Rcpp Workshop
Part III: Advanced Rcpp

Dr. Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org

Sponsored by ASA, CTSI and PCOR
Medical College of Wisconsin
Milwaukee, WI
May 11, 2013

Outline

Motivating Sugar

Recall the earlier example of a simple (albeit contrived for the purposes of this discussion) R vector expression:

```
ifelse(x < y, x*x, -(y*y))
```

which for a given vector `x` will execute a simple transformation.

We saw a basic C implementation. How would we write it in C++ ?

Motivating sugar

examples/part3/sugarEx1.cpp

Maybe like this.

```
SEXP foo(SEXP xx, SEXP yy) {
  int n = x.size();
  NumericVector res1( n );
  double x_ = 0.0, y_ = 0.0;
  for (int i=0; i<n; i++) {
    x_ = x[i];
    y_ = y[i];
    if (R_IsNA(x_) || R_IsNA(y_)) {
      res1[i] = NA_REAL;
    } else if (x_ < y_) {
      res1[i] = x_ * x_;
    } else {
      res1[i] = -(y_ * y_);
    }
  }
  return(x);
}
```

Motivating sugar

examples/part3/sugarEx2.cpp

But with `sugar` we can simply write it as

```
SEXP foo( SEXP xx, SEXP yy) {
  NumericVector x(xx), y(yy) ;
  return ifelse( x < y, x*x, -(y*y) ) ;
}
```

Or using Rcpp Attributes:

```
#include <Rcpp.h>
NumericVector f(NumericVector x, NumericVector y) {
  return ifelse( x < y, x*x, -(y*y) ) ;
}
```

Sugar: Another example

examples/part3/sugarEx3.cpp

Sugar also gives us things like `sapply` on C++ vectors:

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

SEXP foo( SEXP xx ){
  NumericVector x(xx) ;
  return sapply( x, square ) ;
}
```

Outline

Sugar: Overview of Contents

logical operators `<`, `>`, `<=`, `>=`, `==`, `!=`

arithmetic operators `+`, `-`, `*`, `/`

functions on vectors `abs`, `all`, `any`, `ceiling`, `cumsum`, `diag`, `diff`, `exp`, `head`, `ifelse`, `is_na`, `lapply`, `mean`, `pmin`, `pmax`, `pow`, `rep`, `rep_each`, `rep_len`, `rev`, `sapply`, `seq_along`, `seq_len`, `sd`, `sign`, `sum`, `tail`, `var`,

functions on matrices `outer`, `col`, `row`, `lower_tri`, `upper_tri`, `diag`

statistical functions (dpqr) `rnorm`, `dpois`, `qlogis`, etc ...

More information in the `Rcpp-sugar` vignette.

Outline

Binary arithmetic operators

Sugar defines the usual binary arithmetic operators : `+`, `-`, `*`, `/`.

```
// two numeric vectors of the same size
NumericVector x ;
NumericVector y ;

// expressions involving two vectors
NumericVector res = x + y ;
NumericVector res = x - y ;
NumericVector res = x * y ;
NumericVector res = x / y ;

// one vector, one single value
NumericVector res = x + 2.0 ;
NumericVector res = 2.0 - x;
NumericVector res = y * 2.0 ;
NumericVector res = 2.0 / y;

// two expressions
NumericVector res = x * y + y / 2.0 ;
NumericVector res = x * ( y - 2.0 ) ;
NumericVector res = x / ( y * y ) ;
```

Binary logical operators

```
// two integer vectors of the same size
NumericVector x ;
NumericVector y ;

// expressions involving two vectors
LogicalVector res = x < y ;
LogicalVector res = x > y ;
LogicalVector res = x <= y ;
LogicalVector res = x >= y ;
LogicalVector res = x == y ;
LogicalVector res = x != y ;

// one vector, one single value
LogicalVector res = x < 2 ;
LogicalVector res = 2 > x;
LogicalVector res = y <= 2 ;
LogicalVector res = 2 != y;

// two expressions
LogicalVector res = ( x + y ) <  ( x*x ) ;
LogicalVector res = ( x + y ) >= ( x*x ) ;
LogicalVector res = ( x + y ) == ( x*x ) ;
```

Unary operators

```
// a numeric vector
NumericVector x ;

// negate x
NumericVector res = -x ;

// use it as part of a numerical expression
NumericVector res = -x * ( x + 2.0 ) ;


// two integer vectors of the same size
NumericVector y ;
NumericVector z ;

// negate the logical expression "y < z"
LogicalVector res = ! ( y < z );
```

Outline

Functions producing a single logical result

```
IntegerVector x = seq_len( 1000 ) ;
all( x*x < 3 ) ;

any( x*x < 3 ) ;


// wrong: will generate a compile error
bool res = any( x < y) ) ;

// ok
bool res = is_true( any( x < y ) )
bool res = is_false( any( x < y ) )
bool res = is_na( any( x < y ) )
```

Functions producing sugar expressions

```
IntegerVector x = IntegerVector::create( 0, 1, NA_INTEGER, 3 ) ;

is_na( x )
all( is_na( x ) )
any( ! is_na( x ) )

seq_along( x )
seq_along( x * x * x * x * x * x * x * x )

IntegerVector x = seq_len( 10 ) ;

pmin( x, x*x );
pmin( x*x, 2 );

IntegerVector x, y;

ifelse( x < y, x, (x+y)*y );
ifelse( x > y, x, 2 );

sign( xx );
sign( xx * xx );

diff( xx );
```

Mathematical functions

```
IntegerVector x;

abs( x )
exp( x )
log( x )
log10( x )
floor( x )
ceil( x )
sqrt( x )
pow(x, z)      # x to the power of z
```

plus the regular trigonometrics functions and more.

Statistical function d/q/p/r

```
x1 = dnorm(y1, 0, 1);   // density of y1 at m=0, sd=1
x2 = pnorm(y2, 0, 1);   // distribution function of y2
x3 = qnorm(y3, 0, 1);   // quantiles of y3
x4 = rnorm(n, 0, 1);    // 'n' RNG draws of N(0, 1)
```

For beta, binom, caucht, exp, f, gamma, geom, hyper, lnorm, logis, nbeta, nbinom, nbinom_mu, nchisq, nf, norm, nt, pois, t, unif and weibull.

Use something like `RNGScope scope;` to set/reset the RNGs.

Outline

Sugar: benchmarks

expression		sugar	R	R / sugar
`any(x*y<0)`		0.000451	5.17	11450
`ifelse(x<y,x*x,-(y*y))`		1.378	13.15	9.54
`ifelse(x<y,x*x,-(y*y))`	(⋆)	1.254	13.03	10.39
`sapply(x,square)`		0.220	113.38	515.24

Source: `examples/SugarPerformance/` using R 2.13.0, **Rcpp** 0.9.4, `g++-4.5`, Linux 2.6.32, i7 cpu.

⋆ : version includes optimization related to the absence of missing values

Sugar: benchmarks

Benchmarks of the convolution example from Writing R Extensions.

Implementation	Time in millisec	Relative to R API
R API (as benchmark)	234	
Rcpp sugar	158	0.68
`NumericVector::iterator`	236	1.01
`NumericVector::operator[]`	305	1.30
R API *naively*	2199	9.40

Table: Convolution of *x* and *y* (200 values), repeated 5000 times.

Source: `examples/ConvolveBenchmarks/` using R 2.13.0, **Rcpp** 0.9.4, `g++-4.5`, Linux 2.6.32, i7 cpu.

Sugar: Final Example

examples/part3/sugarExample.R

Consider a simple R function of a vector:

```
foo <- function(x) {

    ## sum of
    ##   -- squares of negatives
    ##   -- exponentials of positives
    s <- sum(ifelse( x < 0,  x*x,  exp(x) ))

    return(s)
}
```

Sugar: Final Example

examples/part3/sugarExample.R

Here is one C++ solution:

```
bar <- cxxfunction(signature(xs="numeric"),
                   plugin="Rcpp", body='

    NumericVector x(xs);

    double s = sum( ifelse( x < 0, x*x, exp(x) ));

    return wrap(s);
')
```

Sugar: Final Example

Benchmark from `examples/part3/sugarExample.R`

```
R> library(compiler)
R> cfoo <- cmpfun(foo)
R> library(rbenchmark)
R> x <- rnorm(1e5)
R> benchmark(foo(x), cfoo(x), bar(x),
+            columns=c("test", "elapsed", "relative",
+                      "user.self", "sys.self"),
+            order="relative", replications=10)
     test elapsed relative user.self sys.self
3  bar(x)   0.033   1.0000      0.03        0
1  foo(x)   0.441  13.3636      0.45        0
2 cfoo(x)   0.463  14.0303      0.46        0
R>
```

Outline

Rcpp Modules - Motivation

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

But we can do better still:

- Even more direct interfaces between C++ and R
- Automatic handling / unwrapping of arguments
- Support exposing C++ functions to R
- Also support 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 similar 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;
};
```

This can be used in R:

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

> x <- new( Foo, 10 )
> x$bar( 2.0 )
[1] 64
```

Exposing C++ functions

Consider the simple function :

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

Thanks to Rcpp Attributes, exposing this is now easy.

Attributes was motivated by Modules, and it is useful to study Modules in some detail.

Exposing C++ functions

C++ side:

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

SEXP norm_wrapper(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(y) ;
'
norm <- cxxfunction(signature(x_ = "numeric",
                              y_ = "numeric" ),
                    body = src, includes = inc,
                    plugin = "Rcpp" )
```

Exposing C++ functions (cont.)

So exposing a C++ function to R is straightforward, 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

Rcpp Modules use Template Meta Programming (TMP) to replace these steps by a single step:

- 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 related to module loading to take care of. We will cover them later.)

Exposing C++ functions with modules

Using 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"
             );
}
```

which can be displayed from the R prompt:

```
R> 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 let 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

Rcpp modules supports different types of arguments:

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

Outline

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, and **Rcpp** modules wraps them in a nice to use abstration.

Exposing C++ classes

A 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 to declare what to expose:

- Constructors
- Fields or properties
- Methods
- Finalizers

Exposing C++ classes

A simple 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 7 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:

```
.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,
           "Documentation" )
```

This contains

- the R side name of the property (required)
- address of the getter (required)
- address 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.

Outline

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.

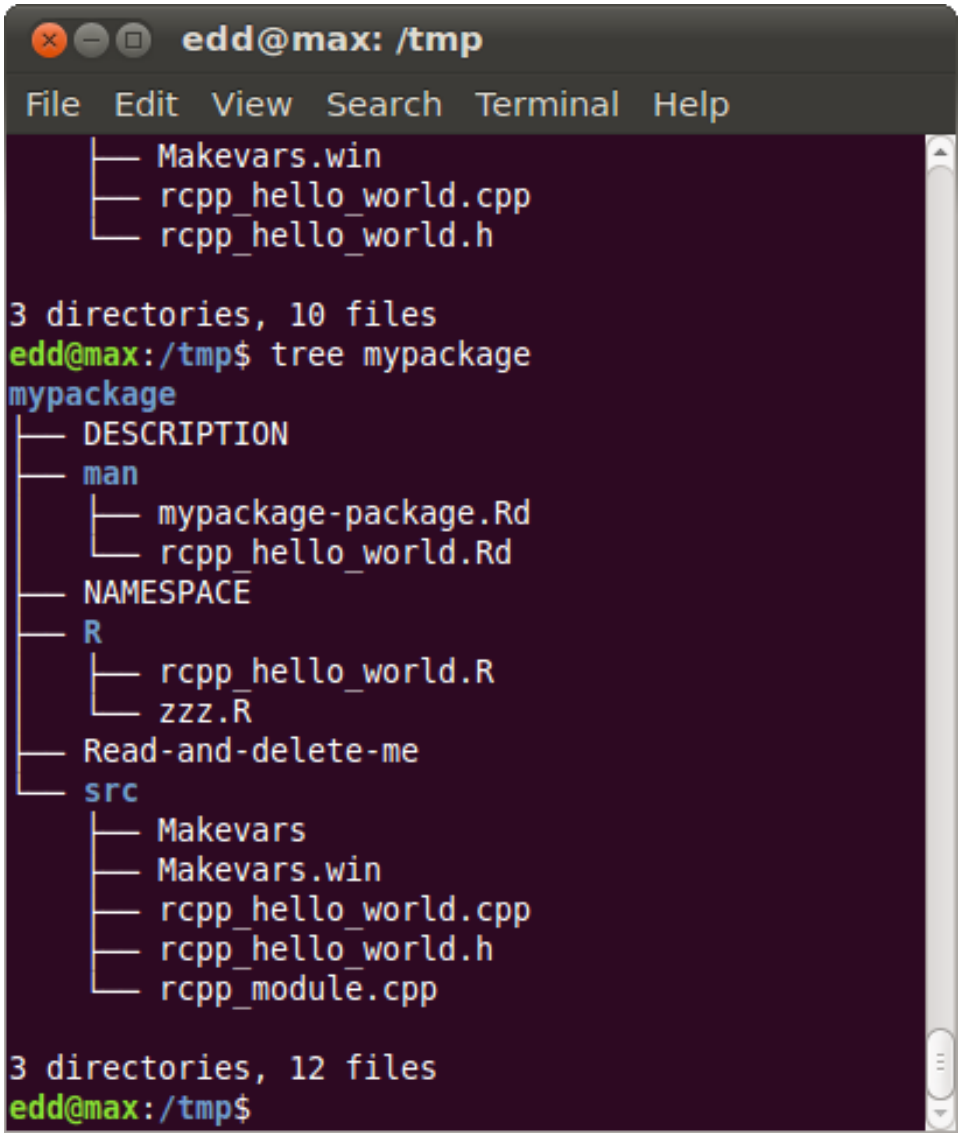
```
> Rcpp.package.skeleton("mypackage",
+                       module = TRUE )
```

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`



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

Also note that the next release will contain two more `cpp` files.

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: 2011-08-15
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 (>= 0.9.6)
LinkingTo: Rcpp
RcppModules: yada
```

Modules and packages: zzz.R

Previously, the `.onLoad()` function (often placed in file `zzz.R` file) had to contai a call to the `loadRcppModules` function.

Now, we simply call `loadModule()` from any file in the `R/` directory of the package.

```
## For R 2.15.1 and later this works.
## Note that calling loadModule() triggers
## a load action, so this does _not_ have
## to be placed in .onLoad() or evalqOnLoad().
loadModule("NumEx", TRUE)
loadModule("yada", TRUE)
loadModule("stdVector", TRUE)
```

Modules and packages: NAMESPACE

The `NAMESPACE` files loads the dyanmic library of the packages, imports from **Rcpp** and exports all local symbols of the package (using regular expression).

```
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

stdVector.cpp (in Rcpp's skeleton and unittest)

```
#include <Rcpp.h>                       // need to include the main Rcpp header file only

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

void vec_assign( vec* obj, Rcpp::NumericVector data) { // helpers
    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;
}

void vec_resize( vec* obj, int n) { obj->resize( n ); }
void vec_push_back( vec* obj, double x ) { obj->push_back( x ); }

// Wrappers for member functions that return a reference -- required on Solaris
double vec_back(vec *obj){ return obj->back() ; }
double vec_front(vec *obj){ return obj->front() ; }
double vec_at(vec *obj, int i){ return obj->at(i) ; }
```

stdVector.cpp cont.

```
RCPP_MODULE(stdVector){
    using namespace Rcpp ;
    // we expose the class std::vector<double> as "vec" on the R side
    class_<vec>("vec")
    // exposing the default constructor
    .constructor()
    // exposing member functions -- taken directly from std::vector<double>
    .method( "size",      &vec::size)
    .method( "max_size",  &vec::max_size)
    .method( "capacity",  &vec::capacity)
    .method( "empty",     &vec::empty)
    .method( "reserve",   &vec::reserve)
    .method( "pop_back",  &vec::pop_back )
    .method( "clear",     &vec::clear )
    // specifically exposing const member functions defined above
    .method( "back",      &vec_back )
    .method( "front",     &vec_front )
    .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 )
    .method( "push_back",&vec_push_back )
    .method( "resize",    &vec_resize)
    // special methods for indexing
    .method( "[[",        &vec_at )
    .method( "[[<-",      &vec_set )
    ;
}
```

stdVector.cpp cont.

Usage from R

```
v <- new(vec)                                    # stdVector module

data <- 1:10
v$assign(data)

v[[3]] <- v[[3]] + 1
data[[4]] <- data[[4]] +1

checkEquals( v$as.vector(), data )

v$size()
v$capacity()
```

Outline

planar/src/multilayer.cpp

```
#include <RcppArmadillo.h>
#include <iostream>

using namespace Rcpp ;
using namespace RcppArmadillo ;
using namespace arma ;
using namespace std;

Rcpp::List multilayer(const arma::colvec& k0,     \
        const arma::cx_mat& kx,     \
        const arma::cx_mat& epsilon,    \
        const arma::colvec& thickness,    \
        const int& polarisation) {
 [...]
Rcpp::List recursive_fresnel(const arma::colvec& k0,    \
        const arma::cx_mat& kx,    \
        const arma::cx_mat& epsilon,   \
        const arma::colvec& thickness,   \
        const int& polarisation) {

[...]

RCPP_MODULE(planar){
  using namespace Rcpp;

  function( "multilayer", &multilayer,      \
    "Calculates reflection and transmission coefficients of a multilayer stack" );
  function( "recursive_fresnel", &recursive_fresnel,      \
    "Calculates the reflection coefficient of a multilayer stack" );
 }
```

cda/src/{cda,utils}.cpp

```
#include "utils.h"
#include "cda.h"
#include <RcppArmadillo.h>
#include <iostream>

using namespace Rcpp;
using namespace RcppArmadillo;
using namespace std;

arma::mat euler(const double phi, const double theta, const double psi) {
[...]
arma::cx_mat interaction_matrix(const arma::mat& R, const double kn,
                                const arma::cx_mat& invAlpha,
                                const arma::mat& Euler, const int full) {
double extinction(const double kn, const arma::cx_colvec& P,
                  const arma::cx_colvec& Eincident) {
[...]
double absorption(const double kn, const arma::cx_colvec& P,
                  const arma::cx_mat& invpolar) {
[...]

RCPP_MODULE(cda){
    using namespace Rcpp ;

    function( "euler", &euler, "Constructs a 3x3 Euler rotation matrix" ) ;
    function( "extinction", &extinction, "Calculates the extinction cross-section" ) ;
    function( "absorption", &absorption, "Calculates the absorption cross-section" ) ;
    function( "interaction_matrix", &interaction_matrix,
              "Constructs the coupled-dipole interaction matrix" ) ;
}
```

GUTS/src/GUTS_rpp_module.cpp

```
#include "GUTS.h"
#include <Rcpp.h>

using namespace Rcpp;

RCPP_MODULE(modguts)
{
  class_<GUTS>( "GUTS" )
    .constructor()

    .method( "setConcentrations",   &GUTS::setConcentrations,
             "Set time series vector of concentrations." )
    .method( "setSurvivors",        &GUTS::setSurvivors,
             "Set time series vector of survivors." )
    [...]

    .method( "setSample", &GUTS::setSample, "Set ordered sample vector." )

    .method( "calcLoglikelihood", &GUTS::calcLoglikelihood,
             "Returns calculated log. of likelihood from complete + valid object." )

    .property( "C",     &GUTS::getC, "Vector of concentrations." )
    .property( "Ct",    &GUTS::getCt, "Time vector of concentrations." )
    [...]
    ;
}
```

RcppBDT/src/RcppBDT.cpp

```
RCPP_MODULE(bdt) {

    using namespace boost::gregorian;
    using namespace Rcpp;

    // exposing a class (boost::gregorian::)date as "date" on the R side
    class_<date>("date")

    // constructors
    .constructor("default constructor")
    .constructor<int, int, int>("constructor from year, month, day")

    .method("setFromLocalClock", &date_localDay, "create a date from local clock")
    .method("setFromUTC", &date_utcDay, "create a date from current universal clock")
    [...]
    .method("getYear", &date_year, "returns the year")
    .method("getMonth", &date_month, "returns the month")
    .method("getDay", &date_day, "returns the day")
    .method("getDayOfYear", &date_dayofyear, "returns the day of the year")
    [...]
    .method("getDate", &date_toDate, "returns an R Date object")
    .method("fromDate", &date_fromDate, "sets date from an R Date object")
    [...]
    .const_method("getWeekNumber", &date::week_number, "returns number of week")
    .const_method("getModJulian", &date::modjulian_day, "returns the mod. Julian day")
    .const_method("getJulian", &date::julian_day, "returns the Julian day")
    [...]
    .method("getNthDayOfWeek", &Date_nthDayOfWeek,
            "return nth week's given day-of-week in given month and year")
    [...]
    ;
}
```

Outline

Overview

Last year, John Chambers added some code to Rcpp. It builds on Rcpp Modules, and allows the R side to modify C++ classes.

This is documented in `help(setRcppClass)` as well as in one test package to to support the unit tests.

Outline

Example

```
setRcppClass("World", module = "yada", fields = list(more = "character"),
         methods = list(test = function(what) message("Testing: ", what, "; ", more)),
         saveAs = "genWorld")
setRcppClass("stdNumeric", "vec", "stdVector")
evalqOnLoad({  # some methods that use C++ methods
    stdNumeric$methods(
        getEl = function(i) {
            i <- as.integer(i)
            if(i < 1 || i > size())
                NA_real_
            else
                at(i-1L)
        },
        setEl = function(i, value) {
            value <- as.numeric(value)
            if(length(value) != 1)
                stop("Only assigns single values")
            i <- as.integer(i)
            if(i < 1 || i > size())
                stop("index out of bounds")
            else
                set(i-1L, value)
        },
        initialize = function(data = numeric()) {
            callSuper()
            data <- as.double(data)
            n <- as.integer(max(50, length(data) * 2))
            reserve(n)
            assign(data)
        }
    )
})
```