Package ‘Rcpp’

March 23, 2013

Title Seamless R and C++ Integration

Version 0.10.3

Date $Date: 2013-03-23 09:49:00 -0500 (Sat, 23 Mar 2013) $

Author Dirk Eddelbuettel and Romain Francois, with contributions by
Douglas Bates, John Chambers and JJ Allaire

Maintainer Dirk Eddelbuettel <edd@debian.org>

Description The Rcpp package provides R functions as well as a C++
library which facilitate the integration of R and C++.  
R data types (SEXP) are matched to C++ objects in a class hierarchy.
All R types are supported (vectors, functions, environment, etc ...
) and each type is mapped to a dedicated class. For example, numeric vectors are represented as instances of the Rcpp::NumericVector class, environments are represented as instances of Rcpp::Environment, functions are represented as Rcpp::Function, etc ... The ‘‘Rcpp-introduction’’ vignette provides a good entry point to Rcpp.

Conversion from C++ to R and back is driven by the templates Rcpp::wrap
and Rcpp::as which are highly flexible and extensible, as documented in the ‘‘Rcpp-extending’’ vignette.

Rcpp also provides Rcpp modules, a framework that allows exposing C++
facts and classes to the R level. The ‘‘Rcpp-modules’’vignette details the current set of features of Rcpp-modules.

Rcpp includes a concept called Rcpp sugar that brings many R functions
into C++. Sugar takes advantage of lazy evaluation and
expression templates to achieve great performance while
exposing a syntax that is much nicer to use than the equivalent
low-level loop code. The ‘‘Rcpp-sugar’’ vignette gives an overview of the feature.

Rcpp attributes provide a high-level syntax for declaring C++ functions
as callable from R and automatically generating the code
required to invoke them. Attributes are intended to facilitate
both interactive use of C++ within R sessions as well as to
support R package development. Attributes are built on top of Rcpp modules and their implementation is based on previous work in the inline package.

Many examples are included, and over 870 unit tests in over 400 unit test functions provide additional usage examples.

An earlier version of Rcpp, containing what we now call the 'classic Rcpp API' was written during 2005 and 2006 by Dominick Samperi. This code has been factored out of Rcpp into the package RcppClassic, and it is still available for code relying on the older interface. New development should always use this Rcpp package instead.

**Depends** R (>= 2.15.1)

**Imports** methods

**Suggests** RUnit, inline, rbenchmark

**URL** http://dirk.eddelbuettel.com/code/rcpp.html,

**License** GPL (>= 2)

**BugReports** http://r-forge.r-project.org/tracker/?atid=637&group_id=155&func=browse

**MailingList** Please send questions and comments regarding Rcpp to rcpp-devel@lists.r-forge.r-project.org

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2013-03-23 17:05:30

**R topics documented:**

- Rcpp-package ................................................................. 3
- .DollarNames-methods ................................................... 4
- C++Class-class ............................................................ 4
- C++Constructor-class .................................................... 5
- C++Field-class ............................................................ 5
- C++Function-class ....................................................... 6
- C++Object-class .......................................................... 7
- C++OverloadedMethods-class .......................................... 7
- compileAttributes ....................................................... 8
- cppFunction ............................................................... 9
- dependsAttribute ....................................................... 11
- evalCpp ................................................................. 12
- exportAttribute ......................................................... 13
- formals<--methods .................................................... 14
- interfacesAttribute .................................................... 15
- loadModule ............................................................... 16
- loadRcppModules ....................................................... 18
Description

The Rcpp package provides C++ classes that greatly facilitate interfacing C or C++ code in R packages using the .Call interface provided by R.

Introduction

Rcpp provides C++ classes to facilitate manipulation of a large number of R data structures: vectors, functions, environments, ...

The “Rcpp-introduction” vignette gives an introduction on the package.

Usage for package building

The “Rcpp-package” vignette documents how to use Rcpp in client packages.

History

The initial versions of Rcpp were written by Dominick Samperi during 2005 and 2006.

Dirk Eddelbuettel made some additions, and became maintainer in 2008.

Dirk Eddelbuettel and Romain Francois have been extending Rcpp since 2009.

Author(s)

Dirk Eddelbuettel and Romain Francois

References

## Not run:

```r
# introduction to Rcpp
vignette("Rcpp-introduction")

# information on how to build a package that uses Rcpp
vignette("Rcpp-package")
```

## End(Not run)

### .DollarNames-methods completion

**Description**

completion

**Methods**

signature(x = "ANY")

signature(x = "C++Object") completes fields and methods of C++ objects

signature(x = "Module") completes functions and classes of modules

### C++Class-class

**Description**

Information about an internal c++ class.

**Objects from the Class**

Objects are usually extracted from a **Module** using the dollar extractor.

**Slots**

.Data: mangled name of the class

.pointer: external pointer to the internal infomation

.module: external pointer to the module

.fields: list of C++Field objects

.constructors: list of C++Constructor objects

.methods: list of C++OverloadedMethods objects

.generator: the generator object for the class
**C++Constructor-class**

- **docstring** description of the class
- **typeid** unmangled typeid of the class
- **enums** enums of the class
- **parents** names of the parent classes of this class

**Methods**

- **show** signature(object = "C++Class"): prints the class.
- $ signature(object = "C++Class"): ...

---

**C++Constructor-class**  
**Class "C++Constructor"**

**Description**

Representation of a C++ constructor

**Extends**

Class "envRefClass", directly. Class ".environment", by class "envRefClass", distance 2. Class "refClass", by class "envRefClass", distance 2. Class "environment", by class "envRefClass", distance 3, with explicit coerce. Class "refObject", by class "envRefClass", distance 3.

**Fields**

- **pointer**: pointer to the internal structure that represent the constructor
- **class_pointer**: pointer to the internal structure that represent the associated C++ class
- **nargs**: Number of arguments the constructor expects
- **signature**: C++ signature of the constructor
- **docstring**: Short description of the constructor

---

**C++Field-class**  
**Class "C++Field"**

**Description**

Metadata associated with a field of a class exposed through Rcpp modules

**Fields**

- **pointer**: external pointer to the internal (C++) object that represents fields
- **cpp_class**: (demangled) name of the C++ class of the field
- **read_only**: Is this field read only
- **class_pointer**: external pointer to the class this field is from.
Methods

No methods defined with class "C++Field" in the signature.

See Also

The fields slot of the C++Class class is a list of C++Field objects

Examples

showClass("C++Field")

c++Function-class

Class "C++Function"

Description

Internal C++ function

Objects from the Class

Objects can be created by the Rcpp::InternalFunction class from the Rcpp library

Slots

.Data: R function that calls back to the internal function
pointer: External pointer to a C++ object pointing to the function
docstring: Short documentation for the function
signature: C++ signature

Extends

Class "function", from data part. Class "OptionalFunction", by class "function", distance 2.
Class "PossibleMethod", by class "function", distance 2.

Methods

show signature(object = "C++Function"): print the object

Examples

showClass("C++Function")
C++Object-class

C++ internal objects

Description

C++ internal objects instanciated from a class exposed in an Rcpp module

Objects from the Class

This is a virtual class. Actual C++ classes are subclasses.

Methods

$ signature(x = "C++Object"): invokes a method on the object, or retrieves the value of a property

$<- signature(x = "C++Object"): set the value of a property

show signature(object = "C++Object"): print the object

C++OverloadedMethods-class

Class "C++OverloadedMethods"

Description

Set of C++ methods

Extends

Class "envRefClass", directly. Class ".environment", by class "envRefClass", distance 2. Class "refClass", by class "envRefClass", distance 2. Class "environment", by class "envRefClass", distance 3, with explicit coerce. Class "refObject", by class "envRefClass", distance 3.

Fields

pointer: Object of class externalptr pointer to the internal structure that represents the set of methods
class_pointer: Object of class externalptr pointer to the internal structure that models the related class
compileAttributes  Compile Rcpp Attributes for a Package

Description
Scan the source files within a package for attributes and generate code as required. Generates the bindings required to call C++ functions from R for functions adorned with the Rcpp::export attribute.

Usage
compileAttributes(pkgdir = ".", verbose = getOption("verbose"))

Arguments
pkgdir
Directory containing the package to compile attributes for (defaults to the current working directory).

verbose
TRUE to print detailed information about generated code to the console.

Details
The source files in the package directory given by pkgdir are scanned for attributes and code is generated as required based on the attributes.

For C++ functions adorned with the Rcpp::export attribute, the C++ and R source code required to bind to the function from R is generated and added (respectively) to src/RcppExports.cpp or R/RcppExports.R.

In order to access the declarations for custom Rcpp::as and Rcpp::wrap handlers the compileAttributes function will also call any inline plugins available for packages listed in the LinkingTo field of the DESCRIPTION file.

Value
Returns (invisibly) a character vector with the paths to any files that were updated as a result of the call.

Note
The compileAttributes function deals only with exporting C++ functions to R. If you want the functions to additionally be publicly available from your package’s namespace another step may be required. Specifically, if your package NAMESPACE file does not use a pattern to export functions then you should add an explicit entry to NAMESPACE for each R function you want publicly available.

In addition to exporting R bindings for C++ functions, the compileAttributes function can also generate a direct C++ interface to the functions using the Rcpp::interfaces attribute.

See Also
Rcpp::export, Rcpp::interfaces
cppFunction

Examples

### Not run:

```r
# Compile attributes for package in the current working dir
cppFunction()
```

### End(Not run)

---

cppFunction | Define an R Function with a C++ Implementation

**Description**

Dynamically define an R function with C++ source code. Compiles and links a shared library with bindings to the C++ function then defines an R function that uses `.Call` to invoke the library.

**Usage**

```r
cppFunction(code, depends = character(), plugins = character(),
             includes = character(), env = parent.frame(), rebuild = FALSE,
             showOutput = verbose, verbose = getOption("verbose"))
```

**Arguments**

- `code`: Source code for the function definition.
- `depends`: Character vector of packages that the compilation depends on. Each package listed will first be queried for an inline plugin to determine header files to include. If no plugin is defined for the package then a header file based the package's name (e.g. `PkgName.hpp`) will be included.
- `plugins`: Character vector of inline plugins to use for the compilation.
- `includes`: Character vector of user includes (inserted after the includes provided by `depends`).
- `env`: The environment in which to define the R function. May be `NULL` in which case the defined function can be obtained from the return value of `cppFunction`.
- `rebuild`: Force a rebuild of the shared library.
- `showOutput`: TRUE to print R CMD SHLIB output to the console.
- `verbose`: TRUE to print detailed information about generated code to the console.

**Details**

Functions defined using `cppFunction` must have return types that are compatible with `Rcpp::wrap` and parameter types that are compatible with `Rcpp::as`.

The shared library will not be rebuilt if the underlying code has not changed since the last compilation.
Value

An R function that uses .Call to invoke the underlying C++ function.

Note

You can also define R functions with C++ implementations using the sourceCpp function, which allows you to separate the C++ code into its own source file. For many use cases this is an easier and more maintainable approach.

See Also

sourceCpp, evalCpp

Examples

```r
## Not run:

cppFunction(
 'int fibonacci(const int x) {
   if (x == 0) return(0);
   if (x == 1) return(1);
   return (fibonacci(x - 1)) + fibonacci(x - 2);
 })

cppFunction(depends = "RcppArmadillo",
 'List fastlm(NumericVector yr, NumericMatrix Xr) {
   int n = Xr.nrow(), k = Xr.ncol();
   arma::mat X(Xr.begin(), n, k, false);
   arma::colvec y(yr.begin(), yr.size(), false);
   arma::colvec coef = arma::solve(X, y);
   arma::colvec resid = y - X*coef;
   double sig2 = arma::as_scalar(arma::trans(resid)*resid/(n-k) );
   arma::colvec stderrrest = arma::sqrt(
     sig2 * arma::diagvec(arma::inv(arma::trans(X)*X)));
   return List::create(Named("coefficients") = coef,
                       Named("stderr") = stderrrest
   );
 })

cppFunction(plugins=c("cpp11"),
 'int useCpp11() {
   auto x = 10;
   return x;
 })

 ## End(Not run)
```
Description

The Rcpp::depends attribute is added to a C++ source file to indicate that it has a compilation dependency on one or more other packages. For example:

```c
// [[Rcpp::depends(RcppArmadillo)]]
```

Arguments

... Packages which the source file depends on for compilation

Details

The Rcpp::depends attribute is used by the implementation of the `sourceCpp` function to correctly setup the build environment for `R CMD SHLIB`.

The include directories of the specified packages are added to the `CLINK_CPPFLAGS` environment variable. In addition, if the referenced package provides an inline plugin it is called to determine additional environment variables required to successfully build.

Note

The Rcpp::depends attribute is specified using a syntax compatible with the new generalized attributes feature of the C++11 standard. Note however that since this feature is not yet broadly supported by compilers it needs to be specified within a comment (see examples below).

See Also

`sourceCpp`

Examples

```r
## Not run:
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::depends(Matrix, RcppGSL)]]
## End(Not run)
```
**evalCpp**  
*Evaluate a C++ Expression*

**Description**

Evaluates a C++ expression. This creates a C++ function using `cppFunction` and calls it to get the result.

**Usage**

```r
evalCpp(code, depends = character(), includes = character(), rebuild = FALSE, showOutput = verbose, verbose = getOption("verbose"))
```

**Arguments**

- `code`: C++ expression to evaluate
- `names`: names of the macros we want to test
- `depends`: see `cppFunction`
- `includes`: see `cppFunction`
- `rebuild`: see `cppFunction`
- `showOutput`: see `cppFunction`
- `verbose`: see `cppFunction`

**Value**

The result of the evaluated C++ expression.

**Note**

The result type of the C++ expression must be compatible with Rcpp::wrap.

**See Also**

`sourceCpp`, `cppFunction`

**Examples**

```r
## Not run:
evalCpp( "__cplusplus" )
evalCpp( "std::numeric_limits<double>::max()" )
```
The `Rcpp::export` attribute is added to a C++ function definition to indicate that it should be made available as an R function. The `sourceCpp` and `compileAttributes` functions process the `Rcpp::export` attribute by generating the code required to call the C++ function from R.

**Arguments**

- **name**  
  Specify an alternate name for the generated R function (optional, defaults to the name of the C++ function if not specified).

**Details**

Functions marked with the `Rcpp::export` attribute must meet several conditions to be correctly handled:

1. Be defined in the global namespace (i.e. not within a C++ namespace declaration).
2. Have a return type that is either void or compatible with `Rcpp::wrap` and parameter types that are compatible with `Rcpp::as` (see sections 3.1 and 3.2 of the *Rcpp-introduction* vignette for more details).
3. Use fully qualified type names for the return value and all parameters. However, Rcpp types may appear without the namespace qualifier (i.e. `DataFrame` is okay as a type name but `std::string` must be specified fully).

If default argument values are provided in the C++ function definition then these defaults are also used for the exported R function. For example, the following C++ function:

```cpp
DataFrame readData(
    CharacterVector file,
    CharacterVector exclude = CharacterVector::create(),
    bool fill = true)
```

Will be exported to R as:

```r
function (file, exclude = character(0), fill = TRUE)
```

Note that C++ rules for default arguments still apply: they must occur consecutively at the end of the function signature and unlike R can’t rely on the values of other arguments.
Note

When a C++ function has export bindings automatically generated by the `compileAttributes` function, it can optionally also have a direct C++ interface generated using the `Rcpp::interfaces` attribute.

The `Rcpp::export` attribute is specified using a syntax compatible with the new generalized attributes feature of the C++11 standard. Note however that since this feature is not yet broadly supported by compilers it needs to be specified within a comment (see examples below).

See Also

`sourceCpp` and `compileAttributes`

Examples

```r
## Not run:
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return (fibonacci(x - 1)) + fibonacci(x - 2);
}

// [[Rcpp::export("convolveCpp")]]
NumericVector convolve(NumericVector a, NumericVector b) {
  int na = a.size(), nb = b.size();
  int nab = na + nb - 1;
  NumericVector xab(nab);

  for (int i = 0; i < na; i++)
    for (int j = 0; j < nb; j++)
      xab[i + j] += a[i] * b[j];

  return xab;
}

## End(Not run)
```

formals<->methods

Set the formal arguments of a C++ function
\textit{interfacesAttribute}

\textbf{Description}

Set the formal arguments of a C++ function

\textbf{Methods}

signature(fun = "C++Function") Set the formal arguments of a C++ function

\textit{Rcpp::interfaces Attribute}

\textbf{Description}

The \texttt{Rcpp::interfaces} attribute is added to a C++ source file to specify which languages to generate bindings for from exported functions. For example:

// [[Rcpp::interfaces(r, cpp)]]

\textbf{Arguments}

\ldots Interfaces to generate for exported functions within the source file. Valid values are \texttt{r} and \texttt{cpp}, and more than one interface can be specified.

\textbf{Details}

The \texttt{Rcpp::interfaces} attribute is used to determine which bindings to generate for exported functions. The default behavior if no \texttt{Rcpp::interfaces} attribute is specified is to generate only an R interface.

When \texttt{cpp} bindings are requested code is generated as follows:

1. Bindings are generated into a header file located in the \texttt{inst/include} directory of the package using the naming convention \texttt{PackageName_RcppExports.h}
2. If not already present, an additional header file named \texttt{PackageName.h} is also generated which in turn includes the Rcpp exports header.
   In the case that you already have a \texttt{PackageName.h} header for your package then you can manually add an include of the Rcpp exports header to it to make the exported functions available to users of your package.
3. The generated header file allows calling the exported C++ functions without any linking dependency on the package (this is based on using the \texttt{R_RegisterCCallable} and \texttt{R_GetCCallable} functions).
4. The exported functions are defined within a C++ namespace that matches the name of the package.

For example, an exported C++ function \texttt{foo} could be called from package \texttt{MyPackage} as follows:
// [[Rcpp::depends(MyPackage)]]
#include <MyPackage.h>

void foo() {
    MyPackage::bar();
}

The above example assumes that the sourceCpp function will be used to compile the code. If rather than that you are building a package then you don’t need to include the Rcpp::depends attribute, but instead should add an entry for the referenced package in the Depends and LinkingTo fields of your package’s DESCRIPTION file.

Note
If a file by the name of PackageName.h that wasn’t generated by compileAttributes already exists in the inst/include directory then it will not be overwritten (rather, an error will occur).
A static naming scheme for generated header files and namespaces is used to ensure consistent usage semantics for clients of exported cpp interfaces. Packages that wish to export more complex interfaces or additional C++ types are therefore typically better off not using this mechanism.
The Rcpp::interfaces attribute is specified using a syntax compatible with the new generalized attributes feature of the C++11 standard. Note however that since this feature is not yet broadly supported by compilers it needs to be specified within a comment (see examples below).

See Also
compileAttributes,Rcpp::export,Rcpp::depends

Examples
## Not run:
// [[Rcpp::interfaces(r, cpp)]]
## End(Not run)

### loadModule

#### Description
One or more calls to loadModule will be included in the source code for a package to load modules and optionally expose objects from them. The actual extraction of the module takes place at load time.

#### Usage
loadModule(module, what = , loadNow, env =)
loadModule

Arguments

module
The name of the C++ module to load. The code for the module should be in the same package as the R call to loadModule.

what
The objects to expose in the package’s namespace corresponding to objects in the module. By default, nothing is exposed. The special value TRUE says to load all the objects in the module that have syntactically standard R names (which all objects in a module will normally have). Otherwise, if supplied this should be a character vector, the elements being objects defined in the module. The vector can have a names attribute, in which case the non-empty names will be used to rename the objects; otherwise, the name of the object in the package namespace will be the same as the name in the C++ module.

loadNow, env
A logical flag to say whether the load actions should happen now, and the environment into which the objects should be inserted. When called from the source of a package, both of these arguments should usually be omitted. The value of loadNow will be set by checking the module’s status. At package installation time, the module cannot be started, in which case a load action (see setLoadAction) is scheduled to do the actual module load. The value of env will default to the package’s namespace.

Details

If the purpose of loading the module is to define classes based on C++ classes, see setRcppClass(), which does the necessary module loading for you.

When the module can be started (at namespace load time), the function Module() returns an environment with a description of the module’s contents. Function loadModule() saves this as a metadata object in the package namespace. Therefore multiple calls to loadModule() are an efficient way to extract different objects from the module.

Requesting an object that does not exist in the module produces a warning.

Since assignments from the call cannot take place until namespace loading time, any computations using the objects must also be postponed until this time. Use load actions (setLoadAction) and make sure that the load action is specified after the call to loadModule().

Value

If the load takes place, the module environment is returned. Usually however the function is called for its side effects.

Note

This function requires version 2.15.0 of R or later, in order to use load actions, introduced in that version. See the note in the help page for setRcppClass() for details.

Author(s)

John Chambers
See Also

setRcppClass() to avoid the explicit call.
loadRcppModules() for a shotgun procedure to load all modules.

Examples

## Not run:
loadModule("yada", TRUE) # load all the objects from module "yada"

## End(Not run)

loadRcppModules

Loads Rcpp modules on package startup

Description

Function to simplify loading Rcpp modules contained in a package. This function must be called from the .onLoad function of a package. It uses the RcppModules field of the package DESCRIPTION file to query the names of the modules that the package should export, loads each module, and populate each module into the package NAMESPACE.

Usage

loadRcppModules(direct=TRUE)

Arguments

direct if TRUE the content of the module is exposed in the namespace. Otherwise, the module is exposed.

See Also

populate

Module

Retrieves an Rcpp module

Description

Retrieves an Rcpp module from a dynamic library, usually associated with a package.

Usage

Module(module, PACKAGE = , where = , mustStart = )
Module-class

Arguments

module  Name of the module, as declared in the RCPP_MODULE macro internally
PACKAGE Passed to getNativeSymbolInfo
where When the module is loaded, S4 classes are defined based on the internal classes. This argument is passed to setClass
mustStart TODO

Value

An object of class Module collecting functions and classes declared in the module.

---

Module-class  Rcpp modules

Description

Collection of internal c++ functions and classes exposed to R

Objects from the Class

modules are created by the link(Module) function

Methods

$ signature(x = "Module"): extract a function or a class from the module.

prompt signature(object = "Module"): generates skeleton of a documentation for a Module.

show signature(object = "Module"): summary information about the module.

initialize signature(.Object = "Module"): ...

See Also

The Module function
Description

The `Rcpp::plugins` attribute is added to a C++ source file to specify the inline plugins that should be used in the compilation.

```
// [[Rcpp::plugins(plugin1, plugin2)]]
```

Arguments

... Plugins to add to the compilation.

Details

Plugins must be registered using the `registerPlugin` function.

When included within a `sourceCpp` translation unit, the configuration-related fields of the plugin (e.g. `env` and `LinkingTo`) are utilized, however the code-generation fields (e.g. `includes` and `body`) are not.

Note

`Rcpp` includes a built-in `cpp11` plugin that adds the flags required to enable C++11 features in the compiler.

See Also

`registerPlugin`

Examples

```r
## Not run:

// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
int useCpp11() {
    auto x = 10;
    return x;
}

## End(Not run)```
**populate**

*Populates a namespace or an environment with the content of a module*

---

**Description**

Populates a namespace or an environment with the content of a module

**Usage**

`populate(module, env)`

**Arguments**

- `module` Rcpp module
- `env` environment or namespace

---

**Rcpp.package.skeleton**

*Create a skeleton for a new package depending on Rcpp*

---

**Description**

Rcpp.package.skeleton automates the creation of a new source package that intends to use features of Rcpp.

It is based on the `package.skeleton` function which it executes first.

**Usage**

```r
def Rcpp.package.skeleton(name = "anRpackage", list = character(),
environment = .GlobalEnv, path = ".", force = FALSE,
namespace = TRUE, code_files = character(), cpp_files = character(),
example_code = TRUE, attributes = FALSE, module = FALSE,
author = "Who wrote it",
maintainer = if(missing( author)) "Who to complain to" else author,
email = "yourfault@somewhere.net",
license = "What Licence is it under ?")
```

**Arguments**

- `name` See `package.skeleton`
- `list` See `package.skeleton`
- `environment` See `package.skeleton`
- `path` See `package.skeleton`
- `force` See `package.skeleton`
namespace See package.skeleton
code_files See package.skeleton
cpp_files A character vector with the paths to C++ source files to add to the package.
example_code If TRUE, example c++ code using Rcpp is added to the package.
attributes If TRUE, example code makes use of Rcpp attributes.
module If TRUE, an example Module is added to the skeleton.
author Author of the package.
maintainer Maintainer of the package.
email Email of the package maintainer.
license License of the package.

Details
In addition to package.skeleton:
The ‘DESCRIPTION’ file gains a Depends line requesting that the package depends on Rcpp and a LinkingTo line so that the package finds Rcpp header files.
The ‘NAMESPACE’, if any, gains a useDynLib directive.
The ‘src’ directory is created if it does not exists and a ‘Makevars’ file is added setting the environment variables ‘PKG_LIBS’ to accomodate the necessary flags to link with the Rcpp library.
If cpp_files are provided then they will be copied to the ‘src’ directory.
If the example_code argument is set to TRUE, example files ‘rcpp_hello_world.h’ and ‘rcpp_hello_world.cpp’ are also created in the ‘src’. An R file ‘rcpp_hello_world.R’ is expanded in the ‘R’ directory, the rcpp_hello_world function defined in this files makes use of the C++ function ‘rcpp_hello_world’ defined in the C++ file. These files are given as an example and should eventually by removed from the generated package.
If the attributes argument is TRUE, then rather than generate the example files as described above, a single ‘rcpp_hello_world.cpp’ file is created in the ‘src’ directory and it’s attributes are compiled.
If the module argument is TRUE, a sample Rcpp module will be generated as well.

Value
Nothing, used for its side effects

References
Read the Writing R Extensions manual for more details.

Once you have created a source package you need to install it: see the R Installation and Administration manual, INSTALL and install.packages.

See Also
package.skeleton
Examples

```r
## Not run:
# simple package
Rcpp.package.skeleton( "foobar" )

# package using attributes
Rcpp.package.skeleton( "foobar", attributes = TRUE )

# package with a module
Rcpp.package.skeleton( "testmod", module = TRUE )

# the Rcpp-package vignette
vignette( "Rcpp-package" )

# the Rcpp-modules vignette for information about modules
vignette( "Rcpp-modules" )

## End(Not run)
```

---

Description

Unit tests results for package Rcpp.

Unit tests are run automatically at build time and reports are included in the ‘doc’ directory as html or text.

Details

See Also

Examples

```r
# unit tests are in the unitTests directory of the package
list.files( system.file("unitTests", package = "Rcpp" ),
          pattern = "^runit", full = TRUE )

# trigger the unit tests preparation, follow printed instructions
# on how to run them
## Not run:
source( system.file("unitTests", "runTests.R", package = "Rcpp" ) )

## End(Not run)
```
registerPlugin  

Register an inline plugin

Description

Register an inline plugin for use with sourceCpp or cppFunction. Inline plugins are functions that return a list with additional includes, environment variables, and other compilation context.

Usage

registerPlugin(name, plugin)

Arguments

name  
Name of the inline plugin

plugin  
Inline plugin function

Details

Plugins can be added to sourceCpp compilations using the Rcpp::plugins attribute.

See Also

Rcpp::plugins

setRcppClass  

Create a Class Extending a C++ Class

Description

A class is defined that includes the fields and methods of a C++ class defined, usually in this package. The R class can include new methods and fields, such as for prototyping new computations for the C++ class.

Usage

setRcppClass(Class, CppClass, module, fields = list(), contains = , methods = , saveAs = Class, where = )
setRcppClass

Arguments

Class
The name for the new class.

CppClass
The C++ class defined in the C++ code for the package that this class extends. By default, the same as Class.

module
The Rcpp module in which the class is defined. The module does not have to be loaded separately; setRcppClass() will arrange to load the module.

fields, contains, methods
Additional fields, superclasses and method definitions in R that extend the C++ class. These arguments are passed on to setRefClass(). See Details for the implementation of methods.

saveAs
Save a generator object for the class in the package’s namespace under this name. By default, the generator object has the name of the class. To avoid saving any generator object, supply this argument as NULL.

where
The environment in which to save the class definition. By default, will be the namespace of the package in which the setRcppClass() call is included.

... Arguments, if any, to pass on to setRefClass().

Details

The call to this function normally appears in the source code for a package. It generates a load action that loads the specified module and extracts the C++ class definition specified.

R code can define new fields and methods for the class, typically as prototypes for possible future C++ implementation. Methods for the R class can refer to methods and fields defined in C++ for the C++ class. These will be mapped into the C++ equivalents by R code generated by setRcppClass. The C++ code may not deal as well as R with incompatible argument types and lengths. Segmentation faults are a definite possibility. If that’s a problem, you should define methods in R that check for legal data types and values.

The fields and methods defined can include overriding C++ fields or methods. Keep in mind, however, that R methods can refer to C++ fields and methods, but not the reverse. If you override a C++ field or method, you essentially need to revise all code that refers to that field or method. Otherwise, the C++ code will continue to use the old C++ definition.

Value

A generator for the new class.

Note

This function and function loadModule() require version 2.15.0 of R or later, in order to use load actions, introduced in that version.

A subtle way this can fail is by somehow loading a legitimate binary version of your package (installed under a valid version of R) into a session with an older R. In this case the load actions created in the binary package will simply not be called. None of the modules will be loaded and none of the classes created.

If your symptom is that classes or other objects from modules don’t exist, check the R version.
Author(s)

John Chambers

Examples

```r
## Not run:
setRcppClass("World",
  module = "yada",
  fields = list(more = "character"),
  methods = list(
    test = function(what) message("Testing: ", what, " ; ", more),
    saveAs = "genWorld"
  )
)
## End(Not run)
```

sourceCpp

**Source C++ Code from a File or String**

Description

sourceCpp parses the specified C++ file or source code and looks for functions marked with the \texttt{Rcpp::export} attribute and RCPP_MODULE declarations. A shared library is then built and its exported functions and Rcpp modules are made available in the specified environment.

Usage

```r
sourceCpp(file = ",", code = NULL, env = globalenv(),
  rebuild = FALSE, showOutput = verbose,
  verbose =getOption("verbose"))
```

Arguments

- **file**: A character string giving the path name of a file.
- **code**: A character string with source code. If supplied, the code is taken from this string instead of a file.
- **env**: Environment where the R functions and modules should be made available.
- **rebuild**: Force a rebuild of the shared library.
- **showOutput**: TRUE to print R CMD SHLIB output to the console.
- **verbose**: TRUE to print detailed information about generated code to the console.
sourceCpp

Details

If the code parameter is provided then the file parameter is ignored.

Functions exported using sourceCpp must meet several conditions, including being defined in the global namespace and having return types that are compatible with Rcpp::wrap and parameter types that are compatible with Rcpp::as. See the Rcpp::export documentation for more details.

Rcpp Modules will be automatically loaded into the specified environment using the Module function. The name of the loaded module object will be the same as the name specified in the RCPP_MODULE declaration.

If the source file has compilation dependencies on other packages (e.g. Matrix, RcppArmadillo) then an Rcpp::depends attribute should be provided naming these dependencies.

It’s possible to embed chunks of R code within a C++ source file by including the R code within a block comment with the prefix of /*** R. For example:

/*** R

# Call the fibonacci function defined in C++
fibonacci(10)

*/

Multiple R code chunks can be included in a C++ file. R code is sourced after the C++ compilation is completed so all functions and modules will be available to the R code.

Value

Returns (invisibly) a list with two elements:

functions Names of exported functions
modules Names of Rcpp modules

Note

The sourceCpp function will not rebuild the shared library if the source file has not changed since the last compilation.

The sourceCpp function is designed for compiling a standalone source file whose only dependencies are R packages. If you are compiling more than one source file or have external dependencies then you should create an R package rather than using sourceCpp. Note that the Rcpp::export attribute can also be used within packages via the compileAttributes function.

If you are sourcing a C++ file from within the src directory of a package then the package's LinkingTo dependencies, inst/include, and src directories are automatically included in the compilation.

If no Rcpp::export attributes or RCPP_MODULE declarations are found within the source file then a warning is printed to the console. You can disable this warning by setting the rcpp.warnNoExports option to FALSE.
See Also

\[ \text{Rcpp::export, Rcpp::depends, cppFunction, evalCpp} \]

Examples

## Not run:

sourceCpp("fibonacci.cpp")

sourceCpp(code='
#include <Rcpp.h>

// [[Rcpp::export]]
int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return (fibonacci(x - 1)) + fibonacci(x - 2);
}
')

## End(Not run)
Index

*Topic classes
  C++Class-class, 4
  C++Constructor-class, 5
  C++Field-class, 5
  C++Function-class, 6
  C++Object-class, 7
  C++OverloadedMethods-class, 7
  Module-class, 19
  setRcppClass, 24

*Topic interface
  loadRcppModules, 18
  Rcpp-package, 3

*Topic manip
  populate, 21

*Topic methods
  .DollarNames-methods, 4
  formals<-methods, 14

*Topic programming
  loadModule, 16
  Module, 18
  Rcpp-package, 3
  Rcpp.package.skeleton, 21
  RcppUnitTests, 23
  .DollarNames,ANY-method
    (.DollarNames-methods), 4
  .DollarNames,C++Object-method
    (.DollarNames-methods), 4
  .DollarNames,Module-method
    (.DollarNames-methods), 4
  .DollarNames-methods, 4
  .environment, 5, 7
  $,C++Class-method(C++Class-class), 4
  $,C++Object-method(C++Object-class), 7
  $,Module-method(Module-class), 19
  $<-,C++Object-method(C++Object-class), 7

  areMacrosDefined (evalCpp), 12

  C++Class, 6

  C++Class-class, 4
  C++Constructor, 4
  C++Constructor-class, 5
  C++Field, 4
  C++Field-class, 5
  C++Function-class, 6
  C++Object-class, 7
  C++OverloadedMethods, 4
  C++OverloadedMethods-class, 7
  compileAttributes, 8, 13, 14, 16, 27
  cppFunction, 9, 12, 24, 28

  dependsAttribute, 11

  environment, 5, 7, 9
  envRefClass, 5, 7
  evalCpp, 10, 12, 28
  exportAttribute, 13

  formals<-,C++Function-method
    (formals<-methods), 14
  formals<--methods, 14
  function, 6

  getNativeSymbolInfo, 19

  initialize,Module-method
    (Module-class), 19

  inline plugin, 9, 11, 24
  inline plugins, 8, 9, 20
  INSTALL, 22
  install.packages, 22

  interfacesAttribute, 15

  loadModule, 16, 25
  loadRcppModules, 18, 18

  Module, 4, 17, 18, 19, 22, 27
  Module-class, 19

  OptionalFunction, 6
package.skeleton, 21, 22
pluginsAttribute, 20
populate, 18, 21
PossibleMethod, 6
prompt, Module-method (Module-class), 19

Rcpp (Rcpp-package), 3
Rcpp-package, 3
Rcpp.package.skeleton, 21
Rcpp::depends, 16, 27, 28
Rcpp::export, 8, 16, 26–28
Rcpp::interfaces, 8, 14
Rcpp::plugins, 24
RcppClass-class (setRcppClass), 24
RcppUnitTests, 23
refClass, 5, 7
refObject, 5, 7
registerPlugin, 20, 24

setClass, 19
setLoadAction, 17
setRcppClass, 17, 18, 24
setRefClass, 25
show, C++Class-method (C++Class-class), 4
show, C++Function-method (C++Function-class), 6
show, C++Object-method (C++Object-class), 7
show, Module-method (Module-class), 19
sourceCpp, 10–14, 20, 24, 26