

Package: vctrs (via r-universe)

December 27, 2024

Title Vector Helpers

Version 0.6.5.9000

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 MIT + file LICENSE

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

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

Depends R (>= 3.5.0)

Imports cli (>= 3.4.0), glue, lifecycle (>= 1.0.3), rlang (>= 1.1.0)

Suggests bit64, covr, crayon, dplyr (>= 0.8.5), generics, knitr, pillar (>= 1.4.4), pkgdown (>= 2.0.1), rmarkdown, testthat (>= 3.0.0), tibble (>= 3.1.3), waldo (>= 0.2.0), withr, xml2, zeallot

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Language en-GB

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Repository <https://r-lib.r-universe.dev>

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

RemoteRef HEAD

RemoteSha 78d9f2b0b24131b5ce2230eb3c2c9f93620b10d9

Contents

data_frame	3
df_list	5
df_ptype2	6
faq-compatibility-types	7
faq-error-incompatible-attributes	9
faq-error-scalar-type	9
howto-faq-coercion	11
howto-faq-coercion-data-frame	15
howto-faq-fix-scalar-type-error	23
internal-faq-matches-algorithm	24
internal-faq-ptype2-identity	31
list_drop_empty	33
list_of	33
missing	34
name_spec	35
new_data_frame	37
obj_is_list	37
reference-faq-compatibility	39
runs	40
theory-faq-coercion	41
theory-faq-recycling	46
vec_rep	47
vec_set	49
vector-checks	53
vec_as_names	55
vec_bind	58
vec_c	61
vec_cast	64
vec_chop	66
vec_compare	69
vec_count	70
vec_detect_complete	71
vec_duplicate	72
vec_equal	74
vec_expand_grid	75
vec_fill_missing	76
vec_init	77
vec_interleave	78
vec_locate_matches	79
vec_match	85
vec_names	87
vec_order	88
vec_ptype	90
vec_ptype2.logical	92
vec_rank	94
vec_recycle	96

vec_seq_along	97
vec_size	98
vec_split	100
vec_unique	101
%0%	102

Index	103
--------------	------------

data_frame	<i>Construct a data frame</i>
------------	-------------------------------

Description

data_frame() constructs a data frame. It is similar to `base::data.frame()`, but there are a few notable differences that make it more in line with vctrs principles. The Properties section outlines these.

Usage

```
data_frame(
  ...,
  .size = NULL,
  .name_repair = c("check_unique", "unique", "universal", "minimal", "unique_quiet",
    "universal_quiet"),
  .error_call = current_env()
)
```

Arguments

...	Vectors to become columns in the data frame. When inputs are named, those names are used for column names.
.size	The number of rows in the data frame. If NULL, this will be computed as the common size of the inputs.
.name_repair	One of "check_unique", "unique", "universal", "minimal", "unique_quiet", or "universal_quiet". See <code>vec_as_names()</code> for the meaning of these options.
.error_call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the call argument of <code>abort()</code> for more information.

Details

If no column names are supplied, "" will be used as a default name for all columns. This is applied before name repair occurs, so the default name repair of "check_unique" will error if any unnamed inputs are supplied and "unique" (or "unique_quiet") will repair the empty string column names appropriately. If the column names don't matter, use a "minimal" name repair for convenience and performance.

Properties

- Inputs are [recycled](#) to a common size with `vec_recycle_common()`.
- With the exception of data frames, inputs are not modified in any way. Character vectors are never converted to factors, and lists are stored as-is for easy creation of list-columns.
- Unnamed data frame inputs are automatically unpacked. Named data frame inputs are stored unmodified as data frame columns.
- NULL inputs are completely ignored.
- The dots are dynamic, allowing for splicing of lists with `!!!` and unquoting.

See Also

`df_list()` for safely creating a data frame's underlying data structure from individual columns. `new_data_frame()` for constructing the actual data frame from that underlying data structure. Together, these can be useful for developers when creating new data frame subclasses supporting standard evaluation.

Examples

```
data_frame(x = 1, y = 2)

# Inputs are recycled using tidyverse recycling rules
data_frame(x = 1, y = 1:3)

# Strings are never converted to factors
class(data_frame(x = "foo")$x)

# List columns can be easily created
df <- data_frame(x = list(1:2, 2, 3:4), y = 3:1)

# However, the base print method is suboptimal for displaying them,
# so it is recommended to convert them to tibble
if (rlang::is_installed("tibble")) {
  tibble::as_tibble(df)
}

# Named data frame inputs create data frame columns
df <- data_frame(x = data_frame(y = 1:2, z = "a"))

# The `x` column itself is another data frame
df$x

# Again, it is recommended to convert these to tibbles for a better
# print method
if (rlang::is_installed("tibble")) {
  tibble::as_tibble(df)
}

# Unnamed data frame input is automatically unpacked
data_frame(x = 1, data_frame(y = 1:2, z = "a"))
```

df_list

*Collect columns for data frame construction***Description**

df_list() constructs the data structure underlying a data frame, a named list of equal-length vectors. It is often used in combination with [new_data_frame\(\)](#) to safely and consistently create a helper function for data frame subclasses.

Usage

```
df_list(
  ...,
  .size = NULL,
  .unpack = TRUE,
  .name_repair = c("check_unique", "unique", "universal", "minimal", "unique_quiet",
    "universal_quiet"),
  .error_call = current_env()
)
```

Arguments

...	Vectors of equal-length. When inputs are named, those names are used for names of the resulting list.
.size	The common size of vectors supplied in If NULL, this will be computed as the common size of the inputs.
.unpack	Should unnamed data frame inputs be unpacked? Defaults to TRUE.
.name_repair	One of "check_unique", "unique", "universal", "minimal", "unique_quiet", or "universal_quiet". See vec_as_names() for the meaning of these options.
.error_call	The execution environment of a currently running function, e.g. caller_env() . The function will be mentioned in error messages as the source of the error. See the call argument of abort() for more information.

Properties

- Inputs are [recycled](#) to a common size with [vec_recycle_common\(\)](#).
- With the exception of data frames, inputs are not modified in any way. Character vectors are never converted to factors, and lists are stored as-is for easy creation of list-columns.
- Unnamed data frame inputs are automatically unpacked. Named data frame inputs are stored unmodified as data frame columns.
- NULL inputs are completely ignored.
- The dots are dynamic, allowing for splicing of lists with !!! and unquoting.

See Also

`new_data_frame()` for constructing data frame subclasses from a validated input. `data_frame()` for a fast data frame creation helper.

Examples

```
# `new_data_frame()` can be used to create custom data frame constructors
new_fancy_df <- function(x = list(), n = NULL, ..., class = NULL) {
  new_data_frame(x, n = n, ..., class = c(class, "fancy_df"))
}

# Combine this constructor with `df_list()` to create a safe,
# consistent helper function for your data frame subclass
fancy_df <- function(...) {
  data <- df_list(...)
  new_fancy_df(data)
}

df <- fancy_df(x = 1)
class(df)
```

df_ptype2

Coercion between two data frames

Description

`df_ptype2()` and `df_cast()` are the two functions you need to call from `vec_ptype2()` and `vec_cast()` methods for data frame subclasses. See [?howto-faq-coercion-data-frame](#). Their main job is to determine the common type of two data frames, adding and coercing columns as needed, or throwing an incompatible type error when the columns are not compatible.

Usage

```
df_ptype2(x, y, ..., x_arg = "", y_arg = "", call = caller_env())

df_cast(x, to, ..., x_arg = "", to_arg = "", call = caller_env())

tib_ptype2(x, y, ..., x_arg = "", y_arg = "", call = caller_env())

tib_cast(x, to, ..., x_arg = "", to_arg = "", call = caller_env())
```

Arguments

<code>x, y, to</code>	Subclasses of data frame.
<code>...</code>	If you call <code>df_ptype2()</code> or <code>df_cast()</code> from a <code>vec_ptype2()</code> or <code>vec_cast()</code> method, you must forward the dots passed to your method on to <code>df_ptype2()</code> or <code>df_cast()</code> .

<code>x_arg, 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 <code>stop_incompatible_type()</code>).
<code>call</code>	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of <code>abort()</code> for more information.
<code>to_arg</code>	Argument name to used in error messages to inform the user about the locations of incompatible types (see <code>stop_incompatible_type()</code>).

Value

- When `x` and `y` are not compatible, an error of class `vctrs_error_incompatible_type` is thrown.
- When `x` and `y` are compatible, `df_ptype2()` returns the common type as a bare data frame. `tib_ptype2()` returns the common type as a bare tibble.

faq-compatibility-types

FAQ - How is the compatibility of vector types decided?

Description

Two vectors are **compatible** when you can safely:

- Combine them into one larger vector.
- Assign values from one of the vectors into the other vector.

Examples of compatible types are integer and double vectors. On the other hand, integer and character vectors are not compatible.

Common type of multiple vectors

There are two possible outcomes when multiple vectors of different types are combined into a larger vector:

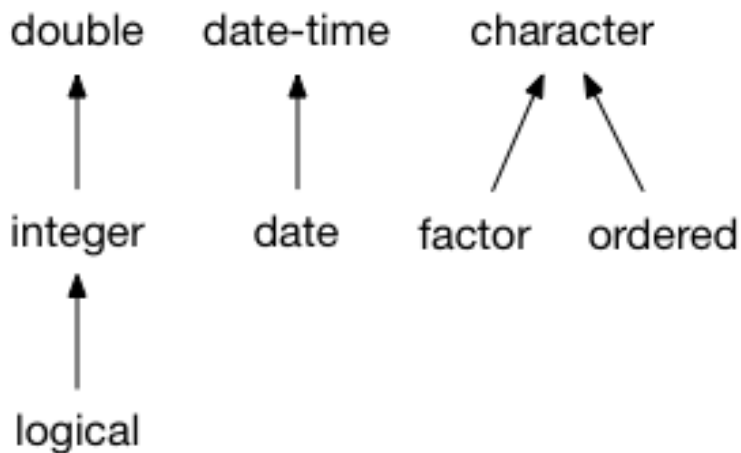
- An incompatible type error is thrown because some of the types are not compatible:

```
df1 <- data.frame(x = 1:3)
df2 <- data.frame(x = "foo")
dplyr::bind_rows(df1, df2)
#> Error in `dplyr::bind_rows()` :
#> ! Can't combine `..1$x` <integer> and `..2$x` <character>.
```

- The vectors are combined into a vector that has the common type of all inputs. In this example, the common type of integer and logical is integer:

```
df1 <- data.frame(x = 1:3)
df2 <- data.frame(x = FALSE)
dplyr::bind_rows(df1, df2)
#>   x
#> 1 1
#> 2 2
#> 3 3
#> 4 0
```

In general, the common type is the *richer* type, in other words the type that can represent the most values. Logical vectors are at the bottom of the hierarchy of numeric types because they can only represent two values (not counting missing values). Then come integer vectors, and then doubles. Here is the vctrs type hierarchy for the fundamental vectors:



Type conversion and lossy cast errors

Type compatibility does not necessarily mean that you can **convert** one type to the other type. That's because one of the types might support a larger set of possible values. For instance, integer and double vectors are compatible, but double vectors can't be converted to integer if they contain fractional values.

When vctrs can't convert a vector because the target type is not as rich as the source type, it throws a lossy cast error. Assigning a fractional number to an integer vector is a typical example of a lossy cast error:

```
int_vector <- 1:3
vec_assign(int_vector, 2, 0.001)
#> Error in `vec_assign()`:
#> ! Can't convert from <double> to <integer> due to loss of precision.
#> * Locations: 1
```


How to make two vector classes compatible?

If you encounter two vector types that you think should be compatible, they might need to implement coercion methods. Reach out to the author(s) of the classes and ask them if it makes sense for their classes to be compatible.

These developer FAQ items provide guides for implementing coercion methods:

- For an example of implementing coercion methods for simple vectors, see [?howto-faq-coercion](#).
- For an example of implementing coercion methods for data frame subclasses, see [?howto-faq-coercion-data-frame](#).

faq-error-incompatible-attributes

FAQ - Error/Warning: Some attributes are incompatible

Description

This error occurs when `vec_ptype2()` or `vec_cast()` are supplied vectors of the same classes with different attributes. In this case, `vctrs` doesn't know how to combine the inputs.

To fix this error, the maintainer of the class should implement self-to-self coercion methods for `vec_ptype2()` and `vec_cast()`.

Implementing coercion methods

- For an overview of how these generics work and their roles in `vctrs`, see [?theory-faq-coercion](#).
- For an example of implementing coercion methods for simple vectors, see [?howto-faq-coercion](#).
- For an example of implementing coercion methods for data frame subclasses, see [?howto-faq-coercion-data-frame](#).
- For a tutorial about implementing `vctrs` classes from scratch, see `vignette("s3-vector")`.

faq-error-scalar-type *FAQ - Error: Input must be a vector*

Description

This error occurs when a function expects a vector and gets a scalar object instead. This commonly happens when some code attempts to assign a scalar object as column in a data frame:

```
fn <- function() NULL
tibble::tibble(x = fn)
#> Error in `tibble::tibble()`:
#> ! All columns in a tibble must be vectors.
#> x Column `x` is a function.
```

```
fit <- lm(1:3 ~ 1)
tibble::tibble(x = fit)
#> Error in `tibble::tibble()`:
#> ! All columns in a tibble must be vectors.
#> x Column `x` is a `lm` object.
```

Vectoriness in base R and in the tidyverse

In base R, almost everything is a vector or behaves like a vector. In the tidyverse we have chosen to be a bit stricter about what is considered a vector. The main question we ask ourselves to decide on the vectoriness of a type is whether it makes sense to include that object as a column in a data frame.

The main difference is that S3 lists are considered vectors by base R but in the tidyverse that's not the case by default:

```
fit <- lm(1:3 ~ 1)

typeof(fit)
#> [1] "list"
class(fit)
#> [1] "lm"

# S3 lists can be subset like a vector using base R:
fit[c(1, 4)]
#> $coefficients
#> (Intercept)
#>          2
#>
#> $rank
#> [1] 1

# But not in vctrs
vctrs::vec_slice(fit, c(1, 4))
#> Error in `vctrs::vec_slice()`:
#> !`x` must be a vector, not a <lm> object.
```

Defused function calls are another (more esoteric) example:

```
call <- quote(foo(bar = TRUE, baz = FALSE))
call
#> foo(bar = TRUE, baz = FALSE)

# They can be subset like a vector using base R:
call[1:2]
#> foo(bar = TRUE)
lapply(call, function(x) x)
#> [[1]]
#> foo
#>
#> $bar
#> [1] TRUE
#>
#> $baz
#> [1] FALSE
```

```
# But not with vctrs:
vctrs::vec_slice(call, 1:2)
#> Error in `vctrs::vec_slice()`:
#> ! `x` must be a vector, not a call.
```

I get a scalar type error but I think this is a bug

It's possible the author of the class needs to do some work to declare their class a vector. Consider reaching out to the author. We have written a [developer FAQ page](#) to help them fix the issue.

howto-faq-coercion *FAQ - How to implement ptype2 and cast methods?*

Description

This guide illustrates how to implement `vec_ptype2()` and `vec_cast()` methods for existing classes. Related topics:

- For an overview of how these generics work and their roles in vctrs, see [?theory-faq-coercion](#).
- For an example of implementing coercion methods for data frame subclasses, see [?howto-faq-coercion-data-frame](#).
- For a tutorial about implementing vctrs classes from scratch, see `vignette("s3-vector")`

The natural number class:

We'll illustrate how to implement coercion methods with a simple class that represents natural numbers. In this scenario we have an existing class that already features a constructor and methods for `print()` and `subset`.

```
## @export
new_natural <- function(x) {
  if (is.numeric(x) || is.logical(x)) {
    stopifnot(is_whole(x))
    x <- as.integer(x)
  } else {
    stop("Can't construct natural from unknown type.")
  }
  structure(x, class = "my_natural")
}

is_whole <- function(x) {
  all(x %% 1 == 0 | is.na(x))
}

## @export
print.my_natural <- function(x, ...) {
  cat("<natural>\n")
  x <- unclass(x)
  NextMethod()
}
```

```

#' @export
`[.my_natural` <- function(x, i, ...) {
  new_natural(NextMethod())
}

new_natural(1:3)
#> <natural>
#> [1] 1 2 3
new_natural(c(1, NA))
#> <natural>
#> [1] 1 NA

```

Roxygen workflow:

To implement methods for generics, first import the generics in your namespace and redocument:

```

#' @importFrom vctrs vec_ptype2 vec_cast
NULL

```

Note that for each batches of methods that you add to your package, you need to export the methods and redocument immediately, even during development. Otherwise they won't be in scope when you run unit tests e.g. with `testthat`.

Implementing double dispatch methods is very similar to implementing regular S3 methods. In these examples we are using `roxygen2` tags to register the methods, but you can also register the methods manually in your `NAMESPACE` file or lazily with `s3_register()`.

Implementing `vec_ptype2()`:

The self-self method:

The first method to implement is the one that signals that your class is compatible with itself:

```

#' @export
vec_ptype2.my_natural.my_natural <- function(x, y, ...) {
  x
}

```

```

vec_ptype2(new_natural(1), new_natural(2:3))
#> <natural>
#> integer(0)

```

`vec_ptype2()` implements a fallback to try and be compatible with simple classes, so it may seem that you don't need to implement the self-self coercion method. However, you must implement it explicitly because this is how `vctrs` knows that a class that is implementing `vctrs` methods (for instance this disable fallbacks to `base::c()`). Also, it makes your class a bit more efficient.

The parent and children methods:

Our natural number class is conceptually a parent of `<logical>` and a child of `<integer>`, but the class is not compatible with `logical`, `integer`, or `double` vectors yet:

```

vec_ptype2(TRUE, new_natural(2:3))
#> Error:
#> ! Can't combine `TRUE` <logical> and `new_natural(2:3)` <my_natural>.

```

```
vec_ptype2(new_natural(1), 2:3)
#> Error:
#> ! Can't combine `new_natural(1)` <my_natural> and `2:3` <integer>.
```

We'll specify the twin methods for each of these classes, returning the richer class in each case.

```
#' @export
vec_ptype2.my_natural.logical <- function(x, y, ...) {
  # The order of the classes in the method name follows the order of
  # the arguments in the function signature, so `x` is the natural
  # number and `y` is the logical
  x
}
#' @export
vec_ptype2.logical.my_natural <- function(x, y, ...) {
  # In this case `y` is the richer natural number
  y
}
```

Between a natural number and an integer, the latter is the richer class:

```
#' @export
vec_ptype2.my_natural.integer <- function(x, y, ...) {
  y
}
#' @export
vec_ptype2.integer.my_natural <- function(x, y, ...) {
  x
}
```

We no longer get common type errors for logical and integer:

```
vec_ptype2(TRUE, new_natural(2:3))
#> <natural>
#> integer(0)
```

```
vec_ptype2(new_natural(1), 2:3)
#> integer(0)
```

We are not done yet. Pairwise coercion methods must be implemented for all the connected nodes in the coercion hierarchy, which include double vectors further up. The coercion methods for grand-parent types must be implemented separately:

```
#' @export
vec_ptype2.my_natural.double <- function(x, y, ...) {
  y
}
#' @export
vec_ptype2.double.my_natural <- function(x, y, ...) {
  x
}
```

Incompatible attributes:

Most of the time, inputs are incompatible because they have different classes for which no `vec_ptype2()` method is implemented. More rarely, inputs could be incompatible because of their attributes. In that case incompatibility is signalled by calling `stop_incompatible_type()`.

In the following example, we implement a self-self `ptype2` method for a hypothetical subclass of `<factor>` that has stricter combination semantics. The method throws an error when the levels of the two factors are not compatible.

```
#' @export
vec_ptype2.my_strict_factor.my_strict_factor <- function(x, y, ..., x_arg = "", y_arg = "") {
  if (!setequal(levels(x), levels(y))) {
    stop_incompatible_type(x, y, x_arg = x_arg, y_arg = y_arg)
  }

  x
}
```

Note how the methods need to take `x_arg` and `y_arg` parameters and pass them on to `stop_incompatible_type()`. These argument tags help create more informative error messages when the common type determination is for a column of a data frame. They are part of the generic signature but can usually be left out if not used.

Implementing `vec_cast()`:

Corresponding `vec_cast()` methods must be implemented for all `vec_ptype2()` methods. The general pattern is to convert the argument `x` to the type of `to`. The methods should validate the values in `x` and make sure they conform to the values of `to`.

Please note that for historical reasons, the order of the classes in the method name is in reverse order of the arguments in the function signature. The first class represents `to`, whereas the second class represents `x`.

The self-self method is easy in this case, it just returns the target input:

```
#' @export
vec_cast.my_natural.my_natural <- function(x, to, ...) {
  x
}
```

The other types need to be validated. We perform input validation in the `new_natural()` constructor, so that's a good fit for our `vec_cast()` implementations.

```
#' @export
vec_cast.my_natural.logical <- function(x, to, ...) {
  # The order of the classes in the method name is in reverse order
  # of the arguments in the function signature, so `to` is the natural
  # number and `x` is the logical
  new_natural(x)
}
vec_cast.my_natural.integer <- function(x, to, ...) {
  new_natural(x)
}
vec_cast.my_natural.double <- function(x, to, ...) {
  new_natural(x)
}
```

With these methods, `vctrs` is now able to combine logical and natural vectors. It properly returns the richer type of the two, a natural vector:

```
vec_c(TRUE, new_natural(1), FALSE)
#> <natural>
#> [1] 1 1 0
```

Because we haven't implemented conversions *from* natural, it still doesn't know how to combine natural with the richer integer and double types:

```
vec_c(new_natural(1), 10L)
#> Error in `vec_c()` :
#> ! Can't convert `..1` <my_natural> to <integer>.
vec_c(1.5, new_natural(1))
#> Error in `vec_c()` :
#> ! Can't convert `..2` <my_natural> to <double>.
```

This is quick work which completes the implementation of coercion methods for vctrs:

```
#' @export
vec_cast.logical.my_natural <- function(x, to, ...) {
  # In this case `to` is the logical and `x` is the natural number
  attributes(x) <- NULL
  as.logical(x)
}
#' @export
vec_cast.integer.my_natural <- function(x, to, ...) {
  attributes(x) <- NULL
  as.integer(x)
}
#' @export
vec_cast.double.my_natural <- function(x, to, ...) {
  attributes(x) <- NULL
  as.double(x)
}
```

And we now get the expected combinations.

```
vec_c(new_natural(1), 10L)
#> [1] 1 10

vec_c(1.5, new_natural(1))
#> [1] 1.5 1.0
```

Description

This guide provides a practical recipe for implementing `vec_ptype2()` and `vec_cast()` methods for coercions of data frame subclasses. Related topics:

- For an overview of the coercion mechanism in `vctrs`, see [?theory-faq-coercion](#).
- For an example of implementing coercion methods for simple vectors, see [?howto-faq-coercion](#).

Coercion of data frames occurs when different data frame classes are combined in some way. The two main methods of combination are currently row-binding with `vec_rbind()` and col-binding with `vec_cbind()` (which are in turn used by a number of `dplyr` and `tidyr` functions). These functions take multiple data frame inputs and automatically coerce them to their common type.

`vctrs` is generally strict about the kind of automatic coercions that are performed when combining inputs. In the case of data frames we have decided to be a bit less strict for convenience. Instead of throwing an incompatible type error, we fall back to a base data frame or a tibble if we don't know how to combine two data frame subclasses. It is still a good idea to specify the proper coercion behaviour for your data frame subclasses as soon as possible.

We will see two examples in this guide. The first example is about a data frame subclass that has no particular attributes to manage. In the second example, we implement coercion methods for a tibble subclass that includes potentially incompatible attributes.

Roxygen workflow:

To implement methods for generics, first import the generics in your namespace and redocument:

```
#' @importFrom vctrs vec_ptype2 vec_cast
NULL
```

Note that for each batches of methods that you add to your package, you need to export the methods and redocument immediately, even during development. Otherwise they won't be in scope when you run unit tests e.g. with `testthat`.

Implementing double dispatch methods is very similar to implementing regular S3 methods. In these examples we are using `roxygen2` tags to register the methods, but you can also register the methods manually in your `NAMESPACE` file or lazily with `s3_register()`.

Parent methods:

Most of the common type determination should be performed by the parent class. In `vctrs`, double dispatch is implemented in such a way that you need to call the methods for the parent class manually. For `vec_ptype2()` this means you need to call `df_ptype2()` (for data frame subclasses) or `tib_ptype2()` (for tibble subclasses). Similarly, `df_cast()` and `tib_cast()` are the workhorses for `vec_cast()` methods of subtypes of `data.frame` and `tbl_df`. These functions take the union of the columns in `x` and `y`, and ensure shared columns have the same type.

These functions are much less strict than `vec_ptype2()` and `vec_cast()` as they accept any subclass of data frame as input. They always return a `data.frame` or a `tbl_df`. You will probably want to write similar functions for your subclass to avoid repetition in your code. You may want to export them as well if you are expecting other people to derive from your class.

A data.table example:

This example is the actual implementation of `vctrs` coercion methods for `data.table`. This is a simple example because we don't have to keep track of attributes for this class or manage incompatibilities. See the tibble section for a more complicated example.

We first create the `dt_ptype2()` and `dt_cast()` helpers. They wrap around the parent methods `df_ptype2()` and `df_cast()`, and transform the common type or converted input to a data table. You may want to export these helpers if you expect other packages to derive from your data frame class.

These helpers should always return data tables. To this end we use the conversion generic `as.data.table()`. Depending on the tools available for the particular class at hand, a constructor might be appropriate as well.

```
dt_ptype2 <- function(x, y, ...) {
  as.data.table(df_ptype2(x, y, ...))
}
dt_cast <- function(x, to, ...) {
  as.data.table(df_cast(x, to, ...))
}
```

We start with the self-self method:

```
#' @export
vec_ptype2.data.table.data.table <- function(x, y, ...) {
  dt_ptype2(x, y, ...)
}
```

Between a data frame and a data table, we consider the richer type to be data table. This decision is not based on the value coverage of each data structures, but on the idea that data tables have richer behaviour. Since data tables are the richer type, we call `dt_ptype2()` from the `vec_ptype2()` method. It always returns a data table, no matter the order of arguments:

```
#' @export
vec_ptype2.data.table.data.frame <- function(x, y, ...) {
  dt_ptype2(x, y, ...)
}
#' @export
vec_ptype2.data.frame.data.table <- function(x, y, ...) {
  dt_ptype2(x, y, ...)
}
```

The `vec_cast()` methods follow the same pattern, but note how the method for coercing to data frame uses `df_cast()` rather than `dt_cast()`.

Also, please note that for historical reasons, the order of the classes in the method name is in reverse order of the arguments in the function signature. The first class represents `to`, whereas the second class represents `x`.

```
#' @export
vec_cast.data.table.data.table <- function(x, to, ...) {
  dt_cast(x, to, ...)
}
#' @export
vec_cast.data.table.data.frame <- function(x, to, ...) {
  # `x` is a data.frame to be converted to a data.table
  dt_cast(x, to, ...)
}
```

```
#' @export
vec_cast.data.frame.data.table <- function(x, to, ...) {
  # `x` is a data.table to be converted to a data.frame
  df_cast(x, to, ...)
}
```

With these methods `vctrs` is now able to combine data tables with data frames:

```
vec_cbind(data.frame(x = 1:3), data.table(y = "foo"))
#>      x      y
#>   <int> <char>
#> 1:     1   foo
#> 2:     2   foo
#> 3:     3   foo
```

A tibble example:

In this example we implement coercion methods for a tibble subclass that carries a colour as a scalar metadata:

```
# User constructor
my_tibble <- function(colour = NULL, ...) {
  new_my_tibble(tibble::tibble(...), colour = colour)
}
# Developer constructor
new_my_tibble <- function(x, colour = NULL) {
  stopifnot(is.data.frame(x))
  tibble::new_tibble(
    x,
    colour = colour,
    class = "my_tibble",
    nrow = nrow(x)
  )
}

df_colour <- function(x) {
  if (inherits(x, "my_tibble")) {
    attr(x, "colour")
  } else {
    NULL
  }
}

#'@export
print.my_tibble <- function(x, ...) {
  cat(sprintf("<%s: %s>\n", class(x)[[1]], df_colour(x)))
  cli::cat_line(format(x)[-1])
}
```

This subclass is very simple. All it does is modify the header.

```

red <- my_tibble("red", x = 1, y = 1:2)
red
#> <my_tibble: red>
#>       x     y
#>   <dbl> <int>
#> 1     1     1
#> 2     1     2

red[2]
#> <my_tibble: red>
#>       y
#>   <int>
#> 1     1
#> 2     2

green <- my_tibble("green", z = TRUE)
green
#> <my_tibble: green>
#>       z
#>   <lgl>
#> 1 TRUE

```

Combinations do not work properly out of the box, instead `vctrs` falls back to a bare tibble:

```

vec_rbind(red, tibble::tibble(x = 10:12))
#> # A tibble: 5 x 2
#>       x     y
#>   <dbl> <int>
#> 1     1     1
#> 2     1     2
#> 3    10    NA
#> 4    11    NA
#> 5    12    NA

```

Instead of falling back to a data frame, we would like to return a `<my_tibble>` when combined with a data frame or a tibble. Because this subclass has more metadata than normal data frames (it has a colour), it is a *supertype* of tibble and data frame, i.e. it is the richer type. This is similar to how a grouped tibble is a more general type than a tibble or a data frame. Conceptually, the latter are pinned to a single constant group.

The coercion methods for data frames operate in two steps:

- They check for compatible subclass attributes. In our case the tibble colour has to be the same, or be undefined.
- They call their parent methods, in this case `tib_ptype2()` and `tib_cast()` because we have a subclass of tibble. This eventually calls the data frame methods `df_ptype2()` and `tib_ptype2()` which match the columns and their types.

This process should usually be wrapped in two functions to avoid repetition. Consider exporting these if you expect your class to be derived by other subclasses.

We first implement a helper to determine if two data frames have compatible colours. We use the `df_colour()` accessor which returns `NULL` when the data frame colour is undefined.

```

has_compatible_colours <- function(x, y) {
  x_colour <- df_colour(x) %||% df_colour(y)
  y_colour <- df_colour(y) %||% x_colour
  identical(x_colour, y_colour)
}

```

Next we implement the coercion helpers. If the colours are not compatible, we call `stop_incompatible_cast()` or `stop_incompatible_type()`. These strict coercion semantics are justified because in this class `colour` is a *data* attribute. If it were a non essential *detail* attribute, like the `timezone` in a `datetime`, we would just standardise it to the value of the left-hand side.

In simpler cases (like the `data.table` example), these methods do not need to take the arguments suffixed in `_arg`. Here we do need to take these arguments so we can pass them to the `stop_` functions when we detect an incompatibility. They also should be passed to the parent methods.

```

#' @export
my_tib_cast <- function(x, to, ..., x_arg = "", to_arg = "") {
  out <- tib_cast(x, to, ..., x_arg = x_arg, to_arg = to_arg)

  if (!has_compatible_colours(x, to)) {
    stop_incompatible_cast(
      x,
      to,
      x_arg = x_arg,
      to_arg = to_arg,
      details = "Can't combine colours."
    )
  }

  colour <- df_colour(x) %||% df_colour(to)
  new_my_tibble(out, colour = colour)
}

#' @export
my_tib_ptype2 <- function(x, y, ..., x_arg = "", y_arg = "") {
  out <- tib_ptype2(x, y, ..., x_arg = x_arg, y_arg = y_arg)

  if (!has_compatible_colours(x, y)) {
    stop_incompatible_type(
      x,
      y,
      x_arg = x_arg,
      y_arg = y_arg,
      details = "Can't combine colours."
    )
  }

  colour <- df_colour(x) %||% df_colour(y)
  new_my_tibble(out, colour = colour)
}

```

Let's now implement the coercion methods, starting with the self-self methods.

```

#' @export
vec_ptype2.my_tibble.my_tibble <- function(x, y, ...) {
  my_tib_ptype2(x, y, ...)
}
#' @export
vec_cast.my_tibble.my_tibble <- function(x, to, ...) {
  my_tib_cast(x, to, ...)
}

```

We can now combine compatible instances of our class!

```

vec_rbind(red, red)
#> <my_tibble: red>
#>       x     y
#>   <dbl> <int>
#> 1     1     1
#> 2     1     2
#> 3     1     1
#> 4     1     2

```

```

vec_rbind(green, green)
#> <my_tibble: green>
#>       z
#>   <lgl>
#> 1 TRUE
#> 2 TRUE

```

```

vec_rbind(green, red)
#> Error in `my_tib_ptype2()`:
#> ! Can't combine `..1` <my_tibble> and `..2` <my_tibble>.
#> Can't combine colours.

```

The methods for combining our class with tibbles follow the same pattern. For `ptype2` we return our class in both cases because it is the richer type:

```

#' @export
vec_ptype2.my_tibble.tbl_df <- function(x, y, ...) {
  my_tib_ptype2(x, y, ...)
}
#' @export
vec_ptype2.tbl_df.my_tibble <- function(x, y, ...) {
  my_tib_ptype2(x, y, ...)
}

```

For `cast` are careful about returning a tibble when casting to a tibble. Note the call to `vctrs::tib_cast()`:

```

#' @export
vec_cast.my_tibble.tbl_df <- function(x, to, ...) {
  my_tib_cast(x, to, ...)
}
#' @export

```

```
vec_cast.tbl_df.my_tibble <- function(x, to, ...) {
  tib_cast(x, to, ...)
}
```

From this point, we get correct combinations with tibbles:

```
vec_rbind(red, tibble::tibble(x = 10:12))
#> <my_tibble: red>
#>       x     y
#>   <dbl> <int>
#> 1     1     1
#> 2     1     2
#> 3    10    NA
#> 4    11    NA
#> 5    12    NA
```

However we are not done yet. Because the coercion hierarchy is different from the class hierarchy, there is no inheritance of coercion methods. We're not getting correct behaviour for data frames yet because we haven't explicitly specified the methods for this class:

```
vec_rbind(red, data.frame(x = 10:12))
#> # A tibble: 5 x 2
#>       x     y
#>   <dbl> <int>
#> 1     1     1
#> 2     1     2
#> 3    10    NA
#> 4    11    NA
#> 5    12    NA
```

Let's finish up the boiler plate:

```
#' @export
vec_ptype2.my_tibble.data.frame <- function(x, y, ...) {
  my_tib_ptype2(x, y, ...)
}
#' @export
vec_ptype2.data.frame.my_tibble <- function(x, y, ...) {
  my_tib_ptype2(x, y, ...)
}

#' @export
vec_cast.my_tibble.data.frame <- function(x, to, ...) {
  my_tib_cast(x, to, ...)
}
#' @export
vec_cast.data.frame.my_tibble <- function(x, to, ...) {
  df_cast(x, to, ...)
}
```

This completes the implementation:

```
vec_rbind(red, data.frame(x = 10:12))
#> <my_tibble: red>
#>       x     y
#>   <dbl> <int>
#> 1     1     1
#> 2     1     2
#> 3    10    NA
#> 4    11    NA
#> 5    12    NA
```

howto-faq-fix-scalar-type-error

FAQ - Why isn't my class treated as a vector?

Description

The tidyverse is a bit stricter than base R regarding what kind of objects are considered as vectors (see the [user FAQ](#) about this topic). Sometimes `vctrs` won't treat your class as a vector when it should.

Why isn't my list class considered a vector?:

By default, S3 lists are not considered to be vectors by `vctrs`:

```
my_list <- structure(list(), class = "my_class")
```

```
vctrs::vec_is(my_list)
#> [1] FALSE
```

To be treated as a vector, the class must either inherit from "list" explicitly:

```
my_explicit_list <- structure(list(), class = c("my_class", "list"))
vctrs::vec_is(my_explicit_list)
#> [1] TRUE
```

Or it should implement a `vec_proxy()` method that returns its input if explicit inheritance is not possible or troublesome:

```
#' @export
vec_proxy.my_class <- function(x, ...) x
```

```
vctrs::vec_is(my_list)
#> [1] FALSE
```

Note that explicit inheritance is the preferred way because this makes it possible for your class to dispatch on list methods of S3 generics:

```
my_generic <- function(x) UseMethod("my_generic")
my_generic.list <- function(x) "dispatched!"
```

```
my_generic(my_list)
#> Error in UseMethod("my_generic"): no applicable method for 'my_generic' applied to an object of class
```

```
my_generic(my_explicit_list)
#> [1] "dispatched!"
```

Why isn't my data frame class considered a vector?:

The most likely explanation is that the data frame has not been properly constructed.

However, if you get an “Input must be a vector” error with a data frame subclass, it probably means that the data frame has not been properly constructed. The main cause of these errors are data frames whose *base class* is not “data.frame”:

```
my_df <- data.frame(x = 1)
class(my_df) <- c("data.frame", "my_class")

vctrs::obj_check_vector(my_df)
#> Error:
#> ! `my_df` must be a vector, not a <data.frame/my_class> object.
```

This is problematic as many tidyverse functions won't work properly:

```
dplyr::slice(my_df, 1)
#> Error in `vec_slice()`:
#> ! `x` must be a vector, not a <data.frame/my_class> object.
```

It is generally not appropriate to declare your class to be a superclass of another class. We generally consider this undefined behaviour (UB). To fix these errors, you can simply change the construction of your data frame class so that “data.frame” is a base class, i.e. it should come last in the class vector:

```
class(my_df) <- c("my_class", "data.frame")

vctrs::obj_check_vector(my_df)

dplyr::slice(my_df, 1)
#>   x
#> 1 1
```

internal-faq-matches-algorithm

Internal FAQ - Implementation of vec_locate_matches()

Description

`vec_locate_matches()` is similar to `vec_match()`, but detects *all* matches by default, and can match on conditions other than equality (like `>=` and `<`). There are also various other arguments to limit or adjust exactly which kinds of matches are returned. Here is an example:


```

x <- c("a", "b", "a", "c", "d")
y <- c("d", "b", "a", "d", "a", "e")

# For each value of `x`, find all matches in `y`
# - The "c" in `x` doesn't have a match, so it gets an NA location by default
# - The "e" in `y` isn't matched by anything in `x`, so it is dropped by default
vec_locate_matches(x, y)
#>   needles haystack
#> 1      1      3
#> 2      1      5
#> 3      2      2
#> 4      3      3
#> 5      3      5
#> 6      4     NA
#> 7      5      1
#> 8      5      4

```

Algorithm description

Overview and ==:

The simplest (approximate) way to think about the algorithm that `df_locate_matches_recurse()` uses is that it sorts both inputs, and then starts at the midpoint in `needles` and uses a binary search to find each needle in `haystack`. Since there might be multiple of the same needle, we find the location of the lower and upper duplicate of that needle to handle all duplicates of that needle at once. Similarly, if there are duplicates of a matching haystack value, we find the lower and upper duplicates of the match.

If the condition is `==`, that is pretty much all we have to do. For each needle, we then record 3 things: the location of the needle, the location of the lower match in the haystack, and the match size (i.e. `loc_upper_match - loc_lower_match + 1`). This later gets expanded in `expand_compact_indices()` into the actual output.

After recording the matches for a single needle, we perform the same procedure on the LHS and RHS of that needle (remember we started on the midpoint needle). i.e. from `[1, loc_needle-1]` and `[loc_needle+1, size_needles]`, again taking the midpoint of those two ranges, finding their respective needle in the haystack, recording matches, and continuing on to the next needle. This iteration proceeds until we run out of needles.

When we have a data frame with multiple columns, we add a layer of recursion to this. For the first column, we find the locations of the lower/upper duplicate of the current needle, and we find the locations of the lower/upper matches in the haystack. If we are on the final column in the data frame, we record the matches, otherwise we pass this information on to another call to `df_locate_matches_recurse()`, bumping the column index and using these refined lower/upper bounds as the starting bounds for the next column.

I think an example would be useful here, so below I step through this process for a few iterations:

```

# these are sorted already for simplicity
needles <- data_frame(x = c(1, 1, 2, 2, 2, 3), y = c(1, 2, 3, 4, 5, 3))
haystack <- data_frame(x = c(1, 1, 2, 2, 3), y = c(2, 3, 4, 4, 1))

needles

```

```
#>  x y
#> 1 1 1
#> 2 1 2
#> 3 2 3
#> 4 2 4
#> 5 2 5
#> 6 3 3

haystack
#>  x y
#> 1 1 2
#> 2 1 3
#> 3 2 4
#> 4 2 4
#> 5 3 1

## Column 1, iteration 1

# start at midpoint in needles
# this corresponds to x==2
loc_mid_needles <- 3L

# finding all x==2 values in needles gives us:
loc_lower_duplicate_needles <- 3L
loc_upper_duplicate_needles <- 5L

# finding matches in haystack give us:
loc_lower_match_haystack <- 3L
loc_upper_match_haystack <- 4L

# compute LHS/RHS bounds for next needle
lhs_loc_lower_bound_needles <- 1L # original lower bound
lhs_loc_upper_bound_needles <- 2L # lower_duplicate-1

rhs_loc_lower_bound_needles <- 6L # upper_duplicate+1
rhs_loc_upper_bound_needles <- 6L # original upper bound

# We still have a 2nd column to check. So recurse and pass on the current
# duplicate and match bounds to start the 2nd column with.

## Column 2, iteration 1

# midpoint of [3, 5]
# value y==4
loc_mid_needles <- 4L

loc_lower_duplicate_needles <- 4L
loc_upper_duplicate_needles <- 4L
```

```
loc_lower_match_haystack <- 3L
loc_upper_match_haystack <- 4L

# last column, so record matches
# - this was location 4 in needles
# - lower match in haystack is at loc 3
# - match size is 2

# Now handle LHS and RHS of needle midpoint
lhs_loc_lower_bound_needles <- 3L # original lower bound
lhs_loc_upper_bound_needles <- 3L # lower_duplicate-1

rhs_loc_lower_bound_needles <- 5L # upper_duplicate+1
rhs_loc_upper_bound_needles <- 5L # original upper bound

## Column 2, iteration 2 (using LHS bounds)

# midpoint of [3,3]
# value of y==3
loc_mid_needles <- 3L

loc_lower_duplicate_needles <- 3L
loc_upper_duplicate_needles <- 3L

# no match! no y==3 in haystack for x==2
# lower-match will always end up > upper-match in this case
loc_lower_match_haystack <- 3L
loc_upper_match_haystack <- 2L

# no LHS or RHS needle values to do, so we are done here

## Column 2, iteration 3 (using RHS bounds)

# same as above, range of [5,5], value of y==5, which has no match in haystack

## Column 1, iteration 2 (LHS of first x needle)

# Now we are done with the x needles from [3,5], so move on to the LHS and RHS
# of that. Here we would do the LHS:

# midpoint of [1,2]
loc_mid_needles <- 1L

# ...

## Column 1, iteration 3 (RHS of first x needle)
```

```
# midpoint of [6,6]
loc_mid_needles <- 6L
```

```
# ...
```

In the real code, rather than comparing the double values of the columns directly, we replace each column with pseudo "joint ranks" computed between the i -th column of `needles` and the i -th column of `haystack`. It is approximately like doing `vec_rank(vec_c(needles$x, haystack$x), type = "dense")`, then splitting the resulting ranks back up into their corresponding needle/haystack columns. This keeps the recursion code simpler, because we only have to worry about comparing integers.

Non-equi conditions and containers:

At this point we can talk about non-equi conditions like `<` or `>=`. The general idea is pretty simple, and just builds on the above algorithm. For example, start with the `x` column from `needles/haystack` above:

```
needles$x
#> [1] 1 1 2 2 2 3
```

```
haystack$x
#> [1] 1 1 2 2 3
```

If we used a condition of `<=`, then we'd do everything the same as before:

- Midpoint in `needles` is location 3, value `x==2`
- Find lower/upper duplicates in `needles`, giving locations `[3, 5]`
- Find lower/upper *exact* match in `haystack`, giving locations `[3, 4]`

At this point, we need to "adjust" the `haystack` match bounds to account for the condition. Since `haystack` is ordered, our "rule" for `<=` is to keep the lower match location the same, but extend the upper match location to the upper bound, so we end up with `[3, 5]`. We know we can extend the upper match location because every `haystack` value after the exact match should be less than the needle. Then we just record the matches and continue on normally.

This approach is really nice, because we only have to exactly match the needle in `haystack`. We don't have to compare each needle against every value in `haystack`, which would take a massive amount of time.

However, it gets slightly more complex with data frames with multiple columns. Let's go back to our original `needles` and `haystack` data frames and apply the condition `<=` to each column. Here is another worked example, which shows a case where our "rule" falls apart on the second column.

```
needles
#>   x y
#> 1 1 1
#> 2 1 2
#> 3 2 3
#> 4 2 4
#> 5 2 5
#> 6 3 3
```

```
haystack
#>   x y
#> 1 1 2
#> 2 1 3
#> 3 2 4
#> 4 2 4
#> 5 3 1

# `condition = c("<=", "<=")`

## Column 1, iteration 1

# x == 2
loc_mid_needles <- 3L

loc_lower_duplicate_needles <- 3L
loc_upper_duplicate_needles <- 5L

# finding exact matches in haystack give us:
loc_lower_match_haystack <- 3L
loc_upper_match_haystack <- 4L

# because haystack is ordered we know we can expand the upper bound automatically
# to include everything past the match. i.e. needle of x==2 must be less than
# the haystack value at loc 5, which we can check by seeing that it is x==3.
loc_lower_match_haystack <- 3L
loc_upper_match_haystack <- 5L

## Column 2, iteration 1

# needles range of [3, 5]
# y == 4
loc_mid_needles <- 4L

loc_lower_duplicate_needles <- 4L
loc_upper_duplicate_needles <- 4L

# finding exact matches in haystack give us:
loc_lower_match_haystack <- 3L
loc_upper_match_haystack <- 4L

# lets try using our rule, which tells us we should be able to extend the upper
# bound:
loc_lower_match_haystack <- 3L
loc_upper_match_haystack <- 5L

# but the haystack value of y at location 5 is y==1, which is not less than y==4
# in the needles! looks like our rule failed us.
```

If you read through the above example, you'll see that the rule didn't work here. The problem is that while haystack is ordered (by `vec_order()`'s standards), each column isn't ordered *independently* of the others. Instead, each column is ordered within the "group" created by previous columns. Concretely, haystack here has an ordered `x` column, but if you look at `haystack$y` by itself, it isn't ordered (because of that 1 at the end). That is what causes the rule to fail.

```
haystack
#>   x y
#> 1 1 2
#> 2 1 3
#> 3 2 4
#> 4 2 4
#> 5 3 1
```

To fix this, we need to create haystack "containers" where the values within each container are all *totally* ordered. For haystack that would create 2 containers and look like:

```
haystack[1:4,]
#> # A tibble: 4 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
#> 2     1     3
#> 3     2     4
#> 4     2     4
```

```
haystack[5,]
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     3     1
```

This is essentially what `computing_nesting_container_ids()` does. You can actually see these ids with the helper, `compute_nesting_container_info()`:

```
haystack2 <- haystack
```

```
# we really pass along the integer ranks, but in this case that is equivalent
# to converting our double columns to integers
haystack2$x <- as.integer(haystack2$x)
haystack2$y <- as.integer(haystack2$y)
```

```
info <- compute_nesting_container_info(haystack2, condition = c("<=", "<="))
```

```
# the ids are in the second slot.
# container ids break haystack into [1, 4] and [5, 5].
info[[2]]
#> [1] 0 0 0 0 1
```

So the idea is that for each needle, we look in each haystack container and find all the matches, then we aggregate all of the matches once at the end. `df_locate_matches_with_containers()` has the job of iterating over the containers.

Computing totally ordered containers can be expensive, but luckily it doesn't happen very often in normal usage.

- If there are all == conditions, we don't need containers (i.e. any equi join)
- If there is only 1 non-equi condition and no conditions after it, we don't need containers (i.e. most rolling joins)
- Otherwise the typical case where we need containers is if we have something like `date >= lower`, `date <= upper`. Even so, the computation cost generally scales with the number of columns in haystack you compute containers with (here 2), and it only really slows down around 4 columns or so, which I haven't ever seen a real life example of.

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. `vctrs` 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 `vctrs` make use of the same mechanism. They are finalised with `vec_ptype_finalise()`.

list_drop_empty	<i>Drop empty elements from a list</i>
-----------------	--

Description

list_drop_empty() removes empty elements from a list. This includes NULL elements along with empty vectors, like integer(0). This is equivalent to, but faster than, vec_slice(x, list_sizes(x) != 0L).

Usage

```
list_drop_empty(x)
```

Arguments

x A list.

Dependencies

- [vec_slice\(\)](#)

Examples

```
x <- list(1, NULL, integer(), 2)
list_drop_empty(x)
```

list_of	<i>list_of S3 class for homogenous lists</i>
---------	--

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, ...)

is_list_of(x)

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

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

Arguments

<code>...</code>	Vectors to coerce.
<code>.ptype</code>	If NULL, 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_guessing")</code> is TRUE 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, 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 stop_incompatible_type()).

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))
```

missing

Missing values

Description

- `vec_detect_missing()` returns a logical vector the same size as `x`. For each element of `x`, it returns TRUE if the element is missing, and FALSE otherwise.
- `vec_any_missing()` returns a single TRUE or FALSE depending on whether or not `x` has *any* missing values.

Differences with `is.na()`:

Data frame rows are only considered missing if every element in the row is missing. Similarly, [record vector](#) elements are only considered missing if every field in the record is missing. Put another way, rows with *any* missing values are considered [incomplete](#), but only rows with *all* missing values are considered missing.

List elements are only considered missing if they are NULL.

Usage

```
vec_detect_missing(x)
```

```
vec_any_missing(x)
```

Arguments

x A vector

Value

- `vec_detect_missing()` returns a logical vector the same size as `x`.
- `vec_any_missing()` returns a single TRUE or FALSE.

Dependencies

- [vec_proxy_equal\(\)](#)

See Also

[vec_detect_complete\(\)](#)

Examples

```
x <- c(1, 2, NA, 4, NA)

vec_detect_missing(x)
vec_any_missing(x)

# Data frames are iterated over rowwise, and only report a row as missing
# if every element of that row is missing. If a row is only partially
# missing, it is said to be incomplete, but not missing.
y <- c("a", "b", NA, "d", "e")
df <- data_frame(x = x, y = y)

df$missing <- vec_detect_missing(df)
df$incomplete <- !vec_detect_complete(df)
df
```

name_spec

Name specifications

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}"`.
- An `rlang::zap()` object, in which case both outer and inner names are ignored and the result is unnamed.

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))
```

new_data_frame	<i>Assemble attributes for data frame construction</i>
----------------	--

Description

`new_data_frame()` constructs a new data frame from an existing list. It is meant to be performant, and does not check the inputs for correctness in any way. It is only safe to use after a call to `df_list()`, which collects and validates the columns used to construct the data frame.

Usage

```
new_data_frame(x = list(), n = NULL, ..., class = NULL)
```

Arguments

<code>x</code>	A named list of equal-length vectors. The lengths are not checked; it is responsibility of the caller to make sure they are equal.
<code>n</code>	Number of rows. If <code>NULL</code> , will be computed from the length of the first element of <code>x</code> .
<code>..., class</code>	Additional arguments for creating subclasses. The following attributes have special behavior: <ul style="list-style-type: none"> • <code>"names"</code> is preferred if provided, overriding existing names in <code>x</code>. • <code>"row.names"</code> is preferred if provided, overriding both <code>n</code> and the size implied by <code>x</code>.

See Also

`df_list()` for a way to safely construct a data frame's underlying data structure from individual columns. This can be used to create a named list for further use by `new_data_frame()`.

Examples

```
new_data_frame(list(x = 1:10, y = 10:1))
```

obj_is_list	<i>List checks</i>
-------------	--------------------

Description

- `obj_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"`.
- `list_all_vectors()` takes a list and returns `TRUE` if all elements of that list are vectors.
- `list_all_size()` takes a list and returns `TRUE` if all elements of that list have the same size.
- `obj_check_list()`, `list_check_all_vectors()`, and `list_check_all_size()` use the above functions, but throw a standardized and informative error if they return `FALSE`.

Usage

```
obj_is_list(x)

obj_check_list(x, ..., arg = caller_arg(x), call = caller_env())

list_all_vectors(x)

list_check_all_vectors(x, ..., arg = caller_arg(x), call = caller_env())

list_all_size(x, size)

list_check_all_size(x, size, ..., arg = caller_arg(x), call = caller_env())
```

Arguments

x	For <code>vec_*()</code> functions, an object. For <code>list_*()</code> functions, a list.
...	These dots are for future extensions and must be empty.
arg	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of <code>abort()</code> for more information.
size	The size to check each element for.

Details

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

See Also

[list_sizes\(\)](#)

Examples

```
obj_is_list(list())
obj_is_list(list_of(1))
obj_is_list(data.frame())

list_all_vectors(list(1, mtcars))
list_all_vectors(list(1, environment()))

list_all_size(list(1:2, 2:3), 2)
list_all_size(list(1:2, 2:4), 2)

# `list`-prefixed functions assume a list:
try(list_all_vectors(environment()))
```

reference-faq-compatibility

FAQ - Is my class compatible with vctrs?

Description

vctrs provides a framework for working with vector classes in a generic way. However, it implements several compatibility fallbacks to base R methods. In this reference you will find how vctrs tries to be compatible with your vector class, and what base methods you need to implement for compatibility.

If you're starting from scratch, we think you'll find it easier to start using `new_vctr()` as documented in `vignette("s3-vector")`. This guide is aimed for developers with existing vector classes.

Aggregate operations with fallbacks:

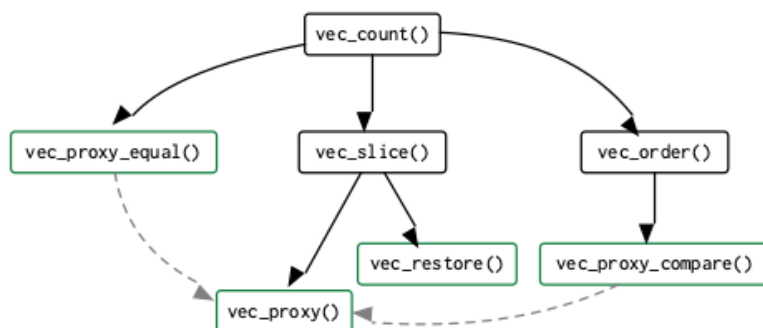
All vctrs operations are based on four primitive generics described in the next section. However there are many higher level operations. The most important ones implement fallbacks to base generics for maximum compatibility with existing classes.

- `vec_slice()` falls back to the base `[]` generic if no `vec_proxy()` method is implemented. This way foreign classes that do not implement `vec_restore()` can restore attributes based on the new subsetted contents.
- `vec_c()` and `vec_rbind()` now fall back to `base::c()` if the inputs have a common parent class with a `c()` method (only if they have no self-to-self `vec_ptype2()` method). vctrs works hard to make your `c()` method success in various situations (with `NULL` and `NA` inputs, even as first input which would normally prevent dispatch to your method). The main downside compared to using vctrs primitives is that you can't combine vectors of different classes since there is no extensible mechanism of coercion in `c()`, and it is less efficient in some cases.

The vctrs primitives:

Most functions in vctrs are aggregate operations: they call other vctrs functions which themselves call other vctrs functions. The dependencies of a vctrs functions are listed in the Dependencies section of its documentation page. Take a look at `vec_count()` for an example.

These dependencies form a tree whose leaves are the four vctrs primitives. Here is the diagram for `vec_count()`:



The coercion generics:

The coercion mechanism in vctrs is based on two generics:

- [vec_ptype2\(\)](#)
- [vec_cast\(\)](#)

See the [theory overview](#).

Two objects with the same class and the same attributes are always considered compatible by `ptype2` and `cast`. If the attributes or classes differ, they throw an incompatible type error.

Coercion errors are the main source of incompatibility with vctrs. See the [howto guide](#) if you need to implement methods for these generics.

The proxy and restoration generics:

- [vec_proxy\(\)](#)
- [vec_restore\(\)](#)

These generics are essential for vctrs but mostly optional. `vec_proxy()` defaults to an [identity](#) function and you normally don't need to implement it. The proxy a vector must be one of the atomic vector types, a list, or a data frame. By default, S3 lists that do not inherit from "list" do not have an identity proxy. In that case, you need to explicitly implement `vec_proxy()` or make your class inherit from list.

runs

Runs

Description

- `vec_identify_runs()` returns a vector of identifiers for the elements of `x` that indicate which run of repeated values they fall in. The number of runs is also returned as an attribute, `n`.
- `vec_run_sizes()` returns an integer vector corresponding to the size of each run. This is identical to the `times` column from `vec_unrep()`, but is faster if you don't need the run keys.
- `vec_unrep()` is a generalized [base::rle\(\)](#). It is documented alongside the "repeat" functions of `vec_rep()` and `vec_rep_each()`; look there for more information.

Usage

```
vec_identify_runs(x)
```

```
vec_run_sizes(x)
```

Arguments

`x` A vector.

Details

Unlike [base::rle\(\)](#), adjacent missing values are considered identical when constructing runs. For example, `vec_identify_runs(c(NA, NA))` will return `c(1, 1)`, not `c(1, 2)`.

Value

- For `vec_identify_runs()`, an integer vector with the same size as `x`. A scalar integer attribute, `n`, is attached.
- For `vec_run_sizes()`, an integer vector with size equal to the number of runs in `x`.

See Also

[vec_unrep\(\)](#) for a generalized `base::rle()`.

Examples

```
x <- c("a", "z", "z", "c", "a", "a")

vec_identify_runs(x)
vec_run_sizes(x)
vec_unrep(x)

y <- c(1, 1, 1, 2, 2, 3)

# With multiple columns, the runs are constructed rowwise
df <- data_frame(
  x = x,
  y = y
)

vec_identify_runs(df)
vec_run_sizes(df)
vec_unrep(df)
```

theory-faq-coercion *FAQ - How does coercion work in vctrs?*

Description

This is an overview of the usage of `vec_ptype2()` and `vec_cast()` and their role in the `vctrs` coercion mechanism. Related topics:

- For an example of implementing coercion methods for simple vectors, see [?howto-faq-coercion](#).
- For an example of implementing coercion methods for data frame subclasses, see [?howto-faq-coercion-data-frame](#).
- For a tutorial about implementing `vctrs` classes from scratch, see `vignette("s3-vector")`.

Combination mechanism in vctrs:

The coercion system in `vctrs` is designed to make combination of multiple inputs consistent and extensible. Combinations occur in many places, such as row-binding, joins, subset-assignment, or grouped summary functions that use the split-apply-combine strategy. For example:

```

vec_c(TRUE, 1)
#> [1] 1 1

vec_c("a", 1)
#> Error in `vec_c()` :
#> ! Can't combine `..1` <character> and `..2` <double>.

vec_rbind(
  data.frame(x = TRUE),
  data.frame(x = 1, y = 2)
)
#>   x y
#> 1 1 NA
#> 2 1  2

vec_rbind(
  data.frame(x = "a"),
  data.frame(x = 1, y = 2)
)
#> Error in `vec_rbind()` :
#> ! Can't combine `..1$x` <character> and `..2$x` <double>.

```

One major goal of `vctrs` is to provide a central place for implementing the coercion methods that make generic combinations possible. The two relevant generics are `vec_ptype2()` and `vec_cast()`. They both take two arguments and perform **double dispatch**, meaning that a method is selected based on the classes of both inputs.

The general mechanism for combining multiple inputs is:

1. Find the common type of a set of inputs by reducing (as in `base::Reduce()` or `purrr::reduce()`) the `vec_ptype2()` binary function over the set.
2. Convert all inputs to the common type with `vec_cast()`.
3. Initialise the output vector as an instance of this common type with `vec_init()`.
4. Fill the output vector with the elements of the inputs using `vec_assign()`.

The last two steps may require `vec_proxy()` and `vec_restore()` implementations, unless the attributes of your class are constant and do not depend on the contents of the vector. We focus here on the first two steps, which require `vec_ptype2()` and `vec_cast()` implementations.

`vec_ptype2()`:

Methods for `vec_ptype2()` are passed two *prototypes*, i.e. two inputs emptied of their elements. They implement two behaviours:

- If the types of their inputs are compatible, indicate which of them is the richer type by returning it. If the types are of equal resolution, return any of the two.
- Throw an error with `stop_incompatible_type()` when it can be determined from the attributes that the types of the inputs are not compatible.

Type compatibility:

A type is **compatible** with another type if the values it represents are a subset or a superset of the values of the other type. The notion of “value” is to be interpreted at a high level, in particular it is not the same as the memory representation. For example, factors are represented in memory with integers but their values are more related to character vectors than to round numbers:

```
# Two factors are compatible
vec_ptype2(factor("a"), factor("b"))
#> factor()
#> Levels: a b

# Factors are compatible with a character
vec_ptype2(factor("a"), "b")
#> character(0)

# But they are incompatible with integers
vec_ptype2(factor("a"), 1L)
#> Error:
#> ! Can't combine `factor("a")` <factor<4d52a>> and `1L` <integer>.
```

Richness of type:

Richness of type is not a very precise notion. It can be about richer data (for instance a double vector covers more values than an integer vector), richer behaviour (a `data.table` has richer behaviour than a `data.frame`), or both. If you have trouble determining which one of the two types is richer, it probably means they shouldn't be automatically coercible.

Let's look again at what happens when we combine a factor and a character:

```
vec_ptype2(factor("a"), "b")
#> character(0)
```

The `ptype2` method for `<character>` and `<factor<"a">>` returns `<character>` because the former is a richer type. The factor can only contain "a" strings, whereas the character can contain any strings. In this sense, factors are a *subset* of character.

Note that another valid behaviour would be to throw an incompatible type error. This is what a strict factor implementation would do. We have decided to be laxer in `vctrs` because it is easy to inadvertently create factors instead of character vectors, especially with older versions of R where `stringsAsFactors` is still true by default.

Consistency and symmetry on permutation:

Each `ptype2` method should strive to have exactly the same behaviour when the inputs are permuted. This is not always possible, for example factor levels are aggregated in order:

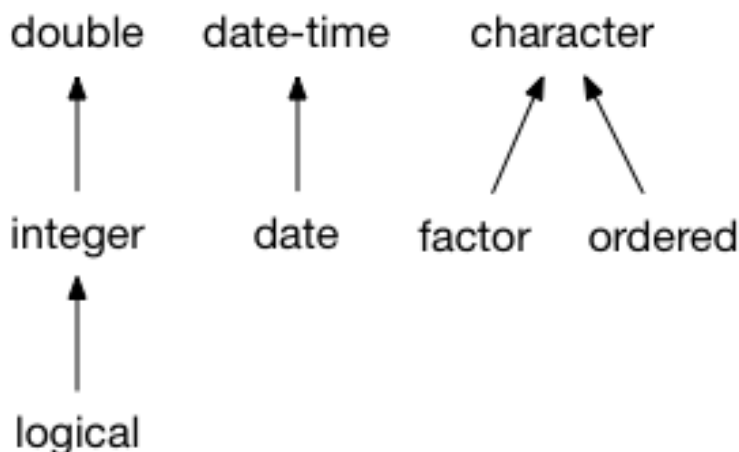
```
vec_ptype2(factor(c("a", "c")), factor("b"))
#> factor()
#> Levels: a c b
```

```
vec_ptype2(factor("b"), factor(c("a", "c")))
#> factor()
#> Levels: b a c
```

In any case, permuting the input should not return a fundamentally different type or introduce an incompatible type error.

Coercion hierarchy:

The classes that you can coerce together form a coercion (or subtyping) hierarchy. Below is a schema of the hierarchy for the base types like integer and factor. In this diagram the directions of the arrows express which type is richer. They flow from the bottom (more constrained types) to the top (richer types).



A coercion hierarchy is distinct from the structural hierarchy implied by memory types and classes. For instance, in a structural hierarchy, factors are built on top of integers. But in the coercion hierarchy they are more related to character vectors. Similarly, subclasses are not necessarily coercible with their superclasses because the coercion and structural hierarchies are separate.

Implementing a coercion hierarchy:

As a class implementor, you have two options. The simplest is to create an entirely separate hierarchy. The date and date-time classes are an example of an S3-based hierarchy that is completely separate. Alternatively, you can integrate your class in an existing hierarchy, typically by adding parent nodes on top of the hierarchy (your class is richer), by adding children node at the root of the hierarchy (your class is more constrained), or by inserting a node in the tree.

These coercion hierarchies are *implicit*, in the sense that they are implied by the `vec_ptype2()` implementations. There is no structured way to create or modify a hierarchy, instead you need to implement the appropriate coercion methods for all the types in your hierarchy, and diligently return the richer type in each case. The `vec_ptype2()` implementations are not transitive nor inherited, so all pairwise methods between classes lying on a given path must be implemented manually. This is something we might make easier in the future.

`vec_cast()`:

The second generic, `vec_cast()`, is the one that looks at the data and actually performs the conversion. Because it has access to more information than `vec_ptype2()`, it may be stricter and cause an error in more cases. `vec_cast()` has three possible behaviours:

- Determine that the prototypes of the two inputs are not compatible. This must be decided in exactly the same way as for `vec_ptype2()`. Call `stop_incompatible_cast()` if you can determine from the attributes that the types are not compatible.
- Detect incompatible values. Usually this is because the target type is too restricted for the values supported by the input type. For example, a fractional number can't be converted to an integer. The method should throw an error in that case.
- Return the input vector converted to the target type if all values are compatible. Whereas `vec_ptype2()` must return the same type when the inputs are permuted, `vec_cast()` is *directional*. It always returns the type of the right-hand side, or dies trying.

Double dispatch:

The dispatch mechanism for `vec_ptype2()` and `vec_cast()` looks like S3 but is actually a custom mechanism. Compared to S3, it has the following differences:

- It dispatches on the classes of the first two inputs.
- There is no inheritance of `ptype2` and `cast` methods. This is because the S3 class hierarchy is not necessarily the same as the coercion hierarchy.
- `NextMethod()` does not work. Parent methods must be called explicitly if necessary.
- The default method is hard-coded.

Data frames:

The determination of the common type of data frames with `vec_ptype2()` happens in three steps:

1. Match the columns of the two input data frames. If some columns don't exist, they are created and filled with adequately typed NA values.
2. Find the common type for each column by calling `vec_ptype2()` on each pair of matched columns.
3. Find the common data frame type. For example the common type of a grouped tibble and a tibble is a grouped tibble because the latter is the richer type. The common type of a data table and a data frame is a data table.

`vec_cast()` operates similarly. If a data frame is cast to a target type that has fewer columns, this is an error.

If you are implementing coercion methods for data frames, you will need to explicitly call the parent methods that perform the common type determination or the type conversion described above. These are exported as `df_ptype2()` and `df_cast()`.

Data frame fallbacks:

Being too strict with data frame combinations would cause too much pain because there are many data frame subclasses in the wild that don't implement `vctrs` methods. We have decided to implement a special fallback behaviour for foreign data frames. Incompatible data frames fall back to a base data frame:

```
df1 <- data.frame(x = 1)
df2 <- structure(df1, class = c("foreign_df", "data.frame"))
```

```
vec_rbind(df1, df2)
```

```
#>   x
#> 1 1
#> 2 1
```

When a tibble is involved, we fall back to tibble:

```
df3 <- tibble::as_tibble(df1)
```

```
vec_rbind(df1, df3)
```

```
#> # A tibble: 2 x 1
#>   x
#>   <dbl>
#> 1     1
#> 2     1
```

These fallbacks are not ideal but they make sense because all data frames share a common data structure. This is not generally the case for vectors. For example factors and characters have different representations, and it is not possible to find a fallback time mechanically.

However this fallback has a big downside: implementing `vctrs` methods for your data frame subclass is a breaking behaviour change. The proper coercion behaviour for your data frame class should be specified as soon as possible to limit the consequences of changing the behaviour of your class in R scripts.

theory-faq-recycling *FAQ - How does recycling work in `vctrs` and the tidyverse?*

Description

Recycling describes the concept of repeating elements of one vector to match the size of another. There are two rules that underlie the “tidyverse” recycling rules:

- Vectors of size 1 will be recycled to the size of any other vector
- Otherwise, all vectors must have the same size

Examples

Vectors of size 1 are recycled to the size of any other vector:

```
tibble(x = 1:3, y = 1L)
#> # A tibble: 3 x 2
#>   x     y
#>   <int> <int>
#> 1     1     1
#> 2     2     1
#> 3     3     1
```

This includes vectors of size 0:

```
tibble(x = integer(), y = 1L)
#> # A tibble: 0 x 2
#> # i 2 variables: x <int>, y <int>
```

If vectors aren't size 1, they must all be the same size. Otherwise, an error is thrown:

```
tibble(x = 1:3, y = 4:7)
#> Error in `tibble()`:
#> ! Tibble columns must have compatible sizes.
#> * Size 3: Existing data.
#> * Size 4: Column `y`.
#> i Only values of size one are recycled.
```

vctrs backend

Packages in r-lib and the tidyverse generally use `vec_size_common()` and `vec_recycle_common()` as the backends for handling recycling rules.

- `vec_size_common()` returns the common size of multiple vectors, after applying the recycling rules
- `vec_recycle_common()` goes one step further, and actually recycles the vectors to their common size

```
vec_size_common(1:3, "x")
#> [1] 3
```

```
vec_recycle_common(1:3, "x")
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "x" "x" "x"
```

```
vec_size_common(1:3, c("x", "y"))
#> Error:
#> ! Can't recycle `..1` (size 3) to match `..2` (size 2).
```

Base R recycling rules

The recycling rules described here are stricter than the ones generally used by base R, which are:

- If any vector is length 0, the output will be length 0
- Otherwise, the output will be length $\max(\text{length}_x, \text{length}_y)$, and a warning will be thrown if the length of the longer vector is not an integer multiple of the length of the shorter vector.

We explore the base R rules in detail in `vignette("type-size")`.

 vec-rep

Repeat a vector

Description

- `vec_rep()` repeats an entire vector a set number of times.
- `vec_rep_each()` repeats each element of a vector a set number of times.
- `vec_unrep()` compresses a vector with repeated values. The repeated values are returned as a key alongside the number of times each key is repeated.

Usage

```
vec_rep(
  x,
  times,
  ...,
  error_call = current_env(),
  x_arg = "x",
  times_arg = "times"
)
```

```
vec_rep_each(
  x,
  times,
  ...,
  error_call = current_env(),
  x_arg = "x",
  times_arg = "times"
)
```

```
vec_unrep(x)
```

Arguments

<code>x</code>	A vector.
<code>times</code>	For <code>vec_rep()</code> , a single integer for the number of times to repeat the entire vector. For <code>vec_rep_each()</code> , an integer vector of the number of times to repeat each element of <code>x</code> . <code>times</code> will be recycled to the size of <code>x</code> .
<code>...</code>	These dots are for future extensions and must be empty.
<code>error_call</code>	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of abort() for more information.
<code>x_arg, times_arg</code>	Argument names for errors.

Details

Using `vec_unrep()` and `vec_rep_each()` together is similar to using [base::rle\(\)](#) and [base::inverse.rle\(\)](#). The following invariant shows the relationship between the two functions:

```
compressed <- vec_unrep(x)
identical(x, vec_rep_each(compressed$key, compressed$times))
```

There are two main differences between `vec_unrep()` and [base::rle\(\)](#):

- `vec_unrep()` treats adjacent missing values as equivalent, while `rle()` treats them as different values.
- `vec_unrep()` works along the size of `x`, while `rle()` works along its length. This means that `vec_unrep()` works on data frames by compressing repeated rows.

Value

For `vec_rep()`, a vector the same type as `x` with size `vec_size(x) * times`.

For `vec_rep_each()`, a vector the same type as `x` with size `sum(vec_recycle(times, vec_size(x)))`.

For `vec_unrep()`, a data frame with two columns, `key` and `times`. `key` is a vector with the same type as `x`, and `times` is an integer vector.

Dependencies

- `vec_slice()`

Examples

```
# Repeat the entire vector
vec_rep(1:2, 3)

# Repeat within each vector
vec_rep_each(1:2, 3)
x <- vec_rep_each(1:2, c(3, 4))
x

# After using `vec_rep_each()`, you can recover the original vector
# with `vec_unrep()`
vec_unrep(x)

df <- data.frame(x = 1:2, y = 3:4)

# `rep()` repeats columns of data frames, and returns lists
rep(df, each = 2)

# `vec_rep()` and `vec_rep_each()` repeat rows, and return data frames
vec_rep(df, 2)
vec_rep_each(df, 2)

# `rle()` treats adjacent missing values as different
y <- c(1, NA, NA, 2)
rle(y)

# `vec_unrep()` treats them as equivalent
vec_unrep(y)
```

vec-set

Set operations

Description

- `vec_set_intersect()` returns all values in both `x` and `y`.
- `vec_set_difference()` returns all values in `x` but not `y`. Note that this is an asymmetric set difference, meaning it is not commutative.

- `vec_set_union()` returns all values in either `x` or `y`.
- `vec_set_symmetric_difference()` returns all values in either `x` or `y` but not both. This is a commutative difference.

Because these are *set* operations, these functions only return unique values from `x` and `y`, returned in the order they first appeared in the original input. Names of `x` and `y` are retained on the result, but names are always taken from `x` if the value appears in both inputs.

These functions work similarly to `intersect()`, `setdiff()`, and `union()`, but don't strip attributes and can be used with data frames.

Usage

```
vec_set_intersect(  
  x,  
  y,  
  ...,  
  ptype = NULL,  
  x_arg = "x",  
  y_arg = "y",  
  error_call = current_env()  
)
```

```
vec_set_difference(  
  x,  
  y,  
  ...,  
  ptype = NULL,  
  x_arg = "x",  
  y_arg = "y",  
  error_call = current_env()  
)
```

```
vec_set_union(  
  x,  
  y,  
  ...,  
  ptype = NULL,  
  x_arg = "x",  
  y_arg = "y",  
  error_call = current_env()  
)
```

```
vec_set_symmetric_difference(  
  x,  
  y,  
  ...,  
  ptype = NULL,  
  x_arg = "x",  
  y_arg = "y",
```

```

    error_call = current_env()
  )

```

Arguments

<code>x, y</code>	A pair of vectors.
<code>...</code>	These dots are for future extensions and must be empty.
<code>ptype</code>	If NULL, the default, the output type is determined by computing the common type between <code>x</code> and <code>y</code> . If supplied, both <code>x</code> and <code>y</code> will be cast to this type.
<code>x_arg, y_arg</code>	Argument names for <code>x</code> and <code>y</code> . These are used in error messages.
<code>error_call</code>	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of abort() for more information.

Details

Missing values are treated as equal to other missing values. For doubles and complexes, NaN are equal to other NaN, but not to NA.

Value

A vector of the common type of `x` and `y` (or `ptype`, if supplied) containing the result of the corresponding set function.

Dependencies

`vec_set_intersect()`:

- [vec_proxy_equal\(\)](#)
- [vec_slice\(\)](#)
- [vec_ptype2\(\)](#)
- [vec_cast\(\)](#)

`vec_set_difference()`:

- [vec_proxy_equal\(\)](#)
- [vec_slice\(\)](#)
- [vec_ptype2\(\)](#)
- [vec_cast\(\)](#)

`vec_set_union()`:

- [vec_proxy_equal\(\)](#)
- [vec_slice\(\)](#)
- [vec_ptype2\(\)](#)
- [vec_cast\(\)](#)
- [vec_c\(\)](#)

`vec_set_symmetric_difference()`:

- `vec_proxy_equal()`
- `vec_slice()`
- `vec_ptype2()`
- `vec_cast()`
- `vec_c()`

Examples

```
x <- c(1, 2, 1, 4, 3)
y <- c(2, 5, 5, 1)

# All unique values in both `x` and `y`.
# Duplicates in `x` and `y` are always removed.
vec_set_intersect(x, y)

# All unique values in `x` but not `y`
vec_set_difference(x, y)

# All unique values in either `x` or `y`
vec_set_union(x, y)

# All unique values in either `x` or `y` but not both
vec_set_symmetric_difference(x, y)

# These functions can also be used with data frames
x <- data_frame(
  a = c(2, 3, 2, 2),
  b = c("j", "k", "j", "l")
)
y <- data_frame(
  a = c(1, 2, 2, 2, 3),
  b = c("j", "l", "j", "l", "j")
)

vec_set_intersect(x, y)
vec_set_difference(x, y)
vec_set_union(x, y)
vec_set_symmetric_difference(x, y)

# Vector names don't affect set membership, but if you'd like to force
# them to, you can transform the vector into a two column data frame
x <- c(a = 1, b = 2, c = 2, d = 3)
y <- c(c = 2, b = 1, a = 3, d = 3)

vec_set_intersect(x, y)

x <- data_frame(name = names(x), value = unname(x))
y <- data_frame(name = names(y), value = unname(y))

vec_set_intersect(x, y)
```

vector-checks	<i>Vector checks</i>
---------------	----------------------

Description

- `obj_is_vector()` tests if `x` is considered a vector in the `vctrs` sense. See *Vectors and scalars* below for the exact details.
- `obj_check_vector()` uses `obj_is_vector()` and throws a standardized and informative error if it returns `FALSE`.
- `vec_check_size()` tests if `x` has size `size`, and throws an informative error if it doesn't.

Usage

```
obj_is_vector(x)
```

```
obj_check_vector(x, ..., arg = caller_arg(x), call = caller_env())
```

```
vec_check_size(x, size, ..., arg = caller_arg(x), call = caller_env())
```

Arguments

<code>x</code>	For <code>obj_*()</code> functions, an object. For <code>vec_*()</code> functions, a vector.
<code>...</code>	These dots are for future extensions and must be empty.
<code>arg</code>	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
<code>call</code>	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of <code>abort()</code> for more information.
<code>size</code>	The size to check for.

Value

- `obj_is_vector()` returns a single `TRUE` or `FALSE`.
- `obj_check_vector()` returns `NULL` invisibly, or errors.
- `vec_check_size()` returns `NULL` invisibly, or errors.

Vectors and scalars

Informally, a vector is a collection that makes sense to use as column in a data frame. The following rules define whether or not `x` is considered a vector.

If no `vec_proxy()` method has been registered, `x` is a vector if:

- The `base type` of the object is atomic: `"logical"`, `"integer"`, `"double"`, `"complex"`, `"character"`, or `"raw"`.
- `x` is a list, as defined by `obj_is_list()`.

- x is a `data.frame`.

If a `vec_proxy()` method has been registered, x is a vector if:

- The proxy satisfies one of the above conditions.
- The base type of the proxy is "list", regardless of its class. S3 lists are thus treated as scalars unless they implement a `vec_proxy()` method.

Otherwise an object is treated as scalar and cannot be used as a vector. In particular:

- NULL is not a vector.
- S3 lists like `lm` objects are treated as scalars by default.
- Objects of type `expression` are not treated as vectors.

Technical limitations

- Support for S4 vectors is currently limited to objects that inherit from an atomic type.
- Subclasses of `data.frame` that *append* their class to the back of the "class" attribute are not treated as vectors. If you inherit from an S3 class, always prepend your class to the front of the "class" attribute for correct dispatch. This matches our general principle of allowing subclasses but not mixins.

Examples

```
obj_is_vector(1)

# Data frames are vectors
obj_is_vector(data_frame())

# Bare lists are vectors
obj_is_vector(list())

# S3 lists are vectors if they explicitly inherit from `list`
x <- structure(list(), class = c("my_list", "list"))
obj_is_list(x)
obj_is_vector(x)

# But if they don't explicitly inherit from `list`, they aren't
# automatically considered to be vectors. Instead, vctrs considers this
# to be a scalar object, like a linear model returned from `lm`.
y <- structure(list(), class = "my_list")
obj_is_list(y)
obj_is_vector(y)

# `obj_check_vector()` throws an informative error if the input
# isn't a vector
try(obj_check_vector(y))

# `vec_check_size()` throws an informative error if the size of the
# input doesn't match `size`
vec_check_size(1:5, size = 5)
try(vec_check_size(1:5, size = 4))
```

vec_as_names

*Retrieve and repair names***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", "unique_quiet",
            "universal_quiet"),
  repair_arg = NULL,
  quiet = FALSE,
  call = caller_env()
)
```

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", "universal", "unique_quiet", or "universal_quiet". 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"> • Minimal names are never NULL or NA. When an element doesn't have a name, its minimal name is an empty string.

- 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.

The options "unique_quiet" and "universal_quiet" are here to help the user who calls this function indirectly, via another function which exposes repair but not quiet. Specifying repair = "unique_quiet" is like specifying repair = "unique", quiet = TRUE. When the "*_quiet" options are used, any setting of quiet is silently overridden.

repair_arg	If specified and repair = "check_unique", any errors will include a hint to set the repair_arg.
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. Users can silence the name repair messages by setting the "rlib_name_repair_verbosity" global option to "quiet".
call	The execution environment of a currently running function, e.g. caller_env(). The function will be mentioned in error messages as the source of the error. See the call argument of <code>abort()</code> for more information.

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.

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 = rlang::zap(),
  .name_repair = c("unique", "universal", "check_unique", "unique_quiet",
    "universal_quiet"),
  .name_spec = NULL,
  .error_call = current_env()
)

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

Arguments

...	Data frames or vectors. When the inputs are named: <ul style="list-style-type: none"> • <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>. • <code>vec_cbind()</code> creates packed data frame columns with named inputs. 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> .
<code>.ptype</code>	If NULL, the default, the output type is determined by computing the common type across all elements of ... Alternatively, you can supply <code>.ptype</code> to give the output known type. If <code>getOption("vctrs.no_guessing")</code> is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.
<code>.names_to</code>	This controls what to do with input names supplied in ...

- By default, input names are [zapped](#).
 - If a string, specifies a column where the input names will be copied. These names are often useful to identify rows with their original input. If a column name is supplied and `...` is not named, an integer column is used instead.
 - If NULL, the input names are used as row names.
- `.name_repair` One of "unique", "universal", "check_unique", "unique_quiet", or "universal_quiet". See [vec_as_names\(\)](#) for the meaning of these options.
- With `vec_rbind()`, the repair function is applied to all inputs separately. This is because `vec_rbind()` needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, `vec_cbind()` applies the repair function after all inputs have been concatenated together in a final data frame. Hence `vec_cbind()` allows the more permissive minimal names repair.
- `.name_spec` A name specification (as documented in [vec_c\(\)](#)) for combining the outer inputs names in `...` and the inner row names of the inputs. This only has an effect when `.names_to` is set to NULL, which causes the input names to be assigned as row names.
- `.error_call` The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of [abort\(\)](#) for more information.
- `.size` If, NULL, the default, will determine the number of rows in `vec_cbind()` output by using the tidyverse [recycling rules](#).
Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.

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))`

Dependencies

vctrs dependencies:

- `vec_cast_common()`
- `vec_proxy()`
- `vec_init()`
- `vec_assign()`
- `vec_restore()`

base dependencies of `vec_rbind()`:

- `base::c()`

If columns to combine inherit from a common class, `vec_rbind()` falls back to `base::c()` if there exists a `c()` method implemented for this class hierarchy.

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)

```

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", "unique_quiet",
    "universal_quiet"),
  .error_arg = "",
  .error_call = current_env()
)
```

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_guessing")</code> is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.</p>
.name_spec	<p>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:</p> <ul style="list-style-type: none"> • 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 "<code>{outer}_{inner}</code>". • An <code>rlang::zap()</code> object, in which case both outer and inner names are ignored and the result is unnamed. <p>See the name specification topic.</p>
.name_repair	How to repair names, see repair options in <code>vec_as_names()</code> .
.error_arg	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
.error_call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the call argument of <code>abort()</code> for more information.

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)`.

Dependencies**vctrs dependencies:**

- [vec_cast_common\(\)](#) with fallback
- [vec_proxy\(\)](#)
- [vec_restore\(\)](#)

base dependencies:

- [base::c\(\)](#)

If inputs inherit from a common class hierarchy, `vec_c()` falls back to `base::c()` if there exists a `c()` method implemented for this class hierarchy.

See Also

[vec_cbind\(\)/vec_rbind\(\)](#) for combining data frames by rows or columns.

Examples

```
vec_c(FALSE, 1L, 1.5)

# 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_cast	<i>Cast a vector to a specified type</i>
----------	--

Description

`vec_cast()` provides directional conversions from one type of vector to another. Along with [vec_ptype2\(\)](#), this generic forms the foundation of type coercions in vctrs.

Usage

```
vec_cast(x, to, ..., x_arg = caller_arg(x), to_arg = "", call = caller_env())

vec_cast_common(..., .to = NULL, .arg = "", .call = caller_env())

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

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

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

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

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

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

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

Arguments

<code>x</code>	Vectors to cast.
<code>to, .to</code>	Type to cast to. If <code>NULL</code> , <code>x</code> will be returned as is.
<code>...</code>	For <code>vec_cast_common()</code> , vectors to cast. For <code>vec_cast()</code> , <code>vec_cast_default()</code> , and <code>vec_restore()</code> , these dots are only for future extensions and should be empty.

x_arg	Argument name for x, used in error messages to inform the user about the locations of incompatible types (see stop_incompatible_type()).
to_arg	Argument name to used in error messages to inform the user about the locations of incompatible types (see stop_incompatible_type()).
call, .call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of abort() for more information.
.arg	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.

Value

A vector the same length as `x` with the same type as `to`, or an error if the cast is not possible. An error is generated if information is lost when casting between compatible types (i.e. when there is no 1-to-1 mapping for a specific value).

Implementing coercion methods

- For an overview of how these generics work and their roles in `vctrs`, see [?theory-faq-coercion](#).
- For an example of implementing coercion methods for simple vectors, see [?howto-faq-coercion](#).
- For an example of implementing coercion methods for data frame subclasses, see [?howto-faq-coercion-data-frame](#).
- For a tutorial about implementing `vctrs` classes from scratch, see `vignette("s3-vector")`.

Dependencies of `vec_cast_common()`

vctrs dependencies:

- [vec_ptype2\(\)](#)
- [vec_cast\(\)](#)

base dependencies:

Some functions enable a base-class fallback for `vec_cast_common()`. In that case the inputs are deemed compatible when they have the same [base type](#) and inherit from the same base class.

See Also

Call [stop_incompatible_cast\(\)](#) when you determine from the attributes that an input can't be cast to the target type.

Examples

```
# x is a double, but no information is lost
vec_cast(1, integer())

# When information is lost the cast fails
try(vec_cast(c(1, 1.5), integer()))
try(vec_cast(c(1, 2), logical()))

# You can suppress this error and get the partial results
```

```

allow_lossy_cast(vec_cast(c(1, 1.5), integer()))
allow_lossy_cast(vec_cast(c(1, 2), logical()))

# By default this suppress all lossy cast errors without
# distinction, but you can be specific about what cast is allowed
# by supplying prototypes
allow_lossy_cast(vec_cast(c(1, 1.5), integer()), to_ptype = integer())
try(allow_lossy_cast(vec_cast(c(1, 2), logical()), to_ptype = integer()))

# No sensible coercion is possible so an error is generated
try(vec_cast(1.5, factor("a")))

# Cast to common type
vec_cast_common(factor("a"), factor(c("a", "b")))

```

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()`.
- `list_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()`, but typically a little faster.

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

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

Usage

```
vec_chop(x, ..., indices = NULL, sizes = NULL)
```

```

list_unchop(
  x,
  ...,
  indices = NULL,
  ptype = NULL,
  name_spec = NULL,
  name_repair = c("minimal", "unique", "check_unique", "universal", "unique_quiet",
    "universal_quiet"),
  error_arg = "x",
  error_call = current_env()
)

```

Arguments

x	A vector
...	These dots are for future extensions and must be empty.
indices	For <code>vec_chop()</code> , a list of positive integer vectors to slice <code>x</code> with, or <code>NULL</code> . Can't be used if <code>sizes</code> is already specified. If both <code>indices</code> and <code>sizes</code> are <code>NULL</code> , <code>x</code> is split into its individual elements, equivalent to using an <code>indices</code> of <code>as.list(vec_seq_along(x))</code> . For <code>list_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 <code>indices</code> 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> .
sizes	An integer vector of non-negative sizes representing sequential indices to slice <code>x</code> with, or <code>NULL</code> . Can't be used if <code>indices</code> is already specified. For example, <code>sizes = c(2, 4)</code> is equivalent to <code>indices = list(1:2, 3:6)</code> , but is typically faster. <code>sum(sizes)</code> must be equal to <code>vec_size(x)</code> , i.e. <code>sizes</code> must completely partition <code>x</code> , but an individual size is allowed to be \emptyset .
ptype	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.
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 <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: <ul style="list-style-type: none"> • 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 "<code>{outer}_{inner}</code>". • An <code>rlang::zap()</code> object, in which case both outer and inner names are ignored and the result is unnamed. See the name specification topic .
name_repair	How to repair names, see repair options in <code>vec_as_names()</code> .
error_arg	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
error_call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of <code>abort()</code> for more information.

Value

- `vec_chop()`: A list where each element has the same type as `x`. The size of the list is equal to `vec_size(indices)`, `vec_size(sizes)`, or `vec_size(x)` depending on whether or not `indices` or `sizes` is provided.

- `list_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)`.

Dependencies of `vec_chop()`

- `vec_slice()`

Dependencies of `list_unchop()`

- `vec_c()`

Examples

```
vec_chop(1:5)

# These two are equivalent
vec_chop(1:5, indices = list(1:2, 3:5))
vec_chop(1:5, sizes = c(2, 3))

# Can also be used on data frames
vec_chop(mtcars, indices = list(1:3, 4:6))

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

# When unchopping, size 1 elements of `x` are recycled
# to the size of the corresponding index
list_unchop(list(1, 2:3), indices = 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)
list_unchop(lst, indices = list(c(3, 2), c(1, 4)), name_spec = "{outer}_{inner}")

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

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

ave2(breaks, wool, mean)
```

```

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

# If you know your input is sorted and you'd like to split on the groups,
# `vec_run_sizes()` can be efficiently combined with `sizes`
df <- data_frame(
  g = c(2, 5, 5, 6, 6, 6, 6, 8, 9, 9),
  x = 1:10
)
vec_chop(df, sizes = vec_run_sizes(df$g))

# If you have a list of homogeneous vectors, sometimes it can be useful to
# unchop, apply a function to the flattened vector, and then rechop according
# to the original indices. This can be done efficiently with `list_sizes()`.
x <- list(c(1, 2, 1), c(3, 1), 5, double())
x_flat <- list_unchop(x)
x_flat <- x_flat + max(x_flat)
vec_chop(x_flat, sizes = list_sizes(x))

```

vec_compare

Compare two vectors

Description

Compare two vectors

Usage

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

Arguments

<code>x, y</code>	Vectors with compatible types and lengths.
<code>na_equal</code>	Should NA values be considered equal?
<code>.ptype</code>	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 create a proxy that is used in the comparison.

Dependencies

- `vec_cast_common()` with fallback
- `vec_recycle_common()`
- `vec_proxy_compare()`

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"> • "count", the default, puts most frequent values at top • "key", orders by the output key column (i.e. unique values of <code>x</code>) • "location", orders by location where key first seen. This is useful if you want to match the counts up to other unique/duplicated functions. • "none", leaves unordered. This is not guaranteed to produce the same ordering across R sessions, but is the fastest method. |

Value

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

Dependencies

- [vec_proxy_equal\(\)](#)
- [vec_slice\(\)](#)
- [vec_order\(\)](#)

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_detect_complete  Complete
```

Description

`vec_detect_complete()` detects "complete" observations. An observation is considered complete if it is non-missing. For most vectors, this implies that `vec_detect_complete(x) == !vec_detect_missing(x)`. For data frames and matrices, a row is only considered complete if all elements of that row are non-missing. To compare, `!vec_detect_missing(x)` detects rows that are partially complete (they have at least one non-missing value).

Usage

```
vec_detect_complete(x)
```

Arguments

```
x          A vector
```

Details

A [record](#) type vector is similar to a data frame, and is only considered complete if all fields are non-missing.

Value

A logical vector with the same size as x.

See Also

[stats::complete.cases\(\)](#)

Examples

```
x <- c(1, 2, NA, 4, NA)

# For most vectors, this is identical to `!vec_detect_missing(x)`
vec_detect_complete(x)
!vec_detect_missing(x)

df <- data_frame(
  x = x,
  y = c("a", "b", NA, "d", "e")
)

# This returns `TRUE` where all elements of the row are non-missing.
# Compare that with `!vec_detect_missing()`, which detects rows that have at
# least one non-missing value.
df2 <- df
df2$all_non_missing <- vec_detect_complete(df)
df2$any_non_missing <- !vec_detect_missing(df)
df2
```

vec_duplicate

Find duplicated values

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 NAs to be equal. (Similarly, all NaN are also considered to be equal.)

Dependencies

- [vec_proxy_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`*Equality*

Description

`vec_equal()` tests if two vectors are equal.

Usage

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

Arguments

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

Value

A logical vector the same size as the common size of `x` and `y`. Will only contain NAs if `na_equal` is `FALSE`.

Dependencies

- [vec_cast_common\(\)](#) with fallback
- [vec_recycle_common\(\)](#)
- [vec_proxy_equal\(\)](#)

See Also

[vec_detect_missing\(\)](#)

Examples

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

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

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

vec_expand_grid *Create a data frame from all combinations of the inputs*

Description

vec_expand_grid() creates a new data frame by creating a grid of all possible combinations of the input vectors. It is inspired by [expand.grid\(\)](#). Compared with [expand.grid\(\)](#), it:

- Produces sorted output by default by varying the first column the slowest, rather than the fastest. Control this with `.vary`.
- Never converts strings to factors.
- Does not add additional attributes.
- Drops NULL inputs.
- Can expand any vector type, including data frames and [records](#).

Usage

```
vec_expand_grid(
  ...,
  .vary = "slowest",
  .name_repair = "check_unique",
  .error_call = current_env()
)
```

Arguments

...	Name-value pairs. The name will become the column name in the resulting data frame.
.vary	One of: <ul style="list-style-type: none"> • "slowest" to vary the first column slowest. This produces sorted output and is generally the most useful. • "fastest" to vary the first column fastest. This matches the behavior of expand.grid().
.name_repair	One of "check_unique", "unique", "universal", "minimal", "unique_quiet", or "universal_quiet". See vec_as_names() for the meaning of these options.
.error_call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of abort() for more information.

Details

If any input is empty (i.e. size 0), then the result will have 0 rows.

If no inputs are provided, the result is a 1 row data frame with 0 columns. This is consistent with the fact that `prod()` with no inputs returns 1.

Value

A data frame with as many columns as there are inputs in `...` and as many rows as the `prod()` of the sizes of the inputs.

Examples

```
vec_expand_grid(x = 1:2, y = 1:3)

# Use `.vary` to match `expand.grid()`:
vec_expand_grid(x = 1:2, y = 1:3, .vary = "fastest")

# Can also expand data frames
vec_expand_grid(
  x = data_frame(a = 1:2, b = 3:4),
  y = 1:4
)
```

<code>vec_fill_missing</code>	<i>Fill in missing values with the previous or following value</i>
-------------------------------	--

Description**[Experimental]**

`vec_fill_missing()` fills gaps of missing values with the previous or following non-missing value.

Usage

```
vec_fill_missing(
  x,
  direction = c("down", "up", "downup", "updown"),
  max_fill = NULL
)
```

Arguments

<code>x</code>	A vector
<code>direction</code>	Direction in which to fill missing values. Must be either "down", "up", "downup", or "updown".
<code>max_fill</code>	A single positive integer specifying the maximum number of sequential missing values that will be filled. If NULL, there is no limit.

Examples

```
x <- c(NA, NA, 1, NA, NA, NA, 3, NA, NA)

# Filling down replaces missing values with the previous non-missing value
vec_fill_missing(x, direction = "down")

# To also fill leading missing values, use `"downup"`
vec_fill_missing(x, direction = "downup")

# Limit the number of sequential missing values to fill with `max_fill`
vec_fill_missing(x, max_fill = 1)

# Data frames are filled rowwise. Rows are only considered missing
# if all elements of that row are missing.
y <- c(1, NA, 2, NA, NA, 3, 4, NA, 5)
df <- data_frame(x = x, y = y)
df

vec_fill_missing(df)
```

vec_init

*Initialize a vector***Description**

Initialize a vector

Usage

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

Arguments

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

Dependencies

- vec_slice()

Examples

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

# The "missing" value for a data frame is a row that is entirely missing
vec_init(mtcars, 2)

# The "missing" value for a list is `NULL`
vec_init(list(), 3)
```

vec_interleave *Interleave many vectors into one vector*

Description

vec_interleave() combines multiple vectors together, much like `vec_c()`, but does so in such a way that the elements of each vector are interleaved together.

It is a more efficient equivalent to the following usage of `vec_c()`:

```
vec_interleave(x, y) == vec_c(x[1], y[1], x[2], y[2], ..., x[n], y[n])
```

Usage

```
vec_interleave(
  ...,
  .ptype = NULL,
  .name_spec = NULL,
  .name_repair = c("minimal", "unique", "check_unique", "universal", "unique_quiet",
    "universal_quiet")
)
```

Arguments

...	Vectors to interleave. These will be recycled to a common size.
.ptype	If NULL, the default, the output type is determined by computing the common type across all elements of ... Alternatively, you can supply .ptype to give the output known type. If <code>getOption("vctrs.no_guessing")</code> is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.
.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 <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: <ul style="list-style-type: none"> • 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 <code>"{outer}_{inner}"</code>. • An <code>rlang::zap()</code> object, in which case both outer and inner names are ignored and the result is unnamed. See the name specification topic .
.name_repair	How to repair names, see repair options in <code>vec_as_names()</code> .

Dependencies**vctrs dependencies:**

- [list_unchop\(\)](#)

Examples

```
# The most common case is to interleave two vectors
vec_interleave(1:3, 4:6)

# But you aren't restricted to just two
vec_interleave(1:3, 4:6, 7:9, 10:12)

# You can also interleave data frames
x <- data_frame(x = 1:2, y = c("a", "b"))
y <- data_frame(x = 3:4, y = c("c", "d"))

vec_interleave(x, y)
```

vec_locate_matches *Locate observations matching specified conditions*

Description**[Experimental]**

vec_locate_matches() is a more flexible version of [vec_match\(\)](#) used to identify locations where each value of needles matches one or multiple values in haystack. Unlike [vec_match\(\)](#), [vec_locate_matches\(\)](#) returns all matches by default, and can match on binary conditions other than equality, such as >, >=, <, and <=.

Usage

```
vec_locate_matches(
  needles,
  haystack,
  ...,
  condition = "==",
  filter = "none",
  incomplete = "compare",
  no_match = NA_integer_,
  remaining = "drop",
  multiple = "all",
  relationship = "none",
  nan_distinct = FALSE,
  chr_proxy_collate = NULL,
  needles_arg = "needles",
  haystack_arg = "haystack",
  error_call = current_env()
)
```

Arguments

needles, haystack	<p>Vectors used for matching.</p> <ul style="list-style-type: none"> • needles represents the vector to search for. • haystack represents the vector to search in. <p>Prior to comparison, needles and haystack are coerced to the same type.</p>
...	These dots are for future extensions and must be empty.
condition	<p>Condition controlling how needles should be compared against haystack to identify a successful match.</p> <ul style="list-style-type: none"> • One of: "==" , ">" , ">=" , "<" , or "<=". • For data frames, a length 1 or ncol(needles) character vector containing only the above options, specifying how matching is determined for each column.
filter	<p>Filter to be applied to the matched results.</p> <ul style="list-style-type: none"> • "none" doesn't apply any filter. • "min" returns only the minimum haystack value matching the current needle. • "max" returns only the maximum haystack value matching the current needle. • For data frames, a length 1 or ncol(needles) character vector containing only the above options, specifying a filter to apply to each column. <p>Filters don't have any effect on "==" conditions, but are useful for computing "rolling" matches with other conditions.</p> <p>A filter can return multiple haystack matches for a particular needle if the maximum or minimum haystack value is duplicated in haystack. These can be further controlled with <code>multiple</code>.</p>
incomplete	<p>Handling of missing and incomplete values in needles.</p> <ul style="list-style-type: none"> • "compare" uses condition to determine whether or not a missing value in needles matches a missing value in haystack. If condition is ==, >=, or <=, then missing values will match. • "match" always allows missing values in needles to match missing values in haystack, regardless of the condition. • "drop" drops incomplete values in needles from the result. • "error" throws an error if any needles are incomplete. • If a single integer is provided, this represents the value returned in the haystack column for values of needles that are incomplete. If no_match = NA, setting incomplete = NA forces incomplete values in needles to be treated like unmatched values. <p>nan_distinct determines whether a NA is allowed to match a NaN.</p>
no_match	<p>Handling of needles without a match.</p> <ul style="list-style-type: none"> • "drop" drops needles with zero matches from the result. • "error" throws an error if any needles have zero matches.

- If a single integer is provided, this represents the value returned in the haystack column for values of needles that have zero matches. The default represents an unmatched needle with NA.
- remaining Handling of haystack values that needles never matched.
- "drop" drops remaining haystack values from the result. Typically, this is the desired behavior if you only care when needles has a match.
 - "error" throws an error if there are any remaining haystack values.
 - If a single integer is provided (often NA), this represents the value returned in the needles column for the remaining haystack values that needles never matched. Remaining haystack values are always returned at the end of the result.
- multiple Handling of needles with multiple matches. For each needle:
- "all" returns all matches detected in haystack.
 - "any" returns any match detected in haystack with no guarantees on which match will be returned. It is often faster than "first" and "last" if you just need to detect if there is at least one match.
 - "first" returns the first match detected in haystack.
 - "last" returns the last match detected in haystack.
- relationship Handling of the expected relationship between needles and haystack. If the expectations chosen from the list below are invalidated, an error is thrown.
- "none" doesn't perform any relationship checks.
 - "one-to-one" expects:
 - Each value in needles matches at most 1 value in haystack.
 - Each value in haystack matches at most 1 value in needles.
 - "one-to-many" expects:
 - Each value in needles matches any number of values in haystack.
 - Each value in haystack matches at most 1 value in needles.
 - "many-to-one" expects:
 - Each value in needles matches at most 1 value in haystack.
 - Each value in haystack matches any number of values in needles.
 - "many-to-many" expects:
 - Each value in needles matches any number of values in haystack.
 - Each value in haystack matches any number of values in needles.

This performs no checks, and is identical to "none", but is provided to allow you to be explicit about this relationship if you know it exists.
 - "warn-many-to-many" doesn't assume there is any known relationship, but will warn if needles and haystack have a many-to-many relationship (which is typically unexpected), encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying "many-to-many".
- relationship is applied after filter and multiple to allow potential multiple matches to be filtered out first.
- relationship doesn't handle cases where there are zero matches. For that, see no_match and remaining.

nan_distinct	A single logical specifying whether or not NaN should be considered distinct from NA for double and complex vectors. If TRUE, NaN will always be ordered between NA and non-missing numbers.
chr_proxy_collate	A function generating an alternate representation of character vectors to use for collation, often used for locale-aware ordering. <ul style="list-style-type: none"> • If NULL, no transformation is done. • Otherwise, this must be a function of one argument. If the input contains a character vector, it will be passed to this function after it has been translated to UTF-8. This function should return a character vector with the same length as the input. The result should sort as expected in the C-locale, regardless of encoding. <p>For data frames, chr_proxy_collate will be applied to all character columns. Common transformation functions include: tolower() for case-insensitive ordering and stringi::stri_sort_key() for locale-aware ordering.</p>
needles_arg, haystack_arg	Argument tags for needles and haystack used in error messages.
error_call	The execution environment of a currently running function, e.g. caller_env(). The function will be mentioned in error messages as the source of the error. See the call argument of abort() for more information.

Details

vec_match() is identical to (but often slightly faster than):

```
vec_locate_matches(
  needles,
  haystack,
  condition = "==",
  multiple = "first",
  nan_distinct = TRUE
)
```

vec_locate_matches() is extremely similar to a SQL join between needles and haystack, with the default being most similar to a left join.

Be very careful when specifying match conditions. If a condition is misspecified, it is very easy to accidentally generate an exponentially large number of matches.

Value

A two column data frame containing the locations of the matches.

- needles is an integer vector containing the location of the needle currently being matched.
- haystack is an integer vector containing the location of the corresponding match in the haystack for the current needle.

Dependencies of vec_locate_matches()

- [vec_order_radix\(\)](#)
- [vec_detect_complete\(\)](#)

Examples

```
x <- c(1, 2, NA, 3, NaN)
y <- c(2, 1, 4, NA, 1, 2, NaN)

# By default, for each value of `x`, all matching locations in `y` are
# returned
matches <- vec_locate_matches(x, y)
matches

# The result can be used to slice the inputs to align them
data_frame(
  x = vec_slice(x, matches$needles),
  y = vec_slice(y, matches$haystack)
)

# If multiple matches are present, control which is returned with `multiple`
vec_locate_matches(x, y, multiple = "first")
vec_locate_matches(x, y, multiple = "last")
vec_locate_matches(x, y, multiple = "any")

# Use `relationship` to add constraints and error on multiple matches if
# they aren't expected
try(vec_locate_matches(x, y, relationship = "one-to-one"))

# In this case, the `NA` in `y` matches two rows in `x`
try(vec_locate_matches(x, y, relationship = "one-to-many"))

# By default, `NA` is treated as being identical to `NaN`.
# Using `nan_distinct = TRUE` treats `NA` and `NaN` as different values, so
# `NA` can only match `NA`, and `NaN` can only match `NaN`.
vec_locate_matches(x, y, nan_distinct = TRUE)

# If you never want missing values to match, set `incomplete = NA` to return
# `NA` in the `haystack` column anytime there was an incomplete value
# in `needles`.
vec_locate_matches(x, y, incomplete = NA)

# Using `incomplete = NA` allows us to enforce the one-to-many relationship
# that we couldn't before
vec_locate_matches(x, y, relationship = "one-to-many", incomplete = NA)

# `no_match` allows you to specify the returned value for a needle with
# zero matches. Note that this is different from an incomplete value,
# so specifying `no_match` allows you to differentiate between incomplete
# values and unmatched values.
vec_locate_matches(x, y, incomplete = NA, no_match = 0L)
```

```

# If you want to require that every `needle` has at least 1 match, set
# `no_match` to `"error"`:
try(vec_locate_matches(x, y, incomplete = NA, no_match = "error"))

# By default, `vec_locate_matches()` detects equality between `needles` and
# `haystack`. Using `condition`, you can detect where an inequality holds
# true instead. For example, to find every location where `x[[i]] >= y`:
matches <- vec_locate_matches(x, y, condition = ">=")

data_frame(
  x = vec_slice(x, matches$needles),
  y = vec_slice(y, matches$haystack)
)

# You can limit which matches are returned with a `filter`. For example,
# with the above example you can filter the matches returned by `x[[i]] >= y`
# down to only the ones containing the maximum `y` value of those matches.
matches <- vec_locate_matches(x, y, condition = ">=", filter = "max")

# Here, the matches for the `3` needle value have been filtered down to
# only include the maximum haystack value of those matches, `2`. This is
# often referred to as a rolling join.
data_frame(
  x = vec_slice(x, matches$needles),
  y = vec_slice(y, matches$haystack)
)

# In the very rare case that you need to generate locations for a
# cross match, where every value of `x` is forced to match every
# value of `y` regardless of what the actual values are, you can
# replace `x` and `y` with integer vectors of the same size that contain
# a single value and match on those instead.
x_proxy <- vec_rep(1L, vec_size(x))
y_proxy <- vec_rep(1L, vec_size(y))
nrow(vec_locate_matches(x_proxy, y_proxy))
vec_size(x) * vec_size(y)

# By default, missing values will match other missing values when using
# `==`, `>=`, or `<=` conditions, but not when using `>` or `<` conditions.
# This is similar to how `vec_compare(x, y, na_equal = TRUE)` works.
x <- c(1, NA)
y <- c(NA, 2)

vec_locate_matches(x, y, condition = "<=")
vec_locate_matches(x, y, condition = "<")

# You can force missing values to match regardless of the `condition`
# by using `incomplete = "match"`
vec_locate_matches(x, y, condition = "<", incomplete = "match")

# You can also use data frames for `needles` and `haystack`. The
# `condition` will be recycled to the number of columns in `needles`, or
# you can specify varying conditions per column. In this example, we take

```

```

# a vector of date `values` and find all locations where each value is
# between lower and upper bounds specified by the `haystack`.
values <- as.Date("2019-01-01") + 0:9
needles <- data_frame(lower = values, upper = values)

set.seed(123)
lower <- as.Date("2019-01-01") + sample(10, 10, replace = TRUE)
upper <- lower + sample(3, 10, replace = TRUE)
haystack <- data_frame(lower = lower, upper = upper)

# (values >= lower) & (values <= upper)
matches <- vec_locate_matches(needles, haystack, condition = c(">=", "<="))

data_frame(
  lower = vec_slice(lower, matches$haystack),
  value = vec_slice(values, matches$needle),
  upper = vec_slice(upper, matches$haystack)
)

```

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,
  needles_arg = "",
  haystack_arg = ""
)

vec_in(
  needles,
  haystack,
  ...,
  na_equal = TRUE,
  needles_arg = "",
  haystack_arg = ""
)

```

Arguments

needles, haystack	Vector of needles to search for in vector haystack. haystack should usually be unique; if not <code>vec_match()</code> 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.
needles_arg, haystack_arg	Argument tags for needles and haystack used in error messages.

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.

Dependencies

- `vec_cast_common()` with fallback
- `vec_proxy_equal()`

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_names

*Get or set the names of a vector***Description**

These functions work like `rlang::names2()`, `names()` and `names<-()`, except that they return or modify the the rowwise names of the vector. These are:

- The usual `names()` for atomic vectors and lists
 - The row names for data frames and matrices
 - The names of the first dimension for arrays
- Rowwise names are size consistent: the length of the names always equals `vec_size()`.

`vec_names2()` returns the repaired names from a vector, even if it is unnamed. See `vec_as_names()` for details on name repair.

`vec_names()` is a bare-bones version that returns `NULL` if the vector is unnamed.

`vec_set_names()` sets the names or removes them.

Usage

```
vec_names2(
  x,
  ...,
  repair = c("minimal", "unique", "universal", "check_unique", "unique_quiet",
            "universal_quiet"),
  quiet = FALSE
)
```

```
vec_names(x)
```

```
vec_set_names(x, names)
```

Arguments

- | | |
|---------------------|---|
| <code>x</code> | A vector with names |
| <code>...</code> | These dots are for future extensions and must be empty. |
| <code>repair</code> | Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", "universal", "unique_quiet", or "universal_quiet". 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"> • Minimal names are never <code>NULL</code> or <code>NA</code>. When an element doesn't have a name, its minimal name is an empty string. • 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.

The options "unique_quiet" and "universal_quiet" are here to help the user who calls this function indirectly, via another function which exposes repair but not quiet. Specifying repair = "unique_quiet" is like specifying repair = "unique", quiet = TRUE. When the "*_quiet" options are used, any setting of quiet is silently overridden.

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. Users can silence the name repair messages by setting the "rlib_name_repair_verbosity" global option to "quiet".
names	A character vector, or NULL.

Value

vec_names2() returns the names of x, repaired. vec_names() returns the names of x or NULL if unnamed. vec_set_names() returns x with names updated.

Examples

```
vec_names2(1:3)
vec_names2(1:3, repair = "unique")
vec_names2(c(a = 1, b = 2))

# `vec_names()` consistently returns the rowwise names of data frames and arrays:
vec_names(data.frame(a = 1, b = 2))
names(data.frame(a = 1, b = 2))
vec_names(mtcars)
names(mtcars)
vec_names(Titanic)
names(Titanic)

vec_set_names(1:3, letters[1:3])
vec_set_names(data.frame(a = 1:3), letters[1:3])
```

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
...	These dots are for future extensions and must be empty.
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`.

Differences with `order()`

Unlike the `na.last` argument of `order()` which decides the positions of missing values irrespective of the decreasing argument, the `na_value` argument of `vec_order()` interacts with `direction`. If missing values are considered the largest value, they will appear last in ascending order, and first in descending order.

Dependencies of `vec_order()`

- [vec_proxy_order\(\)](#)

Dependencies of `vec_sort()`

- [vec_proxy_order\(\)](#)
- [vec_order\(\)](#)
- [vec_slice\(\)](#)

Examples

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

# Missing values interpreted as largest values are last when
# in increasing order:
vec_order(c(1, NA), na_value = "largest", direction = "asc")
vec_order(c(1, NA), na_value = "largest", direction = "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, ..., x_arg = "", call = caller_env())

vec_ptype_common(..., .ptype = NULL, .arg = "", .call = caller_env())

vec_ptype_show(...)
```

Arguments

x	A vector
...	For vec_ptype(), these dots are for future extensions and must be empty. For vec_ptype_common() and vec_ptype_show(), vector inputs.
x_arg	Argument name for x. This is used in error messages to inform the user about the locations of incompatible types.
call, .call	The execution environment of a currently running function, e.g. caller_env(). The function will be mentioned in error messages as the source of the error. See the call argument of abort() for more information.

<code>.ptype</code>	If NULL, the default, the output type is determined by computing the common type across all elements of Alternatively, you can supply <code>.ptype</code> to give the output known type. If <code>getOption("vctrs.no_guessing")</code> is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.
<code>.arg</code>	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.

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.

`vec_ptype()` is a *performance* generic. It is not necessary to implement it because the default method will work for any `vctrs` type. However the default method builds around other `vctrs` primitives like `vec_slice()` which incurs performance costs. If your class has a static prototype, you might consider implementing a custom `vec_ptype()` method that returns a constant. This will improve the performance of your class in many cases ([common type](#) imputation in particular).

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.

Dependencies of `vec_ptype()`

- [vec_slice\(\)](#) for returning an empty slice

Dependencies of `vec_ptype_common()`

- [vec_ptype2\(\)](#)
- [vec_ptype_finalise\(\)](#)

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_ptype2.logical *Find the common type for a pair of vectors*

Description

vec_ptype2() defines the coercion hierarchy for a set of related vector types. Along with `vec_cast()`, this generic forms the foundation of type coercions in `vctrs`.

vec_ptype2() is relevant when you are implementing `vctrs` methods for your class, but it should not usually be called directly. If you need to find the common type of a set of inputs, call `vec_ptype_common()` instead. This function supports multiple inputs and *finalises* the common type.

Usage

```

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

```

```

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

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

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

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

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

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

vec_ptype2(
  x,
  y,
  ...,
  x_arg = caller_arg(x),
  y_arg = caller_arg(y),
  call = caller_env()
)

```

Arguments

<code>x, y</code>	Vector types.
<code>...</code>	These dots are for future extensions and must be empty.
<code>x_arg, 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 stop_incompatible_type()).
<code>call</code>	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of abort() for more information.

Implementing coercion methods

- For an overview of how these generics work and their roles in `vctrs`, see [?theory-faq-coercion](#).
- For an example of implementing coercion methods for simple vectors, see [?howto-faq-coercion](#).
- For an example of implementing coercion methods for data frame subclasses, see [?howto-faq-coercion-data-frame](#).
- For a tutorial about implementing `vctrs` classes from scratch, see `vignette("s3-vector")`.

Dependencies

- [vec_ptype\(\)](#) is applied to `x` and `y`

See Also

`stop_incompatible_type()` when you determine from the attributes that an input can't be cast to the target type.

 vec_rank

Compute ranks

Description

`vec_rank()` computes the sample ranks of a vector. For data frames, ranks are computed along the rows, using all columns after the first to break ties.

Usage

```
vec_rank(
  x,
  ...,
  ties = c("min", "max", "sequential", "dense"),
  incomplete = c("rank", "na"),
  direction = "asc",
  na_value = "largest",
  nan_distinct = FALSE,
  chr_proxy_collate = NULL
)
```

Arguments

<code>x</code>	A vector
<code>...</code>	These dots are for future extensions and must be empty.
<code>ties</code>	Ranking of duplicate values. <ul style="list-style-type: none"> • "min": Use the current rank for all duplicates. The next non-duplicate value will have a rank incremented by the number of duplicates present. • "max": Use the current rank + <code>n_duplicates - 1</code> for all duplicates. The next non-duplicate value will have a rank incremented by the number of duplicates present. • "sequential": Use an increasing sequence of ranks starting at the current rank, applied to duplicates in order of appearance. • "dense": Use the current rank for all duplicates. The next non-duplicate value will have a rank incremented by 1, effectively removing any gaps in the ranking.
<code>incomplete</code>	Ranking of missing and incomplete observations. <ul style="list-style-type: none"> • "rank": Rank incomplete observations normally. Missing values within incomplete observations will be affected by <code>na_value</code> and <code>nan_distinct</code>. • "na": Don't rank incomplete observations at all. Instead, they are given a rank of NA. In this case, <code>na_value</code> and <code>nan_distinct</code> have no effect.

direction	<p>Direction to sort in.</p> <ul style="list-style-type: none"> • A single "asc" or "desc" for ascending or descending order respectively. • For data frames, a length 1 or ncol(x) character vector containing only "asc" or "desc", specifying the direction for each column.
na_value	<p>Ordering of missing values.</p> <ul style="list-style-type: none"> • A single "largest" or "smallest" for ordering missing values as the largest or smallest values respectively. • For data frames, a length 1 or ncol(x) character vector containing only "largest" or "smallest", specifying how missing values should be ordered within each column.
nan_distinct	<p>A single logical specifying whether or not NaN should be considered distinct from NA for double and complex vectors. If TRUE, NaN will always be ordered between NA and non-missing numbers.</p>
chr_proxy_collate	<p>A function generating an alternate representation of character vectors to use for collation, often used for locale-aware ordering.</p> <ul style="list-style-type: none"> • If NULL, no transformation is done. • Otherwise, this must be a function of one argument. If the input contains a character vector, it will be passed to this function after it has been translated to UTF-8. This function should return a character vector with the same length as the input. The result should sort as expected in the C-locale, regardless of encoding. <p>For data frames, chr_proxy_collate will be applied to all character columns. Common transformation functions include: tolower() for case-insensitive ordering and stringi::stri_sort_key() for locale-aware ordering.</p>

Details

Unlike `base::rank()`, when `incomplete = "rank"` all missing values are given the same rank, rather than an increasing sequence of ranks. When `nan_distinct = FALSE`, NaN values are given the same rank as NA, otherwise they are given a rank that differentiates them from NA.

Like `vec_order_radix()`, ordering is done in the C-locale. This can affect the ranks of character vectors, especially regarding how uppercase and lowercase letters are ranked. See the documentation of `vec_order_radix()` for more information.

Dependencies

- `vec_order_radix()`
- `vec_slice()`

Examples

```
x <- c(5L, 6L, 3L, 3L, 5L, 3L)

vec_rank(x, ties = "min")
vec_rank(x, ties = "max")
```

```

# Sequential ranks use an increasing sequence for duplicates
vec_rank(x, ties = "sequential")

# Dense ranks remove gaps between distinct values,
# even if there are duplicates
vec_rank(x, ties = "dense")

y <- c(NA, x, NA, NaN)

# Incomplete values match other incomplete values by default, and their
# overall position can be adjusted with `na_value`
vec_rank(y, na_value = "largest")
vec_rank(y, na_value = "smallest")

# NaN can be ranked separately from NA if required
vec_rank(y, nan_distinct = TRUE)

# Rank in descending order. Since missing values are the largest value,
# they are given a rank of `1` when ranking in descending order.
vec_rank(y, direction = "desc", na_value = "largest")

# Give incomplete values a rank of `NA` by setting `incomplete = "na"`
vec_rank(y, incomplete = "na")

# Can also rank data frames, using columns after the first to break ties
z <- c(2L, 3L, 4L, 4L, 5L, 2L)
df <- data_frame(x = x, z = z)
df

vec_rank(df)

```

vec_recycle

Vector recycling

Description

`vec_recycle(x, size)` recycles a single vector to a given size. `vec_recycle_common(...)` recycles multiple vectors to their common size. All functions obey the [vctrs recycling rules](#), 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 = "", call = caller_env())
```

```
vec_recycle_common(..., .size = NULL, .arg = "", .call = caller_env())
```


Arguments

x	A vector to recycle.
size	Desired output size.
...	Depending on the function used: <ul style="list-style-type: none"> • For <code>vec_recycle_common()</code>, vectors to recycle. • For <code>vec_recycle()</code>, these dots should be empty.
x_arg	Argument name for x. These are used in error messages to inform the user about which argument has an incompatible size.
call, .call	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of <code>abort()</code> for more information.
.size	Desired output size. If omitted, will use the common size from <code>vec_size_common()</code> .
.arg	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.

Dependencies

- `vec_slice()`

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_seq_along

Useful sequences

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))
```

vec_size	<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.

`list_sizes()` returns an integer vector containing the size of each element of a list. It is nearly equivalent to, but faster than, `map_int(x, vec_size)`, with the exception that `list_sizes()` will error on non-list inputs, as defined by `obj_is_list()`. `list_sizes()` is to `vec_size()` as `lengths()` is to `length()`.

Usage

```
vec_size(x)

vec_size_common(
  ...,
  .size = NULL,
  .absent = 0L,
  .arg = "",
  .call = caller_env()
)

list_sizes(x)

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.
<code>.arg</code>	An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.
<code>.call</code>	The execution environment of a currently running function, e.g. <code>caller_env()</code> . The function will be mentioned in error messages as the source of the error. See the <code>call</code> argument of <code>abort()</code> for more information.

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.

Dependencies

- `vec_proxy()`

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)

list_sizes(list("a", 1:5, letters))
```

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, the 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.

Dependencies

- [vec_group_loc\(\)](#)
- [vec_chop\(\)](#)

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*Find and count unique values*

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.

Dependencies

- `vec_proxy_equal()`

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_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

?howto-faq-coercion-data-frame, [6](#)
%0%, [102](#)
%in%, [86](#)

abort(), [3](#), [5](#), [7](#), [38](#), [48](#), [51](#), [53](#), [56](#), [59](#), [62](#), [65](#),
[67](#), [75](#), [82](#), [90](#), [93](#), [97](#), [99](#)
anyDuplicated(), [72](#)
as.list(), [66](#)
as_list_of(list_of), [33](#)

base type, [53](#), [65](#)
base::c(), [39](#), [60](#), [63](#)
base::data.frame(), [3](#)
base::inverse.rle(), [48](#)
base::make.names(), [55](#), [57](#)
base::rank(), [95](#)
base::rle(), [40](#), [41](#), [48](#)

common type, [91](#)

data.frame, [54](#)
data_frame, [3](#)
data_frame(), [6](#)
developer FAQ page, [11](#)
df_cast(df_ptype2), [6](#)
df_cast(), [45](#)
df_list, [5](#)
df_list(), [4](#), [37](#)
df_ptype2, [6](#)
df_ptype2(), [19](#), [45](#)
duplicated(), [72](#)

expand.grid(), [75](#)
expression, [54](#)

faq-compatibility-types, [7](#)
faq-error-incompatible-attributes, [9](#)
faq-error-scalar-type, [9](#)
finalised, [91](#)
finalises, [92](#)

howto guide, [40](#)
howto-faq-coercion, [11](#)
howto-faq-coercion-data-frame, [15](#)
howto-faq-fix-scalar-type-error, [23](#)

identity, [40](#)
incomplete, [34](#), [80](#), [94](#)
internal-faq-matches-algorithm, [24](#)
internal-faq-ptype2-identity, [31](#), [91](#)
intersect(), [50](#)
is.na(), [34](#)
is_list_of(list_of), [33](#)

length(), [98](#)
lengths(), [98](#)
list_all_size(obj_is_list), [37](#)
list_all_vectors(obj_is_list), [37](#)
list_check_all_size(obj_is_list), [37](#)
list_check_all_vectors(obj_is_list), [37](#)
list_drop_empty, [33](#)
list_of, [33](#)
list_sizes(vec_size), [98](#)
list_sizes(), [38](#)
list_unchop(vec_chop), [66](#)
list_unchop(), [79](#)

match(), [86](#)
merge(), [86](#)
missing, [34](#)

name specification topic, [36](#), [62](#), [67](#), [78](#)
name_spec, [35](#)
names(), [87](#)
new_data_frame, [37](#)
new_data_frame(), [4–6](#)
new_vctr(), [39](#)

obj_check_list(obj_is_list), [37](#)
obj_check_vector(vector_checks), [53](#)
obj_is_list, [37](#)
obj_is_list(), [53](#), [98](#)

- obj_is_vector (vector-checks), 53
- prod(), 76
- record, 72
- record vector, 34
- records, 75
- recycle, 100
- recycled, 4, 5, 48, 78
- recycling rules, 59
- reference-faq-compatibility, 39
- reserved, 57
- rlang::names2(), 57, 87
- rlang::zap(), 36, 62, 67, 78
- runs, 40
- seq_along(), 97
- setdiff(), 50
- size, 91
- split(), 100
- stats::complete.cases(), 72
- stop_incompatible_cast(), 65
- stop_incompatible_type(), 7, 34, 65, 93, 94
- theory overview, 40
- theory-faq-coercion, 41
- theory-faq-recycling, 46
- tib_cast(df_ptype2), 6
- tib_cast(), 19
- tib_ptype2(df_ptype2), 6
- tib_ptype2(), 19
- type, 99
- union(), 50
- unique(), 101
- unspecified, 91
- user FAQ, 23
- vctrs recycling rules, 96
- vec_rep, 47
- vec_set, 49
- vec_any_missing(missing), 34
- vec_as_names, 55
- vec_as_names(), 3, 5, 59, 62, 67, 75, 78, 87
- vec_assign(), 60
- vec_bind, 58
- vec_c, 61
- vec_c(), 39, 51, 52, 59, 60, 66–68, 78
- vec_cast, 64
- vec_cast(), 9, 40, 51, 52, 65, 92
- vec_cast.vctrs_list_of(list_of), 33
- vec_cast_common(vec_cast), 64
- vec_cast_common(), 60, 63, 70, 74, 86
- vec_cbind(vec_bind), 58
- vec_cbind(), 16, 63
- vec_check_size(vector-checks), 53
- vec_chop, 66
- vec_chop(), 100
- vec_compare, 69
- vec_count, 70
- vec_count(), 39
- vec_detect_complete, 71
- vec_detect_complete(), 35, 83
- vec_detect_missing(missing), 34
- vec_detect_missing(), 74
- vec_duplicate, 72, 102
- vec_duplicate_any(vec_duplicate), 72
- vec_duplicate_detect(vec_duplicate), 72
- vec_duplicate_id(vec_duplicate), 72
- vec_equal, 74
- vec_expand_grid, 75
- vec_fill_missing, 76
- vec_group_loc(), 100
- vec_identify_runs(runs), 40
- vec_in(vec_match), 85
- vec_init, 77
- vec_init(), 60
- vec_init_along(vec_seq_along), 97
- vec_interleave, 78
- vec_is_empty(vec_size), 98
- vec_is_empty(), 102
- vec_locate_matches, 79
- vec_match, 85
- vec_match(), 79, 82
- vec_names, 87
- vec_names2(vec_names), 87
- vec_order, 88
- vec_order(), 71, 89
- vec_order_radix(), 83, 95
- vec_proxy(), 39, 40, 53, 60, 63, 99
- vec_proxy_compare(), 69, 70
- vec_proxy_equal(), 35, 51, 52, 71, 73, 74, 86, 101
- vec_proxy_order(), 89
- vec_ptype, 90
- vec_ptype(), 93
- vec_ptype2(vec_ptype2.logical), 92

`vec_ptype2()`, [9](#), [40](#), [51](#), [52](#), [64](#), [65](#), [91](#)
`vec_ptype2.logical`, [92](#)
`vec_ptype2.vctrs_list_of(list_of)`, [33](#)
`vec_ptype_common(vec_ptype)`, [90](#)
`vec_ptype_common()`, [92](#)
`vec_ptype_finalise()`, [91](#)
`vec_ptype_show(vec_ptype)`, [90](#)
`vec_rank`, [94](#)
`vec_rbind(vec_bind)`, [58](#)
`vec_rbind()`, [16](#), [39](#), [63](#)
`vec_recycle`, [96](#)
`vec_recycle()`, [100](#)
`vec_recycle_common(vec_recycle)`, [96](#)
`vec_recycle_common()`, [4](#), [5](#), [47](#), [70](#), [74](#)
`vec_rep(vec-rep)`, [47](#)
`vec_rep()`, [40](#)
`vec_rep_each(vec-rep)`, [47](#)
`vec_rep_each()`, [40](#)
`vec_restore()`, [39](#), [40](#), [60](#), [63](#)
`vec_run_sizes(runs)`, [40](#)
`vec_seq_along`, [97](#)
`vec_set_difference(vec-set)`, [49](#)
`vec_set_intersect(vec-set)`, [49](#)
`vec_set_names(vec_names)`, [87](#)
`vec_set_symmetric_difference(vec-set)`,
[49](#)
`vec_set_union(vec-set)`, [49](#)
`vec_size`, [98](#)
`vec_size()`, [87](#), [96](#)
`vec_size_common(vec_size)`, [98](#)
`vec_size_common()`, [47](#), [97](#)
`vec_slice()`, [33](#), [39](#), [49](#), [51](#), [52](#), [68](#), [71](#), [89](#),
[91](#), [95](#), [97](#), [100](#)
`vec_sort(vec_order)`, [88](#)
`vec_split`, [100](#)
`vec_unique`, [101](#)
`vec_unique()`, [73](#)
`vec_unique_count(vec_unique)`, [101](#)
`vec_unique_loc(vec_unique)`, [101](#)
`vec_unrep(vec-rep)`, [47](#)
`vec_unrep()`, [40](#), [41](#)
`vector-checks`, [53](#)
`vector_recycling_rules`
[\(theory-faq-recycling\)](#), [46](#)

`zapped`, [59](#)