Exposing C++ functions and classes with **Rcpp** modules

Dirk Eddelbuettel Romain François

June 9, 2010

Abstract

This note discusses *Rcpp modules* which have been introduced in version 0.8.1 of the **Rcpp** package. *Rcpp modules* allow programmers to expose C++ functions and classes to R with relative ease. *Rcpp modules* are inspired from the `Boost.Python` C++ library (Abrahams and Grosse-Kunstleve, 2003) which provides the same features (and much more) for Python.

1 Motivation

Exposing C++ functionality to R is greatly facilitated by the **Rcpp** package and underlying C++ library (Eddelbuettel and François, 2010). **Rcpp** smoothes many of the rough edges in R and C++ integration by replacing the traditional R API (R Development Core Team, 2010) with a consistent set of C++ classes.

However, these facilities are limited to a function by function basis. The programmer has to implement a `.Call` compatible function using classes of the **Rcpp** API.

1.1 Exposing functions

Exposing existing C++ functions to R through **Rcpp** usually involves several steps. One often writes either an additional wrapper function that is responsible for converting input objects to the appropriate types, calling the actual worker function and converting the results back to a suitable type that can be returned to R. Alternatively, one can alter the worker function by changes to its signature and return value of the interface prescribed by the `.Call()` function of the R API. The return type has to the traditional `SEXP` from the R API. But with **Rcpp** we can also use one of the many types from the **Rcpp** API that offers implicit conversion to `SEXP`.

Consider the `hello` function below:

```
const char* hello( std::string who ){
 std::string result( "hello " ) ;
 result += who ;
 return result.c_str() ;
}
```

One can expose a such a function using **Rcpp** converters

```
RcppExport SEXP hello_wrapper( SEXP who){
    std::string input = Rcpp::as<std::string>( who )
    const char* result = hello( input ) ;
    return Rcpp::wrap( result );
}
```

Here we use the (templated) **Rcpp** converter `as()` which can transform from a `SEXP` to a number of different C++ and **Rcpp** types. The **Rcpp** function `wrap()` offers the opposite functionality and converts many known types to a `SEXP`.

For comparison, the traditionally approach using the R API looks similar

```
extern "C" SEXP hello_wrapper( SEXP who){
    std::string input = CHAR(STRING_ELT(input,0)) ;
    const char* result = hello( input ) ;
    return mkString( result );
}
```

Either way requires direct involvement from the programmer. This quickly becomes a time sink when many functions are involved. *Rcpp modules* provides a much more efficient way to expose the `hello` function to R.

1.2 Exposing classes

Exposing C++ classes or structs is even more of a challenge because it requires writing glue code for each member function that is to be exposed. Consider the simple `World` class below:

```
class World {
public:
    World() : msg("hello"){}
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }

private:
    std::string msg;
};
```

We might want a way to create objects of this class, and use the member functions `greet` and `set` to alter the object. External pointers (R Development Core Team, 2010) are the perfect vessel for this, and using the `Rcpp:::XPtr` template from **Rcpp** we can expose the class by exposing three functions :

```
using namespace Rcpp ;

/** create an external pointer to a World object */
RcppExport SEXP World__new(){
 return Rcpp::XPtr<World>( new World, true ) ;
}

/** invoke the greet method */
RcppExport SEXP World__greet( SEXP xp ) {
 Rcpp::XPtr<World> w(xp) ;
 return Rcpp::wrap( w->greet() ) ;
}

/** invoke the set method */
RcppExport SEXP World__set( SEXP xp, SEXP msg ){
 Rcpp::XPtr<World> w(xp) ;
 w->set( Rcpp::as<std::string>( msg ) ) ;
 return R_NilValue ;
}
```

which can be used from R with some S4 glue code:

```
> setClass( "World", representation( pointer = "externalptr" ) )
> World_method <- function(name){
+         paste( "World", name, sep = "__" )
+ }
> setMethod( "$", "World", function(x, name ){
+         function(...) .Call( World_method(name) , x@pointer, ... )
+ } )
> w <- new( "World", .Call( World_method( "new" ) ) )
> w$set( "hello world" )
> w$greet()
```

Rcpp considerably simplifies the code that would be involved for using external pointers with the traditional R API. This still involves a lot of pattern code that quickly becomes hard to maintain and error prone. *Rcpp modules* offer a much nicer way to expose the `World` class in a way that makes both the internal C++ code and the R code easier.

2 Rcpp modules

Rcpp modules are inspired from Python modules that are generated by the `Boost.Python` library (Abrahams and Grosse-Kunstleve, 2003). They provide an easy way to expose C++ functions and classes to R, grouped together in a single entity.

The module is created in a `cpp` file using the `RCPP_MODULE` macro, which then contains declarative code of what the module exposes to R.

2.1 Exposing C++ functions

Consider the `hello` function from the previous section. We can expose it to R :

```
const char* hello( std::string who ){
 std::string result( "hello " ) ;
 result += who ;
 return result.c_str() ;
}

RCPP_MODULE(yada){
 using namespace Rcpp ;
 function( "hello", &hello ) ;
}
```

The code creates an Rcpp module called `yada` that exposes the `hello` function. **Rcpp** automatically deduces the conversions that are needed for input and output. This alleviates the need for a wrapper function using either **Rcpp** or the R API.

On the R side, the module is simply retrieved by using the `Module` function from **Rcpp**:

```
> require( Rcpp )
> yada <- Module( "yada" )
> yada$hello( "world" )
```

A module can contain any number of calls to `function` to register many internal functions to R. For example, these 6 functions :

```
std::string hello(){
 return "hello" ;
}

int bar( int x){
 return x*2 ;
}

double foo( int x, double y){
 return x * y ;
}

void bla( ){
 Rprintf( "hello\\n" ) ;
}

void bla1( int x){
 Rprintf( "hello (x = %d)\\n", x ) ;
}

void bla2( int x, double y){
 Rprintf( "hello (x = %d, y = %5.2f)\\n", x, y ) ;
}
```

can be exposed with the following minimal code:

```
RCPP_MODULE(yada){
    using namespace Rcpp ;

    function( "hello" , &hello ) ;
    function( "bar"   , &bar   ) ;
    function( "foo"   , &foo   ) ;
    function( "bla"   , &bla   ) ;
    function( "bla1"  , &bla1   ) ;
    function( "bla2"  , &bla2   ) ;
}
```

and used from R:

```
> require( Rcpp )
> yada <- Module( "yada" )
> yada$bar( 2L )
> yada$foo( 2L, 10.0 )
> yada$hello()
> yada$bla()
> yada$bla1( 2L)
> yada$bla2( 2L, 5.0 )
```

The requirements on the functions to be exposed are:

- It takes between 0 and 65 parameters.
- The type of each input parameter must be manageable by the `Rcpp::as` template.
- The output type must be either `void` or any type that can be managed by the `Rcpp::wrap` template.
- The function name itself has to be unique, in other words no two functions with the same name but different signatures itself are allowed (whereas this is possible in C++ itself).

2.2 Exposing C++ classes

Rcpp modules also provide a mechanism for exposing C++ classes. The mechanism internally uses external pointers, but the user should consider this as hidden implementation details as this is properly encapsulated.

A class is exposed using the `class_` class. The `World` class may be exposed to R :

```
class World {
public:
    World() : msg("hello"){}
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }

private:
    std::string msg;
};

void clearWorld( World* w){
 w->set( "" ) ;
}

RCPP_MODULE(yada){
 using namespace Rcpp ;

class_<World>( "World" )
  .method( "greet", &World::greet )
  .method( "set", &World::set )
  .method( "clear", &clearWorld )
 ;

}
```

`class_` is templated by the C++ class or struct that is to be exposed to R. The parameter of the `class_<World>` constructor is the name we will use on the R side. It usually makes sense to use the same name as the class name, but this is not forced, which might be useful when exposing a class generated from a template.

The construction of the object is then followed by two calls to the `method` member function of `class_<World>`. The `method` methods can expose :

- member functions of the target class, such as `greet` or `set`, by providing the name that will be used on the R side (e.g. `greet`) and a pointer to the actual member function (e.g. `&World::greet`)
- free functions that take a pointer to the target class as their first parameter such as the C++ function `clearWorld` in the previous example. Again, we provide the R name for the method (`clear`) and a pointer to the C++ function.

The module exposes the default constructor of the `World` class as well to support creation of `World` objects from R. The **Rcpp** module assumes responsibilities for type conversion for input and output types.

```
> require( Rcpp )
> # load the module
> yada <- Module( "yada" )
> # grab the World class
> World <- yada$World
> # create a new World object
> w <- new( World )
> # use methods of the class
> w$greet()
> w$set( "hello world" )
> w$greet()
> w$clear()
> w$greet()
```

2.2.1 Const and non-const member functions

`method` is able to expose both `const` and `non const` member functions of a class. There are however situations where a class defines two versions of the same method, differing only in their signature by the `const`-ness. It is for example the case of the member functions `back` of the `std::vector` template from the STL.

```
reference back ( );
const_reference back ( ) const;
```

To resolve the ambiguity, it is possible to use `const_method` or `nonconst_method` instead of `method` in order to restrict the candidate methods.

2.2.2 S4 dispatch

When a C++ class is exposed by the `class_` template, a new S4 class is registered as well. This allows implementation of R-level (S4) dispatch. For example, one might implement the `show` method for C++ `World` objects:

```
> setMethod( "show", "World", function(object){
+         msg <- paste( "World object with message : ", object$greet() )
+         writeLines( msg )
+ } )
```

2.2.3 Special methods

Rcpp considers the methods `[[` and `[[<-` special, and promotes them to indexing methods on the R side.

2.2.4 Properties

A C++ class exposed by a module may expose data members as properties. Properties are declared by the `property` method of `class_`.

```
class Num{
public:
    Num() : x(0.0), y(0){} ;

    double getX() { return x ; }
    void setX(double value){ x = value ; }

    int getY() { return y ; }

private:
    double x ;
    int y ;
};

RCPP_MODULE(yada){
 using namespace Rcpp ;

class_<Num>( "Num" )

  // read and write property
  .property( "x", &Num::getX, &Num::setX )

  // read-only property
  .property( "y", &Num::getY )
 ;
}
```

The `x` property is declared with both getter (`getX`) and setter (`setX`) so that we can read and write the property at the R level with the dollar operator.

The `y` property only exposes a getter (`getY`) so attempting to set the property from R will generate an error.

```
> mod <- Module( "yada" )
> Num <- mod$Num
> w <- new( Num )
> w$x
> # [1] 0
> w$x <- 2.0
> w$x
> # [1] 2
> w$y
> # [1] 0
> # y is read-only, this generates an error
> w$y <- 10L
```

Getters may be const or non-const member functions of the target class taking no parameters. Free functions taking a pointer to the target class are also allowed as getters.

Setters can be non-const member function taking one parameter or a free function taking a pointer to target class as the first parameter, and the new value as the second parameter.

2.2.5 Full example

The following example illustrates how to use Rcpp modules to expose the class `std::vector<double>` from the STL.

```cpp
// convenience typedef
typedef std::vector<double> vec ;

// helpers
void vec_assign( vec* obj, Rcpp::NumericVector data ){
 obj->assign( data.begin(), data.end() ) ;
}

void vec_insert( vec* obj, int position, Rcpp::NumericVector data){
 vec::iterator it = obj->begin() + position ;
 obj->insert( it, data.begin(), data.end() ) ;
}

Rcpp::NumericVector vec_asR( vec* obj ){
 return Rcpp::wrap( *obj ) ;
}

void vec_set( vec* obj, int i, double value ){
 obj->at( i ) = value ;
}

RCPP_MODULE(yada){
    using namespace Rcpp ;

    // we expose the class std::vector<double> as "vec" on the R side
    class_<vec>( "vec")

    // exposing member functions
    .method( "size", &vec::size)
    .method( "max_size", &vec::max_size)
    .method( "resize", &vec::resize)
    .method( "capacity", &vec::capacity)
    .method( "empty", &vec::empty)
    .method( "reserve", &vec::reserve)
    .method( "push_back", &vec::push_back )
    .method( "pop_back", &vec::pop_back )
    .method( "clear", &vec::clear )

    // specifically exposing const member functions
    .const_method( "back", &vec::back )
    .const_method( "front", &vec::front )
    .const_method( "at", &vec::at )

    // exposing free functions taking a std::vector<double>*
    // as their first argument
    .method( "assign", &vec_assign )
    .method( "insert", &vec_insert )
    .method( "as.vector", &vec_asR )

    // special methods for indexing
    .const_method( "[[", &vec::at )
    .method( "[[<-", &vec_set )

;
}
```

3 Using modules in other packages

3.1 Namespace import/export

When using **Rcpp** modules in a packages, the client package needs to import a set of classes from **Rcpp**. This is achieved by adding the following line to the `NAMESPACE` file.

```
importClassesFrom( Rcpp, "C++ObjectS3", "C++Object", "C++Class", "Module" )
```

Loading modules that are defined in a package is best placed inside the `.onLoad` hook for the package.

```
NAMESPACE <- environment()
# this will be replaced by the real module
yada <- new( "Module" )
.onLoad <- function(libname, pkgname){
        # load the module and store it in our namespace
        unlockBinding( "yada" , NAMESPACE )
        assign( "yada",  Module( "yada" ), NAMESPACE )
        lockBinding( "yada", NAMESPACE )
}
```

3.2 Support for modules in skeleton generator

The `Rcpp.package.skeleton` function has been improved to help **Rcpp** modules. When the `module` argument is set to `TRUE`, the skeleton generator installs code that uses a simple module.

```
> Rcpp.package.skeleton( "testmod", module = TRUE )
```

3.3 Module documentation

Rcpp defines a `prompt` method for the `Module` class, allowing generation of a skeleton of an Rd file containing some information about the module.

```
> yada <- Module( "yada" )
> prompt( yada, "yada-module.Rd" )
```

4 Future extensions

`Boost.Python` has many more features that we would like to port to Rcpp modules : class inheritance, overloading, default arguments, enum types, ...

5 Summary

This note introduced *Rcpp modules* and illustrated how to expose C++ function and classes more easily to R. *Rcpp modules* is a relatively new addition to the **Rcpp** package and will probably undergo a few more changes. We hope that R and C++ programmers find *Rcpp modules* useful.

References

D. Abrahams and R. W. Grosse-Kunstleve. *Building Hybrid Systems with Boost.Python*. Boost Consulting, 2003. URL `http://www.boostpro.com/writing/bpl.pdf`.

D. Eddelbuettel and R. François. *Rcpp R/C++ interface package*, 2010. URL `http://CRAN.R-Project.org/package=Rcpp`. R package version 0.8.1.

R Development Core Team. *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2010. URL `http://cran.r-project.org/doc/manuals/R-exts.html`.