Rcpp Masterclass / Workshop
Part III: Rcpp Modules

Dirk Eddelbuettel[1] Romain François[2]

[1] Debian Project

[2] R Enthusiasts

28 April 2011
preceding *R / Finance 2011*
University of Illinois at Chicago

Outline

1. Introduction
2. C++ functions
3. Exposing C++ classes
4. Modules and packages
5. Exercise

Rcpp Modules - Motivation

The Rcpp API makes it easier to write and maintain C++ extension to R.

But we can do better still:

- Even more direct interfaces between C++ and R
- Automatic handling / unwrapping of arguments
- For exposing C++ functions to R
- And for exposing C++ classes to R

Standing on the shoulders of `Boost.Python`

Boost.Python is a `C++` library which enables seamless interoperability between `C++` and the `Python` programming language.

Rcpp Modules borrows from **Boost.Python** to implement interoperability between `R` and `C++`.

Rcpp Modules

C++ functions and classes:

```
double square( double x ){
  return x*x ;
}

class Foo {
public:
  Foo(double x_) :  x(x_){}

  double bar( double z){
    return pow( x - z, 2.0) ;
  }

private:
  double x ;
} ;
```

used in R:

```
> square( 2.0 )
[1] 4

> x <- new( Foo, 10 )
> x$bar( 2.0 )
[1] 16
```

Outline

1 Introduction

2 C++ functions

3 Exposing C++ classes

4 Modules and packages

5 Exercise

Exposing C++ functions

Consider the simple function :

```
double norm( double x, double y ){
    return sqrt( x*x + y*y ) ;
}
```

Exercise : expose this function to R with what we have learned this morning. We need an R function that does this:

```
> norm( 2, 3 )
[1] 3.605551
```

Exposing C++ functions

C++ side:

```
#include <Rcpp.h>
double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
SEXP norm_wraper(SEXP x_, SEXP y_){

    [...]

}
```

Compile with R CMD SHLIB:

```
$ R CMD SHLIB foo.cpp
```

R side:

```
dyn.load( "foo.so")
norm <- function(x, y){
    .Call( [...]  , x, y )
}
```

Exposing C++ functions

With inline

```
inc <- '
double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
'
src <- '
// convert the inputs
double x = as<double>(x_), y = as<double>(y_);

// call the function and store the result
double res = norm( x, y ) ;

// convert the result
return wrap(res) ;
'
norm <- cxxfunction(
    signature( x_= "numeric", y_= "numeric"),
    src, includes = inc, plugin = "Rcpp")
```

So exposing a C++ function to R is somewhat easy yet also somewhat tedious.

- Convert the inputs (from SEXP) to the appropriate types
- Call the function and store the result
- Convert the result to a SEXP

Modules use Template Meta Programming to replace these steps by :

- Declare which function to expose

Exposing C++ functions with modules

Within a package

C++ side:

```
#include <Rcpp.h>

double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
RCPP_MODULE(foo){
    function( "norm", &norm ) ;
}
```

R side:

```
.onLoad <- function(libname, pkgname){
    loadRcppModules()
}
```

(Other details to take care of. We will cover them later.)

Exposing C++ functions with modules

With inline

```
fx <- cxxfunction( , "", includes =
'
double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
RCPP_MODULE(foo){
    function( "norm", &norm ) ;
}
', plugin = "Rcpp" )
foo <- Module( "foo", getDynLib(fx) )
norm <- foo$norm
```

Exposing C++ functions

Documentation

`.function` can take an additional argument to document the exposed function:

```
double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
RCPP_MODULE(foo){
    function( "norm", &norm,

        "Some documentation about the function"

        ) ;
}
```

```
> show( mod$norm )
internal C++ function <0x1c21220>
docstring : Some documentation about the function
signature : double norm(double, double)
```

Exposing C++ functions
Formal arguments

Modules also lets you supply formal arguments for more flexibility:

```
using namespace Rcpp;
double norm( double x, double y ) {
    return sqrt( x*x + y*y );
}

RCPP_MODULE(mod_formals2) {
    function( "norm", &norm,

        List::create( _["x"], _["y"] = 0.0 ),

        "Provides a simple vector norm"
        );
}
```

Exposing C++ functions

Formal arguments

- Argument without default value : `_["x"]`
- Argument with default value : `_["y"] = 2`
- Ellipsis (...) : `_["..."]`

Outline

1. Introduction
2. C++ functions
3. Exposing C++ classes
4. Modules and packages
5. Exercise

Exposing C++ classes

Motivation

Motivation: We want to manipulate C++ objects

- Create instances
- Retrieve/Set data members
- Call methods

External pointers are useful for that, Rcpp modules wraps them in a nice to use abstration.

Exposing C++ classes

Simple C++ class:

```
class Uniform {
    public:

        // constructor
        Uniform(double min_, double max_) :
            min(min_), max(max_) {}

        // method
        NumericVector draw(int n) const {
            RNGScope scope;
            return runif( n, min, max );
        }

        // fields
        double min, max;
};
```

Exposing C++ classes

Modules can expose the `Uniform` class to allow this syntax:

```
> u <- new( Uniform, 0, 10 )
> u$draw( 10L )
[1] 3.00874606 7.00303770 6.17387340 0.06449014 7.40344856
[6] 6.48737922 1.73829428 7.53417005 0.38615597 6.66649310
> u$min
[1] 0
> u$max
[1] 10
> u$min <- 5
> u$draw(10)
[1] 7.02818458 8.19557570 5.42092100 6.02311031 8.18770124
[6] 6.18817312 8.60004068 6.60542979 5.41539068 9.96131797
```

Exposing C++ classes

Since C++ does not have reflection capabilities, modules need declaration of what to expose:

- Constructors
- Fields or properties
- Methods
- Finalizers

Exposing C++ classes

Example

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};

RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
        .constructor<double,double>()

        .field( "min", &Uniform::min )
        .field( "max", &Uniform::max  )

        .method( "draw", &Uniform::draw )
        ;
}
```

Exposing C++ classes

... Exposing constructors

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};

RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
        .constructor<double,double>()

        .field( "min", &Uniform::min )
        .field( "max", &Uniform::max  )

        .method( "draw", &Uniform::draw )
        ;
}
```

Exposing C++ classes

... Exposing fields

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};

RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
        .constructor<double,double>()

        .field( "min", &Uniform::min )
        .field( "max", &Uniform::max )

        .method( "draw", &Uniform::draw )
        ;
}
```

Exposing C++ classes

... Exposing methods

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};

RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
        .constructor<double,double>()

        .field( "min", &Uniform::min )
        .field( "max", &Uniform::max  )

        .method( "draw", &Uniform::draw )
        ;
}
```

Exposing C++ classes
... Constructors

The `.constructor` method of `class_` can expose public constructors taking between 0 and 6 arguments.

The argument types are specified as template parameters of the `.constructor` methods.

It is possible to expose several constructors that take the same number of arguments, but this require the developper to implement dispatch to choose the appropriate constructor.

Exposing C++ classes
... Fields

Public data fields are exposed with the `.field` member function:

```
.field( "x", &Uniform::x )
```

If you do not wish the R side to have write access to a field, you can use the `.field_readonly`:

```
.field_readonly( "x", &Uniform::x )
```

Exposing C++ classes
... Properties

Properties let the developper associate getters (and optionally setters) instead of retrieving the data directly. This can be useful for:

- Private or protected fields
- To keep track of field access
- To add operations when a field is retrieved or set
- To create a pseudo field that is not directly related to a data member of the class

Exposing C++ classes
... Properties

Properties are declared with one of the `.property` overloads:

```
.property( "z", &Foo::get_z, &Foo::set_z , "Doc" )
```

- 🟨 : R side name of the property (required)
- 🟥 : Address of the getter (required)
- 🟦 : Adress of the setter (optional).
- ⬜ : Documentation for the property (optional)

Exposing C++ classes

... Properties, getters

Getters can be:

```
class Foo{
public:
    double get(){ ... }
    ...
} ;
double outside_get( Foo* foo ){ ... }
```

- 🟨 Public member functions of the target class that take no argument and return something
- 🟥 Free functions that take a pointer to the target class as unique argument and returns something

Exposing C++ classes

... Properties, setters

Setters can be:

```
class Foo{
public:
    void set(double x) { ... }
    ...
} ;
void outside_set( Foo* foo , double x) { ... }
```

- 🟨 Public member functions that take exactly one argument (which must match with the type used in the getter)
- 🟥 Free function that takes exactly two arguments: a pointer to the target class, and another variable (which must match the type used in the getter).

Exposing C++ classes

Fields and properties example

```
class Foo{
public:
    double x, y ;
    double get_z(){ return z; }
    void set_z( double new_z ){ z = new_z ; }
private:
    double z ;
};
double get_w(Foo* foo){ ... }
void set_w(Foo* foo, double w ){ ...Â }

RCPP_MODULE(bla){
    class_<Foo>("Foo")
        ...
        .field( "x", &Foo::x )
        .field_readonly( "y", &Foo::y )
        .property( "z", &Foo::get_z, Foo::set_z )
        .property( "w", &get_w, &set_w )
    ;
}
```

... Methods

The `.method` member function of `class_` is used to expose methods, which can be:

- A public member function of the target class, const or non const, that takes between 0 and 65 parameters and returns either void or something
- A free function that takes a pointer to the target class, followed by between 0 and 65 parameters, and returns either void or something

... Methods, examples

```
class Foo{
public:
    ...
    void bla() ;
    double bar( int x, std::string y ) ;

} ;
double yada(Foo* foo) { ... }

RCPP_MODULE(mod){
    class_<Foo>
        ...
        .method( "bla" , &Foo::bla )
        .method( "bar" , &Foo::bar )
        .method( "yada", &yada )
    ;
}
```

... Finalizers

When the R reference object that wraps the internal C++ object goes out of scope, it becomes candidate for GC.

When it is GC'ed, the destructor of the target class is called.

Finalizers allow the developper to add behavior right before the destructor is called (free resources, etc ...)

Finalizers are associated to exposed classes with the `class_::.finalizer` method. A finalizer is a free function that takes a pointer to the target class as unique argument and returns void.

Exercize : expose this class

```
class Normal{
public:
    // 3 constructors
    Normal() : mean(0.0), sd(1.0){}
    Normal(double mean_) : mean(mean_), sd(1.0){}
    Normal(double mean_, double sd_) : mean(mean_), sd(sd_){}

    // one method
    NumericVector draw(int n){
        RNGScope scope ;
        return rnorm( n, mean, sd ) ;
    }

    // two fields (declare them read-only)
    double mean, sd ;
} ;
```

Outline

1. Introduction
2. C++ functions
3. Exposing C++ classes
4. Modules and packages
5. Exercise

Modules and packages

The best way to use Rcpp modules is to embed them in an R package.

The `Rcpp.package.skeleton` (and its `module` argument) creates a package skeleton that has an Rcpp module.

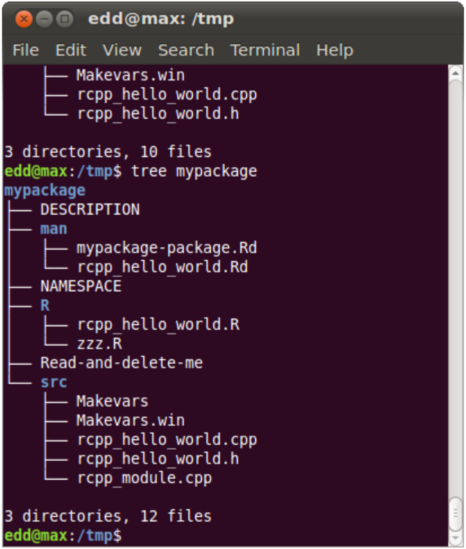
```
> Rcpp.package.skeleton( "mypackage",
+     module = TRUE )
```

Modules and packages

```
> Rcpp.package.skeleton( "mypackage", module=TRUE )
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './mypackage/Read-and-delete-me'.

Adding Rcpp settings
>> added RcppModules:  yada
>> added Depends:  Rcpp
>> added LinkingTo:  Rcpp
>> added useDynLib directive to NAMESPACE
>> added Makevars file with Rcpp settings
>> added Makevars.win file with Rcpp settings
>> added example header file using Rcpp classes
>> added example src file using Rcpp classes
>> added example R file calling the C++ example
>> added Rd file for rcpp_hello_world
>> copied the example module
```

Calling `Rcpp.package.skeleton`



```
    ├── Makevars.win
    ├── rcpp_hello_world.cpp
    └── rcpp_hello_world.h

3 directories, 10 files
edd@max:/tmp$ tree mypackage
mypackage
├── DESCRIPTION
├── man
│   ├── mypackage-package.Rd
│   └── rcpp_hello_world.Rd
├── NAMESPACE
├── R
│   ├── rcpp_hello_world.R
│   └── zzz.R
├── Read-and-delete-me
└── src
    ├── Makevars
    ├── Makevars.win
    ├── rcpp_hello_world.cpp
    ├── rcpp_hello_world.h
    └── rcpp_module.cpp

3 directories, 12 files
edd@max:/tmp$
```

We will discuss the individual files in the next few slides.

rcpp_module.cpp

```
#include <Rcpp.h>

[...]
int bar( int x){
  return x*2 ;
}
double foo( int x, double y){
  return x * y ;
}
[...]

class World {
public:

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

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

rcpp_module.cpp

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

  [...]

  function( "bar", &bar,
    List::create( _["x"] = 0.0 ),
    "documentation for bar ") ;

  function( "foo"   , &foo    ,
    List::create( _["x"] = 1, _["y"] = 1.0 ),
    "documentation for foo ") ;

  class_<World>( "World")
    .constructor()
    .method( "greet", &World::greet ,  "get the message")
    .method( "set", &World::set     ,  "set the message")
  ;
}
```

Modules and packages. DESCRIPTION

```
Package:  mypackage
Type:  Package
Title:  What the package does (short line)
Version:  1.0
Date:  2011-04-27
Author:  Who wrote it
Maintainer:  Who to complain to <yourfault@somewhere.net>
Description:  More about what it does (maybe more than one line)
License:  What Licence is it under ?
LazyLoad:  yes
Depends:  methods, Rcpp (>= 0.9.4.1)
LinkingTo:  Rcpp
RcppModules:  yada
```

Modules and packages. zzz.R

The `.onLoad` function, typically included in the `zzz.R` file of a package must contain a call to the `loadRcppModules` function:

```
.onLoad <- function(libname, pkgname){
    loadRcppModules()
}
```

Modules and packages. NAMESPACE

```
useDynLib(mypackage)
exportPattern("^[[:alpha:]]+")
import( Rcpp )
```

Modules and packages. Using the package

```
> require( mypackage )
> foo
internal C++ function <0x100612350>
    docstring : documentation for foo
    signature : double foo(int, double)
> foo( 2, 3 )
[1] 6
> World
C++ class 'World' <0x10060edc0>
Constructors:
    World()

Fields: No public fields exposed by this class

Methods:
     std::string greet()
           docstring : get the message
     void set(std::string)
           docstring : set the message
> w <- new( World )
> w$set( "bla bla")
> w$greet()
[1] "bla bla"
```

Outline

1. Introduction
2. C++ functions
3. Exposing C++ classes
4. Modules and packages
5. Exercise

Exercise

- Create a package with a module : start by using `Rcpp.package.skeleton`
- Expose two C++ functions
- Expose a C++ class

```
double fun1( NumericVector x){
  return sum(
    head(x,-1) - tail(x,-1)
  ) ;
}

double fun2( NumericVector x
){
  return mean(x) / sd(x) ;
}
```

```
class Normal {
  public:
    Normal(
      double mean_, double sd_
    ) ;
    NumericVector draw( int n ) ;
    int get_ndraws() ;

  private:
    double mean, sd ;
    int ndraws ;
} ;
```