tidyCpp Motivation

Dirk Eddelbuettel[1]

[1]Department of Statistics, University of Illinois, Urbana-Champaign, IL, USA

Initial (draft) version dated September 2020

The `tidyCpp` package is introduced with two motivating examples contrasting code using the standard C API for R alongside with code using `tidyCpp` to provide shorter and, to our eyes, cleaner and more readable code. In one word: tidier.

Introducing tidyCpp. The `tidyCpp` package offers a simple, small and clean C++ layer over the C API offered by R. This vignette highlights a few usage examples, often taken from the *Writing R Extensions* vignette that comes with R, to highlight some features.

`tidyCpp` has no further dependencies on any other package. It can however be used with Rcpp simply to take advantage of its helper functions `cppFunction()` or `sourceCpp()`.

`tidyCpp` is still a fairly young and small package. Please free to contribute by make suggestions, or sending bugfixes or extension proposals.

Snippet One: dimnames. This example comes from Writing R Extension, Section 5.9.4 which highlights attribute setting from the C API.

It takes two (named) numeric vectors, computes the outer product matrix and uses the names to set row- and column names. Note that we modified the existing example ever so slight by ensuring (as is frequently done) remapping of symbols. For example, `length` (which can clash easily with existing symbols in the global namespace) is now `Rf_length`. We also added an `export` tag for `Rcpp` simply to facilitate integration into R. No Rcpp header or data structures are used; we simply rely on its logic in getting C or C++ source into R.

Using the C API for R

```
#define R_NO_REMAP
#include <R.h>
#include <Rinternals.h>

// [[Rcpp::export]]
SEXP out(SEXP x, SEXP y)
{
  int nx = Rf_length(x), ny = Rf_length(y);
  SEXP ans = PROTECT(Rf_allocMatrix(REALSXP,
                                    nx, ny));
  double *rx = REAL(x),
    *ry = REAL(y),
    *rans = REAL(ans);

  for(int i = 0; i < nx; i++) {
    double tmp = rx[i];
    for(int j = 0; j < ny; j++)
      rans[i + nx*j] = tmp * ry[j];
  }

  SEXP dimnames = PROTECT(Rf_allocVector(VECSXP,2));
  SET_VECTOR_ELT(dimnames, 0,
                 Rf_getAttrib(x, R_NamesSymbol));
  SET_VECTOR_ELT(dimnames, 1,
                 Rf_getAttrib(y, R_NamesSymbol));
  Rf_setAttrib(ans, R_DimNamesSymbol, dimnames);

  UNPROTECT(2);
  return ans;
}
```

Using tidyCpp

```
#include <tidyCpp>
// [[Rcpp::depends(tidyCpp)]]


// [[Rcpp::export]]
SEXP out(SEXP x, SEXP y)
{
  int nx = R::length(x), ny = R::length(y);
  SEXP ans = R::Shield(R::allocMatrixReal(nx, ny));

  double *rx = R::numericPointer(x),
    *ry = R::numericPointer(y),
    *rans = R::numericPointer(ans);

  for(int i = 0; i < nx; i++) {
    double tmp = rx[i];
    for(int j = 0; j < ny; j++)
      rans[i + nx*j] = tmp * ry[j];
  }

  SEXP dimnames = R::Shield(R::allocVectorList(2));
  R::setVectorElement(dimnames, 0, R::getNames(x));

  R::setVectorElement(dimnames, 1, R::getNames(y));

  R::setDimNames(ans, dimnames);

  return ans;
}
```

Some key differences:

- a single header `tidyCpp`: simple and clean;
- no `PROTECT` and `UNPROTECT` with manual calling of the number of calls made: C++ takes care of that for us via `Shield` which we borrowed (in a simplified form) from Rcpp;
- no `Rf_*` calls: everything used comes from a clean new namespace `R` and is easily identified;

- types are made explicit in the name of the called function sequence rather than enum;
- consistent naming that aligns with language convention:
  - *types* such as `Shield` are capitalized, and
  - *verbs* such as the allocators or converters are camelCase;
- overall less wordy and shorter, *e.g.*, `R::getNames(x)` instead of `Rf_getAttrib(x, R_NamesSymbol)`;
- no macros are being used.

Note that the use of `Rcpp::export` does not imply use of Rcpp data structures. We simply take advantaged of the tried and true code generation to make it easy to call the example from R. You can copy either example into a temporary file and use `Rcpp::sourceCpp("filenameHere")` on it to run the example.

Snippet Two: convolution. This example comes from Writing R Extension, Section 5.10.1 which introduces the `.Call()` interface of the C API for R.

It takes two numeric vectors and computes a convolution. Note that as above we modified the existing example ever so slight by ensuring (as is frequently done) remapping of symbols, once again added an `export` tag for Rcpp simply to facilitate integration into R, and changing whitespace. No Rcpp header or data structures are used; we simply rely on its logic in getting C or C++ source into R.

Using the C API for R

```
#define R_NO_REMAP
#include <R.h>
#include <Rinternals.h>

// [[Rcpp::export]]
SEXP convolve2(SEXP a, SEXP b)
{
  int na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  a = PROTECT(Rf_coerceVector(a, REALSXP));
  b = PROTECT(Rf_coerceVector(b, REALSXP));
  na = Rf_length(a);
  nb = Rf_length(b);
  nab = na + nb - 1;
  ab = PROTECT(Rf_allocVector(REALSXP, nab));
  xa = REAL(a);
  xb = REAL(b);
  xab = REAL(ab);
  for(int i = 0; i < nab; i++)
    xab[i] = 0.0;
  for(int i = 0; i < na; i++)
    for(int j = 0; j < nb; j++)
      xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return ab;
}
```

Using tidyCpp

```
#include <tidyCpp>
// [[Rcpp::depends(tidyCpp)]]


// [[Rcpp::export]]
SEXP convolve2(SEXP a, SEXP b)
{
  int na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  a = R::Shield(R::coerceVectorNumeric(a));
  b = R::Shield(R::coerceVectorNumeric(b));
  na = R::length(a);
  nb = R::length(b);
  nab = na + nb - 1;
  ab = R::Shield(R::allocVectorNumeric(nab));
  xa = R::numericPointer(a);
  xb = R::numericPointer(b);
  xab = R::numericPointer(ab);
  for(int i = 0; i < nab; i++)
    xab[i] = 0.0;
  for(int i = 0; i < na; i++)
    for(int j = 0; j < nb; j++)
      xab[i + j] += xa[i] * xb[j];
  return ab;
}
```

Like the previous example, the new version operates without macros, does not require manual counting in `PROTECT` and `UNPROTECT` and is, to our eyes, a little more readable.

Discussion. `tidyCpp` provides a cleaner layer on top of the C API for R. That has its advantages: we find it more readable. It conceivably has possible disadvantages. Those familiar with the C API for R may not need this, and may find it an unnecessary new dialect. Time will tell if new adoption and use may outway possible hesitation by other. In the meantime, the package "does no harm", has no further dependencies and can be used, or dropped, at will

Summary. The `tidyCpp` package provides a cleaner simplifying layer on top of the time-tested but somewhat crusty C API for R.