

## Chapter 2

# Tools and Setup

**Abstract** Chapter 1 provided a gentle introduction to **Rcpp** and some of its key features. In this chapter, we look more closely at the required toolchain of compilers and related **R** packages needed to deploy the **Rcpp** package. In particular, on Windows, the **Rtools** collection is used and non-gcc compilers are not supported. On Unix-alike systems such as Linux and OS X, gcc/g++ is the default.

### 2.1 Overall Setup

The **Rcpp** package provides a C++ Application Programming Interface (API) as an extension to the **R** system. Because of these very close ties to **R** itself, it is both bound by the choices made by the **R** build system and influenced by how **R** is configured.

Some of the requirements for working with **Rcpp** and **R** are:

- The development environment has to comprise a suitable compiler (which is discussed more in the next section), as well as header files and libraries for a number of required components (R Development Core Team 2012a).
- **R** should be built in a way that permits both dynamic linking and embedding; on Unix-alike systems this is typically ensured by the `--enable-shared-lib` option to `configure` (R Development Core Team 2012d, Chapter 8) and most binary distributions of **R** are built this way.
- Common development tools such as `make` are needed which should be standard on Unix-alike systems (though OS X requires installation of developer tools) whereas Windows users will have to install the **Rtools** suite provided via the CRAN mirror network (R Development Core Team 2012a, Appendix D).

In general, the standard environment for building a CRAN package from source is required. The (even stronger) requirement of being able to build **R** itself is a possible guideline as is documented in R Development Core Team (2012a,d).

There are a few additional CRAN packages that are very useful along with **Rcpp**, and which the package itself depends upon. These are:

**inline** which is invaluable for direct compilation, linking and loading of short code snippets, and used throughout this book too.

**rbenchmark** which is used to run simple timing comparisons and benchmarks; it is also recommended by **Rcpp** but not required.

**RUnit** which is used for unit testing; the package is recommended and will only be needed to rerun these tests but it is not strictly required.

We already saw two of these packages in use in the preceding chapter.

Lastly, users who want to build **Rcpp** from the repository source (rather than the distributed tarfile) also need the **highlight** binary by André Simon which is used to provide colored source code in several of the vignettes.

## 2.2 Compilers

### 2.2.1 General Setup

A basic requirement for extending a program with *suitable* loadable binary modules relates to the compiler being used. But exactly what is *suitable* can depend on a number of factors.

The choice of compilers generally matters, and more so for some languages than for others. The C language has a simpler interface for callable functions which makes it possible to have a program compiled with one compiler load a module built with another compiler. In general, this is not an option for C++ due to a much more complicated interface reflecting some of the richer structures in the C++ language. For example, how function names, and member function names inside classes, are represented is not standardized between compiler makers, and this generally prevents mixing of object code between different compilers.

As **Rcpp** is of course a C++ application, this last restriction applies and we need to stick with the compilers used to build R on the different platforms. The CRAN repository generally employs the same approach of using one main compiler per platform, and this approach is the one supported by the CRAN maintainers and the R Core team.

In practice, this means that on almost all platforms, the GNU Compiler Collection (or `gcc`, which is also the name of its C language compiler) has to be used along with the corresponding `g++` compiler for the C++ language. One notable exception is Solaris where the Sun compiler can be used as well; however, this platform is not as widely available and used, and we will not discuss its particular aspects any further. Also, on Windows, the prescribed way to access the suitable compiler is via the `Rtools` package contributed by the Windows R maintainers (R Development Core Team 2012a, Appendix D). OS X is an exception as Apple will not ship `gcc`

versions past 4.2.1. Its transition to the `clang++` compiler of the LLVM project is not yet complete as this book is being written. Users on the OS X platform may have to download tools provided by Simon Urbanek, the R Core maintainer supporting OS X.

So on Windows, OS X and Linux, the compiler of choice for **Rcpp** generally is the `g++` compiler. A minimum suitable version is a final 4.2.\* release; releases earlier than 4.2.\* were lacking some C++ features used by **Rcpp**. Later versions are preferred as version 4.2.1 has some known bugs. But generally speaking, as of 2013 the (current) default compilers on all the common platforms are suitable. As of R version 2.12.0, the Windows platform has switched to version 4.5.1 of `g++` in order to support both 32- and 64-bit builds.

More advanced C++ features from the next C++ standard, C++11, which has recently been approved by the standards committee will become available once the compilers support them by default.

### 2.2.2 Platform-Specific Notes

#### Windows

Windows is both the most common platform for R use—yet quite possibly the hardest to develop on. The reason for this difficulty with R development on Windows is that the build environment and tools do not come standard with the operating system. However, due to the popularity of the platform, good support exists in the form of a third-party package kindly provided by some of the R Core developers who focus on Windows, namely Brian Ripley and Duncan Murdoch. The `Rtools` package, initially distributed via a site maintained by Duncan Murdoch but now available via the CRAN network, contains all the required tools in a single package. Complete instructions specific to Windows are available in the “R Administration” manual (R Development Core Team 2012a, Appendix D).

To stress again what was hinted at above: other compilers are not supported on Windows. In particular, the popular series of compilers produced by Microsoft cannot be used to build R from source (for reasons that are beyond the scope of this discussion) as these compilers are simply not supported by R Core. While it may be possible to compile some C++ extensions for R using these compilers, the **Rcpp** package follows the recommendation of the R Core team and sticks to the officially supported compilers. So for the last few years, and presumably for the next few years too, this limits the choice on Windows to the version of `g++` in the **Rtools** bundle.

## OS X

OS X has become a popular choice of operating system among developers. As noted in the “R Administration” manual (R Development Core Team 2012a, Appendix C.4), the Apple Developer Tools (e.g., Xcode) have to be installed (as well as `gfortran` if R or Fortran-using packages are to be built). Some older versions of OS X do not have a C++ compiler that is recent enough for some of the template code in the **Rcpp**; releases starting from “Snow Leopard” should be sufficient.

Unfortunately, Apple and the Free Software Foundation (the organization backing all the GNU software) are at an impasse over licensing. The GNU Compiler Collection now uses version 3 of GNU General Public License which Apple deems unsuitable for its operating system. As became clear in 2011, it seems that `g++` version 4.2.1 will be the last version available from Apple, which is unfortunate as more current `g++` releases have made great strides towards adding new features of the upcoming C++ language standard. However, the `clang++` compiler from the LLVM should eventually provide a full-featured replacement.

## Linux

On Linux, developers need to install the standard development packages. Some distributions provide helper packages which pull in all the required packages; the `r-base-dev` package on Debian and Ubuntu is an example.

In general, whatever tools are needed to build R itself will be sufficient to build **Rcpp** from source, and to build packages utilizing **Rcpp**.

## Other Platforms

Few other platforms appear to be in widespread use. The CRAN archive runs regression tests against Solaris and its Sun compiler. However, as we do not have direct access to the platform, development and debugging of **Rcpp** is somewhat cumbersome on this platform. Moreover, we have not yet detected measurable interest among the population of possible users. That said, **Rcpp** plugs into general R facilities for building packages, and the clear intent is to have **Rcpp** install and work on every platform supported by R itself.

## 2.3 The R Application Programming Interface

The R language and environment supports an application programming interface, or API for short. The API is described in the “Writing R Extensions” manual (R Development Core Team 2012d), and defined in the header files provided with every R installation. The R Core group usually stresses that only the public API should be used as other (undocumented) functions could change without notice.

Several books describe the API and its use. Venables and Ripley (2000) is an important early source. Gentleman (2009) and Matloff (2011) are more recent additions, while Chambers (2008) is authoritative in the context of “Programming with Data.”

There are two fundamental extension functions provided: `.C()` and `.Call()`. The first, `.C()` first appeared in an earlier version of the R language and is much more restrictive. It only supports pointers to basic C types which is a very severe restriction. More current code uses the richer `.Call()` interface exclusively. It can operate on the so-called SEXP objects, which stands for pointers to S expression objects. Essentially everything inside R is represented as such a SEXP object, and by permitting exchange of such objects between the C and C++ languages on the one hand, and R on the other hand, programmers have the ability to operate directly on R objects. This is key for **Rcpp** as well—and the principal reason why **Rcpp** works exclusively with `.Call()`.

**Rcpp** essentially sits on top of this API offered by R itself and provides a complementary interface to those aiming to extend R. By leveraging facilities available to C++ programmers (but not in plain C), **Rcpp** can offer what we think is an easier to use and possibly even more consistent interface that is closer to the way R programmers work with their data.

## 2.4 A First Compilation with Rcpp

Having discussed the required compiler and toolkit setup, and having seen introductory examples in Chap. 1, it is now appropriate to address how to use these tools on an actual source file. In doing so, we will use the explicit commands to illustrate the different steps required. Shorter and more convenient alternatives will be discussed later.

We consider the first example from the introductory chapter and assume that both the fibonacci function and the wrapper have been saved in a file `fibonacci.cpp`. Then, on a 64-bit Linux computer with **Rcpp** installed in a standard location, we can compile it via the example shown in Listing 2.1.

```

sh> PKG_CXXFLAGS="-I/usr/local/lib/R/site-library/Rcpp/include" \
2  PKG_LIBS="-L/usr/local/lib/R/site-library/Rcpp/lib -lRcpp" \
  R CMD SHLIB fibonacci.cpp
4 g++ -I/usr/share/R/include -DNDEBUG \
  -I/usr/local/lib/R/site-library/Rcpp/include \
6  -fpic -g -O3 -Wall -c fibonacci.cpp -o fibonacci.o
g++ -shared -o fibonacci.so fibonacci.o \
8  -L/usr/local/lib/R/site-library/Rcpp/lib -lRcpp
  -Wl,-rpath,/usr/local/lib/R/site-library/Rcpp/lib \
10 -L/usr/lib/R/lib -lR

```

**Listing 2.1** A first manual compilation with **Rcpp**

Execution of `R CMD SHLIB` triggers two distinct `g++` invocations. The first command (on line four) corresponds to `R CMD COMPILE` to turn a given source file into an object file. The second command (on line eight) corresponds to `R CMD LINK` and uses the `g++` compiler a second time to link the object file into a shared library. This creates the file `fibonacci.so` which we can load into `R`. Also note how two environment variables are defined on lines 1 and 2 to let `R` know where to find the header files and libraries required for use with **Rcpp**.

But before we get to that step, let us review a few of the issues with the approach described here:

1. On line one, we have to set two environment variables, one each for the header file location (via `PKG_CXXFLAGS`) and one for the library location and name (via `PKG_LIBS`).
2. Both these variables use explicit path settings which are not portable across computers, let alone operating systems.
3. File extensions are operating-system dependent, the shared library ends on `.so` on Linux but `.dylib` under OS X.

To address some of these concerns, **Rcpp** offers two helper functions which can be invoked using the scripting front-end `Rscript`.

---

```
sh> PKG_CXXFLAGS='Rscript -e 'Rcpp::CxxFlags()' '\
2   PKG_LIBS='Rscript -e 'Rcpp::LdFlags()' '\
   R CMD SHLIB fibonacci.cpp
```

---

**Listing 2.2** A first manual compilation with **Rcpp** using `Rscript`

Running the example in Listing 2.2 results in the same two commands as above. But this approach improves over the previous one:

1. By using commands to request information which is returned in a portable fashion freeing the user from having to specify these details.
2. The helper functions are part of the **Rcpp** package and can therefore impute the relevant locations in a portable manner.
3. Moreover, the helper functions also know the operating system details and therefore are able to supply the required per-operating system details such as file extensions.

The end result is that we have a single command that works across platforms,<sup>1</sup> including portably in a `Makefile`. So we can now use this file in `R`.

---

```
1 R> dyn.load("fibonacci.so")
R> .Call("fibWrapper", 10)
3 [1] 55
```

---

**Listing 2.3** Using the first manual compilation from `R`

---

<sup>1</sup> Well, Windows user may have to set the two environment variables differently but that is a shell limitation in Windows and not an issue with **Rcpp**.

We can load the shared library via the `dyn.load()` function. It uses the full filename, including the explicit platform-dependent extension which is `.so` on Unix, `.dll` on Windows, and `.dylib` on OS X. Once the shared library is loaded into the R session, we can then call the function `fibWrapper` using the standard `.Call()` interface. We supply the argument `n` to compute the corresponding Fibonacci number and obtain the requested result.

So this example proves the point we were trying to make in this section: we can extend R with simple C++ functions, even though the process of doing so may seem somewhat involved and intimidating at first. The **inline** package discussed in the next section and the **Rcpp** attributes extension discussed in the following section make the build process a lot more seamless to use.

## 2.5 The Inline Package

We saw in the previous chapter how to compile, link, and load a new function for use by R. We will now look more closely at a tool first mentioned in that introductory chapter which greatly simplifies this process.

### 2.5.1 Overview

Extending R with compiled code requires a mechanism for reliably compiling, linking, and loading the code. Doing this in the context of a package is preferable in the long run, but it may well be too involved for quick explorations. Undertaking the compilation manually is certainly possible. But, as the previous section showed, also somewhat laborious.

A better alternative is provided by the **inline** package (Sklyar et al. 2012) which compiles, links, and loads a C, C++, or Fortran function—directly from the R prompt using simple functions `cfunction` and `cxxfunction`. The latter provides an extension which works particularly well with **Rcpp** via the so-called plugins which provide information about additional header file and library locations; and a third function, `rcpp`, which defaults to selecting that plugin for use with **Rcpp**.

The use of **inline** is possible as **Rcpp** itself can be installed and updated just like any other R package using, for example, the `install.packages()` function for initial installation as well as `update.packages()` for upgrades. So even though R / C++ interfacing would otherwise require source code, the **Rcpp** library is always provided ready for use as a pre-built library through the CRAN package mechanism.<sup>2</sup>

---

<sup>2</sup> This presumes a platform for which pre-built binaries are provided. **Rcpp** is available in binary form for Windows and OS X users from CRAN, and as a `.deb` package for Debian and Ubuntu users. For other systems, the **Rcpp** library is automatically built from source during installation or upgrades.

The library and header files provided by **Rcpp** for use by other packages are installed along with the **Rcpp** package. When building a package, the `LinkingTo: Rcpp` directive in the `DESCRIPTION` file lets **R** properly reference the header files automatically. That makes usage easier than for direct compilation via `R CMD SHLIB` (as in the previous section) where the function `Rcpp:::CxxFlags()` can be used to export the header file location and the appropriate `-I` switch. The **Rcpp** package also provides appropriate information for the `-L` switch needed for linking via the function `Rcpp:::LdFlags()`. It can be used by `Makevars` files of other packages, or to directly set the variables `PKG_CXXFLAGS` and `PKG_LIBS`, respectively.

The **inline** package makes use of both these facilities. All of this is done behind the scenes without the need for explicitly setting compiler or linker options. Moreover, by specifying the desired outcome rather than explicitly encode it, we provide a suitable level of indirection that permits the **Rcpp** package to completely abstract away the operating system-specific components. Usage of **Rcpp** via **inline** is therefore as portable as **R** itself: the same code will run on Windows, OS X, and Linux (provided the required tools are present as discussed earlier).

A standard example for a function extending **R** is a convolution of two vectors; this example is used throughout the “Writing R Extensions” manual (R Development Core Team 2012d). This convolution example can also be rewritten for use by **inline** as shown below. The function body is provided by the **R** character variable `src`, the function header (and its variables and their names) is defined by the argument `signature`, and we only need to enable `plugin=="Rcpp"` to obtain a new **R** function `fun` based on the **C++** code in `src`:

---

```

1 R> src <- '
  +   Rcpp::NumericVector xa(a);
3 +   Rcpp::NumericVector xb(b);
  +   int n_xa = xa.size(), n_xb = xb.size();
5 +
  +   Rcpp::NumericVector xab(n_xa + n_xb - 1);
7 +   for (int i = 0; i < n_xa; i++)
  +     for (int j = 0; j < n_xb; j++)
9 +       xab[i + j] += xa[i] * xb[j];
  +   return xab;
11 + '
R> fun <- cxxfunction(signature(a="numeric", b="numeric"),
13 +                   src, plugin="Rcpp")
R> fun( 1:4, 2:5 )
15 [1]  2  7 16 30 34 31 20

```

---

**Listing 2.4** Convolution example using **inline**

With one assignment—albeit spanning lines one to eleven—to the **R** variable `src`, and one call of the **R** function `cxxfunction` (provided by the **inline** package), we have created a new **R** function `fun` that uses the **C++** code we assigned to `src`—and all this functionality can be used directly from the **R** prompt making prototyping with **C++** functions straightforward.



Note that with version 0.3.10 or later of **inline**, a convenience wrapper `rcpp` is available which automatically adds the `plugin="Rcpp"` argument so that the invocation in Listing 2.4 could also have been written as

```
fun <- rcpp(signature(a="numeric", b="numeric"), src)
```

but we will generally use the `cxxfunction()` form.

A few further options are noteworthy at this stage. Adding `verbose=TRUE` shows both the temporary file created by `cxxfunction()` and the invocations by R CMD SHLIB. This can be useful for debugging if needed. Listing 2.5 shows the generated file. Noteworthy aspects include the function declaration with the randomized function name, and the signature with the two variable names implied from the `signature()` argument to `cxxfunction`. Also shown are the macros `BEGIN_RCPP` and `END_RCPP` discussed in Sect. 2.7.

Other options permit us to set additional compiler flags as well as additional include directories as shown in the next section.

### 2.5.2 Using Includes

As mentioned in the previous section, `cxxfunction` offers a number of other options. One aspect that we would like to focus on now is `includes`. As seen in Sect. 1.2.7, it allows us to include another block of code to, say, define a new struct or class type.

An example is provided by the following code sample from the Rcpp FAQ which was created after a user question on the **Rcpp** mailing list. A simple templated class which squares its argument is created in a code snippet supplied via `include`. The main function then uses this templated class on two different types:

```
1 R> inc <- '
2 +   template <typename T>
3 +   class square : public std::unary_function<T,T> {
4 +   public:
5 +       T operator()( T t) const { return t*t ;}
6 +   };
7 '
8
9 R> src <- '
10 +   double x = Rcpp::as<double>(xs);
11 +   int i = Rcpp::as<int>(is);
12 +   square<double> sqdbl;
13 +   square<int> sqint;
14 +   Rcpp::DataFrame df =
15 +       Rcpp::DataFrame::create(Rcpp::Named("x", sqdbl(x)),
16 +                               Rcpp::Named("i", sqint(i)));
17 +   return df;
18 '
19
20 R> fun <- cxxfunction(signature(xs="numeric", is="integer"),
21 +                     body=src, include=inc, plugin="Rcpp")
22
23 R> fun(2.2, 3L)
```

---

```

1  >> Program source :

3      1 :
4      2 : // includes from the plugin
5      3 :
6      4 : #include <Rcpp.h>
7      5 :
8      6 :
9      7 : #ifndef BEGIN_RCPP
10     8 : #define BEGIN_RCPP
11     9 : #endif
12    10 :
13    11 : #ifndef END_RCPP
14    12 : #define END_RCPP
15    13 : #endif
16    14 :
17    15 : using namespace Rcpp;
18    16 :
19    17 :
20    18 :
21    19 : // user includes
22    20 :
23    21 :
24    22 : // declarations
25    23 : extern "C" {
26    24 :     SEXP file2370678f8cfe( SEXP a, SEXP b ) ;
27    25 : }
28    26 :
29    27 : // definition
30    28 :
31    29 : SEXP file2370678f8cfe( SEXP a, SEXP b ){
32    30 :     BEGIN_RCPP
33    31 :
34    32 :     Rcpp::NumericVector xa(a);
35    33 :     Rcpp::NumericVector xb(b);
36    34 :     int n_xa = xa.size(), n_xb = xb.size();
37    35 :     Rcpp::NumericVector xab(n_xa + n_xb - 1);
38    36 :     for (int i = 0; i < n_xa; i++)
39    37 :         for (int j = 0; j < n_xb; j++)
40    38 :             xab[i + j] += xa[i] * xb[j];
41    39 :     return xab;
42    40 :
43    41 :     END_RCPP
44    42 : }
45    43 :
46    44 :

```

---

**Listing 2.5** Program source from convolution example using **inline** in verbose mode

```

21      x i
1 4.84 9

```

**Listing 2.6** Using `inline` with `include=`

This code example uses a few `Rcpp` items we have not yet encountered such as the `DataFrame` class or the static `create` method (and these will be discussed later). We again see the explicit converter `Rcpp::as<>()` used to access scalar types `integer` and `double` passed to `C++` from `R`.

More important is the definition of the sample helper class `square`. It derives from a public class `std::unary_function` templated to the same argument and return type. It also defines just one operator `()` which, unsurprisingly for a class called `square`, returns its argument squared.

The example demonstrates that while `cxxfunction` may be of primary use for short and simple test applications, it can also be used to test in more complicated setups. In fact, the plugin structure discussed in the next section allows for even more customization, should it be needed. The **RcppArmadillo**, **RcppEigen**, and **RcppGSL** packages discussed in the final part of the book all use this facility via a plugin generator.

### 2.5.3 Using Plugins

We have seen the use of the options `plugin="Rcpp"` in the previous examples. Plugins provide a general mechanism for packages using **Rcpp** to supply additional information which may be needed to compile and link the particular package. Examples may include additional header files and directories, as well as additional library names to link against as well as their locations.

Without going into too much detail about how to write a plugin, we can easily illustrate the use of a plugin. Below is a example which shows the code underlying the `fastLm()` example from **RcppArmadillo**. We will rebuild it using `cxxfunction` from **inline**:

```

R> src <- '
2 + Rcpp::NumericVector yr(ys);
+ Rcpp::NumericMatrix Xr(Xs);
4 + int n = Xr.nrow(), k = Xr.ncol();
+
6 + arma::mat X(Xr.begin(), n, k, false);
+ arma::colvec y(yr.begin(), yr.size(), false);
8 +
+ arma::colvec coef = arma::solve(X, y); // fit y ~ X
10 + arma::colvec res = y - X*coef; // residuals
+
12 + double s2 = std::inner_product(res.begin(), res.end(),
+ res.begin(), double())
14 + / (n - k);
+ arma::colvec se = arma::sqrt(s2 *

```

```

16 +       arma::diagvec(arma::inv(arma::trans(X)*X));
17 +
18 +   return Rcpp::List::create(Rcpp::Named("coef")= coef,
19 +                             Rcpp::Named("se")   = se,
20 +                             Rcpp::Named("df")    = n-k);
21 + '
22 R> fun <- cxxfunction(signature(ys="numeric", Xs="numeric"),
23 +                       src, plugin="RcppArmadillo")
24 R> ## could now run fun(y, X) to regress y ~ X

```

**Listing 2.7** A first **RcppArmadillo** example for **inline**

This illustrates nicely how **inline** can be used to compile, link, and load packages on the fly, even when these packages depend on several other **R** packages. In the case of **RcppArmadillo**, which integrates the Armadillo **C++** library, the dependency is on both **RcppArmadillo** and **Rcpp**. The plugin provides the necessary information to compile and link this example.

## 2.5.4 Creating Plugins

A simple example of how to modify a plugin is provided in the **Rcpp**-FAQ vignette. This example is centered around using the GNU Scientific Library (or GNU GSL, or just GSL for short) along with **R**. The GSL is described in Galassi et al. (2010). The example here illustrates how to set a fixed header location. A more comprehensive example might also attempt to determine the location, possibly by querying the `gsl-config` helper script as done in the **RcppGSL** package discussed in Chap. 11.

```

R> gslrng <- '
2 +   int seed = Rcpp::as<int>(par) ;
3 +   gsl_rng_env_setup();
4 +   gsl_rng *r = gsl_rng_alloc (gsl_rng_default);
5 +   gsl_rng_set (r, (unsigned long) seed);
6 +   double v = gsl_rng_get (r);
7 +   gsl_rng_free(r);
8 +   return Rcpp::wrap(v);
9 + '
10 R> plug <- Rcpp::Rcpp.plugin.maker(
11 +   include.before = "#include <gsl/gsl_rng.h>",
12 +   libs = paste("-L/usr/local/lib/R/site-library/Rcpp/lib ",
13 +               "-lRcpp -Wl,-rpath,",
14 +               "/usr/local/lib/R/site-library/Rcpp/lib ",
15 +               "-L/usr/lib -lgsl -lgslcblas -lm", sep="")
16 R> registerPlugin("gslDemo", plug)
17 R> fun <- cxxfunction(signature(par="numeric"),
18 +                       gslrng, plugin="gslDemo")
19 R> fun(0)
20 [1] 4293858116
21 R> fun(42)
22 [1] 1608637542
R>

```

**Listing 2.8** Creating a plugin for use with **inline**

Here the **Rcpp** function `Rcpp.plugin.maker` is used to create a plugin named `plug`. We specify the inclusion of the GSL header file declaring the random number generator functions. We also specify the required libraries for linking against the GSL (with values suitable for a Linux system). Subsequently, the plugin is registered and deployed in a call to `cxxfunction()`. Finally, we test the new function and generate two random draws for two different initial seeds.

## 2.6 Rcpp Attributes

A recent addition to **Rcpp** provides an even more direct connection between **C++** and **R**. This feature is called “attributes” as it is inspired by a **C++** extension of the same name in the new **C++11** standard (which will be available to **R** users only when CRAN permits use of these extension, which may be years away).

Simply put, “Rcpp attributes” internalizes key features of the **inline** package while at the same time reusing some of the infrastructure built for use by **inline** such as the plugins.

“Rcpp attributes” adds new functions `sourceCpp` to source a **C++** function (similar to how `source` is used for **R** code), `cppFunction` for a similar creation of a function from a character argument, `evalCpp` for a direct evaluation of a **C++** expression and more.

Behind the scenes, these functions make use of the existing wrappers `as<>` and `wrap` and do in fact rely heavily on them: any arguments with existing converters to or from **SEXP** types can be used. The standard build commands such as `R CMD SHLIB` and `R CMD SHLIB` are executed behind the scenes, and template programming is used to provide compile-time bindings and conversion.

An example may illustrate this:

---

```

1 cpptxt <- '
  int fibonacci(const int x) {
3     if (x < 2) return(x);
    return (fibonacci(x - 1)) + fibonacci(x - 2);
5  }'

7 fibCpp <- cppFunction(cpptxt)      # compiles, load, links, ...

```

---

**Listing 2.9** Example of new `cppFunction`

`cppFunction` returns an **R** function which calls a wrapper, also created by `cppFunction` in a temporary file which it also builds. The wrapper function in turn calls the **C++** function we passed as a character string. The build process administered by `cppFunction` uses a caching mechanism which ensures that only one compilation is needed per session (as long as the source code used is unchanged).

Alternatively, we could pass the name of a file containing the code to the function `sourceCpp` which would compile, link, and load the corresponding **C++** code and assign it to the **R** function on the left-hand side of the assignment.

These new attributes can also use **inline** plugins. The following simple example uses the plugin for the **RcppGSL** package (which is discussed more fully in Chap. 11). The program itself is not that interesting: we merely use the definitions of five physical constants.

---

```

1 R>code <- '
  + #include <gsl/gsl_const_mksa.h>                // decl of constants
3 +
  + std::vector<double> volumes() {
5 +   std::vector<double> v(5);
  +   v[0] = GSL_CONST_MKSA_US_GALLON;             // 1 US gallon
7 +   v[1] = GSL_CONST_MKSA_CANADIAN_GALLON;        // 1 Canadian gallon
  +   v[2] = GSL_CONST_MKSA_UK_GALLON;             // 1 UK gallon
9 +   v[3] = GSL_CONST_MKSA_QUART;                 // 1 quart
  +   v[4] = GSL_CONST_MKSA_PINT;                  // 1 pint
11 +   return v;
  + }'
13 R>
  R> gslVolumes <- cppFunction(code, depends="RcppGSL")
15 R> gslVolumes()
  [1] 0.003785412 0.004546090 0.004546092 0.000946353 0.000473176
17 R>

```

---

**Listing 2.10** Example of new `cppFunction` with plugin

But as **inline** is very mature and tested, and as the attributes functions are at this point not of comparable maturity, the remainder of the book will continue to use **inline** and its slightly more verbose expression. Going forward more new documentation will probably be written using the new functions once the interface stabilizes. Transitioning from one system to the other is seamless as the examples above indicated.

## 2.7 Exception Handling

C++ has a mechanism for handling exceptions. At a conceptual level, this is similar to what R programmers may already be familiar with via the `tryCatch()` function, or its simpler version `try()`.

In essence, inside a segment of code preceded by the keyword `try`, an exception can be thrown via the keyword `throw` followed by an appropriately typed exception object which is typically inherited from the `std::exception` type.

The following example may illustrate this.

---

```

1 extern "C" SEXP fun( SEXP x ) {
  try {
3     int dx = Rcpp::as<int>(x);
      if (dx > 10)
5         throw std::range_error("too big");
      return Rcpp::wrap(dx * dx);
7  } catch( std::exception& __ex__ ) {

```

---

```

    forward_exception_to_r(__ex__);
9   } catch(...) {
        ::Rf_error( "c++ exception (unknown reason)" );
11  }
    return R_NilValue; // not reached
13 }

```

**Listing 2.11** C++ example of throwing and catching an exception

For reasons that will become apparent in a moment, we are showing a complete function rather than a just short snippet used with `cxxfunction()` from the **inline** package.

If this function is compiled and linked (with appropriate flags to find the **Rcpp** headers and library), we can call it as

```

1 R> .Call("fun", 4)
  [1] 16
3 R> .Call("fun", -4)
  [1] 16
5 R> .Call("fun", 11)
Error in cpp_exception(message = "too big",
7   class = "std::range_error") : too big
R>

```

**Listing 2.12** Using C++ example of throwing and catching an exception

As the code tests only whether the argument is larger than 10, both 4 and  $-4$  are properly squared by this (not very interesting) function. For the argument 11, however, the exception is triggered via the `throw` followed by exception of type `std::range_error` with a short text indicating that the argument is too large for the assumed parameter limitation.

What happens after the `throw` is that a suitable `catch()` segment is identified. Here, as the exception was typed with a type inherited from the standard exception, the first branch is the one the code enters. The exception is then passed to an internal **Rcpp** function which converts it into an R error message. And indeed, at the R level, we see both that an exception was caught and what its type was.

This is a very useful mechanism that permits the programmer to return control to the calling instance (here the R program) with a clearly defined message.

We can illustrate this last point with a second example. What happens when we call the function with a non-numeric argument?

```

R> .Call("fun", "ABC")
2 Error in cpp_exception(message = "not compatible with INTSXP",
   class = "Rcpp::not_compatible") :
   not compatible with INTSXP
4 R>

```

**Listing 2.13** C++ example of example from **Rcpp**-type checks

Here the function is called with a character variable which cannot be used in the assignment to the integer variable `dx`. So an exception is thrown by the templated **Rcpp** function as which is templated to an integer type (written as `as<int>`)

here. The exception that is thrown is of type `Rcpp::not_compatible` which also inherits from the standard exception and a proper R error message is generated. Similar messages will be shown if the **Rcpp** types discussed in the next two chapters are instantiated with inappropriate types.

If no matching type is found, the default `catch` branch is executed. Here, it simply calls the error function of the R API with a constant text message.

Because the framework of the `try` statement (preceding the actual code block) and the `catch` clauses at the end are in fact invariant, they can also be expressed as a simple unconditional macro. Such macros are provided by **Rcpp**. Their definitions are shown in Listing 2.14.

---

```

1 #ifndef BEGIN_RCPP
2 #define BEGIN_RCPP try{
3 #endif
4
5 #ifndef VOID_END_RCPP
6 #define VOID_END_RCPP } \
7     catch (std::exception& __ex__) { \
8         forward_exception_to_r(__ex__); \
9     } \
10    catch(...) { \
11        ::Rf_error("C++ exception (unknown reason)"); \
12    }
13 #endif
14
15 #ifndef END_RCPP
16 #define END_RCPP VOID_END_RCPP return R_NilValue;
17 #endif

```

---

**Listing 2.14** C++ macros for **Rcpp** exception handling

These macros are also used by `cxxfunction()` so that the following function is fully equivalent to Listing 2.11.

---

```

1 src <- 'int dx = Rcpp::as<int>(x);
2     if( dx > 10 )
3         throw std::range_error("too big");
4     return Rcpp::wrap( dx * dx );
5 '
6 fun <- cxxfunction(x="integer", body=src, plugin="Rcpp")
7 fun(3)
8 [1] 9
9 fun(13)
10 Error: too big

```

---

**Listing 2.15** inline version of C++ example of throwing and catching an exception

Thanks to **inline**, this version is much easier to compile, link, and load. And of course, an **Rcpp** attributes version can be written just as easily:

---

```

1 cppFunction('
2     int fun2(int dx) {
3         if ( dx > 10 )

```

---



```
4         throw std::range_error("too big");
           return dx * dx;
6     }
    ')
8 fun2(3)
   [1] 9
10 fun2(13)
   Error: too big
```

---

**Listing 2.16** Rcpp attributes version of C++ example of throwing and catching an exception

The proper exception handling framework by **Rcpp** is provided automatically in both cases by adding the required code to the generated files.

<http://www.springer.com/978-1-4614-6867-7>

Seamless R and C++ Integration with Rcpp

Eddelbuettel, D.

2013, XXVIII, 220 p. 7 illus., 4 illus. in color., Softcover

ISBN: 978-1-4614-6867-7