Package 'vctrs'

November 12, 2020

Title Vector Helpers

Version 0.3.5

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/

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

Depends R (>= 3.3)

Imports ellipsis (>= 0.2.0),
digest,
glue,
rlang (>= 0.4.7)

Suggests bit64,
covr,
crayon,
dplyr (>= 0.8.5),
generics,
knitr,
pillar (>= 1.4.4),
pkgdown,
rmarkdown,
testthat (>= 2.3.0),
tibble,
withr,
xml2,
waldo (>= 0.2.0),
zeallot

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.1

R topics documented:

`data_frame` *Construct a data frame*

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")
)
```

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"` or `"minimal"`. See `vec_as_names()` for the meaning of these options.

Details

If no column names are supplied, `""` will be used as a default 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"` 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 spliced. 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 spliced
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,
  .name_repair = c("check_unique", "unique", "universal", "minimal")
)
```

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.
.name_repair	One of "check_unique", "unique", "universal" or "minimal". See vec_as_names() for the meaning of these options.

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 spliced. 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 = "")

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

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

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

Arguments

`x`, `y`, `to`	Subclasses of data frame.
`...`	If you call `df_ptype2()` or `df_cast()` from a `vec_ptype2()` or `vec_cast()` method, you must forward the dots passed to your method on to `df_ptype2()` or `df_cast()`.
`x_arg`	Argument names for `x` and `y`. These are used in error messages to inform the user about the locations of incompatible types (see `stop_incompatible_type()`).
`y_arg`	Argument names for `x` and `y`. These are used in error messages to inform the user about the locations of incompatible types (see `stop_incompatible_type()`).
`to_arg`	Argument names for `x` and `to`. These are used in error messages to inform the user about the locations of incompatible types (see `stop_incompatible_type()`).

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:

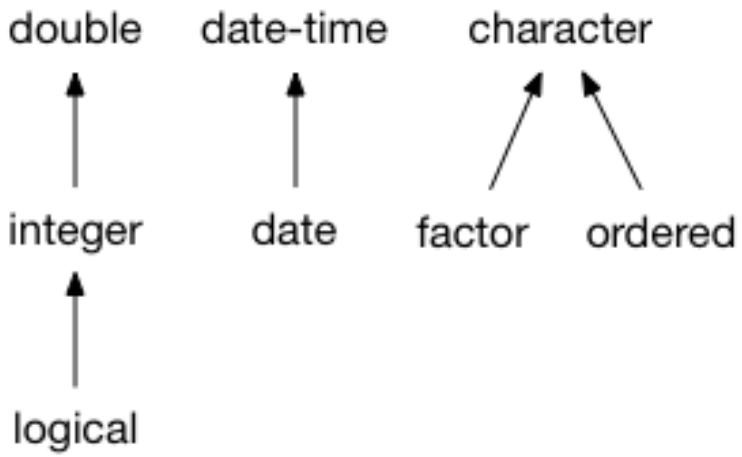
- An incompatible type error is thrown because some of the types are not compatible:

```
df1 <- data.frame(x = 1:3)
df2 <- data.frame(x = "foo")
dplyr::bind_rows(df1, df2)
#> Error: 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: 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-fr

---

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-fr
- 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: All columns in a tibble must be vectors.
#> x Column `x` is a function.

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

Vectorness 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 vectorness 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[1:3]
#> $coefficients
#> (Intercept)
#>           2
#>
#> $residuals
#>             1             2             3
#> -1.000000e+00 -3.885781e-16  1.000000e+00
#>
#> $effects
#> (Intercept)
#>  -3.4641016   0.3660254   1.3660254

# But not in vctrs
vctrs::vec_slice(fit, 1:3)
#> Error: Input 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: Input 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-fr`
- 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 <logical> and <my_natural>.

vec_ptype2(new_natural(1), 2:3)
#> Error: Can't combine <my_natural> and <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 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: Can't convert <my_natural> to <integer>.
vec_c(1.5, new_natural(1))
#> Error: Can't convert <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
```

`howto-faq-coercion-data-frame`

FAQ - How to implement ptype2 and cast methods? (Data frames)

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_type2()` 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
#> 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: 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

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::vec_assert(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: Input 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::vec_assert(my_df)

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

`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_of list_of *S3 class for homogenous lists*

Description

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

Usage

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

as_list_of(x, ...)

validate_list_of(x)

is_list_of(x)

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

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

Arguments

...	Vectors to coerce.
.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 getOption("vctrs.no_guess is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.
x	For as_list_of(), a vector to be coerced to list_of.
y, to	Arguments to vec_ptype2() and vec_cast().
x_arg	Argument names for x and y. These are used in error messages to inform the user about the locations of incompatible types (see stop_incompatible_type()).
y_arg	Argument names for x and y. 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))
```

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` *Assemble attributes for data frame construction*

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

`x`	A named list of equal-length vectors. The lengths are not checked; it is responsibility of the caller to make sure they are equal.
`n`	Number of rows. If `NULL`, will be computed from the length of the first element of `x`.
`..., class`	Additional arguments for creating subclasses. The `"names"` and `"row.names"` attributes override input in `x` and `n`, respectively: • `"names"` is used if provided, overriding existing names in `x` • `"row.names"` is used if provided, if `n` is provided it must be consistent.

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))
```

`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:

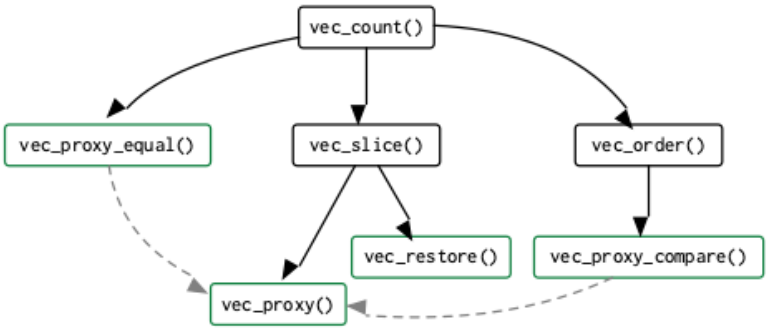
All vctrs operations are based on four primitive generics described in the next section. However there are many higher level operations. The most important ones implement fallbacks to base generics for maximum compatibility with existing classes.

- `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.

---

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-fr
- 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: 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: 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(0)
#> 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<127a2>> and <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.

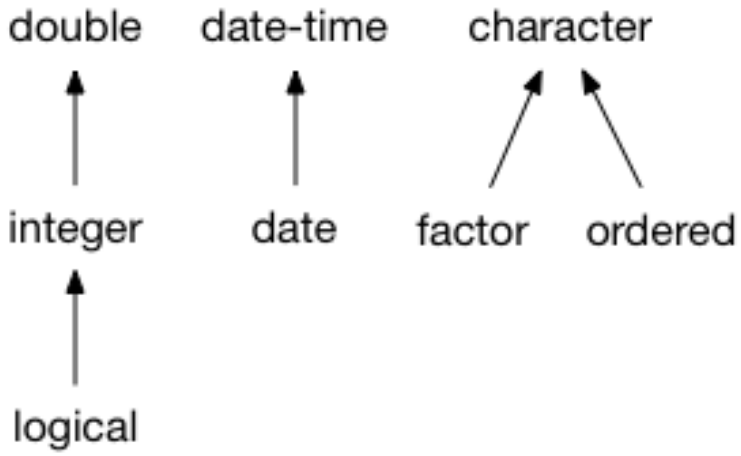
Note that another valid behaviour would be to throw an incompatible type error. This is what a strict factor implementation would do. We have decided to be laxer in vctrs because it is easy to inadvertently create factors instead of character vectors, especially with older versions of R where `stringsAsFactors` 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(0)
#> Levels: a c b

vec_ptype2(factor("b"), factor(c("a", "c")))
#> factor(0)
#> 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:
Coercible classes form a coercion (or subtyping) hierarchy. Here is a simplified diagram of the hierarchy for base types. 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).



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.

`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)

vec_rep_each(x, times)

vec_unrep(x)
```

Arguments

`x`	A vector.
`times`	For `vec_rep()`, a single integer for the number of times to repeat the entire vector. For `vec_rep_each()`, an integer vector of the number of times to repeat each element of `x`. `times` will be recycled to the size of `x`.

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_assert *Assert an argument has known prototype and/or size*

Description

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

Usage

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

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

Arguments

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

Value

vec_is() returns TRUE or FALSE. vec_assert() either throws a typed error (see section on error types) or returns x, invisibly.

Scalars and vectors

Informally, a vector is a collection that makes sense to use as column in a data frame. An object is a vector if one of the following conditions hold:

- A vec_proxy() method is implemented for the class of the object.
- The base type of the object is atomic: "logical", "integer", "double", "complex", "character", "raw"
- The object is a data.frame.
- The base type is "list", and one of:
  - The object is a bare "list" without a "class" attribute.
  - The object explicitly inherits from "list". That is, the "class" attribute contains "list" and inherits(x,"list") is TRUE.

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.
- 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 "class" attribute are not treated as vectors. If you inherit from an S3 class, always prepend your class to the "class" attribute for correct dispatch.

Error types

vec_is() never throws. vec_assert() throws the following errors:

- If the input is not a vector, an error of class "vctrs_error_scalar_type" is raised.
- If the prototype doesn't match, an error of class "vctrs_error_assert_ptype" is raised.
- If the size doesn't match, an error of class "vctrs_error_assert_size" is raised.

Both errors inherit from "vctrs_error_assert".

vec_as_names *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"),
  repair_arg = "",
  quiet = FALSE
)
```

Arguments

names	A character vector.
...	These dots are for future extensions and must be empty.
repair	Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", or "universal". If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown. • 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.
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.

`minimal` names

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

Examples:

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

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

`unique` names

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

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

Example:

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

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

`universal` names

`universal` names are `unique` and syntactic, meaning they:

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

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

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

Examples:

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

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

See Also

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

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

Examples

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

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

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

---

vec_bind *Combine many data frames into one data frame*

Description

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

Usage

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

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

Arguments

`...`	Data frames or vectors. When the inputs are named: • `vec_rbind()` assigns names to row names unless `.names_to` is supplied. In that case the names are assigned in the column defined by `.names_to`. • `vec_cbind()` 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 `.ptype`.

`.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 `getOption("vctrs.no_guess` is `TRUE` you must supply this value: this is a convenient way to make production code demand fixed types.

`.names_to` 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"`, or `"check_unique"`. 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.

`.size` If, `NULL`, the default, will determine the number of rows in `vec_cbind()` output by using the standard 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(x))`

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)
```

---

vec_c *Combine many vectors into one vector*

Description

Combine all arguments into a new vector of common type.

Usage

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

Arguments

...	Vectors to coerce.
.ptype	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 getOption("vctrs.no_guess 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 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.
.name_repair	How to repair names, see repair options in vec_as_names().

Value

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

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

Invariants

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

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 *Cast a vector to a specified type*

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 = "", to_arg = "")

vec_cast_common(..., .to = NULL)

## 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

x	Vectors to cast.
to, .to	Type to cast to. If NULL, x will be returned as is.
...	For vec_cast_common(), vectors to cast. For vec_cast(), vec_cast_default(), and vec_restore(), these dots are only for future extensions and should be empty.
x_arg, to_arg	Argument names for x and to. These are used in error messages to inform the user about the locations of incompatible types (see stop_incompatible_type()).

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-fr
- 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()`.
- `vec_unchop()` combines a list of vectors into a single vector, placing elements in the output according to the locations specified by `indices`. It is similar to `vec_c()`, but gives greater control over how the elements are combined. When no indices are supplied, it is identical to `vec_c()`.

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

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

Usage

```
vec_chop(x, indices = NULL)

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

Arguments

x	A vector
indices	For vec_chop(), a list of positive integer vectors to slice x with, or NULL. If NULL, x is split into its individual elements, equivalent to using an indices of as.list(vec_seq_along(x)). For vec_unchop(), a list of positive integer vectors specifying the locations to place elements of x in. Each element of x is recycled to the size of the corresponding index vector. The size of indices must match the size of x. If NULL, x is combined in the order it is provided in, which is equivalent to using vec_c().
ptype	If NULL, the default, the output type is determined by computing the common type across all elements of x. Alternatively, you can supply ptype 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 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.
name_repair	How to repair names, see repair options in vec_as_names().

Value

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

Dependencies of vec_chop()

- vec_slice()

Dependencies of vec_unchop()

- vec_c()

Examples

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

# If `indices` selects every value in `x` exactly once,
```

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

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

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

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

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

ave2(breaks, wool, mean)

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

`vec_compare` *Compare two vectors*

Description

Compare two vectors

Usage

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

Arguments

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

Value

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

S3 dispatch

vec_compare() is not generic for performance; instead it uses vec_proxy_compare() to

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

x	A vector (including a data frame).
sort	One of "count", "key", "location", or "none". • "count", the default, puts most frequent values at top • "key", orders by the output key column (i.e. unique values of x) • "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.

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_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 `NA`s 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 *Test if two vectors are equal*

Description

`vec_equal_na()` tests a special case: equality with `NA`. It is similar to is.na but:

- Considers the missing element of a list to be `NULL`.
- Considered data frames and records to be missing if every component is missing. This preserves the invariant that `vec_equal_na(x)` is equal to `vec_equal(x,vec_init(x),na_equal = TRUE)`.

Usage

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

vec_equal_na(x)
```

Arguments

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

Value

A logical vector the same size as. Will only contain NAs if na_equal is FALSE.

Dependencies

- vec_cast_common() with fallback
- vec_recycle_common()
- vec_proxy_equal()

Examples

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

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

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

---

vec_fill_missing *Fill in missing values with the previous or following value*

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

x	A vector
direction	Direction in which to fill missing values. Must be either "down", "up", "downup", or "updown".
max_fill	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_identify_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.

Usage

```
vec_identify_runs(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

An integer vector with the same size as x. A scalar integer attribute, n, is attached.

Examples

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

vec_identify_runs(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_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)
vec_init(mtcars, 2)
```

vec_is_list *Is the object a list?*

Description

`vec_is_list()` tests if `x` is considered a list in the vctrs sense. It returns `TRUE` if:

- `x` is a bare list with no class.
- `x` is a list explicitly inheriting from `"list"`.

Usage

```
vec_is_list(x)
```

Arguments

`x`	An object.

Details

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

Examples

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

vec_is_list(data.frame())
```

vec_match *Find matching observations across vectors*

Description

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

Usage

```
vec_match(
  needles,
  haystack,
  ...,
  na_equal = TRUE,
  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 vec_match() will only return the location of the first match.

  needles and haystack are coerced to the same type prior to comparison.

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

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

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"),
  quiet = FALSE
)

vec_names(x)

vec_set_names(x, names)
```

Arguments

`x`	A vector with names
`...`	These dots are for future extensions and must be empty.
`repair`	Either a string or a function. If a string, it must be one of `"check_unique"`, `"minimal"`, `"unique"`, or `"universal"`. If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown. • 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.
`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.
`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
`direction`	Direction to sort in. Defaults to ascending.
`na_value`	Should `NA`s 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`.

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, "desc")

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

`vec_ptype` *Find the prototype of a set of vectors*

Description

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

Usage

```
vec_ptype(x, ..., x_arg = "")

vec_ptype_common(..., .ptype = NULL)

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.
`.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 `getOption("vctrs.no_guess` is `TRUE` you must supply this value: this is a convenient way to make production code demand fixed types.

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 `NA`s to represent missing values for any 1d vector type.

See internal-faq-ptype2-identity for more information about identity values.

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

`vec_ptype_common()`

`vec_ptype_common()` first finds the prototype of each input, then successively calls `vec_ptype2()` to find a common type. It returns a finalised prototype.

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 = "", y_arg = "")
```

Arguments

`x, y`	Vector types.
`...`	These dots are for future extensions and must be empty.
`x_arg, y_arg`	Argument names for `x` and `y`. These are used in error messages to inform the user about the locations of incompatible types (see `stop_incompatible_type()`).

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-fr
- 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_recycle *Vector recycling*

Description

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

Usage

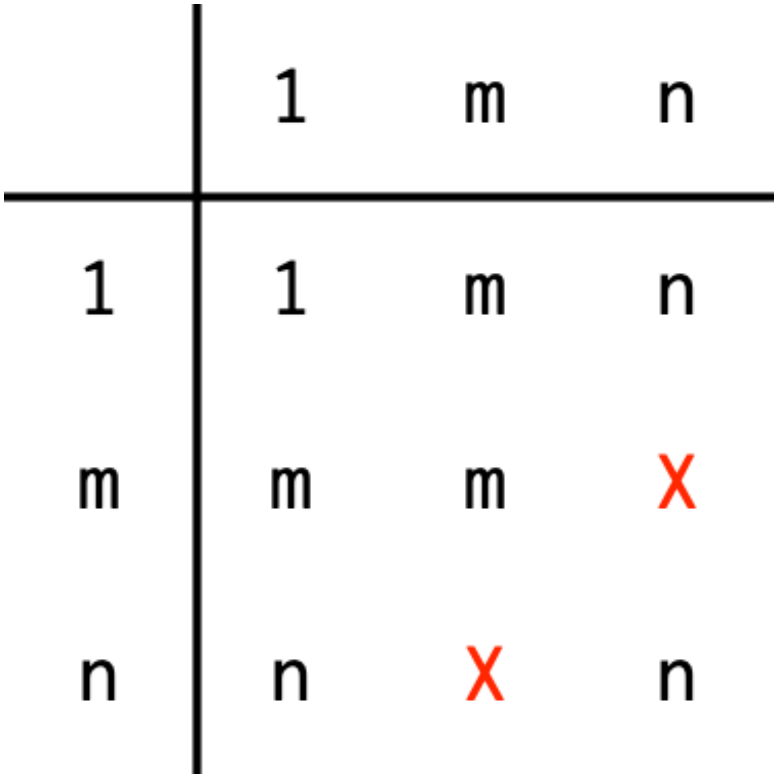
```
vec_recycle(x, size, ..., x_arg = "")

vec_recycle_common(..., .size = NULL)
```

Arguments

x	A vector to recycle.
size	Desired output size.
...	• For vec_recycle_common(), vectors to recycle. – For vec_recycle(), 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.
.size	Desired output size. If omitted, will use the common size from vec_size_common().

Recycling rules

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



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

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

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

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 *Number of observations*

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 `vec_is_list()`. `list_sizes()` is to `vec_size()` as `lengths()` is to `length()`.

Usage

```
vec_size(x)

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

list_sizes(x)

vec_is_empty(x)
```

Arguments

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

Details

There is no vctrs helper that retrieves the number of columns: as this is a property of the type.

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

Value

An integer (or double for long vectors).

vec_size_common() returns .absent if all inputs are NULL or absent, 0L by default.

Invariants

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

The size of NULL

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

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

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

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