Package 'ring'

April 24, 2017

Title Circular / Ring Buffers

Version 1.0.0

Description Circular / ring buffers in R and C. There are a couple of different buffers here with different implementations that represent different trade-offs.

License MIT + file LICENSE

LazyData true

URL https://github.com/richfitz/ring

BugReports https://github.com/richfitz/ring/issues

Imports R6

Suggests knitr, rmarkdown, testthat

RoxygenNote 5.0.1

VignetteBuilder knitr

NeedsCompilation yes

Author Rich FitzJohn [aut, cre]

Maintainer Rich FitzJohn <rich.fitzjohn@gmail.com>

Repository CRAN

Date/Publication 2017-04-24 14:51:02 UTC

R topics documented:

`ring_buffer_bytes` *Byte array based ring buffer*

Description

Construct a ring buffer where the buffer holds a stream of bytes. Optionally, the buffer can be "strided" so that the bytes naturally fall into chunks of exactly the same size. It is implemented in C in the hope that it will be fast, with the limitation that any data transfer to or from R will always involve copies.

Usage

```
ring_buffer_bytes(size, stride = 1L, on_overflow = "overwrite")
```

Arguments

`size`	Number of elements in the buffer, each of which will be `stride` bytes long.
`stride`	Number of bytes per buffer element. Defaults to 1 byte. If you want to store anything other than a bytestream in the buffer, you will probably want more than one byte per element; for example, on most R platforms an integer takes 4 bytes and a double takes 8 (see `.Machine`, and also `ring_buffer_bytes_typed`).
`on_overflow`	Behaviour on buffer overflow. The default is to overwrite the oldest elements in the buffer (`"overwrite"`). Alternative actions are `"error"` which will throw an error if a function tries to add more elements than there are space for, or `"grow"` which will grow the buffer to accept the new elements (this uses an approximately golden ratio approach; see details below).

Details

In contrast with `ring_buffer_env`, every element of this buffer has the same size; this makes it less flexible (because you have to decide ahead of time what you will be storing), but at the same time this can make using the buffer easier to think about (because you decided ahead of time what you are storing).

If you want to use this to store fixed-size arrays of integers, numerics, etc, see `ring_buffer_bytes_typed` which wraps this with fast conversion functions.

If the `on_overflow` action is `"grow"` and the buffer overflows, then the size of the buffer will grow geometrically (this is also the case if you manually `$grow()` the buffer with `exact = FALSE`). When used this way, let `n` is the number of *additional* elements that space is needed for; `ring` then looks at the total needed capacity (used plus `n` relative to `size()`). *If* the buffer needs to be made larger to fit `n` elements in then it is grown by a factor of phi (the golden ratio, approximately 1.6). So if to fit `n` elements in the buffer needs to be increased in size by `m` then the smallest of `size * phi`, `size * phi^2`, `size * phi^3`, ... will be used as the new size.

In contrast, using the `grow()` method with `exact = TRUE` will *always* increase the size of the buffer so long as `n` is positive.

Methods

Note that this methods reference section is repeated verbatim between the three main ring buffer classes; `ring_buffer_env` ("env"), `ring_buffer_bytes` ("bytes") and `ring_buffer_bytes_typed` ("typed"). Almost all methods have the same arguments and behaviour, but hopefully by listing everything together, the differences between implementations will be a bit more apparent.

`reset` Reset the state of the buffer. This "zeros" the head and tail pointer (and may or may not actually reset the data) so that the buffer can be used as if fresh.

Usage: `reset(clear = FALSE)`

Arguments:

- `clear`: Logical, indicating if the memory should also be cleared. Generally this is not necessary, but with environment buffers this can let the garbage collector clean up large elements. For the bytes buffer this zeros the memory.

Return value: Nothing; called for the side effect only. &side_effect

`duplicate` Clone the ring buffer, creating a copy. Copies both the underlying data and the position of the head and tail.

Usage: `duplicate()`

Return value: A new ring buffer object

`grow` Increase the size of the buffer by `n` elements.

Usage:

- bytes, typed: `grow(n)`
- env: `grow(n, exact = FALSE)`

Arguments:

- `n`: The number of additional elements that space should be reserved for (scalar non-negative integer).
- `exact`: (For bytes buffer only) Logical scalar indicating if growth should increase the size by *exactly* `n` elements (if `TRUE`) or so that *at least* `n` additional elements will fit (growing the buffer geometrically if needed).

Return value: _yaml.bad-anchor_

`size` Return the capacity (maximum size) of the ring buffer

Usage:

- env: `size()`
- bytes, typed: `size(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`bytes_data` Return the total size of the data storage used in this object.

Usage:

- env: *(not supported)*
- bytes, typed: `bytes_data()`

Return value: A scalar integer

`stride` Length of each element in the ring buffer, in bytes. Only implemented (and meaningful) for the bytes buffer; the environment buffer does not support this function as it makes no sense there.

Usage:

- env: *(not supported)*
- bytes, typed: `stride()`

Return value: A scalar integer

`used` Return the amount of space used in the ring buffer.

Usage:

- env: `used()`
- bytes, typed: `used(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`free` Return the amount of space free in the ring buffer.

Usage:

- env: `free()`
- bytes, typed: `free(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`is_empty` Test if the ring buffer is empty

Usage: `is_empty()`

Return value: A scalar logical

`is_full` Test if the ring buffer is full

Usage: `is_full()`

Return value: A scalar logical

`head_pos` Return the number of entries from the "start" of the ring buffer the head is. This is mostly useful for debugging.

Usage:

- env: `head_pos()`
- bytes, typed: `head_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`tail_pos` Return the number of entries from the "start" of the ring buffer the tail is. This is mostly useful for debugging.

Usage:

- env: `tail_pos()`
- bytes, typed: `tail_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`head` Return the contents of the head (the most recently written element in the ring buffer).

Usage: `head()`

Return value: It depends a little here. For `ring_buffer_env` this is a single R object. For `ring_buffer_bytes` it is a raw vector, the same length as the stride of the ring buffer. For `ring_buffer_bytes_typed`, a single R object that has been translated from raw.

`tail` Return the contents of the tail (the least recently written element in the ring buffer).

Usage: `tail()`

Return value: As for `head`

`set` Set a number of ring entries to the same value. The exact behaviour here varies depending on the type of ring buffer. This function may overflow the ring buffer; in this case the tail will be moved.

Usage: `set(data, n)`

Arguments:

- `data`: The data to set each ring element to. For an environment buffer, this may be any R object. For a bytes buffer it may be either a single byte (in which case each ring element will be set to that byte, repeated `stride` times), or a raw vector of length `stride`.
- `n`: The number of entries to set to `data`

Return value: Invisibly returns the number of elements actually written (which may be less than `n` if the buffer overflows). Primarily called for its side effect.

`push` Push elements onto the ring buffer head. This may overflow the ring buffer, destroying the oldest elememnts in the buffer (and moving the position of the tail).

Usage:

- env: `push(data, iterate = TRUE)`
- bytes, typed: `push(data)`

Arguments:

- `data`: Data to push onto the ring buffer. For `ring_buffer_bytes`, this must be a raw vector with a length that is a multiple of the buffer stride. For `ring_buffer_bytes_typed` it must be a vector of the appropriate type. For `ring_buffer_env` it may be an arbitrary R object (but see `iterate` .
- `iterate`: For `ring_buffer_env` only, changes the behaviour with vectors and lists. Because each element of a `ring_buffer_env` can b an arbitrary R object, for a list `x` it is ambiguous if `push(x)` should push one object onto the buffer, or `length(x)` objects (i.e. equivalent to `push(x[[1]])`, `push(x[[2]])`, etc. The `iterate` argument switches between interpretations; if `TRUE` (the default) the push will iterate over the object using `for (el in x)` (with appropriate S3 dispatch). If `iterate = FALSE`, then the entire object is pushed at once, so always updating only by a single element.

Return value: For `ring_buffer_bytes`, the data invisibly. For `ring_buffer_bytes` and `ring_buffer_bytes_typed`, the position of the head pointer (relative to the beginning of the storage region).

`take` Destructively take elements from the ring buffer. This consumes from the tail (the least recently added elements). It is not possibly to underflow the buffer; if more elements are requested than can be supplied then an error will be thrown and the state of the buffer unmodified.

Usage: `take(n)`

Arguments:

- `n`: The number of elements to take.

Return value: For `ring_buffer_env` a `list` of `n` elements. For `ring_buffer_bytes`, a raw vector of `n * stride` bytes. For `ring_buffer_bytes_typed`, an vector of `n` elements of the storage mode of the ring.

`read` Nondestructively read elements from the ring buffer. This is identical to `take` except that the state of the buffer is not modified.

Usage: `read(n)`

Arguments:

- `n`: The number of elements to read.

Return value: For `ring_buffer_env` a `list` of `n` elements. For `ring_buffer_bytes`, a raw vector of `n * stride` bytes. For `ring_buffer_bytes_typed`, an vector of `n` elements of the storage mode of the ring.

`copy` Copy from *this* ring buffer into a different ring buffer. This is destructive with respect to both ring buffers; the tail pointer will be moved in this ring buffer as data are taken, and if the destination ring buffer overflows, the tail pointer will be moved too.

Usage: `copy(dest, n)`

Arguments:

- `dest`: The destination ring buffer - will be modified by this call.
- `n`: The number of elements to copy

`mirror` Mirror the contents of *this* ring buffer into a different ring buffer. This differs from `copy` in that *this* ring buffer is unaffected and in that *all* of this ring buffer is copied over (including head/tail positions). This provides an alternative way of duplicating state to `duplicate` if you already have an appropriately sized ring buffer handy. No allocations will be done.

Usage: `mirror(dest)`

Arguments:

- `dest`: The destination ring buffer - will be modified by this call.

Return value: _yaml.bad-anchor_

`head_offset` Nondestructively read the contents of the `head` of the buffer, offset by `n` entries.

Usage: `head_offset(n)`

Arguments:

- `n`: Head offset. This moves away from the most recently added item. An offset of 0 reads the most recently added element, 1 reads the element added before that.

Return value: As for `head`

`tail_offset` Nondestructively read the contents of the `tail` of the buffer, offset by `n` entries.

Usage: `tail_offset(n)`

Arguments:

- `n`: Tail offset. This moves away from the oldest item. An offset of 0 reads the oldest element, 1 reads the element added after that.

Return value: As for `tail` (see `head`)

`take_head` As for `take`, but operating on the head rather than the tail. This is destructive with respect to the head.

Usage: `take_head(n)`

Arguments:

- `n`: Number of elements to take.

Return value: As for `take`

`read_head` As for `read`, but operating on the head rather than the tail. This is not destructive with respect to the tail.

Usage: `read_head(n)`

Arguments:

- `n`: Number of elements to read.

Return value: As for `read`

`head_set` Set data to the head *without advancing*. This is useful in cases where the head data will be set and advanced seprately (with `head_advance`). This is unlikely to be useful for all users. It is used extensively in dde (but called from C).

Usage: `head_set(data)`

Arguments:

- `data`: Data to set into the head. For the bytes buffer this must be exactly `stride` bytes long, and for the environment buffer it corresponds to a single "element".

Return value: _yaml.bad-anchor_

`head_data` Retrieve the current data stored in the head but not advanced. For many cases this may be junk - if the byte buffer has looped then it will be the bytes that will be overwritten on the next write. However, when using `head_set` it will be the data that have been set into the buffer but not yet committed with `head_advance`.

Usage: `head_data()`

Return value: As for `head`

`head_advance` Shift the head around one position. This commits any data written by `head_set`.

Usage: `head_advance()`

Return value: _yaml.bad-anchor_

Examples

```
# Create a ring buffer of 100 bytes
b <- ring_buffer_bytes(100)

# Get the length, number of used and number of free bytes:
b$size()
```

```
b$used()
b$free()

# Nothing is used because we're empty:
b$is_empty()

# To work with a bytes buffer you need to use R's raw vectors;
# here are 30 random bytes:
bytes <- as.raw(as.integer(sample(256, 30, TRUE) - 1L))
bytes

# Push these onto the bytes buffer:
b$push(bytes)
b$used()

# The head of the buffer points at the most recently added item
b$head()
bytes[[length(bytes)]]

# ...and the tail at the oldest (first added in this case)
b$tail()
bytes[[1]]

# Elements are taken from the tail; these will be the oldest items:
b$take(8)
bytes[1:8]
b$used()

# To read from the buffer without removing elements, use read:
b$read(8)
bytes[9:16]

# It is not possible to take or read more elements than are
# present in the buffer; it will throw an error:
## Not run:
b$read(50) # error because there are only 22 bytes present

## End(Not run)

# More elements can be pushed on:
b$push(as.raw(rep(0, 50)))
b$used()
b$read(b$used())

# If many new elements are added, they will displace the old elements:
b$push(as.raw(1:75))
b$read(b$used())
```

---

ring_buffer_bytes_translate

Translating bytes ring buffer

Description

This ring buffer is based on ring_buffer_bytes but performs conversion to/from bytes to something useful as data is stored/retrieved from the buffer. This is the interface through which ring_buffer_bytes_typed is implemented.

Usage

```
ring_buffer_bytes_translate(size, stride, to, from, on_overflow = "overwrite")
```

Arguments

size	Number of elements in the buffer, each of which will be stride bytes long.
stride	Number of bytes per buffer element. Defaults to 1 byte. If you want to store anything other than a bytestream in the buffer, you will probably want more than one byte per element; for example, on most R platforms an integer takes 4 bytes and a double takes 8 (see .Machine, and also ring_buffer_bytes_typed).
to	Function to convert an R object to a set of exactly stride bytes. It must take one argument (being an R object) and return a raw vector of a length that is a multiple of stride (including zero). It may throw an error if it is not possible to convert an object to a bytes vector.
from	Function to convert a set of bytes to an R object. It must take one argument (being a raw vector of a length that is a multiple of stride, including zero). It should not throw an error as all data added to the buffer will have passed through to on the way in to the buffer.
on_overflow	Behaviour on buffer overflow. The default is to overwrite the oldest elements in the buffer ("overwrite"). Alternative actions are "error" which will throw an error if a function tries to add more elements than there are space for, or "grow" which will grow the buffer to accept the new elements (this uses an approximately golden ratio approach; see details below).

Details

The idea here is that manually working with raw vectors can get tedious, and if you are planning on using a bytes-based buffer while working in R you may have a way of doing converstion from and to R objects. This interface lets you specify the functions once and then will apply your conversion function in every case where they are needed.

Methods

Note that this methods reference section is repeated verbatim between the three main ring buffer classes; ring_buffer_env ("env"), ring_buffer_bytes ("bytes") and ring_buffer_bytes_typed ("typed"). Almost all methods have the same arguments and behaviour, but hopefully by listing everything together, the differences between implementations will be a bit more apparent.

reset Reset the state of the buffer. This "zeros" the head and tail pointer (and may or may not actually reset the data) so that the buffer can be used as if fresh.

Usage: `reset(clear = FALSE)`

Arguments:

- `clear`: Logical, indicating if the memory should also be cleared. Generally this is not necessary, but with environment buffers this can let the garbage collector clean up large elements. For the bytes buffer this zeros the memory.

Return value: Nothing; called for the side effect only. &side_effect

`duplicate` Clone the ring buffer, creating a copy. Copies both the underlying data and the position of the head and tail.

Usage: `duplicate()`

Return value: A new ring buffer object

`grow` Increase the size of the buffer by `n` elements.

Usage:

- bytes, typed: `grow(n)`
- env: `grow(n, exact = FALSE)`

Arguments:

- `n`: The number of additional elements that space should be reserved for (scalar non-negative integer).
- `exact`: (For bytes buffer only) Logical scalar indicating if growth should increase the size by *exactly* `n` elements (if `TRUE`) or so that *at least* `n` additional elements will fit (growing the buffer geometrically if needed).

Return value: _yaml.bad-anchor_

`size` Return the capacity (maximum size) of the ring buffer

Usage:

- env: `size()`
- bytes, typed: `size(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`bytes_data` Return the total size of the data storage used in this object.

Usage:

- env: *(not supported)*
- bytes, typed: `bytes_data()`

Return value: A scalar integer

`stride` Length of each element in the ring buffer, in bytes. Only implemented (and meaningful) for the bytes buffer; the environment buffer does not support this function as it makes no sense there.

Usage:

- env: *(not supported)*
- bytes, typed: `stride()`

Return value: A scalar integer

`used` Return the amount of space used in the ring buffer.

Usage:

- env: `used()`
- bytes, typed: `used(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`free` Return the amount of space free in the ring buffer.

Usage:

- env: `free()`
- bytes, typed: `free(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`is_empty` Test if the ring buffer is empty

Usage: `is_empty()`

Return value: A scalar logical

`is_full` Test if the ring buffer is full

Usage: `is_full()`

Return value: A scalar logical

`head_pos` Return the number of entries from the "start" of the ring buffer the head is. This is mostly useful for debugging.

Usage:

- env: `head_pos()`
- bytes, typed: `head_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`tail_pos` Return the number of entries from the "start" of the ring buffer the tail is. This is mostly useful for debugging.

Usage:

- env: `tail_pos()`
- bytes, typed: `tail_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`head` Return the contents of the head (the most recently written element in the ring buffer).

Usage: `head()`

Return value: It depends a little here. For `ring_buffer_env` this is a single R object. For `ring_buffer_bytes` it is a raw vector, the same length as the stride of the ring buffer. For `ring_buffer_bytes_typed`, a single R object that has been translated from raw.

`tail` Return the contents of the tail (the least recently written element in the ring buffer).

Usage: `tail()`

Return value: As for `head`

`set` Set a number of ring entries to the same value. The exact behaviour here varies depending on the type of ring buffer. This function may overflow the ring buffer; in this case the tail will be moved.

Usage: `set(data, n)`

Arguments:

- `data`: The data to set each ring element to. For an environment buffer, this may be any R object. For a bytes buffer it may be either a single byte (in which case each ring element will be set to that byte, repeated `stride` times), or a raw vector of length `stride`.
- `n`: The number of entries to set to `data`

Return value: Invisibly returns the number of elements actually written (which may be less than `n` if the buffer overflows). Primarily called for its side effect.

`push` Push elements onto the ring buffer head. This may overflow the ring buffer, destroying the oldest elememnts in the buffer (and moving the position of the tail).

Usage:

- env: `push(data, iterate = TRUE)`
- bytes, typed: `push(data)`

Arguments:

- `data`: Data to push onto the ring buffer. For `ring_buffer_bytes`, this must be a raw vector with a length that is a multiple of the buffer stride. For `ring_buffer_bytes_typed` it must be a vector of the appropriate type. For `ring_buffer_env` it may be an arbitrary R object (but see `iterate` .
- `iterate`: For `ring_buffer_env` only, changes the behaviour with vectors and lists. Because each element of a `ring_buffer_env` can b an arbitrary R object, for a list `x` it is ambiguous if `push(x)` should push one object onto the buffer, or `length(x)` objects (i.e. equivalent to `push(x[[1]])`, `push(x[[2]])`, etc. The `iterate` argument switches between interpretations; if `TRUE` (the default) the push will iterate over the object using `for (el in x)` (with appropriate S3 dispatch). If `iterate = FALSE`, then the entire object is pushed at once, so always updating only by a single element.

Return value: For `ring_buffer_bytes`, the data invisibly. For `ring_buffer_bytes` and `ring_buffer_bytes_typed`, the position of the head pointer (relative to the beginning of the storage region).

`take` Destructively take elements from the ring buffer. This consumes from the tail (the least recently added elements). It is not possibly to underflow the buffer; if more elements are requested than can be supplied then an error will be thrown and the state of the buffer unmodified.

Usage: `take(n)`

Arguments:

- n: The number of elements to take.

Return value: For ring_buffer_env a list of n elements. For ring_buffer_bytes, a raw vector of n * stride bytes. For ring_buffer_bytes_typed, an vector of n elements of the storage mode of the ring.

read Nondestructively read elements from the ring buffer. This is identical to take except that the state of the buffer is not modified.

Usage: read(n)

Arguments:

- n: The number of elements to read.

Return value: For ring_buffer_env a list of n elements. For ring_buffer_bytes, a raw vector of n * stride bytes. For ring_buffer_bytes_typed, an vector of n elements of the storage mode of the ring.

copy Copy from *this* ring buffer into a different ring buffer. This is destructive with respect to both ring buffers; the tail pointer will be moved in this ring buffer as data are taken, and if the destination ring buffer overflows, the tail pointer will be moved too.

Usage: copy(dest, n)

Arguments:

- dest: The destination ring buffer - will be modified by this call.
- n: The number of elements to copy

mirror Mirror the contents of *this* ring buffer into a different ring buffer. This differs from copy in that *this* ring buffer is unaffected and in that *all* of this ring buffer is copied over (including head/tail positions). This provides an alternative way of duplicating state to duplicate if you already have an appropriately sized ring buffer handy. No allocations will be done.

Usage: mirror(dest)

Arguments:

- dest: The destination ring buffer - will be modified by this call.

Return value: _yaml.bad-anchor_

head_offset Nondestructively read the contents of the head of the buffer, offset by n entries.

Usage: head_offset(n)

Arguments:

- n: Head offset. This moves away from the most recently added item. An offset of 0 reads the most recently added element, 1 reads the element added before that.

Return value: As for head

tail_offset Nondestructively read the contents of the tail of the buffer, offset by n entries.

Usage: tail_offset(n)

Arguments:

- n: Tail offset. This moves away from the oldest item. An offset of 0 reads the oldest element, 1 reads the element added after that.

Return value: As for tail (see head)

take_head As for take, but operating on the head rather than the tail. This is destructive with respect to the head.

Usage: take_head(n)

Arguments:

- n: Number of elements to take.

Return value: As for take

read_head As for read, but operating on the head rather than the tail. This is not destructive with respect to the tail.

Usage: read_head(n)

Arguments:

- n: Number of elements to read.

Return value: As for read

head_set Set data to the head *without advancing*. This is useful in cases where the head data will be set and advanced seprately (with head_advance). This is unlikely to be useful for all users. It is used extensively in dde (but called from C).

Usage: head_set(data)

Arguments:

- data: Data to set into the head. For the bytes buffer this must be exactly stride bytes long, and for the environment buffer it corresponds to a single "element".

Return value: _yaml.bad-anchor_

head_data Retrieve the current data stored in the head but not advanced. For many cases this may be junk - if the byte buffer has looped then it will be the bytes that will be overwritten on the next write. However, when using head_set it will be the data that have been set into the buffer but not yet committed with head_advance.

Usage: head_data()

Return value: As for head

head_advance Shift the head around one position. This commits any data written by head_set.

Usage: head_advance()

Return value: _yaml.bad-anchor_

Author(s)

Rich FitzJohn

Examples

```
# The "typed" ring buffers do not allow for character vectors to
# be stored, because strings are generally hard and have unknown
# lengths.  But if you wanted to store strings that are *always*
# the same length, this is straightforward to do.

# You can convert from string to bytes with charToRaw (or
# as.raw(utf8ToInt(x))):
bytes <- charToRaw("hello!")
bytes

# And back again with rawToChar (or intToUtf8(as.integer(x)))
rawToChar(bytes)

# So with these functions we can make a buffer for storing
```

```
# fixed-length strings:
b <- ring_buffer_bytes_translate(100, 8, charToRaw, rawToChar)

# And with this we can store 8 character strings:
b$push("abcdefgh")
b$tail()

# Other length strings cannot be added:
try(
  b$push("hello!")
) # error

# Because the 'from' and 'to' arguments can be arbitrary R
# functions we could tweak this to pad the character vector with
# null bytes, and strip these off on return:
char_to_raw <- function(x, max_len) {
  if (!(is.character(x) && length(x) == 1L)) {
    stop("Expected a single string")
  }
  n <- nchar(x)
  if (n > max_len) {
    stop("String is too long")
  }
  c(charToRaw(x), rep(raw(1), max_len - n))
}
char_from_raw <- function(x) {
  rawToChar(x[x != raw(1)])
}

# Because max_len is the same thing as stride, wrap this all up a
# little:
char_buffer <- function(size, max_len) {
  to <- function(x) char_to_raw(x, max_len)
  ring_buffer_bytes_translate(size, max_len, to, char_from_raw)
}

b <- char_buffer(100, 30) # 100 elements of up to 30 characters each
b$push("x")
b$tail()

b$push("hello world!")
b$head()

try(
  b$push("supercalafragalisticexpealadocious")
) # error: string is too long
```

---

ring_buffer_bytes_typed

Typed bytes ring buffer

Description

Create a ring buffer, backed by a `ring_buffer_bytes`, where each element corresponds to a fixed-size vector of one of R's atomic numeric types (logical, integer, double, and complex).

Usage

```
ring_buffer_bytes_typed(size, what, len = NULL, on_overflow = "overwrite")
```

Arguments

`size`	The maximum number of elements the buffer can hold. Each element will be multiple bytes long.
`what`	Either a vector on the style of `vapply` (e.g., `integer(4)` to indicate that each element of the buffer is a 4-element integer, or the `name` of a storage mode if `len` is also provided.
`len`	If given, then the length of the storage. If it is given, then if `length(what)` is zero, the storage mode of `what` is used as the type. Otherwise `what` is interpreted as the *name* of the storage mode (one of "logical", "integer", "double" or "complex".
`on_overflow`	Behaviour on buffer overflow. The default is to overwrite the oldest elements in the buffer (`"overwrite"`). Alternative actions are `"error"` which will throw an error if a function tries to add more elements than there are space for, or `"grow"` which will grow the buffer to accept the new elements (this uses an approximately golden ratio approach; see details below).

Details

Note that a logical ring buffer and an integer ring buffer take the same number of bytes because a logical vector is stored as an integer (4 bytes per element) to deal with missing values; see "writing R extensions".

Note that it is not possible to store character vectors in a ring buffer of this type because each element of a character vector can be any number of bytes.

Methods

Note that this methods reference section is repeated verbatim between the three main ring buffer classes; `ring_buffer_env` ("env"), `ring_buffer_bytes` ("bytes") and `ring_buffer_bytes_typed` ("typed"). Almost all methods have the same arguments and behaviour, but hopefully by listing everything together, the differences between implementations will be a bit more apparent.

`reset` Reset the state of the buffer. This "zeros" the head and tail pointer (and may or may not actually reset the data) so that the buffer can be used as if fresh.

Usage: `reset(clear = FALSE)`

Arguments:

- `clear`: Logical, indicating if the memory should also be cleared. Generally this is not necessary, but with environment buffers this can let the garbage collector clean up large elements. For the bytes buffer this zeros the memory.

Return value: Nothing; called for the side effect only. &side_effect

`duplicate` Clone the ring buffer, creating a copy. Copies both the underlying data and the position of the head and tail.

Usage: `duplicate()`

Return value: A new ring buffer object

`grow` Increase the size of the buffer by `n` elements.

Usage:

- bytes, typed: `grow(n)`
- env: `grow(n, exact = FALSE)`

Arguments:

- `n`: The number of additional elements that space should be reserved for (scalar non-negative integer).
- `exact`: (For bytes buffer only) Logical scalar indicating if growth should increase the size by *exactly* `n` elements (if `TRUE`) or so that *at least* `n` additional elements will fit (growing the buffer geometrically if needed).

Return value: _yaml.bad-anchor_

`size` Return the capacity (maximum size) of the ring buffer

Usage:

- env: `size()`
- bytes, typed: `size(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`bytes_data` Return the total size of the data storage used in this object.

Usage:

- env: *(not supported)*
- bytes, typed: `bytes_data()`

Return value: A scalar integer

`stride` Length of each element in the ring buffer, in bytes. Only implemented (and meaningful) for the bytes buffer; the environment buffer does not support this function as it makes no sense there.

Usage:

- env: *(not supported)*
- bytes, typed: `stride()`

Return value: A scalar integer

`used` Return the amount of space used in the ring buffer.

Usage:

- env: `used()`
- bytes, typed: `used(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`free` Return the amount of space free in the ring buffer.

Usage:

- env: `free()`
- bytes, typed: `free(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`is_empty` Test if the ring buffer is empty

Usage: `is_empty()`

Return value: A scalar logical

`is_full` Test if the ring buffer is full

Usage: `is_full()`

Return value: A scalar logical

`head_pos` Return the number of entries from the "start" of the ring buffer the head is. This is mostly useful for debugging.

Usage:

- env: `head_pos()`
- bytes, typed: `head_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`tail_pos` Return the number of entries from the "start" of the ring buffer the tail is. This is mostly useful for debugging.

Usage:

- env: `tail_pos()`
- bytes, typed: `tail_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`head` Return the contents of the head (the most recently written element in the ring buffer).

Usage: `head()`

Return value: It depends a little here. For `ring_buffer_env` this is a single R object. For `ring_buffer_bytes` it is a raw vector, the same length as the stride of the ring buffer. For `ring_buffer_bytes_typed`, a single R object that has been translated from raw.

`tail` Return the contents of the tail (the least recently written element in the ring buffer).

Usage: `tail()`

Return value: As for `head`

`set` Set a number of ring entries to the same value. The exact behaviour here varies depending on the type of ring buffer. This function may overflow the ring buffer; in this case the tail will be moved.

Usage: `set(data, n)`

Arguments:

- `data`: The data to set each ring element to. For an environment buffer, this may be any R object. For a bytes buffer it may be either a single byte (in which case each ring element will be set to that byte, repeated `stride` times), or a raw vector of length `stride`.
- `n`: The number of entries to set to `data`

Return value: Invisibly returns the number of elements actually written (which may be less than `n` if the buffer overflows). Primarily called for its side effect.

`push` Push elements onto the ring buffer head. This may overflow the ring buffer, destroying the oldest elememnts in the buffer (and moving the position of the tail).

Usage:

- env: `push(data, iterate = TRUE)`
- bytes, typed: `push(data)`

Arguments:

- `data`: Data to push onto the ring buffer. For `ring_buffer_bytes`, this must be a raw vector with a length that is a multiple of the buffer stride. For `ring_buffer_bytes_typed` it must be a vector of the appropriate type. For `ring_buffer_env` it may be an arbitrary R object (but see `iterate` .
- `iterate`: For `ring_buffer_env` only, changes the behaviour with vectors and lists. Because each element of a `ring_buffer_env` can b an arbitrary R object, for a list `x` it is ambiguous if `push(x)` should push one object onto the buffer, or `length(x)` objects (i.e. equivalent to `push(x[[1]])`, `push(x[[2]])`, etc. The `iterate` argument switches between interpretations; if `TRUE` (the default) the push will iterate over the object using `for (el in x)` (with appropriate S3 dispatch). If `iterate = FALSE`, then the entire object is pushed at once, so always updating only by a single element.

Return value: For `ring_buffer_bytes`, the data invisibly. For `ring_buffer_bytes` and `ring_buffer_bytes_typed`, the position of the head pointer (relative to the beginning of the storage region).

`take` Destructively take elements from the ring buffer. This consumes from the tail (the least recently added elements). It is not possibly to underflow the buffer; if more elements are requested than can be supplied then an error will be thrown and the state of the buffer unmodified.

Usage: `take(n)`

Arguments:

- `n`: The number of elements to take.

Return value: For `ring_buffer_env` a `list` of `n` elements. For `ring_buffer_bytes`, a raw vector of `n * stride` bytes. For `ring_buffer_bytes_typed`, an vector of `n` elements of the storage mode of the ring.

`read` Nondestructively read elements from the ring buffer. This is identical to `take` except that the state of the buffer is not modified.

Usage: `read(n)`

Arguments:

- `n`: The number of elements to read.

Return value: For `ring_buffer_env` a `list` of `n` elements. For `ring_buffer_bytes`, a raw vector of `n * stride` bytes. For `ring_buffer_bytes_typed`, an vector of `n` elements of the storage mode of the ring.

`copy` Copy from *this* ring buffer into a different ring buffer. This is destructive with respect to both ring buffers; the tail pointer will be moved in this ring buffer as data are taken, and if the destination ring buffer overflows, the tail pointer will be moved too.

Usage: `copy(dest, n)`

Arguments:

- `dest`: The destination ring buffer - will be modified by this call.
- `n`: The number of elements to copy

`mirror` Mirror the contents of *this* ring buffer into a different ring buffer. This differs from `copy` in that *this* ring buffer is unaffected and in that *all* of this ring buffer is copied over (including head/tail positions). This provides an alternative way of duplicating state to `duplicate` if you already have an appropriately sized ring buffer handy. No allocations will be done.

Usage: `mirror(dest)`

Arguments:

- `dest`: The destination ring buffer - will be modified by this call.

Return value: _yaml.bad-anchor_

`head_offset` Nondestructively read the contents of the `head` of the buffer, offset by `n` entries.

Usage: `head_offset(n)`

Arguments:

- `n`: Head offset. This moves away from the most recently added item. An offset of 0 reads the most recently added element, 1 reads the element added before that.

Return value: As for `head`

`tail_offset` Nondestructively read the contents of the `tail` of the buffer, offset by `n` entries.

Usage: `tail_offset(n)`

Arguments:

- `n`: Tail offset. This moves away from the oldest item. An offset of 0 reads the oldest element, 1 reads the element added after that.

Return value: As for `tail` (see `head`)

`take_head` As for `take`, but operating on the head rather than the tail. This is destructive with respect to the head.

Usage: `take_head(n)`

Arguments:

- `n`: Number of elements to take.

Return value: As for `take`

`read_head` As for `read`, but operating on the head rather than the tail. This is not destructive with respect to the tail.

Usage: `read_head(n)`

Arguments:

- `n`: Number of elements to read.

Return value: As for `read`

`head_set` Set data to the head *without advancing*. This is useful in cases where the head data will be set and advanced seprately (with `head_advance`). This is unlikely to be useful for all users. It is used extensively in dde (but called from C).

Usage: `head_set(data)`

Arguments:

- `data`: Data to set into the head. For the bytes buffer this must be exactly `stride` bytes long, and for the environment buffer it corresponds to a single "element".

Return value: _yaml.bad-anchor_

`head_data` Retrieve the current data stored in the head but not advanced. For many cases this may be junk - if the byte buffer has looped then it will be the bytes that will be overwritten on the next write. However, when using `head_set` it will be the data that have been set into the buffer but not yet committed with `head_advance`.

Usage: `head_data()`

Return value: As for `head`

`head_advance` Shift the head around one position. This commits any data written by `head_set`.

Usage: `head_advance()`

Return value: _yaml.bad-anchor_

Author(s)

Rich FitzJohn

Examples

```
# Create a ring buffer of 30 integers:
b <- ring_buffer_bytes_typed(30, integer(1))

# Alternatively you can create the same buffer this way:
b <- ring_buffer_bytes_typed(30, "integer", 1)

# The buffer is empty to start with
b$is_empty()

# Note that the buffer has a stride of 4 (see ?ring_buffer_bytes)
b$stride()

# Push some numbers into the buffer:
b$push(as.integer(1:10))

# Report the number of elements used:
b$used()
```

```
# Get the first added element:
b$tail()

# The buffer behaves basically the same way now as
# "ring_buffer_env" but will typecheck all inputs:
## Not run:
  b$push(pi) # error because not an integer
  b$push(1)  # error because not an integer (you must convert to int)

## End(Not run)

# Recycling: the typed buffer operates by converting the input
# vector to a set of bytes and then pushing them onto the buffer;
# this works so long as the vector of bytes has the correct
# length.
b <- ring_buffer_bytes_typed(30, integer(3))

# These both fail because 2 and 4 do not end up as multiples of 3:
## Not run:
  b$push(c(1L, 2L))
  b$push(c(1L, 2L, 3L, 4L))

## End(Not run)

# But this is fine:
b$push(seq_len(6))
b$tail()
b$tail_offset(1)
```

`ring_buffer_env` *Environment-based ring buffer*

Description

An environment based ring buffer. In contrast with `ring_buffer_bytes`, this ring buffer is truely circular, implemented as a doubly linked list that loops back on itself. Each element of the ring buffer can hold an arbitrary R object, and no checking is done to make sure that objects are similar types; in this way they are most similar to a circular version of an R `list`.

Usage

```
ring_buffer_env(size, on_overflow = "overwrite")
```

Arguments

`size`	The (maximum) number of entries the buffer can contain.

on_overflow Behaviour on buffer overflow. The default is to overwrite the oldest elements in the buffer ("overwrite"). Alternative actions are "error" which will throw an error if a function tries to add more elements than there are space for, or "grow" which will grow the buffer to accept the new elements.

Details

When pushing objects onto the buffer, you must be careful about the iterate argument. By default if the object has a length() greater than 1 then $push() will iterate over the object (equivalent to $push(data[[1]],iterate=FALSE), $push(data[[2]], iterate=FALSE), and so on).

For more information and usage examples, see the vignette (vignette("ring")).

On underflow (and overflow if on_overflow = "error") ring will raise custom exceptions that can be caught specially by tryCatch. These will have class ring_underflow (and ring_overflow for overflow). This is not supported in the bytes buffer yet. See the examples for usage.

Methods

Note that this methods reference section is repeated verbatim between the three main ring buffer classes; ring_buffer_env ("env"), ring_buffer_bytes ("bytes") and ring_buffer_bytes_typed ("typed"). Almost all methods have the same arguments and behaviour, but hopefully by listing everything together, the differences between implementations will be a bit more apparent.

reset Reset the state of the buffer. This "zeros" the head and tail pointer (and may or may not actually reset the data) so that the buffer can be used as if fresh.

Usage: reset(clear = FALSE)

Arguments:

- clear: Logical, indicating if the memory should also be cleared. Generally this is not necessary, but with environment buffers this can let the garbage collector clean up large elements. For the bytes buffer this zeros the memory.

Return value: Nothing; called for the side effect only. &side_effect

duplicate Clone the ring buffer, creating a copy. Copies both the underlying data and the position of the head and tail.

Usage: duplicate()

Return value: A new ring buffer object

grow Increase the size of the buffer by n elements.

Usage:

- bytes, typed: grow(n)
- env: grow(n, exact = FALSE)

Arguments:

- n: The number of additional elements that space should be reserved for (scalar non-negative integer).
- exact: (For bytes buffer only) Logical scalar indicating if growth should increase the size by *exactly* n elements (if TRUE) or so that *at least* n additional elements will fit (growing the buffer geometrically if needed).

Return value: _yaml.bad-anchor_

`size` Return the capacity (maximum size) of the ring buffer

Usage:

- env: `size()`
- bytes, typed: `size(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`bytes_data` Return the total size of the data storage used in this object.

Usage:

- env: *(not supported)*
- bytes, typed: `bytes_data()`

Return value: A scalar integer

`stride` Length of each element in the ring buffer, in bytes. Only implemented (and meaningful) for the bytes buffer; the environment buffer does not support this function as it makes no sense there.

Usage:

- env: *(not supported)*
- bytes, typed: `stride()`

Return value: A scalar integer

`used` Return the amount of space used in the ring buffer.

Usage:

- env: `used()`
- bytes, typed: `used(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`free` Return the amount of space free in the ring buffer.

Usage:

- env: `free()`
- bytes, typed: `free(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the size should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`is_empty` Test if the ring buffer is empty

Usage: `is_empty()`

Return value: A scalar logical

`is_full` Test if the ring buffer is full

Usage: `is_full()`

Return value: A scalar logical

`head_pos` Return the number of entries from the "start" of the ring buffer the head is. This is mostly useful for debugging.

Usage:

- env: `head_pos()`
- bytes, typed: `head_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`tail_pos` Return the number of entries from the "start" of the ring buffer the tail is. This is mostly useful for debugging.

Usage:

- env: `tail_pos()`
- bytes, typed: `tail_pos(bytes = FALSE)`

Arguments:

- `bytes`: (for `ring_buffer_bytes` only) Logical, indicating if the position should be returned in bytes (rather than logical entries, which is the default).

Return value: A scalar integer

`head` Return the contents of the head (the most recently written element in the ring buffer).

Usage: `head()`

Return value: It depends a little here. For `ring_buffer_env` this is a single R object. For `ring_buffer_bytes` it is a raw vector, the same length as the stride of the ring buffer. For `ring_buffer_bytes_typed`, a single R object that has been translated from raw.

`tail` Return the contents of the tail (the least recently written element in the ring buffer).

Usage: `tail()`

Return value: As for `head`

`set` Set a number of ring entries to the same value. The exact behaviour here varies depending on the type of ring buffer. This function may overflow the ring buffer; in this case the tail will be moved.

Usage: `set(data, n)`

Arguments:

- `data`: The data to set each ring element to. For an environment buffer, this may be any R object. For a bytes buffer it may be either a single byte (in which case each ring element will be set to that byte, repeated `stride` times), or a raw vector of length `stride`.
- `n`: The number of entries to set to `data`

Return value: Invisibly returns the number of elements actually written (which may be less than `n` if the buffer overflows). Primarily called for its side effect.

`push` Push elements onto the ring buffer head. This may overflow the ring buffer, destroying the oldest elememnts in the buffer (and moving the position of the tail).

Usage:

- env: `push(data, iterate = TRUE)`
- bytes, typed: `push(data)`

Arguments:

- `data`: Data to push onto the ring buffer. For `ring_buffer_bytes`, this must be a raw vector with a length that is a multiple of the buffer stride. For `ring_buffer_bytes_typed` it must be a vector of the appropriate type. For `ring_buffer_env` it may be an arbitrary R object (but see `iterate` .
- `iterate`: For `ring_buffer_env` only, changes the behaviour with vectors and lists. Because each element of a `ring_buffer_env` can b an arbitrary R object, for a list `x` it is ambiguous if `push(x)` should push one object onto the buffer, or `length(x)` objects (i.e. equivalent to `push(x[[1]])`, `push(x[[2]])`, etc. The `iterate` argument switches between interpretations; if `TRUE` (the default) the push will iterate over the object using `for (el in x)` (with appropriate S3 dispatch). If `iterate = FALSE`, then the entire object is pushed at once, so always updating only by a single element.

Return value: For `ring_buffer_bytes`, the data invisibly. For `ring_buffer_bytes` and `ring_buffer_bytes_typed`, the position of the head pointer (relative to the beginning of the storage region).

`take` Destructively take elements from the ring buffer. This consumes from the tail (the least recently added elements). It is not possibly to underflow the buffer; if more elements are requested than can be supplied then an error will be thrown and the state of the buffer unmodified.

Usage: `take(n)`

Arguments:

- `n`: The number of elements to take.

Return value: For `ring_buffer_env` a `list` of `n` elements. For `ring_buffer_bytes`, a raw vector of `n * stride` bytes. For `ring_buffer_bytes_typed`, an vector of `n` elements of the storage mode of the ring.

`read` Nondestructively read elements from the ring buffer. This is identical to `take` except that the state of the buffer is not modified.

Usage: `read(n)`

Arguments:

- `n`: The number of elements to read.

Return value: For `ring_buffer_env` a `list` of `n` elements. For `ring_buffer_bytes`, a raw vector of `n * stride` bytes. For `ring_buffer_bytes_typed`, an vector of `n` elements of the storage mode of the ring.

`copy` Copy from *this* ring buffer into a different ring buffer. This is destructive with respect to both ring buffers; the tail pointer will be moved in this ring buffer as data are taken, and if the destination ring buffer overflows, the tail pointer will be moved too.

Usage: `copy(dest, n)`

Arguments:

- `dest`: The destination ring buffer - will be modified by this call.

- n: The number of elements to copy

`mirror` Mirror the contents of *this* ring buffer into a different ring buffer. This differs from `copy` in that *this* ring buffer is unaffected and in that *all* of this ring buffer is copied over (including head/tail positions). This provides an alternative way of duplicating state to `duplicate` if you already have an appropriately sized ring buffer handy. No allocations will be done.

Usage: `mirror(dest)`

Arguments:

- `dest`: The destination ring buffer - will be modified by this call.

Return value: _yaml.bad-anchor_

`head_offset` Nondestructively read the contents of the `head` of the buffer, offset by `n` entries.

Usage: `head_offset(n)`

Arguments:

- `n`: Head offset. This moves away from the most recently added item. An offset of 0 reads the most recently added element, 1 reads the element added before that.

Return value: As for `head`

`tail_offset` Nondestructively read the contents of the `tail` of the buffer, offset by `n` entries.

Usage: `tail_offset(n)`

Arguments:

- `n`: Tail offset. This moves away from the oldest item. An offset of 0 reads the oldest element, 1 reads the element added after that.

Return value: As for `tail` (see `head`)

`take_head` As for `take`, but operating on the head rather than the tail. This is destructive with respect to the head.

Usage: `take_head(n)`

Arguments:

- `n`: Number of elements to take.

Return value: As for `take`

`read_head` As for `read`, but operating on the head rather than the tail. This is not destructive with respect to the tail.

Usage: `read_head(n)`

Arguments:

- `n`: Number of elements to read.

Return value: As for `read`

`head_set` Set data to the head *without advancing*. This is useful in cases where the head data will be set and advanced seprately (with `head_advance`). This is unlikely to be useful for all users. It is used extensively in dde (but called from C).

Usage: `head_set(data)`

Arguments:

- `data`: Data to set into the head. For the bytes buffer this must be exactly `stride` bytes long, and for the environment buffer it corresponds to a single "element".

Return value: _yaml.bad-anchor_

`head_data` Retrieve the current data stored in the head but not advanced. For many cases this may be junk - if the byte buffer has looped then it will be the bytes that will be overwritten on the next write. However, when using `head_set` it will be the data that have been set into the buffer but not yet committed with `head_advance`.

Usage: `head_data()`

Return value: As for `head`

`head_advance` Shift the head around one position. This commits any data written by `head_set`.

Usage: `head_advance()`

Return value: _yaml.bad-anchor_

Author(s)

Rich FitzJohn

Examples

```
buf <- ring_buffer_env(10)
buf$push(1:10)
buf$take(3)
buf$push(11:15)
buf$take(2)

# The "on_overflow" argument by default allows for the buffer to
# overwrite on overflow.
buf <- ring_buffer_env(10)
buf$push(1:10)
unlist(buf$read(buf$used())) # 1:10
# Over-write the first 5
buf$push(11:15)
unlist(buf$read(buf$used())) # 6:15

# Unlike ring_buffer_bytes, these ring buffers can hold any R
# object.  However, you must be careful about use of iterate!
buf$push(lm(mpg ~ cyl, mtcars), iterate = FALSE)
buf$take(1)

# Alternatively, grow the buffer as overwriting happens
buf <- ring_buffer_env(10, "grow")
buf$push(1:10)
buf$push(11:15)
unlist(buf$read(buf$used())) # 1:15

# Or throw an error on overflow
buf <- ring_buffer_env(10, "error")
buf$push(1:10)
try(buf$push(11:15))

# The errors that are thrown on underflow / overflow are typed so
# can be caught by tryCatch:
tryCatch(buf$read(100),
         ring_underflow = function(e) message("nope"))
```

```
tryCatch(buf$push(100),
         ring_overflow = function(e) message("nope again"))
```

Index