). The function bound to *retry-id* accepts each *arg-id* as an argument, each *arg-id* can be used in `= value-expr`s, and each *init-expr*s provides the initial value for the corresponding *arg-id*.

  For example,

  ```
  (_fun #:retry (again [count 0])
        _string _int -> (r : _int)
        -> (if (and (= r ERR_BUSY)
                    (< count 5))
               (again (add1 count))
               r))
  ```

  produces a wrapper that calls the foreign function up to five times if it continues to produce a number equal to `ERR_BUSY`.

- In rare cases where complete control over the input arguments is needed, the wrapper's argument list can be specified as *maybe-args* with a *formals* as for `lambda` (including keyword arguments and/or a "rest" argument). When an argument *type-spec* includes a label that matches an binding identifier in *formals*, then the identifier is used as the default value for the argument. All argument *type-spec*s must include either explicit `= value-expr` annotations or an implicit one through a matching label.

  For example,

  ```
  (_fun (n s) :: (s : _string) (n : _int) -> _int)
  ```

  produces a wrapper that receives an integer and a string, but the foreign function receives the string first.

Changed in version 6.2 of package `base`: Added the `#:retry` option.
Changed in version 6.3: Added the `#:lock-name` option.
Changed in version 6.12.0.2: Added the `#:blocking?` option.
Changed in version 7.9.0.16: Added the `#:varargs-after` option.
Changed in version 8.0.0.8: Added the `#:callback-exns?` option.

```
(function-ptr ptr-or-proc fun-type) → cpointer?
  ptr-or-proc : (or cpointer? procedure?)
  fun-type : ctype?
```

Casts *ptr-or-proc* to a function pointer of type *fun-type*.

```
->
```

A literal used in `_fun` forms. (It's unfortunate that this literal has the same name as `->` from `racket/contract`, but it's a different binding.)

### 3.6.1 Custom Function Types

The behavior of the `_fun` type can be customized via *custom function types*, which are pieces of syntax that can behave as C types and C type constructors, but they can interact with function calls in several ways that are not possible otherwise. When the `_fun` form is expanded, it tries to expand each of the given type expressions, and ones that expand to certain keyword-value lists interact with the generation of the foreign function wrapper. This expansion makes it possible to construct a single wrapper function, avoiding the costs involved in compositions of higher-order functions.

Custom function types are macros that expand to a sequence `(key: val ...)`, where each `key:` is from a short list of known keys. Each key interacts with generated wrapper functions in a different way, which affects how its corresponding argument is treated:

- `type:` specifies the foreign type that should be used, if it is `#f` then this argument does not participate in the foreign call.
- `expr:` specifies an expression to be used for arguments of this type, removing it from wrapper arguments.
- `bind:` specifies a name that is bound to the original argument if it is required later (e.g., `_box` converts its associated value to a C pointer, and later needs to refer back to the original box).
- `1st-arg:` specifies a name that can be used to refer to the first argument of the foreign call (good for common cases where the first argument has a special meaning, e.g., for method calls).
- `prev-arg:` similar to `1st-arg:`, but refers to the previous argument.
- `pre:` a pre-foreign code chunk that is used to change the argument's value.
- `post:` a similar post-foreign code chunk.
- `keywords:` specifies keyword/value expressions that will be used with the surrounding `_fun` form. (Note: the keyword/value sequence follows `keywords:`, not parenthesized.)

The `pre:` and `post:` bindings can be of the form `(id => expr)` to use the existing value. Note that if the `pre:` expression is not `(id => expr)`, then it means that there is no input for this argument to the `_fun`-generated procedure. Also note that if a custom type is used as an output type of a function, then only the `post:` code is used.

Most custom types are meaningful only in a `_fun` context, and will raise a syntax error if used elsewhere. A few such types can be used in non-`_fun` contexts: types which use only `type:`, `pre:`, `post:`, and no others. Such custom types can be used outside a `_fun` by expanding them into a usage of `make-ctype`, using other keywords makes this impossible, because it means that the type has specific interaction with a function call.

```
(define-fun-syntax id transformer-expr)
```

Binds *id* as a custom function type as well as a syntax transformer (i.e, macro). The type is expanded by applying the procedure produced by *transformer-expr* to a use of the custom function type.

For instance, the following defines a new type that automatically coerces the input number to an inexact form which is compatible with the `_float` type.

```
(define-fun-syntax _float*
  (syntax-id-rules (_float*)
    [_float* (type: _float pre: (x => (+ 0.0 x)))]))

(_fun _float* -> _bool)
```

```
_?
```

A custom function type that is a marker for expressions that should not be sent to the foreign function. Use this to bind local values in a computation that is part of an ffi wrapper interface, or to specify wrapper arguments that are not sent to the foreign function (e.g., an argument that is used for processing the foreign output).

Examples:

```
(_fun _? ; not sent to foreign function
      _int -> _int)
(_fun [init : _?] ; init is used for pre-processing
      [boxed : (_box _int) = (box init)]
      -> _void)
(_fun [offset : _?] ; offset is used for post-processing
      -> [res : _int]
      -> (+ res offset))
```

```
(_ptr mode type-expr maybe-malloc-mode)
```

```
             mode = i
                  | o
                  | io

maybe-malloc-mode =
                  | #f
                  | raw
                  | atomic
                  | nonatomic
                  | tagged
                  | atomic-interior
                  | interior
                  | zeroed-atomic
                  | zeroed-atomic-interior
                  | stubborn
                  | uncollectable
                  | eternal
```

Creates a C pointer type, where *mode* indicates input or output pointers (or both). The *mode* can be one of the following (matched as a symbol independent of binding):

- `i` — indicates an *input* pointer argument: the wrapper arranges for the function call to receive a value that can be used with the `type` and to send a pointer to this value to the foreign function. After the call, the value is discarded.
- `o` — indicates an *output* pointer argument: the foreign function expects a pointer to a place where it will save some value, and this value is accessible after the call, to be used by an extra return expression. If `_ptr` is used in this mode, then the generated wrapper does not expect an argument, since one will be freshly allocated before the call.
- `io` — combines the above into an *input/output* pointer argument: the wrapper gets the Racket value, allocates and set a pointer using this value, and then references the value after the call. The "`_ptr`" name can be confusing here: it means that the foreign function expects a pointer, but the generated wrapper uses an actual value. (Note that if this is used with structs, a struct is created when calling the function, and a copy of the return value is made too—which is inefficient, but ensures that structs are not modified by C code.)

For example, the `_ptr` type can be used in output mode to create a foreign function wrapper that returns more than a single argument. The following type:

```
(_fun (i : (_ptr o _int))
      -> (d : _double)
      -> (values d i))
```

creates a function that calls the foreign function with a fresh integer pointer, and use the value that is placed there as a second return value.

The pointer argument created by `_ptr` is allocated using allocated using `(malloc type-expr)` if *maybe-malloc-mode* is not specified or if it is `#f`, `(malloc type-expr 'maybe-malloc-mode)` otherwise.

Changed in version 7.7.0.6 of package `base`: The modes `i`, `o`, and `io` match as symbols instead of free identifiers.
Changed in version 8.0.0.13: Added *maybe-malloc-mode*.
Changed in version 8.14.0.4: Added the `zeroed-atomic` and `zeroed-atomic-interior` allocation modes.

```
(_box type maybe-malloc-mode)
```

A custom function type similar to a `(_ptr io type)` argument, where the input is expected to be a box holding an appropriate value, which is unboxed on entry and modified accordingly on exit. The optional *maybe-malloc-mode* is the same as for `_ptr`.

Example:

```
(_fun (_box _int) -> _void)
(_fun [boxed : (_box _int) = (box 0)]
      -> [res : _int]
      -> (values res (unbox boxed)))
```

```
(_list mode type maybe-len maybe-mode)

     mode = i
          | o
          | io

 maybe-len =
          | len-expr

maybe-mode =
          | atomic
          | raw
          | atomic
          | nonatomic
          | tagged
          | atomic-interior
          | interior
          | zeroed-atomic
          | zeroed-atomic-interior
          | stubborn
          | uncollectable
          | eternal
```

A custom function type that is similar to `_ptr`, except that it is used for converting lists to/from C vectors. The optional *maybe-len* argument is needed for output values where it is used in the post code, and in the pre code of an output mode to allocate the block. (If the length is 0, then NULL is passed in and an empty list is returned.) In either case, it can refer to a previous binding for the length of the list which the C function will most likely require. The *maybe-mode*, if provided, is quoted and passed to `malloc` as needed to allocate the C representation.

For example, the following type corresponds to a function that takes a vector argument of type `*float` (from a Racket list input) and a length argument of type `int` for the vector:

```
(_fun [vec : (_list i _float)]
      ; this argument is implicitly provided
      [_int = (length vec)]
      -> _void)
```

In this next example, the type specifies a function that provides output through a given output vector (represented as a list on the Racket side) and through a boolean return value. The FFI-bound function will take an integer argument and return two values, the vector and the boolean.

```
(_fun [len : _int]
      [vec : (_list o _float len)]
      -> [res : _bool]
      -> (values vec res))
```

Changed in version 7.7.0.2 of package `base`: Added *maybe-mode*.
Changed in version 7.7.0.6: The modes *i*, *o*, and *io* match as symbols instead of free identifiers.
Changed in version 8.14.0.4: Added the `zeroed-atomic` `zeroed-atomic-interior` allocation modes.

```
(_vector mode type maybe-len maybe-mode)
```

A custom function type like `_list`, except that it uses Racket vectors instead of lists.

Examples:

```
(_fun [vec : (_vector i _float)]
      [_int = (length vec)]
      -> _void)
(_fun [len : _int]
      [vec : (_vector o _float len)]
      -> [res : _bool]
      -> (values vec res))
```

See `_list` for more explanation about the examples.

Changed in version 7.7.0.2 of package base: Added *maybe-mode*.
Changed in version 7.7.0.6: The modes i, o, and io match as symbols instead of free identifiers.

```
_bytes
(_bytes o len-expr)
```

The _bytes form by itself corresponds to C's char* type; a byte string is passed as _bytes without any copying. Beware that a Racket byte string is not necessarily nul terminated; see also _bytes/nul-terminated.

In the BC implementation of Racket, a C non-NULL result value is converted to a Racket byte string without copying; the pointer is treated as potentially managed by the garbage collector (see _gcpointer for caveats). In the CS implementation of Racket, conversion requires copying to represent a C char* result as a Racket byte string, and the original pointer is *not* treated as managed by the garbage collector. In both cases, the C result must have a nul terminator to determine the Racket byte string's length.

A (_bytes o *len-expr*) form is a custom function type. As an argument, a byte string is allocated with the given length; in the BC implementation, that byte string includes an extra byte for the nul terminator, and (_bytes o *len-expr*) as a result wraps a C non-NULL char* pointer as a byte string of the given length. For the CS implementation, the allocated argument does not include a nul terminator and a copy is made for a result string.

As usual, _bytes treats #f as NULL and vice versa. As a result type, (_bytes o *len-expr*) works only for non-NULL results.

```
_bytes/nul-terminated
(_bytes/nul-terminated o len-expr)
```

The _bytes/nul-terminated type is like _bytes, but an explicit nul-terminator byte is added to a byte-string argument, which implies copying. As a result type, a char* is copied to a fresh byte string (without an explicit nul terminator).

When (_bytes/nul-terminated o *len-expr*) is used as an argument type, a byte string of length *len-expr* is allocated. Similarly, when (_bytes/nul-terminated o *len-expr*) is used as a result type, a char* result is copied to a fresh byte string of length *len-expr*.

As usual, _bytes/nul-terminated treats #f as NULL and vice versa. As a result type, (_bytes/nul-terminated o *len-expr*) works only for non-NULL results.

Added in version 6.12.0.2 of package base.

## 3.7 C Struct Types

```
(make-cstruct-type  types
                   [abi
                    alignment
                    malloc-mode]) → ctype?
  types : (non-empty-listof ctype?)
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  alignment : (or/c #f 1 2 4 8 16) = #f
  malloc-mode : (or/c 'raw 'atomic 'nonatomic 'tagged
                      'atomic-interior 'interior
                      'zeroed-atomic 'zeroed-atomic-interior
                      'stubborn 'uncollectable 'eternal)
              = 'atomic
```

The primitive type constructor for creating new C struct types. These types are actually new primitive types; they have no conversion functions associated. The corresponding Racket objects that are used for structs are pointers, but when these types are used, the value that the pointer *refers to* is used, rather than the pointer itself. This value is basically made of a number of bytes that is known according to the given list of *types* list.

If *alignment* is `#f`, then the natural alignment of each type in *types* is used for its alignment within the struct type. Otherwise, *alignment* is used for all struct type members.

The *malloc-mode* argument is used when an instance of the type is allocated to represent the result of a function call. This allocation mode is *not* used for an argument to a callback, because temporary space allocated on the C stack (possibly by the calling convention) is used in that case.

Changed in version 7.3.0.8 of package `base`: Added the *malloc-mode* argument.
Changed in version 8.14.0.4: Added the `'zeroed-atomic` `'zeroed-atomic-interior` allocation modes.

```
(_list-struct [#:alignment alignment
               #:malloc-mode malloc-mode]
               type ...+)                → ctype?
  alignment : (or/c #f 1 2 4 8 16) = #f
  malloc-mode : (or/c 'raw 'atomic 'nonatomic 'tagged
                      'atomic-interior 'interior
                      'zeroed-atomic 'zeroed-atomic-interior
                      'stubborn 'uncollectable 'eternal)
              = 'atomic
  type : ctype?
```

A type constructor that builds a struct type using `make-cstruct-type` function and wraps it in a type that marshals a struct as a list of its components. Note that space for structs must be allocated using `malloc` with *malloc-mode*; the converter for a `_list-struct` type immediately allocates and uses a list from the allocated space, so it is inefficient. Use `define-cstruct` below for a more efficient approach.

Changed in version 6.0.0.6 of package `base`: Added `#:malloc-mode`.
Changed in version 8.14.0.4: Added the `'zeroed-atomic` `'zeroed-atomic-interior` allocation modes.

```
(define-cstruct id/sup ([field-id type-expr field-option ...] ...)
  property ...)

      id/sup = _id
             | (_id _super-id)

field-option = #:offset offset-expr

    property = #:alignment alignment-expr
             | #:malloc-mode malloc-mode-expr
             | #:property prop-expr val-expr
             | #:no-equal
             | #:define-unsafe

  offset-expr : exact-integer?
  alignment-expr : (or/c #f 1 2 4 8 16)
  malloc-mode-expr :  (or/c 'raw 'atomic 'nonatomic 'tagged
                            'atomic-interior 'interior
                            'zeroed-atomic 'zeroed-atomic-interior
                            'stubborn 'uncollectable 'eternal)
  prop-expr : struct-type-property?
```

Defines a new C struct type, but unlike `_list-struct`, the resulting type deals with C structs in binary form, rather than marshaling them to Racket values. The syntax is similar to `define-struct`, providing accessor functions for raw struct values (which are pointer objects); the *`_id`* must start with `_`, at most one `#:offset` can be supplied for a field, and at most one `#:alignment` or `#:malloc-mode` can be supplied. If no *`_super-id`* is provided, then at least one field must be specified.

The resulting bindings are as follows:

- *`_id`* : the new C type for this struct.
- *`_id`*`-pointer`: a pointer type that should be used when a pointer to values of this struct are used.
- *`_id`*`-pointer/null`: like *`_id`*`-pointer`, but allowing NULL pointers (as represented on the Racket side by `#f`).
- *`id`*`?`: a predicate for the new type.
- *`id`*`-tag`: the tag object that is used with instances. The tag object may be the symbol form of *`id`* or a list of symbols containing the *`id`* symbol and other symbols, such as the *`super-id`* symbol.

- `make-id` : a constructor, which expects an argument for each field.
- `id-field-id` : an accessor function for each `field-id`; if the field has a C struct type, then the result of the accessor is a pointer to the field within the enclosing structure, rather than a copy of the field.
- `set-id-field-id!` : a mutator function for each `field-id`.
- `id-field-id-offset` : the absolute offset, in bytes, of each `field-id`, if `#:define-unsafe` is present.
- `unsafe-id-field-id` : an unsafe accessor function for each `field-id`, if `#:define-unsafe` is present.
- `unsafe-set-id-field-id!` : an unsafe mutator function for each `field-id`, if `#:define-unsafe` is present.
- `id`: structure-type information compatible with `struct-out` or `match` (but not `struct` or `define-struct`); currently, this information is correct only when no `super-id` is specified.
- `id->list`, `list->id` : a function that converts a struct into a list of field values and vice versa.
- `id->list*`, `list*->id` : like `id->list`, `list->id`, but fields that are structs are recursively unpacked to lists or packed from lists.
- `struct:cpointer:id`: only when a `#:property` is specified — a structure type that corresponds to a wrapper to reflect properties (see below).
- `make-wrap-id`: only when a `#:property` is specified — a function that takes a cpointer and returns a wrapper structure that holds the cpointer.

Objects of the new type are actually C pointers, with a type tag that is the symbol form of `id` or a list that contains the symbol form of `id`. Since structs are implemented as pointers, they can be used for a `_pointer` input to a foreign function: their address will be used. To make this a little safer, the corresponding cpointer type is defined as `_id-pointer`. The `_id` type should not be used when a pointer is expected, since it will cause the struct to be copied rather than use the pointer value, leading to memory corruption.

Field offsets within the structure are normally computed automatically, but the offset for a field can be specified with `#:offset`. Specifying `#:offset` for a field affects the default offsets computed for all remaining fields.

Instances of the new type are not normally Racket structure instances. However, if at least one `#:property` modifier is specified, then struct creation and coercions from `_id` variants wrap a non-NULL C pointer representation in a Racket structure that has the specified properties. The wrapper Racket structure also has a `prop:cpointer` property, so that wrapped C pointers can be treated the same as unwrapped C pointers. If a `super-id` is provided

and it corresponds to a C struct type with a wrapper structure type, then the wrapper structure type is a subtype of `super-id`'s wrapper structure type. If a `#:property` modifier is specified, `#:no-equal` is not specified, and if `prop:equal+hash` is not specified as any `#:property`, then the `prop:equal+hash` property is automatically implemented for the wrapper structure type to use `ptr-equal?`.

If the first field is itself a C struct type, its tag will be used in addition to the new tag. This feature supports common cases of object inheritance, where a sub-struct is made by having a first field that is its super-struct. Instances of the sub-struct can be considered as instances of the super-struct, since they share the same initial layout. Using the tag of an initial C struct field means that the same behavior is implemented in Racket; for example, accessors and mutators of the super-struct can be used with the new sub-struct. See the example below.

Providing a `super-id` is shorthand for using an initial field named `super-id` and using `_super-id` as its type. Thus, the new struct will use `_super-id`'s tag in addition to its own tag, meaning that instances of *`_id`* can be used as instances of `_super-id`. Aside from the syntactic sugar, the constructor function is different when this syntax is used: instead of expecting a first argument that is an instance of `_super-id`, the constructor will expect arguments for each of `_super-id`'s fields, in addition for the new fields. This adjustment of the constructor is, again, in analogy to using a supertype with `define-struct`.

Structs are allocated using `malloc` with the result of *`malloc-mode-expr`*, which defaults to `'atomic`. (This allocation mode does not apply to arguments of a callback; see also `define-cstruct-type`.) The default allocation of `'atomic` means that the garbage collector ignores the content of a struct; thus, struct fields can hold only non-pointer values, pointers to memory outside the GC's control, and otherwise-reachable pointers to immobile GC-managed values (such as those allocated with `malloc` and `'internal` or `'internal-atomic`).

As an example, consider the following C code:

```
typedef struct { int x; char y; } A;
typedef struct { A a; int z; } B;

A* makeA() {
  A *p = malloc(sizeof(A));
  p->x = 1;
  p->y = 2;
  return p;
}

B* makeB() {
  B *p = malloc(sizeof(B));
  p->a.x = 1;
  p->a.y = 2;
  p->z   = 3;
```

```
  return p;
}

char gety(A* a) {
  return a->y;
}
```

Using the simple `_list-struct`, you might expect this code to work:

```
(define makeB
  (get-ffi-obj 'makeB "foo.so"
    (_fun -> (_list-struct (_list-struct _int _byte) _int))))
(makeB) ; should return '((1 2) 3)
```

The problem here is that `makeB` returns a pointer to the struct rather than the struct itself. The following works as expected:

```
(define makeB
  (get-ffi-obj 'makeB "foo.so" (_fun -> _pointer)))
(ptr-ref (makeB) (_list-struct (_list-struct _int _byte) _int))
```

As described above, `_list-struct`s should be used in cases where efficiency is not an issue. We continue using `define-cstruct`, first define a type for `A` which makes it possible to use `makeA`:

```
(define-cstruct _A ([x _int] [y _byte]))
(define makeA
  (get-ffi-obj 'makeA "foo.so"
    (_fun -> _A-pointer))) ; using _A is a memory-corrupting bug!
(define a (makeA))
(list a (A-x a) (A-y a))
; produces an A containing 1 and 2
```

Using `gety` is also simple:

```
(define gety
  (get-ffi-obj 'gety "foo.so"
    (_fun _A-pointer -> _byte)))
(gety a) ; produces 2
```

We now define another C struct for `B`, and expose `makeB` using it:

```
(define-cstruct _B ([a _A] [z _int]))
(define makeB
  (get-ffi-obj 'makeB "foo.so"
    (_fun -> _B-pointer)))
(define b (makeB))
```

We can access all values of `b` using a naive approach:

```
(list (A-x (B-a b)) (A-y (B-a b)) (B-z b))
```

but this is inefficient as it allocates and copies an instance of `A` on every access. Inspecting the tags `(cpointer-tag b)` we can see that `A`'s tag is included, so we can simply use its accessors and mutators, as well as any function that is defined to take an `A` pointer:

```
(list (A-x b) (A-y b) (B-z b))
(gety b)
```

Constructing a `B` instance in Racket requires allocating a temporary `A` struct:

```
(define b (make-B (make-A 1 2) 3))
```

To make this more efficient, we switch to the alternative `define-cstruct` syntax, which creates a constructor that expects arguments for both the super fields and the new ones:

```
(define-cstruct (_B _A) ([z _int]))
(define b (make-B 1 2 3))
```

Changed in version 6.0.0.6 of package `base`: Added `#:malloc-mode`.
Changed in version 6.1.1.8: Added `#:offset` for fields.
Changed in version 6.3.0.13: Added `#:define-unsafe`.
Changed in version 8.14.0.4: Added the `'zeroed-atomic` `'zeroed-atomic-interior` allocation modes.

```
(compute-offsets types [alignment declare])
 → (listof exact-integer?)
  types : (listof ctype?)
  alignment : (or/c #f 1 2 4 8 16) = #f
  declare : (listof (or/c #f exact-integer?)) = '()
```

Given a list of types in a C struct type, return the offset of those types.

The *types* list describes a C struct type and is identical to the list in `make-cstruct-type`.

The C struct's alignment is set with *alignment* The behavior is identical to `make-cstruct-type`.

Explicit positions can be set with *declare*. If provided, it is a list with the same length as as *types*. At each index, if a number is provided, that type is at that offset. Otherwise, the type is *alignment* bytes after the offset.

Examples:

```
> (compute-offsets (list _int _bool _short))
'(0 4 8)
> (compute-offsets (list _int _bool _short) 1)
'(0 4 8)
> (compute-offsets (list _int _int _int) #f (list #f 5 #f))
'(0 5 12)
```

Added in version 6.10.1.2 of package `base`.

## 3.8 C Array Types

```
(make-array-type type count) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

The primitive type constructor for creating new C array types. Like C struct types, array types are new primitive types with no conversion functions associated. When used as a function argument or return type, array types behave like pointer types; otherwise, array types behave like struct types (i.e., a struct with as many fields as the array has elements), particularly when used for a field within a struct type.

Since an array is treated like a struct, `cast`ing a pointer type to an array type does not work. Instead, use `ptr-ref` with a pointer, an array type constructed with `_array`, and index `0` to convert a pointer to a Racket representation that works with `array-ref` and `array-set!`.

```
(_array type count ...+) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

Creates an array type whose Racket representation is an array that works with `array-ref` and `array-set!`. The array is not copied; the Racket representation is backed by the underlying C representation.

Supply multiple *count*s for a multidimensional array. Since C uses row-major order for arrays, `(_array t n m)` is equivalent to `(_array (_array t m) n)`, which is different from an array of pointers to arrays.

When a value is used as an instance of an array type (e.g., as passed to a foreign function), checking ensures that the given value is an array of at least the expected length and whose elements have the same representation according to `ctype->layout`; the array can have additional elements, and it can have a different element type as long as that type matches the layout of the expected type.

```
(array? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a Racket representation of a C value via `_array`, `#f` otherwise.

```
(array-ref a i ...+) → any/c
  a : array?
  i : exact-nonnegative-integer?
```

Extracts an element from an array. Use multiple `i` indices for a multidimensional array access; using fewer indices than the array dimension produces a sub-array.

```
(array-set! a i ...+ v) → void?
  a : array?
  i : exact-nonnegative-integer?
  v : any/c
```

Sets an element in an array. Use multiple `i` indices for a multidimensional array update; using fewer indices than the array dimension sets a sub-array (i.e., `v` must be an array of the same size as the sub-array and `v` is copied into the sub-array).

```
(array-ptr a) → cpointer?
  a : array?
```

Extracts the pointer for an array's storage.

```
(array-length a) → exact-nonnegative-integer?
  a : array?
```

Extracts the length of an array. For a multidimensional array, the result is still a single number; extract an element to get a sub-array to get the length of the next dimension, and so on.

```
(array-type a) → ctype?
  a : array?
```

Extracts the type of the array. For a multidimensional array, the result is the ctype of the nested array.

```
(in-array a [start stop step]) → sequence?
  a : array?
  start : exact-nonnegative-integer? = 0
  stop : (or/c exact-integer? #f) = #f
  step : (and/c exact-integer? (not/c zero?)) = 1
```

Returns a sequence equivalent to `a` when no optional arguments are supplied.

The optional arguments *start*, *stop*, and *step* are as in `in-vector`.

```
(_array/list type count ...+) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

Like _array, but the Racket representation is a list (or list of lists for a multidimensional array) of elements copied to and from an underlying C array.

```
(_array/vector type count ...+) → ctype?
  type : ctype?
  count : exact-nonnegative-integer?
```

Like _array, but the Racket representation is a vector (or vector of vectors for a multidimensional array) of elements copied to and from an underlying C array.

## 3.9 C Union Types

```
(make-union-type type ...+) → ctype?
  type : ctype?
```

The primitive type constructor for creating new C union types. Like C struct types, union types are new primitive types with no conversion functions associated. Unions are always treated like structs with 'atomic allocation mode.

Example:

```
> (make-union-type (_list-struct _int _int)
                   (_list-struct _double _double))
#<compound-ctype>
```

```
(_union type ...+) → ctype?
  type : ctype?
```

Creates a union type whose Racket representation is a union that works with union-ref and union-set!. The union is not copied; the Racket representation is backed by the underlying C representation.

Example:

```
> (_union (_list-struct _int _int)
          (_list-struct _double _double))
#<compound-ctype>
```

```
(union? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a Racket representation of a C value via `_union`, `#f` otherwise.

Examples:

```
> (define a-union-type
    (_union (_list-struct _int _int)
            (_list-struct _double _double)))
> (define a-union-val
    (cast (list 3.14 2.71)
          (_list-struct _double _double)
          a-union-type))
> (union? a-union-val)
#t
> (union? 3)
#f
```

```
(union-ref u i) → any/c
  u : union?
  i : exact-nonnegative-integer?
```

Extracts a variant from a union. The variants are indexed starting at `0`.

Examples:

```
; see examples for union? for definitions
> (union-ref a-union-val 1)
'(3.14 2.71)
```

```
(union-set! u i v) → void?
  u : union?
  i : exact-nonnegative-integer?
  v : any/c
```

Sets a variant in a union.

Examples:

```
; see examples for union? for definitions
> (union-set! a-union-val 0 (list 4 5))
> a-union-val
#<union>
> (union-ref a-union-val 0)
'(4 5)
```

```
(union-ptr u) → cpointer?
  u : union?
```

Extracts the pointer for a union's storage.

Example:

```
> (union-ptr a-union-val)
#<cpointer+offset>
```

## 3.10 Enumerations and Masks

Although the constructors below are described as procedures, they are implemented as syntax, so that error messages can report a type name where the syntactic context implies one.

```
(_enum symbols [basetype #:unknown unknown]) → ctype?
  symbols : list?
  basetype : ctype? = _ufixint
  unknown : any/c = (lambda (x) (error ....))
```

Takes a list of symbols and generates an enumeration type. The enumeration maps between a symbol in the given *symbols* list and corresponding integers, counting from `0`.

To call a foreign function that takes an enum as a parameter simply provide the symbol of the desired enum as an argument.

```
; example sdl call
(sdl-create-window "title" ... 'SDL_WINDOW_OPENGL)
```

The list *symbols* can also set the values of symbols by putting `'=` and an exact integer after the symbol. For example, the list `'(x y = 10 z)` maps `'x` to `0`, `'y` to `10`, and `'z` to `11`.

The *basetype* argument specifies the base type to use.

The *unknown* argument specifies the result of converting an unknown integer from the foreign side: it can be a one-argument function to be applied on the integer, or a value to return instead. The default is to throw an exception.

Examples:

```
; example from snappy-c.h
> (define _snappy_status
    (_enum '(ok = 0
             invalid_input
             buffer_too_small)))
```

Note that the default basetype is `_ufixint`. This differs from C enumerations that can use any value in `_fixint`. Any `_enum` using negative values should use `_fixint` for the base type.

Example:

```
> (define _negative_enum
    (_enum '(unkown = -1
             error = 0
             ok = 1)
           _fixint))
```

```
(_bitmask symbols [basetype]) → ctype?
  symbols : (or symbol? list?)
  basetype : ctype? = _uint
```

Similar to `_enum`, but the resulting mapping translates a list of symbols to a number and back, using `bitwise-ior` on the values of individual symbols, where A single symbol is equivalent to a list containing just the symbol.

In other words, to call a foreign function that uses bitmask parameters simply call the procedure with the list of wanted flags.

```
; example call from curl_global_init in curl.h
(curl-global-init '(CURL_GLOBAL_SSL CURL_GLOBAL_WIN32))
```

When a symbol does not have a given value (via `'=` after the symbol in *symbols*), its value is the next power of 2 greater than the previous symbol's assignment (or `1` for the first symbol).

The default *basetype* is `_uint`, since high bits are often used for flags.

Examples:

```
; example from curl.h
> (define _curl_global_flag
    (_bitmask `(CURL_GLOBAL_SSL = 1
                CURL_GLOBAL_WIN32 = 2
                CURL_GLOBAL_ALL = 3
                CURL_GLOBAL_NOTHING = 0
                CURL_GLOBAL_DEFAULT = 3
                CURL_GLOBAL_ACK_EINTR = 4)))
; example from XOrg
> (define _Modifiers
    (_bitmask '(ShiftMask = 1
                LockMask = 2
                ControlMask = 4
```

```
Mod1Mask = 8
Mod2Mask = 16
Mod3Mask = 32
Mod4Mask = 64
Mod5Mask = 128
Button1Mask = 256
Button2Mask = 512
Button3Mask = 1024
Button4Mask = 2048
Button5Mask = 4096
Any = 32768)))
```

# 4 Pointer Functions

```
(cpointer? v) → boolean?
  v : any/c
```

Returns #t if *v* is a C pointer or a value that can be used as a pointer: #f (used as a NULL pointer), byte strings (used as memory blocks), or a structure instance with the prop:cpointer structure type property. Returns #f for other values.

```
(ptr-equal? cptr1 cptr2) → boolean?
  cptr1 : cpointer?
  cptr2 : cpointer?
```

Compares the values of the two pointers. Two different Racket pointer objects can contain the same pointer.

If the values are both pointers that are not represented by #f, a byte string, a callback, a pointer based on _fpointer, or a structure with the prop:cpointer property, then the ptr-equal? comparison is the same as using equal?.

```
(ptr-add cptr offset [type]) → cpointer?
  cptr : cpointer?
  offset : exact-integer?
  type : ctype? = _byte
```

Returns a cpointer that is like *cptr* offset by *offset* instances of ctype.

The resulting cpointer keeps the base pointer and offset separate. The two pieces are combined at the last minute before any operation on the pointer, such as supplying the pointer to a foreign function. In particular, the pointer and offset are not combined until after all allocation leading up to a foreign-function call; if the called function does not itself call anything that can trigger a garbage collection, it can safely use pointers that are offset into the middle of a GCable object.

```
(offset-ptr? cptr) → boolean?
  cptr : cpointer?
```

A predicate for cpointers that have an offset, such as pointers that were created using ptr-add. Returns #t even if such an offset happens to be 0. Returns #f for other cpointers and non-cpointers.

```
(ptr-offset cptr) → exact-integer?
  cptr : cpointer?
```

Returns the offset of a pointer that has an offset. The resulting offset is always in bytes.

```
(cpointer-gcable? cptr) → boolean?
  cptr : cpointer?
```

Returns #t if *cptr* is treated as a reference to memory that is (assumed to be) managed by the garbage collector, #f otherwise.

For a pointer based on _gcpointer as a result type, cpointer-gcable? will return #t. For a pointer based on _pointer as a result type, cpointer-gcable? will return #f.

## 4.1 Pointer Dereferencing

```
(set-ptr-offset! cptr offset [ctype]) → void?
  cptr : cpointer?
  offset : exact-integer?
  ctype : ctype? = _byte
```

Sets the offset component of an offset pointer. The arguments are used in the same way as ptr-add. If *cptr* has no offset, the exn:fail:contract exception is raised.

```
(ptr-add! cptr offset [ctype]) → void?
  cptr : cpointer?
  offset : exact-integer?
  ctype : ctype? = _byte
```

Like ptr-add, but destructively modifies the offset contained in a pointer. The same operation could be performed using ptr-offset and set-ptr-offset!.

```
(ptr-ref cptr type [offset]) → any
  cptr : cpointer?
  type : ctype?
  offset : exact-nonnegative-integer? = 0
(ptr-ref cptr type abs-tag offset) → any
  cptr : cpointer?
  type : ctype?
  abs-tag : 'abs
  offset : exact-nonnegative-integer?
(ptr-set! cptr type val) → void?
  cptr : cpointer?
  type : ctype?
  val : any/c
(ptr-set! cptr type offset val) → void?
  cptr : cpointer?
  type : ctype?
```

```
  offset : exact-nonnegative-integer?
  val : any/c
(ptr-set! cptr type abs-tag offset val) → void?
  cptr : cpointer?
  type : ctype?
  abs-tag : 'abs
  offset : exact-nonnegative-integer?
  val : any/c
```

The ptr-ref procedure returns the object referenced by *cptr*, using the given *type*. The ptr-set! procedure stores the *val* in the memory *cptr* points to, using the given *type* for the conversion.

In each case, *offset* defaults to 0 (which is the only value that should be used with ffi-obj objects, see §7 "Unexported Primitive Functions"). If an *offset* index is non-0, the value is read or stored at that location, considering the pointer as a vector of *type*s — so the actual address is the pointer plus the size of *type* multiplied by *offset*. In addition, a 'abs flag can be used to use the *offset* as counting bytes rather then increments of the specified *type*.

Beware that the ptr-ref and ptr-set! procedure do not keep any meta-information on how pointers are used. It is the programmer's responsibility to use this facility only when appropriate. For example, on a little-endian machine:

```
> (define block (malloc _int 5))
> (ptr-set! block _int 0 196353)
> (map (lambda (i) (ptr-ref block _byte i)) '(0 1 2 3))
(1 255 2 0)
```

In addition, ptr-ref and ptr-set! cannot detect when offsets are beyond an object's memory bounds; out-of-bounds access can easily lead to a segmentation fault or memory corruption.

```
(memmove  cptr
          [offset]
           src-cptr
          [src-offset]
           count
          [type])       → void?
  cptr : cpointer?
  offset : exact-integer? = 0
  src-cptr : cpointer?
  src-offset : exact-integer? = 0
  count : exact-nonnegative-integer?
  type : ctype? = _byte
```

Copies to *cptr* from *src-cptr*. The destination pointer can be offset by an optional *offset*, which is in *type* instances. The source pointer can be similarly offset by *src-offset*. The number of bytes copied from source to destination is determined by *count*, which is in *type* instances when supplied.

```
(memcpy  cptr
         [offset]
          src-cptr
         [src-offset]
          count
         [type])       → void?
  cptr : cpointer?
  offset : exact-integer? = 0
  src-cptr : cpointer?
  src-offset : exact-integer? = 0
  count : exact-nonnegative-integer?
  type : ctype? = _byte
```

Like memmove, but the result is undefined if the destination and source overlap.

```
(memset cptr [offset] byte count [type]) → void?
  cptr : cpointer?
  offset : exact-integer? = 0
  byte : byte?
  count : exact-nonnegative-integer?
  type : ctype? = _byte
```

Similar to memmove, but the destination is uniformly filled with *byte* (i.e., an exact integer between 0 and 255 inclusive). When a *type* argument is present, the result is that of a call to memset with no *type* argument and the *count* multiplied by the size associated with the *type*.

```
(cpointer-tag cptr) → any
  cptr : cpointer?
```

Returns the Racket object that is the tag of the given *cptr* pointer.

```
(set-cpointer-tag! cptr tag) → void?
  cptr : cpointer?
  tag : any/c
```

Sets the tag of the given *cptr*. The *tag* argument can be any arbitrary value; other pointer operations ignore it. When a cpointer value is printed, its tag is shown if it is a symbol, a byte string, a string. In addition, if the tag is a pair holding one of these in its car, the car is shown (so that the tag can contain other information).

## 4.2 Memory Management

For general information on C-level memory management with Racket, see *Inside: Racket C API*.

```
(malloc  bytes-or-type
         [type-or-bytes
          cptr
          mode
          fail-mode])    → cpointer?
  bytes-or-type : (or/c (and/c exact-nonnegative-integer? fixnum?)
                        ctype?)
  type-or-bytes : (or/c (and/c exact-nonnegative-integer? fixnum?)
                        ctype?)
                  = absent
  cptr : cpointer? = absent
  mode : (or/c 'raw 'atomic 'nonatomic 'tagged        = absent
               'atomic-interior 'interior
               'zeroed-atomic 'zeroed-atomic-interior
               'stubborn 'uncollectable 'eternal)
  fail-mode : (or/c 'fail-ok 'failok) = absent
```

Allocates a memory block of a specified size using a specified allocation. The result is a C pointer to the allocated memory, or `#f` if the requested size is zero. Although not reflected above, the four arguments can appear in any order, since they are all different types of Racket objects; a size specification is required at minimum:

- If a C type *bytes-or-type* is given, its size is used to determine the block allocation size.
- If an integer *bytes-or-type* is given, it specifies the required size in bytes.
- If both *bytes-or-type* and *type-or-bytes* are given, then the allocated size is for a vector of values (the multiplication of the size of the C type and the integer).
- If a *cptr* pointer is given, its content is copied to the new block.
- A symbol *mode* argument can be given, which specifies what allocation function to use. It should be one of the following:
  - `'raw` — Allocates memory that is outside the garbage collector's space and is not traced by the garbage collector (i.e., is treated as holding no pointers to collectable memory). This memory must be freed with `free`. The initial content of the memory is unspecified.
  - `'atomic` — Allocates memory that can be reclaimed by the garbage collector but is not traced by the garbage collector. The initial content of the memory is unspecified.

For the BC Racket implementation, this allocation mode corresponds to `scheme_malloc_atomic` in the C API.

- `'nonatomic` — Allocates memory that can be reclaimed by the garbage collector, is treated by the garbage collector as holding only pointers, and is initially filled with zeros. The memory is allowed to contain a mixture of references to objects managed by the garbage collector and addresses that are outside the garbage collector's space.

  For the BC Racket implementation, this allocation mode corresponds to `scheme_malloc` in the C API.

- `'atomic-interior` — Like `'atomic`, but the allocated object will not be moved by the garbage collector as long as the allocated object is retained.

  A better name for this allocation mode would be `'atomic-immobile`, but it's `'atomic-interior` for historical reasons.

  For the BC Racket implementation, a reference can point to the interior of the object, instead of its starting address. This allocation mode corresponds to `scheme_malloc_atomic_allow_interior` in the C API.

- `'interior` — Like `'nonatomic`, but the allocated object will not be moved by the garbage collector as long as the allocated object is retained.

  A better name for this allocation mode would be `'nonatomic-immobile`, but it's `'interior` for historical reasons.

  For the BC Racket implementation, a reference can point to the interior of the object, instead of its starting address. This allocation mode corresponds to `scheme_malloc_allow_interior` in the C API.

- `'zeroed-atomic` — Like `'atomic`, but the allocated object is filled with zeros, instead of having unspecified initial content.

- `'zeroed-atomic-interior` — Like `'atomic-interior`, but the allocated object is filled with zeros, instead of having unspecified initial content.

- `'tagged` — Allocates memory that must start with a `short` value that is registered as a tag with the garbage collector.

  This mode is supported only for the BC Racket implementation, and it corresponds to `scheme_malloc_tagged` in the C API.

- `'stubborn` — Like `'nonatomic`, but supports a hint to the GC via `end-stubborn-change` after all changes to the object have been made.

  This mode is supported only for the BC Racket implementation, and it corresponds to `scheme_malloc_stubborn` in the C API.

- `'eternal` — Like `'raw`, except the allocated memory cannot be freed.

  This mode is supported only for the CGC Racket variant, and it corresponds to `scheme_malloc_uncollectable` in the C API.

- `'uncollectable` — Allocates memory that is never collected, cannot be freed, and potentially contains pointers to collectable memory.

  This mode is supported only for the CGC Racket variant, and it corresponds to `scheme_malloc_uncollectable` in the C API.

- If an additional `'failok` or `'fail-ok` flag is given, then some effort may be made to detect an allocation failure and raise `exn:fail:out-of-memory` instead of crashing.

If no mode is specified, then `'nonatomic` allocation is used when the type is a `_gcpointer`- or `_scheme`-based type, and `'atomic` allocation is used otherwise.

Changed in version 6.4.0.10 of package `base`: Added the `'tagged` allocation mode.
Changed in version 8.0.0.13: Changed CS to support the `'interior` allocation mode.
Changed in version 8.1.0.6: Changed CS to remove constraints on the use of memory allocated with the `'nonatomic` and `'interior` allocation modes.
Changed in version 8.14.0.4: Added the `'zeroed-atomic` `'zeroed-atomic-interior` allocation modes.
Changed in version 9.2.0.2: Added support for both `'failok` and `'fail-ok` to accomodate an earlier mismatch between CS and BC implementations.

```
(free cptr) → void
  cptr : cpointer?
```

Uses the operating system's `free` function for `'raw`-allocated pointers, and for pointers that a foreign library allocated and we should free. Note that this is useful as part of a finalizer (see below) procedure hook (e.g., on the Racket pointer object, freeing the memory when the pointer object is collected, but beware of aliasing).

Memory allocated with `malloc` and modes other than `'raw` must not be `free`d, since those modes allocate memory that is managed by the garbage collector.

```
(end-stubborn-change cptr) → void?
  cptr : cpointer?
```

Uses `scheme_end_stubborn_change` on the given stubborn-allocated pointer.

```
(malloc-immobile-cell v) → cpointer?
  v : any/c
```

Allocates memory large enough to hold one arbitrary (collectable) Racket value, but that is not itself collectable or moved by the memory manager. The cell is initialized with `v`; use the type `_scheme` with `ptr-ref` and `ptr-set!` to get or set the cell's value. The cell must be explicitly freed with `free-immobile-cell`.

```
(free-immobile-cell cptr) → void?
  cptr : cpointer?
```

Frees an immobile cell created by `malloc-immobile-cell`.

```
(register-finalizer obj finalizer) → void?
  obj : any/c
  finalizer : (any/c . -> . any)
```

Registers a finalizer procedure *finalizer* with the given *obj*, which can be any Racket (GC-able) object. The finalizer is registered with a "late" will executor that makes wills ready for a value only after all weak references (such as in a weak box) for the value have been cleared, which implies that the value is unreachable and no normal will executor has a will ready for the value. The finalizer is invoked when the will for *obj* becomes ready in the "late" will executor, which means that the value is unreachable (even from wills, and even from itself) by safe code.

The finalizer is invoked in a thread that is in charge of triggering will executors for `register-finalizer`. The given *finalizer* procedure should generally not rely on the environment of the triggering thread, and it must not use any parameters or call any parameter functions, except that relying on a default logger and/or calling `current-logger` is allowed.

Finalizers are mostly intended to be used with cpointer objects (for freeing unused memory that is not under GC control), but it can be used with any Racket object—even ones that have nothing to do with foreign code. Note, however, that the finalizer is registered for the *Racket* object that represents the pointer. If you intend to free a pointer object, then you must be careful to not register finalizers for two cpointers that point to the same address. Also, be careful to not make the finalizer a closure that holds on to the object. Finally, beware that the finalizer is not guaranteed to be run when a place exits; see `ffi/unsafe/alloc` and `register-finalizer-and-custodian-shutdown` for more complete solutions.

As an example for `register-finalizer`, suppose that you're dealing with a foreign function that returns a C pointer that you should free, but you mostly want to use the memory at a 16-byte offset. Here is an attempt at creating a suitable type:

```
(define _pointer-at-sixteen/free
  (make-ctype _pointer
              #f ; i.e., just _pointer as an argument type
              (lambda (x)
                (let ([p (ptr-add x 16)])
                  (register-finalizer x free)
                  p))))
```

The above code is wrong: the finalizer is registered for `x`, which is no longer needed after the new pointer `p` is created. Changing the example to register the finalizer for `p` corrects the problem, but then `free` is invoked `p` instead of on `x`. In the process of fixing this problem, we might be careful and log a message for debugging:

```
(define _pointer-at-sixteen/free
  (make-ctype _pointer
              #f
              (lambda (x)
                (let ([p (ptr-add x 16)])
                  (register-finalizer p
```

```
    (lambda (ignored)
      (log-debug (format "Releasing ~s\n" p))
      (free x)))
  p))))
```

Now, we never see any logged event. The problem is that the finalizer is a closure that keeps a reference to `p`. Instead of referencing the value that is finalized, use the input argument to the finalizer; simply changing `ignored` to `p` above solves the problem. (Removing the debugging message also avoids the problem, since the finalization procedure would then not close over `p`.)

```
(make-late-weak-box v) → weak-box?
  v : any/c
(make-late-weak-hasheq v) → (and/c hash? hash-eq? hash-weak?)
  v : any/c
```

Like `make-weak-box` and `make-weak-hasheq`, but with "late" weak references that last longer than references in the result of `make-weak-box` or `make-weak-hasheq`. Specifically, a "late" weak reference remains intact if a value is unreachable but not yet processed by a finalizer installed with `register-finalizer`. "Late" weak references are intended for use by such finalizers.

```
(make-sized-byte-string cptr length) → bytes?
  cptr : cpointer?
  length : exact-nonnegative-integer?
```

Returns a byte string made of the given pointer and the given length in the BC implementation of Racket; no copying is performed. In the CS implementation, the `exn:fail:unsupported` exception is raised.

Beware that the representation of a Racket byte string normally requires a nul terminator at the end of the byte string (after *length* bytes), but some functions work with a byte-string representation that has no such terminator—notably `bytes-copy`.

If *cptr* is an offset pointer created by `ptr-add`, the offset is immediately added to the pointer. Thus, this function cannot be used with `ptr-add` to create a substring of a Racket byte string, because the offset pointer would be to the middle of a collectable object (which is not allowed).

```
(void/reference-sink v ...) → void?
  v : any/c
```

Returns `#<void>`, but unlike calling the `void` function where the compiler may optimize away the call and replace it with a `#<void>` result, calling `void/reference-sink` ensures that the arguments are considered reachable by the garbage collector until the call returns.

Added in version 6.10.1.2 of package `base`.

## 4.3 Pointer Structure Property

```
prop:cpointer : struct-type-property?
```

A structure type property that causes instances of a structure type to work as C pointer values. The property value must be either an exact non-negative integer indicating an immutable field in the structure (which must, in turn, be initialized to a C pointer value), a procedure that takes the structure instance and returns a C pointer value, or a C pointer value.

The `prop:cpointer` property allows a structure instance to be used transparently as a C pointer value, or it allows a C pointer value to be transparently wrapped by a structure that may have additional values or properties.

# 5 Derived Utilities

## 5.1 Safe Homogenous Vectors

```
(require ffi/vector)
```
package: base

Homogenous vectors are similar to C vectors (see §5.2 “Safe C Vectors”), except that they define different types of vectors, each with a fixed element type. An exception is the u8 family of bindings, which are just aliases for byte-string bindings; for example, make-u8vector is an alias for make-bytes.

```
(make-u8vector len) → u8vector?
  len : exact-nonnegative-integer?
(u8vector val ...) → u8vector?
  val : byte?
(u8vector? v) → boolean?
  v : any/c
(u8vector-length vec) → exact-nonnegative-integer?
  vec : u8vector?
(u8vector-ref vec k) → byte?
  vec : u8vector?
  k : exact-nonnegative-integer?
(u8vector-set! vec k val) → void?
  vec : u8vector?
  k : exact-nonnegative-integer?
  val : byte?
(list->u8vector lst) → u8vector?
  lst : (listof byte?)
(u8vector->list vec) → (listof byte?)
  vec : u8vector?
(u8vector->cpointer vec) → cpointer?
  vec : u8vector?
```

Like _cvector, but for vectors of _uint8 elements. These are aliases for byte operations, where u8vector->cpointer is the identity function.

```
(_u8vector mode maybe-len)
_u8vector
```

Like _cvector, but for vectors of _uint8 elements.

```
(make-s8vector len) → s8vector?
  len : exact-nonnegative-integer?
(s8vector val ...) → s8vector?
```

```
  val : (integer-in -128 127)
(s8vector? v) → boolean?
  v : any/c
(s8vector-length vec) → exact-nonnegative-integer?
  vec : s8vector?
(s8vector-ref vec k) → (integer-in -128 127)
  vec : s8vector?
  k : exact-nonnegative-integer?
(s8vector-set! vec k val) → void?
  vec : s8vector?
  k : exact-nonnegative-integer?
  val : (integer-in -128 127)
(list->s8vector lst) → s8vector?
  lst : (listof (integer-in -128 127))
(s8vector->list vec) → (listof (integer-in -128 127))
  vec : s8vector?
(s8vector->cpointer vec) → cpointer?
  vec : s8vector?
```

Like `make-vector`, etc., but for `_int8` elements. The `s8vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_s8vector mode maybe-len)
_s8vector
```

Like `_cvector`, but for vectors of `_int8` elements.

```
(make-s16vector len) → s16vector?
  len : exact-nonnegative-integer?
(s16vector val ...) → s16vector?
  val : (integer-in -32768 32767)
(s16vector? v) → boolean?
  v : any/c
(s16vector-length vec) → exact-nonnegative-integer?
  vec : s16vector?
(s16vector-ref vec k) → (integer-in -32768 32767)
  vec : s16vector?
  k : exact-nonnegative-integer?
(s16vector-set! vec k val) → void?
  vec : s16vector?
  k : exact-nonnegative-integer?
  val : (integer-in -32768 32767)
(list->s16vector lst) → s16vector?
  lst : (listof (integer-in -32768 32767))
```

```
(s16vector->list vec) → (listof (integer-in -32768 32767))
  vec : s16vector?
(s16vector->cpointer vec) → cpointer?
  vec : s16vector?
```

Like make-vector, etc., but for _int16 elements. The s16vector->cpointer function extracts a plain pointer to the underlying array.

```
(_s16vector mode maybe-len)
_s16vector
```

Like _cvector, but for vectors of _int16 elements.

```
(make-u16vector len) → u16vector?
  len : exact-nonnegative-integer?
(u16vector val ...) → u16vector?
  val : (integer-in 0 65535)
(u16vector? v) → boolean?
  v : any/c
(u16vector-length vec) → exact-nonnegative-integer?
  vec : u16vector?
(u16vector-ref vec k) → (integer-in 0 65535)
  vec : u16vector?
  k : exact-nonnegative-integer?
(u16vector-set! vec k val) → void?
  vec : u16vector?
  k : exact-nonnegative-integer?
  val : (integer-in 0 65535)
(list->u16vector lst) → u16vector?
  lst : (listof (integer-in 0 65535))
(u16vector->list vec) → (listof (integer-in 0 65535))
  vec : u16vector?
(u16vector->cpointer vec) → cpointer?
  vec : u16vector?
```

Like make-vector, etc., but for _uint16 elements. The u16vector->cpointer function extracts a plain pointer to the underlying array.

```
(_u16vector mode maybe-len)
_u16vector
```

Like _cvector, but for vectors of _uint16 elements.

```
(make-s32vector len) → s32vector?
  len : exact-nonnegative-integer?
(s32vector val ...) → s32vector?
  val : (integer-in -2147483648 2147483647)
(s32vector? v) → boolean?
  v : any/c
(s32vector-length vec) → exact-nonnegative-integer?
  vec : s32vector?
(s32vector-ref vec k) → (integer-in -2147483648 2147483647)
  vec : s32vector?
  k : exact-nonnegative-integer?
(s32vector-set! vec k val) → void?
  vec : s32vector?
  k : exact-nonnegative-integer?
  val : (integer-in -2147483648 2147483647)
(list->s32vector lst) → s32vector?
  lst : (listof (integer-in -2147483648 2147483647))
(s32vector->list vec)
 → (listof (integer-in -2147483648 2147483647))
  vec : s32vector?
(s32vector->cpointer vec) → cpointer?
  vec : s32vector?
```

Like `make-vector`, etc., but for `_int32` elements. The `s32vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_s32vector mode maybe-len)
_s32vector
```

Like `_cvector`, but for vectors of `_int32` elements.

```
(make-u32vector len) → u32vector?
  len : exact-nonnegative-integer?
(u32vector val ...) → u32vector?
  val : (integer-in 0 4294967295)
(u32vector? v) → boolean?
  v : any/c
(u32vector-length vec) → exact-nonnegative-integer?
  vec : u32vector?
(u32vector-ref vec k) → (integer-in 0 4294967295)
  vec : u32vector?
  k : exact-nonnegative-integer?
(u32vector-set! vec k val) → void?
```

```
  vec : u32vector?
  k : exact-nonnegative-integer?
  val : (integer-in 0 4294967295)
(list->u32vector lst) → u32vector?
  lst : (listof (integer-in 0 4294967295))
(u32vector->list vec) → (listof (integer-in 0 4294967295))
  vec : u32vector?
(u32vector->cpointer vec) → cpointer?
  vec : u32vector?
```

Like make-vector, etc., but for _uint32 elements. The u32vector->cpointer function extracts a plain pointer to the underlying array.

```
(_u32vector mode maybe-len)
_u32vector
```

Like _cvector, but for vectors of _uint32 elements.

```
(make-s64vector len) → s64vector?
  len : exact-nonnegative-integer?
(s64vector val ...) → s64vector?
  val : (integer-in -9223372036854775808 9223372036854775807)
(s64vector? v) → boolean?
  v : any/c
(s64vector-length vec) → exact-nonnegative-integer?
  vec : s64vector?
(s64vector-ref vec k)
 → (integer-in -9223372036854775808 9223372036854775807)
  vec : s64vector?
  k : exact-nonnegative-integer?
(s64vector-set! vec k val) → void?
  vec : s64vector?
  k : exact-nonnegative-integer?
  val : (integer-in -9223372036854775808 9223372036854775807)
(list->s64vector lst) → s64vector?
  lst : (listof (integer-in -9223372036854775808 9223372036854775807))
(s64vector->list vec)
 → (listof (integer-in -9223372036854775808 9223372036854775807))
  vec : s64vector?
(s64vector->cpointer vec) → cpointer?
  vec : s64vector?
```

Like make-vector, etc., but for _int64 elements. The s64vector->cpointer function extracts a plain pointer to the underlying array.

```
(_s64vector mode maybe-len)
_s64vector
```

Like _cvector, but for vectors of _int64 elements.

```
(make-u64vector len) → u64vector?
  len : exact-nonnegative-integer?
(u64vector val ...) → u64vector?
  val : (integer-in 0 18446744073709551615)
(u64vector? v) → boolean?
  v : any/c
(u64vector-length vec) → exact-nonnegative-integer?
  vec : u64vector?
(u64vector-ref vec k) → (integer-in 0 18446744073709551615)
  vec : u64vector?
  k : exact-nonnegative-integer?
(u64vector-set! vec k val) → void?
  vec : u64vector?
  k : exact-nonnegative-integer?
  val : (integer-in 0 18446744073709551615)
(list->u64vector lst) → u64vector?
  lst : (listof (integer-in 0 18446744073709551615))
(u64vector->list vec)
 → (listof (integer-in 0 18446744073709551615))
  vec : u64vector?
(u64vector->cpointer vec) → cpointer?
  vec : u64vector?
```

Like make-vector, etc., but for _uint64 elements. The u64vector->cpointer function extracts a plain pointer to the underlying array.

```
(_u64vector mode maybe-len)
_u64vector
```

Like _cvector, but for vectors of _uint64 elements.

```
(make-f32vector len) → f32vector?
  len : exact-nonnegative-integer?
(f32vector val ...) → f32vector?
  val : real?
(f32vector? v) → boolean?
  v : any/c
```

```
(f32vector-length vec) → exact-nonnegative-integer?
  vec : f32vector?
(f32vector-ref vec k) → real?
  vec : f32vector?
  k : exact-nonnegative-integer?
(f32vector-set! vec k val) → void?
  vec : f32vector?
  k : exact-nonnegative-integer?
  val : real?
(list->f32vector lst) → f32vector?
  lst : (listof real?)
(f32vector->list vec) → (listof real?)
  vec : f32vector?
(f32vector->cpointer vec) → cpointer?
  vec : f32vector?
```

Like `make-vector`, etc., but for `_float` elements. The `f32vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_f32vector mode maybe-len)
_f32vector
```

Like `_cvector`, but for vectors of `_float` elements.

```
(make-f64vector len) → f64vector?
  len : exact-nonnegative-integer?
(f64vector val ...) → f64vector?
  val : real?
(f64vector? v) → boolean?
  v : any/c
(f64vector-length vec) → exact-nonnegative-integer?
  vec : f64vector?
(f64vector-ref vec k) → real?
  vec : f64vector?
  k : exact-nonnegative-integer?
(f64vector-set! vec k val) → void?
  vec : f64vector?
  k : exact-nonnegative-integer?
  val : real?
(list->f64vector lst) → f64vector?
  lst : (listof real?)
(f64vector->list vec) → (listof real?)
  vec : f64vector?
(f64vector->cpointer vec) → cpointer?
```

```
  vec : f64vector?
```

Like `make-vector`, etc., but for `_double*` elements. The `f64vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_f64vector mode maybe-len)
_f64vector
```

Like `_cvector`, but for vectors of `_double*` elements.

```
(make-f80vector len) → f80vector?
  len : exact-nonnegative-integer?
(f80vector val ...) → f80vector?
  val : extflonum?
(f80vector? v) → boolean?
  v : any/c
(f80vector-length vec) → exact-nonnegative-integer?
  vec : f80vector?
(f80vector-ref vec k) → extflonum?
  vec : f80vector?
  k : exact-nonnegative-integer?
(f80vector-set! vec k val) → void?
  vec : f80vector?
  k : exact-nonnegative-integer?
  val : extflonum?
(list->f80vector lst) → f80vector?
  lst : (listof extflonum?)
(f80vector->list vec) → (listof extflonum?)
  vec : f80vector?
(f80vector->cpointer vec) → cpointer?
  vec : f80vector?
```

Like `make-vector`, etc., but for `_longdouble` elements. The `f80vector->cpointer` function extracts a plain pointer to the underlying array.

```
(_f80vector mode maybe-len)
_f80vector
```

Like `_cvector`, but for vectors of `_longdouble` elements.

## 5.2 Safe C Vectors

`(require ffi/cvector)` package: `base`

```
(require ffi/unsafe/cvector)
```

The ffi/unsafe/cvector library exports the bindings of this section. The ffi/cvector library exports the same bindings, except for the unsafe make-cvector* operation.

The cvector form can be used as a type C vectors (i.e., a pointer to a memory block).

```
(_cvector mode type maybe-len)
_cvector
```

Like _bytes, _cvector can be used as a simple type that corresponds to a pointer that is managed as a safe C vector on the Racket side. The longer form behaves similarly to the _list and _vector custom types, except that _cvector is more efficient; no Racket list or vector is needed.

```
(make-cvector type length) → cvector?
  type : ctype?
  length : exact-nonnegative-integer?
```

Allocates a C vector using the given type and length. The resulting vector is not guaranteed to contain any particular values.

```
(cvector type val ...) → cvector?
  type : ctype?
  val : any/c
```

Creates a C vector of the given type, initialized to the given list of vals.

```
(cvector? v) → boolean?
  v : any/c
```

Returns #t if v is a C vector, #f otherwise.

```
(cvector-length cvec) → exact-nonnegative-integer?
  cvec : cvector?
```

Returns the length of a C vector.

```
(cvector-type cvec) → ctype?
  cvec : cvector?
```

Returns the C type object of a C vector.

```
(cvector-ptr cvec) → cpointer?
  cvec : cvector?
```

Returns the pointer that points at the beginning block of the given C vector.

```
(cvector-ref cvec k) → any
  cvec : cvector?
  k : exact-nonnegative-integer?
```

References the *k*th element of the *cvec* C vector. The result has the type that the C vector uses.

```
(cvector-set! cvec k val) → void?
  cvec : cvector?
  k : exact-nonnegative-integer?
  val : any
```

Sets the *k*th element of the *cvec* C vector to *val*. The *val* argument should be a value that can be used with the type that the C vector uses.

```
(cvector->list cvec) → list?
  cvec : cvector?
```

Converts the *cvec* C vector object to a list of values.

```
(list->cvector lst type) → cvector?
  lst : list?
  type : ctype?
```

Converts the list *lst* to a C vector of the given *type*.

```
(make-cvector* cptr type length) → cvector?
  cptr : any/c
  type : ctype?
  length : exact-nonnegative-integer?
```

Constructs a C vector using an existing pointer object. This operation is not safe, so it is intended to be used in specific situations where the *type* and *length* are known.

## 5.3 Tagged C Pointer Types

The unsafe cpointer-has-tag? and cpointer-push-tag! operations manage tags to distinguish pointer types.

```
(_cpointer  tag
           [ptr-type
            racket-to-c
            c-to-racket]) → ctype?
```

```
  tag : any/c
  ptr-type : (or/c ctype? #f) = _pointer
  racket-to-c : (or/c (any/c . -> . any/c) #f) = values
  c-to-racket : (or/c (any/c . -> . any/c) #f) = values
(_cpointer/null  tag
                 [ptr-type
                  racket-to-c
                  c-to-racket]) → ctype?
  tag : any/c
  ptr-type : (or/c ctype? #f) = _pointer
  racket-to-c : (or/c (any/c . -> . any/c) #f) = values
  c-to-racket : (or/c (any/c . -> . any/c) #f) = values
```

Constructs a C pointer type, *_tag*, that gets a specific tag when converted to Racket, and accept only such tagged pointers when going to C. For any optional argument, `#f` is treated the same as the default value of the argument.

The *ptr-type* is used as the base pointer type for *_tag*. Values of *ptr-type* must be represented as pointers.

Although any value can be used as *tag*, by convention it is the symbol form of a type name—without a leading underscore. For example, a pointer type `_animal` would normally use `'animal` as the tag.

Pointer tags are checked with `cpointer-has-tag?` and changed with `cpointer-push-tag!`, which means that other tags are preserved on an existing pointer value. Specifically, if a base *ptr-type* is given and is itself produced by `_cpointer`, then the new type will handle pointers that have the new tag in addition to *ptr-type*'s tag(s). When the tag is a pair, its first value is used for printing, so the most recently pushed tag (which corresponds to the inheriting type) is displayed.

A Racket value to be used as a *_tag* value is first passed to *racket-to-c*, and the result must be a pointer that is tagged with *tag*. Similarly, a C value to be returned as a *_tag* value is initially represented as pointer tagged with *tag*, but then it is passed to *c-to-racket* to obtain the Racket representation. Thus, a *_tag* value is represented by a pointer at the C level, but (unlike the given *ptr-type*) it can have any representation at the Racket level as implemented by *racket-to-c* and *c-to-racket*.

The `_cpointer/null` function is similar to `_cpointer`, except that it tolerates `NULL` pointers both going to C and back. Note that `NULL` pointers are represented as `#f` in Racket, so they are not tagged.

```
(define-cpointer-type _id)
(define-cpointer-type _id #:tag tag-expr)
(define-cpointer-type _id ptr-type-expr)
(define-cpointer-type _id ptr-type-expr #:tag tag-expr)
```

```
(define-cpointer-type _id ptr-type-expr
                      racket-to-c-expr c-to-racket-expr)
(define-cpointer-type _id ptr-type-expr
                      racket-to-c-expr c-to-racket-expr
                      #:tag tag-expr)
```

A macro version of `_cpointer` and `_cpointer/null`, using the defined name for a tag symbol, and defining a predicate too. The *_id* must start with `_`.

The optional expressions produce optional arguments to `_cpointer`.

In addition to defining *_id* to a type generated by `_cpointer`, *_id*`/null` is bound to a type produced by `_cpointer/null` type. Finally, *id*`?` is defined as a predicate, and *id*`-tag` is defined as an accessor to obtain a tag. If provided, the tag is *tag-expr*, otherwise it is the symbol form of *id*.

```
(cpointer-predicate-procedure? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a predicate procedure generated by `define-cpointer-type` or `define-cstruct`, `#f` otherwise.

Added in version 6.6.0.1 of package `base`.

```
(cpointer-has-tag? cptr tag) → boolean?
  cptr : cpointer?
  tag : any/c
(cpointer-push-tag! cptr tag) → void?
  cptr : cpointer?
  tag : any/c
```

These two functions treat pointer tags as lists of tags. As described in §4 “Pointer Functions”, a pointer tag does not have any role, except for Racket code that uses it to distinguish pointers; these functions treat the tag value as a list of tags, which makes it possible to construct pointer types that can be treated as other pointer types, mainly for implementing inheritance via upcasts (when a struct contains a super struct as its first element).

The `cpointer-has-tag?` function checks whether if the given *cptr* has the *tag*. A pointer has a tag *tag* when its tag is either `eq?` to *tag* or a list that contains (in the sense of `memq`) *tag*.

The `cpointer-push-tag!` function pushes the given *tag* value on *cptr*’s tags. The main properties of this operation are: (a) pushing any tag will make later calls to `cpointer-has-tag?` succeed with this tag, and (b) the pushed tag will be used when printing the pointer (until a new value is pushed). Technically, pushing a tag will simply set it if there is no tag set, otherwise push it on an existing list or an existing value (treated as a single-element list).

## 5.4 Serializable C Struct Types

```
(require ffi/serialize-cstruct)
     package: serialize-cstruct-lib
```

```
(define-serializable-cstruct _id ([field-id type-expr] ...)
                               property ...)

property = #:alignment alignment-expr
         | #:malloc-mode malloc-mode-expr
         | #:serialize-inplace
         | #:deserialize-inplace
         | #:version vers
         | #:other-versions ([other-vers deserialize-chain-expr
                                         convert-proc-expr
                                         unconvert-proc-expr
                                         cycle-convert-proc-expr]
                             ...)
         | #:property prop-expr val-expr
```

Like `define-cstruct`, but defines a serializable type, with several changed additional bindings:

- `make-id` — always uses `'atomic` allocation, even if `#:malloc-mode` is specified (for historical reasons).
- `make-id/mode` — like behaves like `make-id` but uses the mode or allocator specified via *malloc-mode-expr* (for historical reasons).
- `deserialize:cstruct:id` (for a *vers* of `0`) or `deserialize:cstruct:id-vvers` (for a *vers* of `1` or more) — deserialization information that is automatically exported from a `deserialize-info` submodule.
- `deserialize-chain:cstruct:id` (for a *vers* of `0`) or `deserialize-chain:cstruct:id-vvers` (for a *vers* of `1` or more) — deserialization information for use via `#:other-versions` in other `define-serializable-cstruct` forms.
- `deserialize:cstruct:id` (for an *other-vers* of `0`) or `deserialize:cstruct:id-vother-vers` (for an *other-vers* of `1` or more) — deserialization information that is automatically exported from a `deserialize-info` submodule.

Instances of the new type fulfill the `serializable?` predicate and can be used with `serialize` and `deserialize`. Serialization may fail if one of the fields contains an arbitrary pointer, an embedded non-serializable C struct, or a pointer to a non-serializable C struct. Array-types are supported as long as they don't contain one of these types.

The default *vers* is 0, and *vers* must be a literal, exact, non-negative integer. An #:other-versions clause provides deserializers for previous versions of the structure with the name *id*, so that previously serialized data can be deserialized after a change to the declaration of *id*. For each *other-vers*, *deserialize-chain-expr* should be the value of a deserialize:cstruct:*other-id* binding for some other "other-id" declared with define-serializable-cstruct that has the same shape that the previous version of *id*; the function produced by *convert-proc-expr* should convert an instance of *other-id* to an instance of *id*. The functions produced by *unconvert-proc-expr* and *cycle-convert-proc-expr* are used if a record is involved in a cycle; the function from *unconvert-proc-expr* takes an *id* instance produced by *convert-proc-expr*'s function back to a *other-id*, while *cycle-convert-proc-expr* returns two values: a shell instance of id and function to accept a filled *other-id* whose content should be moved to the shell instance of id.

The *malloc-mode-expr* arguments control the memory allocation for this type during deserialization and make-*id*/mode. It can be one of the mode arguments to malloc, or a procedure

```
(-> exact-positive-integer? cpointer?)
```

that allocates memory of the given size. The default is malloc with 'atomic.

When #:serialize-inplace is specified, the serialized representation shares memory with the C struct object. While being more efficient, especially for large objects, changes to the object after serialization may lead to changes in the serialized representation.

A #:deserialize-inplace option reuses the memory of the serialized representation, if possible. This option is more efficient for large objects, but it may fall back to allocation via *malloc-mode-expr* for cyclic structures. As the allocation mode of the serialized representation will be 'atomic by default or may be arbitrary if #:serialize-inplace is specified, inplace deserialisation should be used with caution whenever the object contains pointers.

When the C struct contains pointers, it is advisable to use a custom allocator. It should be based on a non-moving-memory allocation like 'raw, potentially with manual freeing to avoid memory leaks after garbage collection.

Changed in version 1.1 of package serialize-cstruct-lib: Added #:version and #:other-versions.

Examples:

```
> (define-serializable-cstruct fish ([color _int]))
> (define f0/s (serialize (make-fish 1)))
> (fish-color (deserialize f0/s))
1
> (define-serializable-cstruct aq ([a (_gcable fish-pointer)]
                                   [d (_gcable aq-pointer/null)])
```

```
    #:malloc-mode 'nonatomic)
> (define aq1 (make-aq/mode (make-fish 6) #f))
> (set-aq-d! aq1 aq1) ; create a cycle
> (define aq0/s (serialize aq1))
> (aq-a (aq-d (aq-d (deserialize aq0/s))))
#<cpointer:fish>
; Same shape as original aq:
> (define-serializable-cstruct old-aq ([a (_gcable fish-pointer)]
                                       [d (_gcable _pointer)])
          #:malloc-mode 'nonatomic)
; Replace the original aq:
> (define-serializable-cstruct aq ([a (_gcable fish-pointer)]
                                   [b (_gcable fish-pointer)]
                                   [d (_gcable aq-pointer/null)])
    #:malloc-mode 'nonatomic
    #:version 1
    #:other-versions ([0 deserialize-chain:cstruct:old-aq
                         (lambda (oa)
                           (make-aq/mode (old-aq-a oa)
                                         (old-aq-a oa)
                                         (cast (old-aq-
d oa) _pointer aq-pointer)))
                         (lambda (a)
                           (make-old-aq/mode (aq-a a)
                                             (aq-d a)))
                         (lambda ()
                           (define tmp-fish (make-fish 0))
                           (define a (make-aq/mode tmp-fish tmp-
fish #f))
                           (values a
                                   (lambda (oa)
                                     (set-aq-a! a (old-aq-a oa))
                                     (set-aq-b! a (old-aq-a oa))
                                     (set-aq-d! a (cast (old-aq-
d oa) _pointer aq-pointer)))))]))
; Deserialize old instance to new cstruct:
> (fish-color (aq-a (aq-d (aq-d (deserialize aq0/s)))))
6
> (define aq1/s (serialize (make-aq/mode (make-fish 1) (make-
fish 2) #f)))
; New version of fish:
> (define-serializable-cstruct old-fish ([color _int]))
> (define-serializable-cstruct fish ([weight _float]
                                     [color _int])
    #:version 1
    #:other-versions ([0 deserialize-chain:cstruct:old-fish
```

```
                          (lambda (of)
                            (make-fish 10.0 (old-fish-color of)))
                          (lambda (a) (error "cycles not
possible!"))
                          (lambda () (error "cycles not
possible!"))]))
; Deserialized content upgraded to new fish:
> (fish-color (aq-b (deserialize aq1/s)))
2
> (fish-weight (aq-b (deserialize aq1/s)))
10.0
```

## 5.5 Static Callout and Callback Cores

```
(require ffi/unsafe/static)
```
package: base

The ffi/unsafe/static library provides the same bindings as ffi/unsafe, but with a replacement _fun form.

Added in version 8.11.0.2 of package base.

```
(_fun fun-option ... maybe-args type-spec ... -> type-spec
      maybe-wrapper)
```

Like _fun from ffi/unsafe, but triggers an error at compile time in the CS implementation of Racket if the compiler is unable to infer enough information about the resulting C type to statically generate code for callouts and callbacks using the type.

The *type-spec* forms and some *fun-option* forms within _fun are arbitrary expressions that can compute C types and options at run time. If the optimizer can statically infer underlying representations, then it can generate the necessary code for a callout or callback statically, instead of deferring code generation to run time. This optimization applies even when using _fun from ffi/unsafe, but _fun from ffi/unsafe/static insists that the optimization must apply.

Currently, the benefit of static generation for callout and callback code is limited, because run-time code generation is fast and cached. In the long run, static generation may provide more benefit.

## 5.6 Defining Bindings

```
(require ffi/unsafe/define)
```
package: base

```
(define-ffi-definer define-id ffi-lib-expr
  option ...)

option = #:provide provide-id
       | #:define core-define-id
       | #:default-make-fail default-make-fail-expr
       | #:make-c-id make-c-id
```

Binds *define-id* as a definition form to extract bindings from the library produced by *ffi-lib-expr*. The syntax of *define-id* is

```
(define-id id type-expr
  bind-option ...)

bind-option = #:c-id c-id
            | #:c-id (unquote c-id-expr)
            | #:wrap wrap-expr
            | #:make-fail make-fail-expr
            | #:fail fail-expr
            | #:variable
```

A *define-id* form binds `id` by extracting a binding with the foreign name *c-id* from the library produced by *ffi-lib-expr*, where *c-id* defaults to *id*. The other options support further wrapping and configuration:

- Before the extracted result is bound as *id*, it is passed to the result of *wrap-expr*, which defaults to `values`. Expressions such as `(allocator delete)` or `(deallocator)` are useful as *wrap-expr*s.
- The `#:make-fail` and `#:fail` options are mutually exclusive; if *make-fail-expr* is provided, it is applied to `'id` to obtain the last argument to `get-ffi-obj`; if *fail-expr* is provided, it is supplied directly as the last argument to `get-ffi-obj`. The `make-not-available` function is useful as *make-fail-expr* to cause a use of *id* to report an error when it is applied if *c-id* was not found in the foreign library.
- If the `#:c-id` option is provided with an identifier argument *c-id*, then *c-id* is used as the foreign name. If the argument has the form `(unquote c-id-expr)`, then the foreign name is the result of evaluating *c-id-expr*. In either case, the `#:make-c-id` argument is ignored.
- If the `#:c-id` option is absent, the foreign name is based on *id*. If *define-id* was defined with the `#:make-c-id` option, it computes the foreign name using an ffi identifier convention, such as converting hyphens to underscores or camel case. Several conventions are provided by `ffi/unsafe/define/conventions`. If `#:make-c-id` was absent, then *id* is used as the foreign name.

- If the #:variable keyword is given, then make-c-parameter is used instead of get-ffi-obj to get the foreign value.

If *provide-id* is provided to define-ffi-definer, then *define-id* also provides its binding using *provide-id*. The provide-protected form is usually a good choice for *provide-id*.

If *core-define-id* is provided to define-ffi-definer, then *core-define-id* is used in place of define in the expansion of *define-id* for each binding.

If *default-make-fail-expr* is provided to define-ffi-definer, it serves as the default #:make-fail value for *define-id*.

For example,

```
(define-ffi-definer define-gtk gtk-lib)
```

binds define-gtk to extract FFI bindings from gtk-lib, so that gtk_rc_parse could be bound as

```
(define-gtk gtk_rc_parse (_fun _path -> _void))
```

If gtk_rc_parse is not found, then define-gtk reports an error immediately. If define-gtk is instead defined with

```
(define-ffi-definer define-gtk gtk-lib
  #:default-make-fail make-not-available)
```

then if gtk_rc_parse is not found in gtk-lib, an error is reported only when gtk_rc_parse is called.

Changed in version 6.9.0.5 of package base: Added #:make-c-id parameter.
Changed in version 8.4.0.5: Added #:variable option. Added unquote variant of #:c-id argument.

```
(make-not-available name) → procedure?
  name : symbol?
```

Returns a procedure that takes any number of arguments, including keyword arguments, and raises and exn:fail:unsupported exception from *name*. This function is intended for using with #:make-fail or #:default-make-fail in define-ffi-definer

Changed in version 8.3.0.5 of package base: Added support for keyword arguments.
Changed in version 9.2.0.2: Changed to raise exn:fail:unsupported.

```
(provide-protected provide-spec ...)
```

Equivalent to (provide (protect-out *provide-spec* ...)). The provide-protected identifier is useful with #:provide in define-ffi-definer.

### 5.6.1 FFI Identifier Conventions

```
(require ffi/unsafe/define/conventions)
```
package: base

This module provides several *FFI identifier conventions* for use with `#:make-c-id` in `define-ffi-definer`. A FFI identifier convention is any syntax transformer that converts one identifier to another.

Added in version 6.9.0.5 of package base.

```
convention:hyphen->underscore
```

A convention that converts hyphens in an identifier to underscores. For example, the identifier `underscore-variable` will transform to `underscore_variable`.

```
(define-ffi-definer define-unlib underscore-lib
  #:make-c-id convention:hyphen->underscore)
(define-unlib underscore-variable (_fun -> _void))
```

```
convention:hyphen->camelCase
```

Similar to `convention:hyphen->underscore`, but converts the identifier to "camelCase," following the `string-downcase` and `string-titlecase` functions. For example, the identifier `camel-case-variable` (or even `cAmeL-CAsE-vaRiaBlE`) will transform to `camelCaseVariable`.

```
(define-ffi-definer define-calib camel-lib
  #:make-c-id convention:hyphen->camelCase)
(define-calib camel-case-variable (_fun -> _void))
```

Added in version 8.11.1.8 of package base.

```
convention:hyphen->PascalCase
```

Like `convention:hyphen->camelCase`, but converts the identifier to "PascalCase," following the `string-titlecase` function. For example, the identifier `pascal-case-variable` (or even `paSCaL-CAsE-vaRiaBlE`) will transform to `PascalCaseVariable`.

```
(define-ffi-definer define-palib pascal-lib
  #:make-c-id convention:hyphen->PascalCase)
(define-palib pascal-case-variable (_fun -> _void))
```

Added in version 8.11.1.8 of package base.

```
convention:hyphen->camelcase
```

**NOTE:** This convention is deprecated; use `convention:hyphen->PascalCase`, instead. This convention unfortunately converts to "PascalCase" as opposed to what its name suggests.

Changed in version 8.11.1.8 of package `base`: Deprecated due to the wrong behavior.

## 5.7 Allocation and Finalization

`(require ffi/unsafe/alloc)` package: `base`

The `ffi/unsafe/alloc` library provides utilities for ensuring that values allocated through foreign functions are reliably deallocated.

```
((allocator dealloc
            [#:merely-uninterruptible? uninterruptible?])
 alloc)
 → (or/c procedure? #f)
 dealloc : (any/c . -> . any)
 uninterruptible? : any/c = #f
 alloc : (or/c procedure? #f)
```

Produces an *allocator* procedure that behaves like *alloc*, but each result *v* of the allocator, if not `#f`, is given a finalizer that calls *dealloc* on *v*—unless the call has been canceled by applying a deallocator (produced by `deallocator`) to *v*. Any existing *dealloc* registered for *v* is canceled. If and only if *alloc* is `#f`, `((allocator dealloc) alloc)` produces `#f`.

The resulting allocator calls *alloc* in atomic mode (see `call-as-atomic`), unless *uninterruptible?* is true, in which case uninterruptible mode is used (see `call-as-uninterruptible`). The result from *alloc* is received and registered in atomic or uninterruptible mode, so that the result is reliably deallocated as long as no exception is raised.

The *dealloc* procedure will be called in atomic mode, and it must obey the same constraints as a finalizer procedure provided to `register-finalizer`. The *dealloc* procedure itself need not be specifically a deallocator produced by `deallocator`. If a deallocator is called explicitly, it need not be the same as *dealloc*.

When a non-main place exits, after all custodian-shutdown actions, for every *dealloc* still registered via an allocator or retainer (from `allocator` or `retainer`), the value to deallocate is treated as immediately unreachable. At that point, *dealloc* functions are called in reverse order of their registrations. Note that references in a *dealloc* function's closure do *not* prevent running a *dealloc* function for any other value. If deallocation needs to proceed in an order different than reverse of allocation, use a retainer to insert a new deallocation action that will run earlier.

Changed in version 7.0.0.4 of package base: Added atomic mode for *dealloc* and changed non-main place exits to call all remaining *dealloc*s.
Changed in version 7.4.0.4: Produce #f when *alloc* is #f.
Changed in version 8.17.0.7: Added the #:merely-uninterruptible? optional argument.

```
((deallocator [get-arg
               #:merely-uninterruptible? uninterruptible?])
  dealloc)
 → procedure?
  get-arg : (list? . -> . any/c) = car
  uninterruptible? : any/c = #f
  dealloc : procedure?
((releaser [get-arg]) dealloc) → procedure?
  get-arg : (list? . -> . any/c) = car
  dealloc : procedure?
```

Produces a *deallocator* procedure that behaves like *dealloc*. The deallocator calls *dealloc* in atomic mode (see call-as-atomic) or uninterruptible mode (see call-as-uninterruptible), and for one of its arguments, the it cancels the most recent remaining deallocator registered by a allocator or retainer.

The optional *get-arg* procedure determines which of *dealloc*'s arguments correspond to the released object; *get-arg* receives a list of arguments passed to *dealloc*, so the default car selects the first one. Note that *get-arg* can only choose one of the by-position arguments to *dealloc*, though the deallocator will require and accept the same keyword arguments as *dealloc*, if any.

The releaser procedure is a synonym for deallocator.

Changed in version 8.17.0.7 of package base: Added the #:merely-uninterruptible? optional argument.

```
((retainer  release
            [get-arg
             #:merely-uninterruptible? uninterruptible?])
  retain)
 → procedure?
  release : (any/c . -> . any)
  get-arg : (list? . -> . any/c) = car
  uninterruptible? : any/c = #f
  retain : procedure?
```

Produces a *retainer* procedure that behaves like *retain*. A retainer acts the same as an allocator produced by allocator, except that

- a retainer does not cancel any existing *release* or *dealloc* registrations when registering *release*; and

- *release* is registered for a value *v* that is is an argument to the retainer, instead of the result for an allocator.

The optional *get-arg* procedure determines which of the retainer's arguments (that is, which of *retain*'s arguments) correspond to the retained object *v*; *get-arg* receives a list of arguments passed to *retain*, so the default car selects the first one. Note that *get-arg* can only choose one of the by-position arguments to *retain*, though the retainer will require and accept the same keyword arguments as *retain*, if any.

Changed in version 7.0.0.4 of package base: Added atomic mode for *release* and changed non-main place exits to call all remaining *release*s.
Changed in version 8.17.0.7: Added the #:merely-uninterruptible? optional argument.

## 5.8 Custodian Shutdown Registration

```
(require ffi/unsafe/custodian)
```
package: base

The ffi/unsafe/custodian library provides utilities for registering shutdown callbacks with custodians.

```
(register-custodian-shutdown  v
                              callback
                             [custodian
                              #:at-exit? at-exit?
                              #:weak? weak?
                              #:ordered? ordered?]) → cpointer?
  v : any/c
  callback : (any/c . -> . any)
  custodian : custodian? = (current-custodian)
  at-exit? : any/c = #f
  weak? : any/c = #f
  ordered? : any/c = #f
```

Registers *callback* to be applied (in atomic mode and an unspecified Racket thread) to *v* when *custodian* is shutdown. If *custodian* is already shut down, the result is #f and *v* is not registered. Otherwise, the result is a pointer that can be supplied to unregister-custodian-shutdown to remove the registration.

If *at-exit?* is true, then *callback* is applied when Racket exits, even if the custodian is not explicitly shut down.

If *weak?* is true, then *callback* may not be called if *v* is determined to be unreachable during garbage collection. The value *v* is initially weakly held by the custodian, even if *weak?* is #f. A value associated with a custodian can therefore be finalized via will executors, at

least through will registrations and `register-finalizer` uses *after* calling `register-custodian-shutdown`, but the value becomes strongly held when there are no other strong references and no later-registered finalizers or wills apply.

If *ordered?* is true when `weak` is `#f`, then *v* is retained in a way that allows finalization of *v* via `register-finalizer` to proceed. For the CS implementation of Racket, *v* must not refer to itself or to a value that can refer back to *v*.

Normally, *weak?* should be false. To trigger actions based on finalization or custodian shutdown—whichever happens first—leave *weak?* as `#f` and have a finalizer run in atomic mode to check that the custodian shutdown has not happened and then cancel the shutdown action via `unregister-custodian-shutdown`. If *weak?* is true or if the finalizer is not run in atomic mode, then there's no guarantee that either of the custodian or finalizer callbacks has completed by the time that the custodian shutdown has completed; *v* might be no longer registered to the custodian, while the finalizer for *v* might be still running or merely queued to run. Furthermore, if finalization is via `register-finalizer` (as opposed to a will executor), then supply *ordered?* as true; if *ordered?* is false while *weak?* is false, then *custodian* may retain *v* in a way that does not allow finalization to be triggered when *v* is otherwise inaccessible. See also `register-finalizer-and-custodian-shutdown`.

Changed in version 7.8.0.8 of package `base`: Added the `#:ordered?` argument.

```
(unregister-custodian-shutdown v
                               registration) → void?
  v : any/c
  registration : cpointer?
```

Cancels a custodian-shutdown registration, where *registration* is a previous result from `register-custodian-shutdown` applied to *v*. If *registration* is `#f`, then no action is taken.

```
(register-finalizer-and-custodian-shutdown
 v
 callback
 [custodian
  #:at-exit? at-exit?
  #:custodian-available available-callback
  #:custodian-unavailable unavailable-callback])
→ any
 v : any/c
 callback : (any/c . -> . any)
 custodian : custodian? = (current-custodian)
 at-exit? : any/c = #f
 available-callback : ((any/c . -> . void?) . -> . any)
                    = (lambda (unreg) (void))
 unavailable-callback : ((-> void?) . -> . any)
                      = (lambda (reg-fnl) (reg-fnl))
```

Registers *callback* to be applied (in atomic mode and an unspecified Racket thread) to *v* when *custodian* is shutdown or when *v* is about to be collected by the garbage collector, whichever happens first. The *callback* is only applied to *v* once. The object *v* is subject to the the constraints of `register-finalizer`—particularly the constraint that *v* must not be reachable from itself.

When *v* is successfully registered with *custodian* and a finalizer is registered, then *available-callback* is called with a function *unreg* that unregisters the *v* and disables the use of *callback* through the custodian or a finalizer. The value *v* must be provided to *unreg* (otherwise it would be in *unreg*'s closure, possibly preventing the value from being finalized). The *available-callback* function is called in tail position, so its result is the result of `register-finalizer-and-custodian-shutdown`.

If *custodian* is already shut down, then *unavailable-callback* is applied in tail position to a function `reg-fnl` that registers a finalizer. By default, a finalizer is registered anyway, but usually a better choice is to report an error.

Added in version 6.1.1.6 of package `base`.
Changed in version 8.1.0.6: Added the `#:custodian-available` argument.

`(make-custodian-at-root)` → `custodian?`

Creates a custodian that is a child of the root custodian, bypassing the `current-custodian` setting.

Creating a child of the root custodian is useful for registering a shutdown function that will be triggered only when the current place terminates.

Added in version 6.9.0.5 of package `base`.

## 5.9 Atomic Execution

`(require ffi/unsafe/atomic)` package: `base`

*Atomic mode* evaluates a Racket expression without switching among Racket threads and with limited support for events. An atomic computation in this sense is *not* atomic with respect to other places, futures, or parallel threads, but only to coroutine threads within a place. Asynchronous break exceptions (in the sense of `break-thread`) are also disabled in atomic mode except as started by `start-breakable-atomic`.

Atomic mode is unsafe, because the Racket scheduler is not able to operate while execution is in atomic mode: (1) the scheduler cannot switch coroutine threads or poll certain kinds of events, which can lead to deadlock or starvation of other threads; and (2) calling a scheduler-related function in atomic mode has unspecified behavior, where misuse is not necessarily caught with a check. Functions that are directly scheduler-related include `thread`, `sleep`, `semaphore-wait`, `semaphore-post`, `channel-put`, `channel-get`, and `sync`. Beware

that other operations can involve such synchronization, such as writing to an output port or using an equal?-based hash table. Even if an output target is known to be free of synchronization, beware that values can have arbitrary printing procedures attached through prop:custom-write. Successful use of atomic mode requires a detailed knowledge of any implementation that might be reached during atomic mode to ensure that it terminates and does not involve synchronization.

Uninterruptible mode is related to atomic mode. It is also unsafe and practically the same as atomic mode in a coroutine thread, but uninterruptible mode does not force a parallel thread to synchronize with all coroutine threads. Uninterruptible mode also allows the use of *uncontested* semaphores and equal?-based hash tables.

```
(start-atomic) → void?
(end-atomic) → void?
```

Starts and ends atomic mode by disabling/re-enabling concurrency with any Racket coroutine threads in the same place, and also suspends/resumes delivery of break exceptions (independent of the result of break-enabled). Calls to start-atomic and end-atomic can be nested.

Note that pairing start-atomic and end-atomic with dynamic-wind is useful only when

- the current exception handler is known to safely escape atomic mode, or else all possible escapes are through known continuation jumps or aborts (because breaks are disabled and no other exceptions are possible) that escape safely; and
- exception constructions, if any, avoid printing values in the exception message, or else the error value conversion handler is always used and known to be safe for atomic mode.

Using call-as-atomic is somewhat safer than using start-atomic and end-atomic, because call-as-atomic catches exceptions and re-raises them after exiting atomic mode, and it wraps any call to the error value conversion handler with call-as-nonatomic. The latter is safe for a particular atomic region, however, only if the region can be safely interrupted by a non-atomic exception construction. See also call-as-nonatomic-retry-point.

Unlike call-as-atomic, start-atomic and end-atomic can be called from any OS thread as supported by ffi/unsafe/os-thread, although the calls have no effect outside of Racket threads. Using start-atomic in a future that is not running in a Racket thread blocks the future until it is resumed by touch in a Racket thread. Using start-atomic in a parallel thread synchronizes with all coroutine threads in the same place, but not other parallel threads or futures.

See also the caveat that atomic mode is unsafe.

```
(start-breakable-atomic) → void?
(end-breakable-atomic) → void?
```

Like start-atomic and end-atomic, but the delivery of break exceptions is not suspended.

These functions are not significantly faster than start-atomic and end-atomic, so they provide no benefit in a context where breaks are disabled.

```
(call-as-atomic thunk) → any
  thunk : (-> any)
```

Calls *thunk* in atomic mode, where call-as-nonatomic can be used during the dynamic extent of the call to revert to non-atomic mode for a nested computation.

When call-as-atomic is used in the dynamic extent of call-as-atomic, then *thunk* is called directly as a non-tail call.

If *thunk* raises an exception, the exception is caught and re-raised after exiting atomic mode. Any call to the current error value conversion handler is effectively wrapped with call-as-nonatomic.

See also the caveat that atomic mode is unsafe.

```
(call-as-nonatomic thunk) → any
  thunk : (-> any)
```

Within the dynamic extent of a call to call-as-atomic, calls *thunk* in non-atomic mode. Beware that the current thread may be suspended or terminated by other threads during the execution of *thunk*.

When used not in the dynamic extent of a call to call-as-atomic, call-as-nonatomic raises exn:fail:contract.

```
(in-atomic-mode?) → boolean?
```

Returns #t when in atomic mode or uninterruptible mode (within the current place), #f otherwise.

```
(start-uninterruptible) → void?
(end-uninterruptible) → void?
```

Like start-atomic and end-atomic, but the continuation after start-uninterruptible and before end-uninterruptible may run concurrently with other Racket threads (both coroutine threads and parallel threads), but in *uninterruptible*

*mode*: the continuation will reach `end-uninterruptible` without interruption from other threads. Uninterruptable mode is unsafe, just like atomic mode is unsafe.

Unlike `start-atomic`, `start-uninterruptible` in the CS implementation does not block a future that is running concurrently with Racket threads, and it does not cause a parallel thread to synchronize with coroutine threads. At the same time, such a future or parallel thread must not perform any action that blocks the future or requires synchronization with coroutine threads; successful use of uninterruptible mode in a future or parallel thread thus requires knowledge of Racket's internals.

As a special case, an `equal?`-based mutable hash table (created by `make-hash`, for example) or a semaphore (created by `make-semaphore`) can be used an in uninterruptible mode if it is used *only* in uninterruptible mode, never concurrently; in the case of a semaphore, the semaphore's internal counter must be non-zero when waiting on the semaphore. Direct use of a semaphore in uninterrupted mode would not make sense, but the special case allows uninterrupted mode to use data structures that normally rely on semaphores for locking, as long as they are used consistently that way. See also `make-uninterruptible-lock`.

Calls to `start-uninterruptible` and `end-uninterruptible` can be nested, and they can be mutually nested with calls to `start-atomic` and `end-atomic` in a coroutine thread. Since `start-atomic` is blocking for futures and requires synchronization in a parallel thread, it cannot be used in uninterruptible mode in a future or parallel thread.

Added in version 8.17.0.7 of package `base`.
Changed in version 8.18.0.2: Constrain the use of uninterruptible mode in futures and parallel threads.

```
(call-as-uninterruptible thunk) → any
  thunk : (-> any)
```

Similar to `call-as-atomic`, but for uninterruptible mode.

Added in version 8.17.0.7 of package `base`.

```
(make-uninterruptible-lock) → any/c
(uninterruptible-lock-acquire lock) → void?
  lock : any/c
(uninterruptible-lock-release lock) → void?
  lock : any/c
```

An *uninterruptible lock* provides low-level synchronization that cooperates with uninterruptible mode across parallel threads and futures. In particular, since a hash table or semaphore can be used in uninterruptible mode but non-concurrently, an uninterruptible lock can be used to guard access to the hash table or semaphore.

An uninterruptible lock does not cooperate with coroutine thread scheduling. The `uninterruptible-lock-acquire` function enters uninterruptible mode immediately, before taking the lock or even waiting for it, and `uninterruptible-lock-release` exits

uninterruptible mode only after releasing the lock. An uninterruptible lock is therefore useful only to guard predictably short actions that are reasonably considered atomic from the perspective of Racket threads.

Added in version 8.18.0.5 of package `base`.

## 5.10 Speculatively Atomic Execution

```
(require ffi/unsafe/try-atomic)
```
package: `base`

The `ffi/unsafe/try-atomic` library supports atomic execution that can be suspended and resumed in non-atomic mode if it takes too long or if some external event causes the attempt to be abandoned.

```
(call-as-nonatomic-retry-point thunk) → any
  thunk : (-> any)
```

Calls *thunk* in atomic mode (see `start-atomic` and `end-atomic`) while allowing *thunk* to use `try-atomic`. Any incomplete computations started with `try-atomic` are run non-atomically after *thunk* returns. The result of *thunk* is used as the result of `call-as-nonatomic-retry-point`.

```
(try-atomic  thunk
             default-val
            [#:should-give-up? give-up-proc
             #:keep-in-order? keep-in-order?]) → any
  thunk : (-> any)
  default-val : any/c
  give-up-proc : (-> any/c) = run-200-milliseconds
  keep-in-order? : any/c = #t
```

Within the dynamic extent of a `call-as-nonatomic-retry-point` call, attempts to run *thunk* in the existing atomic mode. The *give-up-proc* procedure is called periodically to determine whether atomic mode should be abandoned; the default *give-up-proc* returns true after 200 milliseconds. If atomic mode is abandoned, the computation is suspended, and *default-val* is returned, instead. The computation is resumed later by the enclosing `call-as-nonatomic-retry-point` call.

If *keep-in-order?* is true, then if `try-atomic` is called after an earlier computation was suspended for the same `call-as-nonatomic-retry-point` call, then *thunk* is immediately enqueued for completion by `call-as-nonatomic-retry-point` and *default-val* is returned.

The *give-up-proc* callback is polled only at points where the level of atomic-mode nesting (see `start-atomic`, `start-breakable-atomic`, and `call-as-atomic`) is the same as at the point of calling `try-atomic`.

If *thunk* aborts the current continuation using (default-continuation-prompt-tag), the abort is suspended the resumed by the enclosing call-as-nonatomic-retry-point. Escapes to the context of the call to *thunk* using any other prompt tag or continuation are blocked (using dynamic-wind) and simply return (void) from *thunk*.

## 5.11 Thread Scheduling

(require ffi/unsafe/schedule) package: base

The ffi/unsafe/schedule library provides functions for cooperating with the thread scheduler and manipulating it. The library's operations are unsafe because callbacks run in atomic mode and in an unspecified thread.

Running an operation in an unspecified thread means that the Racket scheduler for coroutine threads selects a convenient coroutine thread and switches the thread's evaluation temporarily to the operation. The thread will not have been in atomic mode, but the operation using the thread is always run in atomic mode. An operation that is run in an unspecified thread can use (current-thread) check parameter values, and inspect the continuation, but doing so is unsafe: the current thread and its continuation are not specified, and misusing a thread or it continuation can leak information or change the thread's behavior.

Added in version 6.11.0.1 of package base.

```
(unsafe-poller poll) → any/c
  poll : (evt? (or/c #f any/c) . -> . (values (or/c #f list?) evt?))
```

Produces a *poller* value that is allowed as a prop:evt value, even though it is not a procedure or itself an evt?. The *poll* callback is called in atomic mode in an unspecified thread to check whether the event is ready or to allow it to register a wakeup trigger.

The first argument to *poll* is always the object that is used as a synchronizable event with the poller as its prop:evt value. Let's call that value *evt*.

The second argument to *poll* is #f when *poll* is called to check whether the event is ready. The result must be two values. The first result value is a list of results if *evt* is ready, or it is #f if *evt* is not ready. The second result value is #f if *evt* is ready, or it is an event to replace *evt* (often just *evt* itself) if *evt* is not ready.

When the thread scheduler has determined that the Racket process should sleep until an external event or timeout, then *poll* is called with a non-#f second argument, *wakeups*. In that case, if the first result value is a list, then the sleep will be canceled, but the list is not recorded as the result (and *poll* most likely will be called again). In addition to returning a #f initial value, *poll* can call unsafe-poll-ctx-fd-wakeup, unsafe-poll-ctx-eventmask-wakeup, and/or unsafe-poll-ctx-milliseconds-wakeup on *wakeups* to register wakeup triggers.

```
(unsafe-poll-fd fd mode [socket?]) → boolean?
  fd : exact-integer?
  mode : '(read write)
  socket? : any/c = #t
```

Checks whether the given file descriptor or socket is currently ready for reading or writing, as selected by *mode*.

Added in version 7.2.0.6 of package base.

```
(unsafe-poll-ctx-fd-wakeup wakeups fd mode) → void?
  wakeups : any/c
  fd : fixnum?
  mode : '(read write error)
```

Registers a file descriptor (Unix and Mac OS) or socket (all platforms) to cause the Racket process to wake up and resume polling if the file descriptor or socket becomes ready for reading, writing, or error reporting, as selected by *mode*. The *wakeups* argument must be a non-#f value that is passed by the scheduler to a unsafe-poller-wrapped procedure.

```
(unsafe-poll-ctx-eventmask-wakeup wakeups
                                  mask)   → void?
  wakeups : any/c
  mask : fixnum?
```

On Windows, registers an eventmask to cause the Racket process to wake up and resume polling if an event selected by the mask becomes available.

```
(unsafe-poll-ctx-milliseconds-wakeup wakeups
                                     msecs)  → void?
  wakeups : any/c
  msecs : flonum?
```

Causes the Racket process to wake up and resume polling at the point when (current-inexact-monotonic-milliseconds) starts returning a value that is *msecs* or greater.

Changed in version 8.3.0.9 of package base: unsafe-poll-ctx-milliseconds-wakeup previously used current-inexact-milliseconds.

```
(unsafe-set-sleep-in-thread! foreground-sleep
                             fd)              → void?
  foreground-sleep : (-> any/c)
  fd : fixnum?
```

Registers *foreground-sleep* as a procedure to implement sleeping for the Racket process when the thread scheduler determines at the process will sleep. Meanwhile, during a call

to *foreground-sleep*, the scheduler's default sleeping function will run in a separate OS-level thread. When that default sleeping function wakes up, a byte is written to *fd* as a way of notifying *foreground-sleep* that it should return immediately.

This function works on when OS-level threads are available within the Racket implementation. It always works for Mac OS.

```
(unsafe-signal-received) → void?
```

For use with unsafe-set-sleep-in-thread! by *foreground-sleep* or something that it triggers, causes the default sleeping function to request *foreground-sleep* to return.

```
(unsafe-make-signal-received) → (-> void?)
```

Returns a function that is like unsafe-signal-received, but it can be called in any place or in any OS thread as supported by ffi/unsafe/os-thread to ensure a subsequent round of polling by the thread scheduler in the place where unsafe-make-signal-received was called.

Synchronizaiton between the result of unsafe-make-signal-received and the scheduler will ensure the equivalent of (memory-order-release) before the call to the function produced by unsafe-make-signal-received and the equivalent of (memory-order-acquire) before the scheduler's invocation of pollers.

Added in version 8.0.0.4 of package base.

## 5.12 Ports

(require ffi/unsafe/port) package: base

The ffi/unsafe/port library provides functions for working with ports, file descriptors, and sockets. The library's operations are unsafe, because no checking is performed on file descriptors and sockets, and misuse of file descriptors and sockets can break other objects.

Added in version 6.11.0.4 of package base.

```
(unsafe-file-descriptor->port fd name mode)
 → (or/c port? (values input-port? output-port?))
  fd : exact-integer?
  name : any/c
  mode : (listof (or/c 'read 'write 'text 'regular-file))
(unsafe-socket->port socket name mode) → input-port? output-port?
  socket : exact-integer?
  name : bytes?
  mode : (listof (or/c 'no-close))
```

Returns an input port and/or output port for the given file descriptor or socket. On Windows, a "file descriptor" corresponds to a file HANDLE, while a socket corresponds to a SOCKET. On Unix, a socket is a file descriptor, but using the socket-specific unsafe-socket->port may enable socket-specific functionality, such as address reporting via tcp-addresses.

The *name* argument determines the port's name as reported by object-name. The *name* must be a UTF-8 encoding that is converted to a symbol for the socket name.

For a file descriptor, the *mode* list must include at least one of 'read or 'write, and two ports are returned if *mode* includes both 'read and 'write. The 'text mode affects only Windows ports. The 'regular-file mode indicates that the file descriptor corresponds to a regular file (which has the property, for example, that reading never blocks). Closing all returned file-descriptor ports closes the file descriptor.

For a socket, the *mode* list can include 'no-close, in which case closing both of the returned ports does not close the socket.

For any kind of result port, closing the resulting ports readies and unregisters any semaphores for the file descriptor or socket that were previously created with unsafe-file-descriptor->semaphore or unsafe-socket->semaphore.

```
(unsafe-port->file-descriptor p) → (or/c exact-integer? #f)
  p : port?
(unsafe-port->socket p) → (or/c exact-integer? #f)
  p : port?
```

Returns a file descriptor (which is a HANDLE value on Windows) of a socket for port if it has one, #f otherwise.

On Unix and Mac OS, the result of unsafe-port->file-descriptor can be #f if it corresponds to a port that is waiting for its peer as reported by port-waiting-peer?, such as the write end of a fifo where no reader is connected. Wait until such is ready by using sync).

Changed in version 7.4.0.5 of package base: Accommodate a fifo write end blocked on a reader by returning #f.

```
(unsafe-file-descriptor->semaphore fd mode)
 → (or/c semaphore? #f)
  fd : exact-integer?
  mode : (or/c 'read 'write 'check-read 'check-write 'remove)
(unsafe-socket->semaphore socket mode) → (or/c semaphore? #f)
  socket : exact-integer?
  mode : (or/c 'read 'write 'check-read 'check-write 'remove)
```

Returns a semaphore that becomes ready when *fd* or *socket* is ready for reading or writing, as selected by *mode*. Specifically, these functions provide a one-shot, *edge-triggered* indicator; the semaphore is posted the *first time* any of the following cases holds:

- *fd* or *socket* is ready for reading or writing (depending on *mode*),
- ports were created from *fd* or *socket* using unsafe-file-descriptor->port or unsafe-socket->port, and those ports were closed, or
- a subsequent call occurred with the same *fd* or *socket* and with 'remove for *mode*.

The result is #f if a conversion to a semaphore is not supported for the current platform or for the given file descriptor or socket.

The 'check-read and 'check-write modes are like 'read and 'write, but the result if #f if a semaphore is not already generated for the specified file descriptor or socket in the specified mode.

The 'remove mode readies and unregisters any semaphores previously created for the given file descriptor or socket. Semaphores must be unregistered before the file descriptor or socket is closed. Beware that closing a port from unsafe-file-descriptor->port or unsafe-socket->port will also ready and unregister semaphores. In all of those cases, however, the semaphore is made ready asynchronously, so there may be a detectable delay.

```
(unsafe-fd->evt fd mode [socket?]) → (or/c evt? #f)
  fd : exact-integer?
  mode : (or/c 'read 'write 'check-read 'check-write 'remove)
  socket? : any/c = #t
```

Returns an event that is ready when *fd* is ready for reading or writing, as selected by *mode*. Specifically, it returns a multi-use, *level-triggered* indicator; the event is ready *whenever* any of the following cases holds:

- *fd* is ready for reading or writing (depending on *mode*),
- a subsequent call occurred with the same *fd* and with 'remove for *mode* (once removed, the event is perpetually ready).

The synchronization result of the event is the event itself.

The 'check-read and 'check-write modes are like 'read and 'write, but the result is #f if an event is not already generated for the specified file descriptor or socketin the specified mode.

The 'remove mode readies and unregisters any events previously created for the given file descriptor or socket. Events must be unregistered before the file descriptor or socket is closed. Unlike the semaphore result of unsafe-file-descriptor->semaphore and unsafe-socket->semaphore, the event result of unsafe-fd->evt is not triggered or unregistered by closing a port—not even a port from unsafe-file-descriptor->port or unsafe-socket->port.

Added in version 7.2.0.6 of package base.

## 5.13 Process-Wide and Place-Wide Registration

```
(require ffi/unsafe/global)
```
package: base

The ffi/unsafe/global library provides a utility registering information that is local to a place or spans all places in the Racket process.

Added in version 6.9.0.5 of package base.

```
(register-process-global key val) → cpointer?
  key : bytes?
  val : cpointer?
```

Gets or sets a value in a process-global table (i.e., shared across multiple places, if any).

If *val* is #f, the current mapping for *key* is reported.

If *val* is not #f, and no value has been installed for *key*, then the value is installed and #f is returned. If a value has already been installed, then no new value is installed and the old value is returned. The given *val* must not refer to garbage-collected memory.

This function is intended for infrequent use with a small number of keys.

```
(get-place-table) → hash?
```

Returns a place-specific, mutable, eq?-based hash table. The result is always the same for a particular place.

Added in version 6.11.0.6 of package base.

## 5.14 Operating System Threads

```
(require ffi/unsafe/os-thread)
```
package: base

The ffi/unsafe/os-thread library provides functions for running constrained Racket code in a separate thread at the operating-system level. Except for os-thread-enabled?, the functions of ffi/unsafe/os-thread are currently supported only when (system-type 'vm) returns 'chez-scheme, and even then only in certain build modes. The functions raise exn:fail:unsupported when not supported.

Added in version 6.90.0.9 of package base.

```
(os-thread-enabled?) → boolean?
```

Returns #t if the other functions of ffi/unsafe/os-thread work without raising exn:fail:unsupported, #f otherwise.

```
(call-in-os-thread thunk) → void?
  thunk : (-> any)
```

Runs *thunk* in a separate operating-system thread, which runs concurrently to all Racket threads.

The *thunk* is run in atomic mode, and it must not inspect its continuation or use any Racket thread functions (such as `thread` or `current-thread`), any Racket synchronization functions (such as `semaphore-post` or `sync`), or any parameters (such as `current-output-port`). Variables may be safely mutated with `set!`, and vectors, mutable pairs, boxes, mutable structure fields, and `eq?`- and `eqv?`-based hash tables can be mutated, but the visibility of mutations to other threads is unspecified except as synchronized through `os-semaphore-wait` and `os-semaphore-post`.

```
(make-os-semaphore) → any
```

Creates a semaphore that can be used with `os-semaphore-wait` and `os-semaphore-post` to synchronize an operating-system thread with Racket threads and other operating-system threads.

```
(os-semaphore-post sema) → void?
  sema : any/c
```

Analogous to `semaphore-post`, but posts to a semaphore created by `make-os-semaphore`.

```
(os-semaphore-wait sema) → void?
  sema : any/c
```

Analogous to `semaphore-wait`, but waits on a semaphore created by `make-os-semaphore`. Waiting blocks the current thread; if the current thread is a Racket coroutine threads, then waiting also blocks all other coroutine threads in the same place. Waiting from a parallel thread, does not necessarily block other Racket threads, but it consumes a processor resource for the thread's pool, and it may block coroutine threads or others if they attempt to synchronize with the blocked parallel thread.

### 5.14.1 Operating System Asynchronous Channels

```
(require ffi/unsafe/os-async-channel)
```
package: `base`

The `ffi/unsafe/os-async-channel` library provides an asynchronous channels that work with operating-system threads, where normal racket channels or place channels are not allowed. These channels are typically used in combination with `ffi/unsafe/os-thread`.

An asynchronous operating-system channel is a synchronizable event, so can it can be used with sync to receive a value in a Racket thread. Other threads must use os-async-channel-try-get or os-async-channel-get.

When a thread is blocked on an otherwise inaccessible asynchronous channel that was produced by make-os-async-channel, the thread is *not* available for garbage collection. That's different from a thread is blocked on a regular Racket channel or a place channel.

Added in version 8.0.0.4 of package base.

```
(make-os-async-channel) → os-async-channel?
```

Creates a new, empty asynchronous channel for use with operating-system threads.

```
(os-async-channel? v) → boolean?
  v : any/c
```

Returns #t if *v* is an asynchronous channel produced by make-os-async-channel, #f otherwise.

```
(os-async-channel-put ch v) → void?
  ch : os-async-channel?
  v : any/c
```

Enqueues *v* in the asynchronous channel *ch*. This function can be called from a Racket thread or any operating-system thread.

```
(os-async-channel-try-get ch [default-v]) → any/c
  ch : os-async-channel?
  default-v : any/c = #f
```

Dequeues a value from the the asynchronous channel *ch* and returns it, if a value is available. If no value is immediately available in the channel, *default-v* is returned. This function can be called from a Racket thread or any operating-system thread.

```
(os-async-channel-get ch) → any/c
  ch : os-async-channel?
```

Dequeues a value from the the asynchronous channel *ch* and returns it, blocking until a value is available. This function can be called from any non-Racket operating-system thread. This function should *not* be called from a Racket thread, since it blocks in a way that will block all Racket threads within a place; in a Racket thread, use sync, instead.

## 5.15 Garbage Collection Callbacks

```
(require ffi/unsafe/collect-callback)
```
package: base

The `ffi/unsafe/collect-callback` library provides functions to register constrained callbacks that are run just before and after a garbage collection.

Added in version 7.0.0.9 of package `base`.

```
(unsafe-add-collect-callbacks pre post) → any/c
  pre : (vectorof vector?)
  post : (vectorof vector?)
```

Registers descriptions of foreign functions to be called just before and just after a garbage collection. The foreign functions must not allocate garbage-collected memory, and they are called in a way that does not allocate, which is why *pre_desc* and *post_desc* are function descriptions instead of thunks.

A description is a vector of vectors, where each of the inner vectors describes a single call, and the calls are performed in sequence. Each call vector starts with a symbol that indicates the protocol of the foreign function to be called. The following protocols are supported:

The apparently arbitrary and whimsical set of supported protocols is enough to allow DrRacket to show a garbage-collection icon.

- `'int->void` corresponds to `void (*)(int)`.
- `'ptr_ptr_ptr->void` corresponds to `void (*)(void*, void*, void*)`.
- `'ptr_ptr->save` corresponds to `void* (*)(void*, void*)`, but the result is recorded as the current "save" value. The current "save" value starts as `NULL`.
- `'save!_ptr->void` corresponds to `void (*)(void*, void*)`, but only if the current "save" value is not a `NULL` pointer, and passing that pointer as the function's first argument (so that only one additional argument is us from the description vector).
- `'ptr_ptr_ptr_int->void` corresponds to `void (*)(void*, void*, void*, int)`.
- `'ptr_ptr_float->void` corresponds to `void (*)(void*, void*, float)`.
- `'ptr_ptr_double->void` corresponds to `void (*)(void*, void*, double)`.
- `'ptr_ptr_ptr_int_int_int_int_int_int_int_int_int->void` corresponds to `void (*)(void*, void*, void*, int, int, int, int, int, int, int, int, int)`.
- `'osapi_ptr_int->void` corresponds to `void (*)(void*, int)`, but using the stdcall calling convention on Windows.
- `'osapi_ptr_ptr->void` corresponds to `void (*)(void*, void*)`, but using the stdcall calling convention on Windows.
- `'osapi_ptr_int_int_int_int_ptr_int_int_long->void` corresponds to `void (*)(void*, int, int, int, int, void*, int, int, long)`, but using the stdcall calling convention on Windows.

After the protocol symbol, the vector should contain a pointer to a foreign function and then an element for each of the function's arguments. Pointer values are represented as for the `_pointer` representation defined by `ffi/unsafe`.

The result is a key for use with `unsafe-remove-collect-callbacks`. If the key becomes inaccessible, then the callback will be removed automatically (but beware that the pre-callback will have executed and the post-callback will not have executed)

```
(unsafe-remove-collect-callbacks key) → void?
  key : any/c
```

Unregisters pre- and post-collection callbacks that were previously registered by a call to `unsafe-add-collect-callbacks` that returned `v`.

## 5.16 Objective-C FFI

```
(require ffi/unsafe/objc)
```
package: `base`

The `ffi/unsafe/objc` library builds on `ffi/unsafe` to support interaction with Objective-C.

The library supports Objective-C interaction in two layers. The upper layer provides syntactic forms for sending messages and deriving subclasses. The lower layer is a thin wrapper on the Objective-C runtime library functions. Even the upper layer is unsafe and relatively low-level compared to normal Racket libraries, because argument and result types must be declared in terms of FFI C types (see §3.1 "Type Constructors").

### 5.16.1 FFI Types and Constants

```
_id : ctype?
```

The type of an Objective-C object, an opaque pointer.

```
_Class : ctype?
```

The type of an Objective-C class, which is also an `_id`.

```
_Protocol : ctype?
```

The type of an Objective-C protocol, which is also an `_id`.

```
_SEL : ctype?
```

The type of an Objective-C selector, an opaque pointer.

```
_BOOL : ctype?
```

The Objective-C boolean type. Racket values are converted for C in the usual way: `#f` is false and any other value is true. C values are converted to Racket booleans.

```
YES : boolean?
```

Synonym for `#t`

```
NO : boolean?
```

Synonym for `#f`

### 5.16.2 Syntactic Forms and Procedures

```
(tell result-type obj-expr method-id)
(tell result-type obj-expr arg ...)

result-type =
            | #:type ctype-expr

        arg = method-id arg-expr
            | method-id #:type ctype-expr arg-expr
```

Sends a message to the Objective-C object produced by *obj-expr*. When a type is omitted for either the result or an argument, the type is assumed to be `_id`, otherwise it must be specified as an FFI C type (see §3.1 “Type Constructors”).

If a single *method-id* is provided with no arguments, then *method-id* must not end with `:`; otherwise, each *method-id* must end with `:`.

Examples:

```
> (tell NSString alloc)
#<cpointer:id>
> (tell (tell NSString alloc)
        initWithUTF8String: #:type _string "Hello")
#<cpointer:id>
```

```
(tellv obj-expr method-id)
(tellv obj-expr arg ...)
```

Like `tell`, but with a result type `_void`.

```
(import-class class-id ...)
```

Defines each *class-id* to the class (a value with FFI type `_Class`) that is registered with the string form of *class-id*. The registered class is obtained via `objc_lookUpClass`.

Example:

```
> (import-class NSString)
```

A class accessed by `import-class` is normally declared as a side effect of loading a foreign library. For example, if you want to import the class `NSString` on Mac OS, the `"Foundation"` framework must be loaded, first. Beware that if you use `import-class` in DrRacket or a module that `requires` `racket/gui/base`, then `"Foundation"` will have been loaded into the Racket process already. To avoid relying on other libraries to load `"Foundation"`, explicitly load it with `ffi-lib`:

```
> (ffi-lib
   "/System/Library/Frameworks/Foundation.framework/Foundation")
> (import-class NSString)
```

```
(import-protocol protocol-id ...)
```

Defines each *protocol-id* to the protocol (a value with FFI type `_Protocol`) that is registered with the string form of *protocol-id*. The registered class is obtained via `objc_getProtocol`.

Example:

```
> (import-protocol NSCoding)
```

```
(define-objc-class class-id superclass-expr
  maybe-mixins
  maybe-protocols
  [field-id ...]
  method ...)
```

```
   maybe-mixins =
                | #:mixins (mixin-expr ...)

maybe-protocols =
                | #:protocols (protocol-expr ...)

         method = (mode maybe-async result-ctype-expr (method-id) body ...+)
                | (mode maybe-async result-ctype-expr (arg ...+) body ...+)

           mode = +
                | -
                | +a
                | -a

    maybe-async =
                | #:async-apply async-apply-expr

            arg = method-id [ctype-expr arg-id]
```

Defines *class-id* as a new, registered Objective-C class (of FFI type `_Class`). The *superclass-expr* should produce an Objective-C class or `#f` for the superclass. An optional `#:mixins` clause can specify mixins defined with `define-objc-mixin`. An optional `#:protocols` clause can specify Objective-C protocols to be implemented by the class, where a `#f` result for a *protocol-expr* is ignored.

Each *field-id* is an instance field that holds a Racket value and that is initialized to `#f` when the object is allocated. The *field-id*s can be referenced and `set!` directly when the method *body*s. Outside the object, they can be referenced and set with `get-ivar` and `set-ivar!`.

Each *method* adds or overrides a method to the class (when *mode* is `-` or `-a`) to be called on instances, or it adds a method to the meta-class (when *mode* is `+` or `+a`) to be called on the class itself. All result and argument types must be declared using FFI C types (see §3.1 "Type Constructors"). When *mode* is `+a` or `-a`, the method is called in atomic mode (see `_cprocedure`). An optional `#:async-apply` specification determines how the method works when called from a foreign thread in the same way as for `_cprocedure`.

If a *method* is declared with a single *method-id* and no arguments, then *method-id* must not end with `:`. Otherwise, each *method-id* must end with `:`.

If the special method `dealloc` is declared for mode `-`, it must not call the superclass method, because a `(super-tell dealloc)` is added to the end of the method automatically. In addition, before `(super-tell dealloc)`, space for each *field-id* within the instance is deallocated.

Example:

```
> (define-objc-class MyView NSView
    [bm] ; <- one field
    (- _racket (swapBitwmap: [_racket new-bm])
       (begin0 bm (set! bm new-bm)))
    (- _void (drawRect: [_NSRect exposed-rect])
       (super-tell drawRect: exposed-rect)
       (draw-bitmap-region bm exposed-rect))
    (- _void (dealloc)
       (when bm (done-with-bm bm))))
```

Changed in version 6.90.0.26 of package `base`: Changed `#:protocols` handling to ignore `#f` expression results.

```
(define-objc-mixin (class-id superclass-id)
  maybe-mixins
  maybe-protocols
  [field-id ...]
  method ...)
```

Like `define-objc-class`, but defines a mixin to be combined with other method definitions through either `define-objc-class` or `define-objc-mixin`. The specified *field-id*s are not added by the mixin, but must be a subset of the *field-id*s declared for the class to which the methods are added.

```
self
```

When used within the body of a `define-objc-class` or `define-objc-mixin` method, refers to the object whose method was called. This form cannot be used outside of a `define-objc-class` or `define-objc-mixin` method.

```
(super-tell result-type method-id)
(super-tell result-type arg ...)
```

When used within the body of a `define-objc-class` or `define-objc-mixin` method, calls a superclass method. The *result-type* and *arg* sub-forms have the same syntax as in `tell`. This form cannot be used outside of a `define-objc-class` or `define-objc-mixin` method.

```
(get-ivar obj-expr field-id)
```

Extracts the Racket value of a field in a class created with `define-objc-class`.

```
(set-ivar! obj-expr field-id value-expr)
```

Sets the Racket value of a field in a class created with `define-objc-class`.

```
(selector method-id)
```

Returns a selector (of FFI type _SEL) for the string form of *method-id*.

Example:

```
> (tellv button setAction: #:type _SEL (selector terminate:))
```

```
(objc-is-a? obj cls) → boolean?
  obj : _id
  cls : _Class
```

Check whether *obj* is an instance of the Objective-C class *cls* or a subclass.

Changed in version 6.1.0.5 of package base: Recognize subclasses, instead of requiring an exact class match.

```
(objc-subclass? subcls cls) → boolean?
  subcls : _Class
  cls : _Class
```

Check whether *subcls* is *cls* or a subclass.

Added in version 6.1.0.5 of package base.

```
(objc-get-class obj) → _Class
  obj : _id
```

Extract the class of *obj*.

Added in version 6.3 of package base.

```
(objc-set-class! obj cls) → void?
  obj : _id
  cls : _Class
```

Changes the class of *obj* to *cls*. The object's existing representation must be compatible with the new class.

Added in version 6.3 of package base.

```
(objc-get-superclass cls) → _Class
  cls : _Class
```

Returns the superclass of *cls*.

Added in version 6.3 of package base.

```
(objc-class-has-instance-method? cls sel) → boolean?
  cls : _Class
  sel : _SEL
```

Reports report *cls* offers an instance method *sel*.

Added in version 9.0.0.6 of package base.

```
(objc-dispose-class cls) → void?
  cls : _Class
```

Destroys *cls*, which must have no existing instances or subclasses.

Added in version 6.3 of package base.

```
(objc-block function-type? proc #:keep keep) → cpointer?
  function-type? : ctype
  proc : procedure?
  keep : (box/c list?)
```

Wraps a Racket function *proc* as an Objective-C block. The procedure must accept an initial pointer argument that is the "self" argument for the block, and that extra argument must be included in the given function-type.

Extra records that are allocated to implement the block are added to the list in *keep*, which might also be included in function-type through a #:keep option to _fun. The pointers registered in *keep* must be retained as long as the block remains in use.

Added in version 6.3 of package base.

```
(objc-block-function-pointer block) → fpointer?
  block : cpointer?
```

Extracts the function pointer of an Objective-C block. Cast this function pointer to a suitable function type to call it, where the block itself must be passed as the first argument to the function.

Added in version 8.13.0.1 of package base.

```
(with-blocking-tell form ...+)
```

Causes any tell, tellv, or super-tell expression syntactically within the *form*s to be blocking in the sense of the #:blocking? argument to _cprocedure. Otherwise, (with-blocking-tell *form* ...+) is equivalent to (let () *form* ...+).

Added in version 7.0.0.19 of package base.

### 5.16.3 Raw Runtime Functions

```
(objc_lookUpClass s) → (or/c _Class #f)
  s : string?
```

Finds a registered class by name.

```
(objc_getProtocol s) → (or/c _Protocol #f)
  s : string?
```

Finds a registered protocol by name.

```
(sel_registerName s) → _SEL
  s : string?
```

Interns a selector given its name in string form.

```
(objc_allocateClassPair cls s extra) → _Class
  cls : _Class
  s : string?
  extra : integer?
```

Allocates a new Objective-C class.

```
(objc_registerClassPair cls) → void?
  cls : _Class
```

Registers an Objective-C class.

```
(object_getClass obj) → _Class
  obj : _id
```

Returns the class of an object (or the meta-class of a class).

```
(class_getSuperclass cls) → _Class
  cls : _Class
```

Returns the superclass of *cls* or #f if *cls* has no superclass.

Added in version 6.1.0.5 of package base.

```
(class_addMethod cls
                 sel
                 imp
                 type
                 type-encoding) → boolean?
```

```
  cls : _Class
  sel : _SEL
  imp : procedure?
  type : ctype?
  type-encoding : string?
```

Adds a method to a class. The *type* argument must be a FFI C type (see §3.1 “Type Constructors”) that matches both *imp* and the not Objective-C type string *type-encoding*.

```
(class_addIvar cls
               name
               size
               log-alignment
               type-encoding) → boolean?
  cls : _Class
  name : string?
  size : exact-nonnegative-integer?
  log-alignment : exact-nonnegative-integer?
  type-encoding : string?
```

Adds an instance variable to an Objective-C class.

```
(object_getInstanceVariable obj name) → _Ivar any/c
  obj : _id
  name : string?
```

Gets the value of an instance variable whose type is `_pointer`.

```
(object_setInstanceVariable obj name val) → _Ivar
  obj : _id
  name : string?
  val : any/c
```

Sets the value of an instance variable whose type is `_pointer`.

```
_Ivar : ctype?
```

The type of an Objective-C instance variable, an opaque pointer.

```
((objc_msgSend/typed types) obj sel arg) → any/c
  types : (vector/c result-ctype arg-ctype ...)
  obj : _id
  sel : _SEL
  arg : any/c
```

Calls the Objective-C method on `_id` named by *sel*. The *types* vector must contain one more than the number of supplied *arg*s; the first FFI C type in `type` is used as the result type.

```
((objc_msgSendSuper/typed types)
  super
  sel
  arg)                              → any/c
  types : (vector/c result-ctype arg-ctype ...)
  super : _objc_super
  sel : _SEL
  arg : any/c
```

Like `objc_msgSend/typed`, but for a super call.

```
(make-objc_super id super) → _objc_super
  id : _id
  super : _Class
_objc_super : ctype?
```

Constructor and FFI C type use for super calls.

```
((objc_msgSend/typed/blocking types)
  obj
  sel
  arg)                                  → any/c
  types : (vector/c result-ctype arg-ctype ...)
  obj : _id
  sel : _SEL
  arg : any/c
((objc_msgSendSuper/typed/blocking types)
  super
  sel
  arg)                                      → any/c
  types : (vector/c result-ctype arg-ctype ...)
  super : _objc_super
  sel : _SEL
  arg : any/c
```

The same as `objc_msgSend/typed` and `objc_msgSendSuper/typed`, but specifying that the send should be blocking in the sense of the `#:blocking?` argument to `_cprocedure`.

Added in version 7.0.0.19 of package `base`.

### 5.16.4 Legacy Library

```
(require ffi/objc)
```
package: base

The ffi/objc library is a deprecated entry point to ffi/unsafe/objc. It exports only safe operations directly, and unsafe operations are imported using objc-unsafe!, analogous to scheme/foreign.

```
(objc-unsafe!)
```

Makes unsafe bindings of ffi/unsafe/objc available in the importing module.

## 5.17 Cocoa Foundation

The ffi/unsafe/nsalloc and ffi/unsafe/nsstring libraries provide basic facilities for working with Cocoa and/or Mac OS Foundation libraries (usually along with ffi/objc).

### 5.17.1 Strings

```
(require ffi/unsafe/nsstring)
```
package: base

```
_NSString : ctype?
```

A type that converts between Racket strings and NSString* (a.k.a. CFStringRef) values. That is, use _NSString as a type for a foreign-function NSString* argument or result.

The _NSString conversion keeps a weak mapping from Racket strings to converted strings, so that converting the same string (in the equal? sense) multiple times may avoid allocating multiple NSString objects.

### 5.17.2 Allocation Pools

```
(require ffi/unsafe/nsalloc)
```
package: base

Calling any Foundation API that allocates requires an NSAutoreleasePool installed. The ffi/unsafe/nsalloc library provides a function and shorthand syntactic form for setting up such a context. (The _NSString type creates an autorelease pool implicitly while converting from/to a Racket string, however.)

```
(call-with-autorelease thunk) → any
  thunk : (-> any)
```

Calls *thunk* in atomic mode and with a fresh `NSAutoreleasePool` that is `released` after *thunk* returns.

```
(with-autorelease expr)
```

A shorthand for `(call-with-autorelease (lambda () expr))`.

## 5.18 COM (Common Object Model)

The `ffi/com` and `ffi/unsafe/com` libraries support COM interaction in two layers. The safe upper layer provides functions for creating COM objects and dynamically constructing method calls based on COM automation (i.e., reflective information provided by the object). The unsafe lower layer provides a syntactic form and functions for working more directly with COM objects and interfaces.

A *COM object* instantiates a particular *COM class*. A COM class can be specified in either of two ways:

- A *CLSID* (class id), which is represented as a GUID. A *GUID* (globally unique identifier) is a 16-byte structure. GUIDs are typically written in string forms such as `"{A3B0AF9E-2AB0-11D4-B6D2-0060089002FE}"`. The `string->guid` and `guid->string` convert between string and GUID forms. The `string->clsid` function is the same as `string->guid`, but its use suggests that the resulting GUID is to be used as a CLSID.
- A *ProgID* is a human-readable name, such as `"MzCom.MzObj.5.2.0.7"`, which includes a version number. The version number can be omitted in a ProgID, in which case the most recent available version is used. The operating system provides a mapping between ProgIDs and CLSIDs that is available via `progid->clsid` and `clsid->progid`.

A COM object can be instantiated either on the local machine or on a remote machine. The latter relies on the operating system's *DCOM* (distributed COM) support.

Each COM object supports some number of *COM interfaces*. A COM interface has a programmatic name, such as `IDispatch`, that corresponds to a C-layer protocol. Each interface also has an *IID* (interface id) that is represented as a GUID such as `"{00020400-0000-0000-C000-000000000046}"`. Direct calls to COM methods require extracting a suitable interface pointer from an object using `QueryInterface` and the desired IID; the result is effectively cast it to a pointer to a dispatch-table pointer, where the dispatch table has a statically known size and foreign-function content. The `define-com-interface` form simplifies description and use of interface pointers. The COM automation layer uses a fixed number of reflection interfaces internally, notably `IDispatch`, to call methods by name and with safe argument marshaling.

### 5.18.1 COM Automation

```
(require ffi/com)
```
package: base

The ffi/com library builds on COM automation to provide a safe use of COM objects that support the IDispatch interface.

The ffi/com library is based on the *MysterX* library by Paul Steckler. MysterX is included with Racket but deprecated, and it will be replaced in the next version with a partial compatibility library that redirects to this one.

#### GUIDs, CLSIDs, IIDs, and ProgIDs

```
(guid? v) → boolean?
  v : any/c
(clsid? v) → boolean?
  v : any/c
(iid? v) → boolean?
  v : any/c
```

Returns #t if *v* is a structure representing a GUID, #f otherwise. The clsid? and iid? functions are the same as guid?.

A GUID corresponds an a _GUID structure at the unsafe layer.

```
(string->guid str) → guid?
  str : string?
(string->clsid str) → clsid?
  str : string?
(string->iid str) → iid?
  str : string?
```

Converts a string of the form "{00000000-0000-0000-0000-0000000000}", where each 0 can be a hexadecimal digit, to a GUID. If *str* does not have te expected form, the exn:fail exception is raised.

The string->clsid and string->iid functions are the same as string->guid.

```
(guid->string g) → string?
  g : guid?
```

Converts a GUID to its string form.

```
(guid=? g1 g2) → boolean?
  g1 : guid?
  g2 : guid?
```

Determines whether *g1* and *g2* represent the same GUID.

```
(progid->clsid progid) → clsid?
  progid : string?
(clsid->progid clsid) → (or/c string? #f)
  clsid : clsid?
```

Converts a ProgID to a CLSID or vice versa. Not evey COM class has a ProgID, so the result of `clsid->progid` can be `#f`.

The `progid->clsid` function accepts a versionless ProgID, in which case it produces the CLSID of the most recent available version. The `clsid->progid` function always produces a ProgID with its version.

### COM Objects

```
(com-object? obj) → boolean?
  obj : com-object?
```

Returns `#t` if the argument represents a COM object, `#f` otherwise.

```
(com-create-instance clsid-or-progid [where]) → com-object?
  clsid-or-progid : (or/c clsid? string?)
  where : (or/c 'local 'remote string?) = 'local
```

Returns an instance of the COM class specified by *clsid-or-progid*, which is either a CLSID or a ProgID.

The optional *where* argument indicates a location for running the instance, and may be `'local`, `'remote`, or a string indicating a machine name. See §5.18.1.8 “Remote COM servers (DCOM)” for more information.

An object can be created this way for any COM class, but functions such as `com-invoke` work only if the object supports the `IDispatch` COM automation interface.

The resulting object is registered with the current custodian, which retains a reference to the object until it is released with `com-release` or the custodian is shut down.

```
(com-release obj) → void?
  obj : com-object?
```

Releases the given COM object. The given *obj* is subsequently unusable, and the underlying COM object is destroyed unless its reference count has been incremented (via COM methods or unsafe operations).

If *obj* has already been released, `com-release` has no effect.

```
(com-get-active-object clsid-or-progid) → com-object?
  clsid-or-progid : (or/c clsid? string?)
```

Like com-create-instance, but gets an existing active object (always local) instead of creating a new one.

```
(com-object-clsid obj) → clsid?
  obj : com-object?
```

Returns the "CLSID" of the COM class instantiated by *obj*, or raises an error if the COM class is not known.

```
(com-object-set-clsid! obj clsid) → void?
  obj : com-object?
  clsid : clsid?
```

Sets the COM CLSID for *obj* to *clsid*. This is useful when COM event-handling procedures can obtain only ambiguous information about the object's COM class.

```
(com-object-eq? obj1 obj2) → boolean?
  obj1 : com-object?
  obj2 : com-object?
```

Returns #t if *obj1* and *obj2* refer to the same COM object, #f otherwise.

If two references to a COM object are the same according to com-object-eq?, then they are also the same according to equal?. Two com-object-eq? references are not necessarily eq?, however.

```
(com-type? v) → boolean?
  v : any/c
```

Returns #t if *v* represents reflective information about a COM object's type, #f otherwise.

```
(com-object-type obj) → com-type?
  obj : com-object?
```

Returns a representation of a COM object's type that is independent of the object itself.

```
(com-type=? t1 t2) → boolean?
  t1 : com-type?
  t2 : com-type?
```

Returns #t if *t1* and *t2* represent the same type information, #f otherwise.

**COM Methods**

```
(com-methods obj/type) → (listof string?)
  obj/type : (or/c com-object? com-type?)
```

Returns a list of strings indicating the names of methods on *obj/type*.

```
(com-method-type obj/type method-name)
 → (list/c '-> (listof type-description?)
               type-description?)
  obj/type : (or/c com-object? com-type?)
  method-name : string?
```

Returns a list indicating the type of the specified method in *obj/type*. The list after the '-> represents the argument types, and the final value represents the result type. See §5.18.1.9 "COM Types" for more information.

```
(com-invoke obj method-name v ...) → any/c
  obj : com-object?
  method-name : string?
  v : any/c
```

Invokes *method-name* on *obj* with *v*s as the arguments. The special value com-omit may be used for optional arguments, which useful when values are supplied for arguments after the omitted argument(s).

The types of arguments are determined via com-method-type, if possible, and type-describe wrappers in the *v*s are simply replaced with the values that they wrap. If the types are not available from com-method-type, then types are inferred for each *v* with attention to descriptions in any type-describe wrappers in *v*.

```
com-omit : any/c
```

A constant for use with com-invoke in place of an optional argument.

```
(com-omit? v) → boolean?
  v : any/c
```

Returns #t if *v* is com-omit, #f otherwise.

Added in version 6.3.0.3 of package base.

### COM Properties

```
(com-get-properties obj/type) → (listof string?)
  obj/type : (or/c com-object? com-type?)
```

Returns a list of strings indicating the names of readable properties in *obj/type*.

```
(com-get-property-type obj/type
                       property-name)
```

```
→ (list/c '-> '() type-description?)
  obj/type : (or/c com-object? com-type?)
  property-name : string?
```

Returns a type for *property-name* like a result of com-method, where the result type corresponds to the property value type. See §5.18.1.9 “COM Types” for information on the symbols.

```
(com-get-property obj property ...+) → any/c
  obj : com-object?
  property : (or/c string?
                   (cons/c string? list?))
```

Returns the value of the final property by following the indicated path of *property*s, where each intermediate property must be a COM object.

Each *property* is either a property-name string or a list that starts with a property-name string and continues with arguments for a parameterized property.

```
(com-get-property* obj property v ...) → any/c
  obj : com-object?
  property : string?
  v : any/c
```

Returns the value of a parameterized property, which behaves like a method and accepts the *v*s as arguments (like com-invoke). When no *v*s are provided, com-get-property* is the same as com-get-property.

```
(com-set-properties obj/type) → (listof string?)
  obj/type : (or/c com-object? com-type?)
```

Returns a list of strings indicating the names of writeable properties in *obj/type*.

```
(com-set-property-type obj/type
                       property-name)
 → (list/c '-> (list/c type-description?) 'void)
  obj/type : (or/c com-object? com-type?)
  property-name : string?
```

Returns a type for *property-name* like a result of com-method, where the sole argument type corresponds to the property value type. See §5.18.1.9 “COM Types” for information on the symbols.

```
(com-set-property! obj property ...+ v) → void?
  obj : com-object?
```

```
  property : (or/c string?
                   (cons/c string? list?))
  v : any/c
```

Sets the value of the final property in *obj* to *v* by following the *property*s, where the value of each intermediate property must be a COM object. A *property* can be a list instead of a string to represent a parameterized property and its arguments.

The type of the property is determined via com-property-type, if possible, and type-describe wrappers in *v* are then replaced with the values that they wrap. If the type is not available from com-property-type, then a type is inferred for *v* with attention to the descriptions in any type-describe wrappers in *v*.

**COM Events**

```
(com-events obj/type) → (listof string?)
  obj/type : (or/c com-object? com-type?)
```

Returns a list of strings indicating the names of events on *obj/type*.

```
(com-event-type obj/type event-name)
 → (list/c '-> (listof type-description?) 'void)
  obj/type : (or/c com-object? com-type?)
  event-name : string?
```

Returns a list indicating the type of the specified events in *obj/type*. The list after the '-> represents the argument types. See §5.18.1.9 “COM Types” for more information.

```
(com-event-executor? v) → boolean?
  v : any/c
```

Returns #t if *v* is a *COM event executor*, which queues event callbacks. A COM event executor *com-ev-ex* is a synchronizable event in the sense of sync, and (sync *com-ev-ex*) returns a thunk for a ready callback.

```
(com-make-event-executor) → com-event-executor?
```

Creates a fresh COM event executor for use with com-register-event-callback.

```
(com-register-event-callback obj
                             name
                             proc
                             com-ev-ex) → void?
  obj : com-object?
  name : string?
  proc : procedure?
  com-ev-ex : com-event-executor?
```

Registers a callback for the event named by *name* in *obj*. When the event fires, an invocation of *proc* to event arguments (which depends on *obj* and *name*) is queued in *com-ev-ex*. Synchronizing on *com-ev-ex* produces a thunk that applies *proc* to the event arguments and returns the result.

Only one callback can be registered for each *obj* and *name* combination.

Registration of event callbacks relies on prior registration of the COM class implemented by `"myssink.dll"` as distributed with Racket. (The DLL is the same for all Racket versions.)

```
(com-unregister-event-callback obj name) → void?
  obj : com-object?
  name : string?
```

Removes any existing callback for *name* in *obj*.

### COM Enumerations

```
(com-enumerate-to-list obj) → list?
  obj : com-object?
```

Produces the elements that *obj* would generate as the driver of a for-each loop in Visual Basic or PowerShell.

A call `(com-enumerate-to-list obj)` is equivalent to `(com-enumeration-to-list (com-get-property obj "_NewEnum"))`.

Added in version 6.2 of package `base`.

```
(com-enumeration-to-list obj) → list?
  obj : com-object?
```

Given a COM object that implements `IEnumVARIANT`, extracts the enumerated values into a list.

Added in version 6.2 of package `base`.

### Interface Pointers

```
(com-object-get-iunknown obj) → com-iunkown?
  obj : com-object?
(com-object-get-idispatch obj) → com-idispatch?
  obj : com-object?
```

Extracts an `IUnknown` or `IDispatch` pointer from *obj*. The former succeeds for any COM object that has not been released via `com-release`. The latter succeeds only when the COM object supports `IDispatch`, otherwise `exn:fail` is raised.

```
(com-iunknown? v) → boolean?
  v : any/c
```

Returns `#t` if *v* corresponds to an unsafe `_IUnknown-pointer`, `#f` otherwise. Every COM interface extends `IUnknown`, so `com-iunknown?` returns `#t` for every interface pointers.

```
(com-idispatch? v) → boolean?
  v : any/c
```

Returns `#t` if *v* corresponds to an unsafe `IDispatch`, `#f` otherwise.

**Remote COM servers (DCOM)**

The optional *where* argument to `com-create-instance` can be `'remote`. In that case, the server instance is run at the location given by the Registry key

HKEY_CLASSES_ROOT\AppID\⟨*CLSID*⟩\RemoteServerName

where ⟨*CLSID*⟩ is the CLSID of the application. This key may be set using the `dcomcnfg` utility. From `dcomcnfg`, pick the application to be run on the Applications tab, then click on the Properties button. On the Location tab, choose Run application on the following computer, and enter the machine name.

To run a COM remote server, the registry on the client machine must contain an entry at

HKEY_CLASSES_ROOT\CLSID\⟨*CLSID*⟩

where ⟨*CLSID*⟩ is the CLSID for the server. The server application itself need not be installed on the client machine.

There are a number of configuration issues relating to DCOM. See

http://www.distribucon.com/dcom95.html

for more information on how to setup client and server machines for DCOM.

**COM Types**

In the result of a function like `com-method-type`, symbols are used to represent various atomic types:

- `'int` — a 32-bit signed integer

- `'unsigned-int` — a 32-bit unsigned integer
- `'short-int` — a 16-bit signed integer
- `'unsigned-short` — a 16-bit unsigned integer
- `'signed-char` — an 8-bit signed integer
- `'char` — an 8-bit unsigned integer
- `'long-long` — a 64-bit signed integer
- `'unsigned-long-long` — a 64-bit unsigned integer
- `'float` — a 32-bit floating-point number
- `'double` — a 64-bit floating-point number
- `'currency` — an exact number that, when multiplied by 10,000, is a 64-bit signed integer
- `'boolean` — a boolean
- `'string` — a string
- `'date` — a `date` or `date*`; when converting to a `date*`, the timezone is reported as `"UTC"` and the `year-day` field is `0`
- `'com-object` — a COM object as in `com-object?`
- `'iunknown` — like `'com-object`, but also accepts an `IUnknown` pointer as in `com-iunknown?`
- `'com-enumeration` — a 32-bit signed integer
- `'any` — any of the above, or an array when not nested in an array type
- `'...` — treated like `'any`, but when it appears at the end of the sequence of types for arguments, allows the preceding type 0 or more times
- `'void` — no value

A type symbol wrapped in a list with `'box`, such as `'(box int)`, is a call-by-reference argument. A box supplied for the argument is updated with a new value when the method returns.

A type wrapped in a list with `'opt`, such as `'(opt (box int))`, is an optional argument. The argument can be omitted or replaced with `com-omit`.

A type wrapped in a list with `'array` and a positive exact integer, such as `'(array 7 int)`, represents a vector of values to be used as a COM array. A `'?` can be used in place of the

length integer to support a vector of any length. Array types with non-'? lengths can be nested to specify a multidimensional array as represented by nested vectors.

A type wrapped in a list with 'variant, such as '(variant (array 7 int)), is the same as the wrapped type, but a 'variant wrapper within an 'array type prevents construction of another array dimension. For example, '(array 2 (array 3 int)) is a two-dimensional array of integers, but '(array 2 (variant (array 3 int))) is a one-dimensional array whose elements are one-dimensional arrays of integers.

When type information is not available, functions like com-invoke infer type descriptions from arguments. Inference chooses 'boolean for booleans; the first of 'int, 'unsigned-int, 'long-long, 'unsigned-long-long that fits for an exact integer; 'double for inexact real numbers; 'string for a string; 'com-object and 'iunknown for corresponding COM object references; and an 'array type for a vector, where the element type is inferred from vector values, resorting to 'any if any two elements have different inferred types.

```
(type-description? v) → boolean?
  v : any/c
```

Return #t if v is a COM argument or result type description as above, #f otherwise.

```
(type-described? v) → boolean?
  v : any/c
(type-describe v desc) → type-described?
  v : any/c
  desc : type-description?
(type-described-value td) → any/c
  td : type-described?
(type-described-description td) → type-description?
  td : type-described?
```

The type-described? predicate recognizes wrappers produced with type-describe, and type-described-value and type-described-description extract the value and description parts of a type-describe value.

A type-describe wrapper combines a base value with a type description. The description is used instead of an automatically inferred COM argument type when no type is available for from COM automation a method for com-invoke or a property for com-set-property!. A wrapper can be placed on an immediate value, or it can be on a value within a box or vector.

### Class Display Names

```
(require ffi/com-registry)
```
package: base

The ffi/com-registry library provides a mapping from coclass names to CLSIDs for compatibility with the older MysterX interface.

A *coclass* name corresponds to the display name of a COM class; the display name is not uniquely mapped to a COM class, and some COM classes have no display name.

```
(com-all-coclasses) → (listof string?)
```

Returns a list of coclass strings for all COM classes registered on a system.

```
(com-all-controls) → (listof string?)
```

Returns a list of coclass strings for all COM classes in the system registry that have the `"Control"` subkey.

```
(coclass->clsid coclass) → clsid?
  coclass : string?
(clsid->coclass clsid) → string?
  clsid : clsid?
```

Converts a coclass string to/from a CLSID. This conversion is implemented by an enumeration an COM classes from the system registry.

### 5.18.2 COM Classes and Interfaces

```
(require ffi/unsafe/com)
```
package: `base`

The `ffi/unsafe/com` library exports all of `ffi/com`, and it also supports direct, FFI-based calls to COM object methods.

#### Describing COM Interfaces

```
(define-com-interface (_id _super-id)
  ([method-id ctype-expr maybe-alloc-spec] ...))

maybe-alloc-spec =
                 | #:release-with-function function-id
                 | #:release-with-method method-id
                 | #:releases
```

Defines *_id* as an interface that extends *_super-id*, where *_super-id* is often `_IUnknown`, and that includes methods named by *method-id*. The *_id* and *_super-id* identifiers must start with an underscore. A *_super-id* `_vt` must also be defined for deriving a virtual-method table type.

The order of the *method-id*s must match the specification of the COM interface, not including methods inherited from *_super-id*. Each method type produced by *ctype-expr*

that is not _fpointer must be a function type whose first argument is the "self" pointer, usually constructed with _mfun or _hmfun.

The define-com-interface form binds _id, id ?, _id -pointer, _id _ vt (for the virtual-method table), _id _ vt-pointer, and method-id for each method whose ctype-expr is not _fpointer. (In other words, use _fpointer as a placeholder for methods of the interface that you do not need to call.) An instance of the interface will have type _id -pointer. Each defined method-id is bound to a function-like macro that expects a _id -pointer as its first argument and the method arguments as the remaining arguments.

A maybe-alloc-spec describes allocation and finalization information for a method along the lines of ffi/unsafe/alloc. If the maybe-alloc-spec is #:release-with-function function-id, then function-id is used to deallocate the result produced by the method, unless the result is explicitly deallocated before it becomes unreachable; for example, #:release-with-function Release is suitable for a method that returns a COM interface reference that must be eventually released. The #:release-with-method method-id form is similar, except that the deallocator is a method on the same object as the allocating method (i.e., one of the other method-ids or an inherited method). A #:releases annotation indicates that a method is a deallocator (so that a value should not be automatically deallocated if it is explicitly deallocated using the method).

See §5.18.2.4 "COM Interface Example" for an example using define-com-interface.

**Obtaining COM Interface References**

```
(QueryInterface iunknown
                iid
                intf-pointer-type) → (or/c cpointer? #f)
  iunknown : com-iunknown?
  iid : iid?
  intf-pointer-type : ctype?
```

Attempts to extract a COM interface pointer for the given COM object. If the object does not support the requested interface, the result is #f, otherwise it is cast to the type intf-pointer-type.

Specific IIDs and intf-pointer-types go together. For example, IID_IUnknown goes with _IUnknown-pointer.

For a non-#f result, Release function is the automatic deallocator for the resulting pointer. The pointer is register with a deallocator after the cast to intf-pointer-type, which is why QueryInterface accepts the intf-pointer-type argument (since a cast generates a fresh reference).

```
(AddRef iunknown) → exact-positive-integer?
  iunknown : com-iunknown?
(Release iunknown) → exact-nonnegative-integer?
  iunknown : com-iunknown?
```

Increments or decrements the reference count on *iunknown*, returning the new reference count and releasing the interface reference if the count goes to zero.

```
(make-com-object  iunknown
                  clsid
                 [#:manage? manage?]) → com-object?
  iunknown : com-iunknown?
  clsid : (or/c clsid? #f)
  manage? : any/c = #t
```

Converts a COM object into an object that can be used with the COM automation functions, such as `com-invoke`.

If *manage?* is true, the resulting object is registered with the current custodian and a finalizer to call `com-release` when the custodian is shut down or when the object becomes inaccessible.

### COM FFI Helpers

```
(_wfun fun-option ... maybe-args type-spec ... -> type-spec
   maybe-wrapper)
```

Like `_fun`, but adds `#:abi winapi`.

```
(_mfun fun-option ... maybe-args type-spec ... -> type-spec
   maybe-wrapper)
```

Like `_wfun`, but adds a `_pointer` type (for the "self" argument of a method) as the first argument *type-spec*.

```
(_hfun fun-option ... type-spec ... -> id maybe-allow output-expr)

maybe-allow =
            | #:allow [result-id allow?-expr]
```

Like `_wfun`, but for a function that returns an `_HRESULT`. The result is bound to *result-id* if `#:allow` is specified, otherwise the result is not directly accessible.

The `_hfun` form handles the `_HRESULT` value of the foreign call as follows:

- If the result is zero or if `#:allow` is specified and *allow?-expr* produces `#t`, then *output-expr* (as in a *maybe-wrapper* for `_fun`) determines the result.
- If the result is `RPC_E_CALL_REJECTED` or `RPC_E_SERVERCALL_RETRYLATER`, the call is automatically retried up to `(current-hfun-retry-count)` times with a delay of `(current-hfun-retry-delay)` seconds between each attempt.

- Otherwise, an error is raised using `windows-error` and using *id* as the name of the failed function.

Changed in version 6.2 of package `base`: Added `#:allow` and automatic retries.

```
(_hmfun fun-option ... type-spec ... -> id output-expr)
```

Like `_hfun`, but lke `_mfun` in that `_pointer` is added for the first argument.

```
(current-hfun-retry-count) → count
(current-hfun-retry-count exact-nonnegative-integer?) → void?
  exact-nonnegative-integer? : count
(current-hfun-retry-delay) → (>=/c 0.0)
(current-hfun-retry-delay secs) → void?
  secs : (>=/c 0.0)
```

Parameters that determine the behavior of automatic retries for `_hfun`.

Added in version 6.2 of package `base`.

```
(HRESULT-retry? r) → boolean?
  r : exact-nonnegative-integer?
```

Returns `#t` if *r* is `RPC_E_CALL_REJECTED` or `RPC_E_SERVERCALL_RETRYLATER`, `#f` otherwise.

Added in version 6.2 of package `base`.

```
_GUID : ctype?
_GUID-pointer : ctype?
_HRESULT : ctype?
_LCID : ctype?
```

Some C types that commonly appear in COM interface specifications.

```
LOCALE_SYSTEM_DEFAULT : exact-integer?
```

The usual value for a `_LCID` argument.

```
(SysFreeString str) → void?
  str : _pointer
(SysAllocStringLen content len) → cpointer?
  content : _pointer
  len : integer?
```

COM interfaces often require or return srings that must be allocated or freed as system strings.

When receiving a string value, `cast` it to `_string/utf-16` to extract a copy of the string, and then free the original pointer with `SysFreeString`.

```
IID_NULL : iid?
IID_IUnknown : iid?
```

Commonly used IIDs.

```
_IUnknown : ctype?
_IUnknown-pointer : ctype?
_IUnknown_vt : ctype?
```

Types for the `IUnknown` COM interface.

```
(windows-error msg hresult) → any
  msg : string?
  hresult : exact-integer?
```

Raises an exception. The *msg* string provides the base error message, but *hresult* and its human-readable interpretation (if available) are added to the message.

**COM Interface Example**

Here's an example using the Standard Component Categories Manager to enumerate installed COM classes that are in the different system-defined categories. The example illustrates instantiating a COM class by CLSID, describing COM interfaces with `define-com-interface`, and using allocation specifications to ensure that resources are reclaimed even if an error is encountered or the program is interrupted.

```
#lang racket/base
(require ffi/unsafe
         ffi/unsafe/com)

(provide show-all-classes)

; The function that uses COM interfaces defined further below:

(define (show-all-classes)
  (define ccm
    (com-create-instance CLSID_StdComponentCategoriesMgr))
  (define icat (QueryInterface (com-object-get-iunknown ccm)
                               IID_ICatInformation
                               _ICatInformation-pointer))
```

```
  (define eci (EnumCategories icat LOCALE_SYSTEM_DEFAULT))
  (for ([catinfo (in-producer (lambda () (Next/ci eci)) #f)])
    (printf "~a:\n"
            (cast (array-ptr (CATEGORYINFO-szDescription catinfo))
                  _pointer
                  _string/utf-16))
    (define eg
      (EnumClassesOfCategories icat (CATEGORYINFO-catid catinfo)))
    (for ([guid (in-producer (lambda () (Next/g eg)) #f)])
      (printf " ~a\n" (or (clsid->progid guid)
                          (guid->string guid))))
    (Release eg))
  (Release eci)
  (Release icat))

; The class to instantiate:

(define CLSID_StdComponentCategoriesMgr
  (string->clsid "{0002E005-0000-0000-C000-000000000046}"))

; Some types and variants to match the specification:

(define _ULONG _ulong)
(define _CATID _GUID)
(define _REFCATID _GUID-pointer)
(define-cstruct _CATEGORYINFO ([catid _CATID]
                               [lcid _LCID]
                               [szDescription (_array _short 128)]))

; ---- IEnumGUID -----

(define IID_IEnumGUID
  (string->iid "{0002E000-0000-0000-C000-000000000046}"))

(define-com-interface (_IEnumGUID _IUnknown)
  ([Next/g (_mfun (_ULONG = 1) ; simplifed to just one
                  (guid : (_ptr o _GUID))
                  (got : (_ptr o _ULONG))
                  -> (r : _HRESULT)
                  -> (cond
                       [(zero? r) guid]
                       [(= r 1) #f]
                       [else (windows-error "Next/g failed" r)]))]
   [Skip _fpointer]
   [Reset _fpointer]
   [Clone _fpointer]))
```

```
; ---- IEnumCATEGORYINFO -----

(define IID_IEnumCATEGORYINFO
  (string->iid "{0002E011-0000-0000-C000-000000000046}"))

(define-com-interface (_IEnumCATEGORYINFO _IUnknown)
  ([Next/ci (_mfun (_ULONG = 1) ; simplifed to just one
                   (catinfo : (_ptr o _CATEGORYINFO))
                   (got : (_ptr o _ULONG))
                   -> (r : _HRESULT)
                   -> (cond
                       [(zero? r) catinfo]
                       [(= r 1) #f]
                       [else (windows-error "Next/ci
failed" r)]))]
   [Skip _fpointer]
   [Reset _fpointer]
   [Clone _fpointer]))

; ---- ICatInformation -----

(define IID_ICatInformation
  (string->iid "{0002E013-0000-0000-C000-000000000046}"))

(define-com-interface (_ICatInformation _IUnknown)
  ([EnumCategories (_hmfun _LCID
                           (p : (_ptr o _IEnumCATEGORYINFO-
pointer))
                           -> EnumCategories p)]
   [GetCategoryDesc (_hmfun _REFCATID _LCID
                            (p : (_ptr o _pointer))
                            -> GetCategoryDesc
                            (begin0
                             (cast p _pointer _string/utf-16)
                             (SysFreeString p)))]
   [EnumClassesOfCategories (_hmfun (_ULONG = 1) ; simplifed
                                    _REFCATID
                                    (_ULONG = 0) ; simplifed
                                    (_pointer = #f)
                                    (p : (_ptr o
                                              _IEnumGUID-
pointer))
                                    -> EnumClassesOfCategories p)
                            #:release-with-function Release]
   [IsClassOfCategories _fpointer]
```

```
  [EnumImplCategoriesOfClass _fpointer]
  [EnumReqCategoriesOfClass _fpointer]))
```

### 5.18.3 ActiveX Controls

An ActiveX control is a COM object that needs a container to manage its graphical representation. Although `ffi/com` does not provide direct support for ActiveX controls, you can use `ffi/com` to drive Internet Explorer as an ActiveX container.

The following code demonstrates using Internet Explorer to instantiate the "Sysmon" ActiveX control that is included with Windows.

```
#lang racket
(require ffi/com
         xml)

;; The control we want to run:
(define control-progid "Sysmon")

;; Start IE:
(define ie (com-create-instance "InternetExplorer.Application.1"))

;; Set up an event callback so that we know when the initial document
;; is ready:
(define ex (com-make-event-executor))
(void (thread (lambda () (let loop () ((sync ex)) (loop)))))
(define ready (make-semaphore))
(com-register-event-callback ie "DocumentComplete"
                             (lambda (doc url) (semaphore-
post ready))
                             ex)

;; Navigate to get an initial document:
(com-invoke ie "Navigate" "about:blank")
(define READYSTATE_COMPLETE 4)
(unless (= (com-get-property ie "READYSTATE") READYSTATE_COMPLETE)
  (semaphore-wait ready))
(define doc (com-get-property ie "Document"))

;; Install HTML to show the ActiveX control:
(com-invoke doc "write"
            (xexpr->string
             `(html
               (head (title "Demo"))
               (body
```

```
                (object ((class "object")
                         (CLASSID ,(format
                                    "CLSID:~a"
                                    (let ([s (guid->string
                                              (progid->clsid
                                               control-progid))])
                                      ;; must remove curly braces:
                                      (define len
                                        (string-length s))
                                      (substring s 1 (sub1 len))))))))))

;; Configure the IE window and show it:
(com-set-property! ie "MenuBar" #f)
(com-set-property! ie "ToolBar" 0)
(com-set-property! ie "StatusBar" #f)
(com-set-property! ie "Visible" #t)

;; Extract the ActiveX control from the IE document:
(define ctl (com-get-property
             (com-invoke (com-invoke doc "getElementsByTagName" "object")
                         "item"
                         0)
             "object"))

;; At this point, `ctl' is the ActiveX control;
;; demonstrate by getting a list of method names:
(com-methods ctl)
```

## 5.19 File Security-Guard Checks

```
(require ffi/file)
```
package: base

```
(security-guard-check-file who path perms) → void?
  who : symbol?
  path : path-string?
  perms : (listof (or/c 'read 'write 'execute 'delete 'exists))
```

Checks whether `(current-security-guard)` permits access to the file specified by *path* with the permissions *perms*. See `make-security-guard` for more information on *perms*.

The symbol *who* should be the name of the function on whose behalf the security check is performed; it is passed to the security guard to use in access-denied errors.

```
(_file/guard perms [who]) → ctype?
  perms : (listof (or/c 'read 'write 'execute 'delete 'exists))
```

```
  who : symbol? = '_file/guard
```

Like `_file` and `_path`, but conversion from Racket to C first completes the path using `path->complete-path` then cleanses it using `cleanse-path`, then checks that the current security guard grants access on the resulting complete path with *perms*. As an output value, identical to `_path`.

```
_file/r : ctype?
_file/rw : ctype?
```

Equivalent to `(_file/guard '(read) '_file/r)` and `(_file/guard '(read write) '_file/rw)`, respectively.

```
(security-guard-check-file-link who
                                path
                                dest) → void?
  who : symbol?
  path : path-string?
  dest : path-string?
```

Checks whether `(current-security-guard)` permits link creation of *path* as a link *dest*. The symbol *who* is the same as for `security-guard-check-file`.

Added in version 6.9.0.5 of package `base`.

```
(security-guard-check-network who
                              host
                              port
                              mode) → void?
  who : symbol?
  host : (or/c string? #f)
  port : (or/c (integer-in 1 65535) #f)
  mode : (or/c 'client 'server)
```

Checks whether `(current-security-guard)` permits network access at *host* and *port* in server or client mode as specified by *mode*. The symbol *who* is the same as for `security-guard-check-file`.

Added in version 6.9.0.5 of package `base`.

## 5.20 Windows API Helpers

`(require ffi/winapi)` package: `base`

```
win64? : boolean?
```

Indicates whether the current platform is 64-bit Windows: `#t` if so, `#f` otherwise.

```
winapi : (or/c 'stdcall 'default)
```

Suitable for use as an ABI specification for a Windows API function: `'stdcall` on 32-bit Windows, `'default` on 64-bit Windows or any other platform.

## 5.21 Windows VCRuntime Librraies

`(require ffi/vcruntime)` package: `base`

The `ffi/vcruntime` module has no exports, but `require`ing it has the side effect on Windows of loading `"vcruntime140.dll"`, loading `"vcruntime140_1.dll"`, and declaring those DLLs to be carried along by `raco exe` and `raco dist`.

To avoid conflicts, use `ffi/vcruntime` instead of including your own copies of `"vcruntime140.dll"` and `"vcruntime140_1.dll"` in a racket package.

Added in version 9.3.0.2 of package `base`.

## 5.22 Virtual Machine Primitives

`(require ffi/unsafe/vm)` package: `base`

The `ffi/unsafe/vm` library provides access to functionality in the underlying virtual machine that is used to implement Racket.

Added in version 7.6.0.7 of package `base`.

```
(vm-primitive name) → any/c
  name : symbol?
```

Accesses a primitive values at the level of the running Racket virtual machine, or returns `#f` if *name* is not the name of a primitive.

Virtual-machine primitives are the ones that can be referenced in a linklet body. The specific set of primitives depends on the virtual machine. Many "primitives" at the `racket/base` level or even the `'#%kernel` level are not primitives at the virtual-machine level. For example, if `'eval` is available as a primitive, it is not the `eval` from `racket/base`.

In general, primitives are unsafe and can only be used with enough knowledge about Racket's implementation. Here are some tips for currently available virtual machines:

- `(system-type 'vm)` is `'racket` — The primitives in this virtual machine are mostly the same as the ones available from libraries like `racket/base` and `racket/unsafe/ops`. As a result, accessing virtual machine primitives with `vm-primitive` is rarely useful.

- `(system-type 'vm)` is `'chez-scheme` — The primitives in this virtual machine are Chez Scheme primitives, except as replaced by a Racket compatibility layer. The `'eval` primitive is Chez Scheme's `eval`.

  Beware of directly calling a Chez Scheme primitive that uses Chez Scheme parameters or `dynamic-wind` internally. Note that `eval`, in particular, is such a primitive. The problem is that Chez Scheme's `dynamic-wind` does not automatically cooperate with Racket's continuations or threads. To call such primitives, use the `call-with-system-wind` primitive, which takes a procedure of no arguments to run in a context that bridges Chez Scheme's `dynamic-wind` and Racket continuations and threads. For example,

  ```
  (define primitive-eval (vm-primitive 'eval))
  (define call-with-system-wind (vm-primitive 'call-with-
  system-wind))
  (define (vm-eval s)
    (call-with-system-wind
     (lambda ()
      (primitive-eval s))))
  ```

  is how `vm-eval` is implemented on Chez Scheme.

  Symbols, numbers, booleans, pairs, vectors, boxes, strings, byte strings (i.e., bytevectors), and structures (i.e., records) are interchangeable between Racket and Chez Scheme. A Chez Scheme procedure is a Racket procedure, but not all Racket procedures are Chez Scheme procedures. To call a Racket procedure from Chez Scheme, use the `#%app` form that is defined in the Chez Scheme environment when it hosts Racket.

  Note that you can access Chez Scheme primitives, including ones that are shadowed by Racket's primitives, through the Chez Scheme `$primitive` form. For example, `(vm-eval '($primitive call-with-current-continuation))` accesses the Chez Scheme `call-with-current-continuation` primitive instead of Racket's replacement (where the replacement works with Racket continuations and threads).

```
(vm-eval s-expr) → any/c
  s-expr : any/c
```

Evaluates *s-expr* using the most primitive available evaluator:

- `(system-type 'vm)` is `'racket` — Uses `compile-linklet` and `instantiate-linklet`.

- `(system-type 'vm)` is `'chez-scheme` — Uses Chez Scheme's `eval`.

See `vm-primitive` for some information about how virtual-machine primitives interact with Racket.

## 5.23 Declaring Foreign Libraries Needed at Run Time

`(require ffi/unsafe/runtime-lib)` package: `base`

The `ffi/unsafe/runtime-lib` library is similar to `racket/runtime-path` (and builds on it), but provides more support for platform-specific choices of libraries.

Added in version 9.2.0.6 of package `base`.

```
(define-runtime-lib id
   maybe-ffi-lib-args
   [platform-spec lib-spec ...]
   ...
   [else else-body ...+])

     platform-spec = os-id
                   | os*-id
                   | arch-id
                   | platform-string
                   | 32
                   | 64
                   | (or platform-spec ...)
                   | (and platform-spec ...)

          lib-spec = (so lib-string)
                   | (so lib-string (vers ...))

              vers = string
                   | #f

maybe-ffi-lib-args =
                   | #:ffi-lib-args (arg ...)
```

Similar to `define-runtime-path`, but binds *id* to a result from `ffi-lib` for a platform that matches one of the *platform-spec*s (which are tried in order), or to the result of the *else-body* sequence otherwise.

A *platform-spec* implies a set of libraries that are loaded in order as enumerated by the accompanying *lib-spec*s, where the *lib-string* within a *lib-spec* becomes the first

argument to ffi-lib, and the *vers* sequence (if present) is quoted as the second argument. The defined *id* is bound to a ffi-lib result for the last *lib-spec*, or to #f if no *lib-spec*s are present in the matching clause. Besides loading the library for each *lib-spec* at run time, the libraries are declared at compile time for use by tools such as raco exe and raco dist. Cross compilation is handled automatically, so that the target platforms libraries are listed at compile time, while run time loads libraries suitable to the host platform.

If no *platform-spec* matches, then *id* is bound to the result of the *else-body* sequence—which does not have to be a result from ffi-lib, but typically it is. The *else-body* sequence is evaluated only at run-time, and no libraries are declared at compile time.

Each *platform-spec* is is compared to the result of (system-type 'os), (system-type 'os*), (system-type 'arch), (system-type 'platform), and/or (system-type 'word). In an and form, all *platform-spec*s much match, while only one matching *platform-spec* is needed ro an or form.

Extra keyword arguments can be supplied the run-time ffi-lib call for a *platform-spec* match through an optional #:ffi-lib-args declaration. There's a single #:ffi-lib-args declaration, because it needs to be independent of the *platform-spec* that turns out to match. The run-time call receives either an absolute path based on resolved *lib-spec*, or it receives just the initial *lib-string* within a *lib-spec* (due to limitations of define-runtime-path), so #:fail is the most likely useful extra argument to ffi-lib.

# 6 Miscellaneous Support

```
(list->cblock  lst
               type
              [expect-length
               #:malloc-mode malloc-mode]) → cpointer?
  lst : list?
  type : ctype?
  expect-length : (or/c exact-nonnegative-integer? #f) = #f
  malloc-mode : (or/c #f symbol?) = #f
```

Allocates a memory block of an appropriate size—using `malloc` with *type* and `(length` *lst*`)`—and initializes it using values from *lst*. The *lst* must hold values that can all be converted to C values according to the given *type*.

If *expect-length* is not `#f` and not the same as `(length` *lst*`)`, then an exception is raised instead of allocating memory.

If *malloc-mode* is not `#f`, it is provided as an additional argument to `malloc`.

Changed in version 7.7.0.2 of package `base`: Added the `#:malloc-mode` argument.

```
(vector->cblock  vec
                 type
                [expect-length
                 #:malloc-mode malloc-mode]) → cpointer?
  vec : vector?
  type : ctype?
  expect-length : (or/c exact-nonnegative-integer? #f) = #f
  malloc-mode : (or/c #f symbol?) = #f
```

Like `list->cblock`, but using values from a vector instead of a list.

Changed in version 7.7.0.2 of package `base`: Added the `#:malloc-mode` argument.

```
(vector->cpointer vec) → cpointer?
  vec : vector?
```

Returns a pointer to an array of `_scheme` values, which is the internal representation of *vec*.

```
(flvector->cpointer flvec) → cpointer?
  flvec : flvector?
```

Returns a pointer to an array of `_double` values, which is the internal representation of *flvec*.

```
(saved-errno) → exact-integer?
(saved-errno new-value) → void?
  new-value : exact-integer?
```

Returns or sets the error code saved for the current Racket thread. The saved error code is set after a foreign call with a non-`#f` `#:save-errno` option (see `_fun` and `_cprocedure`), but it can also be set explicitly (for example, to create mock foreign functions for testing).

Changed in version 6.4.0.9 of package `base`: Added the one-argument variant.

```
(lookup-errno sym) → (or/c exact-integer? #f)
  sym : symbol?
```

Returns a platform-specific positive integer corresponding to a POSIX `errno` code, or `#f` if the code is unknown. A code's value is known if the code is one of the recognized symbols described below *and* the code was defined by the `"errno.h"` header used to compile Racket. Note that the contents of `"errno.h"` vary based on platform and compiler.

The recognized symbols currently consist of the 81 codes defined by IEEE Std 1003.1, 2013 Edition (also known as POSIX.1), including `'EINTR`, `'EEXIST`, and `'EAGAIN`.

See also `exn-classify-errno`.

Changed in version 6.6.0.5 of package `base`: Relaxed the contract and added support for more symbols.

```
(cast v from-type to-type) → any/c
  v : any/c
  from-type : ctype?
  to-type : ctype?
```

Converts *v* from a value matching *from-type* to a value matching *to-type*, where `(ctype-sizeof` *from-type*`)` matches `(ctype-sizeof` *to-type*`)`.

The conversion is roughly equivalent to

```
(let ([p (malloc from-type)])
  (ptr-set! p from-type v)
  (ptr-ref p to-type))
```

If *v* is a cpointer, `(cpointer-gcable?` *v*`)` is true, and *from-type* and *to-type* are both based on `_pointer` or `_gcpointer`, then *from-type* is implicitly converted with `_gcable` to ensure that the result cpointer is treated as referring to memory that is managed by the garbage collector.

If *v* is a pointer with an offset component (e.g., from `ptr-add`), `(cpointer-gcable?` *v*`)` is true, and the result is a cpointer, then the result pointer has the same offset component as

`v`. If `(cpointer-gcable? v)` is false, then any offset is folded into the pointer base for the result.

```
(cblock->list cblock type length) → list?
  cblock : any/c
  type : ctype?
  length : exact-nonnegative-integer?
```

Converts C *cblock*, which is a vector of *type*s, to a Racket list. The arguments are the same as in the `list->cblock`. The *length* must be specified because there is no way to know where the block ends.

```
(cblock->vector cblock type length) → vector?
  cblock : any/c
  type : ctype?
  length : exact-nonnegative-integer?
```

Like `cblock->list`, but for Racket vectors.

# 7 Unexported Primitive Functions

Parts of the ffi/unsafe library are implemented by the Racket built-in '#%foreign module. The '#%foreign module is not intended for direct use, but it exports the following procedures (among others).

```
(ffi-obj objname lib) → ffi-obj?
  objname : bytes?
  lib : (or/c ffi-lib? path-string? #f)
```

Pulls out a foreign object from a library, returning a value that can be used as a C pointer. If *lib* is a path or string, then ffi-lib is used to create a library object.

```
(ffi-obj? x) → boolean?
  x : any/c
(ffi-obj-lib obj) → ffi-lib?
  obj : ffi-obj?
(ffi-obj-name obj) → bytes?
  obj : ffi-obj?
```

A predicate for objects returned by ffi-obj, and accessor functions that return its corresponding library object and name. These values can also be used as C pointer objects.

```
(ctype-basetype type) → (or/c ctype? #f)
  type : ctype?
(ctype-scheme->c type) → procedure?
  type : ctype?
(ctype-c->scheme type) → procedure?
  type : ctype?
```

Accessors for the components of a C type object, made by make-ctype. The ctype-basetype selector returns a symbol for primitive types that names the type, a list of ctypes for cstructs, and another ctype for user-defined ctypes.

```
(ffi-call  ptr
           in-types
           out-type
          [abi
           save-errno?
           orig-place?
           lock-name
           blocking?
           varargs-after]) → procedure?
  ptr : cpointer?
  in-types : (listof ctype?)
```

```
  out-type : ctype?
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  save-errno? : any/c = #f
  orig-place? : any/c = #f
  lock-name : (or/c #f string?) = #f
  blocking? : any/c = #f
  varargs-after : (or/c #f positive-exact-integer?) = #f
```

The primitive mechanism that creates Racket callout values for `_cprocedure`. The given *ptr* is wrapped in a Racket-callable primitive function that uses the types to specify how values are marshaled.

```
(ffi-call-maker  in-types
                 out-type
                [abi
                 save-errno?
                 orig-place?
                 lock-name
                 blocking?
                 varargs-after]) → (cpointer . -> . procedure?)
  in-types : (listof ctype?)
  out-type : ctype?
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  save-errno? : any/c = #f
  orig-place? : any/c = #f
  lock-name : (or/c #f string?) = #f
  blocking? : any/c = #f
  varargs-after : (or/c #f positive-exact-integer?) = #f
```

A curried variant of `ffi-call` that takes the foreign-procedure pointer separately.

```
(ffi-callback  proc
               in-types
               out-type
              [abi
               atomic?
               async-apply
               varargs-after]) → ffi-callback?
  proc : procedure?
  in-types : any/c
  out-type : any/c
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  atomic? : any/c = #f
  async-apply : (or/c #f ((-> any) . -> . any) box?) = #f
  varargs-after : (or/c #f positive-exact-integer?) = #f
```

The symmetric counterpart of `ffi-call`. It receives a Racket procedure and creates a callback object, which can also be used as a C pointer.

```
(ffi-callback-maker  in-types
                     out-type
                    [abi
                     atomic?
                     async-apply
                     varargs-after])
 → (procedure? . -> . ffi-callback?)
  in-types : any/c
  out-type : any/c
  abi : (or/c #f 'default 'stdcall 'sysv) = #f
  atomic? : any/c = #f
  async-apply : (or/c #f ((-> any) . -> . any) box?) = #f
  varargs-after : (or/c #f positive-exact-integer?) = #f
```

A curried variant of `ffi-callback` that takes the callback procedure separately.

```
(ffi-callback? v) → boolean?
  v : any/c
```

A predicate for callback values that are created by `ffi-callback`.

```
(make-late-will-executor) → will-executor?
```

Creates a “late” will executor that readies a will for a value `v` only if no normal will executor has a will registered for `v`. In addition, for the BC implementation of Racket, normal weak references to `v` are cleared before a will for `v` is readied by the late will executor, but late weak references created by `make-late-weak-box` and `make-late-weak-hasheq` are not. For the CS implementation of Racket, a will is readied for `v` only when it is not reachable from any value that has a late will; if a value `v` is reachable from itself (i.e., through any field of `v`, as opposed to the immediate value itself), a “late” will for `v` never becomes ready.

Unlike a normal will executor, if a late will executor becomes inaccessible, the values for which it has pending wills are retained within the late will executor’s place.

A late will executor is intended for use in the implementation of `register-finalizer`.

# Bibliography

[Barzilay04] Eli Barzilay and Dmitry Orlovsky, “Foreign Interface for PLT Scheme,” Workshop on Scheme and Functional Programming, 2004.

# Index


->
_?
_array
_array/list
_array/vector
_bitmask
_BOOL
_bool
_box
_byte
_bytes
_bytes/eof
_bytes/nul-terminated
_Class
_cpointer
_cpointer/null
_cprocedure
_cvector
_double
_double*
_enum
_f32vector
_f64vector
_f80vector
_file
_file/guard
_file/r
_file/rw
_fixint
_fixnum
_float
_fpointer
_fun
_fun
_gcable
_gcpointer
_GUID
_GUID-pointer
_hfun
_hmfun
_HRESULT
_id
_int
_int16
_int32
_int64
_int8
_intmax
_intptr
_IUnknown
_IUnknown-pointer
_IUnknown_vt
_Ivar
_LCID
_list
_list-struct
_llong
_long
_longdouble
_mfun
_NSString
_objc_super
_or-null
_path
_pointer
_Protocol
_ptr
_ptrdiff
_racket
_s16vector
_s32vector
_s64vector
_s8vector
_sbyte
_scheme
_SEL
_short
_sint
_sint16
_sint32
_sint64
_sint8

_sintptr
_size
_sllong
_slong
_sshort
_ssize
_stdbool
_string
_string*/latin-1
_string*/locale
_string*/utf-8
_string/eof
_string/latin-1
_string/locale
_string/ucs-4
_string/utf-16
_string/utf-8
_string/utf-8/immobile
_sword
_symbol
_u16vector
_u32vector
_u64vector
_u8vector
_ubyte
_ufixint
_ufixnum
_uint
_uint16
_uint32
_uint64
_uint8
_uintmax
_uintptr
_ullong
_ulong
_union
_ushort
_uword
_vector
_void
_wchar
_wfun
_word
ActiveX Controls
AddRef
Allocation and Finalization
Allocation Pools
allocator
*allocator*
array-length
array-ptr
array-ref
array-set!
array-type
array?
'atomic
Atomic Execution
*Atomic mode*
'atomic-interior
By-Reference Arguments
C Array Types
*C pointer*
C Struct Types
C Structs
C Types
*C types*
C Union Types
call-as-atomic
call-as-nonatomic
call-as-nonatomic-retry-point
call-as-uninterruptible
call-in-os-thread
call-with-autorelease
*callback*
*callout*
cast
cblock->list
cblock->vector
CFStringRef
Class Display Names
class_addIvar
class_addMethod
class_getSuperclass

*CLSID*
clsid->coclass
clsid->progid
clsid?
*coclass*
coclass->clsid
Cocoa Foundation
COM (Common Object Model)
COM Automation
*COM class*
COM Classes and Interfaces
COM Enumerations
*COM event executor*
COM Events
COM FFI Helpers
COM Interface Example
*COM interfaces*
COM Methods
*COM object*
COM Objects
COM Properties
COM Types
com-all-coclasses
com-all-controls
com-create-instance
com-enumerate-to-list
com-enumeration-to-list
com-event-executor?
com-event-type
com-events
com-get-active-object
com-get-properties
com-get-property
com-get-property*
com-get-property-type
com-idispatch?
com-invoke
com-iunknown?
com-make-event-executor
com-method-type
com-methods
com-object-clsid
com-object-eq?
com-object-get-idispatch
com-object-get-iunknown
com-object-set-clsid!
com-object-type
com-object?
com-omit
com-omit?
com-register-event-callback
com-release
com-set-properties
com-set-property!
com-set-property-type
com-type=?
com-type?
com-unregister-event-callback
compiler-sizeof
compute-offsets
convention:hyphen->camelCase
convention:hyphen->camelcase
convention:hyphen->PascalCase
convention:hyphen->underscore
cpointer-gcable?
cpointer-has-tag?
cpointer-predicate-procedure?
cpointer-push-tag!
cpointer-tag
cpointer?
ctype->layout
ctype-alignof
ctype-basetype
ctype-c->scheme
ctype-scheme->c
ctype-sizeof
ctype?
current-hfun-retry-count
current-hfun-retry-delay
Custodian Shutdown Registration
Custom Function Types
*custom function types*
cvector
cvector->list

cvector-length
cvector-ptr
cvector-ref
cvector-set!
cvector-type
cvector?
*DCOM*
deallocator
*deallocator*
Declaring Foreign Libraries Needed at Run Time
default-_string-type
define-c
define-com-interface
define-cpointer-type
define-cstruct
define-ffi-definer
define-fun-syntax
define-objc-class
define-objc-mixin
define-runtime-lib
define-serializable-cstruct
Defining Bindings
Derived Utilities
Describing COM Interfaces
dynamically loaded libraries
end-atomic
end-breakable-atomic
end-stubborn-change
end-uninterruptible
Enumerations and Masks
errno
'eternal
f32vector
f32vector->cpointer
f32vector->list
f32vector-length
f32vector-ref
f32vector-set!
f32vector?
f64vector
f64vector->cpointer
f64vector->list
f64vector-length
f64vector-ref
f64vector-set!
f64vector?
f80vector
f80vector->cpointer
f80vector->list
f80vector-length
f80vector-ref
f80vector-set!
f80vector?
'fail-ok
'failok
*FFI*
FFI Identifier Conventions
*FFI identifier conventions*
FFI Types and Constants
ffi-call
ffi-call-maker
ffi-callback
ffi-callback-maker
ffi-callback?
ffi-lib
ffi-lib?
ffi-obj
ffi-obj-lib
ffi-obj-name
ffi-obj-ref
ffi-obj?
ffi/com
ffi/com-registry
ffi/cvector
ffi/file
ffi/objc
ffi/serialize-cstruct
ffi/unsafe
ffi/unsafe/alloc
ffi/unsafe/atomic
ffi/unsafe/collect-callback
ffi/unsafe/com
ffi/unsafe/custodian

ffi/unsafe/cvector
ffi/unsafe/define
ffi/unsafe/define/conventions
ffi/unsafe/global
ffi/unsafe/nsalloc
ffi/unsafe/nsstring
ffi/unsafe/objc
ffi/unsafe/os-async-channel
ffi/unsafe/os-thread
ffi/unsafe/port
ffi/unsafe/runtime-lib
ffi/unsafe/schedule
ffi/unsafe/static
ffi/unsafe/try-atomic
ffi/unsafe/vm
ffi/vcruntime
ffi/vector
ffi/winapi
File Security-Guard Checks
Fixed Auto-Converting String Types
flvector->cpointer
*foreign-library value*
free
free-immobile-cell
Function Types
function-ptr
Function-Type Bells and Whistles
Garbage Collection Callbacks
get-ffi-obj
get-ivar
get-place-table
GetLastError
*GUID*
guid->string
guid=?
guid?
GUIDs, CLSIDs, IIDs, and ProgIDs
HRESULT-retry?
IDispatch
*IID*
iid?
IID_IUnknown
IID_NULL
import-class
import-protocol
in-array
in-atomic-mode?
Interface Pointers
'interior
Legacy Library
Libraries, C Types, and Objects
list->cblock
list->cvector
list->f32vector
list->f64vector
list->f80vector
list->s16vector
list->s32vector
list->s64vector
list->s8vector
list->u16vector
list->u32vector
list->u64vector
list->u8vector
Loading Foreign Libraries
LOCALE_SYSTEM_DEFAULT
lookup-errno
make-array-type
make-c-parameter
make-com-object
make-cstruct-type
make-ctype
make-custodian-at-root
make-cvector
make-cvector*
make-f32vector
make-f64vector
make-f80vector
make-late-weak-box
make-late-weak-hasheq
make-late-will-executor
make-not-available
make-objc_super
make-os-async-channel

make-os-semaphore
make-s16vector
make-s32vector
make-s64vector
make-s8vector
make-sized-byte-string
make-u16vector
make-u32vector
make-u64vector
make-u8vector
make-uninterruptible-lock
make-union-type
malloc
malloc-immobile-cell
memcpy
memmove
Memory Management
memset
Miscellaneous Support
More Examples
*MysterX*
NO
'nonatomic
NSString*
Numeric Types
objc-block
objc-block-function-pointer
objc-class-has-instance-method?
objc-dispose-class
objc-get-class
objc-get-superclass
objc-is-a?
objc-set-class!
objc-subclass?
objc-unsafe!
objc_allocateClassPair
objc_getProtocol
objc_lookUpClass
objc_msgSend/typed
objc_msgSend/typed/blocking
objc_msgSendSuper/typed
objc_msgSendSuper/typed/blocking
objc_registerClassPair
object_getClass
object_getInstanceVariable
object_setInstanceVariable
Objective-C FFI
Obtaining COM Interface References
offset-ptr?
Operating System Asynchronous Channels
Operating System Threads
os-async-channel-get
os-async-channel-put
os-async-channel-try-get
os-async-channel?
os-semaphore-post
os-semaphore-wait
os-thread-enabled?
Other Atomic Types
Other String Types
Overview
Pointer Dereferencing
Pointer Functions
Pointer Structure Property
Pointer Types
Pointers and GC-Managed Allocation
Pointers and Manual Allocation
*poller*
Ports
Primitive String Types
Process-Wide and Place-Wide Registration
*ProgID*
progid->clsid
prop:cpointer
provide-protected
ptr-add
ptr-add!
ptr-equal?
ptr-offset
ptr-ref
ptr-set!
QueryInterface
'raw
Raw Runtime Functions

register-custodian-shutdown
register-finalizer
register-finalizer-and-custodian-shutdown
register-process-global
Release
releaser
Reliable Release of Resources
Remote COM servers (DCOM)
retainer
*retainer*
s16vector
s16vector->cpointer
s16vector->list
s16vector-length
s16vector-ref
s16vector-set!
s16vector?
s32vector
s32vector->cpointer
s32vector->list
s32vector-length
s32vector-ref
s32vector-set!
s32vector?
s64vector
s64vector->cpointer
s64vector->list
s64vector-length
s64vector-ref
s64vector-set!
s64vector?
s8vector
s8vector->cpointer
s8vector->list
s8vector-length
s8vector-ref
s8vector-set!
s8vector?
Safe C Vectors
Safe Homogenous Vectors
saved-errno
security-guard-check-file
security-guard-check-file-link
security-guard-check-network
sel_registerName
selector
self
Serializable C Struct Types
set-cpointer-tag!
set-ffi-obj!
set-ivar!
set-ptr-offset!
shared libraries
shared objects
Speculatively Atomic Execution
start-atomic
start-breakable-atomic
start-uninterruptible
Static Callout and Callback Cores
String Types
string->clsid
string->guid
string->iid
Strings
'stubborn
super-tell
Syntactic Forms and Procedures
SysAllocStringLen
SysFreeString
'tagged
Tagged C Pointer Types
tell
tellv
The Racket Foreign Interface
Thread Scheduling
Threads and Places
try-atomic
Type Constructors
type-describe
type-described-description
type-described-value
type-described?
type-description?

u16vector
u16vector->cpointer
u16vector->list
u16vector-length
u16vector-ref
u16vector-set!
u16vector?
u32vector
u32vector->cpointer
u32vector->list
u32vector-length
u32vector-ref
u32vector-set!
u32vector?
u64vector
u64vector->cpointer
u64vector->list
u64vector-length
u64vector-ref
u64vector-set!
u64vector?
u8vector
u8vector->cpointer
u8vector->list
u8vector-length
u8vector-ref
u8vector-set!
u8vector?
'uncollectable
Unexported Primitive Functions
*uninterruptible lock*
*uninterruptible mode*
uninterruptible-lock-acquire
uninterruptible-lock-release
union-ptr
union-ref
union-set!
union?
unregister-custodian-shutdown
unsafe-add-collect-callbacks
unsafe-fd->evt
unsafe-file-descriptor->port
unsafe-file-descriptor->semaphore
unsafe-make-signal-received
unsafe-poll-ctx-eventmask-wakeup
unsafe-poll-ctx-fd-wakeup
unsafe-poll-ctx-milliseconds-
wakeup
unsafe-poll-fd
unsafe-poller
unsafe-port->file-descriptor
unsafe-port->socket
unsafe-remove-collect-callbacks
unsafe-set-sleep-in-thread!
unsafe-signal-received
unsafe-socket->port
unsafe-socket->semaphore
Variable Auto-Converting String Type
vector->cblock
vector->cpointer
Virtual Machine Primitives
vm-eval
vm-primitive
void/reference-sink
wchar_t
win64?
winapi
Windows API Helpers
Windows VCRuntime Librraies
windows-error
with-autorelease
with-blocking-tell
YES
'zeroed-atomic
'zeroed-atomic-interior