# Thirteen Simple Steps for Creating An R Package with an External C++ Library

**Dirk Eddelbuettel**[1]

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

This version was compiled on September 12, 2024

**We desribe how we extend R with an external C++ code library by using the Rcpp package. Our working example uses the recent machine learning library and application 'Corels' providing optimal yet easily interpretable rule lists (Angelino *et al.*, 2017) which we bring to R in the form of the RcppCorels package (Eddelbuettel, 2019). We discuss each step in the process, and derive a set of simple rules and recommendations which are illustrated with the concrete example.**

## Introduction

The process of building a new package with Rcpp can range from the very easy—a single simple C++ function—to the very complex. If, and how, external resources are utilised makes a big difference as this too can range from the very simple—making use of a header-only library, or directly including a few C++ source files without further dependencies—to the very complex.

Yet a lot of the important action happens in the middle ground. Packages may bring their own source code, but also depend on just one or two external libraries. This paper describes one such approach in detail: how we turned the Corels application (Angelino *et al.*, 2017; Laurus-Stone, 2019) (provided as a standalone C++-based executable) into an R-callable package **RcppCorels** (Eddelbuettel, 2019) via **Rcpp** (Eddelbuettel *et al.*, 2024; Eddelbuettel and François, 2011).

## The Thirteen Key Steps

**Ensure Use of a Suitable license.** Before embarking on such a journey, it is best to ensure that the licensing framework is suitable. Many different open-source licenses exists, yet a few key ones dominate and can generally be used *with each other*. There is however a fair amount of possible legalese involved, so it is useful to check inter-license compatibility, as well as general usability of the license in question. Several sites can help via license recommendations, and checks for interoperability. One example is the site at choosealicense.com (which is backed by GitHub) can help, as can tldrlegal.com. License choice is a complex topic, and general recommendations are difficult to make besides the key point of sticking to already-established and known licenses.

**Ensure the Software builds.** In order to see how hard it may to combine an external entity, either a program a library, with R, it helps to ensure that the external entity actually still builds and runs.

This may seem like a small and obvious steps, but experience suggests that it worth asserting the ability to build with current tools, and possibly also with more than one compiler or build-system. Consideration to other platforms used by R also matter a great deal as one of the strengths of the R package system is its ability to cover the three key operating system families.

**Ensure it still works.** This may seem like a variation on the previous point, but besides the ability to *build* we also need to ensure the ability to *run* the software. If the external entity has tests and demo, it is highly recommended to run them. If there are reference results, we should ensure that they are still obtained, and also that the run-time performance it still (at a minimum) reasonable.

**Ensure it is compelling.** This is of course a very basic litmus test: is the new software relevant? Is is helpful? Would others benefit from having it packaged and maintained?

**Start an Rcpp package.** The first step in getting a new package combing R and C++ is often the creation of a new Rcpp package. There are several helper functions to choose from. A natural first choice is `Rcpp.package.skeleton()` from the **Rcpp** package (Eddelbuettel *et al.*, 2024). It can be improved by having the optional helper package **pkgKitten** (Eddelbuettel, 2023) around as its `kitten()` function smoothes some rougher edges left by the underlying Base R function `package.skeleton()`. This step is shown below in then appendix, and corresponds to the first commit, followed by a first edit of file `DESCRIPTION`.

Any code added by the helper functions, often just a simple `helloWorld()` variant, can be run to ensure that the package is indeed functional. More importantly, at this stage, we can also start building the package as a compressed tar archive and run the R checker on it.

**Integrate External Package.** Given a basic package with C++ support, we can now turn to integrating the external package. This complexity of this step can, as alluded to earlier, vary from very easy to very complex. Simple cases include just dependending on library headers which can either be copied to the package, or be provided by another package such as **BH** (Eddelbuettel *et al.*, 2023). It may also be a dependency on a fairly standard library available on most if not all systems. The graphics formats bmp, jpeg or png may be example; text formats like JSON or XML are another. One difficulty, though, may be that *run-time* support does not always guarantee *compile-time* support. In these cases, a `-dev` or `-devel` package may need to be installed.

In the concrete case of Corels, we

- copied all existing C++ source and header files over into the `src/` directory;
- renamed all header files from `*.hh` to `*.h` to comply with an R preference;
- create a minimal `src/Makevars` file, initially with link instructions for GMP later relaxed to conditional use of GMP (see below);
- moved `main.cc` to a subdirectory as we cannot build with another `main()` function (and R will not include files from subdirectories);
- added a minimal R-callable function along with a `logger` instance.

Here, the last step was needed as the file `main.cc` provided a global instance referred to from other files. Hence, a minimal R-callable wrapper is being added at this stage (shown in the appendix as well). Actual functionality will be added later.

We will come back to the step concerning the link instructions.

As this point we have a package for R also containing the library we want to add.

**Make the External Code compliant with R Policies.** R has fairly strict guidelines, defined both in the *CRAN Repository Policy* document at the CRAN website, and in the manual *Writing R Extension*. Certain standard C and C++ functions are not permitted as their use could interfere with running code from R. This includes somewhat obvious recommendations ("do not call `abort`" as it would terminate the R sessions) but extends to not using native print methods in order to cooperate better with the input and output facilities of R. So here, and reflecting that last aspect, we changed all calls to `printf()` to calls to `Rprintf()`. Similarly, R prefers its own (well-tested) random-number generators so we replaced one (scaled) call to `random() / RAND_MAX` with the equivalent call to R's `unif_rand()`. We also avoided one use of `stdout` in `rulelib.h`.

The requirement for such changes may seem excessive at first, but the value added stemming from consistent application of the CRAN Policies is appreciated by most R users.

**Complete the Interface.** In order to further test the package, and of course also for actual use, we need to expose the key parameters and arguments. Corels parsed command-line arguments; we can translate this directly into suitable arguments for the main function. At a first pass, we created the following interface:

```
// [[Rcpp::export]]
bool corels(std::string rules_file,
            std::string labels_file,
            std::string log_dir,
            std::string meta_file = "",
            bool run_bfs = false,
            bool calculate_size = false,
            bool run_curiosity = false,
            int curiosity_policy = 0,
            bool latex_out = false,
            int map_type = 0,
            int verbosity = 0,
            int max_num_nodes = 100000,
            double regularization = 0.01,
            int logging_frequency = 1000,
            int ablation = 0) {

  // actual function body omitted
}
```

Rcpp facilities the integration by adding another wrapper exposing all the function arguments, and setting up required arguments without default (the first three) along with optional arguments given a default. The user can now call `corels()` from R with three required arguments (the two input files plus the log directory) as well as number of optional arguments.

**Add Sample Data.** R package can access data files that are shipped with them. That is very useful feature, and we therefore also copy in the files include in the Corels repository and its `data/` directory.

```
fs::dir_tree("../../../rcppcorels/inst/sample_data")
#  ../../../rcppcorels/inst/sample_data
#  +-- compas_test-binary.csv
#  +-- compas_test.csv
#  +-- compas_test.label
#  +-- compas_test.out
#  +-- compas_train-binary.csv
#  +-- compas_train.csv
#  +-- compas_train.label
#  +-- compas_train.minor
#  \-- compas_train.out
```

**Set up working example.** Combining the two preceding steps, we can now offer an illustrative example. It is included in the helpd page for function `corels()` and can be run from R via `example("corels")`.

```
library(RcppCorels)

.sysfile <- function(f)    # helper function
  system.file("sample_data",f,package="RcppCorels")

rules_file <- .sysfile("compas_train.out")
label_file <- .sysfile("compas_train.label")
meta_file <- .sysfile("compas_train.minor")
logdir <- tempdir()

stopifnot(file.exists(rules_file),
          file.exists(labels_file),
          file.exists(meta_file),
          dir.exists(logdir))

corels(rules_file, labels_file, logdir, meta_file,
       verbosity = 100,
       regularization = 0.015,
       curiosity_policy = 2,    # by lower bound
       map_type = 1)            # permutation map

cat("See ", logdir, " for result file.")
```

In the example, we pass the two required arguments for rules and labels files, the optional argument for the 'meta' file as well as an added required argument for the output directory. R policy prohibits writing in user-directories, we default to using the temporary directory of the current session, and report its value at the end. For other arguments default values are used.

**Finesse Library Dependencies.** One common difficulty when bringing an external library to R via a package consists in dealing with an external dependency. In the case of 'Corels', the GNU GMP library for multi-precision arithmetic is an optional extension which, if available, improves and accelerates internal processing.

The simplest approach is to declare a compile-time variable in the `src/Makevars` file. Using `-DGMP` *defines* the `GMP` variable at the level of the C and C++ code. One can then condition on the variable. A very standard approach, also used here is `#if defined(GMP) ... #else ... #endif` where one of the two code branches is in effect depending on whether the `GMP` variable is defined or not.

In order to detect presence of a required (or optional) library, tools like 'autoconf' or 'cmake' are often used. For example, one

can rely of an existing 'autoconf' macro provided by the GMP documentation to detect presence of the the GNU GMP header and library. We are making use of this facility here to deploy GMP when it is available. As 'Corels' can be built with and without GMP, the build and installation succeeds either way—but deployment of the more-featureful variant with use GMP is automated.

**Finalise License and Copyright.** It is good (and common) practice to clearly attribute authorship. Here, credit is given to the 'Corels' team and authors as well as to the authors of the underlying 'rulelib' code used by 'Corels' via the file `inst/AUTHORS` (which will be installed as `AUTHORS` with the package. In addition, the file `inst/LICENSE` clarifies the GNU GPL-3 license for 'RcppCorels' and 'Corels', and the MIT license for 'rulelib'.

**Additional Bonus: Some more 'meta' files.** Several files help to improve the package. For example, `.Rbuildignore` allows to exclude listed files from the resulting R package keeping it well-defined. Similarly, `.gitignore` can exclude files from being added to the `git` repository. We also like `.editorconfig` for consistent editing default across a range of modern editors.

## Summary

We describe s series of steps to turn the standalone library 'Corels' describes by Angelino *et al.* (2017) into a R package **RcppCorels** using the facilities offered by **Rcpp** (Eddelbuettel *et al.*, 2024). Along the way, we illustrate key aspects of the R package standards and CRAN Repository Policy proving a template for other research software wishing to provide their implementations in a form that is accessibly by R users.

## References


Angelino E, Larus-Stone N, Alabi D, Seltzer M, Rudin C (2017). "Learning Certifiably Optimal Rule Lists for Categorical Data." arXiv:1704.01701.

Eddelbuettel D (2019). "RcppCorels: R binding for the 'Certifiably Optimal RulE ListS (Corels)' Learner." https://github.com/eddelbuettel/rcppcorels.

Eddelbuettel D (2023). *pkgKitten: Create Simple Packages Which Do not Upset R Package Checks*. R package version 0.2.3, URL https://CRAN.R-Project.org/package=pkgKitten.

Eddelbuettel D, Emerson JW, Kane MJ (2023). *BH: Boost C++ Header Files*. R package version 1.81.0-1, URL https://CRAN.R-Project.org/package=BH.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08. URL https://doi.org/10.18637/jss.v040.i08.

Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russel N, Chambers J, Bates D (2024). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.13, URL https://CRAN.R-Project.org/package=Rcpp.

Laurus-Stone N (2019). "corels: Learning Certifiably Optimal Rule Lists." https://github.com/nlarusstone/corels. Also online at https://corels.eecs.harvard.edu/corels/.

**Appendix 1: Creating the basic package.**

```
edd@rob:~/git$ r --packages Rcpp --eval 'Rcpp.package.skeleton("RcppCorels")'

Attaching package: ‘utils’

The following objects are masked from ‘package:Rcpp’:

    .DollarNames, prompt

Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './RcppCorels/Read-and-delete-me'.

Adding Rcpp settings
 >> added Imports: Rcpp
 >> added LinkingTo: Rcpp
 >> added useDynLib directive to NAMESPACE
 >> added importFrom(Rcpp, evalCpp) directive to NAMESPACE
 >> added example src file using Rcpp attributes
 >> added Rd file for rcpp_hello_world
 >> compiled Rcpp attributes
edd@rob:~/git$
edd@rob:~/git$ mv RcppCorels/ rcppcorels  # prefer lowercase directories
edd@rob:~/git$
```

**Appendix 2: A Minimal src/Makevars.** In the file shown here, use of GMP is unconditional: we define `GMP` as a compiler flag, and instruct the linker to link with the GMP library.

```
CXX_STD = CXX11

PKG_CXXFLAGS = -I. -DGMP

PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS) -lgmp
```

**Appendix 3: A Placeholder Wrapper.**

```
#include "queue.h"

#include <Rcpp.h>

/*
 * Logs statistics about the execution of the algorithm and dumps it to a file.
 * To turn off, pass verbosity <= 1
 */
NullLogger* logger;

// [[Rcpp::export]]
bool corels() {
  return true;                  // more to fill in, naturally
}
```