Rcpp Attributes

J.J. Allaire Dirk Eddelbuettel Romain François

Rcpp version 0.10.0.4 as of November 26, 2012

Abstract

Rcpp attributes provide a high-level syntax for declaring C++ functions as callable from R and automatically generating the code required to invoke them. Attributes are intended to facilitate both interactive use of C++ within R sessions as well as to support R package development. Attributes are built on top of **Rcpp** modules and their implementation is based on previous work in the **inline** package (Sklyar, Murdoch, Smith, Eddelbuettel, and François, 2012).

1 Introduction

Attributes are a new feature of **Rcpp** version 0.10.0 (Eddelbuettel and François, 2012, 2011) that provide infrastructure for seamless language bindings between R and C++. The motivation for attributes is several-fold:

1. Reduce the learning curve associated with using C++ and R together
2. Eliminate boilerplate conversion and marshaling code wherever possible
3. Seamless use of C++ within interactive R sessions
4. Unified syntax for interactive work and package development

The core concept is to add annotations to C++ source files that provide the context required to automatically generate R bindings to C++ functions. Attributes and their supporting functions include:

- `Rcpp::export` attribute to export a C++ function to R
- `sourceCpp` function to source exported functions from a file
- `cppFunction` and `evalCpp` functions for inline declarations and execution
- `Rcpp::depends` attribute for specifying additional build dependencies for `sourceCpp`

Attributes can also be used for package development via the `compileAttributes` function, which generates an **Rcpp** module for all exported functions within a package.

2 Using Attributes

Attributes are annotations that are added to C++ source files to provide additional information to the compiler. **Rcpp** supports attributes to indicate that C++ functions should be made available as R functions, as well as to optionally specify additional build dependencies for source files.

C++11 specifies a standard syntax for attributes (Maurer and Wong, 2008). Since this standard isn't yet fully supported across all compilers, **Rcpp** attributes are included in source files using specially formatted comments.

2.1 Exporting C++ Functions

The `sourceCpp` function parses a C++ file and looks for functions marked with the `Rcpp::export` attribute. A shared library is then built and its exported functions are made available as R functions in the specified environment. For example, this source file contains an implementation of convolve (note the `Rcpp::export` attribute in the comment above the function):

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector convolveCpp(NumericVector a, NumericVector b) {

    int na = a.size(), nb = b.size();
    int nab = na + nb - 1;
    NumericVector xab(nab);

    for (int i = 0; i < na; i++)
        for (int j = 0; j < nb; j++)
            xab[i + j] += a[i] * b[j];

    return xab;
}
```

The addition of the export attribute allows us to do this from the R prompt:

```
> sourceCpp("convolve.cpp")
> convolveCpp(x, y)
```

We can now write C++ functions using built-in C++ types and **Rcpp** wrapper types and then source them just as we would an R script.

The `sourceCpp` function performs caching based on the last modified date of the source file so as long as the source file does not change the compilation will occur only once per R session.

2.2 Specifying Argument Defaults

If default argument values are provided in the C++ function definition then these defaults are also used for the exported R function. For example, the following C++ function:

```
DataFrame readData(
     CharacterVector file,
     CharacterVector cols = CharacterVector::create(),
     std::string comment = "#",
     bool header = true)
```

Will be exported to R as:

```
function (file, cols = character(0), comment = "#", header = TRUE)
```

Note that C++ rules for default arguments still apply: they must occur consecutively at the end of the function signature and (unlike R) can't rely on the values of other arguments.

2.3 Signaling Errors

Within R code the `stop` function is typically used to signal errors. Within R extensions written in C the `Rf_error` function is typically used. However, within C++ code you cannot safely use `Rf_error` because it results in a `longjmp` over any C++ destructors on the stack.

The correct way to signal errors within C++ functions is to throw an `Rcpp::exception`. For example:

```
if (unexpectedCondition)
    throw Rcpp::exception("Unexpected condition occurred");
```

There is also an `Rcpp::stop` function that is shorthand for throwing an `Rcpp::exception`. For example:

```
if (unexpectedCondition)
    Rcpp::stop("Unexpected condition occurred");
```

In both cases the C++ exception will be caught by **Rcpp** prior to returning control to R and converted into the correct signal to R that execution should stop with the specified message.

2.4 Embedding R Code

Typically C++ and R code are kept in their own source files. However, it's often convenient to bundle code from both languages into a common source file that can be executed using single call to `sourceCpp`.

To embed chunks of R code within a C++ source file you include the R code within a block comment that has the prefix of `/*** R`. For example:

```
/*** R

# Call the fibonacci function defined in C++
fibonacci(10)

*/
```

Multiple R code chunks can be included in a C++ file. The `sourceCpp` function will first compile the C++ code into a shared library and then source the embedded R code.

2.5 Modifying Function Names

You can change the name of an exported function as it appears to R by adding a name parameter to `Rcpp::export`. For example:

```
// [[Rcpp::export(".convolveCpp")]]
NumericVector convolveCpp(NumericVector a, NumericVector b)
```

Note that in this case since the specified name is prefaced by a `.` the exported R function will be hidden.

2.6 Function Requirements

Functions marked with the `Rcpp::export` attribute must meet several requirements to be correctly handled:

- Be defined in the global namespace (i.e. not within a C++ namespace declaration)
- Have a return type that is either void or compatible with `Rcpp::wrap` and parameter types that are compatible with `Rcpp::as` (see sections 3.1 and 3.2 of the '*Rcpp-introduction*' vignette for more details).
- Use fully qualified type names for the return value and all parameters. Rcpp types may however appear without a namespace qualifier (i.e. `DataFrame` is okay as a type name but `std::string` must be specified fully).

2.7 Importing Dependencies

It's also possible to use the `Rcpp::depends` attribute to declare dependencies on other packages. For example:

```
// [[Rcpp::depends(RcppArmadillo)]]

#include <RcppArmadillo.h>
using namespace Rcpp

// [[Rcpp::export]]
List fastLm(NumericVector yr, NumericMatrix Xr) {

    int n = Xr.nrow(), k = Xr.ncol();

    arma::mat X(Xr.begin(), n, k, false);
    arma::colvec y(yr.begin(), yr.size(), false);

    arma::colvec coef = arma::solve(X, y);
    arma::colvec resid = y - X*coef;

    double sig2 = arma::as_scalar(arma::trans(resid)*resid/(n-k));
    arma::colvec stderrest = arma::sqrt(
          sig2 * arma::diagvec( arma::inv(arma::trans(X)*X)) );

    return List::create(Named("coefficients") = coef,
                        Named("stderr")       = stderrest);
}
```

The inclusion of the `Rcpp::depends` attribute causes `sourceCpp` to configure the build environment to correctly compile and link against the **RcppArmadillo** package. Source files can declare more than one dependency either by using multiple `Rcpp::depends` attributes or with syntax like this:

```
// [[Rcpp::depends(Matrix, RcppArmadillo)]]
```

Dependencies are discovered both by scanning for package include directories and by invoking **inline** plugins if they are available for a package.

2.8 Including C++ Inline

Maintaining C++ code in it's own source file provides several benefits including the ability to use C++ aware text-editing tools and straightforward mapping of compilation

errors to lines in the source file. However, it's also possible to do inline declaration and execution of C++ code.

There are several ways to accomplish this, including passing a code string to `sourceCpp` or using the shorter-form `cppFunction` or `evalCpp` functions. For example:

```
> cppFunction('
    int fibonacci(const int x) {
        if (x < 2)
            return x;
        else
            return (fibonacci(x - 1)) + fibonacci(x - 2);
    }
')

> evalCpp('std::numeric_limits<double>::max()')
```

You can also specify a depends parameter to `cppFunction` or `evalCpp`:

```
> cppFunction(depends = 'RcppArmadillo', code = '...')
```

3 Package Development

One of the goals of **Rcpp** attributes is to simultaneously facilitate ad-hoc and interactive work with C++ while also making it very easy to migrate that work into an R package. There are several benefits of moving code from a standalone C++ source file to a package:

1. Your code can be made available to users without C++ development tools (at least on Windows or Mac OS X where binary packages are common)
2. Multiple source files and their dependencies are handled automatically by the R package build system
3. Packages provide additional infrastructure for testing, documentation and consistency

3.1 Package Creation

To create a package that is based on **Rcpp** you should follow the guidelines in the '*Rcpp-package*' vignette. For a new package this is most conveniently done using the `Rcpp.package.skeleton` function.

To generate a new package with a simple hello, world function that uses attributes you can do the following:

```
> Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

To generate a package based on C++ files that you've been using with `sourceCpp` you can use the `cpp_files` parameter:

```
> Rcpp.package.skeleton("NewPackage", example_code = FALSE,
                        cpp_files = c("convolve.cpp"))
```

3.2 Exporting R Functions

Within interactive sessions you call the `sourceCpp` function on individual files to export C++ functions into the global environment. However, for packages you call a single utility function to export all C++ functions within the package.

The `compileAttributes` function scans the source files within a package for export attributes and generates code as required. For example, executing this from within the package working directory:

```
> compileAttributes()
```

Results in the generation of the following two source files:

- `src/RcppExports.cpp` – An **Rcpp** module that exports the functions
- `R/RcppExports.R` – The R code required to load the **Rcpp** module

You should re-run `compileAttributes` whenever functions are added, removed, or have their signatures changed.

The `compileAttributes` function deals only with exporting C++ functions to R. If you want the functions to additionally be publicly available from your package's namespace another step may be required. Specifically, if your package `NAMESPACE` file does not use a pattern to export functions then you should add an explicit entry to `NAMESPACE` for each R function you want publicly available.

3.3 Specifying Dependencies

Once you've migrated C++ code into a package, the dependencies for source files are derived from the `Depends` and `LinkingTo` fields in the package `DESCRIPTION` file rather than the `Rcpp::depends` attribute. For every package you import C++ code from (including **Rcpp**) you need to add these entries.

For example, if your package depends on **Rcpp** and **RcppArmadillo** you would have the following in your `DESCRIPTION` file:

```
Depends: Rcpp (>= 0.10.0), RcppArmadillo (>= 0.3.4.4)
LinkingTo: Rcpp, RcppArmadillo
```

3.4 Roxygen Comments

The **roxygen2** package (Wickham, Danenberg, and Eugster, 2011) provides a facility for automatically generating R documentation files based on specially formatted comments in R source code.

If you include roxygen comments in your C++ source file with a `//'` prefix then `compileAttributes` will transpose them into R roxygen comments within `R/RcppExports.R`. For example the following code in a C++ source file:

```
//' The length of a string (in characters).
//' @param str input character vector
//' @return characters in each element of the vector
// [[Rcpp::export]]
NumericVector strLength(CharacterVector str)
```

Results in the following code in the generated R source file:

```
#' The length of a string (in characters).
#' @param str input character vector
#' @return characters in each element of the vector
strLength <- function(str)
```

Two notes about the R functions written to `RcppExports.R` for use with roxygen:

1. The generated R functions have an empty body. This is because they are only present for binding to roxygen comments. A call to `Rcpp::loadModule` will replace the empty definitions with the appropriate C++ functions.
2. The functions do not include argument defaults since they are not known at the time of generation. If you have argument defaults that you'd like included in the Rd usage section you can do this by adding an explicit `@usage` roxygen tag.

3.5 Providing a C++ Interface

The interface exposed from R packages is most typically a set of R functions. However, the R package system also provides a mechanism to allow the exporting of C and C++ interfaces using package header files. This is based on the `R_RegisterCCallable` and `R_GetCCallable` functions described in '*Writing R Extensions*' (R Development Core Team, 2012).

C++ interfaces to a package are published within the top level `include` directory of the package (which within the package source directory is located at `inst/include`). The R build system automatically adds the required `include` directories for all packages specified in the `LinkingTo` field of the package `DESCRIPTION` file.

3.5.1 Interfaces Attribute

The `Rcpp::interfaces` attribute can be used to automatically generate a header-only interface to your C++ functions within the `include` directory of your package.

The `Rcpp::interfaces` attribute is specified on a per-source file basis, and indicates which interfaces (R, C++, or both) should be provided for exported functions within the file.

For example, the following specifies that both R and C++ interfaces should be generated for a source file:

```
// [[Rcpp::interfaces(r, cpp)]]
```

Note that the default behavior if an `Rcpp::interfaces` attribute is not included in a source file is to generate an R interface only.

3.5.2 Generated Code

If you request a `cpp` interface for a source file then `compileAttributes` generates the following header files (substituting *Package* with the name of the package code is being generated for):

```
inst/include/Package.h
inst/include/Package_RcppExports.h
```

The `Package_RcppExports.h` file has inline definitions for all exported C++ functions that enable calling them using the `R_GetCCallable` mechanism.

The `Package.h` file does nothing other than include the `Package_RcppExports.h` header. This is done so that package authors can replace the `Package.h` header with a custom one and still be able to include the automatically generated exports (details on doing this are provided in the next section).

The exported functions are defined within a C++ namespace that matches the name of the package. For example, an exported C++ function `bar` could be called from package `MyPackage` as follows:

```
// [[Rcpp::depends(MyPackage)]]

#include <MyPackage.h>

void foo() {
    MyPackage::bar();
}
```

3.5.3 Including Additional Code

You might wish to use the `Rcpp::interfaces` attribute to generate a part of your package's C++ interface but also provide additional custom C++ code. In this case

you should replace the generated `Package.h` file with one of your own.

Note that the way **Rcpp** distinguishes user verses generated files is by checking for the presence a special token in the file (if it's present then it's known to be generated and thus safe to overwrite). You'll see this token at the top of the generated `Package.h` file, be sure to remove it if you want to provide a custom header.

Once you've established a custom package header file, you need only include the `Package_RcppExports.h` file within your header to make available the automatically generated code alongside your own.

If you need to include code from your custom header files within the compilation of your package source files, you will also need to add the following entry to `Makevars` and `Makevars.win` (both are in the `src` directory of your package):

```
PKG_CPPFLAGS += -I../inst/include/
```

Note that the R package build system does not automatically force a rebuild when headers in `inst/include` change, so you should be sure to perform a full rebuild of the package after making changes to these headers.

3.5.4 Binary Compatibility

An additional consideration related to exposing C++ types from a package is binary compatibility. If a user of your package compiles against one version of the package and then runs against another version the binary layout of C++ types used must remain stable in order for things to work correctly. Note that this is an issue that affects all C++ interfaces exposed by shared libraries and is not specific to R packages or **Rcpp**. A summary of the issues presented can be found in '*Binary Compatibility Issues with C++*' (KDE-TechBase, 2012).

Note that if your package is used within a C++ source file compiled with `sourceCpp` then binary compatibility is not a concern (because the compilation is always synchronized with the currently installed version of your package). Also, packages published on `CRAN` are automatically rebuilt when a package they depend on via `LinkingTo` is updated, so users with updated versions of all packages will similarly not have binary compatibility problems.

However, even when installing from `CRAN` it's possible for users to have the latest version of one package but not of another. If you want to minimize binary compatibility problems even in the face of out-of-sync package versions then it's strongly recommended that you use only use built-in C++ types and **Rcpp** wrapper types in your interfaces.

References

Dirk Eddelbuettel and Romain François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. URL http://www.jstatsoft.org/v40/i08/.

Dirk Eddelbuettel and Romain François. *Rcpp: Seamless R and C++ Integration*, 2012. URL http://CRAN.R-Project.org/package=Rcpp. R package version 0.9.13.

KDE-TechBase. Binary compatibility issues with C++. http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++, 2012. [Online; accessed 24-November-2012].

Jens Maurer and Michael Wong. Towards support for attributes in C++ (revision 6). In *JTC1/SC22/WG21 - The C++ Standards Committee*, 2008. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf. N2761=08-0271.

R Development Core Team. *Writing R extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL http://CRAN.R-Project.org/doc/manuals/R-exts.html. ISBN 3-900051-11-9.

Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel, and Romain François. *inline: Inline C, C++, Fortran function calls from R*, 2012. URL http://CRAN.R-Project.org/package=inline. R package version 0.3.10.

Hadley Wickham, Peter Danenberg, and Manuel Eugster. *roxygen2: In-source documentation for R*, 2011. URL http://CRAN.R-Project.org/package=roxygen2. R package version 2.2.12.