Package 'cppally'

November 6, 2026

Title A 'C++20' API for R

Version 1.2.1

Maintainer Nick Christofides <nick.christofides.r@gmail.com>

Description A header-only 'C++20' API for manipulating R data structures from 'C++'. Provides 'C++20' concepts specific to R, custom scalar and vector classes with built-in NA handling, automatic object protection, 'SIMD' (single-instruction-multiple-data), parallelisation, and a streamlined system for registering 'C++' functions, including templates, to R. Full API reference and documentation are available at <https://nicchr.github.io/cppally/>.

License MIT + file LICENSE

URL https://nicchr.github.io/cppally/

BugReports https://github.com/NicChr/cppally/issues/

Depends R (>= 4.5.0)

Suggests bench, bit64, brio, callr, cli, cpp11, decor, desc, devtools, fs, ggplot2, glue, knitr, pkgbuild, pkgload, purrr, readr, rmarkdown, roxygen2, rstudioapi, stringr, testthat (>= 3.0.0), tidyr, usethis, vctrs, withr

VignetteBuilder knitr

Config/Needs/cppally/cpp_register brio, cli, decor, desc, glue, purrr, readr, stringr, vctrs, withr

Config/testthat/edition 3

Encoding UTF-8

SystemRequirements C++20

Config/roxygen2/version 8.1.1

Config/Needs/website rmarkdown

NeedsCompilation no

Author Nick Christofides [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-9743-7342>), Martin Leitner-Ankerl [cph] (Author of bundled ankerl::unordered_dense

library),
Malte Skarupke [cph] (Author of ska_sort, from which to_radix_key in sort.h is adapted),
Ryo Suzuki [cph] (Author of bundled Xoshiro-cpp library),
Posit Software, PBC [cph] (SEXP protection mechanism in r_sexp.h inspired by cpp11)

Repository CRAN

Date/Publication 2026-11-06 09:50:08 UTC

Contents

cpp_register 2
cpp_source 3
document 9
load_all 9
use_check_data_frames 10
use_check_factors 11
use_copy_on_modify 11
use_cppally 12
use_openmp 12
use_preserve_altrep_flag 13
use_template_dispatch_candidates 13

Index **15**

`cpp_register` *Generates wrappers for registered C++ functions*

Description

Register C++ functions to be callable from R. C++ functions decorated with `[[cppally::register]]` will be registered (including template functions).

Usage

```
cpp_register(
  path = ".",
  quiet = !is_interactive(),
  extension = c(".cpp", ".cc"),
  cppally_header = c("cppally.hpp", "cppally_light.hpp")
)
```

Arguments

path	Path to package root directory.
quiet	If TRUE suppresses output from this function.
extension	The file extension to use for the generated src/cppally file. Options are either '.cpp' (the default) or '.cc'.
cppally_header	Which header should be included with the registered C++ code? The default is the full library "cppally.hpp". Choose "cppally_light.hpp" for the lighter header, which may provide quicker compile times, at the cost of less features.

Value

The paths to the generated R and C++ source files.

See Also

cpp_source

---

cpp_source *Compile C++20 code*

Description

cpp11-style helpers to compile cppally code outside of a cppally-linked package context.

`cpp_source()` compiles and loads a single C++ file for use in R, either from an expression or a cpp file. This may include multiple C++ functions.

`cpp_eval()` evaluates a single C++ expression and returns the result. For example `cpp_eval('get_threads()')` will run the C++ function `cppally::get_threads()` and return the number of OMP threads currently set for use. For expressions that do not return a result, the call is evaluated and `NULL` is returned invisibly.

Usage

```
cpp_source(
  file = NULL,
  code = NULL,
  env = parent.frame(),
  clean = TRUE,
  quiet = TRUE,
  debug = FALSE,
  preserve_altrep = FALSE,
  check_factors = FALSE,
  check_data_frames = FALSE,
  copy_on_modify = FALSE,
  openmp = TRUE,
  cxx_std = Sys.getenv("CXX_STD", "CXX20"),
```

```
  dir = tempfile(),
  ...
)

cpp_eval(
  code,
  env = curr_env(),
  clean = TRUE,
  quiet = TRUE,
  debug = FALSE,
  preserve_altrep = FALSE,
  check_factors = FALSE,
  check_data_frames = FALSE,
  copy_on_modify = FALSE,
  openmp = TRUE,
  simplify = TRUE,
  cxx_std = Sys.getenv("CXX_STD", "CXX20"),
  cppally_header = c("cppally.hpp", "cppally_light.hpp"),
  ...
)
```

Arguments

file	C++ file.
code	For cpp_source() - If file is NULL then a string of C++ code to compile. This can include the contents of a cpp file which can contain multiple [[cppally::register]] tagged functions. For cpp_eval - This can be a character vector of single-line expressions.
env	Environment where R functions should be defined.
clean	Should files be cleaned up after sourcing? Default is TRUE. Only newly created directories are removed, existing ones are left as-is.
quiet	Should compiler output be suppressed? Default is TRUE.
debug	Should C++ code be compiled in a debug build? Default is FALSE.
preserve_altrep	Should ALTREP vectors be preserved by avoiding materialisation where possible? Default is FALSE.
check_factors	Should factor levels be validated when using r_factors objects? Default is FALSE. When TRUE, factor levels are checked once on r_factors construction to ensure they are valid, reducing the chance of R crashing when passing factors with invalid levels.
check_data_frames	Should data frames be validated when constructing r_df objects from SEXP? Default is FALSE.
copy_on_modify	Should copy-on-modify be used everywhere? Default is FALSE.
openmp	Should code be compiled with OpenMP flags? Default is TRUE.
cxx_std	C++ standard to use. Should be >= C++20.

dir	Directory to store the source files. The default is a temporary directory via tempfile() which is removed when clean = TRUE.
...	Further arguments passed to use_template_dispatch_candidates().
simplify	Applies to cpp_eval. A list of results is returned unless length(code) == 1 and simplify = TRUE.
cppally_header	Which header should be included with the registered C++ code? The default is the full library "cppally.hpp". Choose "cppally_light.hpp" for the lighter header, which may provide quicker compile times, at the cost of less features.

Value

cpp_source() invisibly compiles the C++ code and registers the [[cppally::register]] tagged functions to R.
cpp_eval() returns the results of the evaluated C++ expressions.

See Also

cpp_register use_template_dispatch_candidates

Examples

```
library(cppally)
library(bit64)

cpp_eval <- function(...){
  # We don't need the full cppally header for these examples
  cppally::cpp_eval(..., cppally_header = "cppally_light.hpp")
}


cpp_eval('print("hello world!")')

# Default values of all cppally scalars
cpp_eval(c(
  'r_lgl()',
  'r_int()',
  'r_dbl()',
  'r_int64()',
  'r_str()',
  'r_raw()',
  'r_cplx()',
  'r_date()',
  'r_psxct()'
))

cpp_source(code = '
  #include <cppally.hpp>
  using namespace cppally;

  [[cppally::register]]
  r_dbl add(r_dbl x, r_dbl y){
```

```
    return x + y;
  }
', debug = TRUE)
add(1, 2)
add(2, NA)

### ALTREP

# cppally also supports lazy ALTREP materialisation as an opt-in feature.
# To opt-in, set `preserve_altrep = TRUE`

cpp_source(
  code = '
  #include <cppally_light.hpp>
  using namespace cppally;

  [[cppally::register]]
  r_int last_altrep_unaware(r_vec<r_int> x){
    r_int out;
    r_size_t n = x.length();

    if (n > 0){
      out = x.get(n - 1);
    }
    return out;
  }
', debug = TRUE
)

cpp_source(
  code = '
  #include <cppally_light.hpp>
  using namespace cppally;

  [[cppally::register]]
  r_int last_altrep_aware(r_vec<r_int> x){
    r_int out;
    r_size_t n = x.length();

    if (n > 0){
      out = x.get(n - 1);
    }
    return out;
  }
', debug = TRUE,
  preserve_altrep = TRUE
)

library(bench)
mark(last_altrep_aware(1:10^5)) # No materialisation
mark(last_altrep_unaware(1:10^5)) # Materialises full vector

### Copy-on-modify
```

```
# cppally supports copy-on-modify as an opt-in feature
# It is disabled by default because it incurs a major performance penalty
# and has been deemed not worth it even for the safety benefits
# That being said, if you prefer absolute safety over speed then you can
# enable it globally via `cppally::use_copy_on_modify()` or
# via the arg `copy_on_modify` if  using `cpp_source()`

cpp_source(
  code = '
  #include <cppally_light.hpp>
  using namespace cppally;

  [[cppally::register]]
  r_vec<r_int> reverse(r_vec<r_int> x){
    x.rev(); // in-place reverse
    return x;
  }
', copy_on_modify = TRUE
)

x <- c(1L, 2L, 3L)
reverse(x)
x # x was preserved and not updated by reference (as expected)

x <- sample.int(10^5)
mark(reverse(x)) # Memory allocated, therefore x was copied before reversing

# The cppally preferred approach is to allocate a fresh vector or copy the
# existing vector
cpp_source(
  code = '
  #include <cppally_light.hpp>
  using namespace cppally;

  [[cppally::register]]
  r_vec<r_int> cppally_reverse(r_vec<r_int> x){
    r_vec<r_int> out = x.copy();
    out.rev();
    return out;
  }
', copy_on_modify = FALSE
)

mark(
  r_reverse = rev(x),
  cppally_copy_on_modify_reverse = reverse(x),
  cppally_no_copy_on_modify_reverse = cppally_reverse(x)
)

### Speeding up template-heavy compilation

# When writing C++ code that only ever uses a small subset of cppally types,
```

```
# we can restrict cppally template dispatch to only ever consider these types,
# potentially speeding up compilation times. Only do this if you are certain
# that these are the only viable types your code will ever reasonably accept.
# If you are unsure, then do not use this feature and instead use
# C++ concepts and template constraints.

# Example: restrict template dispatch on `r_int` and `r_dbl`
# `cppally::unique()` is templated on RComposite, which accepts all R vectors,
# factors, and data frames. Therefore writing our own template which depends
# on `unique()` being available for any generic type, might get expensive if say we
# are only interested in integers and doubles.

mark(

  unrestricted = cpp_source(
    code = '
    #include <cppally.hpp>
    using namespace cppally;

    template <RVector T>
    requires (requires (T obj){ unique(obj); })
    [[cppally::register]]
    T sorted_unique(T x){
      return unique(x, /*sort = */ true );
    }
  ',
    debug = TRUE
  ),
  restricted = cpp_source(
    code = '
    #include <cppally.hpp>
    using namespace cppally;

    template <RVector T>
    requires (requires (T obj){ unique(obj); })
    [[cppally::register]]
    T sorted_unique2(T x){
      return unique(x, /*sort = */ true );
    }
  ',
    debug = TRUE,
    scalar_types = c("r_int", "r_dbl"),
    r_sexp = FALSE,
    data_frames = FALSE,
    factors = FALSE
  ),
  check = FALSE,
  memory = FALSE,
  iterations = 1
)

sorted_unique(c(1, 1, 2, 2, 3, 3))
sorted_unique2(c(1, 1, 2, 2, 3, 3))
```

```
sorted_unique(c("A", "A", "B", "B", "C", "C"))

# Expected to fail
try(sorted_unique2(c("A", "A", "B", "B", "C", "C")))

# In production code and regular usage,
# you would call `use_template_dispatch_candidates(c("r_int", "r_dbl"))`
# which will add the relevant flag to your Makevars file(s).

rm(cpp_eval)
```

`document` *A wrapper around* `devtools::document()` *to support cppally package development*

Description

A wrapper around `devtools::document()` to support cppally package development

Usage

```
document(pkg = ".", roclets = NULL, quiet = FALSE)
```

Arguments

`pkg`	See `?devtools::document`
`roclets`	See `?devtools::document`
`quiet`	See `?devtools::document`

Value

Invisibly updates roxygen documentation, compiles C++ code and exports cppally tagged functions to R.

`load_all` *A wrapper around* `devtools::load_all()` *specifically for cppally*

Description

A wrapper around `devtools::load_all()` specifically for cppally

Usage

```
load_all(
  path = ".",
  debug = FALSE,
  cppally_header = c("cppally.hpp", "cppally_light.hpp"),
  compile = NA,
  ...
)
```

Arguments

path	Path to package.
debug	Should package be built without optimisations? Default is FALSE which builds with optimisations.
cppally_header	Which header should be included with the registered C++ code? The default is the full library "cppally.hpp". Choose "cppally_light.hpp" for the lighter header, which may provide quicker compile times, at the cost of less features.
compile	See ?pkgload::load_all.
...	Further arguments passed on to pkgload::load_all()

Value

Invisibly registers cppally tagged functions and compiles C++ code.

---

use_check_data_frames *Deprecated.*

Description

Deprecated, do not use.

Usage

```
use_check_data_frames(quiet = FALSE)
```

Arguments

quiet	[logical(1)] - Should messages be suppressed? Default is FALSE.

Value

Invisibly adds the CPPALLY_CHECK_DATA_FRAMES flag to Makevars.

use_check_factors *Adds the* CPPALLY_CHECK_FACTORS *flag to Makevars*

Description

Adds a flag to Makevars which enables stricter validation on factor levels at the point of `r_factors` construction. This avoids creating `r_factors` objects with invalid levels and avoiding potential R crashes.

The default behaviour is NOT to validate factor levels, which is naturally faster when calling C++ functions that take `r_factors` inputs.

Usage

```
use_check_factors(quiet = FALSE)
```

Arguments

`quiet`	`[logical(1)]` - Should messages be suppressed? Default is `FALSE`.

Value

Invisibly adds the `CPPALLY_CHECK_FACTORS` flag to Makevars.

use_copy_on_modify *Adds the* CPPALLY_COPY_ON_MODIFY *flag to Makevars*

Description

Adds a flag to Makevars which enables copy-on-modify behaviour like in R. This ensures you can never modify object B's data by modifying object A (unless you do it manually via `r_vec::data()`).

The default behaviour is that `r_vec::set()` always modifies-in-place with no checks to shared or referenced objects. This default behaviour is generally much faster.

Parallelisation:

Adding this global flag effectively kills most parallelisation. To achieve copy-on-modify we have to check whether the `r_sexp` object being modified is referenced anywhere else at each and every call to `set()`. This check (via `r_sexp::is_exclusive()`) is not thread-safe and therefore most cppally parallelisation is disabled once CPPALLY_COPY_ON_MODIFY is set.

Usage

```
use_copy_on_modify(quiet = FALSE)
```

Arguments

`quiet`	`[logical(1)]` - Should messages be suppressed? Default is `FALSE`.

Value

Invisibly adds the CPPALLY_COPY_ON_MODIFY flag to Makevars.

---

use_cppally *Helper for developing packages with cppally*

Description

usethis style helper to add the necessary setup to a new package to help users get started with writing C++ code.

Usage

```
use_cppally(quiet = FALSE)
```

Arguments

quiet	[logical(1)] - Should messages be suppressed? Default is FALSE.

Value

Invisibly sets up the necessary conditions for developing a package with cppally.

---

use_openmp *Add OpenMP flags to Makevars*

Description

Adds the correct flags to Makevars, enabling SIMD vectorisation and multi-threading via OpenMP.

Usage

```
use_openmp(quiet = FALSE)
```

Arguments

quiet	[logical(1)] - Should messages be suppressed? Default is FALSE.

Value

Invisibly adds the '$(SHLIB_OPENMP_CXXFLAGS)' value to the Makevars variables 'PKG_LIBS' and 'PKG_CXXFLAGS', enabling multi-threading and SIMD vectorisation via OpenMP.

use_preserve_altrep_flag

Adds the CPPALLY_PRESERVE_ALTREP flag to Makevars

Description

Adds a flag to Makevars which enables lazy materialisation of ALTREP vectors.

Usage

```
use_preserve_altrep_flag(quiet = FALSE)
```

Arguments

quiet	[logical(1)] - Should messages be suppressed? Default is FALSE.

Value

Invisibly adds the CPPALLY_PRESERVE_ALTREP flag to Makevars.

use_template_dispatch_candidates

Restrict the R/C++ types a template function will dispatch on

Description

Use this when:

- You want to speed up compilation time.
- Your code only ever will accept a subset of types.

For example, let's say you are writing a C++ matrix algebra library using cppally and you are only interested in working with integers and doubles. Because the only types your library accepts are based on these two types, you could restrict all template accepted types to `r_int` and `r_dbl` (and equivalently `r_vector<r_int>` and `r_vector<r_dbl>`). Doing this will dramatically improve development time by reducing compile times and also reducing the size of your package dll.

Because cppally is a header-only library, templates you call from your registered templated functions, including cppally's own, are instantiated once per surviving candidate rather than once per default candidate.

Usage

```
use_template_dispatch_candidates(
  scalar_types = cppally_scalar_types,
  scalars = TRUE,
  vectors = TRUE,
  factors = TRUE,
  data_frames = TRUE,
  r_sexp = TRUE,
  quiet = FALSE
)
```

Arguments

scalar_types	[character(n)] - Template candidate scalar types to allow package-wide. Vector candidates are derived from this, e.g. r_dbl allows both scalar r_dbl and vector r_vector<r_dbl>. The default includes all cppally scalar types.
scalars	[logical(1)] - Should bare scalar candidates be included? Default is TRUE. Scalars share a type mapping with their vector counterpart. Setting this to FALSE prevents scalars from being valid template arguments, but the vector equivalents will still be allowed.
vectors	[logical(1)] - Should r_vector<> candidates be included? Default is TRUE.
factors	[logical(1)] - Should r_factors be a candidate? Default is TRUE.
data_frames	[logical(1)] - Should r_df be a candidate? Default is TRUE.
r_sexp	[logical(1)] - Should r_sexp be a candidate? Default is TRUE. r_sexp is not a scalar type, so it has its own switch rather than belonging to scalar_types.
quiet	[logical(1)] - Should messages be suppressed? Default is FALSE.

Details

Adds flags to Makevars restricting the set of R types the dispatcher considers when calling a registered templated function.

The dispatch table holds N^k entries for k template parameters, so the saving grows sharply with the number of template parameters on a function.

Calling it with no arguments restores the defaults.

This changes behaviour, not just build time:

A type left out of the candidate set is no longer accepted at the R boundary. Dropping r_date does not make a Date argument slower, it makes it an error. The rejection is explicit and happens before dispatch, so an excluded type is never quietly claimed by the catch-all and reinterpreted as something else. The same source therefore behaves differently depending on these flags, so narrow the set only to types the package genuinely intends to support.

This does not affect the r_sexp visitor functions defined in r_visit.h. Those dispatch over their own type list and continue to handle every R type regardless of what is set here.

Value

Invisibly adds dispatch candidate flags to Makevars.

Index

cpp_eval *(*cpp_source*)*, 3
cpp_register, 2, *5*
cpp_source, *3*, 3

document, 9

load_all, 9

use_check_data_frames, 10
use_check_factors, 11
use_copy_on_modify, 11
use_cppally, 12
use_openmp, 12
use_preserve_altrep_flag, 13
use_template_dispatch_candidates, *5*, 13