Package ‘inline’
May 31, 2021

Version 0.3.19
Date 2021-05-25
Title Functions to Inline C, C++, Fortran Function Calls from R
Author Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel,
        Romain Francois, Karline Soetaert, Johannes Ranke
Maintainer Dirk Eddelbuettel <edd@debian.org>
Imports methods
Suggests Rcpp (>= 0.11.0), tinytest
Description Functionality to dynamically define R functions and S4 methods
        with ‘inlined’ C, C++ or Fortran code supporting the .C and .Call calling
        conventions.
License LGPL
Copyright Oleg Sklyar, 2005-2010 and other authors per their commits
LazyLoad yes
URL https://github.com/eddelbuettel/inline,
        https://dirk.eddelbuettel.com/code/inline.html
BugReports https://github.com/eddelbuettel/inline/issues
NeedsCompilation no
Repository CRAN
Date/Publication 2021-05-31 08:40:02 UTC

R topics documented:

    inline-package                                    2
    cfunction                                         2
    cxxfunction                                       7
    getDynLib-methods                                9
    package.skeleton-methods                          9
    plugins                                           10
    utilities                                         11

Index                  15
inline-package  

*Functions to Inline C, C++, Fortran Function Calls from R*

**Description**

Functionality to dynamically define R functions and S4 methods with 'inlined' C, C++ or Fortran code supporting the .C and .Call calling conventions.

**Maintainer**

Dirk Eddelbuettel <edd@debian.org>

**Author(s)**

Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel, Romain Francois, Karline Soetaert, Johannes Ranke

**See Also**

cfunction, cxxfunction

cfunction  

*Inline C, C++, Fortran function calls from R*

**Description**

Functionality to dynamically define R functions and S4 methods with in-lined C, C++ or Fortran code supporting .C and .Call calling conventions.

**Usage**

cfunction(sig=character(), body=character(), includes=character(),
otherdefs=character(),
language=c("C++", "C", "Fortran", "F95", "ObjectiveC", "ObjectiveC++"),
verbose=FALSE,
convention=c(".Call", ".C", ".Fortran"),
Rcpp=FALSE,
cppargs=character(), cxxargs=character(), libargs=character(),
dim=NULL, implicit=NULL, module=NULL, name=NULL)

## S4 methods for signatures

# f='character', sig='list', body='list'
# f='character', sig='character', body='character'

setCMethod(f, sig, body, ...)
cfunction

## Further arguments:
# setCMetho(d(f, sig, body, includes="", otherdefs="", cpp=TRUE,
# verbose=FALSE, where=topenv(.GlobalEnv), ...

Arguments

**f**
A single character value if sig and body are character vectors or a character vector of the same length and the length of sig or body with the name(s) of methods to create.

**sig**
A match of formal argument names for the function with the character-string names of corresponding classes. Alternatively, a named list of such character vectors. The names of the list elements will be used as function names (see example). If sig is not a list, the function name used in the code can be specified by the name argument.

**body**
A character vector with C, C++ or Fortran code omitting function declaration (only the body, i.e. in case of C starting after the function opening curly bracket and ending before the closing curly bracket, brackets excluded). In case of setCMethod with signature list – a list of such character vectors.

**includes**
A character vector of additional includes and preprocessor statements etc that will be put between the R includes and the user function(s).

**otherdefs**
A characted vector with the code for any further definitions of functions, classes, types, forward declarations, namespace usage clauses etc which is inserted between the includes and the declarations of the functions defined in sig.

**language**
A character value that specifies the source language of the inline code. The possible values for language include all those supported by R CMD SHLIB on any platform, which are currently C, C++, Fortran, F95, ObjectiveC and ObjectiveC++; they may not all be supported on your platform. One can specify the language either in full as above, or using any of the following case insensitive shortened forms: c, cpp, c++, f, f95, objc, objcpp, objc++. Defaults to C++.

**verbose**
If TRUE prints the compilation output, the source code of the resulting program and the definitions of all declared methods. If FALSE, the function is silent, but it prints compiler warning and error messages and the source code if compilation fails.

**convention**
Which calling convention to use? See the Details section.

**Rcpp**
If TRUE adds inclusion of Rcpp.h to includes, also queries the Rcpp package about the location of header and library files and sets environment variables PKG_CXXFLAGS and PKG_LIBS accordingly so that the R / C++ interface provided by the Rcpp package can be used. Default value is FALSE.

**cppargs**
Optional character vector of tokens to be passed to the compiler via the PKG_CXXFLAGS environment variable. Elements should be fully formed as for example c("-I/usr/local/lib/foo","-DDEBUG") and are passed along verbatim.

**cxxargs**
Optional character vector of tokens to be passed to the compiler via the PKG_CXXFLAGS environment variable. Elements should be fully formed as for example c("-I/usr/local/lib/foo","-DDEBUG") and are passed along verbatim.
### cfunction

**libargs**
Optional character vector of tokens to be passed to the compiler via the `PKG_LIBS` environment variable. Elements should be fully formed as for example `c("-L/usr/local/lib/foo -lfoo", "--lpthread")` and are passed along verbatim.

**dim**
Optional character vector defining the dimensionality of the function arguments. Of same length as `sig`. Fortran or F95 only.

**implicit**
A character vector defining the implicit declaration in Fortran or F95; the default is to use the implicit typing rules for Fortran, which is integer for names starting with the letters I through N, and real for names beginning with any other letter. As R passes double precision, this is not the best choice. Safest is to choose `implicit = "none"` which will require all names in the subroutine to be explicitly declared.

**module**
Name(s) of any modules to be used in the Fortran or F95 subroutine.

**name**
Function name to be used in the code. Only used if `sig` is not a list. This is useful if the DLL created is to be used in conjunction with the `ode` function of the deSolve package.

... Reserved.

### Details

To declare multiple functions in the same library one can use `setCMethod` supplying lists of signatures and implementations. In this case, provide as many method names in `f` as you define methods. Avoid clashes when selecting names of the methods to declare, i.e. if you provide the same name several times you must ensure that signatures are different but can share the same generic!

The source code in the body should not include the header or "front-matter" of the function or the close, e.g. in C or C++ it must start after the C-function opening curly bracket and end before the C-function closing curly bracket, brackets should not be included. The header will be automatically generated from the R-signature argument. Arguments will will carry the same name as used in the signature, so avoid variable names that are not legal in the target language (e.g. names with dots).

C/C++: If `convention == ".Call"` (the default), the `.Call` mechanism is used and its result is returned directly as the result of the call of the generated function. As the last line of the generated C/C++ code a `return R_NilValue;` is added in this case and a warning is generated in case the user has forgotten to provide a return value. To suppress the warning and still return NULL, add `return R_NilValue;` explicitly.

Special care is needed with types, memory allocation and protection – exactly the same as if the code was not inline: see the Writing R Extension manual for information on `.Call`.

If `convention == ".C"` or `convention == ".Fortran"`, the `.C` or `.Fortran` mechanism respectively is used, and the return value is a list containing all arguments.

Attached R includes include `R.h` for `.C`, and additionally `Rdefines.h` and `R_ext\Error.h` for `.Call`.

### Value

If `sig` is a single character vector, `cfunction` returns a single `function`; if it is a list, it returns a list of functions.

`setCMethod` declares new methods with given names and signatures and returns invisible `NULL`.
**Author(s)**
Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel

**See Also**
*Foreign Function Interface*

**Examples**

```r
x <- as.numeric(1:10)
n <- as.integer(10)

## Not run:
## A simple Fortran example - n and x: assumed-size vector
code <- "
integer i
  do 1 i=1, n(1)
    1 x(i) = x(i)**3
"
cubefn <- cfunction(signature(n="integer", x="numeric"), code, convention=".Fortran")
print(cubefn)
cubefn(n, x)$x

## Same Fortran example - now n is one number
code2 <- "
integer i
  do 1 i=1, n
    1 x(i) = x(i)**3
"
cubefn2 <- cfunction(signature(n="integer", x="numeric"), implicit = "none",
                      dim = c("", "(*)"), code2, convention=".Fortran")
cubefn2(n, x)$x

## Same in F95, now x is fixed-size vector (length = n)
code3 <- "%x = x*x*x"
cubefn3 <- cfunction(sig = signature(n="integer", x="numeric"), implicit = "none",
                      dim = c("", "(n)"), code3, language="F95")
cubefn3(20, 1:20)
print(cubefn3)

## Same example in C
code4 <- "
int i;
  for (i = 0; i < *n; i++)
    x[i] = x[i]*x[i]*x[i];
"
cubefn4 <- cfunction(signature(n="integer", x="numeric"), code4, language = "C", convention = ".C")
cubefn4(20, 1:20)
```
## Give the function in the source code a name

```r
cubefn5 <- cfunction(signature(n="integer", x="numeric"), code4, language = "C", convention = ".C", 
  name = "cubefn")

code(cubefn5)

## End(Not run)
```

## use of a module in F95

```r
modct <- "module modcts
  double precision, parameter :: pi = 3.14159265358979
  double precision, parameter :: e = 2.71828182845905
end"

getcode <- "x(1) = pi
  x(2) = e"

cgetcts <- cfunction(getconstants, module = "modcts", implicit = "none",
  includes = modct, sig = c(x = "double"), dim = c("(2)"), language = "F95")

cgetcts(x = 1:2)

print(cgetcts)
```

## Use of .C convention with C code

## Defining two functions, one of which calls the other

```r
sigSq <- signature(n="integer", x="numeric")
codeSq <- "
  for (int i=0; i < *n; i++) {
    x[i] = x[i]*x[i];
  }
"

sigQd <- signature(n="integer", x="numeric")
codeQd <- "
  squarefn(n, x);
  squarefn(n, x);
"

fns <- cfunction( list(squarefn=sigSq, quadfn=sigQd), 
  list(codeSq, codeQd), 
  convention=".C")

squarefn <- fns[["squarefn"]]
quadfn <- fns[["quadfn"]]

squarefn(n, x)$x
quadfn(n, x)$x
```

## Alternative declaration using 'setCMethod'

```r
setCMethod(c("squarefn", "quadfn"), list(sigSq, sigQd), 
  list(codeSq, codeQd), convention=".C")

squarefn(n, x)$x
quadfn(n, x)$x
```

## Use of .Call convention with C code
## Multiplying each image in a stack with a 2D Gaussian at a given position

code <- 
```c
SEXP res;
  int nprotect = 0, nx, ny, nz, x, y;
  PROTECT(res = Rf_duplicate(a)); nprotect++;
  nx = INTEGER(GET_DIM(a))[0];
  ny = INTEGER(GET_DIM(a))[1];
  nz = INTEGER(GET_DIM(a))[2];
  double sigma2 = REAL(s)[0] * REAL(s)[0], d2;
  double cx = REAL(centre)[0], cy = REAL(centre)[1], *data, *rdata;
  for (int im = 0; im < nz; im++) {
    data = &(REAL(a)[im*nx*ny]); rdata = &(REAL(res)[im*nx*ny]);
    for (x = 0; x < nx; x++)
      for (y = 0; y < ny; y++) {
        d2 = (x-cx)*(x-cx) + (y-cy)*(y-cy);
        rdata[x + y*nx] = data[x + y*nx] * exp(-d2/sigma2);
      }
  }
  UNPROTECT(nprotect);
  return res;
```

funx <- cfunction(signature(a="array", s="numeric", centre="numeric"), code)

x <- array(runif(50*50), c(50,50,1))
res <- funx(a=x, s=10, centre=c(25,15))
if (interactive()) image(res[,1])

## Same but done by registering an S4 method

setCMethod("funy", signature(a="array", s="numeric", centre="numeric"), code, verbose=TRUE)

res <- funy(x, 10, c(35,35))
if (interactive()) { x11(); image(res[,1]) }

---

cxxfunction

**inline C++ function**

### Description

Functionality to dynamically define an R function with inlined C++ code using the `.Call` calling convention.

The `rcpp()` wrapper sets the plugin to the “Rcpp” value suitable for using Rcpp.

### Usage

```r
cxxfunction(sig = character(), body = character(),
           plugin = "default", includes = ",",
           settings = getPlugin(plugin), ..., verbose = FALSE)
rcpp(..., plugin=“Rcpp”)
```
Arguments

- **sig**: Signature of the function. A named character vector.
- **body**: A character vector with C++ code to include in the body of the compiled C++ function.
- **plugin**: Name of the plugin to use. See **getPlugin** for details about plugins.
- **includes**: User includes, inserted after the includes provided by the plugin.
- **settings**: Result of the call to the plugin.
- **...**: Further arguments to the plugin.
- **verbose**: verbose output

Value

A function

See Also

cfunction

Examples

```r
## Not run:
# default plugin
fx <- cxxfunction(signature(x = "integer", y = "numeric"),
                 "return ScalarReal(INTEGER(x)[0] * REAL(y)[0]);")
fx(2L, 5)

# Rcpp plugin
if (requireNamespace("Rcpp", quietly=TRUE)) {
  fx <- cxxfunction(signature(x = "integer", y = "numeric"),
                    "return wrap(as<int>(x) * as<double>(y));",
                    plugin = "Rcpp"
  )
  fx(2L, 5)
}

## equivalent shorter form using rcpp()
fx <- rcpp(signature(x = "integer", y = "numeric"),
           "return wrap(as<int>(x) * as<double>(y));")
}

# RcppArmadillo plugin
if (requireNamespace(RcppArmadillo)) {
  fx <- cxxfunction(signature(x = "integer", y = "numeric"),
                    "int dim = as<int>(x);
                    arma::mat z = as<double>(y) * arma::eye<arma::mat>(dim, dim);
                    return wrap(arma::accu(z));",
                    plugin = "RcppArmadillo"
  )
  fx(2L, 5)
}
```
getDynLib-methods

Retrieve the dynamic library (or DLL) associated with a package of a function generated by cfunction

Description

The getDynLib function retrieves the dynamic library (or DLL) associated with a package or with a function generated by cfunction.

Methods

signature(x = "CFunc") Retrieves the dynamic library associated with the function generated by cfunction. The library is dynamically loaded if necessary.

signature(x = "CFuncList") Retrieves the dynamic library associated with a set of functions generated by cfunction. The library is dynamically loaded if necessary.

signature(x = "character") Retrieves the dynamic library of the given name. This typically refers to package names, but can be any name of the list returned by getLoadedDLLs.

See Also

getLoadedDLLs, dyn.load

Examples

## Not run:
getDynLib( "base" )

f <- cfunction( signature() , "return R_NilValue ;" )
getDynLib( f )

## End(Not run)

package.skeleton-methods

Generate the skeleton of a package

Description

Generate the skeleton of a package.
Methods

signature(name = "ANY", list = "ANY") Standard method. See package.skeleton
signature(name = "character", list = "CFunc") Method for a single generated by cfunction
or cxxfunction
signature(name = "character", list = "CFuncList") Method for a set functions generated by
cfunction or cxxfunction

Examples

## Not run:

fx <- cxxfunction(signature(x = "integer", y = "numeric"),
    "return ScalarReal( INTEGER(x)[0] * REAL(y)[0];"),
package.skeleton("foo", fx)

functions <- cxxfunction(list(ff = signature(),
    gg = signature(x = "integer", y = "numeric")),
    c("return R_NilValue ;",
    "return ScalarReal(INTEGER(x)[0] * REAL(y)[0]);"),
package.skeleton("foobar", functions)

## End(Not run)

---

plugins Plugin system for cxxfunction

Description

cxxfunction uses a plugin system to assembly the code that it compiles. These functions allow to
register and get plugins by their name.

Usage

getPlugin(name, ...)
registerPlugin(name, plugin)

Arguments

name name of the plugin.
...
Further arguments to pass to the plugin.
plugin plugin function.
Details

plugins are functions that return a list with:

includes mandatory. It is included at the top of the compiled file by \texttt{cxxfunction}.

body optional. A function that takes one argument (the body of the \texttt{c++} function) and returned a modified version of the body. The "Rcpp" plugin uses this to surround the code with the \texttt{BEGIN_RCPP} and \texttt{END_RCPP} macros.

LinkingTo optional. Character vector containing the list of packages that the code needs to link to. This adds the include path of the given packages. The "Rcpp" and "RcppArmadillo" plugins use this.

eqv optional. Named list of environment variables. For example, the "Rcpp" plugin uses this to add Rcpp user library to the \texttt{PKG_LIBS} environment variable.

Plugins can be manually registered using the \texttt{registerPlugin} function. Alternatively, a package may supply an inline plugin implicitly by defining a function called \texttt{inlineCxxPlugin}, which does not necessarily need to be exported from the namespace of the package.

Known packages implementing this scheme include \texttt{Rcpp} and \texttt{RcppArmadillo}.

Value

\texttt{getPlugin} retrieves the plugin and invokes it with the \ldots arguments.
\texttt{registerPlugin} does not return anything.

See Also

\texttt{cxxfunction}

Examples

```r
## Not run:
getPlugin( "Rcpp" )
## End(Not run)
```

Description

\texttt{moveDLL} moves the DLL used by a compiled function to a user defined location.

\texttt{writeCFunc} saves a \texttt{CFunc} object after the DLL has been moved to the desired location using \texttt{moveDLL}.

\texttt{readCFunc} reads a \texttt{CFunc} object that has been saved using \texttt{writeCFunc}.

The print and code methods respectively print the entire object or only the code parts.
Usage

moveDLL(x, ...)
## S4 method for signature 'CFunc'
moveDLL(x, name, directory, unload = FALSE, overwrite = FALSE, verbose = FALSE)

writeCFunc(x, file)
readCFunc(file)

## S4 method for signature 'CFunc'
print(x)
## S4 method for signature 'CFuncList'
print(x)

## S4 method for signature 'CFunc'
code(x, linenumbers = TRUE)
## S4 method for signature 'CFuncList'
code(x, linenumbers = TRUE)

Arguments

x A CFunc or CFuncList object as created by cfunction
name The base of the file name that the DLL should be moved to. The file name
      extension will depend on the operating system used
directory The directory that the DLL should be written to
unload In case the new path constructed from name and directory points to a loaded
         DLL, should we unload it?
overwrite In case there is a file at the new path constructed from name and directory
         should we overwrite that file?
verbose Should we print a message stating where the DLL was copied if the operation
         was successful?
file The file path for writing and reading the object generated by cfunction. Con-
      sider using a file name extension like .rda or .RData to indicate that this is a
      serialized R object.
linenumber If TRUE all code lines will be numbered.
... May be used in future methods

Details

If you move the DLL to a user defined location with moveDLL, this will keep an on-disk copy of
the DLL which will prevent it from being lost at session termination - unless written to the session
tempdir. Saving and reloading the CFunc object with standard tools like save or saveRDS will
still loose the pointer to the DLL. However, when the DLL has been moved using moveDLL, CFunc
objects can be saved by writeCFunc and restored by readCFunc.
Value

Function `readDynLib` returns a `CFunc` object.

Function `writeDynLib` returns the name of the `.CFunc` file that was created.

Note

- The code of a `CFunc` or `CFuncList` object `x` can be extracted (rather than printed), using:
  ```
  x@code
  ```
- To write the code to a file (here called "fn"), without the new-line character "\n":
  ```
  write (strsplit(x,"\n")[[1]],file = "fn")
  ```

Author(s)

Karline Soetaert and Johannes Ranke

See Also

`getDynLib`

Examples

```r
x <- as.numeric(1:10)
n <- as.integer(10)

code <- "
  integer i
  do i=1, n(1)
  1 x(i) = x(i)**3
"
cubefn <- cfunction(signature(n="integer", x="numeric"), code,
  convention=".Fortran")

cubefn(n, x)$x
```
cfn(3, 1:3)$x

## End(Not run)
## Index

<table>
<thead>
<tr>
<th>Category</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>* file</td>
<td>cfunction, 2</td>
</tr>
<tr>
<td></td>
<td>utilities, 11</td>
</tr>
<tr>
<td>* inline function call</td>
<td>cfunction, 2</td>
</tr>
<tr>
<td>* interface</td>
<td>cxxfunction, 7</td>
</tr>
<tr>
<td></td>
<td>plugins, 10</td>
</tr>
<tr>
<td>* methods</td>
<td>getDynLib-methods, 9</td>
</tr>
<tr>
<td></td>
<td>package.skeleton-methods, 9</td>
</tr>
<tr>
<td>* package</td>
<td>inline-package, 2</td>
</tr>
<tr>
<td>* programming</td>
<td>cxxfunction, 7</td>
</tr>
<tr>
<td></td>
<td>plugins, 10</td>
</tr>
<tr>
<td></td>
<td>.C, 4</td>
</tr>
<tr>
<td></td>
<td>.Call, 4, 7</td>
</tr>
<tr>
<td></td>
<td>.Fortran, 4</td>
</tr>
<tr>
<td></td>
<td>cfunction, 2, 8–10, 12</td>
</tr>
<tr>
<td></td>
<td>code (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>code, CFunc-method (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>code, CFuncList-method (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>code, character-method (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>code-methods (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>cxxfunction, 2, 7, 10, 11</td>
</tr>
<tr>
<td></td>
<td>dyn.load, 9</td>
</tr>
<tr>
<td></td>
<td>Foreign, 5</td>
</tr>
<tr>
<td></td>
<td>function, 4</td>
</tr>
<tr>
<td></td>
<td>getDynLib, 13</td>
</tr>
<tr>
<td></td>
<td>getDynLib (getDynLib-methods), 9</td>
</tr>
<tr>
<td></td>
<td>getDynLib, CFunc-method (getDynLib-methods), 9</td>
</tr>
<tr>
<td></td>
<td>getDynLib, CFuncList-method (getDynLib-methods), 9</td>
</tr>
<tr>
<td></td>
<td>getDynLib, character-method (getDynLib-methods), 9</td>
</tr>
<tr>
<td></td>
<td>getDynLib-methods, 9</td>
</tr>
<tr>
<td></td>
<td>getLoadedDLLs, 9</td>
</tr>
<tr>
<td></td>
<td>getPlugin, 8</td>
</tr>
<tr>
<td></td>
<td>getPlugin (plugins), 10</td>
</tr>
<tr>
<td></td>
<td>inline (inline-package), 2</td>
</tr>
<tr>
<td></td>
<td>inline-package, 2</td>
</tr>
<tr>
<td></td>
<td>moveDLL (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>moveDLL, CFunc-method (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>moveDLL-methods (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>package.skeleton, 10</td>
</tr>
<tr>
<td></td>
<td>package.skeleton, ANY, ANY-method</td>
</tr>
<tr>
<td></td>
<td>(package.skeleton-methods), 9</td>
</tr>
<tr>
<td></td>
<td>package.skeleton, character, CFunc-method (package.skeleton-methods), 9</td>
</tr>
<tr>
<td></td>
<td>package.skeleton, character, CFuncList-method (package.skeleton-methods), 9</td>
</tr>
<tr>
<td></td>
<td>package.skeleton-methods, 9</td>
</tr>
<tr>
<td></td>
<td>plugins, 10</td>
</tr>
<tr>
<td></td>
<td>print, CFunc-method (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>print, CFuncList-method (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>rcpp (cxxfunction), 7</td>
</tr>
<tr>
<td></td>
<td>readCFunc (utilities), 11</td>
</tr>
<tr>
<td></td>
<td>registerPlugin (plugins), 10</td>
</tr>
<tr>
<td></td>
<td>save, 12</td>
</tr>
<tr>
<td></td>
<td>saveRDS, 12</td>
</tr>
<tr>
<td></td>
<td>setCMethod (cfunction), 2</td>
</tr>
<tr>
<td></td>
<td>tempdir, 12</td>
</tr>
<tr>
<td></td>
<td>utilities, 11</td>
</tr>
<tr>
<td></td>
<td>writeCFunc (utilities), 11</td>
</tr>
</tbody>
</table>