Writing R Extensions

Version 1.1.1 (2000 August 15)

R Development Core Team

Table of Contents

Acknowledgements

The contributions of Saikat DebRoy (who wrote the first draft of a guide to using `.Call` and `.External`) and of Adrian Trapletti (who provided information on the C++ interface) are gratefully acknowledged.

1 Creating R packages

Packages provide a mechanism for loading optional code and attached documentation as needed. The R distribution provides several packages, such as **eda**, **mva**, and **stepfun**.

1.1 Package structure

A package consists of a subdirectory containing the files 'DESCRIPTION' and 'INDEX', and the subdirectories 'R', 'data', 'exec', 'inst', 'man', 'src', and 'tests' (some of which can be missing). Optionally the package can also contain script files 'configure' and 'cleanup' which are executed before and after installation, See Section 1.2 [Configure and cleanup], page 5.

1.1.1 The DESCRIPTION file

The 'DESCRIPTION' file contains basic information about the package in the following format:

```
Package: pkgname
Version: 0.5-1
Date: 2000/01/04
Title: My first collection of functions
Author: Friedrich Leisch <F.Leisch@ci.tuwien.ac.at>.
Depends: R (>= 0.99), nlme
Description: A short (one paragraph) description of what
        the package does and why it may be useful.
License: GPL version 2 or newer
URL: http://www.r-project.org, http://www.another.url
```

Continuation lines (for example, for descriptions longer than one line) start with a space or tab. The 'Package', 'Version', 'Author', and 'Description' fields are mandatory, the remaining fields ('Date', 'Depends', 'Address', 'URL', ...) are optional. The 'Author' field should contain an email address in angle brackets (for sending bug reports etc.).

The 'License' field should contain an explicit statement or a well-known abbreviation (such as 'GPL', 'LGPL', 'BSD', or 'Artistic'), perhaps followed by a reference to the actual license file. It is very important that you include this information! Otherwise, it may not even be legally correct for others to distribute copies of the package.

The 'Title' field should give a short description of the package and not have any continuation lines. This information has been contained in a separate 'TITLE' file in older versions of R, please use the title field in the 'DESCRIPTION' file from now on.

The optional 'URL' field may give a list of URL's separated by commas or whitespace, for example the homepage of the author or a page where additional material describing the software can be found. These URLs are converted to active hyperlinks on CRAN.

The optional 'Depends' field gives a comma-separated list of package names which this package depends on. The package name may be optionally followed by comparison operator (currently only '>=' and '<=' are supported) and a version number in parentheses. You can

also use the special package name 'R' if your package depends on a certain version of R. E.g, if the package works only with R version 0.90 or newer, include 'R (>= 0.90)' in the 'Depends' field. Future versions of R will use this field to autoload required packages, hence please do not misuse the 'Depends' field for comments on other software that might be needed or not using proper syntax. Other dependencies should be listed in the 'Description' field or a separate 'README' file. The 'R INSTALL' facility already checks if the running version of R is good enough for a package.

1.1.2 Other files

The file 'INDEX' contains a line for each sufficiently interesting object in the package, giving its name and a description (functions such as print methods not usually called explicitly might not be included). Note that you can automatically create this file using something like `R CMD Rdindex man/*.Rd > INDEX`, provided that Perl is available on your system.

1.1.3 Package subdirectories

The 'R' subdirectory contains R code files. The code files to be installed must start with a (lower or upper case) letter and have one of the extensions '.R', '.S', '.q', '.r', or '.s'. We recommend using '.R', as this extension seems to be not used by any other software. It should be possible to read in the files using `source()`, so R objects must be created by assignments. Note that there need be no connection between the name of the file and the R objects created by it. If necessary, one of these files (historically 'zzz.R') should use `library.dynam()` *inside* `.First.lib()` to load compiled code.

The 'man' subdirectory should contain documentation files for the objects in the package in "R documentation" (Rd) format. The documentation files to be installed must also start with a (lower or upper case) letter and have the extension '.Rd' (the default) or '.rd'. See Chapter 2 [Writing R documentation], page 8, for more information.

The 'R' and 'man' subdirectories may contain OS-specific subdirectories named `unix`, `windows` or `mac`.

The C or FORTRAN source files for the compiled code, and optionally files 'Makevars' and 'Makefile', are in 'src'. When a package is installed using `R CMD INSTALL`, Make is used to control compilation and linking into a shared library for loading into R. There are default variables and rules for this (determined when R is configured and recorded in '$R_HOME/etc/Makeconf'). If a package needs to specify additional directories for searching header files ('-I' options) or additional libraries for linking ('-l' and '-L' options), it should do this by setting the variables `PKG_CPPFLAGS` and `PKG_LIBS` in 'src/Makevars'. (Additional flags to be passed to the C or Fortran compiler can be specified by the variables `PKG_CFLAGS` and `PKG_FFLAGS`, respectively.) Note that this mechanism should be general enough to eliminate the need for a package-specific 'Makefile'. If such a file is to be distributed, considerable care is needed to make it general enough to work on all R platforms.

The 'data' subdirectory is for additional data files the package makes available for loading using `data()`. Currently, data files can have one of three types as indicated by their extension: plain R code ('.R' or '.r'), tables ('.tab', '.txt', or '.csv'), or `save()` images ('.RData' or '.rda'). Note that R code should be "self-sufficient" and not make use of

extra functionality provided by the package, so that the data file can also be used without having to load the package. The '`data`' subdirectory should also contain a '`00Index`' file that describes the datasets available. Ideally this should have a one-line description of each dataset, with full documentation in the '`man`' directory.

The contents of the '`inst`' subdirectory will be copied recursively to the installation directory.

Subdirectory '`tests`' is for additional package-specific test code, similar to the specific tests that come with the R distribution. Test code can either be provided directly in a '`.R`' file, or via a '`.Rin`' file containing code which in turn creates the corresponding '`.R`' file (e.g., by collecting all function objects in the package and then calling them with the strangest arguments). The results of running a '`.R`' file are written to a '`.Rout`' file. If there is a corresponding '`.Rout.save`' file, these two are compared, with differences being reported but not causing an error.

Finally, '`exec`' could contain additional executables the package needs, typically shell or Perl scripts. This mechanism is currently not used by any Unix package, and still experimental.

1.1.4 Package bundles

Sometimes it is convenient to distribute several packages as a *bundle*. (The main current example is **VR** which contains four packages.) The installation procedures on both Unix and Windows can handle package bundles.

The '`DESCRIPTION`' file of a bundle has an extra '`Bundle`' field, as in

```
Bundle: VR
Contains: MASS class nnet spatial
Version: 6.1-6
Date: 1999/11/26
Author: S original by Venables & Ripley.
  R port by Brian Ripley <ripley@stats.ox.ac.uk>, following
  earlier work by Kurt Hornik and Albrecht Gebhardt.
BundleDescription: Various functions from the libraries of
  Venables and Ripley, 'Modern Applied Statistics with S-PLUS'
  (3rd edition).
License: GPL (version 2 or later)
```

The '`Contains`' field lists the packages, which should be contained in separate subdirectories with the names given. These are standard packages in all respects except that the '`DESCRIPTION`' file is replaced by a '`DESCRIPTION.in`' file which just contains fields additional to the '`DESCRIPTION`' file of the bundle, for example

```
Package: spatial
Description: Functions for kriging and point pattern analysis.
```

1.2 Configure and cleanup

If your package needs some system-dependent configuration before installation you can include a script '`configure`' in your package which (if present) is executed by `R CMD INSTALL` before any other action is performed. This can be a script created by the `autoconf` mechanism, but may also be a script written by yourself. Use this to detect if any nonstandard libraries are present such that corresponding code in the package can be disabled at install time rather than giving error messages when the package is compiled or used. To summarize, the full power of `autoconf` is available for your extension package (including variable substitution, searching for libraries, etc.).

The script '`cleanup`' is executed as last thing by `R CMD INSTALL` if present and can be used to clean up the package source tree, especially remove files created by '`configure`'.

As an example consider we want to use functionality provided by a (C or Fortran) library `foo`. Using `autoconf`, we can write a configure script which checks for the library, sets variable `HAVE_FOO` to `TRUE` if it was found and with `FALSE` otherwise, and then substitutes this value into output files (by replacing instances of '`@HAVE_FOO@`' in input files with the value of `HAVE_FOO`). For example, if a function named `bar` is to be made available by linking against library `foo` (i.e., using '`-lfoo`'), one could use

```
AC_CHECK_LIB(foo, fun, HAVE_FOO=TRUE, HAVE_FOO=FALSE)
AC_SUBST(HAVE_FOO)
...
AC_OUTPUT(foo.R)
```

The definition of the respective R function in '`foo.R.in`' could be

```
foo <- function(x) {
    if(!@HAVE_FOO@) stop("Sorry, library 'foo' is not available")
    ...
```

From this file `configure` creates the actual R source file '`foo.R`' looking like

```
foo <- function(x) {
    if(!FALSE) stop("Sorry, library 'foo' is not available")
    ...
```

if library `foo` was not found (with the desired functionality). In this case, the above R code effectively disables the function.

One could also use different file fragments for available and missing functionality, respectively.

You may wish to bear in mind that the '`configure`' script may well not work on Windows systems (this seems normally to be the case for those generated by `autoconf`, although simple shell scripts do work). If your package is to be made publicly available, please give enough information for a user on a non-Unix platform to configure it manually.

1.3 Checking and building packages

Using `R CMD check`, the R package checker, one can test whether *installed* R packages work correctly. This checks whether the examples provided by the packages' documentation can be run successfully (see Chapter 2 [Writing R documentation], page 8, for information on using `\examples` to create executable example code.) If the package sources contain a

'`tests`' directory then the tests specified in that directory are run. (Typically they will consist of a set of '`.R`' source files and target output files '`.Rout.save`'.)

Of course, released packages should be able to run at least their own examples.

Use *`R CMD check --help`* to obtain more information about the usage of the R package checker. Under Windows the equivalent command is `make pkgcheck-mypkg`. Note that some packages will require more than the default workspace size to run their examples: this needs to be specified by setting the environment variables `R_NSIZE` and `R_VSIZE` or (Unix) the command-line options `--nsize` and `--vsize`.

Using `R CMD build`, the R package builder, one can build R packages from their sources (for example, for subsequent release). Prior to actually building the package in the common gzipped tar file format, a variety of diagnostic checks and cleanups are performed. In particular, it is tested whether the '`DESCRIPTION`' file contains the required entries, whether object and data indices exist (it will build them if they do not) and can be assumed to be up-to-date, whether the documentation sources contain the required name, alias, title, description, and keyword information, and whether there are any undocumented user level objects. Run-time checks whether the package works correctly should be performed using `R CMD check` prior to invoking the build procedure.

Use *`R CMD build --help`* to obtain more information about the usage of the R package builder. The Windows equivalent is to set `R_HOME` and then run

```
sh ${R_HOME}/bin/build.sh pkgname
```

(Replace '`pkgname`' by '`--help`' for further details.)

1.4 Submitting a package to CRAN

CRAN is a network of WWW sites holding the R distributions and contributed code, especially R packages. Users of R are encouraged to join in the collaborative project and to submit their own packages to CRAN.

Before submitting a package `mypkg`, do run the following steps to test it is complete and will install properly. (Unix procedures only, run from the directory containing '`mypkg`' as a subdirectory.)

1. Install the package (by `R INSTALL mypkg`). This will warn about missing cross-references and duplicate aliases in help files. If you can, do the install on a 'vanilla' system with no additional packages installed.
2. Run `R CMD check` to check that all the examples on the help pages can be run, and to check the test suite in '`tests`' (if any).
3. Run `R CMD Rd2dvi mypkg/man` to build a LaTeX version of the reference manual. (This checks the `latex`-specific parts of the help files.) An even better check if you have `pdftex` installed is `R CMD Rd2dvi --pdf mypkg/man` as this will build the PDF documentation as installed on CRAN.
4. Run `R CMD build` to run further checks on the help files and documentation files such as '`DESCRIPTION`' and to make the release '`.tar.gz`' file.

Please ensure that you can run through the complete procedure with only warnings that you understand and have reasons not to eliminate.

When all the testing is done, upload the '`.tar.gz`' file to

`ftp://ftp.ci.tuwien.ac.at/incoming`

and send a message to `WWWadmin@ci.tuwien.ac.at` about it. The CRAN maintainers will run these tests before putting a submission in the main archive.

2 Writing R documentation

2.1 Rd format

R objects are documented in files written in "R documentation" (Rd) format, a simple markup language closely resembling (La)TEX, which can be processed into a variety of formats, including LaTEX, HTML and plain text. The translation is carried out by the Perl script `Rdconv` in '`$R_HOME/bin`' and by the installation scripts for packages.

The R distribution contains more than 600 such files which can be found in the '`src/library/`*pkg*`/man`' directories of the R source tree, where *pkg* stands for package **base** where all the standard objects are, and for the standard packages such as **eda** and **mva** which are included in the R distribution.

As an example, let us look at the file '`src/library/base/man/rle.Rd`' which documents the R function `rle`.

```
\name{rle}
\alias{rle}
\title{Run Length Encoding}
\description{
  Compute the lengths and values of runs of equal values in
  a vector.
}
\usage{
rle(x)
}
\arguments{
  \item{x}{a (numerical, logical or character) vector.}
}
\value{
  A list with components
  \item{lengths}{a vector containing the length of each run.}
  \item{values}{a vector of the same length as \code{lengths}
    with the corresponding values.}
}
\examples{
x <- rev(rep(6:10, 1:5))
rle(x)
## $lengths
## [1] 5 4 3 2 1
## $values
## [1] 10  9  8  7  6
z <- c(T,T,F,F,T,F,T,T,T)
rle(z)
rle(as.character(z))
}
\keyword{manip}
```

An Rd file consists of three parts. The header gives basic information about the name of the file, the topics documented, a title, a short textual description and R usage information for the objects documented. The body gives further information (for example, on the function's arguments and return value, as in the above example). Finally, there is a footer with keyword information. The header and footer are mandatory.

See "Guidelines for Rd files" for guidelines for writing documentation in Rd format which should be useful for package writers.

2.1.1 Documenting functions

The basic markup commands used for documenting R objects (in particular, functions) are given in this subsection.

`\name{`*file*`}`
: *file* is the basename of the file.

`\alias{`*topic*`}`
: The `\alias` entries specify all "topics" the file documents. This information (together with the file name) is collected into index data bases for lookup by the on-line (plain text and HTML) help systems.

  There may be several `\alias` entries. Quite often it is convenient to document several R objects in one file. For example, file '`Normal.Rd`' documents the density, distribution function, quantile function and generation of random variates for the normal distribution, and hence starts with

```
\name{Normal}
\alias{dnorm}
\alias{pnorm}
\alias{qnorm}
\alias{rnorm}
```

  Note that the name of the file is not necessarily a topic documented.

`\title{`*Title*`}`
: Title information for the Rd file. This should be capitalized, not end in a period, and not use any markup (which would cause problems for hypertext search).

`\description{...}`
: A short description of what the function(s) do(es) (one paragraph, a few lines only). (If a description is "too long" and cannot easily be shortened, the file probably tries to document too much at once.)

`\usage{`*fun*`(`*arg1*`, `*arg2*`, ...)}`
: One or more lines showing the synopsis of the function(s) and variables documented in the file. These are set verbatim in typewriter font.

  The usage information specified should in general match the function definition *exactly* (such that automatic checking for consistency between code and documentation is possible). Otherwise, include a `\synopsis` section with the actual definition.

  For example, `abline` is a function for adding a straight line to a plot which can be used in several different ways, depending on the named arguments specified. Hence, '`abline.Rd`' contains

```
\synopsis{
abline(a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
       coef = NULL, untf = FALSE, col = par("col"),
       lty = par("lty"), lwd = NULL, ...)
}
\usage{
abline(a, b, \dots)
abline(h=, \dots)
abline(v=, \dots)
...
}
```

`\arguments{...}`
: Description of the function's arguments, using an entry of the form

  `\item{`*arg_i*`}{`*Description of arg_i*`.}`

  for each element of the argument list. There may be optional text before and after these entries.

`\details{...}`
: A detailed if possible precise description of the functionality provided, extending the basic information in the `\description` slot.

`\value{...}`
: Description of the function's return value.

  If a list with multiple values is returned, you can use entries of the form

  `\item{`*comp_i*`}{`*Description of comp_i*`.}`

  for each component of the list returned. Optional text may precede this list (see the introductory example for `rle`).

`\references{...}`
: A section with references to the literature. Use `\url{}` for web pointers.

`\note{...}`
: Use this for a special note you want to have pointed out.

  For example, '`piechart.Rd`' contains

```
\note{
  Pie charts are a very bad way of displaying information.
  The eye is good at judging linear measures and bad at
  judging relative areas.
  ...
}
```

`\author{...}`
: Information about the author(s) of the Rd file. Use `\email{}` without extra delimiters ('`( )`' or '`< >`') to specify email addresses, or `\url{}` for web pointers.

`\seealso{...}`
: Pointers to related R objects, using `\code{\link{...}}` to refer to them (`\code` is the correct markup for R object names, and `\link` produces hyperlinks in output formats which support this. See Section 2.3 [Marking text], page 13, and Section 2.5 [Cross-references], page 13).

`\examples{...}`
: Examples of how to use the function. These are set verbatim in typewriter font.

  Examples are not only useful for documentation purposes, but also provide test code used for diagnostic checking of R. By default, text inside `\examples{}` will be displayed in the output of the help page and run by *`make check`*. You can use `\dontrun{}` for commands that should only be shown, but not run, and `\testonly{}` for extra commands for testing R that should not be shown to users.

  For example,

```
x <- runif(10)        # Shown and run.
\dontrun{plot(x)}     # Only shown.
\testonly{log(x)}     # Only run.
```

  Thus, example code not included in `\dontrun` must be executable! In addition, it should not use any system-specific features or require special facilities (such as Internet access or write permission to specific directories).

  Data needed for making the examples executable can be obtained by random number generation (for example, `x <- rnorm(100)`), or by using standard data sets loadable via `data()` (see `data()` for info).

`\keyword{`*key*`}`
: Each `\keyword` entry should specify one of the standard keywords (as listed in the file '`$R_HOME/doc/KEYWORDS`'). There must be at least one `\keyword` entry, but can be more that one if the R object being documented falls into more than one category.

The R function `prompt` facilitates the construction of files documenting R objects. If `foo` is an R function, then `prompt(foo)` produces file '`prompt.Rd`' which already contains the proper function and argument names of `foo`, and a structure which can be filled in with information.

2.1.2 Documenting datasets

The structure of Rd files which document R data sets is slightly different. Whereas sections such as `\arguments` and `\value` are not needed, the format and source of the data should be explained.

As an example, let us look at '`src/library/base/man/rivers.Rd`' which documents the standard R data set `rivers`.

```
\name{rivers}
\alias{rivers}
\title{Lengths of Major North American Rivers}
\description{
  This data set gives the lengths (in miles) of 141 ``major''
  rivers in North America, as compiled by the US Geological
  Survey.
}
\usage{data(rivers)}
\format{A vector containing 141 observations.}
\source{World Almanac and Book of Facts, 1975, page 406.}
\references{
  McNeil, D. R. (1977) \emph{Interactive Data Analysis}.
  New York: Wiley.
}
\keyword{datasets}
```

This uses the following additional markup commands.

`\format{...}`
: A description of the format of the dataset (as a vector, matrix, data frame, time series, ...). For matrices and data frames this should give a description of each column, preferably as a list or table. See Section 2.4 [Lists and tables], page 13, for more information.

`\source{...}`
: Details of the original source (a reference or URL). In addition, section `\references` could give secondary sources and usages.

Note also that when documenting data set *bar*,

- The `\usage` entry always is `data(`*bar*`)`.
- The `\keyword` entry always is `datasets`.

If `bar` is a data frame, documenting it as a data set can again be initiated via `prompt(bar)`.

2.2 Sectioning

To begin a new paragraph or leave a blank line in an example, just insert an empty line (as in (La)TEX). To break a line, use `\cr`.

In addition to the predefined sections (such as `\description{}`, `\value{}`, etc.), you can "define" arbitrary ones by `\section{`*section_title*`}{...}`. For example

```
\section{Warning}{You must not call this function unless ...}
```

For consistency with the pre-assigned sections, the section name (the first argument to `\section`) should be capitalized (but not all upper case).

Note that the additional named sections are always inserted at fixed positions in the output (before `\note`, `\seealso` and the examples), no matter where they appear in the input.

2.3 Marking text

The following logical markup commands are available for indicating specific kinds of text.

`\bold{`*word*`}`	set *word* in **bold** font if possible
`\emph{`*word*`}`	emphasize *word* using *italic* font if possible
`\code{`*word*`}`	for pieces of code, using `typewriter` font if possible
`\file{`*word*`}`	for file names
`\email{`*word*`}`	for email addresses
`\url{`*word*`}`	for URLs

The first two, `\bold` and `\emph`, should be used in plain text for emphasis.

Fragments of R code, including the names of R objects, should be marked using `\code`. Only backslashes and percent signs need to be escaped (by a backslash) inside `\code`.

2.4 Lists and tables

The `\itemize` and `\enumerate` commands take a single argument, within which there may be one or more `\item` commands. The text following each `\item` is formatted as one or more paragraphs, suitably indented and with the first paragraph marked with a bullet point (`\itemize`) or a number (`\enumerate`).

`\itemize` and `\enumerate` commands may be nested.

The `\describe` command is similar to `\itemize` but allows initial labels to be specified. The `\item`s take two arguments, the label and the body of the item, in exactly the same way as argument and value `\item`s. `\describe` commands are mapped to `<DL>` lists in HTML and `\description` lists in LaTeX.

The `\tabular` command takes two arguments. The first gives for each of the columns the required alignment (`l` for left-justification, `r` for right-justification or `c` for centring.) The second argument consists of an arbitrary number of lines separated by `\cr`, and with fields separated by `\tab`. For example:

```
\tabular{rlll}{
  [,1] \tab Ozone   \tab numeric \tab Ozone (ppb)\cr
  [,2] \tab Solar.R \tab numeric \tab Solar R (lang)\cr
  [,3] \tab Wind    \tab numeric \tab Wind (mph)\cr
  [,4] \tab Temp    \tab numeric \tab Temperature (degrees F)\cr
  [,5] \tab Month   \tab numeric \tab Month (1--12)\cr
  [,6] \tab Day     \tab numeric \tab Day of month (1--31)
}
```

There must be the same number of fields on each line as there are alignments in the first argument, and they must be non-empty (but can contain only spaces).

2.5 Cross-references

The markup `\link{`*foo*`}` (usually in the combination `\code{\link{`*foo*`}}`) produces a hyperlink to the help page for object *foo*. One main usage of `\link` is in the `\seealso` section of the help page, see Section 2.1 [Rd format], page 8. (This only affects the creation

of hyperlinks, for example in the HTML pages used by `help.start()` and the PDF version of the reference manual.)

There are optional arguments specified as `\link[`*pkg*`]{`*foo*`}` and `\link[`*pkg:bar*`]{`*foo*`}` to link to the package *pkg* with topic (file?) *foo* and *bar* respectively.

2.6 Mathematics

Mathematical formulae should be set beautifully for printed documentation yet we still want something useful for text and HTML online help. To this end, the two commands `\eqn{`*latex*`}{`*ascii*`}` and `\deqn{`*latex*`}{`*ascii*`}` are used. Where `\eqn` is used for "inline" formula (corresponding to TEX's `$...$`, `\deqn` gives "displayed equations" (as in LATEX's `displaymath` environment, or TEX's `$$...$$`).

Both commands can also be used as `\eqn{`*latexascii*`}` (only *one* argument) which then is used for both *latex* and *ascii*.

The following example is from the `Poisson` help page:

```
\deqn{p(x) = \frac{\lambda^x e^{-\lambda}}{x!}}{%
      p(x) = lambda^x exp(-lambda)/x!}
for \eqn{x = 0, 1, 2, \ldots}.
```

For the LATEX manual, this becomes

$$p(x) = \lambda^x \frac{e^{-\lambda}}{x!}$$

for $x = 0, 1, 2, \ldots$.

For the HTML and the "direct" (man-like) on-line help we get

```
    p(x) = lambda^x exp(-lambda)/x!

for x = 0, 1, 2, ....
```

2.7 Insertions

Use `\R` for the R system itself (you don't need extra '`{}`' or '`\`'). Use `\dots` for the dots in function argument lists '`...`', and `\ldots` for ellipsis dots in ordinary text.

After a '`%`', you can put your own comments regarding the help text. The rest of the line will be completely disregarded, normally. Therefore, you can also use it to make part of the "help" invisible.

You can produce a backslash ('`\`' by escaping it by another backslash. (Note that `\cr` is used for generating line breaks.)

The "comment" and "control" characters '`%`' and '`\`' *always* need to be escaped. Inside the verbatim-like commands (`\code` and `\examples`), no other[1] characters are special. Note that `\file` is **not** a verbatim-like command.

[1] This is not quite true. Unpaired braces will give problems and should be escaped. See the examples section in the file '`Paren.Rd`' for an example.

In "regular" text (no verbatim, no `\eqn`, ...), you currently must escape most LaTeX special characters, i.e., besides '`%`', '`{`', and '`}`', the four specials '`$`', '`#`', and '`_`' are produced by preceding each with a '`\`'. ('`&`' can also be escaped, but need not be.) Further, enter '`^`' as `\eqn{\mbox{\textasciicircum}}{^}`, and '`~`' by `\eqn{\mbox{\textasciitilde}}{~}` or `\eqn{\sim}{~}` (for a short and long tilde respectively). Also, '`<`', '`>`', and '`|`' must only be used in math mode, i.e., within `\eqn` or `\deqn`.

2.8 Platform-specific documentation

Sometimes the documentation needs to differ by platform. Currently three OS-specific options are available, `unix`, `windows` and `mac`, and lines in the help source file can be enclosed in

```
#ifdef OS
   ...
#endif
```

or

```
#ifndef OS
   ...
#endif
```

for OS-specific inclusion or exclusion.

If the differences between platforms are extensive or the R objects documented are only relevant to one platform, platform-specific Rd files can be put in a '`unix`', '`windows`' or '`mac`' subdirectory.

2.9 Processing Rd format

Under UNIX versions of R there are several commands to process Rd files, if Perl is installed.

Using `R CMD Rdconv` one can convert R documentation format to other formats, or extract the executable examples for run-time testing. Currently, conversions to plain text, HTML, LaTeX, and S documentation formats are supported.

In addition to this low-level conversion tool, the R distribution provides two user-level programs for processing Rd format. `R CMD Rd2txt` produces "pretty" plain text output from an Rd file, and is particularly useful as a previewer when writing Rd format documentation within Emacs. `R CMD Rd2dvi` generates DVI (or, if option '`--pdf`' is given, PDF) output from documentation in Rd files, which can be specified either explicitly or by the path to a directory with the sources of a package. In the latter case, a reference manual for all documented objects in the package is created. Future versions will also add the information in the '`DESCRIPTION`' files to the output.

Using `R CMD Rdindex` one can produce nicely formatted index files displaying names and titles of the Rd files specified as arguments. This can be used to create the '`INDEX`' of an add-on package and, if it also contains data, the '`00Index`' data index in the '`data`' directory. Note that the R package builder `R CMD build` can be used to automatically create these indices when building a package.

Finally, `R CMD Sd2Rd` converts S version 3 documentation files (which use an extended Nroff format) to Rd format. This is useful when porting a package originally written for the S system to R.

The exact usage and a detailed list of available options for each of the above commands can be obtained by running `R CMD command --help`, e.g., `R CMD Rdconv --help`. All available commands can be listed using `R --help`.

All of these have Windows equivalents. The batch files in the '`bin`' directory '`Rd2txt.bat`', '`Rdconv.bat`', '`Rdindex.bat`' and '`Sd2Rd.bat`' can be used in exactly the same way as their Unix counterparts if the environment variable `R_HOME` is set. (You will need Perl installed: you need that on Unix too.) There is a shell script '`Rd2dvi.sh`' which needs a Unix-like environment on Windows.

3 System and foreign language interfaces

3.1 Operating system access

Access to operating system functions is via the R function `system`. The details will differ by platform (see the on-line help), and about all that can safely be assumed is that the first argument will be a string `command` that will be passed for execution (not necessarily by a shell) and the second argument will be `internal` which if true will collect the output of the command into an R character vector.

The function `system.time` is available for timing (although the information available may be limited on non-Unix-like platforms).

3.2 Interface functions `.C` and `.Fortran`

These two functions provide a standard interface to compiled code that has been linked into R, either at build time or via `dyn.load` (see Section 3.3 [dyn.load and dyn.unload], page 18). They are primarily intended for compiled C and Fortran code respectively, but the `.C` function can be used with other languages which can generate C interfaces, for example C++ (see Section 3.5 [Interfacing C++ code], page 19).

The first argument to each function is a character string given the symbol name as known to C or Fortran, that is the function or subroutine name. (The mapping to the symbol name in the load table is given by the functions `symbol.C` and `symbol.For`; that the symbol is loaded can be tested by, for example, `is.loaded(symbol.C("loglin"))`.)

There can be up to 65 further arguments giving R objects to be passed to compiled code. Normally these are copied before being passed in, and copied again to an R list object when the compiled code returns. If the arguments are given names, these are used as names for the components in the returned list object (but not passed to the compiled code).

The following table gives the mapping between the modes of R vectors and the types of arguments to a C function or Fortran subroutine.

R storage mode	C type	Fortran type
`logical`	`int *`	`INTEGER`
`integer`	`int *`	`INTEGER`
`double`	`double *`	`DOUBLE PRECISION`
`complex`	`Rcomplex *`	`DOUBLE COMPLEX`
`character`	`char **`	`CHARACTER*255`

C type `Rcomplex` is a structure with `double` members `r` and `i` defined in the header file '`Complex.h`' included by '`R.h`'. Only a single character string can be passed to or from Fortran, and the success of this is compiler-dependent. Other R objects can be passed to `.C`, but it is better to use one of the other interfaces. An exception is passing an R function for use with `call_R`, when the object can be handled as `void *` en route to `call_R`, but even there `.Call` is to be preferred.

It is possible to pass numeric vectors of storage mode `double` to C as `float *` or Fortran as `REAL` by setting the attribute `Csingle`, most conveniently by using the R functions

`as.single`, `single` or `storage.mode`. This is intended only to be used to aid interfacing to existing C or Fortran code.

Unless formal argument `NAOK` is true, all the other arguments are checked for missing values `NA` and for the IEEE special values `NaN`, `Inf` and `-Inf`, and the presence of any of these generates an error. If it is true, these values are passed unchecked.

Argument `DUP` can be used to suppress copying. It is dangerous: see the on-line help for arguments against its use. It is not possible to pass numeric vectors as `float *` or `REAL` if `DUP=TRUE`.

Finally, argument `PACKAGE` confines the search for the symbol name to a specific shared library (or use `"base"` for code compiled into R). Its use is highly desirable, as there is no way to avoid two package writers using the same symbol name, and such name clashes are normally sufficient to cause R to crash.

Note that the compiled code should not return anything except through its arguments: C functions should be of type `void` and Fortran subprograms should be subroutines.

To fix ideas, let us consider a very simple example which convolves two finite sequences. (This is hard to do fast in interpreted R code, but easy in C code.) We could do this using `.C` by

```
void convolve(double *a, int *na, double *b, int *nb, double *ab)
{
  int i, j, nab = *na + *nb - 1;

  for(i = 0; i < nab; i++)
    ab[i] = 0.0;
  for(i = 0; i < *na; i++)
    for(j = 0; j < *nb; j++)
      ab[i + j] += a[i] * b[j];
}
```

called from R by

```
conv <- function(a, b)
  .C("convolve",
     as.double(a),
     as.integer(length(a)),
     as.double(b),
     as.integer(length(b)),
     ab = double(length(a) + length(b) - 1))$ab
```

Note that we take care to coerce all the arguments to the correct R storage mode before calling `.C`; mistakes in matching the types can lead to wrong results or hard-to-catch errors.

3.3 `dyn.load` and `dyn.unload`

Compiled code to be used with R is loaded as a shared library (Unix, see Section 3.4 [Creating shared libraries], page 19 for more information) or DLL (Windows).

The library/DLL is loaded by `dyn.load` and unloaded by `dyn.unload`. Unloading is not normally necessary, but it is needed to allow the DLL to be re-built on some platforms, including Windows.

The first argument to both functions is a character string giving the path to the library. Programmers should not assume a specific file extension for the library (such as '`.so`') but use a construction like

```
file.path(path1, path2, paste("mylib", .Platform$dynlib.ext, sep=""))
```

for platform independence. On Unix systems the path supplied to `dyn.load` can be an absolute path, one relative to the current directory or, if it starts with '`~`', relative to the user's home directory.

Loading is most often done via a call to `library.dynam` in the `.First.lib` function of a package. This has the form

```
library.dynam("libname", package, lib.loc)
```

where `libname` is the library/DLL name with the extension omitted.

Under some Unix systems there is a choice of how the symbols are resolved when the library is loaded, governed by the arguments `local` and `now`. Only use these if really necessary: in particular using `now=FALSE` and then calling an unresolved symbol will terminate R unceremoniously.

If a library/DLL is loaded more than once the most recent version is used. More generally, if the same symbol name appears in several libraries, the most recently loaded occurrence is used. The `PACKAGE` argument provides a good way to avoid any ambiguity in which occurrence is meant.

3.4 Creating shared libraries

Under Unix, shared libraries for loading into R can be created using `R CMD SHLIB`. This accepts as arguments a list of files which must be object files (with extension '`.o`') or C or Fortran sources (with extensions '`.c`' and '`.f`', respectively). See *`R CMD SHLIB --help`*, or the on-line help for `SHLIB`, for usage information. If compiling the source files does not work "out of the box", you can specify additional flags by setting some of the variables `PKG_CPPFLAGS` (for the C preprocessor, typically '`-I`' flags), `PKG_CFLAGS` and `PKG_FFLAGS` (for the C and Fortran compilers, respectively) in the file '`Makevars`' in the compilation directory, or write a '`Makefile`' in the compilation directory containing the rules required (or, of course, create the object files directly from the command line). Similarly, variable `PKG_LIBS` in '`Makevars`' can be used for additional '`-l`' and '`-L`' flags to be passed to the linker when building the shared library.

If an add-on package *pkg* contains C or Fortran code in its '`src`' subdirectory, `R CMD INSTALL` creates a shared library (for loading into R in the `.First.lib` function of the package) either automatically using the above `R CMD SHLIB` mechanism, or using `make` if directory '`src`' contains a '`Makefile`'. In both cases, if file '`Makevars`' exists it is read first when invoking `make`. If a '`Makefile`' is really needed or provided, it needs to ensure that the shared library created is linked against all Fortran libraries that R was linked against; `make` variable `FLIBS` contains this information.

3.5 Interfacing C++ code

Suppose we have the following hypothetical C++ library, consisting of the two files '`X.hh`' and '`X.cc`', and implementing the 2 classes `X` and `Y` which we want to use in R.

```
// X.hh

class X {
public:
  X (); ~X ();
};

class Y {
public:
  Y (); ~Y ();
};
```

```
// X.cc

#include <iostream.h>
#include "X.hh"

static Y y;

X::X()  { cout << "constructor X" << endl; }
X::~X() { cout << "destructor X" << endl; }
Y::Y()  { cout << "constructor Y" << endl; }
Y::~Y() { cout << "destructor Y" << endl; }
```

To use with R, the only thing we have to do is writing a wrapper function and ensuring that the function is enclosed in

```
extern "C" {

}
```

For example,

```
// X_main.cc:

#include "X.hh"

extern "C" {

void X_main () {
  X x;
}

}
```

Compiling and linking should be done with the C++ compiler-linker (rather than the C compiler-linker or the linker itself); otherwise, the C++ initialization code (and hence the constructor of the static variable `Y`) are not called. On a properly configured Unix system (support for C++ was added in version 1.1), one can simply use

```
R CMD SHLIB X.cc X_main.cc
```

to create the shared library, typically 'X.so' (the file name extension may be different on your platform). Now starting R yields

```
R : Copyright 2000, The R Development Core Team
Version 1.1.0 Under development (unstable) (April 14, 2000)
...
Type    "q()" to quit R.

R> dyn.load(paste("X", .Platform$dynlib.ext, sep = ""))
constructor Y
R> .C("X_main")
constructor X
destructor X
list()
R> q()
Save workspace image? [y/n/c]: y
destructor Y
```

The R for Windows FAQ ('rw-FAQ') contains details of how to compile this example under various Windows compilers.

Using C++ iostreams, as in this example, is best avoided. There is no guarantee that the output will appear in the R console, and indeed it will not on the R for Windows console. Use R code or the C entry points (see Section 4.5 [Printing], page 43) for all I/O if at all possible.

3.6 Handling R objects in C

Using C code to speed up the execution of an R function is often very fruitful. Traditionally this has been done via the `.C` function in R. One restriction of this interface is that the R objects can not be handled directly in C. This becomes more troublesome when one wishes to call R functions from within the C code. There is a C function provided called `call_R` (also known as `call_S` for compatibility with S) that can do that, but it is cumbersome to use, and the mechanisms documented here are usually simpler to use, as well as more powerful.

If a user really wants to write C code using internal R data structures, then that can be done using the `.Call` and `.External` function. The syntax for the calling function in R in each case is similar to that of `.C`, but the two functions have rather different C interfaces. Generally the `.Call` interface (which is modelled on the interface of the same name in S version 4) is a little simpler to use, but `.External` is a little more general.

A call to `.Call` is very similar to `.C`, for example

```
.Call("convolve2", a, b)
```

The first argument should be a character string giving a C symbol name of code that has already been loaded into R. Up to 65 R objects can passed as arguments. The C side of the interface is

```
#include <R.h>
#include <Rinternals.h>
```

```
SEXP convolve2(SEXP a, SEXP b)
 ...
```

A call to `.External` is almost identical

```
.External("convolveE", a, b)
```

but the C side of the interface is different, having only one argument

```
#include <R.h>
#include <Rinternals.h>

SEXP convolveE(SEXP args)
 ...
```

Here `args` is a `LISTSXP`, a Lisp-style list from which the arguments can be extracted.

In each case the R objects are available for manipulation via a set of functions and macros defined in the header file '`Rinternals.h`' or some higher-level macros defined in '`Rdefines.h`'. Details on `.Call` and `.External` are given further below.

Before you decide to use `.Call` or `.External`, you should look at other alternatives. First, consider working in interpreted R code; if this is fast enough, this is normally the best option. You should also see if using `.C` is enough. If the task to be performed in C is simple enough requiring no call to R, `.C` suffices. The new interfaces are recent additions to S and R, and a great deal of useful code has been written using just `.C` before they were available. The `.Call` and `.External` interfaces allow much more control, but they also impose much greater responsibilities so need to be used with care.

There are two approaches that can be taken to handling R objects from within C code. The first (historically) is to use the macros and functions that have been used to implement the core parts of R through `.Internal` calls. A public subset of these is defined in the header file '`Rinternals.h`' in the directory '`$R_HOME/include`' that should be available on any R installation.

A more recent approach is to use R versions of the macros and functions defined for the S version 4 interface `.Call`, which are defined in the header file '`Rdefines.h`'. This is a somewhat simpler approach, and is certainly to be preferred if the code might be shared with S at any stage.

A substantial amount of R is implemented using the functions and macros described here, so the R source code provides a rich source of examples and "how to do it": indeed many of the examples here were developed by examining closely R system functions for similar tasks. Do make use of the source code for inspirational examples.

It is necessary to know something about how R objects are handled in C code. All the R objects you will deal with will be handled with the type *SEXP*[1], which is a pointer to a structure with typedef `SEXPREC`. Think of this structure as a *variant type* that can handle all the usual types of R objects, that is vectors of various modes, functions, environments, language objects and so on. The details are given later in this section, but for most purposes the programmer does not need to know them. Think rather of a model such as that used by Visual Basic, in which R objects are handed around in C code (as they are in interpreted R code) as the variant type, and the appropriate part is extracted for, for example, numerical calculations, only when it is needed. As in interpreted R code, much use is made of coercion to force the variant object to the right type.

[1] SEXP is an acronym for *S*imple *EXP*ression, common in LISP-like language syntaxes.

3.6.1 Handling the effects of garbage collection

We need to know a little about the way R handles memory allocation. The memory allocated for R objects is not freed by the user; instead, the memory is from time to time *garbage collected*. That is, all the allocated memory not being used is freed, and the objects that are in use may be moved.

The R object types are represented by a C structure defined by a typedef `SEXPREC` in '`Rinternals.h`'. It contains several things among which are pointers to data blocks and to other `SEXPREC`s. With the current memory managements scheme, the data blocks may be moved in memory during garbage collection, and the pointers to them updated. Because of this it is not safe to save and re-use pointers to data blocks in an object's structure. A `SEXP` is simply a pointer to a `SEXPREC`.

If you create an R object in your C code, you must tell R that you are using the object by using the `PROTECT` macro on a pointer to the object. This tells R that the object is in use so it is not destroyed. Notice that it is the object which is protected, not the pointer variable. It is a common mistake to believe that if you invoked `PROTECT(`*p*`)` at some point then *p* is protected from then on, but that is not true once a new object is assigned to *p*.

Protecting an R object automatically protects all the R objects pointed to in the corresponding `SEXPREC`.

The programmer is solely responsible for housekeeping the calls to `PROTECT`. There is a corresponding macro `UNPROTECT` that takes as argument an `int` giving the number of objects to unprotect when they are no longer needed. The protection mechanism is stack-based, so `UNPROTECT(`*n*`)` unprotects the last *n* objects which were protected. The calls to `PROTECT` and `UNPROTECT` must balance when the user's code returns. R will warn about `"stack imbalance in .Call"` (or `.External`) if the housekeeping is wrong.

Here is a small example of creating an R numeric vector in C code. First we use the macros in '`Rdefines.h`':

```
#include <R.h>
#include <Rdefines.h>

  SEXP ab;
    ....
  PROTECT(ab = NEW_NUMERIC(2));
  NUMERIC_POINTER(ab)[0] = 123.45;
  NUMERIC_POINTER(ab)[1] = 67.89;
  UNPROTECT(1);
```

and then those in '`Rinternals.h`':

```
#include <R.h>
#include <Rinternals.h>

  SEXP ab;
    ....
  PROTECT(ab = allocVector(REALSXP, 2));
  REAL(ab)[0] = 123.45; REAL(ab)[1] = 67.89;
  UNPROTECT(1);
```

Now, the reader may ask how the R object could possibly get removed during those manipulations, as it is just our C code that is running. As it happens, we can do without the protection in this case, but in general we do not know (nor want to know) what is hiding behind the R macros and functions we use, and any of them might cause memory to be allocated, hence garbage collection and hence our object `ab` to be (re)moved. It is usually wise to err on the side of caution and assume that any of the R macros and functions might (re)move the object.

In some cases it is necessary to keep better track of whether protection is really needed. Be particularly aware of situations where a large number of objects are generated. The pointer protection stack has a fixed size (default 10,000) and can become full. It is not a good idea then to just `PROTECT` everything in sight and `UNPROTECT` several thousand objects at the end. It will almost invariably be possible to either assign the objects as part of another object (which automatically protects them) or unprotect them immediately after use.

Protection is not needed for objects which R already knows are in use. In particular, this applies to function arguments.

There is a less-used macro `UNPROTECT_PTR(`*s*`)` that unprotects the object pointed to by the `SEXP` *s*, even if it is not the top item on the pointer protection stack. This is rarely needed outside the parser (the R sources have one example, in '`plot3d.c`').

3.6.2 Allocating storage

For many purposes it is sufficient to allocate R objects and manipulate those. There are quite a few `alloc`*Xxx* functions defined in '`Rinternals.h`'—you may want to explore them. These allocate R objects of various types, and for the standard vector types there are `NEW_`*XXX* macros defined in '`Rdefines.h`'.

If storage is required for C objects during the calculations this is best allocating by calling `R_alloc`; see Section 4.1 [Memory allocation], page 41. All of these memory allocation routines do their own error-checking, so the programmer may assume that they will raise an error and not return if the memory cannot be allocated.

3.6.3 Details of R types

Users of the '`Rinternals.h`' macros will need to know how the R types are known internally: this is more or less completely hidden if the '`Rdefines.h`' macros are used.

The different R data types are represented in C by *SEXPTYPE*. Some of these are familiar from R and some are internal data types. The usual R object modes are given in the table.

SEXPTYPE	R equivalent
`REALSXP`	numeric with storage mode `double`
`INTSXP`	integer
`CPLXSXP`	complex
`LGLSXP`	logical
`STRSXP`	character
`VECSXP`	list (generic vector)

LISTXP	"dotted-pair" list
DOTSXP	a '...' object
NILSXP	NULL
SYMSXP	name/symbol
CLOSXP	function or function closure
ENVSXP	environment

Among the important internal SEXPTYPEs are LANGSXP, CHARSXP etc.

Unless you are very sure about the type of the arguments, the code should check the data types. Sometimes it may also be necessary to check data types of objects created by evaluating an R expression in the C code. You can use functions like isReal, isInteger and isString to do type checking. See the header file 'Rinternals.h' for definitions of other such functions. All of these take a SEXP as argument and return 1 or 0 to indicate *TRUE* or *FALSE*. Once again there are two ways to do this, and 'Rdefines.h' has macros such as IS_NUMERIC.

What happens if the SEXP is not of the correct type? Sometimes you have no other option except to generate an error. You can use the function error for this. It is usually better to coerce the object to the correct type. For example, if you find that an SEXP is of the type INTEGER, but you need a REAL object, you can change the type by using, equivalently,

```
PROTECT(newSexp = coerceVector(oldSexp, REALSXP));
```

or

```
PROTECT(newSexp = AS_NUMERIC(oldSexp));
```

Protection is needed as a new *object* is created; the object formerly pointed to by the SEXP is re-used is still protected but now unused.

All the coercion functions do their own error-checking, and generate NAs with a warning or stop with an error as appropriate.

So far we have only seen how to create and coerce R objects from C code, and how to extract the numeric data from numeric R vectors. These can suffice to take us a long way in interfacing R objects to numerical algorithms, but we may need to know a little more to create useful return objects.

3.6.4 Attributes

Many R objects have attributes: some of the most useful are classes and the dim and dimnames that mark objects as matrices or arrays. It can also be helpful to work with the names attribute of vectors.

To illustrate this, let us write code to take the outer product of two vectors (which outer and %o% already do). As usual the R code is simple

```
out <- function(x, y) .Call("out", as.double(x), as.double(y))
```

where we expect x and y to be numeric vectors, possibly with names. This time we do the coercion in the calling R code.

C code to do the computations is

```
#include <R.h>
#include <Rinternals.h>
```

```
SEXP out(SEXP x, SEXP y)
{
  int i, j, nx, ny;
  double tmp;
  SEXP ans;

  nx = length(x); ny = length(y);
  PROTECT(ans = allocMatrix(REALSXP, nx, ny));
  for(i = 0; i < nx; i++) {
    tmp = REAL(x)[i];
    for(j = 0; j < ny; j++)
      REAL(ans)[i + nx*j] = tmp * REAL(y)[j];
  }
  UNPROTECT(1);
  return(ans);
}
```

but we would like to set the `dimnames` of the result. Although `allocMatrix` provides a short cut, we will show how to set the `dim` attribute directly.

```
#include <R.h>
#include <Rinternals.h>

SEXP out(SEXP x, SEXP y)
{
  int i, j, nx, ny;
  double tmp;
  SEXP ans, dim, dimnames;

  nx = length(x); ny = length(y);
  PROTECT(ans = allocVector(REALSXP, nx*ny));
  for(i = 0; i < nx; i++) {
    tmp = REAL(x)[i];
    for(j = 0; j < ny; j++)
      REAL(ans)[i + nx*j] = tmp * REAL(y)[j];
  }
  PROTECT(dim = allocVector(INTSXP, 2));
  INTEGER(dim)[0] = nx; INTEGER(dim)[1] = ny;
  setAttrib(ans, R_DimSymbol, dim);

  PROTECT(dimnames = allocVector(VECSXP, 2));
  VECTOR(dimnames)[0] = getAttrib(x, R_NamesSymbol);
  VECTOR(dimnames)[1] = getAttrib(y, R_NamesSymbol);
  setAttrib(ans, R_DimNamesSymbol, dimnames);
  UNPROTECT(3);
  return(ans);
}
```

This example introduces several new features. The `getAttrib` and `setAttrib` functions get and set individual attributes. Their second argument is a `SEXP` defining the name in

the symbol table of the attribute we want; these and many such symbols are defined in the header file 'Rinternals.h'.

There are shortcuts here too: the functions namesgets, dimgets and dimnamesgets are the internal versions of names<-, dim<- and dimnames<-, and there are functions such as GetMatrixDimnames and GetArrayDimnames.

What happens if we want to add an attribute that is not pre-defined? We need to add a symbol for it *via* a call to install. Suppose for illustration we wanted to add an attribute "version" with value 3.0. We could use

```
{ SEXP version;
PROTECT(version = allocVector(REALSXP, 1));
REAL(version) = 3.0;
setAttrib(ans, install("version"), version);
UNPROTECT(1);
}
```

Using install when it is not needed is harmless and provides a simple way to retrieve the symbol from the symbol table if it is already installed.

3.6.5 Classes

In R the class is just the attribute named "class" so it can be handled as such, but there is a shortcut classgets. Suppose we want to give the return value in our example the class "mat". We can use

```
#include <R.h>
#include <Rdefines.h>
    ....
  SEXP ans, dim, dimnames, class;
    ....
  PROTECT(class = allocVector(STRSXP, 1));
  STRING(class)[0] = COPY_TO_USER_STRING("mat");
  classgets(ans, class);
  UNPROTECT(4);
  return(ans);
}
```

As the value is a character vector, we have to know how to create that from a C character array, which we do using the macro COPY_TO_USER_STRING defined in 'Rdefines.h'.

3.6.6 Handling lists

Some care is needed with lists, as R has moved from using LISP-like lists (now called "pairlists") to S-like generic vectors. As a result, the appropriate test for a object of mode list is isNewList, and we need allocVector(VECSXP, n) and *not* allocList(n).

List elements can be retrieved or set by direct access to the elements of the generic vector. Suppose we have a list object

```
a <- list(f=1, g=2, h=3)
```

Then we can access a$g as a[[2]] by

```
double g;
  ....
g = REAL(VECTOR(a)[1])[0];
```

This can rapidly become tedious, and the following function (based on one in package **nls**) is very useful:

```
/* get the list element named str, or return NULL */

SEXP getListElement(SEXP list, char *str)
{
  SEXP elmt = R_NilValue, names = getAttrib(list, R_NamesSymbol);
  int i;

  for (i = 0; i < length(list); i++)
    if(strcmp(CHAR(STRING(names)[i]), str) == 0) {
      elmt = VECTOR(list)[i];
      break;
    }
  return elmt;
}
```

and enables us to say

```
double g;
g = REAL(getListElement(a, "g"))[0];
```

3.6.7 Finding and setting variables

It will be usual that all the R objects needed in our C computations are passed as arguments to `.Call` or `.External`, but it is possible to find the values of R objects from within the C given their names. The following code is the equivalent of `get(name, envir = rho)`.

```
SEXP getvar(SEXP name, SEXP rho)
{
  SEXP ans;

  if(!isString(name) || length(name) != 1)
     error("name is not a single string");
  if(!isEnvironment(rho))
    error("rho should be an environment");
  ans = findVar(install(CHAR(STRING(name)[0])), rho);
  printf("first value is %f\n", REAL(ans)[0]);
  return(R_NilValue);
}
```

The main work is done by `findVar`, but to use it we need to install `name` as a name in the symbol table. As we wanted the value for internal use, we return `NULL`.

Similar functions with syntax

```
void defineVar(SEXP symbol, SEXP value, SEXP rho)
void setVar(SEXP symbol, SEXP value, SEXP rho)
```

can be used to assign values to R objects, in the specified environment frame and to perform the equivalent of `assign(x, value, envir = rho, inherits = TRUE)` respectively.

3.6.8 Changes in R version 1.2

R version 1.2.0 introduces a new "generational" garbage collector which means that strings and vectors (and language objects) are handled differently from the numerical atomic types.

Earlier code was written in a style like

```
VECTOR(dimnames)[0] = getAttrib(x, R_NamesSymbol);
```

but that is no longer allowed. The functions `VECTOR_ELT` and `SET_VECTOR_ELT` must now be used to access and set elements of a generic vector. There are analogous functions `STRING_ELT` and `SET_STRING_ELT` for character vectors.

To convert existing code, use the following replacements.

Original	**Replacement**
`foo = VECTOR(bar)[i]`	`foo = VECTOR_ELT(bar, i)`
`VECTOR(foo)[j] = bar`	`SET_VECTOR_ELT(foo, j, bar)`
`foo = STRING(bar)[i]`	`foo = STRING_ELT(bar, i)`
`STRING(foo)[j] = bar`	`SET_STRING_ELT(foo, j, bar)`

The new garbage collector requires that all assignments into fields containing `SEXP` values go through functions such as `SET_VECTOR_ELT`. Packages compiled under older versions of R may contain binary code that modifies these fields directly. These packages *must* be recompiled and re-installed. Failure to do so may confuse the memory manager and result in heap corruption.

In order to achieve maximal source compatibility in the R 1.x series, add-on packages should either use conditionals of the form

```
#if R_VERSION >= R_Version(1, 2, 0)
  new-style-code
#else
  old-style-code
#endif
```

or, maybe preferably in order to avoid unnecessarily duplicating code, use the new-style interface along with a **private** header file containing the following code:

```
#if R_VERSION < R_Version(1, 2, 0)
# define STRING_ELT(x,i)        (STRING(x)[i])
# define VECTOR_ELT(x,i)        (VECTOR(x)[i])
# define SET_STRING_ELT(x,i,v)  (STRING(x)[i] = (v))
# define SET_VECTOR_ELT(x,i,v)  (VECTOR(x)[i] = (v))
#endif
```

More extensive changes may be needed in code which manipulates language objects.

3.7 Interface functions `.Call` and `.External`

In this section we consider the details of the R/C interfaces.

These two interfaces have almost the same functionality. `.Call` is based on the interface of the same name in S version 4, and `.External` is based on `.Internal`. `.External` is more complex but allows a variable number of arguments.

3.7.1 Calling `.Call`

Let us convert our finite convolution example to use `.Call`, first using the '`Rdefines.h`' macros. The calling function in R is

```
conv <- function(a, b) .Call("convolve2", a, b)
```

which could hardly be simpler, but as we shall see all the type checking must be transferred to the C code, which is

```
#include <R.h>
#include <Rdefines.h>

SEXP convolve2(SEXP a, SEXP b)
{
  int i, j, na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  na = LENGTH(a); nb = LENGTH(b); nab = na + nb - 1;
  PROTECT(ab = NEW_NUMERIC(nab));
  xa = NUMERIC_POINTER(a); xb = NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  for(i = 0; i < nab; i++) xab[i] = 0.0;
  for(i = 0; i < na; i++)
    for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return(ab);
}
```

Note that unlike the macros in S version 4, the R versions of these macros do check that coercion can be done and raise an error if it fails. They will raise warnings if missing values are introduced by coercion. Although we illustrate doing the coercion in the C code here, it often is simpler to do the necessary coercions in the R code.

Now for the version in R-internal style. Only the C code changes.

```
#include <R.h>
#include <Rinternals.h>
SEXP convolve2(SEXP a, SEXP b)
{
  int i, j, na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;
```

```
  PROTECT(a = coerceVector(a, REALSXP));
  PROTECT(b = coerceVector(b, REALSXP));
  na = length(a); nb = length(b); nab = na + nb - 1;
  PROTECT(ab = allocVector(REALSXP, nab));
  xa = REAL(a); xb = REAL(b);
  xab = REAL(ab);
  for(i = 0; i < nab; i++) xab[i] = 0.0;
  for(i = 0; i < na; i++)
    for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return(ab);
}
```

This is called in exactly the same way.

3.7.2 Calling `.External`

We can use the same example to illustrate `.External`. The R code changes only by replacing `.Call` by `.External`

```
conv <- function(a, b) .External("convolveE", a, b)
```

but the main change is how the arguments are passed to the C code, this time as a single SEXP. The only change to the C code is how we handle the arguments.

```
#include <R.h>
#include <Rinternals.h>

SEXP convolveE(SEXP args)
{
  int i, j, na, nb, nab;
  double *xa, *xb, *xab;
  SEXP a, b, ab;

  PROTECT(a = coerceVector(CADR(args), REALSXP));
  PROTECT(b = coerceVector(CADDR(args), REALSXP));
    ...
}
```

Once again we do not need to protect the arguments, as in the R side of the interface they are objects that are already in use. The macros

```
  first = CADR(args);
  second = CADDR(args);
  third = CADDDR(args);
  fourth = CAD4R(args);
```

provide convenient ways to access the first four arguments. More generally we can use the `CDR` and `CAR` macros as in

```
  args = CDR(args); a = CAR(args);
  args = CDR(args); b = CAR(args);
```

which clearly allows us to extract an unlimited number of arguments (whereas `.Call` has a limit, albeit at 65 not a small one).

More usefully, the `.External` interface provides an easy way to handle calls with a variable number of arguments, as `length(args)` will give the number of arguments supplied (of which the first is ignored). We may need to know the names ('tags') given to the actual arguments, which we can by using the `TAG` macro and using something like the following example, that prints the names and the first value of its arguments if they are vector types.

```
#include "R_ext/PrtUtil.h"

SEXP showArgs(SEXP args)
{
  int i, nargs;
  Rcomplex cpl;
  char *name;

  if((nargs = length(args) - 1) > 0) {
    for(i = 0; i < nargs; i++) {
      args = CDR(args);
      name = CHAR(PRINTNAME(TAG(args)));
      switch(TYPEOF(CAR(args))) {
      case REALSXP:
        Rprintf("[%d] '%s' %f\n", i+1, name, REAL(CAR(args))[0]);
        break;
      case LGLSXP:
      case INTSXP:
        Rprintf("[%d] '%s' %d\n", i+1, name, INTEGER(CAR(args))[0]);
        break;
      case CPLXSXP:
        cpl = COMPLEX(CAR(args))[0];
        Rprintf("[%d] '%s' %f + %fi\n", i+1, name, cpl.r, cpl.i);
        break;
      case STRSXP:
        Rprintf("[%d] '%s' %s\n", i+1, name,
                CHAR(STRING(CAR(args))[0]));
        break;
      default:
        Rprintf("[%d] '%s' R type\n", i+1, name);
      }
    }
  }
  return(R_NilValue);
}
```

This can be called by the wrapper function

```
showArgs <- function(...) .External("showArgs", ...)
```

Note that this style of programming is convenient but not necessary, as an alternative style is

```
showArgs <- function(...) .Call("showArgs1", list(...))
```

3.7.3 Missing and special values

One piece of error-checking the `.C` call does (unless `NAOK` is true) is to check for missing (`NA`) and IEEE special values (`Inf`, `-Inf` and `NaN`) and give an error if any are found. With the `.Call` interface these will be passed to our code. In this example the special values are no problem, as IEEE arithmetic will handle them correctly. In the current implementation this is also true of `NA` as it is a type of `NaN`, but it is unwise to rely on such details. Thus we will re-write the code to handle `NA`s using macros defined in '`Arith.h`' included by '`R.h`'.

The code changes are the same in any of the versions of `convolve2` or `convolveE`:

```
  ...
for(i = 0; i < na; i++)
  for(j = 0; j < nb; j++)
      if(ISNA(xa[i]) || ISNA(xb[j]) || ISNA(xab[i + j]))
        xab[i + j] = NA_REAL;
      else
        xab[i + j] += xa[i] * xb[j];
  ...
```

Note that the `ISNA` macro, and the similar macros `ISNAN` (which checks for `NaN` or `NA`) and `R_FINITE` (which is false for `NA` and all the special values), only apply to numeric values of type `double`. Missingness of integers, logicals and character strings can be tested by equality to the constants `NA_INTEGER`, `NA_LOGICAL` and `NA_STRING`. These and `NA_REAL` can be used to set elements of R vectors to `NA`.

The constants `R_NaN`, `R_PosInf`, `R_NegInf` and `R_NaReal` can be used to set `double`s to the special values.

3.8 Evaluating R expressions from C

We noted that the `call_R` interface could be used to evaluate R expressions from C code, but the current interfaces are much more convenient to use. The main function we will use is

```
SEXP eval(SEXP expr, SEXP rho);
```

the equivalent of the interpreted R code `eval(expr, envir = rho)`, although we can also make use of `findVar`, `defineVar` and `findFun` (which restricts the search to functions).

To see how this might be applied, here is a simplified internal version of `lapply` for expressions, used as

```
a <- list(a = 1:5, b = rnorm(10), test = runif(100))
.Call("lapply", a, quote(sum(x)), new.env())
```

with C code

```
SEXP lapply(SEXP list, SEXP expr, SEXP rho)
{
  int i, n = length(list);
  SEXP ans;

  if(!isNewList(list)) error("'list' must be a list");
  if(!isEnvironment(rho)) error("'rho' should be an environment");
  PROTECT(ans = allocVector(VECSXP, n));
```

```
  for(i = 0; i < n; i++) {
    defineVar(install("x"), VECTOR(list)[i], rho);
    VECTOR(ans)[i] = eval(expr, rho);
  }
  setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
  UNPROTECT(1);
  return(ans);
}
```

It would be closer to `lapply` if we could pass in a function rather than an expression. One way to do this is *via* interpreted R code as in the next example, but it is possible (if somewhat obscure) to do this in C code. The following is based on the code in '`src/main/optimize.c`'.

```
SEXP lapply2(SEXP list, SEXP fn, SEXP rho)
{
  int i, n = length(list);
  SEXP R_fcall, ans;

  if(!isNewList(list)) error("'list' must be a list");
  if(!isFunction(fn)) error("'fn' must be a function");
  if(!isEnvironment(rho)) error("'rho' should be an environment");
  PROTECT(R_fcall = lang2(fn, R_NilValue));
  PROTECT(ans = allocVector(VECSXP, n));
  for(i = 0; i < n; i++) {
    CADR(R_fcall) = VECTOR(list)[i];
    VECTOR(ans)[i] = eval(R_fcall, rho);
  }
  setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
  UNPROTECT(2);
  return(ans);
}
```

used by

```
.Call("lapply2", a, sum, new.env())
```

Function `lang2` creates an executable 'list' of two elements, but this will only be clear to those with a knowledge of a LISP-like language.

3.8.1 Zero-finding

In this section we re-work the example in '`demos/dynload`' of `call_R` (based on that for `call_S` in Becker, Chambers & Wilks (1988)) on finding a zero of a univariate function. The R code and an example are

```
zero <- function(f, guesses, tol = 1e-7) {
  f.check <- function(x) {
    x <- f(x)
    if(!is.numeric(x)) stop("Need a numeric result")
    as.double(x)
  }
  .Call("zero", body(f.check), as.double(guesses), as.double(tol),
        new.env())
}
```

```
cube1 <- function(x) (x^2 + 1) * (x - 1.5)
zero(cube1, c(0, 5))
```

where this time we do the coercion and error-checking in the R code. The C code is

```
SEXP mkans(double x)
{
    SEXP ans;
    PROTECT(ans = allocVector(REALSXP, 1));
    REAL(ans)[0] = x;
    UNPROTECT(1);
    return ans;
}

double feval(double x, SEXP f, SEXP rho)
{
    defineVar(install("x"), mkans(x), rho);
    return(REAL(eval(f, rho))[0]);
}

SEXP zero(SEXP f, SEXP guesses, SEXP stol, SEXP rho)
{
    double x0 = REAL(guesses)[0], x1 = REAL(guesses)[1],
           tol = REAL(stol)[0];
    double f0, f1, fc, xc;

    if(tol <= 0.0) error("non-positive tol value");
    f0 = feval(x0, f, rho); f1 = feval(x1, f, rho);
    if(f0 == 0.0) return mkans(x0);
    if(f1 == 0.0) return mkans(x1);
    if(f0*f1 > 0.0) error("x[0] and x[1] have the same sign");
    for(;;) {
        xc = 0.5*(x0+x1);
        if(fabs(x0-x1) < tol) return  mkans(xc);
        fc = feval(xc, f, rho);
        if(fc == 0) return  mkans(xc);
        if(f0*fc > 0.0) {
            x0 = xc; f0 = fc;
        } else {
            x1 = xc; f1 = fc;
        }
    }
}
```

The C code is essentially unchanged from the `call_R` version, just using a couple of functions to convert from `double` to `SEXP` and to evaluate `f.check`.

3.8.2 Calculating numerical derivatives

We will use a longer example (by Saikat DebRoy) to illustrate the use of evaluation and `.External`. This calculates numerical derivatives, something that could be done as effectively in interpreted R code but may be needed as part of a larger C calculation.

An interpreted R version and an example are

```
numeric.deriv <- function(expr, theta, rho=sys.frame(sys.parent()))
{
  eps <- sqrt(.Machine$double.eps)
  ans <- eval(substitute(expr), rho)
  grad <- matrix(,length(ans), length(theta),
                 dimnames=list(NULL, theta))
  for (i in seq(along=theta)) {
    old <- get(theta[i], envir=rho)
    delta <- eps * min(1, abs(old))
    assign(theta[i], old+delta, envir=rho)
    ans1 <- eval(substitute(expr), rho)
    assign(theta[i], old, envir=rho)
    grad[, i] <- (ans1 - ans)/delta
  }
  attr(ans, "gradient") <- grad
  ans
}
omega <- 1:5; x <- 1; y <- 2
numeric.deriv(sin(omega*x*y), c("x", "y"))
```

where `expr` is an expression, `theta` a character vector of variable names and `rho` the environment to be used.

For the compiled version the call from R will be

`.External("numeric_deriv",` *expr*`,` *theta*`,` *rho*`)`

with example usage

```
.External("numeric_deriv", quote(sin(omega*x*y)),
          c("x", "y"), .GlobalEnv)
```

Note the need to quote the expression to stop it being evaluated.

Here is the complete C code which we will explain section by section.

```
#include <R.h> /* for DOUBLE_EPS */
#include <Rinternals.h>

SEXP numeric_deriv(SEXP args)
{
  SEXP theta, expr, rho, ans, ans1, gradient, par, dimnames;
  double tt, xx, delta, eps = sqrt(DOUBLE_EPS);
  int start, i, j;

  expr = CADR(args);
  if(!isString(theta = CADDR(args)))
    error("theta should be of type character");
  if(!isEnvironment(rho = CADDDR(args)))
```

```
    error("rho should be an environment");

  PROTECT(ans = coerceVector(eval(expr, rho), REALSXP));
  PROTECT(gradient = allocMatrix(REALSXP, LENGTH(ans), LENGTH(theta)));

  for(i = 0, start = 0; i < LENGTH(theta); i++, start += LENGTH(ans)) {
    PROTECT(par = findVar(install(CHAR(STRING(theta)[i])), rho));
    tt = REAL(par)[0];
    xx = fabs(tt);
    delta = (xx < 1) ? eps : xx*eps;
    REAL(par)[0] += delta;
    PROTECT(ans1 = coerceVector(eval(expr, rho), REALSXP));
    for(j = 0; j < LENGTH(ans); j++)
      REAL(gradient)[j + start] =
        (REAL(ans1)[j] - REAL(ans)[j])/delta;
    REAL(par)[0] = tt;
    UNPROTECT(2); /* par, ans1 */
  }

  PROTECT(dimnames = allocVector(VECSXP, 2));
  VECTOR(dimnames)[1] = theta;
  dimnamesgets(gradient, dimnames);
  setAttrib(ans, install("gradient"), gradient);
  UNPROTECT(3); /* ans  gradient  dimnames */
  return ans;
}
```

The code to handle the arguments is

```
expr = CADR(args);
if(!isString(theta = CADDR(args)))
  error("theta should be of type character");
if(!isEnvironment(rho = CADDDR(args)))
  error("rho should be an environment");
```

Note that we check for correct types of `theta` and `rho` but do not check the type of `expr`. That is because `eval` can handle many types of R objects other than `EXPRSXP`. There is no useful coercion we can do, so we stop with an error message if the arguments are not of the correct mode.

The first step in the code is to evaluate the expression in the environment `rho`, by

```
PROTECT(ans = coerceVector(eval(expr, rho), REALSXP));
```

We then allocate space for the calculated derivative by

```
PROTECT(gradient = allocMatrix(REALSXP, LENGTH(ans), LENGTH(theta)));
```

The first argument to `allocMatrix` gives the `SEXPTYPE` of the matrix: here we want it to be `REALSXP`. The other two arguments are the numbers of rows and columns.

```
for(i = 0, start = 0; i < LENGTH(theta); i++, start += LENGTH(ans)) {
  PROTECT(par = findVar(install(CHAR(STRING(theta)[i])), rho));
```

Here, we are entering a for loop. We loop through each of the variables. In the `for` loop, we first create a symbol corresponding to the `i`'th element of the `STRSXP theta`. Here,

`STRING(theta)[i]` accesses the `i`'th element of the `STRSXP` `theta`. Macro `CHAR()` extracts the actual character representation of it: it returns a pointer. We then install the name and use `findVar` to find its value.

```
        tt = REAL(par)[0];
        xx = fabs(tt);
        delta = (xx < 1) ? eps : xx*eps;
        REAL(par)[0] += delta;
        PROTECT(ans1 = coerceVector(eval(expr, rho), REALSXP));
```

We first extract the real value of the parameter, then calculate delta, the increment to be used for approximating the numerical derivative. Then we change the value stored in `par` (in environment `rho`) by `delta` and evaluate `expr` in environment `rho` again. Because we are directly dealing with original R memory locations here, R does the evaluation for the changed parameter value.

```
        for(j = 0; j < LENGTH(ans); j++)
          REAL(gradient)[j + start] =
            (REAL(ans1)[j] - REAL(ans)[j])/delta;
        REAL(par)[0] = tt;
        UNPROTECT(2);
      }
```

Now, we compute the `i`'th column of the gradient matrix. Note how it is accessed: R stores matrices by column (like Fortran).

```
      PROTECT(dimnames = allocVector(VECSXP, 2));
      VECTOR(dimnames)[1] = theta;
      dimnamesgets(gradient, dimnames);
      setAttrib(ans, install("gradient"), gradient);
      UNPROTECT(3);
      return ans;
    }
```

First we add column names to the gradient matrix. This is done by allocating a list (a `VECSXP`) whose first element, the row names, is `NULL` (the default) and the second element, the column names, is set as `theta`. This list is then assigned as the attribute having the symbol `R_DimNamesSymbol`. Finally we set the gradient matrix as the gradient attribute of `ans`, unprotect the remaining protected locations and return the answer `ans`.

3.9 Debugging compiled code

Sooner or later programmers will be faced with the need to debug compiled code loaded into R. Some "tricks" are worth knowing.

3.9.1 Finding entry points in dynamically loaded code

Under most compilation environments, compiled code dynamically loaded into R cannot have breakpoints set within it until it is loaded. To use a symbolic debugger on such dynamically loaded code under UNIX use

- Call the debugger on the R executable, for example by *`R -d gdb`*.
- Start R.

- At the R prompt, use `dyn.load` or `library` to load your library.
- Send an interrupt signal. This will put you back to the debugger prompt.
- Set the breakpoints in your code.
- Continue execution of R by typing *`signal 0`*⟨RET⟩.

Under Windows the R engine is itself in a DLL, and the procedure is

- Start R under the debugger after setting a breakpoint for `WinMain`.

```
gdb .../bin/Rgui.exe
(gdb) break WinMain
(gdb) run
[ stops with R.dll loaded ]
(gdb) break R_ReadConsole
(gdb) continue
[ stops with console running ]
(gdb) continue
```

- At the R prompt, use `dyn.load` or `library` to load your library.
- Set the breakpoints in your code.
- Use

```
(gdb) clear R_ReadConsole
(gdb) continue
```

  to continue running with the breakpoints set.

Windows has little support for signals, so the usual idea of running a program under a debugger and sending it a signal to interrupt it and drop control back to the debugger only works with some debuggers.

3.9.2 Inspecting R objects when debugging

The key to inspecting R objects from compiled code is the function `PrintValue(SEXP s)` which uses the normal R printing mechanisms to print the R object pointed to by `s`, or the safer version `R_PV(SEXP s)` which will only print 'objects'.

One way to make use to `PrintValue` is to insert suitable calls into the code to be debugged.

Another way is to call `R_PV` from the symbolic debugger. (`PrintValue` is hidden as `Rf_PrintValue`.) For example, from `gdb` we can use

```
(gdb) p R_PV(ab)
```

using the object `ab` from the convolution example, if we have placed a suitable breakpoint in the convolution C code.

To examine an arbitrary R object we need to work a little harder. For example, let

```
R> DF <- data.frame(a = 1:3, b = 4:6)
```

By setting a breakpoint at `do_get` and typing *`get("DF")`* at the R prompt, one can find out the address in memory of `DF`, for example

```
Value returned is $1 = (SEXPREC *) 0x40583e1c
(gdb) p *$1
$2 = {
```

```
  sxpinfo = {type = 19, obj = 1, named = 1, gp = 0,
    mark = 0, debug = 0, trace = 0, = 0},
  attrib = 0x40583e80,
  u = {
    vecsxp = {
      length = 2,
      type = {c = 0x40634700 "0>X@D>X@0>X@", i = 0x40634700,
        f = 0x40634700, z = 0x40634700, s = 0x40634700},
      truelength = 1075851272,
    },
    primsxp = {offset = 2},
    symsxp = {pname = 0x2, value = 0x40634700, internal = 0x40203008},
    listsxp = {carval = 0x2, cdrval = 0x40634700, tagval = 0x40203008},
    envsxp = {frame = 0x2, enclos = 0x40634700},
    closxp = {formals = 0x2, body = 0x40634700, env = 0x40203008},
    promsxp = {value = 0x2, expr = 0x40634700, env = 0x40203008}
  }
}
```

(Debugger output reformatted for better legibility).

Using `R_PV()` one can "inspect" the values of the various elements of the SEXP, for example,

```
(gdb) p R_PV($1->attrib)
$names
[1] "a" "b"

$row.names
[1] "1" "2" "3"

$class
[1] "data.frame"

$3 = void
```

To find out where exactly the corresponding information is stored, one needs to go "deeper":

```
(gdb) set $a = $1->attrib
(gdb) p $a->u.listsxp.tagval->u.symsxp.pname->u.vecsxp.type.c
$4 = 0x405d40e8 "names"
(gdb) p $a->u.listsxp.carval->u.vecsxp.type.s[1]->u.vecsxp.type.c
$5 = 0x40634378 "b"
(gdb) p $1->u.vecsxp.type.s[0]->u.vecsxp.type.i[0]
$6 = 1
(gdb) p $1->u.vecsxp.type.s[1]->u.vecsxp.type.i[1]
$7 = 5
```

4 The R API: entry points for C code

There are a large number of entry points in the R executable/DLL that can be called from C code (and some that can be called from Fortran code). Only those documented here are stable enough that they will only be changed with considerable notice.

The recommended procedure to use these is to include the header file '`R.h`' in your C code by

```
#include <R.h>
```

This will include a number of other header files from the directory '`$R_HOME/include/R_ext`', and there are other header files there that can be included too, but many of the features they contain should be regarded as undocumented and unstable.

An alternative is to include the header file '`S.h`', which may be useful when porting code from S. This includes rather less than '`R.h`', and has some compatibility definitions (for example the `S_complex` type from S).

> **NOTE:** Because R re-maps many of its external names to avoid clashes with user code, it is *essential* to include the appropriate header files when using these entry points.

4.1 Memory allocation

There are two types of memory allocation available to the C programmer, one in which R manages the clean-up and the other in which user has full control (and responsibility).

4.1.1 Transient storage allocation

Here R will reclaim the memory at the end of the call to `.C`. Use

```
char* R_alloc(long n, int size)
```

which allocates *n* units of *size* bytes each. A typical usage (from package **mva**) is

```
x = (int *) R_alloc(nrows(merge)+2, sizeof(int));
```

There is a similar call, `S_alloc`, for compatibility with S which differs only in zeroing the memory allocated, and

```
S_realloc(char *p, long new, long old, int size)
```

which changes the allocation size from *old* to *new* units, and zeroes the additional units.

This memory is taken from the heap (the region whose size is set by `--vsize`), and released at the end of the `.C`, `.Call` or `.External` call. Users can also manage it, by noting the current position with a call to `vmaxget` and clearing memory allocated subsequently by a call to `vmaxset`. This is only recommended for experts.

4.1.2 User-controlled memory

The other form of memory allocation is an interface to `malloc`, the interface providing R error handling. This memory lasts until freed by the user and is additional to the memory allocated for the R workspace.

The interface functions are

```
type* Calloc(size_t n, type)
type* Realloc(any *p, size_t n, type)
void Free(any *p)
```

providing analogues of `calloc`, `realloc` and `free`. If there is an error it is handled by R, so if these routines return the memory has been successfully allocated or freed. `Free` will set the pointer `p` to `NULL`. (Some but not all versions of S do so.)

4.2 Error handling

The basic error handling routines are the equivalents of `stop` and `warning` in R code, and use the same interface.

```
void error(const char * format, ...);
void warning(const char * format, ...);
```

These have the same call sequences as calls to `printf`, but in the simplest case can be called with a single character string argument giving the error message.

There is also an S-compatibility interface which uses calls of the form

```
PROBLEM ..... ERROR
MESSAGE ..... WARN
PROBLEM ..... RECOVER(NULL_ENTRY)
MESSAGE ..... WARNING(NULL_ENTRY)
```

the last two being the forms available in all S versions. Here `.....` is a set of arguments to `printf`, so can be a string or a format string followed by arguments separated by commas.

4.3 Random number generation

The interface to R's internal random number generation routines is

```
double unif_rand();
double norm_rand();
double exp_rand();
```

giving one uniform, normal or exponential pseudo-random variate. However, before these are used, the user must call

```
GetRNGstate();
```

and after all the required variates have been generated, call

```
PutRNGstate();
```

These essentially read in (or create) `.Random.seed` and write it out after use.

File '`S.h`' defines `seed_in` and `seed_out` for S-compatibility rather than `GetRNGstate` and `PutRNGstate`. These take a `long *` argument which is ignored.

The random number generator is private to R; there is no way to select the kind of RNG or set the seed except by evaluating calls to the R functions.

The C code behind R's `rxxx` functions can be accessed by including the header file '`R_ext/Mathlib.h`'; See Section 4.8 [Distribution functions], page 44. Those calls generate a single variate and should also be enclosed in calls to `GetRNGstate` and `PutRNGstate`.

4.4 Missing and IEEE special values

It is possible to compile R on a platform without IEEE 754-compatible arithmetic, so users should not assume that it is available. Rather a set of functions is provided to test for `NA`, `Inf`, `-Inf` (which exists on all platforms) and `NaN`. These functions are accessed via macros:

`ISNA(`*x*`)`	True for R's `NA` only
`ISNAN(`*x*`)`	True for R's `NA` and IEEE `NaN`
`R_FINITE(`*x*`)`	False for `Inf`, `-Inf`, `NA`, `NaN`

and function `R_IsNaN` is true for `NaN` but not `NA`. Do use these rather than `isnan` or `finite`; the latter in particular is often mendacious.

You can check for `Inf` or `-Inf` by testing equality to `R_PosInf` or `R_NegInf`, and set (but not test) an `NA` as `NA_REAL`.

All of the above apply to *double* variables only. For integer variables there is a variable accessed by the macro `NA_INTEGER` which can used to set or test for missingness.

Beware that these special values may be represented by extreme values which could occur in ordinary computations which run out of control, so you may need to test that they have not been generated inadvertently.

4.5 Printing

The most useful function for printing from a C routine compiled into R is `Rprintf`. This is used in exactly the same way as `printf`, but is guaranteed to write to R's output (which might be a GUI console rather than a file). It is wise to write complete lines (including the `"\n"`) before returning to R.

The function `REprintf` is similar but writes on the error stream (`stderr`) which may or may not be different from the standard output stream. Functions `Rvprintf` and `REvprintf` are the analogues using the `vprintf` interface.

These routines are declared in '`R_ext/PrtUtil.h`' which is not included by '`R.h`'.

4.5.1 Printing from Fortran

In theory Fortran `write` and `print` statements can be used, but the output may not interleave well with that of C, and will be invisible on GUI interfaces. They are best avoided.

Three subroutines are provided to ease the output of information from Fortran code.

```
subroutine dblepr(label, nchar, data, ndata)
subroutine realpr(label, nchar, data, ndata)
subroutine intpr (label, nchar, data, ndata)
```

Here *label* is a character label of up to 255 characters, *nchar* is its length (which can be `-1` if the whole label is to be used), and *data* is an array of length at least *ndata* of the appropriate type (`double precision`, `real` and `integer` respectively). These routines print the label on one line and then print *data* as if it were an R vector on subsequent line(s). They work with zero *ndata*, and so can be used to print a label alone.

4.6 Calling C from Fortran and vice versa

Naming conventions for symbols generated by Fortran differ by platform: it is not safe to assume that Fortran names appear to C with a trailing underscore. To help cover up the platform-specific differences there is a set of macros that should be used.

`F77_SUB(`*name*`)`
: to define a function in C to be called from Fortran

`F77_NAME(`*name*`)`
: to declare a Fortran routine in C before use

`F77_CALL(`*name*`)`
: to call a Fortran routine from C

`F77_COMDECL(`*name*`)`
: to declare a Fortran common block in C

`F77_COM(`*name*`)`
: to access a Fortran common block from C

4.7 Numerical analysis subroutines

R contains a large number of mathematical functions for its own use, for example numerical linear algebra computations and special functions.

The header file '`R_ext/Linpack.h`' contains details of the LINPACK and EISPACK linear algebra functions include in R. These are expressed as calls to Fortran subroutines, and they will also be usable from users' Fortran code. Although not part of the official API, this set of subroutines is unlikely to change (but might be supplemented).

The header file '`R_ext/Mathlib.h`' lists other undocumented calls that are currently available, mainly special functions (such gamma, beta and Bessel functions). These *are* subject to change.

4.8 Distribution functions

The routines used to calculate densities, cumulative distribution functions and quantile functions for the standard statistical distributions are available as entry points.

The arguments for the entry points follow the pattern of those for the normal distribution:

```
double dnorm(double x, double mu, double sigma, int give_log);
double pnorm(double x, double mu, double sigma, int lower_tail,
             int give_log);
double qnorm(double p, double mu, double sigma, int lower_tail,
             int log_p);
double rnorm(double mu, double sigma);
```

That is, the first argument gives the position for the density and CDF and probability for the quantile function, followed by the distribution's parameters. Argument *lower_tail* should be `1` (or `LTRUE`) for normal use, but can be `0` (or `LFALSE`) if the probability of the upper tail is desired or specified.

Note that you directly get the cumulative (or "integrated") *hazard* function, $H(t) = -\log(1 - F(t))$, by using

```
- pdist(t, ..., /*lower_tail = */ LFALSE, /* give_log = */ LTRUE)
```

or shorter (and more cryptic) `- pdist(t, ..., 0, 1)`.

Finally, *give_log* should be non-zero if the result is required on log scale, and *log_p* should be non-zero if *p* has been specified on log scale.

The random-variate generation routine `rnorm` returns one normal variate. See Section 4.3 [Random numbers], page 42, for the protocol in using the random-variate routines.

Note that these argument sequences are (apart from the names and that `rnorm` has no *n*) exactly the same as the corresponding R functions of the same name, so the documentation of the R functions can be used.

For reference, the following table gives the basic name (to be prefixed by '`d`', '`p`', '`q`' or '`r`' apart from the exceptions noted) and distribution-specific arguments for the complete set of distributions.

beta	`beta`	`a, b`
non-central beta	`nbeta`	`a, b, lambda`
binomial	`binom`	`n, p`
Cauchy	`cauchy`	`location, scale`
chi-square	`chisq`	`df`
non-central chi-square	`nchisq`	`df, lambda`
exponential	`exp`	`scale`
F	`f`	`n1, n2`
non-central F	`{p,q}nf`	`n1, n2, ncp`
gamma	`gamma`	`shape, scale`
geometric	`geom`	`p`
hypergeometric	`hyper`	`NR, NB, n`
logistic	`logis`	`location, scale`
lognormal	`lnorm`	`logmean, logsd`
negative binomial	`nbinom`	`n, p`
normal	`norm`	`mu, sigma`
Poisson	`pois`	`lambda`
Student's t	`t`	`n`
non-central t	`{p,q}nt`	`df, delta`
Studentized range	`{p,q}tukey`	`rr, cc, df`
uniform	`unif`	`a, b`
Weibull	`weibull`	`shape, scale`
Wilcoxon rank sum	`wilcox`	`m, n`
Wilcoxon signed rank	`signrank`	`n`

The argument sequences are not all quite the same as the R ones.

4.9 Mathematical Utilities

There are a few other numerical utilities available as entry points.

```
double R_pow(double x, double y);
double R_pow_di(double x, int i);
```

`R_pow(x, y)` and `R_pow_di(x, i)` compute `x ^ y` and `x ^ i`, respectively using `R_FINITE` checks and returning the proper result (the same as R) for the cases where *x*, *y* or *i* are 0 or missing or infinite or `NaN`.

`double pythag(double` *a*`, double` *b*`);`
: `pythag(`*a*`,` *b*`)` computes `sqrt(`*a*`^2 +` *b*`^2)` without overflow or destructive underflow: for example it still works when both *a* and *b* are between `1e200` and `1e300` (in IEEE double precision).

`double log1p(`*x*`)`
: Computes `log(1 +` *x*`)` (*log 1* **p***lus x*), accurately even for small *x*, i.e. $|x| \ll 1$.

`int imax2(int` *x*`, int` *y*`);`
`int imin2(int` *x*`, int` *y*`);`
`double fmax2(double` *x*`, double` *y*`);`
`double fmin2(double` *x*`, double` *y*`);`
: Return the larger (`max`) or smaller (`min`) of two integer or double numbers, respectively.

`double sign(double` *x*`);`
: Compute the *signum* function, where sign(*x*) is 1, 0, or -1, when *x* is positive, 0, or negative, respectively.

`double fsign(double` *x*`, double` *y*`);`
: Performs "transfer of sign" and is defined as $|x|\text{sign}(y)$.

`double fprec(double` *x*`, double` *digits*`);`
: Returns the value of *x* rounded to *digits* decimal digits (after the decimal point).
This is the function used by R's `round()`.

`double fround(double` *x*`, double` *digits*`);`
: Returns the value of *x* rounded to *digits significant* decimal digits.
This is the function used by R's `signif()`.

4.10 Mathematical Constants

R has a set of commonly used mathematical constants encompassing constants usually found '`math.h`' and contains further ones that are used in statistical computations. All these are defined to (at least) 30 digits accuracy in '`R_ext/Mathlib.h`'. The following definitions use `ln(x)` for the natural logarithm (`log(x)` in R).

Name	**Definition (`ln` = `log`)**	**round(*value*, 7)**
`M_E`	= e	2.7182818
`M_LOG2E`	= log2(e)	1.4426950
`M_LOG10E`	= log10(e)	0.4342945
`M_LN2`	= ln(2)	0.6931472
`M_LN10`	= ln(10)	2.3025851
`M_PI`	= pi	3.1415927
`M_PI_2`	= pi/2	1.5707963
`M_PI_4`	= pi/4	0.7853982
`M_1_PI`	= 1/pi	0.3183099
`M_2_PI`	= 2/pi	0.6366198
`M_2_SQRTPI`	= 2/sqrt(pi)	1.1283792
`M_SQRT2`	= sqrt(2)	1.4142136

M_SQRT1_2	= 1/sqrt(2)	0.7071068
M_SQRT_32	= sqrt(32)	5.6568542
M_LOG10_2	= log10(2)	0.3010300
M_SQRT_PI	= sqrt(pi)	1.7724539
M_1_SQRT_2PI	= 1/sqrt(2pi)	0.3989423
M_SQRT_2dPI	= sqrt(2/pi)	0.7978846
M_LN_SQRT_PI	= ln(sqrt(pi))	0.5723649
M_LN_SQRT_2PI	= ln(sqrt(2*pi))	0.9189385
M_LN_SQRT_PId2	= ln(sqrt(pi/2))	0.2257914

There are a set of constants (for example `PI`, `DOUBLE_EPS`) defined in the included header '`R_ext/Constants.h`', mainly for compatibility with S.

4.11 Utility functions

R has a fairly comprehensive set of sort routines which are made available to users' C code. These include the following.

`void R_isort(int* x, int n);`
`void R_rsort(double* x, int n);`
`void R_csort(Rcomplex* x, int n);`
`void rsort_with_index(double* x, int* index, int n);`

The first three sort integer, real (double) and complex data respectively. (Complex numbers are sorted by the real part first then the imaginary part.)

`rsort_with_index` sorts on *x*, and applies the same permutation to *index*.

`void revsort(double* x, int* index, int n);`

is similar to `rsort_with_index` but sorts into decreasing order.

`void iPsort(int* x, int n, int k);`
`void rPsort(double* x, int n, int k);`
`void cPsort(Rcomplex* x, int n, int k);`

all provide (very) partial sorting: they permute *x* so that *x*`[k]` is in the correct place with smaller values to the left, larger ones to the right.

Further utilities in R include

`void R_max_col(double *matrix, int *nr, int *nc, int *maxes)`

Given the `nr` by `ny` matrix `matrix` in row("Fortran") order, `R_max_col()` returns in `maxes[i-1]` the column number of the maximal element in the i-th row (the same as R's `max.col()` function).

4.12 Version information

The header files define `USING_R`, which should be used to test if the code is indeed being used with R.

Header file '`Rversion.h`' (included by '`R.h`') defines a macro `R_VERSION` giving the version number encoded as an integer, plus a macro `R_Version` to do the encoding. This can be used to test if the version of R is late enough, or to include back-compatibility features. For protection against earlier versions of R which did not have this macro, use a construction such as

```
#if defined(R_VERSION) && R_VERSION >= R_Version(0, 99, 0)
  ...
#endif
```

More detailed information is available in the macros `R_MAJOR`, `R_MINOR`, `R_YEAR`, `R_MONTH` and `R_DAY`: see the header file '`Rversion.h`' for their format. Note that the minor version includes the patchlevel (as in `99.0`).

4.13 Using these functions in your own C code

It is possible to build `Mathlib`, the R set of mathematical functions documented in '`R_ext/Mathlib.h`', as a standalone library '`libRmath`' under Unix and Windows. This includes the functions documented in Section 4.8 [Distribution functions], page 44 and Section 4.9 [Mathematical Utilities], page 45 and the constants in Section 4.10 [Mathematical Constants], page 46.

The library is not built automatically when R is installed, but can be built in the directory '`src/nmath/standalone`'. See the file '`README`' there. To use the code in your own C program include

```
#define MATHLIB_STANDALONE
#include "R_ext/Mathlib.h"
```

and link against `-lRmath`. There is an example file '`test.c`'.

A little care is needed to use the random-number routines. You will need to supply the uniform random number generator

```
double unif_rand(void)
```

or use the one supplied (and with a shared library or DLL you will have to use the one supplied, which is the Marsaglia-multicarry with an entry point

```
set_seed(unsigned int, unsigned int)
```

to set its seeds).

Appendix A R (internal) programming miscellania

A.1 `.Internal` and `.Primitive`

C code compiled into R at build time can be called "directly" or via the `.Internal` interface, which is very similar to the `.External` interface except in syntax. More precisely, R maintains a table of R function names and corresponding C functions to call, which by convention all start with '`do_`' and return a SEXP. Via this table (`R_FunTab` in file '`src/main/names.c`') one can also specify how many arguments to a function are required or allowed, whether the arguments are to be evaluated before calling or not, and whether the function is "internal" in the sense that it must be accessed via the `.Internal` interface, or directly accessible in which case it is printed in R as `.Primitive`.

R's functionality can also be extended by providing corresponding C code and adding to this function table.

In general, all such functions use `.Internal()` as this is safer and in particular allows for transparent handling of named and default arguments. For example, `axis` is defined as

```
axis <- function(side, at = NULL, labels = NULL, ...)
    .Internal(axis(side, at, labels, ...))
```

However, for reasons of convenience and also efficiency (as there is some overhead in using the `.Internal` interface), there are exceptions which can be accessed directly. Note that these functions make no use of R code, and hence are very different from the usual interpreted functions. In particular, `args` and `body` return `NULL` for such objects.

These "primitive" functions are fully specified as follows.

1. "Special functions" which really are *language* elements, however exist as "primitive" functions in R:

   ```
   {       (         if       for  while  repeat  break  next
   return  function  on.exit
   ```

2. Basic *operator*s (i.e., functions usually *not* called as `foo(a, b, ...)`) for subsetting, assignment, arithmetic and logic. These are the following 1-, 2-, and N-argument functions:

   ```
              [    [[    $
   <-    <<-  [<-  [[<-  $<-
   ---------------------------------------
   +     -    *    /     ^    %%   %*%  %/%
   <     <=   ==   !=    >=   >
   |     ||   &    &&    !
   ```

3. "Low level" 0- and 1-argument functions which belong to one of the following groups of functions:

   a. Basic mathematical functions with a single argument, i.e.,

      ```
      sign    abs
      floor   ceiling
      ----------------------
      sqrt    exp
      cos     sin      tan
      ```

```
acos    asin     atan
cosh    sinh     tanh
acosh   asinh    atanh
---------------------
cumsum  cumprod
cummax  cummin
---------------------
Im      Re
Arg     Conj     Mod
```

Note however that the R function `log` has an optional named argument `base`, and therefore is defined as

```
log <- function(x, base = exp(1)) {
  if(missing(base))
    .Internal(log(x))
  else
    .Internal(log(x, base))
}
```

in order to ensure that `log(x = pi, base = 2)` is identical to `log(base = 2, x = pi)`.

b. Functions rarely used outside of "programming" (i.e., mostly used inside other functions), such as

```
nargs        missing
interactive  is.xxx
.Primitive   .Internal   .External
symbol.C     symbol.For
globalenv    pos.to.env  unclass
```

(where *xxx* stands for almost 30 different notions, such as `function`, `vector`, `numeric`, and so forth).

c. The programming and session management utilities

```
debug    undebug    trace  untrace
browser  proc.time
```

4. The following basic assignment and extractor functions

```
.Alias      environment<-
length      length<-
class       class<-
attr        attr<-
attributes  attributes<-
dim         dim<-
dimnames    dimnames<-
```

5. The following few N-argument functions are "primitive" for efficiency reasons. Care is taken in order to treat named arguments properly:

```
:          ~          c           list        unlist
call       as.call    expression  substitute
UseMethod  invisible
.C         .Fortran   .Call
```

A.2 Testing R code

When you (as R developer) add new functions to the R base (all the packages distributed with R), be careful to check if `make test-Specific` or particularly, `cd tests; make no-segfault.Rout` still works (without interactive user intervention, and on a standalone computer). If the new function, for example, accesses the Internet, or requires GUI interaction, please add its name to the "stop list" in '`tests/make-no-segfault.R`'.

Appendix B R coding standards

R is meant to run on a wide variety of platforms, including Linux and most variants of Unix as well as 32-bit Windows versions and eventually (again) on the Power Mac. Therefore, when extending R by either adding to the R base distribution or by providing an add-on package, one should not rely on features specific to only a few supported platforms, if this can be avoided. In particular, although most R developers use GNU tools, they should not employ the GNU extensions to standard tools. Whereas some other software packages explicitly rely on e.g. GNU make or the GNU C++ compiler, R does not. Nevertheless, R is a GNU project, and the spirit of the *GNU Coding Standards* should be followed if possible.

The following tools can "safely be assumed" for R extensions.

- An ANSI C compiler. If you do not have access to the ANSI standard, refer to the 2nd edition of Brian W. Kernighan & Dennis M. Ritchie, *The C Programming Language*. Any extensions, such as POSIX, must be tested for, typically using Autoconf (see Section 1.2 [Configure and cleanup], page 5).
- A FORTRAN 77 compiler or `f2c`, the FORTRAN-to-C converter.
- A simple `make`, considering the features of `make` in 4.2 BSD systems as a baseline.

  GNU or other extensions, including pattern rules using '`%`', the automatic variable '`$^`', the '`+=`' syntax to append to the value of a variable, the ("safe") inclusion of makefiles with no error, conditional execution, and many more, must not be used (see Chapter "Features" in the *GNU Make Manual* for more information). On the other hand, building R in a separate directory (not containing the sources) should work provided that `make` supports the `VPATH` mechanism.

  Windows-specific makefiles can assume GNU `make` 3.75 or later, as no other `make` is viable on that platform.
- A Bourne shell and the "traditional" Unix programming tools, including `grep`, `sed`, and `awk`.

  There are POSIX standards for these tools, but these may not fully be supported, and the precise standards are typically hard to access. Baseline features could be determined from a book such as *The UNIX Programming Environment* by Brian W. Kernighan & Rob Pike. Note in particular that '`|`' in a regexp is an extended regexp, and is not supported by all versions of `grep` or `sed`.

  Under Windows, these tools can be assumed because versions (specifically, of `cat`, `cp`, `cut`, `diff`, `echo`, `egrep`, `expr`, `find`,`gawk`, `grep`, `ls`, `mkdir`, `mv`, `rm`, `sed` and `sort`) are provided at CRAN. However, redirection cannot be assumed to be available via `system` as this does not use a standard shell (let alone a Bourne shell).

In addition, the following tools are needed for certain tasks.

- Perl version 5 is needed for converting documentation written in Rd format to plain text, HTML, LaTeX, and to extract the examples. In addition, several other tools, in particular `check` and `build` (see Section 1.3 [Checking and building packages], page 5), require Perl.

  The R Core Team has decided that Perl (version 5) can safely be assumed for building R from source, building and checking add-on packages, and for installing add-on packages from source. On the other hand, Perl cannot be assumed at all for installing *binary* (pre-built) versions of add-on packages, or at run time.

- Makeinfo version 4 is needed to build the Info files for the R manuals written in the GNU Texinfo system. (Future distributions of R will contain the Info files.)

It is also important that code is written in a way that allows others to understand it. This is particularly helpful for fixing problems, and includes using self-descriptive variable names, commenting the code, and also formatting it properly. The R Core Team recommends to use a basic indentation of 4 for R and C (and most likely also Perl) code, and 2 for documentation in Rd format. Emacs users can implement this indentation style by putting the following in one of their startup files.

```
;;; C
(add-hook 'c-mode-hook
          (lambda () (c-set-style "bsd")))
;;; ESS
(add-hook 'ess-mode-hook
          (lambda ()
            (ess-set-style 'C++)
            ;; Because
            ;;                                DEF GNU BSD K&R C++
            ;; ess-indent-level                2   2   8   5   4
            ;; ess-continued-statement-offset  2   2   8   5   4
            ;; ess-brace-offset                0   0  -8  -5  -4
            ;; ess-arg-function-offset         2   4   0   0   0
            ;; ess-expression-offset           4   2   8   5   4
            ;; ess-else-offset                 0   0   0   0   0
            ;; ess-close-brace-offset          0   0   0   0   0
            (add-hook 'local-write-file-hooks
                      (lambda ()
                        (nuke-trailing-whitespace)))))
;;; Perl
(add-hook 'perl-mode-hook
          (lambda () (setq perl-indent-level 4)))
```

(The 'GNU' styles for Emacs' C and R modes use a basic indentation of 2, which has been determined not to display the structure clearly enough when using narrow fonts.)

Function and variable index

P

R

S

U

V

Concept index