

# Package ‘vctrs’

March 8, 2020

**Title** Vector Helpers

**Version** 0.2.4

**Description** Defines new notions of prototype and size that are used to provide tools for consistent and well-founded type-coercion and size-recycling, and are in turn connected to ideas of type- and size-stability useful for analysing function interfaces.

**License** GPL-3

**URL** <https://github.com/r-lib/vctrs>

**BugReports** <https://github.com/r-lib/vctrs/issues>

**Depends** R (>= 3.2)

**Imports** ellipsis (>= 0.2.0),  
digest,  
glue,  
rlang (>= 0.4.5)

**Suggests** bit64,  
covr,  
crayon,  
generics,  
knitr,  
pillar (>= 1.4.1),  
pkgdown,  
rmarkdown,  
testthat (>= 2.3.0),  
tibble,  
xml2,  
zeallot

**VignetteBuilder** knitr

**Encoding** UTF-8

**Language** en-GB

**LazyData** true

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.0.2

## R topics documented:

internal-faq-ptype2-identity . . . . .	2
list_of . . . . .	4
name_spec . . . . .	5
partial_factor . . . . .	6
partial_frame . . . . .	7
vec_assert . . . . .	7
vec_as_names . . . . .	8
vec_bind . . . . .	10
vec_c . . . . .	13
vec_chop . . . . .	15
vec_compare . . . . .	17
vec_count . . . . .	18
vec_data . . . . .	19
vec_default_cast . . . . .	20
vec_duplicate . . . . .	21
vec_equal . . . . .	22
vec_init . . . . .	23
vec_is_list . . . . .	23
vec_match . . . . .	24
vec_order . . . . .	25
vec_ptype . . . . .	25
vec_recycle . . . . .	27
vec_repeat . . . . .	29
vec_seq_along . . . . .	29
vec_size . . . . .	30
vec_split . . . . .	31
vec_unique . . . . .	32
%0% . . . . .	33

<b>Index</b>	<b>34</b>
--------------	-----------

---

internal-faq-ptype2-identity

*Internal FAQ - vec\_ptype2(), NULL, and unspecified vectors*

---

## Description

### Promotion monoid:

Promotions (i.e. automatic coercions) should always transform inputs to their richer type to avoid losing values of precision. `vec_ptype2()` returns the *richer* type of two vectors, or throws an incompatible type error if none of the two vector types include the other. For example, the richer type of integer and double is the latter because double covers a larger range of values than integer. `vec_ptype2()` is a **monoid** over vectors, which in practical terms means that it is a well behaved operation for **reduction**. Reduction is an important operation for promotions because that is how the richer type of multiple elements is computed. As a monoid, `vec_ptype2()` needs an identity element, i.e. a value that doesn't change the result of the reduction. `vec`s has two identity values, `NULL` and **unspecified** vectors.

**The NULL identity:**

As an identity element that shouldn't influence the determination of the common type of a set of vectors, NULL is promoted to any type:

```
vec_ptype2(NULL, "")
#> character(0)
vec_ptype2(1L, NULL)
#> integer(0)
```

The common type of NULL and NULL is the identity NULL:

```
vec_ptype2(NULL, NULL)
#> NULL
```

This way the result of `vec_ptype2(NULL, NULL)` does not influence subsequent promotions:

```
vec_ptype2(
  vec_ptype2(NULL, NULL),
  ""
)
#> character(0)
```

**Unspecified vectors:**

In the `vctrs` coercion system, logical vectors of missing values are also automatically promoted to the type of any other vector, just like NULL. We call these vectors unspecified. The special coercion semantics of unspecified vectors serve two purposes:

1. It makes it possible to assign vectors of NA inside any type of vectors, even when they are not coercible with logical:
 

```
x <- letters[1:5]
vec_assign(x, 1:2, c(NA, NA))
#> [1] NA NA "c" "d" "e"
```
2. We can't put NULL in a data frame, so we need an identity element that behaves more like a vector. Logical vectors of NA seem a natural fit for this.

Unspecified vectors are thus promoted to any other type, just like NULL:

```
vec_ptype2(NA, "")
#> character(0)
vec_ptype2(1L, c(NA, NA))
#> integer(0)
```

**Finalising common types:**

`vctrs` has an internal vector type of class `vctrs_unspecified`. Users normally don't see such vectors in the wild, but they do come up when taking the common type of an unspecified vector with another identity value:

```
vec_ptype2(NA, NA)
#> <unspecified> [0]
vec_ptype2(NA, NULL)
#> <unspecified> [0]
vec_ptype2(NULL, NA)
#> <unspecified> [0]
```

We can't return NA here because `vec_ptype2()` normally returns empty vectors. We also can't return NULL because unspecified vectors need to be recognised as logical vectors if they haven't been promoted at the end of the reduction.

```
vec_ptype_finalise(vec_ptype2(NULL, NA))
#> logical(0)
```

See the output of `vec_ptype_common()` which performs the reduction and finalises the type, ready to be used by the caller:

```
vec_ptype_common(NULL, NULL)
#> NULL
vec_ptype_common(NA, NULL)
#> logical(0)
```

Note that **partial** types in `vec`rs make use of the same mechanism. They are finalised with `vec_ptype_finalise()`.

---

list_of	list_of S3 class for homogenous lists
---------	---------------------------------------

---

## Description

A `list_of` object is a list where each element has the same type. Modifying the list with `$`, `[`, and `[[` preserves the constraint by coercing all input items.

## Usage

```
list_of(..., .ptype = NULL)

as_list_of(x, ...)

validate_list_of(x)

is_list_of(x)

## S3 method for class 'vctrs_list_of'
vec_ptype2(x, y, ..., x_arg = "x", y_arg = "y")

## S3 method for class 'vctrs_list_of'
vec_cast(x, to, ...)
```

## Arguments

<code>...</code>	Vectors to coerce.
<code>.ptype</code>	If <code>NULL</code> , the default, the output type is determined by computing the common type across all elements of <code>...</code> . Alternatively, you can supply <code>.ptype</code> to give the output known type. If <code>getOption("vctrs.no_guess")</code> is <code>TRUE</code> you must supply this value: this is a convenient way to make production code demand fixed types.
<code>x</code>	For <code>as_list_of()</code> , a vector to be coerced to <code>list_of</code> .
<code>y, to</code>	Arguments to <code>vec_ptype2()</code> and <code>vec_cast()</code> .
<code>x_arg</code>	Argument names for <code>x</code> and <code>y</code> . These are used in error messages to inform the user about the locations of incompatible types (see <a href="#">stop_incompatible_type()</a> ).
<code>y_arg</code>	Argument names for <code>x</code> and <code>y</code> . These are used in error messages to inform the user about the locations of incompatible types (see <a href="#">stop_incompatible_type()</a> ).

## Details

Unlike regular lists, setting a list element to NULL using `[[` does not remove it.

## Examples

```
x <- list_of(1:3, 5:6, 10:15)
if (requireNamespace("tibble", quietly = TRUE)) {
  tibble::tibble(x = x)
}

vec_c(list_of(1, 2), list_of(FALSE, TRUE))
```

---

name_spec	<i>Name specifications</i>
-----------	----------------------------

---

## Description

A name specification describes how to combine an inner and outer names. This sort of name combination arises when concatenating vectors or flattening lists. There are two possible cases:

- Named vector:  

```
vec_c(outer = c(inner1 = 1, inner2 = 2))
```
- Unnamed vector:  

```
vec_c(outer = 1:2)
```

In `r-lib` and tidyverse packages, these cases are errors by default, because there's no behaviour that works well for every case. Instead, you can provide a name specification that describes how to combine the inner and outer names of inputs. Name specifications can refer to:

- `outer`: The external name recycled to the size of the input vector.
- `inner`: Either the names of the input vector, or a sequence of integer from 1 to the size of the vector if it is unnamed.

## Arguments

`name_spec`, `.name_spec`

A name specification for combining inner and outer names. This is relevant for inputs passed with a name, when these inputs are themselves named, like `outer = c(inner = 1)`, or when they have length greater than 1: `outer = 1:2`. By default, these cases trigger an error. You can resolve the error by providing a specification that describes how to combine the names or the indices of the inner vector with the name of the input. This specification can be:

- A function of two arguments. The outer name is passed as a string to the first argument, and the inner names or positions are passed as second argument.
- An anonymous function as a purrr-style formula.
- A glue specification of the form `"{outer}_{inner}"`.

See the [name specification topic](#).

**Examples**

```
# By default, named inputs must be length 1:
vec_c(name = 1)          # ok
try(vec_c(name = 1:3))   # bad

# They also can't have internal names, even if scalar:
try(vec_c(name = c(internal = 1))) # bad

# Pass a name specification to work around this. A specification
# can be a glue string referring to `outer` and `inner`:
vec_c(name = 1:3, other = 4:5, .name_spec = "{outer}")
vec_c(name = 1:3, other = 4:5, .name_spec = "{outer}_{inner}")

# They can also be functions:
my_spec <- function(outer, inner) paste(outer, inner, sep = "_")
vec_c(name = 1:3, other = 4:5, .name_spec = my_spec)

# Or purrr-style formulas for anonymous functions:
vec_c(name = 1:3, other = 4:5, .name_spec = ~ paste0(.x, .y))
```

---

partial\_factor

*Partially specify a factor*


---

**Description**

This special class can be passed as a ptype in order to specify that the result should be a factor that contains at least the specified levels.

**Usage**

```
partial_factor(levels = character())
```

**Arguments**

levels                      Character vector of labels.

**Examples**

```
# Assert that `x` is a factor
vec_assert(factor("x"), partial_factor())

# Testing with `factor()` is too strict,
# because it tries to match the levels exactly
# rather than learning them from the data.
try(vec_assert(factor("x"), factor()))

# You can also enforce a minimum set of levels
try(vec_assert(factor("x"), partial_factor("y")))

vec_assert(factor(c("x", "y")), partial_factor("y"))

pf <- partial_factor(levels = c("x", "y"))
pf
```

```
vec_ptype_common(factor("v"), factor("w"), .ptype = pf)
```

---

partial_frame	<i>Partially specify columns of a data frame</i>
---------------	--

---

## Description

This special class can be passed to `.ptype` in order to specify the types of only some of the columns in a data frame.

## Usage

```
partial_frame(...)
```

## Arguments

...                      Attributes of subclass

## Examples

```
pf <- partial_frame(x = double())
pf

vec_rbind(
  data.frame(x = 1L, y = "a"),
  data.frame(x = FALSE, z = 10),
  .ptype = partial_frame(x = double(), a = character())
)
```

---

vec_assert	<i>Assert an argument has known prototype and/or size</i>
------------	---

---

## Description

- `vec_is()` is a predicate that checks if its input conforms to a prototype and/or a size.
- `vec_assert()` throws an error when the input doesn't conform.

## Usage

```
vec_assert(x, ptype = NULL, size = NULL, arg = as_label(substitute(x)))
```

```
vec_is(x, ptype = NULL, size = NULL)
```

## Arguments

x	A vector argument to check.
ptype	Prototype to compare against. If the prototype has a class, its <code>vec_ptype()</code> is compared to that of x with <code>identical()</code> . Otherwise, its <code>typeof()</code> is compared to that of x with <code>==</code> .
size	Size to compare against
arg	Name of argument being checked. This is used in error messages. The label of the expression passed as x is taken as default.

## Value

vec\_is() returns TRUE or FALSE. vec\_assert() either throws a typed error (see section on error types) or returns x, invisibly.

## Error types

- If the prototype doesn't match, an error of class "vctrs\_error\_assert\_ptype" is raised.
- If the size doesn't match, an error of class "vctrs\_error\_assert\_size" is raised.

Both errors inherit from "vctrs\_error\_assert".

---

vec_as_names	<i>Retrieve and repair names</i>
--------------	----------------------------------

---

## Description

vec\_as\_names() takes a character vector of names and repairs it according to the repair argument. It is the r-lib and tidyverse equivalent of [base::make.names\(\)](#).

vctrs deals with a few levels of name repair:

- **minimal** names exist. The names attribute is not NULL. The name of an unnamed element is "" and never NA. For instance, vec\_as\_names() always returns minimal names and data frames created by the tibble package have names that are, at least, minimal.
- **unique** names are minimal, have no duplicates, and can be used where a variable name is expected. Empty names, ..., and .. followed by a sequence of digits are banned.
  - All columns can be accessed by name via df[["name"]] and df\$`name` and with(df, `name`).
- **universal** names are unique and syntactic (see Details for more).
  - Names work everywhere, without quoting: df\$name and with(df, name) and lm(name1 ~ name2, data = df) and dplyr::select(df, name) all work.

universal implies unique, unique implies minimal. These levels are nested.

## Usage

```
vec_as_names(
  names,
  ...,
  repair = c("minimal", "unique", "universal", "check_unique"),
  quiet = FALSE
)
```

## Arguments

- |        |   |
|--------|---|
| names  | A character vector.   |
| ...    | These dots are for future extensions and must be empty.   |
| repair | Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", or "universal". If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown. <ul style="list-style-type: none"> <li>• Minimal names are never NULL or NA. When an element doesn't have a name, its minimal name is an empty string.</li> </ul> |

- Unique names are unique. A suffix is appended to duplicate names to make them unique.
- Universal names are unique and syntactic, meaning that you can safely use the names as variables without causing a syntax error.

The "check\_unique" option doesn't perform any name repair. Instead, an error is raised if the names don't suit the "unique" criteria.

quiet

By default, the user is informed of any renaming caused by repairing the names. This only concerns unique and universal repairing. Set quiet to TRUE to silence the messages.

### minimal names

minimal names exist. The names attribute is not NULL. The name of an unnamed element is "" and never NA.

Examples:

```
Original names of a vector with length 3: NULL
minimal names: "" "" ""
```

```
Original names: "x" NA
minimal names: "x" ""
```

### unique names

unique names are minimal, have no duplicates, and can be used (possibly with backticks) in contexts where a variable is expected. Empty names, ..., and .. followed by a sequence of digits are banned. If a data frame has unique names, you can index it by name, and also access the columns by name. In particular, df[["name"]] and df\$name and also with(df, `name`) always work.

There are many ways to make names unique. We append a suffix of the form ...j to any name that is "" or a duplicate, where j is the position. We also change .# and ... to ...#.

Example:

```
Original names:      ""      "x"      "" "y"      "x"  "..2"  "...
unique names: "...1" "x...2" "...3" "y" "x...5" "...6" "...7"
```

Pre-existing suffixes of the form ...j are always stripped, prior to making names unique, i.e. reconstructing the suffixes. If this interacts poorly with your names, you should take control of name repair.

### universal names

universal names are unique and syntactic, meaning they:

- Are never empty (inherited from unique).
- Have no duplicates (inherited from unique).
- Are not .... Do not have the form ..i, where i is a number (inherited from unique).
- Consist of letters, numbers, and the dot . or underscore \_ characters.
- Start with a letter or start with the dot . not followed by a number.
- Are not a [reserved](#) word, e.g., if or function or TRUE.

If a vector has universal names, variable names can be used "as is" in code. They work well with nonstandard evaluation, e.g., `df$name` works.

`vctrs` has a different method of making names syntactic than `base::make.names()`. In general, `vctrs` prepends one or more dots `.` until the name is syntactic.

Examples:

```
Original names:      ""      "x"      NA      "x"
universal names: "...1" "x...2" "...3" "x...4"
```

```
Original names: "(y)" "_z"  ".2fa"  "FALSE"
universal names: ".y." "._z" ".2fa" ".FALSE"
```

## See Also

`rlang::names2()` returns the names of an object, after making them minimal.

The **Names attribute** section in the "tidyverse package development principles".

## Examples

```
# By default, `vec_as_names()` returns minimal names:
vec_as_names(c(NA, NA, "foo"))

# You can make them unique:
vec_as_names(c(NA, NA, "foo"), repair = "unique")

# Universal repairing fixes any non-syntactic name:
vec_as_names(c("_foo", "+"), repair = "universal")
```

---

vec\_bind

*Combine many data frames into one data frame*

---

## Description

This pair of functions binds together data frames (and vectors), either row-wise or column-wise. Row-binding creates a data frame with common type across all arguments. Column-binding creates a data frame with common length across all arguments.

## Usage

```
vec_rbind(
  ...,
  .ptype = NULL,
  .names_to = NULL,
  .name_repair = c("unique", "universal", "check_unique")
)

vec_cbind(
  ...,
  .ptype = NULL,
  .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

**Arguments**

<code>...</code>	<p>Data frames or vectors.</p> <p>When the inputs are named:</p> <ul style="list-style-type: none"> <li><code>vec_rbind()</code> assigns names to row names unless <code>.names_to</code> is supplied. In that case the names are assigned in the column defined by <code>.names_to</code>.</li> <li><code>vec_cbind()</code> creates packed data frame columns with named inputs.</li> </ul> <p>NULL inputs are silently ignored. Empty (e.g. zero row) inputs will not appear in the output, but will affect the derived <code>.ptype</code>.</p>
<code>.ptype</code>	<p>If NULL, the default, the output type is determined by computing the common type across all elements of <code>...</code>.</p> <p>Alternatively, you can supply <code>.ptype</code> to give the output known type. If <code>getOption("vctrs.no_guesses")</code> is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.</p>
<code>.names_to</code>	<p>Optionally, the name of a column where the names of <code>...</code> arguments are copied. These names are useful to identify which row comes from which input. If supplied and <code>...</code> is not named, an integer column is used to identify the rows.</p>
<code>.name_repair</code>	<p>One of "unique", "universal", or "check_unique". See <a href="#">vec_as_names()</a> for the meaning of these options.</p> <p>With <code>vec_rbind()</code>, the repair function is applied to all inputs separately. This is because <code>vec_rbind()</code> needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, <code>vec_cbind()</code> applies the repair function after all inputs have been concatenated together in a final data frame. Hence <code>vec_cbind()</code> allows the more permissive minimal names repair.</p>
<code>.size</code>	<p>If, NULL, the default, will determine the number of rows in <code>vec_cbind()</code> output by using the standard recycling rules.</p> <p>Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.</p>

**Value**

A data frame, or subclass of data frame.

If `...` is a mix of different data frame subclasses, `vec_ptype2()` will be used to determine the output type. For `vec_rbind()`, this will determine the type of the container and the type of each column; for `vec_cbind()` it only determines the type of the output container. If there are no non-NULL inputs, the result will be `data.frame()`.

**Invariants**

All inputs are first converted to a data frame. The conversion for 1d vectors depends on the direction of binding:

- For `vec_rbind()`, each element of the vector becomes a column in a single row.
- For `vec_cbind()`, each element of the vector becomes a row in a single column.

Once the inputs have all become data frames, the following invariants are observed for row-binding:

- `vec_size(vec_rbind(x,y)) == vec_size(x) + vec_size(y)`
- `vec_ptype(vec_rbind(x,y)) = vec_ptype_common(x,y)`

Note that if an input is an empty vector, it is first converted to a 1-row data frame with 0 columns. Despite being empty, its effective size for the total number of rows is 1.

For column-binding, the following invariants apply:

- `vec_size(vec_cbind(x,y)) == vec_size_common(x,y)`
- `vec_ptype(vec_cbind(x,y)) == vec_cbind(vec_ptype(x),vec_ptype(y))`

### See Also

[vec\\_c\(\)](#) for combining 1d vectors.

### Examples

```
# row binding -----

# common columns are coerced to common class
vec_rbind(
  data.frame(x = 1),
  data.frame(x = FALSE)
)

# unique columns are filled with NAs
vec_rbind(
  data.frame(x = 1),
  data.frame(y = "x")
)

# null inputs are ignored
vec_rbind(
  data.frame(x = 1),
  NULL,
  data.frame(x = 2)
)

# bare vectors are treated as rows
vec_rbind(
  c(x = 1, y = 2),
  c(x = 3)
)

# default names will be supplied if arguments are not named
vec_rbind(
  1:2,
  1:3,
  1:4
)

# column binding -----

# each input is recycled to have common length
vec_cbind(
  data.frame(x = 1),
  data.frame(y = 1:3)
)

# bare vectors are treated as columns
```

```

vec_cbind(
  data.frame(x = 1),
  y = letters[1:3]
)

# if you supply a named data frame, it is packed in a single column
data <- vec_cbind(
  x = data.frame(a = 1, b = 2),
  y = 1
)
data

# Packed data frames are nested in a single column. This makes it
# possible to access it through a single name:
data$x

# since the base print method is suboptimal with packed data
# frames, it is recommended to use tibble to work with these:
if (rlang::is_installed("tibble")) {
  vec_cbind(x = tibble::tibble(a = 1, b = 2), y = 1)
}

# duplicate names are flagged
vec_cbind(x = 1, x = 2)

```

vec\_c

*Combine many vectors into one vector***Description**

Combine all arguments into a new vector of common type.

**Usage**

```

vec_c(
  ...,
  .ptype = NULL,
  .name_spec = NULL,
  .name_repair = c("minimal", "unique", "check_unique", "universal")
)

```

**Arguments**

...	Vectors to coerce.
.ptype	<p>If NULL, the default, the output type is determined by computing the common type across all elements of ...</p> <p>Alternatively, you can supply .ptype to give the output known type. If <code>getOption("vctrs.no_guess")</code> is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.</p>

`.name_spec` A name specification for combining inner and outer names. This is relevant for inputs passed with a name, when these inputs are themselves named, like `outer = c(inner = 1)`, or when they have length greater than 1: `outer = 1:2`. By default, these cases trigger an error. You can resolve the error by providing a specification that describes how to combine the names or the indices of the inner vector with the name of the input. This specification can be:

- A function of two arguments. The outer name is passed as a string to the first argument, and the inner names or positions are passed as second argument.
- An anonymous function as a purrr-style formula.
- A glue specification of the form `"{outer}_{inner}"`.

See the [name specification topic](#).

`.name_repair` How to repair names, see repair options in `vec_as_names()`.

## Value

A vector with class given by `.ptype`, and length equal to the sum of the `vec_size()` of the contents of ....

The vector will have names if the individual components have names (inner names) or if the arguments are named (outer names). If both inner and outer names are present, an error is thrown unless a `.name_spec` is provided.

## Invariants

- `vec_size(vec_c(x,y)) == vec_size(x) + vec_size(y)`
- `vec_ptype(vec_c(x,y)) == vec_ptype_common(x,y)`.

## See Also

[vec\\_cbind\(\)/vec\\_rbind\(\)](#) for combining data frames by rows or columns.

## Examples

```
vec_c(FALSE, 1L, 1.5)
vec_c(FALSE, 1L, "x", .ptype = character())

# Date/times -----
c(Sys.Date(), Sys.time())
c(Sys.time(), Sys.Date())

vec_c(Sys.Date(), Sys.time())
vec_c(Sys.time(), Sys.Date())

# Factors -----
c(factor("a"), factor("b"))
vec_c(factor("a"), factor("b"))

# By default, named inputs must be length 1:
vec_c(name = 1)
try(vec_c(name = 1:3))

# Pass a name specification to work around this:
```

```
vec_c(name = 1:3, .name_spec = "{outer}_{inner}")

# See `?name_spec` for more examples of name specifications.
```

vec\_chop

*Chopping***Description**

- `vec_chop()` provides an efficient method to repeatedly slice a vector. It captures the pattern of `map(indices, vec_slice, x = x)`. When no indices are supplied, it is generally equivalent to `as.list()`.
- `vec_unchop()` combines a list of vectors into a single vector, placing elements in the output according to the locations specified by indices. It is similar to `vec_c()`, but gives greater control over how the elements are combined. When no indices are supplied, it is identical to `vec_c()`.

If indices selects every value in `x` exactly once, in any order, then `vec_unchop()` is the inverse of `vec_chop()` and the following invariant holds:

```
vec_unchop(vec_chop(x, indices), indices) == x
```

**Usage**

```
vec_chop(x, indices = NULL)

vec_unchop(
  x,
  indices = NULL,
  ptype = NULL,
  name_spec = NULL,
  name_repair = c("minimal", "unique", "check_unique", "universal")
)
```

**Arguments**

<code>x</code>	A vector
<code>indices</code>	For <code>vec_chop()</code> , a list of positive integer vectors to slice <code>x</code> with, or <code>NULL</code> . If <code>NULL</code> , <code>x</code> is split into its individual elements, equivalent to using an indices of <code>as.list(vec_seq_along(x))</code> . For <code>vec_unchop()</code> , a list of positive integer vectors specifying the locations to place elements of <code>x</code> in. Each element of <code>x</code> is recycled to the size of the corresponding index vector. The size of indices must match the size of <code>x</code> . If <code>NULL</code> , <code>x</code> is combined in the order it is provided in, which is equivalent to using <code>vec_c()</code> .
<code>ptype</code>	If <code>NULL</code> , the default, the output type is determined by computing the common type across all elements of <code>x</code> . Alternatively, you can supply <code>ptype</code> to give the output a known type.
<code>name_spec</code>	A name specification for combining inner and outer names. This is relevant for inputs passed with a name, when these inputs are themselves named, like <code>outer = c(inner = 1)</code> , or when they have length greater than 1: <code>outer = 1:2</code> . By default, these cases trigger an error. You can resolve the error by providing a

specification that describes how to combine the names or the indices of the inner vector with the name of the input. This specification can be:

- A function of two arguments. The outer name is passed as a string to the first argument, and the inner names or positions are passed as second argument.
- An anonymous function as a purrr-style formula.
- A glue specification of the form "{outer}\_{inner}".

See the [name specification topic](#).

name\_repair      How to repair names, see repair options in [vec\\_as\\_names\(\)](#).

## Value

- `vec_chop()`: A list of size `vec_size(indices)` or, if `indices == NULL`, `vec_size(x)`.
- `vec_unchop()`: A vector of type `vec_ptype_common(!!!x)`, or `ptype`, if specified. The size is computed as `vec_size_common(!!!indices)` unless the indices are `NULL`, in which case the size is `vec_size_common(!!!x)`.

## Examples

```
vec_chop(1:5)
vec_chop(1:5, list(1, 1:2))
vec_chop(mtcars, list(1:3, 4:6))

# If `indices` selects every value in `x` exactly once,
# in any order, then `vec_unchop()` inverts `vec_chop()`
x <- c("a", "b", "c", "d")
indices <- list(2, c(3, 1), 4)
vec_chop(x, indices)
vec_unchop(vec_chop(x, indices), indices)

# When unchopping, size 1 elements of `x` are recycled
# to the size of the corresponding index
vec_unchop(list(1, 2:3), list(c(1, 3, 5), c(2, 4)))

# Names are retained, and outer names can be combined with inner
# names through the use of a `name_spec`
lst <- list(x = c(a = 1, b = 2), y = 1)
vec_unchop(lst, list(c(3, 2), c(1, 4)), name_spec = "{outer}_{inner}")

# An alternative implementation of `ave()` can be constructed using
# `vec_chop()` and `vec_unchop()` in combination with `vec_group_loc()`
ave2 <- function(.x, .by, .f, ...) {
  indices <- vec_group_loc(.by)$loc
  chopped <- vec_chop(.x, indices)
  out <- lapply(chopped, .f, ...)
  vec_unchop(out, indices)
}

breaks <- warpbreaks$breaks
wool <- warpbreaks$wool

ave2(breaks, wool, mean)

identical(
```

```
ave2(breaks, wool, mean),
ave(breaks, wool, FUN = mean)
)
```

---

vec_compare	<i>Compare two vectors</i>
-------------	----------------------------

---

## Description

Compare two vectors

## Usage

```
vec_compare(x, y, na_equal = FALSE, .ptype = NULL)
```

## Arguments

x, y	Vectors with compatible types and lengths.
na_equal	Should NA values be considered equal?
.ptype	Override to optionally specify common type

## Value

An integer vector with values -1 for  $x < y$ , 0 if  $x == y$ , and 1 if  $x > y$ . If `na_equal` is `FALSE`, the result will be NA if either `x` or `y` is NA.

## S3 dispatch

`vec_compare()` is not generic for performance; instead it uses [vec\\_proxy\\_compare\(\)](#) to

## Examples

```
vec_compare(c(TRUE, FALSE, NA), FALSE)
vec_compare(c(TRUE, FALSE, NA), FALSE, na_equal = TRUE)

vec_compare(1:10, 5)
vec_compare(runif(10), 0.5)
vec_compare(letters[1:10], "d")

df <- data.frame(x = c(1, 1, 1, 2), y = c(0, 1, 2, 1))
vec_compare(df, data.frame(x = 1, y = 1))
```

vec\_count

*Count unique values in a vector***Description**

Count the number of unique values in a vector. `vec_count()` has two important differences to `table()`: it returns a data frame, and when given multiple inputs (as a data frame), it only counts combinations that appear in the input.

**Usage**

```
vec_count(x, sort = c("count", "key", "location", "none"))
```

**Arguments**

<code>x</code>	A vector (including a data frame).
<code>sort</code>	One of "count", "key", "location", or "none". <ul style="list-style-type: none"> <li>"count", the default, puts most frequent values at top</li> <li>"key", orders by the output key column (i.e. unique values of <code>x</code>)</li> <li>"location", orders by location where key first seen. This is useful if you want to match the counts up to other unique/duplicated functions.</li> <li>"none", leaves unordered.</li> </ul>

**Value**

A data frame with columns `key` (same type as `x`) and `count` (an integer vector).

**Examples**

```
vec_count(mtcars$vs)
vec_count(iris$Species)

# If you count a data frame you'll get a data frame
# column in the output
str(vec_count(mtcars[c("vs", "am")]))

# Sorting -----

x <- letters[rpois(100, 6)]
# default is to sort by frequency
vec_count(x)

# by can sort by key
vec_count(x, sort = "key")

# or location of first value
vec_count(x, sort = "location")
head(x)

# or not at all
vec_count(x, sort = "none")
```

---

vec_data	<i>Extract underlying data</i>
----------	--------------------------------

---

## Description

Extract the data underlying an S3 vector object, i.e. the underlying (named) atomic vector or list.

- `vec_data()` returns unstructured data. The only attributes preserved are names, dims, and dimnames.

Currently, due to the underlying memory architecture of R, this creates a full copy of the data.

- `vec_proxy()` may return structured data. This generic is the main customisation point in `vctrs`, along with `vec_restore()`. See the section below to learn when you should implement `vec_proxy()`.

Methods must return a vector type. Records and data frames will be processed rowwise.

## Usage

```
vec_data(x)
```

```
vec_proxy(x, ...)
```

## Arguments

<code>x</code>	A vector or object implementing <code>vec_proxy()</code> .
<code>...</code>	These dots are for future extensions and must be empty.

## Value

The data underlying `x`, free from any attributes except the names.

## When should you proxy your type

You should only implement `vec_proxy()` when your type is designed around a non-vector class. I.e. anything that is not either:

- An atomic vector
- A bare list
- A data frame

In this case, implement `vec_proxy()` to return such a vector class. The `vctrs` operations such as `vec_slice()` are applied on the proxy and `vec_restore()` is called to restore the original representation of your type.

The most common case where you need to implement `vec_proxy()` is for S3 lists. In `vctrs`, S3 lists are treated as scalars by default. This way we don't treat objects like model fits as vectors. To prevent `vctrs` from treating your S3 list as a scalar, unclass it in the `vec_proxy()` method. For instance, here is the definition for `list_of`:

```
vec_proxy.vctrs_list_of <- function(x) {
  unclass(x)
}
```

Another case where you need to implement a proxy is [record types](#). Record types should return a data frame, as in the `POSIXlt` method:

```
vec_proxy.POSIXlt <- function(x) {
  new_data_frame(unclass(x))
}
```

Note that you don't need to implement `vec_proxy()` when your class inherits from `vctrs_vctr` or `vctrs_rcrd`.

## See Also

See [vec\\_restore\(\)](#) for the inverse operation: it restores attributes given a bare vector and a prototype; `vec_restore(vec_data(x), x)` will always yield `x`.

---

<code>vec_default_cast</code>	<i>Default cast method</i>
-------------------------------	----------------------------

---

## Description

This function should typically be called from the default [vec\\_cast\(\)](#) method for your class, e.g. `vec_cast.myclass.default()`. It does two things:

- If `x` is an [unspecified](#) vector, it automatically casts it to `to` using [vec\\_init\(\)](#).
- Otherwise, an error is thrown with [stop\\_incompatible\\_cast\(\)](#).

## Usage

```
vec_default_cast(x, to, x_arg = "x", to_arg = "to")
```

## Arguments

<code>x</code>	Vectors to cast.
<code>to</code>	Type to cast to. If <code>NULL</code> , <code>x</code> will be returned as is.
<code>x_arg</code>	Argument names for <code>x</code> and <code>to</code> . These are used in error messages to inform the user about the locations of incompatible types (see <a href="#">stop_incompatible_type()</a> ).
<code>to_arg</code>	Argument names for <code>x</code> and <code>to</code> . These are used in error messages to inform the user about the locations of incompatible types (see <a href="#">stop_incompatible_type()</a> ).

---

vec_duplicate	<i>Find duplicated values</i>
---------------	-------------------------------

---

### Description

- `vec_duplicate_any()`: detects the presence of duplicated values, similar to [anyDuplicated\(\)](#).
- `vec_duplicate_detect()`: returns a logical vector describing if each element of the vector is duplicated elsewhere. Unlike [duplicated\(\)](#), it reports all duplicated values, not just the second and subsequent repetitions.
- `vec_duplicate_id()`: returns an integer vector giving the location of the first occurrence of the value.

### Usage

```
vec_duplicate_any(x)
```

```
vec_duplicate_detect(x)
```

```
vec_duplicate_id(x)
```

### Arguments

`x`                      A vector (including a data frame).

### Value

- `vec_duplicate_any()`: a logical vector of length 1.
- `vec_duplicate_detect()`: a logical vector the same length as `x`.
- `vec_duplicate_id()`: an integer vector the same length as `x`.

### Missing values

In most cases, missing values are not considered to be equal, i.e. `NA == NA` is not `TRUE`. This behaviour would be unappealing here, so these functions consider all `NA`s to be equal. (Similarly, all `NaN` are also considered to be equal.)

### See Also

[vec\\_unique\(\)](#) for functions that work with the dual of duplicated values: unique values.

### Examples

```
vec_duplicate_any(1:10)
vec_duplicate_any(c(1, 1:10))

x <- c(10, 10, 20, 30, 30, 40)
vec_duplicate_detect(x)
# Note that `duplicated()` doesn't consider the first instance to
# be a duplicate
duplicated(x)
```

```
# Identify elements of a vector by the location of the first element that
# they're equal to:
vec_duplicate_id(x)
# Location of the unique values:
vec_unique_loc(x)
# Equivalent to `duplicated()`:
vec_duplicate_id(x) == seq_along(x)
```

---

vec_equal	<i>Test if two vectors are equal</i>
-----------	--------------------------------------

---

## Description

vec\_equal\_na() tests a special case: equality with NA. It is similar to [is.na](#) but:

- Considers the missing element of a list to be NULL.
- Considered data frames and records to be missing if every component is missing. This preserves the invariant that vec\_equal\_na(x) is equal to vec\_equal(x, vec\_init(x), na\_equal = TRUE).

## Usage

```
vec_equal(x, y, na_equal = FALSE, .ptype = NULL)
```

```
vec_equal_na(x)
```

## Arguments

x	Vectors with compatible types and lengths.
y	Vectors with compatible types and lengths.
na_equal	Should NA values be considered equal?
.ptype	Override to optionally specify common type

## Value

A logical vector the same size as. Will only contain NAs if na\_equal is FALSE.

## Examples

```
vec_equal(c(TRUE, FALSE, NA), FALSE)
vec_equal(c(TRUE, FALSE, NA), FALSE, na_equal = TRUE)
vec_equal_na(c(TRUE, FALSE, NA))

vec_equal(5, 1:10)
vec_equal("d", letters[1:10])

df <- data.frame(x = c(1, 1, 2, 1, NA), y = c(1, 2, 1, NA, NA))
vec_equal(df, data.frame(x = 1, y = 2))
vec_equal_na(df)
```

---

vec_init	<i>Initialize a vector</i>
----------	----------------------------

---

**Description**

Initialize a vector

**Usage**

```
vec_init(x, n = 1L)
```

**Arguments**

x	Template of vector to initialize.
n	Desired size of result.

**Examples**

```
vec_init(1:10, 3)
vec_init(Sys.Date(), 5)
vec_init(mtcars, 2)
```

---

vec_is_list	<i>Is the object a list?</i>
-------------	------------------------------

---

**Description**

vec\_is\_list() tests if x is considered a list in the vctrs sense. It returns TRUE if:

- x is a bare list with no class.
- x is a list explicitly inheriting from "list" or "vctrs\_list\_of".
- x is an S3 list that [vec\\_is\(\)](#) returns TRUE for. For this to return TRUE, the class must implement a [vec\\_proxy\(\)](#) method.

**Usage**

```
vec_is_list(x)
```

**Arguments**

x	An object.
---	------------

**Details**

Notably, data frames and S3 record style classes like POSIXlt are not considered lists.

**Examples**

```
vec_is_list(list())
vec_is_list(list_of(1))

vec_is_list(data.frame())
```

vec\_match

*Find matching observations across vectors***Description**

vec\_in() returns a logical vector based on whether needle is found in haystack. vec\_match() returns an integer vector giving location of needle in haystack, or NA if it's not found.

**Usage**

```
vec_match(needles, haystack, ..., na_equal = TRUE)
```

```
vec_in(needles, haystack, ..., na_equal = TRUE)
```

**Arguments**

needles, haystack

Vector of needles to search for in vector haystack. haystack should usually be unique; if not vec\_match() will only return the location of the first match. needles and haystack are coerced to the same type prior to comparison.

...

These dots are for future extensions and must be empty.

na\_equal

If TRUE, missing values in needles can be matched to missing values in haystack. If FALSE, they propagate, missing values in needles are represented as NA in the return value.

**Details**

vec\_in() is equivalent to `%in%`; vec\_match() is equivalent to `match()`.

**Value**

A vector the same length as needles. vec\_in() returns a logical vector; vec\_match() returns an integer vector.

**Missing values**

In most cases places in R, missing values are not considered to be equal, i.e. `NA == NA` is not `TRUE`. The exception is in matching functions like `match()` and `merge()`, where an NA will match another NA. By `vec_match()` and `vec_in()` will match NAs; but you can control this behaviour with the `na_equal` argument.

**Examples**

```
hadley <- strsplit("hadley", "")[[1]]
vec_match(hadley, letters)

vowels <- c("a", "e", "i", "o", "u")
vec_match(hadley, vowels)
vec_in(hadley, vowels)

# Only the first index of duplicates is returned
vec_match(c("a", "b"), c("a", "b", "a", "b"))
```

vec\_order

*Order and sort vectors***Description**

Order and sort vectors

**Usage**

```
vec_order(x, direction = c("asc", "desc"), na_value = c("largest", "smallest"))
```

```
vec_sort(x, direction = c("asc", "desc"), na_value = c("largest", "smallest"))
```

**Arguments**

x	A vector
direction	Direction to sort in. Defaults to ascending.
na_value	Should NAs be treated as the largest or smallest values?

**Value**

- `vec_order()` an integer vector the same size as x.
- `vec_sort()` a vector with the same size and type as x.

**Examples**

```
x <- round(c(runif(9), NA), 3)
vec_order(x)
vec_sort(x)
vec_sort(x, "desc")

# Can also handle data frames
df <- data.frame(g = sample(2, 10, replace = TRUE), x = x)
vec_order(df)
vec_sort(df)
vec_sort(df, "desc")
```

vec\_ptype

*Find the prototype of a set of vectors***Description**

`vec_ptype()` returns the unfinalised prototype of a single vector. `vec_ptype_common()` finds the common type of multiple vectors. `vec_ptype_show()` nicely prints the common type of any number of inputs, and is designed for interactive exploration.

**Usage**

```
vec_ptype(x)

vec_ptype_common(..., .ptype = NULL)

vec_ptype_show(...)
```

**Arguments**

<code>...</code> , <code>x</code>	Vectors inputs
<code>.ptype</code>	<p>If <code>NULL</code>, the default, the output type is determined by computing the common type across all elements of <code>...</code>.</p> <p>Alternatively, you can supply <code>.ptype</code> to give the output known type. If <code>getOption("vctrs.no_guess")</code> is <code>TRUE</code> you must supply this value: this is a convenient way to make production code demand fixed types.</p>

**Value**

`vec_ptype()` and `vec_ptype_common()` return a prototype (a size-0 vector)

**vec\_ptype()**

`vec_ptype()` returns [size 0](#) vectors potentially containing attributes but no data. Generally, this is just `vec_slice(x, 0L)`, but some inputs require special handling.

- While you can't slice `NULL`, the prototype of `NULL` is itself. This is because we treat `NULL` as an identity value in the `vec_ptype2()` monoid.
- The prototype of logical vectors that only contain missing values is the special [unspecified](#) type, which can be coerced to any other 1d type. This allows bare NAs to represent missing values for any 1d vector type.

See [internal-faq-ptype2-identity](#) for more information about identity values.

Because it may contain unspecified vectors, the prototype returned by `vec_ptype()` is said to be **unfinalised**. Call [vec\\_ptype\\_finalise\(\)](#) to finalise it. Commonly you will need the finalised prototype as returned by `vec_slice(x, 0L)`.

**vec\_ptype\_common()**

`vec_ptype_common()` first finds the prototype of each input, then successively calls [vec\\_ptype2\(\)](#) to find a common type. It returns a [finalised](#) prototype.

**Examples**

```
# Unknown types -----
vec_ptype_show()
vec_ptype_show(NA)
vec_ptype_show(NULL)

# Vectors -----
vec_ptype_show(1:10)
vec_ptype_show(letters)
vec_ptype_show(TRUE)

vec_ptype_show(Sys.Date())
```

```

vec_ptype_show(Sys.time())
vec_ptype_show(factor("a"))
vec_ptype_show(ordered("a"))

# Matrices -----
# The prototype of a matrix includes the number of columns
vec_ptype_show(array(1, dim = c(1, 2)))
vec_ptype_show(array("x", dim = c(1, 2)))

# Data frames -----
# The prototype of a data frame includes the prototype of
# every column
vec_ptype_show(iris)

# The prototype of multiple data frames includes the prototype
# of every column that in any data frame
vec_ptype_show(
  data.frame(x = TRUE),
  data.frame(y = 2),
  data.frame(z = "a")
)

```

---

vec\_recycle

*Vector recycling*


---

## Description

`vec_recycle(x, size)` recycles a single vector to given size. `vec_recycle_common(...)` recycles multiple vectors to their common size. All functions obey the `vctrs` recycling rules, described below, and will throw an error if recycling is not possible. See `vec_size()` for the precise definition of size.

## Usage

```

vec_recycle(x, size, ..., x_arg = "x")

vec_recycle_common(..., .size = NULL)

```

## Arguments

<code>x</code>	A vector to recycle.
<code>size</code>	Desired output size.
<code>...</code>	<ul style="list-style-type: none"> <li>For <code>vec_recycle_common()</code>, vectors to recycle.</li> <li>For <code>vec_recycle()</code>, these dots should be empty.</li> </ul>
<code>x_arg</code>	Argument name for <code>x</code> . These are used in error messages to inform the user about which argument has an incompatible size.
<code>.size</code>	Desired output size. If omitted, will use the common size from <code>vec_size_common()</code> .

### Recycling rules

The common size of two vectors defines the recycling rules, and can be summarised with the following table:

	1	m	n
1	1	m	n
m	m	m	X
n	n	X	n

(Note NULLs are handled specially; they are treated like empty arguments and hence don't affect the size)

This is a stricter set of rules than base R, which will usually return output of length  $\max(n_x, n_y)$ , warning if the length of the longer vector is not an integer multiple of the length of the shorter.

We say that two vectors have **compatible size** if they can be recycled to be the same length.

### Examples

```
# Inputs with 1 observation are recycled
vec_recycle_common(1:5, 5)
vec_recycle_common(integer(), 5)
## Not run:
vec_recycle_common(1:5, 1:2)

## End(Not run)

# Data frames and matrices are recycled along their rows
vec_recycle_common(data.frame(x = 1), 1:5)
vec_recycle_common(array(1:2, c(1, 2)), 1:5)
vec_recycle_common(array(1:3, c(1, 3, 1)), 1:5)
```

---

vec_repeat	<i>Expand the length of a vector</i>
------------	--------------------------------------

---

**Description**

This is a special case of `rep()` for the special case of integer times and each values, and works along size, rather than length.

**Usage**

```
vec_repeat(x, each = 1L, times = 1L)
```

**Arguments**

x	A vector.
each	Number of times to repeat each element of x.
times	Number of times to repeat the whole vector of x.

**Value**

A vector the same type as x with size `vec_size(x) * times * each`.

**Examples**

```
# each repeats within
vec_repeat(1:3, each = 2)
# times repeats whole thing
vec_repeat(1:3, times = 2)

df <- data.frame(x = 1:2, y = 1:2)
# rep() repeats columns of data frame, and returns list:
rep(df, each = 2)
# vec_repeat() repeats rows, and returns same data.frame
vec_repeat(df, 2)
```

---

vec_seq_along	<i>Useful sequences</i>
---------------	-------------------------

---

**Description**

`vec_seq_along()` is equivalent to `seq_along()` but uses size, not length. `vec_init_along()` creates a vector of missing values with size matching an existing object.

**Usage**

```
vec_seq_along(x)

vec_init_along(x, y = x)
```

**Arguments**

`x, y`                      Vectors

**Value**

- `vec_seq_along()` an integer vector with the same size as `x`.
- `vec_init_along()` a vector with the same type as `x` and the same size as `y`.

**Examples**

```
vec_seq_along(mtcars)
vec_init_along(head(mtcars))
```

---

<code>vec_size</code>	<i>Number of observations</i>
-----------------------	-------------------------------

---

**Description**

`vec_size(x)` returns the size of a vector. `vec_is_empty()` returns TRUE if the size is zero, FALSE otherwise.

The size is distinct from the `length()` of a vector because it generalises to the "number of observations" for 2d structures, i.e. it's the number of rows in matrix or a data frame. This definition has the important property that every column of a data frame (even data frame and matrix columns) have the same size. `vec_size_common(...)` returns the common size of multiple vectors.

**Usage**

```
vec_size(x)

vec_size_common(..., .size = NULL, .absent = 0L)

vec_is_empty(x)
```

**Arguments**

<code>x, ...</code>	Vector inputs or NULL.
<code>.size</code>	If NULL, the default, the output size is determined by recycling the lengths of all elements of <code>...</code> . Alternatively, you can supply <code>.size</code> to force a known size; in this case, <code>x</code> and <code>...</code> are ignored.
<code>.absent</code>	The size used when no input is provided, or when all input is NULL. If left as NULL when no input is supplied, an error is thrown.

**Details**

There is no `vctrs` helper that retrieves the number of columns: as this is a property of the [type](#).

`vec_size()` is equivalent to `NROW()` but has a name that is easier to pronounce, and throws an error when passed non-vector inputs.

**Value**

An integer (or double for long vectors).

vec\_size\_common() returns .absent if all inputs are NULL or absent, 0L by default.

**Invariants**

- `vec_size(dataframe) == vec_size(dataframe[[i]])`
- `vec_size(matrix) == vec_size(matrix[,i,drop = FALSE])`
- `vec_size(vec_c(x,y)) == vec_size(x) + vec_size(y)`

**The size of NULL**

The size of NULL is hard-coded to 0L in `vec_size()`. `vec_size_common()` returns .absent when all inputs are NULL (if only some inputs are NULL, they are simply ignored).

A default size of 0 makes sense because sizes are most often queried in order to compute a total size while assembling a collection of vectors. Since we treat NULL as an absent input by principle, we return the identity of sizes under addition to reflect that an absent input doesn't take up any size.

Note that other defaults might make sense under different circumstances. For instance, a default size of 1 makes sense for finding the common size because 1 is the identity of the recycling rules.

**See Also**

[vec\\_slice\(\)](#) for a variation of [ compatible with `vec_size()`, and [vec\\_recycle\(\)](#) to recycle vectors to common length.

**Examples**

```
vec_size(1:100)
vec_size(mtcars)
vec_size(array(dim = c(3, 5, 10)))

vec_size_common(1:10, 1:10)
vec_size_common(1:10, 1)
vec_size_common(integer(), 1)
```

---

vec\_split

---

*Split a vector into groups*


---

**Description**

This is a generalisation of [split\(\)](#) that can split by any type of vector, not just factors. Instead of returning the keys in the character names, they are returned in a separate parallel vector.

**Usage**

```
vec_split(x, by)
```

**Arguments**

- |    |  |
|----|--|
| x  | Vector to divide into groups.                  |
| by | Vector whose unique values defines the groups. |

**Value**

A data frame with two columns and size equal to `vec_size(vec_unique(by))`. The key column has the same type as `by`, and the `val` column is a list containing elements of type `vec_ptype(x)`.

Note for complex types, the default `data.frame` print method will be suboptimal, and you will want to coerce into a tibble to better understand the output.

**Examples**

```
vec_split(mtcars$cyl, mtcars$vs)
vec_split(mtcars$cyl, mtcars[c("vs", "am")])

if (require("tibble")) {
  as_tibble(vec_split(mtcars$cyl, mtcars[c("vs", "am")]))
  as_tibble(vec_split(mtcars, mtcars[c("vs", "am")]))
}
```

---

vec_unique	<i>Find and count unique values</i>
------------	-------------------------------------

---

**Description**

- `vec_unique()`: the unique values. Equivalent to `unique()`.
- `vec_unique_loc()`: the locations of the unique values.
- `vec_unique_count()`: the number of unique values.

**Usage**

```
vec_unique(x)

vec_unique_loc(x)

vec_unique_count(x)
```

**Arguments**

`x`                      A vector (including a data frame).

**Value**

- `vec_unique()`: a vector the same type as `x` containing only unique values.
- `vec_unique_loc()`: an integer vector, giving locations of unique values.
- `vec_unique_count()`: an integer vector of length 1, giving the number of unique values.

**Missing values**

In most cases, missing values are not considered to be equal, i.e. `NA == NA` is not `TRUE`. This behaviour would be unappealing here, so these functions consider all NAs to be equal. (Similarly, all NaN are also considered to be equal.)

**See Also**

[vec\\_duplicate](#) for functions that work with the dual of unique values: duplicated values.

**Examples**

```
x <- rpois(100, 8)
vec_unique(x)
vec_unique_loc(x)
vec_unique_count(x)

# `vec_unique()` returns values in the order that encounters them
# use sort = "location" to match to the result of `vec_count()`
head(vec_unique(x))
head(vec_count(x, sort = "location"))

# Normally missing values are not considered to be equal
NA == NA

# But they are for the purposes of considering uniqueness
vec_unique(c(NA, NA, NA, NA, 1, 2, 1))
```

---

*%0%**Default value for empty vectors*

---

**Description**

Use this inline operator when you need to provide a default value for empty (as defined by [vec\\_is\\_empty\(\)](#)) vectors.

**Usage**

```
x %0% y
```

**Arguments**

x	A vector
y	Value to use if x is empty. To preserve type-stability, should be the same type as x.

**Examples**

```
1:10 %0% 5
integer() %0% 5
```

# Index

`%0%`, 33  
`%in%`, 24

`anyDuplicated()`, 21  
`as.list()`, 15  
`as_list_of(list_of)`, 4

`base::make.names()`, 8, 10

`duplicated()`, 21

`finalised`, 26

`internal-faq-ptype2-identity`, 2, 26  
`is.na`, 22  
`is_list_of(list_of)`, 4

`length()`, 30  
`list_of`, 4

`match()`, 24  
`merge()`, 24

`name specification topic`, 5, 14, 16  
`name_spec`, 5

`partial_factor`, 6  
`partial_frame`, 7

`record types`, 20  
`rep()`, 29  
`reserved`, 9  
`rlang::names2()`, 10

`seq_along()`, 29  
`size`, 26  
`split()`, 31  
`stop_incompatible_cast()`, 20  
`stop_incompatible_type()`, 4, 20

`type`, 30  
`typeof()`, 7

`unique()`, 32  
`unspecified`, 20, 26

`validate_list_of(list_of)`, 4  
`vec_as_names`, 8  
`vec_as_names()`, 11, 14, 16  
`vec_assert`, 7  
`vec_bind`, 10  
`vec_c`, 13  
`vec_c()`, 12, 15  
`vec_cast()`, 20  
`vec_cast.vctrs_list_of(list_of)`, 4  
`vec_cbind(vec_bind)`, 10  
`vec_cbind()`, 14  
`vec_chop`, 15  
`vec_compare`, 17  
`vec_count`, 18  
`vec_data`, 19  
`vec_default_cast`, 20  
`vec_duplicate`, 21, 33  
`vec_duplicate_any(vec_duplicate)`, 21  
`vec_duplicate_detect(vec_duplicate)`, 21  
`vec_duplicate_id(vec_duplicate)`, 21  
`vec_equal`, 22  
`vec_equal_na(vec_equal)`, 22  
`vec_in(vec_match)`, 24  
`vec_init`, 23  
`vec_init()`, 20  
`vec_init_along(vec_seq_along)`, 29  
`vec_is(vec_assert)`, 7  
`vec_is()`, 23  
`vec_is_empty(vec_size)`, 30  
`vec_is_empty()`, 33  
`vec_is_list`, 23  
`vec_match`, 24  
`vec_order`, 25  
`vec_proxy(vec_data)`, 19  
`vec_proxy()`, 23  
`vec_proxy_compare()`, 17  
`vec_ptype`, 25  
`vec_ptype()`, 7  
`vec_ptype2()`, 26  
`vec_ptype2.vctrs_list_of(list_of)`, 4  
`vec_ptype_common(vec_ptype)`, 25  
`vec_ptype_finalise()`, 26  
`vec_ptype_show(vec_ptype)`, 25

`vec_rbind(vec_bind)`, 10  
`vec_rbind()`, 14  
`vec_recycle`, 27  
`vec_recycle()`, 31  
`vec_recycle_common(vec_recycle)`, 27  
`vec_repeat`, 29  
`vec_restore()`, 19, 20  
`vec_seq_along`, 29  
`vec_size`, 30  
`vec_size()`, 27  
`vec_size_common(vec_size)`, 30  
`vec_size_common()`, 27  
`vec_slice()`, 19, 31  
`vec_sort(vec_order)`, 25  
`vec_split`, 31  
`vec_unchop(vec_chop)`, 15  
`vec_unique`, 32  
`vec_unique()`, 21  
`vec_unique_count(vec_unique)`, 32  
`vec_unique_loc(vec_unique)`, 32