Extending Rcpp

Dirk Eddelbuettel    Romain François

Rcpp version 0.12.10 as of March 17, 2017

Abstract

This note provides an overview of the steps programmers should follow to extend Rcpp ([Eddelbuettel, François, Allaire, Ushey, Kou, Russel, Chambers, and Bates, 2017; Eddelbuettel and François, 2011]) for use with their own classes. This document is based on our experience in extending Rcpp to work with the Armadillo ([Sanderson, 2010]) classes, available in the separate package RcppArmadillo ([Eddelbuettel, François, and Bates, 2016]). This document assumes knowledge of Rcpp as well as some knowledge of C++ templates ([Abrahams and Gurtovoy, 2004]).

1 Introduction

Rcpp facilitates data interchange between R and C++ through the templated functions Rcpp::as (for conversion of objects from R to C++) and Rcpp::wrap (for conversion from C++ to R). In other words, we convert between the so-called S-expression pointers (in type SEXP) to a templated C++ type, and vice versa. The corresponding function declarations are as follows:

```cpp
// conversion from R to C++
template<typename T> T as(SEXP x);

// conversion from C++ to R
template<typename T> SEXP wrap(const T& object);
```

These converters are often used implicitly, as in the following code chunk:

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

// [[Rcpp::export]]
List fx(List input){ // we get a list from R
    // pull std::vector<double> from R list
    // this is achieved through an implicit call to Rcpp::as
    std::vector<double> x = input["x"];

    // return an R list
    // this is achieved through implicit call to Rcpp::wrap
    return List::create(
        _["front"] = x.front(),
        _["back"] = x.back());
}
```
The Rcpp converter function Rcpp::as and Rcpp::wrap have been designed to be extensible to user-defined types and third-party types.

2 Extending Rcpp::wrap

The Rcpp::wrap converter is extensible in essentially two ways: intrusive and non-intrusive.

2.1 Intrusive extension

When extending Rcpp with your own data type, the recommended way is to implement a conversion to SEXP. This lets Rcpp::wrap know about the new data type. The template meta programming (or TMP) dispatch is able to recognize that a type is convertible to a SEXP and Rcpp::wrap will use that conversion.

The caveat is that the type must be declared before the main header file Rcpp.h is included.

```cpp
#include <RcppCommon.h>

class Foo {
    public:
        Foo() ;

        // this operator enables implicit Rcpp::wrap
        operator SEXP() ;
}

#include <Rcpp.h>
```

This is called intrusive because the conversion to SEXP operator has to be declared within the class.

2.2 Non-intrusive extension

It is often desirable to offer automatic conversion to third-party types, over which the developer has no control and can therefore not include a conversion to SEXP operator in the class definition.

To provide automatic conversion from C++ to R, one must declare a specialization of the Rcpp::wrap template between the includes of RcppCommon.h and Rcpp.h.
2.3 Templates and partial specialization

It is perfectly valid to declare a partial specialization for the Rcpp::wrap template. The compiler will identify the appropriate overload:

```cpp
#include <RcppCommon.h>

// third party library that declares template class Bling<T>
#include <foobar.h>

// declaring the partial specialization
namespace Rcpp {
    namespace traits {
        template <typename T> SEXP wrap( const Bling<T>& ) ;
    }
}

// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```
3 Extending Rcpp::as

Conversion from R to C++ is also possible in both intrusive and non-intrusive ways.

3.1 Intrusive extension

As part of its template meta programming dispatch logic, Rcpp::as will attempt to use the constructor of the target class taking a SEXP.

```cpp
#include <RcppCommon.h>

class Foo{
  public:
    Foo() ;

    // this constructor enables implicit Rcpp::as
    Foo(SEXP) ;
}

// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```

3.2 Non-intrusive extension

It is also possible to fully specialize Rcpp::as to enable non-intrusive implicit conversion capabilities.

```cpp
#include <RcppCommon.h>

// third party library that declares class Bar
#include <foobar.h>

// declaring the specialization
namespace Rcpp {
  template <> Bar as( SEXP ) ;
}

// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```

Furthermore, another non-intrusive option is to opt for sharing an R external pointer. The macro RCPP_EXPORT_AS provides an easy way to extend Rcpp::as to expose R external pointers to C++. It can be used instead of specializing Rcpp::as, and should not be used simultaneously.

```cpp
#include RcppCommon.h
#include foobar.h

RCPP_EXPORT_AS(Bar);
```
With this being said, there is one additional macro that can be used to simultaneously define both Rcpp::wrap and Rcpp::as specialization for an external pointer. The macro RCPP_EXPOSED_CLASS can be used to transparently exchange a class between R and C++ as an external pointer. Do not simultaneously use it alongside RCPP_EXPOSED AS, RCPP_EXPOSED_WRAP, Rcpp::wrap, or Rcpp::as.

### 3.3 Templates and partial specialization

The signature of Rcpp::as does not allow partial specialization. When exposing a templated class to Rcpp::as, the programmer must specialize the Rcpp::traits::Exporter template class. The TMP dispatch will recognize that a specialization of Exporter is available and delegate the conversion to this class. Rcpp defines the Rcpp::traits::Exporter template class as follows:

```cpp
namespace Rcpp {
    namespace traits {
        template<typename T> class Exporter{
            public:
                Exporter(SEXP x) : t(x) {}
                inline T get() { return t; }

            private:
                T t;
            }
        }
    }
}
```

This is the reason why the default behavior of Rcpp::as is to invoke the constructor of the type T taking a SEXP. Since partial specialization of class templates is allowed, we can expose a set of classes as follows:

```cpp
#include <RcppCommon.h>

// third party library that declares template class Bling<T>
#include <foobar.h>

// declaring the partial specialization
namespace Rcpp {
    namespace traits {
        template<typename T> class Exporter< Bling<T> >;
    }
}

// this must appear after the specialization,
// otherwise the specialization will not be seen by Rcpp types
#include <Rcpp.h>
```

Using this approach, the requirements for the Exporter< Bling<T> > class are:
- it should have a constructor taking a SEXP
- it should have a methods called get that returns an instance of the Bling<T> type.
4 Summary

The **Rcpp** package greatly facilitates the transfer of objects between R and C++. This note has shown how to extend **Rcpp** to either user-defined or third-party classes via the **Rcpp::as** and **Rcpp::wrap** template functions. Both intrusive and non-intrusive approaches were discussed.

References


