Writing R Extensions

Version 2.6.2 (2008-02-08)

R Development Core Team

ISBN 3-900051-11-9

Table of Contents

Acknowledgements **1**

1 Creating R packages **2**

- 1.1 Package structure 2
  - 1.1.1 The 'DESCRIPTION' file 3
  - 1.1.2 The 'INDEX' file 5
  - 1.1.3 Package subdirectories 6
  - 1.1.4 Package bundles 8
- 1.2 Configure and cleanup 9
  - 1.2.1 Using 'Makevars' 11
  - 1.2.2 Configure example 12
  - 1.2.3 Using F95 code 14
- 1.3 Checking and building packages 14
  - 1.3.1 Checking packages 14
  - 1.3.2 Building packages 16
  - 1.3.3 Customizing checking and building 17
- 1.4 Writing package vignettes 18
- 1.5 Submitting a package to CRAN 19
- 1.6 Package name spaces 19
  - 1.6.1 Specifying imports and exports 20
  - 1.6.2 Registering S3 methods 21
  - 1.6.3 Load hooks 21
  - 1.6.4 An example 23
  - 1.6.5 Summary – converting an existing package 24
  - 1.6.6 Name spaces with formal classes and methods 24
- 1.7 Writing portable packages 25
  - 1.7.1 Encoding issues 26
- 1.8 Diagnostic messages 27
- 1.9 Internationalization 28
  - 1.9.1 C-level messages 28
  - 1.9.2 R messages 28
- 1.10 Package types 29
  - 1.10.1 Frontend 29
  - 1.10.2 Translation 29
- 1.11 Services 30

2 Writing R documentation files **31**

- 2.1 Rd format 31
  - 2.1.1 Documenting functions 31
  - 2.1.2 Documenting data sets 35
  - 2.1.3 Documenting S4 classes and methods 36
  - 2.1.4 Documenting packages 36
- 2.2 Sectioning 37
- 2.3 Marking text 37
- 2.4 Lists and tables 38
- 2.5 Cross-references 39
- 2.6 Mathematics 39

2.7 Insertions 40
2.8 Indices 40
2.9 Platform-specific documentation 41
2.10 Encoding 41
2.11 Processing Rd format 41

3 Tidying and profiling R code 43

3.1 Tidying R code 43
3.2 Profiling R code for speed 43
3.3 Profiling R code for memory use 45
3.3.1 Memory statistics from `Rprof` 45
3.3.2 Tracking memory allocations 46
3.3.3 Tracing copies of an object 46
3.4 Profiling compiled code 46
3.4.1 Linux 47
3.4.1.1 sprof 47
3.4.1.2 oprofile 47
3.4.2 Solaris 48
3.4.3 MacOS X 48

4 Debugging 49

4.1 Browsing 49
4.2 Debugging R code 50
4.3 Using gctorture and valgrind 54
4.3.1 Using gctorture 54
4.3.2 Using valgrind 54
4.4 Debugging compiled code 55
4.4.1 Finding entry points in dynamically loaded code 56
4.4.2 Inspecting R objects when debugging 57

5 System and foreign language interfaces 59

5.1 Operating system access 59
5.2 Interface functions `.C` and `.Fortran` 59
5.3 `dyn.load` and `dyn.unload` 61
5.4 Registering native routines 62
5.5 Creating shared objects 64
5.6 Interfacing C++ code 65
5.7 Fortran I/O 67
5.8 Handling R objects in C 67
5.8.1 Handling the effects of garbage collection 69
5.8.2 Allocating storage 70
5.8.3 Details of R types 70
5.8.4 Attributes 71
5.8.5 Classes 73
5.8.6 Handling lists 74
5.8.7 Handling character data 74
5.8.8 Finding and setting variables 75
5.8.9 Some convenience functions 75
5.8.10 Named objects and copying 76
5.9 Interface functions `.Call` and `.External` 77
5.9.1 Calling `.Call` 77
5.9.2 Calling `.External` 78
5.9.3 Missing and special values 80

5.10 Evaluating R expressions from C ... 80
5.10.1 Zero-finding ... 82
5.10.2 Calculating numerical derivatives ... 83
5.11 Parsing R code from C ... 86
5.12 External pointers and weak references ... 87
5.13 Vector accessor functions ... 88
5.14 Character encoding issues ... 88

6 The R API: entry points for C code ... 89

6.1 Memory allocation ... 89
6.1.1 Transient storage allocation ... 89
6.1.2 User-controlled memory ... 90
6.2 Error handling ... 90
6.2.1 Error handling from FORTRAN ... 91
6.3 Random number generation ... 91
6.4 Missing and IEEE special values ... 91
6.5 Printing ... 92
6.5.1 Printing from FORTRAN ... 92
6.6 Calling C from FORTRAN and vice versa ... 92
6.7 Numerical analysis subroutines ... 93
6.7.1 Distribution functions ... 93
6.7.2 Mathematical functions ... 94
6.7.3 Numerical Utilities ... 95
6.7.4 Mathematical constants ... 96
6.8 Optimization ... 97
6.9 Integration ... 98
6.10 Utility functions ... 99
6.11 Re-encoding ... 100
6.12 Allowing interrupts ... 101
6.13 Platform and version information ... 101
6.14 Inlining C functions ... 101
6.15 Using these functions in your own C code ... 102
6.16 Organization of header files ... 102

7 Generic functions and methods ... 104

7.1 Adding new generics ... 105

8 Linking GUIs and other front-ends to R ... 106

8.1 Embedding R under Unix-alikes ... 106
8.1.1 Compiling against the R shared library ... 107
8.1.2 Setting R callbacks ... 108
8.1.3 Registering symbols ... 110
8.1.4 Meshing event loops ... 111
8.1.5 Threading issues ... 111
8.2 Embedding R under Windows ... 112
8.2.1 Using (D)COM ... 112
8.2.2 Calling R.dll directly ... 112

Function and variable index ... 116

Concept index ... 119

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.

In the following, we assume that you know the ‘`library()`’ command, including its ‘`lib.loc`’ argument, and we also assume basic knowledge of the `INSTALL` utility. Otherwise, please look at R’s help pages

```
?library
?INSTALL
```

before reading on.

A computing environment including a number of tools is assumed; the “R Installation and Administration” manual describes what is needed. Under a Unix-alike most of the tools are likely to be present by default, but Microsoft Windows and MacOS X will require careful setup.

Once a source package is created, it must be installed by the command `R CMD INSTALL`. See section “Add-on-packages” in *R Installation and Administration*, for further details.

Other types of extensions are supported: See Section 1.10 [Package types], page 29.

1.1 Package structure

A package consists of a subdirectory containing a file ‘`DESCRIPTION`’ and the subdirectories ‘`R`’, ‘`data`’, ‘`demo`’, ‘`exec`’, ‘`inst`’, ‘`man`’, ‘`po`’, ‘`src`’, and ‘`tests`’ (some of which can be missing). The package subdirectory may also contain files ‘`INDEX`’, ‘`NAMESPACE`’, ‘`configure`’, ‘`cleanup`’, ‘`LICENSE`’, ‘`LICENCE`’, and ‘`COPYING`’. Other files such as ‘`README`’, ‘`NEWS`’ or ‘`ChangeLog`’ will be ignored by R, but may be useful to end-users.

The ‘`DESCRIPTION`’ and ‘`INDEX`’ files are described in the sections below. The ‘`NAMESPACE`’ file is described in Section 1.6 [Package name spaces], page 19.

The optional files ‘`configure`’ and ‘`cleanup`’ are (Bourne shell) script files which are executed before and (provided that option ‘`--clean`’ was given) after installation on Unix-alikes, see Section 1.2 [Configure and cleanup], page 9.

The optional file ‘`LICENSE`’/‘`LICENCE`’ or ‘`COPYING`’ (where the former names are preferred) contains a copy of the license to the package, e.g. a copy of the GNU public license. Whereas you should feel free to include a license file in your source distribution, please do not arrange to install yet another copy of the GNU ‘`COPYING`’ or ‘`COPYING.LIB`’ files but refer to the copies on http://www.r-project.org/Licenses/ and included in the R distribution (in directory ‘`share/licenses`’).

For the conventions for files ‘`NEWS`’ and ‘`ChangeLog`’ in the GNU project see http://www.gnu.org/prep/standards/standards.html#Documentation.

The package subdirectory should be given the same name as the package. Because some file systems (e.g., those on Windows) are not case-sensitive, to maintain portability it is strongly recommended that case distinctions not be used to distinguish different packages. For example, if you have a package named ‘`foo`’, do not also create a package named ‘`Foo`’.

To ensure that file names are valid across file systems and supported operating system platforms, the ASCII control characters as well as the characters ‘`"`’, ‘`*`’, ‘`:`’, ‘`/`’, ‘`<`’, ‘`>`’, ‘`?`’, ‘`\`’, and ‘`|`’ are not allowed in file names. In addition, files with names ‘`con`’, ‘`prn`’, ‘`aux`’, ‘`clock$`’, ‘`nul`’, ‘`com1`’ to ‘`com9`’, and ‘`lpt1`’ to ‘`lpt9`’ after conversion to lower case and stripping possible “extensions” (e.g., ‘`lpt5.foo.bar`’), are disallowed. Also, file names in the same directory must not differ only by case (see the previous paragraph). In addition, the names of ‘`.Rd`’ files will be used in URLs and so must be ASCII and not contain `%`. For maximal portability filenames should only contain only ASCII characters not excluded already (that is `A-Za-z0-9._!#$%&+,;=@^(){}'[]`

we exclude space as many utilities do not accept spaces in file paths): non-English alphabetic characters cannot be guaranteed to be supported in all locales. It would be good practice to avoid the shell metacharacters `(){}'[]$`.

The R function `package.skeleton` can help to create the structure for a new package: see its help page for details.

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: 2004-01-01
Title: My First Collection of Functions
Author: Joe Developer <Joe.Developer@some.domain.net>, with
  contributions from A. User <A.User@whereever.net>.
Maintainer: Joe Developer <Joe.Developer@some.domain.net>
Depends: R (>= 1.8.0), nlme
Suggests: MASS
Description: A short (one paragraph) description of what
  the package does and why it may be useful.
License: GPL (>= 2)
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', 'License', 'Description', 'Title', 'Author', and 'Maintainer' fields are mandatory, the remaining fields ('Date', 'Depends', 'URL', ...) are optional.

The 'DESCRIPTION' file should be written entirely in ASCII for maximal portability.

The 'Package' and 'Version' fields give the name and the version of the package, respectively. The name should consist of letters, numbers, and the dot character and start with a letter. The version is a sequence of at least *two* (and usually three) non-negative integers separated by single '.' or '-' characters. The canonical form is as shown in the example, and a version such as '0.01' or '0.01.0' will be handled as if it were '0.1-0'. (Translation packages are allowed names of the form 'Translation-*ll*'.)

The 'License' field should specify the license of the package in the following standardized form. Alternatives are indicated via vertical bars. Individual specifications must be one of

- One of the "standard" short specifications

  `GPL-2 GPL-3 LGPL-2 LGPL-2.1 LGPL-3 AGPL-3 Artistic-1.0 Artistic-2.0`

  as made available via http://www.r-project.org/Licenses/ and contained in subdirectory 'share/licenses' of the R source or home directory.
- The names of abbreviations of free or open software licenses as contained the the license data base in file 'share/licenses/license.db' in the R source or home directory, possibly (for versioned licenses) followed by a version restriction of the form '(*op v*)' with *op* one of the comparison operators '<', '<=', '>', '>=', '==', or '!=' and *v* a numeric version specification (strings of non-negative integers separated by '.'), possibly combined via ',' (see below for an example). For versioned licenses, one can also specify the name followed by the version, or combine an existing abbreviation and the version with a '-'. Further free (see http://www.fsf.org/licenses/license-list.html) or open software (see http://www.opensource.org/licenses/bsd-license.php) licenses will be added to this data base if necessary.
- One of the strings 'file LICENSE' or 'file LICENCE' referring to a file named 'LICENSE' or 'LICENCE' in the package (source and installation) top-level directory.

- The string 'Unlimited', meaning that there no are restrictions on distribution or use other than those imposed by relevant laws.

Examples for standardized specifications include

```
License: GPL-2
License: GPL (>= 2) | BSD
License: LGPL (>= 2.0, < 3) | Mozilla Public License
License: GPL-2 | file LICENCE
```

Please note in particular that "Public domain" is not a valid license. 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 'Description' field should give a comprehensive description of what the package does. One can use several (complete) sentences, but only one paragraph.

The 'Title' field should give a short description of the package. Some package listings may truncate the title to 65 characters in order to keep the overall size of the listing limited. It should be capitalized, not use any markup, not have any continuation lines, and not end in a period. Older versions of R used a separate file 'TITLE' for giving this information; this is now defunct, and the 'Title' field in 'DESCRIPTION' is required.

The 'Author' field describes who wrote the package. It is a plain text field intended for human readers, but not for automatic processing (such as extracting the email addresses of all listed contributors).

The 'Maintainer' field should give a *single* name with a *valid* email address in angle brackets (for sending bug reports etc.). It should not end in a period or comma.

The optional 'Date' field gives the release date of the current version of the package. It is strongly recommended to use the yyyy-mm-dd format conforming to the ISO standard.

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 a comparison operator (currently only '>=' and '<=' are supported), whitespace and a valid version number in parentheses. (List package names even if they are part of a bundle.) 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 2.4.0 or newer, include 'R (>= 2.4.0)' in the 'Depends' field. Both library and the R package checking facilities use this field, hence it is an error to use improper syntax or misuse the 'Depends' field for comments on other software that might be needed. Other dependencies (external to the R system) should be listed in the 'SystemRequirements' field or a separate 'README' file. The R INSTALL facilities check if the version of R used is recent enough for the package being installed, and the list of packages which is specified will be attached (after checking version dependencies) before the current package, both when library is called and when saving an image of the package's code or preparing for lazy-loading.

The optional 'Imports' field lists packages whose name spaces are imported from but which do not need to be attached. Name spaces accessed by the '::' and ':::' operators must be listed here, or in 'Suggests' or 'Enhances' (see below). Ideally this field will include all the standard packages, and it is important to include S4-using packages (as their class definitions can change and the 'DESCRIPTION' file is used to decide which packages to re-install when this happens).

The optional 'Suggests' field uses the same syntax as 'Depends' and lists packages that are not necessarily needed. This includes packages used only in examples or vignettes (see Section 1.4 [Writing package vignettes], page 18), and packages loaded in the body of functions. E.g., suppose an example from package **foo** uses a dataset from package **bar**. Then it is not necessary to have **bar** for routine use of **foo**, unless one wants to execute the examples: it is nice to have **bar**, but not necessary.

Finally, the optional 'Enhances' field lists packages "enhanced" by the package at hand, e.g., by providing methods for classes from these packages.

The general rules are

- Packages whose name space only is needed to load the package using library(*pkgname*) must be listed in the 'Imports' field.
- Packages that need to be attached to successfully load the package using library(*pkgname*) must be listed in the 'Depends' field.
- All packages that are needed to successfully run R CMD check on the package must be listed in one of 'Depends' or 'Suggests' or 'Imports'.

In particular, large packages providing "only" data for examples or vignettes should be listed in 'Suggests' rather than 'Depends' in order to make lean installations possible.

The optional 'URL' field may give a list of URLs 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.

Base and recommended packages (i.e., packages contained in the R source distribution or available from CRAN and recommended to be included in every binary distribution of R) have a 'Priority' field with value 'base' or 'recommended', respectively. These priorities must not be used by "other" packages.

An optional 'Collate' field (or OS-specific variants 'Collate.*OStype*', such as e.g. 'Collate.windows') can be used for controlling the collation order for the R code files in a package when these are concatenated into a single file upon installation from source. The default is to try collating according to the 'C' locale. If present, the collate specification must list *all* R code files in the package (taking possible OS-specific subdirectories into account, see Section 1.1.3 [Package subdirectories], page 6) as a whitespace separated list of file paths relative to the 'R' subdirectory. Paths containing white space or quotes need to be quoted. An applicable OS-specific collation field ('Collate.unix' or 'Collate.windows') will be used instead of 'Collate'.

The optional 'LazyLoad' and 'LazyData' fields control whether the R objects and the datasets (respectively) use lazy-loading: set the field's value to 'yes' or 'true' for lazy-loading and 'no' or 'false' for no lazy-loading. (Capitalized values are also accepted.)

If the package you are writing uses the **methods** package, specify 'LazyLoad: yes'.

The optional 'ZipData' field controls whether the automatic Windows build will zip up the data directory or no: set this to 'no' if your package will not work with a zipped data directory.

If the 'DESCRIPTION' file is not entirely in ASCII it should contain an 'Encoding' field specifying an encoding. This is currently used as the encoding of the 'DESCRIPTION' file itself and of the 'R' and 'NAMESPACE' files, and is taken as the default encoding of '.Rd' files as from R 2.6.0. Only encoding names latin1, latin2 and UTF-8 are known to be portable. (Do not specify an encoding unless one is actually needed: doing so makes the package *less* portable.)

The optional 'Type' field specifies the type of the package: see Section 1.10 [Package types], page 29.

> **Note:** There should be no 'Built' or 'Packaged' fields, as these are added by the package management tools.

1.1.2 The 'INDEX' file

The optional 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). Normally this file is missing, and the corresponding information is automatically generated from the documentation sources (using Rdindex() from package **tools**) when installing from source and when using the package builder (see Section 1.3 [Checking and building packages], page 14).

Rather than editing this file, it is preferable to put customized information about the package into an overview man page (see Section 2.1.4 [Documenting packages], page 36) and/or a vignette (see Section 1.4 [Writing package vignettes], page 18).

1.1.3 Package subdirectories

The 'R' subdirectory contains R code files, only. The code files to be installed must start with an ASCII (lower or upper case) letter or digit 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. Ideally, the R code files should only directly assign R objects and definitely should not call functions with side effects such as `require` and `options`. If computations are required to create objects these can use code 'earlier' in the package (see the 'Collate' field) plus, *only if lazyloading is used*, functions in the 'Depends' packages provided that the objects created do not depend on those packages except *via* name space imports. (Packages without namespaces will work under somewhat less restrictive assumptions.)

Two exceptions are allowed: if the 'R' subdirectory contains a file 'sysdata.rda' (a saved image of R objects) this will be lazy-loaded into the name space/package environment – this is intended for system datasets that are not intended to be user-accessible via `data`. Also, files ending in '.in' will be allowed in the 'R' directory to allow a 'configure' script to generate suitable files.

Only ASCII characters (and the control characters tab, formfeed, LF and CR) should be used in code files. Other characters are accepted in comments, but then the comments may not be readable in e.g. a UTF-8 locale. Non-ASCII characters in object names will normally[1] fail when the package is installed. Any byte will be allowed[2] in a quoted character string (but `\uxxxx` escapes should not be used), but non-ASCII character strings may not be usable in some locales and may display incorrectly in others.

Various R functions in a package can be used to initialize and clean up. For packages without a name space, these are `.First.lib` and `.Last.lib`. (See Section 1.6.3 [Load hooks], page 21, for packages with a name space.) It is conventional to define these functions in a file called 'zzz.R'. If `.First.lib` is defined in a package, it is called with arguments `libname` and `pkgname` after the package is loaded and attached. (If a package is installed with version information, the package name includes the version information, e.g. 'ash_1.0.9'.) A common use is to call `library.dynam` inside `.First.lib` to load compiled code: another use is to call those functions with side effects. If `.Last.lib` exists in a package it is called (with argument the full path to the installed package) just before the package is detached. It is uncommon to detach packages and rare to have a `.Last.lib` function: one use is to call `library.dynam.unload` to unload compiled code.

The 'man' subdirectory should contain (only) documentation files for the objects in the package in *R documentation* (Rd) format. The documentation filenames must start with an ASCII (lower or upper case) letter or digit and have the extension '.Rd' (the default) or '.rd'. Further, the names must be valid in 'file://' URLs, which means[3] they must be entirely ASCII and not contain '%'. See Chapter 2 [Writing R documentation files], page 31, for more information.

[1] This is true for OSes which implement the 'C' locale, unless neither lazy-loading nor saving an image are used, in which case it would fail if loaded in a 'C' locale. (Windows' idea of the 'C' locale uses the WinAnsi charset.)

[2] It is good practice to encode them as octal or hex escape sequences.

[3] More precisely, they can contain the English alphanumeric characters and the symbols '$ - _ . + ! ' ( ) , ; = &'.

Note that all user-level objects in a package should be documented; if a package *pkg* contains user-level objects which are for "internal" use only, it should provide a file '*pkg*-internal.Rd' which documents all such objects, and clearly states that these are not meant to be called by the user. See e.g. the sources for package **grid** in the R distribution for an example. Note that packages which use internal objects extensively should hide those objects in a name space, when they do not need to be documented (see Section 1.6 [Package name spaces], page 19).

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

The sources and headers for the compiled code are in 'src', plus optionally file 'Makevars' or 'Makefile'. When a package is installed using `R CMD INSTALL`, Make is used to control compilation and linking into a shared object for loading into R. There are default variables and rules for this (determined when R is configured and recorded in '*R_HOME*/etc*R_ARCH*/Makeconf'), providing support for C, C++, FORTRAN 77, Fortran 9x[4], Objective C and Objective C++ with associated extensions '.c', '.cc' or '.cpp' or '.C', '.f', '.f90' or '.f95', '.m', and '.mm' or '.M', respectively. We recommend using '.h' for headers, also for C++[5] or Fortran 9x include files. The default rules can be tweaked by setting macros in a file 'src/Makevars' (see Section 1.2.1 [Using Makevars], page 11). Note that this mechanism should be general enough to eliminate the need for a package-specific 'src/Makefile'. If such a file is to be distributed, considerable care is needed to make it general enough to work on all R platforms. It should have an appropriate first target (conventionally called 'all') and a (possibly empty) target 'clean' which removes all files generated by Make (to be used by 'R CMD INSTALL --clean' and 'R CMD INSTALL --preclean'). There are platform-specific file names on Windows: 'src/Makevars.win' takes precedence over 'src/Makevars' and 'src/Makefile.win' must be used.

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', see `?data` for the file formats), or `save()` images ('.RData' or '.rda'). (All ports of R use the same binary (XDR) format and can read compressed images. Use images saved with `save(, compress = TRUE)`, the default, to save space.) 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. It is no longer necessary to provide a '00Index' file in the 'data' directory—the corresponding information is generated automatically from the documentation sources when installing from source, or when using the package builder (see Section 1.3 [Checking and building packages], page 14). If your data files are enormous you can speed up installation by providing a file 'datalist' in the 'data' subdirectory. This should have one line per topic that `data()` will find, in the format 'foo' if `data(foo)` provides 'foo', or 'foo: bar bah' if `data(foo)` provides 'bar' and 'bah'.

The 'demo' subdirectory is for R scripts (for running via `demo()`) that demonstrate some of the functionality of the package. Demos may be interactive and are not checked automatically, so if testing is desired use code in the 'tests' directory. The script files must start with a (lower or upper case) letter and have one of the extensions '.R' or '.r'. If present, the 'demo' subdirectory should also have a '00Index' file with one line for each demo, giving its name and a description separated by white space. (Note that it is not possible to generate this index file automatically.)

The contents of the 'inst' subdirectory will be copied recursively to the installation directory. Subdirectories of 'inst' should not interfere with those used by R (currently, 'R', 'data', 'demo',

[4] Note that Ratfor is not supported. If you have Ratfor source code, you need to convert it to FORTRAN. Only FORTRAN-77 (which we write in upper case) is supported on all platforms, but most also support Fortran-95 (for which we use title case). If you want to ship Ratfor source files, please do so in a subdirectory of 'src' and not in the main subdirectory.

[5] Using '.hpp', although somewhat popular, is not guaranteed to be portable.

'exec', 'libs', 'man', 'help', 'html', 'latex', 'R-ex', 'chtml', and 'Meta'). The copying of the 'inst' happens after 'src' is built so its 'Makefile' can create files to be installed. Note that with the exceptions of 'INDEX', 'LICENSE'/'LICENCE' and 'COPYING', information files at the top level of the package will *not* be installed and so not be known to users of Windows and MacOS X compiled packages (and not seen by those who use R CMD INSTALL or install.packages on the tarball). So any information files you wish an end user to see should be included in 'inst'. One thing you might like to add to 'inst' is a 'CITATION' file for use by the citation function.

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. The directory 'tests' is copied to the check area, and the tests are run with the copy as the working directory and with R_LIBS set to ensure that the copy of the package installed during testing will be found by library(*pkg_name*).

Subdirectory 'exec' could contain additional executables the package needs, typically scripts for interpreters such as the shell, Perl, or Tcl. This mechanism is currently used only by a very few packages, and still experimental.

Subdirectory 'po' is used for files related to *localization*: see Section 1.9 [Internationalization], page 28.

1.1.4 Package bundles

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

The 'DESCRIPTION' file of a bundle has a 'Bundle' field and no 'Package' field, as in

```
Bundle: VR
Priority: recommended
Contains: MASS class nnet spatial
Version: 7.2-36
Date: 2007-08-29
Depends: R (>= 2.4.0), grDevices, graphics, stats, utils
Suggests: lattice, nlme, survival
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.
Maintainer: Brian Ripley <ripley@stats.ox.ac.uk>
BundleDescription: Functions and datasets to support Venables and
  Ripley, 'Modern Applied Statistics with S' (4th edition).
License: GPL-2 | GPL-3
URL: http://www.stats.ox.ac.uk/pub/MASS4/
```

The 'Contains' field lists the packages (space separated), which should be contained in separate subdirectories with the names given. During building and installation, packages will be installed in the order specified. Be sure to order this list so that dependencies are met appropriately.

The packages contained in a bundle 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.
Title: Functions for Kriging and Point Pattern Analysis
```

Any files in the package bundle except the 'DESCRIPTION' file and the named packages will be ignored.

The 'Depends' field in the bundle's 'DESCRIPTION' file should list the dependencies of all the constituent packages (and similarly for 'Imports' and 'Suggests'), and then 'DESCRIPTION.in' files should not contain these fields.

1.2 Configure and cleanup

Note that most of this section is Unix-specific: see the comments later on about the Windows port of R.

If your package needs some system-dependent configuration before installation you can include a (Bourne shell) 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 (Bourne shell) script 'cleanup' is executed as last thing by `R CMD INSTALL` if present and option '--clean' was given, and by `R CMD build` when preparing the package for building from its source. It can be used to clean up the package source tree. In particular, it should remove all 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 create 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_CONFIG_FILES([foo.R])
AC_OUTPUT
```

in 'configure.ac' (assuming Autoconf 2.50 or later).

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 will very likely need to ensure that the same C compiler and compiler flags are used in the '`configure`' tests as when compiling R or your package. Under Unix, you can achieve this by including the following fragment early in '`configure.ac`'

```
: ${R_HOME=`R RHOME`}
if test -z "${R_HOME}"; then
  echo "could not determine R_HOME"
  exit 1
fi
CC=`"${R_HOME}/bin/R" CMD config CC`
CFLAGS=`"${R_HOME}/bin/R" CMD config CFLAGS`
CPPFLAGS=`"${R_HOME}/bin/R" CMD config CPPFLAGS`
```

(using '`${R_HOME}/bin/R`' rather than just '`R`' is necessary in order to use the 'right' version of R when running the script as part of `R CMD INSTALL`.)

Note that earlier versions of this document recommended obtaining the configure information by direct extraction (using grep and sed) from '*`R_HOME`*`/etc`*`R_ARCH`*`/Makeconf`', which only works for variables recorded there as literals. You can use `R CMD config` for getting the value of the basic configuration variables, or the header and library flags necessary for linking against R, see *`R CMD config --help`* for details. (This works on Windows as from R 2.6.0.)

To check for an external BLAS library using the `ACX_BLAS` macro from the official Autoconf Macro Archive, one can simply do

```
F77=`"${R_HOME}/bin/R" CMD config F77`
AC_PROG_F77
FLIBS=`"${R_HOME}/bin/R" CMD config FLIBS`
ACX_BLAS([], AC_MSG_ERROR([could not find your BLAS library], 1))
```

Note that `FLIBS` as determined by R must be used to ensure that FORTRAN 77 code works on all R platforms. Calls to the Autoconf macro `AC_F77_LIBRARY_LDFLAGS`, which would overwrite `FLIBS`, must not be used (and hence e.g. removed from `ACX_BLAS`). (Recent versions of Autoconf in fact allow an already set `FLIBS` to override the test for the FORTRAN linker flags. Also, recent versions of R can detect external BLAS and LAPACK libraries.)

You should 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, or provide a '`configure.win`' script to be used on that platform. (Optionally, there can be a '`cleanup.win`' script as well. Both should be shell scripts to be executed by `ash`, which is a minimal version of Bourne-style `sh`.)

In some rare circumstances, the configuration and cleanup scripts need to know the location into which the package is being installed. An example of this is a package that uses C code and creates two shared object/DLLs. Usually, the object that is dynamically loaded by R is linked against the second, dependent, object. On some systems, we can add the location of this dependent object to the object that is dynamically loaded by R. This means that each user does not have to set the value of the `LD_LIBRARY_PATH` (or equivalent) environment variable, but that the secondary object is automatically resolved. Another example is when a package installs support files that are required at run time, and their location is substituted into an R data structure at installation time. (This happens with the Java Archive files in the **SJava** package.) The names of the top-level library directory (i.e., specifiable via the '`-l`' argument) and the directory of the package itself are made available to the installation scripts via the two shell/environment variables `R_LIBRARY_DIR` and `R_PACKAGE_DIR`. Additionally, the name of the package (e.g., '`survival`' or '`MASS`') being installed is available from the shell variable `R_PACKAGE_NAME`.

1.2.1 Using 'Makevars'

Sometimes writing your own 'configure' script can be avoided by supplying a file 'Makevars': also one of the most common uses of a 'configure' script is to make 'Makevars' from 'Makevars.in'.

The most common use of a 'Makevars' file is to set additional preprocessor (for example include paths) flags via PKG_CPPFLAGS, and additional compiler flags by setting PKG_CFLAGS, PKG_CXXFLAGS and PKG_FFLAGS, for C, C++, or FORTRAN respectively (see Section 5.5 [Creating shared objects], page 64).

Also, 'Makevars' can be used to set flags for the linker, for example '-L' and '-l' options.

When writing a 'Makevars' file for a package you intend to distribute, take care to ensure that it is not specific to your compiler: flags such as '-O2 -Wall -pedantic' are all specific to GCC.

There are some macros which are built whilst configuring the building of R itself, are stored on Unix-alikes in '*R_HOME*/etc*R_ARCH*/Makeconf' and can be used in 'Makevars'. These include

FLIBS
: A macro containing the set of libraries need to link FORTRAN code. This may need to be included in PKG_LIBS.

BLAS_LIBS
: A macro containing the BLAS libraries used when building R. This may need to be included in PKG_LIBS. Beware that if it is empty then the R executable will contain all the double-precision and double-complex BLAS routines, but no single-precision or complex routines. If BLAS_LIBS is included, then FLIBS also needs to be[6], as most BLAS libraries are written in FORTRAN.

LAPACK_LIBS
: A macro containing the LAPACK libraries (and paths where appropriate) used when building R. This may need to be included in PKG_LIBS. This may point to a dynamic library libRlapack which contains all the double-precision LAPACK routines as well as those double-complex LAPACK and BLAS routines needed to build R, or it may point to an external LAPACK library, or may be empty if an external BLAS library also contains LAPACK.

  [There is no guarantee that the LAPACK library will provide more than all the double-precision and double-complex routines, and some do not provide all the auxiliary routines.]

  The macros BLAS_LIBS and FLIBS should always be included *after* LAPACK_LIBS.

SAFE_FFLAGS
: A macro containing flags which are needed to circumvent over-optimization of FORTRAN code: it is typically '-g -O2 -ffloat-store' on 'ix86' platforms using g77 or gfortran. Note that this is **not** an additional flag to be used as part of PKG_FFLAGS, but a replacement for FFLAGS, and that it is intended for the FORTRAN-77 compiler 'F77' and not necessarily for the Fortran 90/95 compiler 'FC'. See the example later in this section.

Setting certain macros in 'Makevars' will prevent R CMD SHLIB setting them: in particular if 'Makevars' sets 'OBJECTS' it will not be set on the make command line. This can be useful in conjunction with implicit rules to allow other types of source code to be compiled and included in the shared object.

Note that 'Makevars' should not normally contain targets, as it is (except on Windows) included before the default makefile and make is called without an explicit target. To circumvent that, use a suitable phony target before any actual targets: for example **fastICA** has

[6] on Unix-alikes: Windows resolves such dependencies at link time.

```
SLAMC_FFLAGS=$(R_XTRA_FFLAGS) $(FPICFLAGS) $(SHLIB_FFLAGS) $(SAFE_FFLAGS)

all: $(SHLIB)

slamc.o: slamc.f
        $(F77) $(SLAMC_FFLAGS) -c -o slamc.o slamc.f
```

to ensure that the LAPACK routines find some constants without infinite looping. The Windows equivalent is

```
slamc.o: slamc.f
        $(F77) $(SAFE_FFLAGS) -c -o slamc.o slamc.f
```

More generally, on a Unix-alike one could have something like

```
.PHONY: all

all: before $(SHLIB) after

before:
        Things that need to be done first like creating libraries

after:
        Cleanup needed after 'before'
```

On Windows, one can add dependencies to the '`all`' target (which is what will get called), e.g. (based on package **rcom**)

```
all: ../inst/tst/bin/rcom_test.exe extraclean

../inst/tst/bin/rcom_test.exe: rcom_test.exe
$(MKDIR) -p ../inst/tst/bin
$(CP) $? $ rcom_test.exe: rcom_test.o
rcom_test-LIBS = -L. -lsupc++ -luuid -lole32 -loleaut32

extraclean:
        $(RM) rcom_test.exe
```

The added dependencies will be built after the DLL: it is also possible (but not advisable) to have a target '`all`' with commands (rather than dependencies)

There are two another targets, '`before`' and '`after`', which by default have neither dependencies nor commands so can be overridden in a '`Makevars.win`'.

1.2.2 Configure example

It may be helpful to give an extended example of using a '`configure`' script to create a '`src/Makevars`' file: this is based on that in the **RODBC** package.

The '`configure.ac`' file follows: '`configure`' is created from this by running `autoconf` in the top-level package directory (containing '`configure.ac`').

```
AC_INIT([RODBC], 1.1.8) dnl package name, version

dnl A user-specifiable option
odbc_mgr=""
AC_ARG_WITH([odbc-manager],
            AC_HELP_STRING([--with-odbc-manager=MGR],
                           [specify the ODBC manager, e.g. odbc or iodbc]),
            [odbc_mgr=$withval])

if test "$odbc_mgr" = "odbc" ; then
  AC_PATH_PROGS(ODBC_CONFIG, odbc_config)
fi

dnl Select an optional include path, from a configure option
dnl or from an environment variable.
AC_ARG_WITH([odbc-include],
            AC_HELP_STRING([--with-odbc-include=INCLUDE_PATH],
```

```
                              [the location of ODBC header files]),
            [odbc_include_path=$withval])
RODBC_CPPFLAGS="-I."
if test [ -n "$odbc_include_path" ] ; then
   RODBC_CPPFLAGS="-I. -I${odbc_include_path}"
else
  if test [ -n "${ODBC_INCLUDE}" ] ; then
     RODBC_CPPFLAGS="-I. -I${ODBC_INCLUDE}"
  fi
fi

dnl ditto for a library path
AC_ARG_WITH([odbc-lib],
            AC_HELP_STRING([--with-odbc-lib=LIB_PATH],
                           [the location of ODBC libraries]),
            [odbc_lib_path=$withval])
if test [ -n "$odbc_lib_path" ] ; then
   LIBS="-L$odbc_lib_path ${LIBS}"
else
  if test [ -n "${ODBC_LIBS}" ] ; then
     LIBS="-L${ODBC_LIBS} ${LIBS}"
  else
    if test -n "${ODBC_CONFIG}"; then
      odbc_lib_path=`odbc_config --libs | sed s/-lodbc//`
      LIBS="${odbc_lib_path} ${LIBS}"
    fi
  fi
fi

dnl Now find the compiler and compiler flags to use
: ${R_HOME=`R RHOME`}
if test -z "${R_HOME}"; then
  echo "could not determine R_HOME"
  exit 1
fi
CC=`"${R_HOME}/bin/R" CMD config CC`
CPP=`"${R_HOME}/bin/R" CMD config CPP`
CFLAGS=`"${R_HOME}/bin/R" CMD config CFLAGS`
CPPFLAGS=`"${R_HOME}/bin/R" CMD config CPPFLAGS`
AC_PROG_CC
AC_PROG_CPP


if test -n "${ODBC_CONFIG}"; then
  RODBC_CPPFLAGS=`odbc_config --cflags`
fi
CPPFLAGS="${CPPFLAGS} ${RODBC_CPPFLAGS}"

dnl Check the headers can be found
AC_CHECK_HEADERS(sql.h sqlext.h)
if test "${ac_cv_header_sql_h}" = no ||
   test "${ac_cv_header_sqlext_h}" = no; then
   AC_MSG_ERROR("ODBC headers sql.h and sqlext.h not found")
fi

dnl search for a library containing an ODBC function
if test [ -n "${odbc_mgr}" ] ; then
  AC_SEARCH_LIBS(SQLTables, ${odbc_mgr}, ,
      AC_MSG_ERROR("ODBC driver manager ${odbc_mgr} not found"))
else
  AC_SEARCH_LIBS(SQLTables, odbc odbc32 iodbc, ,
      AC_MSG_ERROR("no ODBC driver manager found"))
fi

dnl for 64-bit ODBC need SQL[U]LEN, and it is unclear where they are defined.
```

```
AC_CHECK_TYPES([SQLLEN, SQLULEN], , , [# include <sql.h>])
dnl for unixODBC header
AC_CHECK_SIZEOF(long, 4)

dnl substitute RODBC_CPPFLAGS and LIBS
AC_SUBST(RODBC_CPPFLAGS)
AC_SUBST(LIBS)
AC_CONFIG_HEADERS([src/config.h])
dnl and do substitution in the src/Makevars.in and src/config.h
AC_CONFIG_FILES([src/Makevars])
AC_OUTPUT
```

where '`src/Makevars.in`' would be simply

```
PKG_CPPFLAGS = @RODBC_CPPFLAGS@
PKG_LIBS = @LIBS@
```

A user can then be advised to specify the location of the ODBC driver manager files by options like (lines broken for easier reading)

```
R CMD INSTALL
  --configure-args='--with-odbc-include=/opt/local/include
  --with-odbc-lib=/opt/local/lib --with-odbc-manager=iodbc'
  RODBC
```

or by setting the environment variables `ODBC_INCLUDE` and `ODBC_LIBS`.

1.2.3 Using F95 code

R currently does not distinguish between FORTRAN 77 and Fortran 90/95 code, and assumes all FORTRAN comes in source files with extension '`.f`'. Commercial Unix systems typically use a F95 compiler, but only since the release of `gcc 4.0.0` in April 2005 have Linux and other non-commercial OSes had much support for F95. Only wih R 2.6.0 did the Windows port adopt a Fortran 90 compiler.

This means that portable packages need to be written in correct FORTRAN 77, which will also be valid Fortran 95. See http://developer.r-project.org/Portability.html for reference resources. In particular, *free source form* F95 code is not portable.

On some systems an alternative F95 compiler is available: from the `gcc` family this might be `gfortran` or `g95`. Configuring R will try to find a compiler which (from its name) appears to be a Fortran 90/95 compiler, and set it in macro '`FC`'. Note that it does not check that such a compiler is fully (or even partially) compliant with Fortran 90/95. Packages making use of Fortran 90/95 features should use file extension '`.f90`' or '`.f95`' for the source files: the variable `PKG_FCFLAGS` specifies any special flags to be used. There is no guarantee that compiled Fortran 90/95 code can be mixed with any other type of code, nor that a build of R will have support for such packages.

1.3 Checking and building packages

Before using these tools, please check that your package can be installed and loaded. `R CMD check` will *inter alia* do this, but you will get more informative error messages doing the checks directly.

1.3.1 Checking packages

Using `R CMD check`, the R package checker, one can test whether *source* R packages work correctly. It can be run on one or more directories, or gzipped package `tar` archives[7] with extension '`.tar.gz`' or '`.tgz`'. This runs a series of checks, including

[7] This may require GNU `tar`: the command used can be set with environment variable `TAR`.

1. The package is installed. This will warn about missing cross-references and duplicate aliases in help files.
2. The file names are checked to be valid across file systems and supported operating system platforms.
3. The files and directories are checked for sufficient permissions (Unix only).
4. The 'DESCRIPTION' file is checked for completeness, and some of its entries for correctness. Unless installation tests are skipped, checking is aborted if the package dependencies cannot be resolved at run time. One check is that the package name is not that of a standard package, nor of the defunct standard packages ('ctest', 'eda', 'lqs', 'mle', 'modreg', 'mva', 'nls', 'stepfun' and 'ts') which are handled specially by `library`. Another check is that all packages mentioned in `library` or `requires` or from which the 'NAMESPACE' file imports or are called *via* `::` or `:::` are listed (in 'Depends', 'Imports', 'Suggests' or 'Contains'): this is not an exhaustive check of the actual imports.
5. Available index information (in particular, for demos and vignettes) is checked for completeness.
6. The package subdirectories are checked for suitable file names and for not being empty. The checks on file names are controlled by the option '--check-subdirs=*value*'. This defaults to 'default', which runs the checks only if checking a tarball: the default can be overridden by specifying the value as 'yes' or 'no'. Further, the check on the 'src' directory is only run if the package/bundle does not contain a 'configure' script (which corresponds to the value 'yes-maybe') and there is no 'src/Makefile' or 'src/Makefile.in'.

   To allow a 'configure' script to generate suitable files, files ending in '.in' will be allowed in the 'R' directory.
7. The R files are checked for syntax errors. Bytes which are non-ASCII are reported as warnings, but these should be regarded as errors unless it is known that the package will always be used in the same locale.
8. It is checked that the package can be loaded, first with the usual default packages and then only with package **base** already loaded. If the package has a namespace, it is checked if this can be loaded in an empty session with only the **base** namespace loaded. (Namespaces and packages can be loaded very early in the session, before the default packages are available, so packages should work then.)
9. The R files are checked for correct calls to `library.dynam` (with no extension). In addition, it is checked whether methods have all arguments of the corresponding generic, and whether the final argument of replacement functions is called 'value'. All foreign function calls (`.C`, `.Fortran`, `.Call` and `.External` calls) are tested to see if they have a `PACKAGE` argument, and if not, whether the appropriate DLL might be deduced from the name space of the package. Any other calls are reported. (The check is generous, and users may want to supplement this by examining the output of `tools::checkFF("mypkg", verbose=TRUE)`, especially if the intention were to always use a `PACKAGE` argument)
10. The Rd files are checked for correct syntax and meta data, including the presence of the mandatory (`\name`, `\alias`, `\title`, `\description` and `\keyword`) fields. The Rd name and title are checked for being non-empty, and the keywords found are compared to the standard ones. There is a check for missing cross-references (links).
11. A check is made for missing documentation entries, such as undocumented user-level objects in the package.
12. Documentation for functions, data sets, and S4 classes is checked for consistency with the corresponding code.
13. It is checked whether all function arguments given in `\usage` sections of Rd files are documented in the corresponding `\arguments` section.

14. C, C++ and FORTRAN source and header files are tested for portable (LF-only) line endings. If there is a '`Makefile`' or '`Makefile.in`' or '`Makevars`' or '`Makevars.in`' in the '`src`' directory, it is checked for portable line endings and the correct use of '`$(BLAS_LIBS)`'.
15. The examples provided by the package's documentation are run. (see Chapter 2 [Writing R documentation files], page 31, for information on using `\examples` to create executable example code.)

    Of course, released packages should be able to run at least their own examples. Each example is run in a 'clean' environment (so earlier examples cannot be assumed to have been run), and with the variables `T` and `F` redefined to generate an error unless they are set in the example: See section "Logical vectors" in *An Introduction to R*.
16. 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`'.)
17. The code in package vignettes (see Section 1.4 [Writing package vignettes], page 18) is executed.
18. If a working `latex` program is available, the '`.dvi`' version of the package's manual is created (to check that the Rd files can be converted successfully).

Use *`R CMD check --help`* to obtain more information about the usage of the R package checker. A subset of the checking steps can be selected by adding flags.

1.3.2 Building packages

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 few diagnostic checks and cleanups are performed. In particular, it is tested whether object indices exist and can be assumed to be up-to-date, and C, C++ and FORTRAN source files and relvant make files are tested and converted to LF line-endings if necessary.

Run-time checks whether the package works correctly should be performed using `R CMD check` prior to invoking the build procedure.

To exclude files from being put into the package, one can specify a list of exclude patterns in file '`.Rbuildignore`' in the top-level source directory. These patterns should be Perl regexps, one per line, to be matched against the file names relative to the top-level source directory. In addition, directories called '`CVS`' or '`.svn`' or '`.arch-ids`' and files '`GNUMakefile`' or with base names starting with '`.#`', or starting and ending with '`#`', or ending in '`~`', '`.bak`' or '`.swp`', are excluded by default. In addition, those files in the '`R`', '`demo`' and '`man`' directories which are flagged by `R CMD check` as having invalid names will be excluded.

Use *`R CMD build --help`* to obtain more information about the usage of the R package builder.

Unless *`R CMD build`* is invoked with the '`--no-vignettes`' option, it will attempt to rebuild the vignettes (see Section 1.4 [Writing package vignettes], page 18) in the package. To do so it installs the current package/bundle into a temporary library tree, but any dependent packages need to be installed in an available library tree (see the Note: below).

One of the checks that `R CMD build` runs is for empty source directories. These are in most cases unintentional, in which case they should be removed and the build re-run.

It can be useful to run `R CMD check --check-subdirs=yes` on the built tarball as a final check on the contents.

`R CMD build` can also build pre-compiled version of packages for binary distributions, but `R CMD INSTALL --build` is preferred (and is considerably more flexible). In particular, Windows users are recommended to use `R CMD INSTALL --build` and install into the main library tree (the default) so that HTML links are resolved.

> **Note:** `R CMD check` and `R CMD build` run R with '`--vanilla`', so none of the user's startup files are read. If you need `R_LIBS` set (to find packages in a non-standard library) you will need to set it in the environment.
>
> **Note to Windows users:** `R CMD check` and `R CMD build` work well under Windows NT4/2000/XP/2003 but may not work correctly on Windows 95/98/ME because of problems with some versions of Perl on those limited OSes. Experiences vary. To use them you will need to have installed the files for building source packages (which is the default).

1.3.3 Customizing checking and building

In addition to the available command line options, `R CMD check` also allows customization by setting (Perl) configuration variables in a configuration file, the location of which can be specified via the '`--rcfile`' option and defaults to '`$HOME/.R/check.conf`' provided that the environment variable `HOME` is set.

The following configuration variables are currently available.

`$R_check_use_install_log`
: If true, record the output from installing a package as part of its check to a log file ('`00install.out`' by default), even when running interactively. Default: true.

`$R_check_all_non_ISO_C`
: If true, do not ignore compiler (typically GCC) warnings about non ISO C code in *system* headers. Default: false.

`$R_check_weave_vignettes`
: If true, weave package vignettes in the process of checking them. Default: true.

`$R_check_latex_vignettes`
: If true (and `$R_check_weave_vignettes` is also true), LaTeX package vignettes in the process of checking them: this will show up `Sweave` source errors, including missing source files. Default: true.

`$R_check_subdirs_nocase`
: If true, check the case of directories such as '`R`' and '`man`'. Default: false.

`$R_check_subdirs_strict`
: Initial setting for '`--check-subdirs`'. Default: '`default`' (which checks only tarballs, and checks in the '`src`' only if there is no '`configure`' file).

`$R_check_force_suggests`
: If true, give an error if suggested packages are not available. Default: true.

`$R_check_use_codetools`
: If true, make use of the **codetools** package, which provides a detailed analysis of visibility of objects (but may give false positives). Default: true.

`$R_check_Rd_style`
: If true, check whether Rd usage entries for S3 methods use the full function name rather than the appropiate `\method` markup. Default: true.

`$R_check_Rd_xrefs`
: If true, check the cross-references in '`.Rd`' files. Default: true.

Values '`1`' or a string with lower-cased version '`"yes"`' or '`"true"`' can be used for setting the variables to true; similarly, '`0`' or strings with lower-cased version '`"no"`' or '`"false"`' give false.

For example, a configuration file containing

```
$R_check_use_install_log = "TRUE";
$R_check_weave_vignettes = 0;
```

results in using install logs and turning off weaving.

Future versions of R may enhance this customization mechanism, and provide a similar scheme for `R CMD build`.

There are other internal settings that can be changed via environment variables `_R_CHECK_*_`: see the Perl source code.

1.4 Writing package vignettes

In addition to the help files in Rd format, R packages allow the inclusion of documents in arbitrary other formats. The standard location for these is subdirectory '`inst/doc`' of a source package, the contents will be copied to subdirectory '`doc`' when the package is installed. Pointers from package help indices to the installed documents are automatically created. Documents in '`inst/doc`' can be in arbitrary format, however we strongly recommend to provide them in PDF format, such that users on all platforms can easily read them. To ensure that they can be accessed from a browser, the file names should start with an ASCII letter and be comprised entirely of ASCII letters or digits or minus or underscore.

A special case are documents in Sweave format, which we call *package vignettes*. Sweave allows the integration of LaTeX documents and R code and is contained in package **utils** which is part of the base R distribution, see the `Sweave` help page for details on the document format. Package vignettes found in directory '`inst/doc`' are tested by `R CMD check` by executing all R code chunks they contain to ensure consistency between code and documentation. Code chunks with option `eval=FALSE` are not tested. The R working directory for all vignette tests in `R CMD check` is the *installed* version of the '`doc`' subdirectory. Make sure all files needed by the vignette (data sets, ...) are accessible by either placing them in the '`inst/doc`' hierarchy of the source package, or using calls to `system.file()`.

`R CMD build` will automatically create PDF versions of the vignettes for distribution with the package sources. By including the PDF version in the package sources it is not necessary that the vignettes can be compiled at install time, i.e., the package author can use private LaTeX extensions which are only available on his machine.[8]

By default `R CMD build` will run `Sweave` on all files in Sweave format. If no '`Makefile`' is found in directory '`inst/doc`', then `texi2dvi --pdf` is run on all vignettes. Whenever a '`Makefile`' is found, then `R CMD build` will try to run `make` after the `Sweave` step, such that PDF manuals can be created from arbitrary source formats (plain LaTeX files, ...). The '`Makefile`' should take care of both creation of PDF files and cleaning up afterwards, i.e., delete all files that shall not appear in the final package archive. Note that the `make` step is executed independently from the presence of any files in Sweave format.

It is no longer necessary to provide a '`00Index.dcf`' file in the '`inst/doc`' directory—the corresponding information is generated automatically from the `\VignetteIndexEntry` statements in all Sweave files when installing from source, or when using the package builder (see Section 1.3 [Checking and building packages], page 14). The `\VignetteIndexEntry` statement is best placed in LaTeX comment, such that no definition of the command is necessary.

At install time an HTML index for all vignettes is automatically created from the `\VignetteIndexEntry` statements unless a file '`index.html`' exists in directory '`inst/doc`'. This index is linked into the HTML help system for each package.

[8] provided the conditions of the licence are met: many would see this as incompatible with an Open Source licence.

1.5 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. Run `R CMD check` to check that the package will install and will runs its examples, and that the documentation is complete and can be processed. If the package contains code that needs to be compiled, try to enable a reasonable amount of diagnostic messaging ("warnings") when compiling, such as e.g. '`-Wall -pedantic`' for tools from GCC, the Gnu Compiler Collection. (If R was not configured accordingly, one can achieve this e.g. via `PKG_CFLAGS` and related variables.)
2. Run `R CMD build` 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. In principle, packages must pass `R CMD check` without warnings to be admitted to the main CRAN package area.

When all the testing is done, upload the '`.tar.gz`' file, using '`anonymous`' as log-in name and your e-mail address as password, to

    ftp://cran.R-project.org/incoming/

(note: use `ftp` and not `sftp` to connect to this server) and send a message to `cran@r-project.org` about it. The CRAN maintainers will run these tests before putting a submission in the main archive.

Note that the fully qualified name of the '`.tar.gz`' file must be of the form

'*package*_*version*[_*engine*[_*type*]]',

where the '`[ ]`' indicates that the enclosed component is optional, *package* and *version* are the corresponding entries in file '`DESCRIPTION`', *engine* gives the S engine the package is targeted for and defaults to '`R`', and *type* indicated whether the file contains source or binaries for a certain platform, and defaults to '`source`'. I.e.,

```
OOP_0.1-3.tar.gz
OOP_0.1-3_R.tar.gz
OOP_0.1-3_R_source.tar.gz
```

are all equivalent and indicate an R source package, whereas

```
OOP_0.1-3_Splus6_sparc-sun-solaris.tar.gz
```

is a binary package for installation under Splus6 on the given platform.

This naming scheme has been adopted to ensure usability of code across S engines. R code and utilities operating on package '`.tar.gz`' files can only be assumed to work provided that this naming scheme is respected. Of course, `R CMD build` automatically creates valid file names.

Note that CRAN generally does not accept submissions of precompiled binaries due to security reasons.

1.6 Package name spaces

R has a name space management system for packages. This system allows the package writer to specify which variables in the package should be *exported* to make them available to package users, and which variables should be *imported* from other packages.

The current mechanism for specifying a name space for a package is to place a '`NAMESPACE`' file in the top level package directory. This file contains *name space directives* describing the imports and exports of the name space. Additional directives register any shared objects to be loaded and any S3-style methods that are provided. Note that although the file looks like R code (and often has R-style comments) it is not processed as R code. Only very simple conditional processing of `if` statements is implemented.

Like other packages, packages with name spaces are loaded and attached to the search path by calling `library`. Only the exported variables are placed in the attached frame. Loading a package that imports variables from other packages will cause these other packages to be loaded as well (unless they have already been loaded), but they will *not* be placed on the search path by these implicit loads.

Name spaces are *sealed* once they are loaded. Sealing means that imports and exports cannot be changed and that internal variable bindings cannot be changed. Sealing allows a simpler implementation strategy for the name space mechanism. Sealing also allows code analysis and compilation tools to accurately identify the definition corresponding to a global variable reference in a function body.

Note that adding a name space to a package changes the search strategy. The package name space comes first in the search, then the imports, then the base name space and then the normal search path.

1.6.1 Specifying imports and exports

Exports are specified using the `export` directive in the '`NAMESPACE`' file. A directive of the form

```
export(f, g)
```

specifies that the variables `f` and `g` are to be exported. (Note that variable names may be quoted, and reserved words and non-standard names such as `[<-.fractions` must be.)

For packages with many variables to export it may be more convenient to specify the names to export with a regular expression using `exportPattern`. The directive

```
exportPattern("^[^\\.]")
```

exports all variables that do not start with a period.

A package with a name space implicitly imports the base name space. Variables exported from other packages with name spaces need to be imported explicitly using the directives `import` and `importFrom`. The `import` directive imports all exported variables from the specified package(s). Thus the directives

```
import(foo, bar)
```

specifies that all exported variables in the packages **foo** and **bar** are to be imported. If only some of the exported variables from a package are needed, then they can be imported using `importFrom`. The directive

```
importFrom(foo, f, g)
```

specifies that the exported variables `f` and `g` of the package **foo** are to be imported.

It is possible to export variables from a name space that it has imported from other namespaces.

If a package only needs a few objects from another package it can use a fully qualified variable reference in the code instead of a formal import. A fully qualified reference to the function `f` in package **foo** is of the form `foo:::f`. This is less efficient than a formal import and also loses the advantage of recording all dependencies in the '`NAMESPACE`' file, so this approach is usually not recommended. Evaluating `foo:::f` will cause package **foo** to be loaded, but not attached, if it was not loaded already—this can be an advantage is delaying the loading of a rarely used package.

Using `foo:::f` allows access to unexported objects: to confine references to exported objects use `foo::f`.

1.6.2 Registering S3 methods

The standard method for S3-style `UseMethod` dispatching might fail to locate methods defined in a package that is imported but not attached to the search path. To ensure that these methods are available the packages defining the methods should ensure that the generics are imported and register the methods using `S3method` directives. If a package defines a function `print.foo` intended to be used as a `print` method for class `foo`, then the directive

```
S3method(print, foo)
```

ensures that the method is registered and available for `UseMethod` dispatch. The function `print.foo` does not need to be exported. Since the generic `print` is defined in **base** it does not need to be imported explicitly. This mechanism is intended for use with generics that are defined in a name space. Any methods for a generic defined in a package that does not use a name space should be exported, and the package defining and exporting the methods should be attached to the search path if the methods are to be found.

(Note that function and class names may be quoted, and reserved words and non-standard names such as `[<-` and `function` must be.)

1.6.3 Load hooks

There are a number of hooks that apply to packages with name spaces. See `help(".onLoad")` for more details.

Packages with name spaces do not use the `.First.lib` function. Since loading and attaching are distinct operations when a name space is used, separate hooks are provided for each. These hook functions are called `.onLoad` and `.onAttach`. They take the same arguments as `.First.lib`; they should be defined in the name space but not exported.

However, packages with name spaces *do* use the `.Last.lib` function. There is also a hook `.onUnload` which is called when the name space is unloaded (via a call to `unloadNamespace`) with argument the full path to the directory in which the package was installed. `.onUnload` should be defined in the name space and not exported, but `.Last.lib` does need to be exported.

Packages are not likely to need `.onAttach` (except perhaps for a start-up banner); code to set options and load shared objects should be placed in a `.onLoad` function, or use made of the `useDynLib` directive described next.

There can be one or more `useDynLib` directives which allow shared objects that need to be loaded to be specified in the '`NAMESPACE`' file. The directive

```
useDynLib(foo)
```

registers the shared object `foo` for loading with `library.dynam`. Loading of registered object(s) occurs after the package code has been loaded and before running the load hook function. Packages that would only need a load hook function to load a shared object can use the `useDynLib` directive instead.

User-level hooks are also available: see the help on function `setHook`.

The `useDynLib` directive also accepts the names of the native routines that are to be used in R via the `.C`, `.Call`, `.Fortran` and `.External` interface functions. These are given as additional arguments to the directive, for example,

```
useDynLib(foo, myRoutine, myOtherRoutine)
```

By specifying these names in the `useDynLib` directive, the native symbols are resolved when the package is loaded and R variables identifying these symbols are added to the package's name space with these names. These can be used in the `.C`, `.Call`, `.Fortran` and `.External` calls in place of the name of the routine and the `PACKAGE` argument. For instance, we can call the routine `myRoutine` from R with the code

```
 .Call(myRoutine, x, y)
```

rather than

```
.Call("myRoutine", x, y, PACKAGE = "foo")
```

There are at least two benefits to this approach. Firstly, the symbol lookup is done just once for each symbol rather than each time it the routine is invoked. Secondly, this removes any ambiguity in resolving symbols that might be present in several compiled libraries. In particular, it allows for correctly resolving routines when different versions of the same package are loaded concurrently in the same R session.

In some circumstances, there will already be an R variable in the package with the same name as a native symbol. For example, we may have an R function in the package named `myRoutine`. In this case, it is necessary to map the native symbol to a different R variable name. This can be done in the `useDynLib` directive by using named arguments. For instance, to map the native symbol name `myRoutine` to the R variable `myRoutine_sym`, we would use

```
useDynLib(foo, myRoutine_sym = myRoutine, myOtherRoutine)
```

We could then call that routine from R using the command

```
.Call(myRoutine_sym, x, y)
```

Symbols without explicit names are assigned to the R variable with that name.

In some cases, it may be preferable not to create R variables in the package's name space that identify the native routines. It may be too costly to compute these for many routines when the package is loaded if many of these routines are not likely to be used. In this case, one can still perform the symbol resolution correctly using the DLL, but do this each time the routine is called. Given a reference to the DLL as an R variable, say `dll`, we can call the routine `myRoutine` using the expression

```
.Call(dll$myRoutine, x, y)
```

The `$` operator resolves the routine with the given name in the DLL using a call to `getNativeSymbol`. This is the same computation as above where we resolve the symbol when the package is loaded. The only difference is that this is done each time in the case of `dll$myRoutine`.

In order to use this dynamic approach (e.g., `dll$myRoutine`), one needs the reference to the DLL as an R variable in the package. The DLL can be assigned to a variable by using the `variable = dllName` format used above for mapping symbols to R variables. For example, if we wanted to assign the DLL reference for the DLL `foo` in the example above to the variable `myDLL`, we would use the following directive in the 'NAMESPACE' file:

```
myDLL = useDynLib(foo, myRoutine_sym = myRoutine, myOtherRoutine)
```

Then, the R variable `myDLL` is in the package's name space and available for calls such as `myDLL$dynRoutine` to access routines that are not explicitly resolved at load time.

If the package has registration information (see Section 5.4 [Registering native routines], page 62), then we can use that directly rather than specifying the list of symbols again in the `useDynLib` directive in the 'NAMESPACE' file. Each routine in the registration information is specified by giving a name by which the routine is to be specified along with the address of the routine and any information about the number and type of the parameters. Using the `.registration` argument of `useDynLib`, we can instruct the name space mechanism to create R variables for these symbols. For example, suppose we have the following registration information for a DLL named `myDLL`:

```
R_CMethodDef cMethods[] = {
   {"foo", &foo, 4, {REALSXP, INTSXP, STRSXP, LGLSXP}},
   {"bar_sym", &bar, 0},
   {NULL, NULL, 0}
};

R_CallMethodDef callMethods[] = {
   {"R_call_sym", &R_call, 4},
```

```
    {"R_version_sym", &R_version, 0},
    {NULL, NULL, 0}
};
```

Then, the directive in the 'NAMESPACE' file

```
useDynLib(myDLL, .registration = TRUE)
```

causes the DLL to be loaded and also for the R variables `foo`, `bar_sym`, `R_call_sym` and `R_version_sym` to be defined in the package's name space.

Note that the names for the R variables are taken from the entry in the registration information and do not need to be the same as the name of the native routine. This allows the creator of the registration information to map the native symbols to non-conflicting variable names in R, e.g. `R_version` to `R_version_sym` for use in an R function such as

```
R_version <- function()
{
  .Call(R_version_sym)
}
```

Using argument `.fixes` allows an automatic prefix to be added to the registered symbols, which can be useful when working with an existing package. For example, package **KernSmooth** has

```
useDynLib(KernSmooth, .registration = TRUE, .fixes = "F_")
```

which makes the R variables corresponding to the Fortran symbols `F_bkde` and so on, and so avoid clashes with R code in the name space.

More information about this symbol lookup, along with some approaches for customizing it, is available from http://www.omegahat.org/examples/RDotCall.

1.6.4 An example

As an example consider two packages named **foo** and **bar**. The R code for package **foo** in file 'foo.R' is

```
x <- 1
f <- function(y) c(x,y)
foo <- function(x) .Call("foo", x, PACKAGE="foo")
print.foo <- function(x, ...) cat("<a foo>\n")
```

Some C code defines a C function compiled into DLL `foo` (with an appropriate extension). The 'NAMESPACE' file for this package is

```
useDynLib(foo)
export(f, foo)
S3method(print, foo)
```

The second package **bar** has code file 'bar.R'

```
c <- function(...) sum(...)
g <- function(y) f(c(y, 7))
h <- function(y) y+9
```

and 'NAMESPACE' file

```
import(foo)
export(g, h)
```

Calling `library(bar)` loads **bar** and attaches its exports to the search path. Package **foo** is also loaded but not attached to the search path. A call to `g` produces

```
> g(6)
[1]  1 13
```

This is consistent with the definitions of `c` in the two settings: in **bar** the function `c` is defined to be equivalent to `sum`, but in **foo** the variable `c` refers to the standard function `c` in **base**.

1.6.5 Summary – converting an existing package

To summarize, converting an existing package to use a name space involves several simple steps:

- Identify the public definitions and place them in `export` directives.
- Identify S3-style method definitions and write corresponding `S3method` declarations.
- Identify dependencies and replace any `require` calls by `import` directives (and make appropriate changes in the `Depends` and `Imports` fields of the '`DESCRIPTION`' file).
- Replace `.First.lib` functions with `.onLoad` functions or `useDynLib` directives.

Some code analysis tools to aid in this process are currently under development.

1.6.6 Name spaces with formal classes and methods

Some additional steps are needed for packages which make use of formal (S4-style) classes and methods (unless these are purely used internally). The package should have `Depends: methods` in its '`DESCRIPTION`' file and any classes and methods which are to be exported need to be declared in the '`NAMESPACE`' file. For example, the **stats** package has

```
export(mle)
importFrom(graphics, plot)
importFrom(stats, AIC, coef, confint, logLik, optim, profile,
   qchisq, update, vcov)
exportClasses(mle, profile.mle, summary.mle)
exportMethods(BIC, coef, confint, logLik, plot, profile,
              summary, show, update, vcov)
export(AIC)
```

All formal classes need to be listed in an `exportClasses` directive. Generics for which formal methods are defined need to be declared in an `exportMethods` directive, and where the generics are formed by taking over existing functions, those functions need to be imported (explicitly unless they are defined in the `base` name space).

Note that exporting methods on a generic in the namespace will also export the generic, and exporting a generic in the namespace will also export its methods. Where a generic has been created in the package solely to add S4 methods to it, it can be declared *via* either or both of `exports` or `exportMethods`, but the latter seems clearer (and is used in the **stats4** example above). On the other hand, where a generic is created in a package without methods (such as `AIC` in **stats4**), `exports` must be used.

Further, a package using classes and methods defined in another package needs to import them, with directives

```
importClassesFrom(package, ...)
importMethodsFrom(package, ...)
```

listing the classes and functions with methods respectively. Suppose we had two small packages **A** and **B** with **B** using **A**. Then they could have `NAMESPACE` files

```
export(f1, ng1)
exportMethods("[")
exportClasses(c1)
```

and

```
importFrom(A, ng1)
importClassesFrom(A, c1)
importMethodsFrom(A, f1)
export(f4, f5)
exportMethods(f6, "[")
exportClasses(c1, c2)
```

respectively.

Note that `importMethodsFrom` will also import any generics defined in the namespace on those methods.

If your package imports the whole of a name space, it will automatically import the classes from that namespace. It will also import methods, but it is best to do so explicitly, especially where there are methods being imported from more than one namespace.

1.7 Writing portable packages

Portable packages should have simple file names: use only alphanumeric ASCII characters and `.`, and avoid those names not allowed under Windows which are mentioned above.

`R CMD check` provides a basic set of checks, but often further problems emerge when people try to install and use packages submitted to CRAN – many of these involve compiled code. Here are some further checks that you can do to make your package more portable.

- If your package has a '`configure`' script, provide a '`configure.win`' script to be used on Windows. The CRAN binary packages for Windows are built automatically, and if your package does not build without intervention it is unlikely to be easily available to a high proportion of R users.
- Make use of the abilities of your compilers to check the standards-conformance of your code. For example, `gcc` can be used with options '`-Wall -pedantic`' to alert you to potential problems. Do not be tempted to assume that these are pure pedantry: for example R is still used on platforms where the C compiler does not accept C++/C99 comments (starting `//`).

  If you use FORTRAN, `ftnchek` (`http://www.dsm.fordham.edu/~ftnchek/`) provides thorough testing of conformance to the standard.
- Do be very careful with passing arguments between R, C and FORTRAN code. In particular, `long` in C will be 32-bit on most R platforms (including those mostly used by the CRAN maintainers), but 64-bit on many modern Unix and Linux platforms. It is rather unlikely that the use of `long` in C code has been thought through: if you need a longer type than `int` you should use a configure test for a C99 type such as `int_fast64_t` (and failing that, `long long`) and typedef your own type to be `long` or `long long`, or use another suitable type (such as `size_t`). Note that `integer` in FORTRAN corresponds to `int` in C on all R platforms.
- Errors in memory allocation and reading/writing outside arrays are very common causes of crashes (e.g., segfaults) on some machines. See Section 4.3.2 [Using valgrind], page 54 for a tool which can be used to look for this.
- The Mac OS X linker has some restrictions not found on other platforms. Try to ensure that C entry points shared between source files are declared as `extern` in all but one of

the files. (This is no longer needed in recent versions of R, but is if your package is not restricted to such versions.)

- Many platforms will allow unsatisfied entry points in compiled code, but will crash the application (here R) if they are ever used. Some (notably Windows) will not. Looking at the output of

  ```
  nm -pg mypkg.so  # or other extension such as '.sl' or '.dylib'
  ```

  and checking if any of the symbols marked `U` is unexpected is a good way to avoid this.
- Conflicts between symbols in DLLs are handled in very platform-specific ways. Good ways to avoid trouble are to make as many symbols as possible static (check with `nm -pg`), and to use unusual names, as well as ensuring you have used the `PACKAGE` argument that `R CMD check` checks for.

1.7.1 Encoding issues

Care is needed if your package contains non-ASCII text, and in particular if it is intended to be used in more than one locale. It is possible to mark the encoding used in the '`DESCRIPTION`' file and in '`.Rd`' files, as discussed elsewhere in this manual. What was not possible before R 2.5.0 was to mark the encoding used by character strings in R: if you want your package to work with earlier versions of R please consult the advice in the R 2.4.x version of this manual.

First, consider carefully if you really need non-ASCII text. Most users of R will only be able to view correctly text in their native language group (e.g. Western European, Eastern European, Simplified Chinese) and ASCII. Other characters may not be rendered at all, rendered incorrectly, or cause your R code to give an error. For documentation, marking the encoding and including ASCII transliterations is likely to do a reasonable job.

The most favourable circumstance is using UTF-8-encoded text in a package that will only ever be used in a UTF-8 locale (and hence not on Windows, for example). In that case it is likely that text will be rendered correctly in the terminal/console used to run R, and files written will be readable by other UTF-8-aware applications. However, plotting will be problematic. On-screen plotting using the '`X11()`' device will use a font that only covers a small proportion of UTF-8, and different fonts will likely need to be selected for Polish, Russian and Japanese (see `help("X11")`). Using '`postscript`' or '`pdf`' will choose a default 8-bit encoding depending on the language of the UTF-8 locale, and your users would need to be told how to select the '`encoding`' argument.

Another fairly common scenario is where a package will only be used in one language, e.g. French. It is not very safe to assume that all such users would have their computers set to a French locale, but let us assume so. The problem then is that there are several possible encodings for French locales, the most common ones being '`CP1252`' (Windows), '`ISO 8859-1`' (latin-1), '`ISO 8859-15`' (latin-9 which includes the Euro), and '`UTF-8`'. For characters in the French language the first three agree, but they do not agree with '`UTF-8`'. Further, you (or different users) can run R in different locales in different sessions, say '`fr_CA.utf8`' one day and '`fr_CH.iso88591`' the next. As from R 2.5.0, declaring the encoding as either '`latin1`' or '`UTF-8`' in the '`DESCRIPTION`' file will enable this to work. If you have character data in '`.rda`' files (for use by `data` or LazyData) these need to have been prepared and `save`d in R 2.5.0 in an appropriate locale (or marked via `Encoding`). For example (from package **FactoMineR** version 1.02):

```
> library(FactoMineR)
> data(wine)
> Encoding(names(wine)) <- "latin1"
> Encoding(levels(wine$Terroir)) <- "latin1"
> save(wine, file="wine.rda")
```

was used to update a '`.rda`' file.

If you want to run `R CMD check` on a Unix-alike over a package that sets the encoding you may need to specify a suitable locale via an environment variable. The default is equivalent setting `R_ENCODING_LOCALES` to

```
"latin1=en_US:latin2=pl_PL:UTF-8=en_US.utf8:latin9=fr_FR.iso885915@euro"
```

(which is appropriate for a system based on `glibc`) except if the current locale is UTF-8 and '`iconv`' is available, when the package code is translated to UTF-8 for syntax checking.

1.8 Diagnostic messages

Now that diagnostic messages can be made available for translation, it is important to write them in a consistent style. Using the tools described in the next section to extract all the messages can give a useful overview of your consistency (or lack of it).

Some guidelines follow.

- Messages are sentence fragments, and not viewed in isolation. So it is conventional not to capitalize the first word and not to end with a period (or other punctuation).
- Try not to split up messages into small pieces. In C error messages use a single format string containing all English words in the messages.

  In R error messages do not construct a message with `paste` (such messages will not be translated) but via multiple arguments to `stop` or `warning`, or via `gettextf`.
- Do not use colloquialisms such as "can't" and "don't".
- If possible, make quotation marks part of your message as different languages have different conventions. In R messages this means not using `sQuote` or `dQuote` except where the argument is a variable.

  Conventionally single quotation marks are used for quotations such as

  ```
  'ord' must be a positive integer, at most the number of knots
  ```

  and double quotation marks when referring to an R character string such as

  ```
  'format' must be "normal" or "short" - using "normal"
  ```

  Since ASCII does not contain directional quotation marks, it is best to use '' and let the translator (including automatic translation) use directional quotations where available. The range of quotation styles is immense: unfortunately we cannot reproduce them in a portable `texinfo` document. But as a taster, some languages use 'up' and 'down' (comma) quotes rather than left or right quotes, and some use guillemets (and some use what Adobe calls 'guillemotleft' to start and others use it to end).
- Occasionally messages need to be singular or plural (and in other languages there may be no such concept or several plural forms – Slovenian has four). So avoid constructions such as was once used in `library`

  ```
  if((length(nopkgs) > 0) && !missing(lib.loc)) {
      if(length(nopkgs) > 1)
          warning("libraries ",
                  paste(sQuote(nopkgs), collapse = ", "),
                  " contain no packages")
      else
          warning("library ", paste(sQuote(nopkgs)),
                  " contains no package")
  }
  ```

  and was replaced by

  ```
  if((length(nopkgs) > 0) && !missing(lib.loc)) {
      pkglist <- paste(sQuote(nopkgs), collapse = ", ")
      msg <- sprintf(ngettext(length(nopkgs),
  ```

```
                              "library %s contains no packages",
                              "libraries %s contain no packages"),
                            pkglist)
          warning(msg, domain=NA)
      }
```

Note that it is much better to have complete clauses as here, since in another language one might need to say 'There is no package in library %s' or 'There are no packages in libraries %s'.

1.9 Internationalization

There are mechanisms to translate the R- and C-level error and warning messages. There are only available if R is compiled with NLS support (which is requested by `configure` option '`--enable-nls`', the default).

The procedures make use of `msgfmt` and `xgettext` which are part of GNU `gettext` and this will need to be installed: Windows users can find pre-compiled binaries at the GNU archive mirrors and packaged with the `poEdit` package (http://poedit.sourceforge.net/download.php#win32).

1.9.1 C-level messages

The process of enabling translations is

- In a header file that will be included in all the C files containing messages that should be translated, declare

```
#include <R.h>  /* to include Rconfig.h */

#ifdef ENABLE_NLS
#include <libintl.h>
#define _(String) dgettext ("pkg", String)
/* replace pkg as appropriate */
#else
#define _(String) (String)
#endif
```

- For each message that should be translated, wrap it in `_(...)`, for example

```
error(_("'ord' must be a positive integer"));
```

- In the package's '`src`' directory run

```
xgettext --keyword=_ -o pkg.pot *.c
```

The file '`src/pkg.pot`' is the template file, and conventionally this is shipped as '`po/pkg.pot`'. A translator to another language makes a copy of this file and edits it (see the `gettext` manual) to produce say '`ll.po`', where *ll* is the code for the language in which the translation is to be used. (This file would be shipped in the '`po`' directory.) Next run `msgfmt` on '`ll.po`' to produce '`ll.mo`', and copy that to '`inst/po/ll/LC_MESSAGES/pkg.mo`'. Now when the package is loaded after installation it will look for translations of its messages in the '`po/lang/LC_MESSAGES/pkg.mo`' file for any language *lang* that matches the user's preferences (via the setting of the `LANGUAGE` environment variable or from the locale settings).

1.9.2 R messages

Mechanisms to support the automatic translation of R `stop`, `warning` and `message` messages are in place, most easily if the package has a name space. They make use of message catalogs in the same way as C-level messages, but using domain `R-pkg` rather than *pkg*. Translation of character strings inside `stop`, `warning` and `message` calls is automatically enabled, as well as other messages enclosed in calls to `gettext` or `gettextf`. (To suppress this, use argument `domain=NA`.)

Tools to prepare the '`R-pkg.pot`' file are provided in package **tools**: `xgettext2pot` will prepare a file from all strings occurring inside `gettext/gettextf`, `stop`, `warning` and `message` calls. Some of these are likely to be spurious and so the file is likely to need manual editing. `xgettext` extracts the actual calls and so is more useful when tidying up error messages.

Translation of messages which might be singular or plural can be very intricate: languages can have up to four different forms. The R function `ngettext` provides an interface to the C function of the same name, and will choose an appropriate singular or plural form for the selected language depending on the value of its first argument `n`.

Packages without name spaces will need to use `domain="R-pkg"` explicitly in calls to `stop`, `warning`, `message`, `gettext/gettextf` and `ngettext`.

1.10 Package types

The '`DESCRIPTION`' file has an optional field `Type` which if missing is assumed to be `Package`, the sort of extension discussed so far in this chapter. Currently two other types are recognized, both of which need write permission in the R installation tree.

1.10.1 Frontend

This is a rather general mechanism, designed for adding new front-ends such as the **gnomeGUI** package. If a '`configure`' file is found in the top-level directory of the package it is executed, and then if a `Makefile` is found (often generated by '`configure`'), `make` is called. If `R CMD INSTALL --clean` is used `make clean` is called. No other action is taken.

`R CMD build` can package up this type of extension, but `R CMD check` will check the type and skip it.

1.10.2 Translation

Conventionally, a translation package for language *ll* is called **Translation-*ll*** and has `Type: Translation`. It needs to contain the directories '`share/locale/ll`' and '`library/pkgname/po/ll`', or at least those for which translations are available. The files '`.mo`' are installed in the parallel places in the R installation tree.

For example, a package **Translation-it** might be prepared from an installed (and tested) version of R by

```
mkdir Translation-it
cd Translation-it
(cd $R_HOME; tar cf - share/locale/it library/*/po/it) | tar xf -
# the next step is not needed on Windows
msgfmt -c -o share/locale/it/LC_MESSAGES/RGui.mo $R_SRC_HOME/po/RGui-it.gmo
# create a DESCRIPTION file
cd ..
R CMD build Translation-it
```

It is probably appropriate to give the package a version number based on the version of R which has been translated. So the '`DESCRIPTION`' file might look like

```
Package: Translation-it
Type: Translation
Version: 2.2.1-1
Title: Italian Translations for R 2.2.1
Description: Italian Translations for R 2.2.1
Author: The translators
```

```
Maintainer: Some Body <somebody@some.where.net>
License: GPL (>= 2)
```

1.11 Services

Several members of the R project have set up services to assist those writing R packages, particularly those intended for public distribution.

win-builder.r-project.org offers the automated preparation of Windows binaries from well-tested source packages.

R-Forge (R-Forge.r-project.org) and RForge (www.rforge.net) are similar services with similar names. Both provide source-code management through SVN, daily building and checking, mailing lists and a repository that can be accessed *via* `install.packages`. Package developers have the opportunity to present their work on the basis of project websites or news announcements. Mailing lists, forums or wikis provide useRs with convenient instruments for discussions and for exchanging information between developers and/or interested useRs.

2 Writing R documentation files

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 1200 such files which can be found in the 'src/library/*pkg*/man' directories of the R source tree, where *pkg* stands for the standard packages which are included in the R distribution.

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

```
\name{load}
\alias{load}
\title{Reload Saved Datasets}
\description{
  Reload the datasets written to a file with the function
  \code{save}.
}
\usage{
load(file, envir = parent.frame())
}
\arguments{
  \item{file}{a connection or a character string giving the
    name of the file to load.}
  \item{envir}{the environment where the data should be
    loaded.}
}
\seealso{
  \code{\link{save}}.
}
\examples{
## save all data
save(list = ls(), file= "all.Rdata")

## restore the saved values to the current environment
load("all.Rdata")

## restore the saved values to the workspace
load("all.Rdata", .GlobalEnv)
}
\keyword{file}
```

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 the "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{name}`

name typically[1] is the basename of the Rd file containing the documentation. It is the "name" of the Rd object represented by the file and has to be unique in a package.

`\alias{topic}`

The `\alias` entries specify all "topics" the file documents. This information is collected into index data bases for lookup by the on-line (plain text and HTML) help systems. The *topic* can contain spaces, but (for historical reasons) leading and trailing spaces will be stripped.

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

Also, it is often convenient to have several different ways to refer to an R object, and an `\alias` does not need to be the name of an object.

Note that the `\name` is not necessarily a topic documented, and if so desired it needs to have an explicit `\alias` entry (as in this example).

`\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). Use of characters other than English text and punctuation (e.g., '<') may limit portability.

`\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 in typewriter font. This is a verbatim-like command, so some characters need to be escaped (see Section 2.7 [Insertions], page 40).

The usage information specified should match the function definition *exactly* (such that automatic checking for consistency between code and documentation is possible).

It is no longer advisable to use `\synopsis` for the actual synopsis and show modified synopses in the `\usage`. Support for `\synopsis` will be removed eventually. To indicate that a function can be "used" in several different ways, depending on the named arguments specified, use section `\details`. E.g., 'abline.Rd' contains

---

[1] There can be exceptions: for example Rd files are not allowed to start with a dot, and have to be uniquely named on a case-insensitve file system.

```
\details{
  Typical usages are
\preformatted{
abline(a, b, untf = FALSE, \dots)
......
}
```

Use `\method{generic}{class}` to indicate the name of an S3 method for the generic function *generic* for objects inheriting from class `"class"`. In the printed versions, this will come out as *generic* (reflecting the understanding that methods should not be invoked directly but via method dispatch), but `codoc()` and other QC tools always have access to the full name.

For example, '`print.ts.Rd`' contains

```
\usage{
\method{print}{ts}(x, calendar, \dots)
}
```

which will print as

```
Usage:

     ## S3 method for class 'ts':
     print(x, calendar, ...)
```

Usage for replacement functions should be given in the style of `dim(x) <- value` rather than explicitly indicating the name of the replacement function (`"dim<-"` in the above). Similarly, one can use `\method{generic}{class}(arglist) <- value` to indicate the usage of an S3 replacement method for the generic replacement function `"generic<-"` for objects inheriting from class `"class"`.

Usage for S3 methods for extracting or replacing parts of an object, S3 methods for members of the Ops group, and S3 methods for user-defined (binary) infix operators ('`%xxx%`') follows the above rules, using the appropriate function names. E.g., '`Extract.factor.Rd`' contains

```
\usage{
\method{[}{factor}(x, \dots, drop = FALSE)
\method{[[}{factor}(x, i)
\method{[}{factor}(x, \dots) <- value
}
```

which will print as

```
Usage:

     ## S3 method for class 'factor':
     x[..., drop = FALSE]
     ## S3 method for class 'factor':
     x[[i]]
     ## S3 replacement method for class 'factor':
     x[...] <- value
```

`\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, '`pie.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 37, and Section 2.5 [Cross-references], page 39).

`\examples{...}`
Examples of how to use the function. These are set as formatted in typewriter font: see Section 2.7 [Insertions], page 40 for when characters need to be escaped. (Markup `\link` and `\var` will be interpreted, but no other.)

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 `R CMD check`. You can use `\dontrun{}` for commands that should only be shown, but not run, and `\dontshow{}` for extra commands for testing that should not be shown to users, but will be run by `example()`. (Previously this was called `\testonly`, and that is still accepted.)

For example,

```
x <- runif(10)        # Shown and run.
\dontrun{plot(x)}     # Only shown.
\dontshow{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). Code included in `\dontrun` is indicated by comments in the processed help files.

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 listed by `data()` (see `?data` for more info).

`\keyword{key}`
: Each `\keyword` entry should specify one of the standard keywords as listed in file 'KEYWORDS' in the R documentation directory (default 'R_HOME/doc'). Use e.g. `file.show(file.path(R.home("doc"), "KEYWORDS"))` to inspect the standard keywords from within R. There must be at least one `\keyword` entry, but can be more than one if the R object being documented falls into more than one category.

  The special keyword '`internal`' marks a page of internal objects that are not part of the packages' API. If the help page for object `foo` has keyword '`internal`', then `help(foo)` gives this help page, but `foo` is excluded from several object indices, like the alphabetical list of objects in the HTML help system.

The R function `prompt` facilitates the construction of files documenting R objects. If `foo` is an R function, then *prompt(foo)* produces file '`foo.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 data sets

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/datasets/man/rivers.Rd`' which documents the standard R data set `rivers`.

```
\name{rivers}
\docType{data}
\alias{rivers}
\title{Lengths of Major North American Rivers}
\description{
  This data set gives the lengths (in miles) of 141 \dQuote{major}
  rivers in North America, as compiled by the US Geological
  Survey.
}
\usage{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.

`\docType{...}`
: Indicates the "type" of the documentation object. Always '`data`' for data sets.

`\format{...}`
: A description of the format of the data set (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 38, 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 is always *`bar`* or (for packages which do not use lazy-loading of data) `data(`*`bar`*`)`. (In particular, only document a *single* data object per Rd file.)
- The `\keyword` entry is always '`datasets`'.

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

2.1.3 Documenting S4 classes and methods

There are special ways to use the '`?`' operator, namely '`class?`*`topic`*' and '`methods?`*`topic`*', to access documentation for S4 classes and methods, respectively. This mechanism depends on conventions for the topic names used in `\alias` entries. The topic names for S4 classes and methods respectively are of the form

```
class-class
generic,signature_list-method
```

where *signature_list* contains the names of the classes in the signature of the method (without quotes) separated by '`,`' (without whitespace), with '`ANY`' used for arguments without an explicit specification. E.g., '`genericFunction-class`' is the topic name for documentation for the S4 class `"genericFunction"`, and '`coerce,ANY,NULL-method`' is the topic name for documentation for the S4 method for `coerce` for signature `c("ANY", "NULL")`.

Skeletons of documentation for S4 classes and methods can be generated by using the functions `promptClass()` and `promptMethods()` from package **methods**. If it is necessary or desired to provide an explicit function declaration (in a `\usage` section) for an S4 method (e.g., if it has "surprising arguments" to be mentioned explicitly), one can use the special markup

```
\S4method{generic}{signature_list}(argument_list)
```

(e.g., '`\S4method{coerce}{ANY,NULL}(from, to)`').

To allow for making full use of the potential of the on-line documentation system, all user-visible S4 classes and methods in a package should at least have a suitable `\alias` entry in one of the package's Rd files. If a package has methods for a function defined originally somewhere else, and does not change the underlying default method for the function, the package is responsible for documenting the methods it creates, but not for the function itself or the default method.

See *`help("Documentation", package = "methods")`* for more information on using and creating on-line documentation for S4 classes and methods.

2.1.4 Documenting packages

Packages may have an overview man page with an `\alias` *`pkgname`*`-package`, e.g. '`utils-package`' for the **utils** package, when `package?`*`pkgname`* will open that help page. If a topic named *`pkgname`* does not exist in another Rd file, it is helpful to use this as an additional `\alias`.

Skeletons of documentation for a package can be generated using the function `promptPackage()`. If the `final = TRUE` argument is used, then the Rd file will be generated in final form, containing the information that would be produced by `library(help =` *`pkgname`*`)`. Otherwise (the default) comments will be inserted giving suggestions for content.

The only requirement for this page is that it include a `\docType{package}` statement. All other content is optional. We suggest that it should be a short overview, to give a reader unfamiliar with the package enough information to get started. More extensive documentation is better placed into a package vignette (see Section 1.4 [Writing package vignettes], page 18) and referenced from this page, or into individual man pages for the functions, datasets, or classes.

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 additional named sections are always inserted at a fixed position in the output (before `\note`, `\seealso` and the examples), no matter where they appear in the input (but in the same order as the input).

2.3 Marking text

The following logical markup commands are available for emphasizing or quoting text.

`\emph{`*`text`*`}`
`\strong{`*`text`*`}`
: Emphasize *text* using *italic* and **bold** font if possible; `\strong` is stronger.

`\bold{`*`text`*`}`
: Set *text* in **bold** font if possible.

`\sQuote{`*`text`*`}`
`\dQuote{`*`text`*`}`
: Portably single or double quote *text* (without hard-wiring the quotation marks).

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

`\code{`*`text`*`}`
: Indicate text that is a literal example of a piece of a program, e.g., a fragment of R code or the name of an R object, using `typewriter` font if possible. Some characters will need to be escaped (see Section 2.7 [Insertions], page 40). The only markup interpreted inside `\code` is `\link` and `\var`.

`\preformatted{`*`text`*`}`
: Indicate text that is a literal example of a piece of a program, using `typewriter` font if possible. The same characters need to be escaped as for `\code`. All other formatting, e.g. line breaks, is preserved. The closing brace should be on a line by itself.

`\kbd{`*`keyboard-characters`*`}`
: Indicate keyboard input, using *`slanted typewriter`* font if possible, so users can distinguish the characters they are supposed to type from those the computer outputs.

`\samp{`*`text`*`}`
: Indicate text that is a literal example of a sequence of characters.

`\pkg{`*`package_name`*`}`
: Indicate the name of an R package.

`\file{`*`file_name`*`}`
: Indicate the name of a file. Note that special characters do need to be escaped.

`\email{`*`email_address`*`}`
: Indicate an electronic mail address.

`\url{uniform_resource_locator}`
: Indicate a uniform resource locator (URL) for the World Wide Web.

`\var{metasyntactic_variable}`
: Indicate a metasyntactic variable. In some cases this will be rendered distinctly, e.g. in italic, but not in all[2].

`\env{environment_variable}`
: Indicate an environment variable.

`\option{option}`
: Indicate a command-line option.

`\command{command_name}`
: Indicate the name of a command.

`\dfn{term}`
: Indicate the introductory or defining use of a term.

`\cite{reference}`
: Indicate a reference without a direct cross-reference via `\link` (see Section 2.5 [Cross-references], page 39), such as the name of a book.

`\acronym{acronym}`
: Indicate an acronym (an abbreviation written in all capital letters), such as GNU.

Note that unless explicitly stated otherwise, special characters (see Section 2.7 [Insertions], page 40) must be escaped inside the above markup commands.

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] Currently it is rendered differently only in HTML conversions, and latex conversion outside '`\usage`' and '`\examples`' environments.

2.5 Cross-references

The markup `\link{foo}` (usually in the combination `\code{\link{foo}}`) produces a hyperlink to the help for *foo*. Here *foo* is a *topic*, that is the argument of `\alias` markup in another Rd file (possibly in another package). Hyperlinks are supported in some of the formats to which Rd files are converted, for example HTML and PDF, but ignored in others, e.g. the text and S nroff formats.

One main usage of `\link` is in the `\seealso` section of the help page, see Section 2.1 [Rd format], page 31.

Note that whereas leading and trailing spaces are stripped when extracting a topic from a `\alias`, they are not stripped when looking up the topic of a `\link`.

You can specify a link to a different topic than its name by `\link[=dest]{name}` which links to topic *dest* with name *name*. This can be used to refer to the documentation for S3/4 classes, for example `\code{"\link[=abc-class]{abc}"}` would be a way to refer to the documentation of an S4 class `"abc"` defined in your package, and `\code{"\link[=terms.object]{terms}"}` to the S3 `"terms"` class (in package **stats**). To make these easy to read, `\code{"\linkS4class{abc}"}` expands to the form given above.

There are two other forms of optional argument specified as `\link[pkg]{foo}` and `\link[pkg:bar]{foo}` to link to the package *pkg*, to *files* '`foo.html`' and '`bar.html`' respectively. These are rarely needed, perhaps to refer to not-yet-installed packages (but there the HTML help system will resolve the link at run time) or in the normally undesirable event that more than one package offers help on a topic[3] (in which case the present package has precedence so this is only needed to refer to other packages). They are only in used in (C)HTML help (and not for hyperlinks in LaTeX nor S sgml conversions of help pages), and link to the file rather than the topic (since there is no way to know which topics are in which files in an uninstalled package).

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" formulae (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 '`Poisson.Rd`':

```
\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 HTML and text on-line help we get

[3] a common example in CRAN packages is `\link[mgcv]{gam}`.

```
    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" character '`%`' and unpaired braces[4] *always* need to be escaped by '`\`', and '`\\`' can be used for backslash and needs to be when there two or more adjacent backslashes). Inside the verbatim-like commands (`usage`, `\code`, `\preformatted` and `\examples`), no other characters are special. Note that `\file` is **not** a verbatim-like command.

In "regular" text (not verbatim-like, no `\eqn`, ...), you currently must escape most LaTeX special characters, i.e., besides '`%`', '`{`', and '`}`', the 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`.

Text which might need to be represented differently in different encodings should be marked by `\enc`, e.g. `\enc{Jöreskog}{Joreskog}` where the first argument will be used where encodings are allowed and the second should be ASCII (and is used for e.g. the text conversion).

2.8 Indices

The `\alias` command (see Section 2.1.1 [Documenting functions], page 31) is used to specify the "topics" documented, which should include *all* R objects in a package such as functions and variables, data sets, and S4 classes and methods (see Section 2.1.3 [Documenting S4 classes and methods], page 36). The on-line help system searches the index data base consisting of all alias topics.

In addition, it is possible to provide "concept index entries" using `\concept`, which can be used for `help.search()` lookups. E.g., file '`cor.test.Rd`' in the standard package **stats** contains

```
\concept{Kendall correlation coefficient}
\concept{Pearson correlation coefficient}
\concept{Spearman correlation coefficient}
```

so that e.g. *`help.search("Spearman")`* will succeed in finding the help page for the test for association between paired samples using Spearman's ρ. (Note that concepts are not currently supported by the HTML search accessed *via* '`help.start()`'.)

(Note that `help.search()` only uses "sections" of documentation objects with no additional markup.)

If you want to cross reference such items from other help files via `\link`, you need to use `\alias` and not `\concept`.

[4] See the examples section in the file '`Paren.Rd`' for an example.

2.9 Platform-specific documentation

Sometimes the documentation needs to differ by platform. Currently two OS-specific options are available, '`unix`' and '`windows`', 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`' or '`windows`' subdirectory.

2.10 Encoding

Rd files are text files and so it is impossible to deduce the encoding they are written in unless ASCII: files with 8-bit characters could be UTF-8, Latin-1, Latin-9, KOI8-R, EUC-JP, *etc.* So the `\encoding{}` directive must be used to specify the encoding if it is not ASCII. (The `\encoding{}` directive must be on a line by itself, and in particular one containing no non-ASCII characters. As from R 2.6.0 the encoding declared in the '`DESCRIPTION`' file will be used if none is declared in the file.) This is used when creating the header of the HTML conversion (if not present, for back-compatibility the processing to HTML assumes that the file is in Latin-1 (ISO-8859-1)) and to add comments to the text and examples conversions. It is also used to indicate to LaTeX how to process the file (see below).

Wherever possible, avoid non-ASCII chars in Rd files, and even symbols such as '`<`', '`>`', '`$`', '`^`', '`&`', '`|`', '`@`', '`~`', and '`*`' outside verbatim environments (since they may disappear in fonts designed to render text).

For convenience, encoding names '`latin1`' and '`latin2`' are always recognized: these and '`UTF-8`' are likely to work fairly widely.

The `\enc` command (see Section 2.7 [Insertions], page 40) can be used to provide transliterations which will be used in conversions that do not support the declared encoding.

The LaTeX conversion converts an explicit encoding of the file to a

```
\inputencoding{some_encoding}
```

command, and this needs to be matched by a suitable invocation of the `\usepackage{inputenc}` command. The R utility `R CMD Rd2dvi` looks at the converted code and includes the encodings used: it might for example use

```
\usepackage[latin1,latin9,utf8]{inputenc}
```

(Use of `utf8` as an encoding requires LaTeX dated 2003/12/01 or later.)

Note that this mechanism works best with letters and for example the copyright symbol may be rendered as a subscript and the plus–minus symbol cannot be used in text.

2.11 Processing Rd format

There are several commands to process Rd files from the system command line. All of these need Perl to be 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 version 3 or 4 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 (or bundle). In the latter case, a reference manual for all documented objects in the package is created, including the information in the '`DESCRIPTION`' files.

`R CMD Sd2Rd` converts S version 3 documentation files (which use an extended Nroff format) and S version 4 documentation (which uses SGML markup) to Rd format. This is useful when porting a package originally written for the S system to R. S version 3 files usually have extension '`.d`', whereas version 4 ones have extension '`.sgml`' or '`.sgm`'.

`R CMD Sweave` and `R CMD Stangle` process '`Sweave`' documentation files (usually with extension '`.Snw`' or '`.Rnw`'): `R CMD Stangle` is use to extract the R code fragments.

The exact usage and a detailed list of available options for all but the last two 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`* (or *`Rcmd --help`* under Windows).

All of these work under Windows. You will need to have installed the files in the R binary Windows distribution for installing source packages (this is true for a default installation), and for `R CMD Rd2dvi` also the tools to build packages from source as described in the "R Installation and Administration" manual.

3 Tidying and profiling R code

R code which is worth preserving in a package and perhaps making available for others to use is worth documenting, tidying up and perhaps optimizing. The last two of these activities are the subject of this chapter.

3.1 Tidying R code

R treats function code loaded from packages and code entered by users differently. Code entered by users has the source code stored in an attribute, and when the function is listed, the original source is reproduced. Loading code from a package (by default) discards the source code, and the function listing is re-created from the parse tree of the function.

Normally keeping the source code is a good idea, and in particular it avoids comments being moved around in the source. However, we can make use of the ability to re-create a function listing from its parse tree to produce a tidy version of the function, for example with consistent indentation and spaces around operators. This tidied version is much easier to read, not least by other users who are used to the standard format. Although the deparsing cannot do so, we recommend the consistent use of the preferred assignment operator '`<-`' (rather than '`=`') for assignment.

We can subvert the keeping of source in two ways.

1. The option `keep.source` can be set to `FALSE` before the code is loaded into R.
2. The stored source code can be removed by removing the `source` attribute, for example by

   ```
   attr(myfun, "source") <- NULL
   ```

In each case if we then list the function we will get the standard layout.

Suppose we have a file of functions '`myfuns.R`' that we want to tidy up. Create a file '`tidy.R`' containing

```
options(keep.source = FALSE)
source("myfuns.R")
dump(ls(all = TRUE), file = "new.myfuns.R")
```

and run R with this as the source file, for example by `R --vanilla < tidy.R` or by pasting into an R session. Then the file '`new.myfuns.R`' will contain the functions in alphabetical order in the standard layout. You may need to move comments to more appropriate places.

The standard format provides a good starting point for further tidying. Most package authors use a version of Emacs (on Unix or Windows) to edit R code, using the ESS[S] mode of the ESS Emacs package. See section "R coding standards" in *R Internals* for style options within the ESS[S] mode recommended for the source code of R itself.

3.2 Profiling R code for speed

It is possible to profile R code on Windows and most[1] Unix-like versions of R.

The command `Rprof` is used to control profiling, and its help page can be consulted for full details. Profiling works by recording at fixed intervals[2] (by default every 20 msecs) which R function is being used, and recording the results in a file (default '`Rprof.out`' in the working directory). Then the function `summaryRprof` or the command-line utility `R CMD Rprof` *`Rprof.out`* can be used to summarize the activity.

As an example, consider the following code (from Venables & Ripley, 2002).

[1] R has to be built to enable this, but the option '`--enable-R-profiling`' is the default.

[2] For Unix-alikes these are intervals of CPU time, and for Windows of elapsed time.

```
library(MASS); library(boot)
storm.fm <- nls(Time ~ b*Viscosity/(Wt - c), stormer,
                start = c(b=29.401, c=2.2183))
st <- cbind(stormer, fit=fitted(storm.fm))
storm.bf <- function(rs, i) {
    st$Time <-  st$fit + rs[i]
    tmp <- nls(Time ~ (b * Viscosity)/(Wt - c), st,
               start = coef(storm.fm))
    tmp$m$getAllPars()
}
rs <- scale(resid(storm.fm), scale = FALSE) # remove the mean
Rprof("boot.out")
storm.boot <- boot(rs, storm.bf, R = 4999) # pretty slow
Rprof(NULL)
```

Having run this we can summarize the results by

```
R CMD Rprof boot.out

Each sample represents 0.02 seconds.
Total run time: 80.74 seconds.

Total seconds: time spent in function and callees.
Self seconds: time spent in function alone.

   %       total       %       self
 total    seconds     self    seconds    name
100.00     80.74      0.22      0.18     "boot"
 99.65     80.46      1.19      0.96     "statistic"
 96.33     77.78      2.68      2.16     "nls"
 50.21     40.54      1.54      1.24     "<Anonymous>"
 47.11     38.04      1.83      1.48     ".Call"
 23.06     18.62      2.43      1.96     "eval"
 19.87     16.04      0.67      0.54     "as.list"
 18.97     15.32      0.64      0.52     "switch"
 17.88     14.44      0.47      0.38     "model.frame"
 17.41     14.06      1.73      1.40     "model.frame.default"
 17.41     14.06      2.80      2.26     "nlsModel"
 15.43     12.46      1.88      1.52     "qr.qty"
 13.40     10.82      3.07      2.48     "assign"
 12.73     10.28      2.33      1.88     "storage.mode<-"
 12.34      9.96      1.81      1.46     "qr.coef"
 10.13      8.18      5.42      4.38     "paste"
  ...

   %       self        %       total
 self     seconds    total    seconds    name
  5.42      4.38     10.13      8.18     "paste"
  3.37      2.72      6.71      5.42     "as.integer"
  3.29      2.66      5.00      4.04     "as.double"
  3.20      2.58      4.29      3.46     "seq.default"
  3.07      2.48     13.40     10.82     "assign"
  2.92      2.36      5.95      4.80     "names"
  2.80      2.26     17.41     14.06     "nlsModel"
  2.68      2.16     96.33     77.78     "nls"
  2.53      2.04      2.53      2.04     ".Fortran"
  2.43      1.96     23.06     18.62     "eval"
  2.33      1.88     12.73     10.28     "storage.mode<-"
  ...
```

This often produces surprising results and can be used to identify bottlenecks or pieces of R code that could benefit from being replaced by compiled code.

`R CMD Rprof` uses a Perl script that may be a little faster than `summaryRprof` for large files. On the other hand `summaryRprof` does not require Perl and provides the results as an R object.

Two warnings: profiling does impose a small performance penalty, and the output files can be very large if long runs are profiled.

Profiling short runs can sometimes give misleading results. R from time to time performs *garbage collection* to reclaim unused memory, and this takes an appreciable amount of time which profiling will charge to whichever function happens to provoke it. It may be useful to compare profiling code immediately after a call to `gc()` with a profiling run without a preceding call to `gc`.

More detailed analysis of the output can be achieved by the tools in the CRAN package **proftools**: in particular this allows call graphs to be studied.

3.3 Profiling R code for memory use

Measuring memory use in R code is useful either when the code takes more memory than is conveniently available or when memory allocation and copying of objects is responsible for slow code. There are three ways to profile memory use over time in R code. All three require R to have been compiled with '`--enable-memory-profiling`', which is not the default. All can be misleading, for different reasons.

In understanding the memory profiles it is useful to know a little more about R's memory allocation. Looking at the results of `gc()` shows a division of memory into `Vcells` used to store the contents of vectors and `Ncells` used to store everything else, including all the administrative overhead for vectors such as type and length information. In fact the vector contents are divided into two pools. Memory for small vectors (by default 128 bytes or less) is obtained in large chunks and then parcelled out by R; memory for larger vectors is obtained directly from the operating system.

Some memory allocation is obvious in interpreted code, for example,

```
y <- x + 1
```

allocates memory for a new vector `y`. Other memory allocation is less obvious and occurs because `R` is forced to make good on its promise of 'call-by-value' argument passing. When an argument is passed to a function it is not immediately copied. Copying occurs (if necessary) only when the argument is modified. This can lead to surprising memory use. For example, in the 'survey' package we have

```
print.svycoxph <- function (x, ...)
{
    print(x$survey.design, varnames = FALSE, design.summaries = FALSE,
        ...)
    x$call <- x$printcall
    NextMethod()
}
```

It may not be obvious that the assignment to `x$call` will cause the entire object `x` to be copied. This copying to preserve the call-by-value illusion is usually done by the internal C function `duplicate`.

The main reason that memory-use profiling is difficult is garbage collection. Memory is allocated at well-defined times in an R program, but is freed whenever the garbage collector happens to run.

3.3.1 Memory statistics from `Rprof`

The sampling profiler `Rprof` described in the previous section can be given the option `memory.profiling=TRUE`. It then writes the total R memory allocation in small vectors, large vectors, and cons cells or nodes at each sampling interval. It also writes out the number of calls to the internal function `duplicate`, which is called to copy R objects. `summaryRprof` provides summaries of this information. The main reason that this can be misleading is that the memory use is attributed to the function running at the end of the sampling interval. A second reason is that garbage collection can make the amount of memory in use decrease, so a function appears to use little memory. Running under `gctorture` helps with both problems: it slows down the code to effectively increase the sampling frequency and it makes each garbage collection release a smaller amount of memory. Changing the memory limits with `mem.limits()` may also be useful, to see how the code would run under different memory conditions.

3.3.2 Tracking memory allocations

The second method of memory profiling uses a memory-allocation profiler, `Rprofmem()`, which writes out a stack trace to an output file every time a large vector is allocated (with a user-specified threshold for 'large') or a new page of memory is allocated for the R heap. Summary functions for this output are still being designed.

Running the example from the previous section with

```
> Rprofmem("boot.memprof",threshold=1000)
> storm.boot <- boot(rs, storm.bf, R = 4999)
> Rprofmem(NULL)
```

shows that apart from some initial and final work in `boot` there are no vector allocations over 1000 bytes.

3.3.3 Tracing copies of an object

The third method of memory profiling involves tracing copies made of a specific (presumably large) R object. Calling `tracemem` on an object marks it so that a message is printed to standard output when the object is copied via `duplicate` or coercion to another type, or when a new object of the same size is created in arithmetic operations. The main reason that this can be misleading is that copying of subsets or components of an object is not tracked. It may be helpful to use `tracemem` on these components.

In the example above we can run `tracemem` on the data frame `st`

```
> tracemem(st)
[1] "<0x9abd5e0>"
> storm.boot <- boot(rs, storm.bf, R = 4)
memtrace[0x9abd5e0->0x92a6d08]: statistic boot
memtrace[0x92a6d08->0x92a6d80]: $<-.data.frame $<- statistic boot
memtrace[0x92a6d80->0x92a6df8]: $<-.data.frame $<- statistic boot
memtrace[0x9abd5e0->0x9271318]: statistic boot
memtrace[0x9271318->0x9271390]: $<-.data.frame $<- statistic boot
memtrace[0x9271390->0x9271408]: $<-.data.frame $<- statistic boot
memtrace[0x9abd5e0->0x914f558]: statistic boot
memtrace[0x914f558->0x914f5f8]: $<-.data.frame $<- statistic boot
memtrace[0x914f5f8->0x914f670]: $<-.data.frame $<- statistic boot
memtrace[0x9abd5e0->0x972cbf0]: statistic boot
memtrace[0x972cbf0->0x972cc68]: $<-.data.frame $<- statistic boot
memtrace[0x972cc68->0x972cd08]: $<-.data.frame $<- statistic boot
memtrace[0x9abd5e0->0x98ead98]: statistic boot
memtrace[0x98ead98->0x98eae10]: $<-.data.frame $<- statistic boot
memtrace[0x98eae10->0x98eae88]: $<-.data.frame $<- statistic boot
```

The object is duplicated fifteen times, three times for each of the `R+1` calls to `storm.bf`. This is surprising, since none of the duplications happen inside `nls`. Stepping through `storm.bf` in the debugger shows that all three happen in the line

```
st$Time <- st$fit + rs[i]
```

Data frames are slower than matrices and this is an example of why. Using `tracemem(st$Viscosity)` does not reveal any additional copying.

3.4 Profiling compiled code

Profiling compiled code is highly system-specific, but this section contains some hints gleaned from various R users. Some methods need to be different for a compiled executable and for dynamic/shared libraries/objects as used by R packages. We know of no good way to profile DLLs on Windows.

3.4.1 Linux

Options include using `sprof` for a shared object, and `oprofile` (see http://oprofile.sourceforge.net/) for any executable or shared object.

3.4.1.1 sprof

You can select shared objects to be profiled with `sprof` by setting the environment variable `LD_PROFILE`. For example

```
% setenv LD_PROFILE /path/to/R_HOME/library/stats/libs/stats.so
R
... run the boot example
% sprof /path/to/R_HOME/library/stats/libs/stats.so \
  /var/tmp/path/to/R_HOME/library/stats/libs/stats.so.profile

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 76.19      0.32     0.32        0     0.00           numeric_deriv
 16.67      0.39     0.07        0     0.00           nls_iter
  7.14      0.42     0.03        0     0.00           getListElement

rm /path/to/R_HOME/library/stats/libs/stats.so.profile
... to clean up ...
```

It is possible that root access is needed to create the directories used for the profile data.

3.4.1.2 oprofile

`oprofile` works by running a daemon which collects information. The daemon must be started as root, e.g.

```
% su
% opcontrol --no-vmlinux
% opcontrol --start
% exit
```

Then as a user

```
% R
... run the boot example
% opcontrol --dump
% opreport -l /path/to/R_HOME/library/stats/libs/stats.so
...
samples  %        symbol name
1623     75.5939  anonymous symbol from section .plt
349      16.2552  numeric_deriv
113       5.2632  nls_iter
62        2.8878  getListElement
% opreport -l /path/to/R_HOME/bin/exec/R
...
samples  %        symbol name
```

```
76052    11.9912  Rf_eval
54670     8.6198  Rf_findVarInFrame3
37814     5.9622  Rf_allocVector
31489     4.9649  Rf_duplicate
28221     4.4496  Rf_protect
26485     4.1759  Rf_cons
23650     3.7289  Rf_matchArgs
21088     3.3250  Rf_findFun
19995     3.1526  findVarLocInFrame
14871     2.3447  Rf_evalList
13794     2.1749  R_Newhashpjw
13522     2.1320  R_gc_internal
...
```

Shutting down the profiler and clearing the records needs to be done as root. You can use `opannotate` to annotate the source code with the times spent in each section, if the appropriate source code was compiled with debugging support.

3.4.2 Solaris

On 64-bit (only) Solaris, the standard profiling tool `gprof` collects information from shared libraries compiled with '`-pg`'.

3.4.3 MacOS X

Developers have recommended `sample` (or `Sampler.app`, which is a GUI version) and `Shark` (see http://developer.apple.com/tools/sharkoptimize.html and http://developer.apple.com/tools/shark_optimize.html).

4 Debugging

This chapter covers the debugging of R extensions, starting with the ways to get useful error information and moving on to how to deal with errors that crash R. For those who prefer other styles there are contributed packages such as **debug** on CRAN (described in an article in R-News 3/3). (There are notes from 2002 provided by Roger Peng at `http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf` which provide complementary examples to those given here.)

4.1 Browsing

Most of the R-level debugging facilities are based around the built-in browser. This can be used directly by inserting a call to `browser()` into the code of a function (for example, using `fix(my_function)`). When code execution reaches that point in the function, control returns to the R console with a special prompt. For example

```
> fix(summary.data.frame) ## insert browser() call after for() loop
> summary(women)
Called from: summary.data.frame(women)
Browse[1]> ls()
 [1] "digits" "i"      "lbs"    "lw"     "maxsum" "nm"     "nr"     "nv"
 [9] "object" "sms"    "z"
Browse[1]> maxsum
[1] 7
Browse[1]>
     height         weight
 Min.   :58.0   Min.   :115.0
 1st Qu.:61.5   1st Qu.:124.5
 Median :65.0   Median :135.0
 Mean   :65.0   Mean   :136.7
 3rd Qu.:68.5   3rd Qu.:148.0
 Max.   :72.0   Max.   :164.0
> rm(summary.data.frame)
```

At the browser prompt one can enter any R expression, so for example `ls()` lists the objects in the current frame, and entering the name of an object will[1] print it. The following commands are also accepted

- `n`

  Enter 'step-through' mode. In this mode, hitting return executes the next line of code (more precisely one line and any continuation lines). Typing `c` will continue to the end of the current context, e.g. to the end of the current loop or function.

- `c`

  In normal mode, this quits the browser and continues execution, and just return works in the same way. `cont` is a synonym.

- `where`

  This prints the call stack. For example

  ```
  > summary(women)
  Called from: summary.data.frame(women)
  Browse[1]> where
  where 1: summary.data.frame(women)
  ```

[1] With the exceptions of the commands listed below: an object of such a name can be printed *via* an explicit call to `print`.

```
where 2: summary(women)

Browse[1]>
```

- `Q`

  Quit both the browser and the current expression, and return to the top-level prompt.

Errors in code executed at the browser prompt will normally return control to the browser prompt. Objects can be altered by assignment, and will keep their changed values when the browser is exited. If really necessary, objects can be assigned to the workspace from the browser prompt (by using `<<-` if the name is not already in scope).

4.2 Debugging R code

Suppose your R program gives an error message. The first thing to find out is what R was doing at the time of the error, and the most useful tool is `traceback()`. We suggest that this is run whenever the cause of the error is not immediately obvious. Daily, errors are reported to the R mailing lists as being in some package when `traceback()` would show that the error was being reported by some other package or base R. Here is an example from the regression suite.

```
> success <- c(13,12,11,14,14,11,13,11,12)
> failure <- c(0,0,0,0,0,0,0,2,2)
> resp <- cbind(success, failure)
> predictor <- c(0, 5^(0:7))
> glm(resp ~ 0+predictor, family = binomial(link="log"))
Error: no valid set of coefficients has been found: please supply starting values
> traceback()
3: stop("no valid set of coefficients has been found: please supply
         starting values", call. = FALSE)
2: glm.fit(x = X, y = Y, weights = weights, start = start, etastart = etastart,
       mustart = mustart, offset = offset, family = family, control = control,
       intercept = attr(mt, "intercept") > 0)
1: glm(resp ~ 0 + predictor, family = binomial(link ="log"))
```

The calls to the active frames are given in reverse order (starting with the innermost). So we see the error message comes from an explicit check in `glm.fit`. (`traceback()` shows you all the lines of the function calls, which can be limited by setting `option` '`"deparse.max.lines"`'.)

Sometimes the traceback will indicate that the error was detected inside compiled code, for example (from `?nls`)

```
Error in nls(y ~ a + b * x, start = list(a = 0.12345, b = 0.54321), trace = TRUE) :
        step factor 0.000488281 reduced below 'minFactor' of 0.000976563
>  traceback()
2: .Call(R_nls_iter, m, ctrl, trace)
1: nls(y ~ a + b * x, start = list(a = 0.12345, b = 0.54321), trace = TRUE)
```

This will be the case if the innermost call is to `.C`, `.Fortran`, `.Call`, `.External` or `.Internal`, but as it is also possible for such code to evaluate R expressions, this need not be the innermost call, as in

```
> traceback()
9: gm(a, b, x)
8: .Call(R_numeric_deriv, expr, theta, rho, dir)
7: numericDeriv(form[[3]], names(ind), env)
6: getRHS()
5: assign("rhs", getRHS(), envir = thisEnv)
4: assign("resid", .swts * (lhs - assign("rhs", getRHS(), envir = thisEnv)),
       envir = thisEnv)
3: function (newPars)
   {
       setPars(newPars)
       assign("resid", .swts * (lhs - assign("rhs", getRHS(), envir = thisEnv)),
           envir = thisEnv)
```

```
        assign("dev", sum(resid^2), envir = thisEnv)
        assign("QR", qr(.swts * attr(rhs, "gradient")), envir = thisEnv)
        return(QR$rank < min(dim(QR$qr)))
    }(c(-0.00760232418963883, 1.00119632515036))
2: .Call(R_nls_iter, m, ctrl, trace)
1: nls(yeps ~ gm(a, b, x), start = list(a = 0.12345, b = 0.54321))
```

Occasionally `traceback()` does not help, and this can be the case if S4 method dispatch is involved. Consider the following example

```
> xyd <- new("xyloc", x=runif(20), y=runif(20))
Error in as.environment(pkg) : no item called "package:S4nswv"
on the search list
Error in initialize(value, ...) : S language method selection got
an error when called from internal dispatch for function 'initialize'
> traceback()
2: initialize(value, ...)
1: new("xyloc", x = runif(20), y = runif(20))
```

which does not help much, as there is no call to `as.environment` in `initialize` (and the note "called from internal dispatch" tells us so). In this case we searched the R sources for the quoted call, which occurred in only one place, `methods:::.asEnvironmentPackage`. So now we knew where the error was occurring. (This was an unusually opaque example.)

The error message

```
evaluation nested too deeply: infinite recursion / options(expressions=)?
```

can be hard to handle with the default value (5000). Unless you know that there actually is deep recursion going on, it can help to set something like

```
options(expressions=500)
```

and re-run the example showing the error.

Sometimes there is warning that clearly is the precursor to some later error, but it is not obvious where it is coming from. Setting `options(warn = 2)` (which turns warnings into errors) can help here.

Once we have located the error, we have some choices. One way to proceed is to find out more about what was happening at the time of the crash by looking a *post-mortem* dump. To do so, set `options(error=dump.frames)` and run the code again. Then invoke `debugger()` and explore the dump. Continuing our example:

```
> options(error = dump.frames)
> glm(resp ~ 0 + predictor, family = binomial(link ="log"))
Error: no valid set of coefficients has been found: please supply starting values
```

which is the same as before, but an object called `last.dump` has appeared in the workspace. (Such objects can be large, so remove it when it is no longer needed.) We can examine this at a later time by calling the function `debugger`.

```
> debugger()
Message:  Error: no valid set of coefficients has been found: please supply starting values
Available environments had calls:
1: glm(resp ~ 0 + predictor, family = binomial(link = "log"))
2: glm.fit(x = X, y = Y, weights = weights, start = start, etastart = etastart, mus
3: stop("no valid set of coefficients has been found: please supply starting values
Enter an environment number, or 0 to exit  Selection:
```

which gives the same sequence of calls as `traceback`, but in outer-first order and with only the first line of the call, truncated to the current width. However, we can now examine in more detail what was happening at the time of the error. Selecting an environment opens the browser in that frame. So we select the function call which spawned the error message, and explore some of the variables (and execute two function calls).

```
Enter an environment number, or 0 to exit  Selection: 2
Browsing in the environment with call:
   glm.fit(x = X, y = Y, weights = weights, start = start, etas
Called from: debugger.look(ind)
Browse[1]> ls()
 [1] "aic"        "boundary"   "coefold"    "control"    "conv"
 [6] "dev"        "dev.resids" "devold"     "EMPTY"      "eta"
[11] "etastart"   "family"     "fit"        "good"       "intercept"
[16] "iter"       "linkinv"    "mu"         "mu.eta"     "mu.eta.val"
[21] "mustart"    "n"          "ngoodobs"   "nobs"       "nvars"
[26] "offset"     "start"      "valideta"   "validmu"    "variance"
[31] "varmu"      "w"          "weights"    "x"          "xnames"
[36] "y"          "ynames"     "z"
Browse[1]> eta
            1             2             3             4             5
 0.000000e+00 -2.235357e-06 -1.117679e-05 -5.588393e-05 -2.794197e-04
            6             7             8             9
-1.397098e-03 -6.985492e-03 -3.492746e-02 -1.746373e-01
Browse[1]> valideta(eta)
[1] TRUE
Browse[1]> mu
        1         2         3         4         5         6         7         8
1.0000000 0.9999978 0.9999888 0.9999441 0.9997206 0.9986039 0.9930389 0.9656755
        9
0.8397616
Browse[1]> validmu(mu)
[1] FALSE
Browse[1]> c
Available environments had calls:
1: glm(resp ~ 0 + predictor, family = binomial(link = "log"))
2: glm.fit(x = X, y = Y, weights = weights, start = start, etastart = etastart
3: stop("no valid set of coefficients has been found: please supply starting v

Enter an environment number, or 0 to exit  Selection: 0
> rm(last.dump)
```

Because `last.dump` can be looked at later or even in another R session, post-mortem debugging is possible even for batch usage of R. We do need to arrange for the dump to be saved: this can be done either using the command-line flag '`--save`' to save the workspace at the end of the run, or via a setting such as

```
> options(error = quote({dump.frames(to.file=TRUE); q()}))
```

See the help on `dump.frames` for further options and a worked example.

An alternative error action is to use the function `recover()`:

```
> options(error = recover)
> glm(resp ~ 0 + predictor, family = binomial(link = "log"))
Error: no valid set of coefficients has been found: please supply starting values

Enter a frame number, or 0 to exit

1: glm(resp ~ 0 + predictor, family = binomial(link = "log"))
2: glm.fit(x = X, y = Y, weights = weights, start = start, etastart = etastart

Selection:
```

which is very similar to `dump.frames`. However, we can examine the state of the program directly, without dumping and re-loading the dump. As its help page says, `recover` can be routinely used as the error action in place of `dump.calls` and `dump.frames`, since it behaves like `dump.frames` in non-interactive use.

Post-mortem debugging is good for finding out exactly what went wrong, but not necessarily why. An alternative approach is to take a closer look at what was happening just before the error, and a good way to do that is to use `debug`. This inserts a call to the browser at the beginning of the function, starting in step-through mode. So in our example we could use

```
> debug(glm.fit)
> glm(resp ~ 0 + predictor, family = binomial(link ="log"))
debugging in: glm.fit(x = X, y = Y, weights = weights, start = start, etastart = etastart,
    mustart = mustart, offset = offset, family = family, control = control,
    intercept = attr(mt, "intercept") > 0)
debug: {
## lists the whole function
Browse[1]>
debug: x <- as.matrix(x)
...
Browse[1]> start
[1] -2.235357e-06
debug: eta <- drop(x %*% start)
Browse[1]> eta
            1             2             3             4             5
 0.000000e+00 -2.235357e-06 -1.117679e-05 -5.588393e-05 -2.794197e-04
            6             7             8             9
-1.397098e-03 -6.985492e-03 -3.492746e-02 -1.746373e-01
Browse[1]>
debug: mu <- linkinv(eta <- eta + offset)
Browse[1]> mu
        1         2         3         4         5         6         7         8
1.0000000 0.9999978 0.9999888 0.9999441 0.9997206 0.9986039 0.9930389 0.9656755
        9
0.8397616
```

(The prompt `Browse[1]>` indicates that this is the first level of browsing: it is possible to step into another function that is itself being debugged or contains a call to `browser()`.)

`debug` can be used for hidden functions and S3 methods by e.g. `debug(stats:::predict.Arima)`. (It cannot be used for S4 methods, but an alternative is given on the help page for `debug`.) Sometimes you want to debug a function defined inside another function, e.g. the function `arimafn` defined inside `arima`. To do so, set `debug` on the outer function (here `arima`) and step through it until the inner function has been defined. Then call `debug` on the inner function (and use `c` to get out of step-through mode in the outer function).

To remove debugging of a function, call `undebug` with the argument previously given to `debug`; debugging otherwise lasts for the rest of the R session (or until the function is edited or otherwise replaced).

`trace` can be used to temporarily insert debugging code into a function, for example to insert a call to `browser()` just before the point of the error. To return to our running example

```
## first get a numbered listing of the expressions of the function
> page(as.list(body(glm.fit)), method="print")
> trace(glm.fit, browser, at=22)
Tracing function "glm.fit" in package "stats"
[1] "glm.fit"
> glm(resp ~ 0 + predictor, family = binomial(link ="log"))
Tracing glm.fit(x = X, y = Y, weights = weights, start = start,
   etastart = etastart,  .... step 22
Called from: eval(expr, envir, enclos)
Browse[1]> n
## and single-step from here.
> untrace(glm.fit)
```

For your own functions, it may be as easy to use `fix` to insert temporary code, but `trace` can help with functions in a name space (as can `fixInNamespace`). Alternatively, use `trace(,edit=TRUE)` to insert code visually.

4.3 Using gctorture and valgrind

Errors in memory allocation and reading/writing outside arrays are very common causes of crashes (e.g., segfaults) on some machines. Often the crash appears long after the invalid memory access: in particular damage to the structures which R itself has allocated may only become apparent at the next garbage collection (or even at later garbage collections after objects have been deleted).

4.3.1 Using gctorture

We can help to detect memory problems earlier by running garbage collection as often as possible. This is achieved by `gctorture(TRUE)`, which as described on its help page

> Provokes garbage collection on (nearly) every memory allocation. Intended to ferret out memory protection bugs. Also makes R run *very* slowly, unfortunately.

The reference to 'memory protection' is to missing C-level calls to `PROTECT`/`UNPROTECT` (see Section 5.8.1 [Garbage Collection], page 69) which if missing allow R objects to be garbage-collected when they are still in use. But it can also help with other memory-related errors.

Normally running under `gctorture(TRUE)` will just produce a crash earlier in the R program, hopefully close to the actual cause. See the next section for how to decipher such crashes.

It is possible to run all the examples, tests and vignettes covered by `R CMD check` under `gctorture(TRUE)` by using the option '`--use-gct`'.

4.3.2 Using valgrind

If you have access to Linux on an `ix86`, `x86_64` or `ppc32` platform you can use `valgrind` (`http://www.valgrind.org/`, pronounced to rhyme with 'tinned') to check for possible problems. To run some examples under `valgrind` use something like

```
R -d valgrind --vanilla < mypkg-Ex.R
R -d "valgrind --tool=memcheck --leak-check=full" --vanilla < mypkg-Ex.R
```

where '`mypkg-Ex.R`' is a set of examples, e.g. the file created in '`mypkg.Rcheck`' by `R CMD check`. Occasionally this reports memory reads of 'uninitialised values' that are the result of compiler optimization, so can be worth checking under an unoptimized compile. We know there will be some small memory leaks from `readline` and R itself — these are memory areas that are in use right up to the end of the R session. Expect this to run around 20x slower than without `valgrind`, and in some cases even slower than that. Current versions[2] of `valgrind` are not happy with many optimized BLASes that use cpu-specific instructions (3D now, SSE, SSE2, SSE3 and similar) so you may need to build a version of R specifically to use with `valgrind`.

On platforms supported by `valgrind` you can build a version of R with extra instrumentation to help `valgrind` detect errors in the use of memory allocated from the R heap. The configure option is '`--with-valgrind-instrumentation=`*level*', where *level* is 0, 1, or 2. Level 0 is the default and does not add any anything. Level 1 will detect use of uninitialised memory and has little impact on speed. Level 2 will detect many other memory use bugs but makes R much slower when running under `valgrind`. Using this in conjuction with `gctorture` can be even more effective (and even slower).

An example of `valgrind` output is

```
==12539== Invalid read of size 4
==12539==    at 0x1CDF6CBE: csc_compTr (Mutils.c:273)
==12539==    by 0x1CE07E1E: tsc_transpose (dtCMatrix.c:25)
==12539==    by 0x80A67A7: do_dotcall (dotcode.c:858)
==12539==    by 0x80CACE2: Rf_eval (eval.c:400)
==12539==    by 0x80CB5AF: R_execClosure (eval.c:658)
```

[2] Although this is supposed to have been improved, `valgrind` 3.2.0 still aborts using optimized BLASes on an Opteron.

```
==12539==    by 0x80CB98E: R_execMethod (eval.c:760)
==12539==    by 0x1B93DEFA: R_standardGeneric (methods_list_dispatch.c:624)
==12539==    by 0x810262E: do_standardGeneric (objects.c:1012)
==12539==    by 0x80CAD23: Rf_eval (eval.c:403)
==12539==    by 0x80CB2F0: Rf_applyClosure (eval.c:573)
==12539==    by 0x80CADCC: Rf_eval (eval.c:414)
==12539==    by 0x80CAA03: Rf_eval (eval.c:362)
==12539==  Address 0x1C0D2EA8 is 280 bytes inside a block of size 1996 alloc'd
==12539==    at 0x1B9008D1: malloc (vg_replace_malloc.c:149)
==12539==    by 0x80F1B34: GetNewPage (memory.c:610)
==12539==    by 0x80F7515: Rf_allocVector (memory.c:1915)
...
```

This example is from an instrumented version of R, while tracking down a bug in the **Matrix** package in January, 2006. The first line indicates that R has tried to read 4 bytes from a memory address that it does not have access to. This is followed by a C stack trace showing where the error occurred. Next is a description of the memory that was accessed. It is inside a block allocated by `malloc`, called from `GetNewPage`, that is, in the internal R heap. Since this memory all belongs to R, `valgrind` would not (and did not) detect the problem in an uninstrumented build of R. In this example the stack trace was enough to isolate and fix the bug, which was in `tsc_transpose`, and in this example running under `gctorture()` did not provide any additional information. When the stack trace is not sufficiently informative the option '`--db-attach=yes`' to `valgrind` may be helpful. This starts a post-mortem debugger (by default `gdb`) so that variables in the C code can be inspected (see Section 4.4.2 [Inspecting R objects], page 57).

It is possible to run all the examples, tests and vignettes covered by `R CMD check` under `valgrind` by using the option '`--use-valgrind`'. If you do this you will need to select the `valgrind` options some other way, for example by having a '`~/.valgrindrc`' file containing

```
--tool=memcheck
--memcheck:leak-check=full
```

or setting the environment variable `VALGRIND_OPTS`.

4.4 Debugging compiled code

Sooner or later programmers will be faced with the need to debug compiled code loaded into R. This section is geared to platforms using `gdb` with code compiled by `gcc`, but similar things are possible with front-ends to `gdb` such as `ddd` and `insight`, and other debuggers such as Sun's `dbx`.

Consider first 'crashes', that is when R terminated unexpectedly with an illegal memory access (a 'segfault' or 'bus error'), illegal instruction or similar. Unix-alike versions of R use a signal handler which aims to give some basic information. For example

```
 *** caught segfault ***
address 0x20000028, cause 'memory not mapped'

Traceback:
 1: .identC(class1[[1]], class2)
 2: possibleExtends(class(sloti), classi, ClassDef2 = getClassDef(classi,
where = where))
 3: validObject(t(cu))
 4: stopifnot(validObject(cu <- as(tu, "dtCMatrix")), validObject(t(cu)),
validObject(t(tu)))

Possible actions:
1: abort (with core dump)
```

```
2: normal R exit
3: exit R without saving workspace
4: exit R saving workspace
Selection: 3
```

Since the R process may be damaged, the only really safe option is the first.

Another cause of a 'crash' is to overrun the C stack. R tries to track that in its own code, but it may happen in third-party compiled code. For modern POSIX-compilant OSes we can safely catch that and return to the top-level prompt.

```
> .C("aaa")
Error: segfault from C stack overflow
>
```

However, C stack overflows are fatal under Windows and normally defeat attempts at debugging on that platform.

If you have a crash which gives a core dump you can use something like

```
gdb /path/to/R/bin/exec/R core.12345
```

to examine the core dump. If core dumps are disabled or to catch errors that do not generate a dump one can run R directly under a debugger by for example

```
$ R -d gdb --vanilla
...
gdb> run
```

at which point R will run normally, and hopefully the debugger will catch the error and return to its prompt. This can also be used to catch infinite loops or interrupt very long-running code. For a simple example

```
> for(i in 1:1e7) x <- rnorm(100)
[hit Ctrl-C]
Program received signal SIGINT, Interrupt.
0x00397682 in _int_free () from /lib/tls/libc.so.6
(gdb) where
#0  0x00397682 in _int_free () from /lib/tls/libc.so.6
#1  0x00397eba in free () from /lib/tls/libc.so.6
#2  0xb7cf2551 in R_gc_internal (size_needed=313)
    at /users/ripley/R/svn/R-devel/src/main/memory.c:743
#3  0xb7cf3617 in Rf_allocVector (type=13, length=626)
    at /users/ripley/R/svn/R-devel/src/main/memory.c:1906
#4  0xb7c3f6d3 in PutRNGstate ()
    at /users/ripley/R/svn/R-devel/src/main/RNG.c:351
#5  0xb7d6c0a5 in do_random2 (call=0x94bf7d4, op=0x92580e8, args=0x9698f98,
    rho=0x9698f28) at /users/ripley/R/svn/R-devel/src/main/random.c:183
...
```

Some "tricks" are worth knowing.

4.4.1 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-alikes 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 shared object.

- 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 signals may not be able to be used, and if so the procedure is more complicated. See the rw-FAQ and www.stats.uwo.ca/faculty/murdoch/software/debuggingR/gdb.shtml.

4.4.2 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 of `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
```

5 System and foreign language interfaces

5.1 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: these days only on the obsolete Windows 9x/ME).

5.2 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 5.3 [dyn.load and dyn.unload], page 61). They are primarily intended for compiled C and FORTRAN 77 code respectively, but the `.C` function can be used with other languages which can generate C interfaces, for example C++ (see Section 5.6 [Interfacing C++ code], page 65).

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. (That the symbol is loaded can be tested by, for example, `is.loaded("cg")`: it is no longer necessary nor correct to use `symbol.For`, which is defunct as from R 2.5.0.) (Note that the underscore is not a valid character in a FORTRAN 77 subprogram name, and on versions of R prior to 2.4.0 `.Fortran` may not correctly translate names containing underscores.)

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`
`raw`	`unsigned char *`	none

Do please note the first two. On the 64-bit Unix/Linux platforms, `long` is 64-bit whereas `int` and `INTEGER` are 32-bit. Code ported from S-PLUS (which uses `long *` for `logical` and `integer`) will not work on all 64-bit platforms (although it may appear to work on some). Note also that if your compiled code is a mixture of C functions and FORTRAN subprograms the argument types must match as given in the table above.

C type `Rcomplex` is a structure with `double` members `r` and `i` defined in the header file '`R_ext/Complex.h`' included by '`R.h`'. (On most platforms which have it, this is compatible withe C99 `double complex` type.) 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. Similarly, passing an R list as an argument to a C routine should be done using the .Call interface. If one does use the .C function to pass a list as an argument, it is visible to the routine as an array in C of SEXP types (i.e., SEXP *). The elements of the array correspond directly to the elements of the R list. However, this array must be treated as read-only and one must not assign values to its elements within the C routine — doing so bypasses R's memory management facilities and will corrupt the object and the R session.

It is possible to pass numeric vectors of storage mode double to C as float * or to FORTRAN as REAL by setting the attribute Csingle, most conveniently by using the R functions as.single, single or 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=FALSE, and character vectors cannot be used.

Argument PACKAGE confines the search for the symbol name to a specific shared object (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. (If it is not present and the call is from the body of a function defined in a package with a name space, the shared object loaded by the first (if any) useDynLib directive will be used.)

For .C only you can specify an ENCODING argument: this requests that (unless DUP = FALSE) character vectors be re-encoded to the requested encoding before being passed in, and re-encoded from the requested encoding when passed back. Note that encoding names are not standardized, and not all R builds support re-encoding. (The argument is ignored with a warning if re-encoding is not supported at all: R code can test for this *via* capabilities("iconv").) But this can be useful to allow code to work in a UTF-8 locale by specifying ENCODING = "latin1".

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.

Special care is needed in handling `character` vector arguments in C (or C++). Since only `DUP = TRUE` is allowed, on entry the contents of the elements are duplicated and assigned to the elements of a `char **` array, and on exit the elements of the C array are copied to create new elements of a character vector. This means that the contents of the character strings of the `char **` array can be changed, including to `\0` to shorten the string, but the strings cannot be lengthened. It is possible to allocate a new string *via* `R_alloc` and replace an entry in the `char **` array by the new string. However, when character vectors are used other than in a read-only way, the `.Call` interface is much to be preferred.

Passing character strings to FORTRAN code needs even more care, and should be avoided where possible. Only the first element of the character vector is passed in, as a fixed-length (255) character array. Up to 255 characters are passed back to a length-one character vector. How well this works (or even if it works at all) depends on the C and FORTRAN compilers on each platform.

5.3 `dyn.load` and `dyn.unload`

Compiled code to be used with R is loaded as a shared object (Unix and MacOS X, see Section 5.5 [Creating shared objects], page 64 for more information) or DLL (Windows).

The shared object/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 object. Programmers should not assume a specific file extension for the object/DLL (such as '`.so`') but use a construction like

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

for platform independence. On Unix-alike 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 object/DLL name *with the extension omitted*. Note that the first argument, `chname`, should **not** be `package` since this will not work if the package is installed under another name (as it will be with a versioned install).

Under some Unix-alike systems there is a choice of how the symbols are resolved when the object 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.

R provides a way of executing some code automatically when a object/DLL is either loaded or unloaded. This can be used, for example, to register native routines with R's dynamic symbol mechanism, initialize some data in the native code, or initialize a third party library. On loading a DLL, R will look for a routine within that DLL named `R_init_lib` where *lib* is the name of the DLL file with the extension removed. For example, in the command

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

R looks for the symbol named `R_init_mylib`. Similarly, when unloading the object, R looks for a routine named `R_unload_lib`, e.g., `R_unload_mylib`. In either case, if the routine is present, R will invoke it and pass it a single argument describing the DLL. This is a value of type `DllInfo` which is defined in the '`Rdynload.h`' file in the '`R_ext`' directory.

The following example shows templates for the initialization and unload routines for the `mylib` DLL.

```
#include <R.h>
#include <Rinternals.h>
#include <R_ext/Rdynload.h>

void
R_init_mylib(DllInfo *info)
{
  /* Register routines, allocate resources. */
}

void
R_unload_mylib(DllInfo *info)
{
  /* Release resources. */
}
```

If a shared object/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.

5.4 Registering native routines

By 'native' routine, we mean an entry point in compiled code.

In calls to `.C`, `.Call`, `.Fortran` and `.External`, R must locate the specified native routine by looking in the appropriate shared object/DLL. By default, R uses the operating system-specific dynamic loader to lookup the symbol. Alternatively, the author of the DLL can explicitly register routines with R and use a single, platform-independent mechanism for finding the routines in the DLL. One can use this registration mechanism to provide additional information about a routine, including the number and type of the arguments, and also make it available to R programmers under a different name. In the future, registration may be used to implement a form of "secure" or limited native access.

To register routines with R, one calls the C routine `R_registerRoutines`. This is typically done when the DLL is first loaded within the initialization routine `R_init_dll name` described in Section 5.3 [dyn.load and dyn.unload], page 61. `R_registerRoutines` takes 5 arguments. The first is the `DllInfo` object passed by R to the initialization routine. This is where R stores the information about the methods. The remaining 4 arguments are arrays describing the routines for each of the 4 different interfaces: `.C`, `.Call`, `.Fortran` and `.External`. Each argument is a `NULL`-terminated array of the element types given in the following table:

`.C`	`R_CMethodDef`
`.Call`	`R_CallMethodDef`

```
.Fortran     R_FortranMethodDef
.External    R_ExternalMethodDef
```

Currently, the `R_ExternalMethodDef` is the same as `R_CallMethodDef` type and contains fields for the name of the routine by which it can be accessed in R, a pointer to the actual native symbol (i.e., the routine itself), and the number of arguments the routine expects. For routines with a variable number of arguments invoked via the `.External` interface, one specifies `-1` for the number of arguments which tells R not to check the actual number passed. For example, if we had a routine named `myCall` defined as

```
SEXP myCall(SEXP a, SEXP b, SEXP c);
```

we would describe this as

```
R_CallMethodDef callMethods[]  = {
  {"myCall", &myCall, 3},
  {NULL, NULL, 0}
};
```

along with any other routines for the `.Call` interface.

Routines for use with the `.C` and `.Fortran` interfaces are described with similar data structures, but which have two additional fields for describing the type and “style” of each argument. Each of these can be omitted. However, if specified, each should be an array with the same number of elements as the number of parameters for the routine. The types array should contain the `SEXP` types describing the expected type of the argument. (Technically, the elements of the types array are of type `R_NativePrimitiveArgType` which is just an unsigned integer.) The R types and corresponding type identifiers are provided in the following table:

```
numeric      REALSXP
integer      INTSXP
logical      LGLSXP
single       SINGLESXP
character    STRSXP
list         VECSXP
```

Consider a C routine, `myC`, declared as

```
void myC(double *x, int *n, char **names, int *status);
```

We would register it as

```
R_CMethodDef cMethods[] = {
   {"myC", &myC, 4, {REALSXP, INTSXP, STRSXP, LGLSXP}},
   {NULL, NULL, 0}
};
```

One can also specify whether each argument is used simply as input, or as output, or as both input and output. The style field in the description of a method is used for this. The purpose is to allow R to transfer values more efficiently across the R-C/FORTRAN interface by avoiding copying values when it is not necessary. Typically, one omits this information in the registration data.

Having created the arrays describing each routine, the last step is to actually register them with R. We do this by calling `R_registerRoutines`. For example, if we have the descriptions above for the routines accessed by the `.C` and `.Call` we would use the following code:

```
void
R_init_myLib(DllInfo *info)
{
   R_registerRoutines(info, cMethods, callMethods, NULL, NULL);
}
```

This routine will be invoked when R loads the shared object/DLL named myLib. The last two arguments in the call to R_registerRoutines are for the routines accessed by .Fortran and .External interfaces. In our example, these are given as NULL since we have no routines of these types.

When R unloads a shared object/DLL, any registered routines are automatically removed. There is no (direct) facility for unregistering a symbol.

Examples of registering routines can be found in the different packages in the R source tree (e.g., **stats**). Also, there is a brief, high-level introduction in *R News* (volume 1/3, September 2001, pages 20-23).

In addition to registering C routines to be called by R, it can at times be useful for one package to make some of its C routines available to be called by C code in another package. An interface to support this has been provided since R 2.4.0. The interface consists of two routines declared as

```
void R_RegisterCCallable(const char *package, const char *name, DL_FUNC fptr);
DL_FUNC R_GetCCallable(const char *package, const char *name);
```

A package **packA** that wants to make a C routine myCfun available to C code in other packages would include the call

```
R_RegisterCCallable("packA", "myCfun", myCfun);
```

in its initialization function R_init_packA. A package **packB** that wants to use this routine would retrieve the function pointer with a call of the form

```
p_myCfun = R_GetCCallable("packA", "myCfun");
```

The author of **packB** is responsible for insuring that p_myCfun has an appropriate declaration. In the future R may provide some automated tools to simplify exporting larger numbers of routines.

A package that wishes to make use of header files in other packages needs to declare them as a comma-separated list in the field LinkingTo in the 'DESCRIPTION' file. For example

```
Depends: link2, link3
LinkingTo: link2, link3
```

It should also 'Depend' on those packages for they have to be installed prior to this one, and loaded prior to this one (so the path to their compiled code can be found).

This then arranges that the 'include' directories in the installed linked-to packages are added to the include paths for C and C++ code.

5.5 Creating shared objects

Shared objects 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 sources for C, C++, FORTRAN 77, Fortran 9x, Objective C or Objective C++ (with extensions '.c', '.cc' or '.cpp' or '.C', '.f', '.f90' or '.f95', '.m', and '.mm' or '.M', respectively), or commands to be passed to the linker. See *R CMD SHLIB --help* (or the R 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, PKG_CXXFLAGS, PKG_FFLAGS, PKG_FCFLAGS, and PKG_OBJCFLAGS (for the C, C++, FORTRAN 77, Fortran 9x, and Objective C compilers, respectively) in the file 'Makevars' in the compilation directory (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 object. (Supplying linker commands as arguments to R CMD SHLIB will override PKG_LIBS in 'Makevars'.)

It is possible to arrange to include compiled code from other languages by setting the macro 'OBJECTS' in file 'Makevars', together with suitable rules to make the objects.

Flags which are already set (for example in file 'etc*R_ARCH*/Makeconf' on Unix-alikes) can be overridden by the environment variable MAKEFLAGS (at least for systems using a POSIX-compliant make), as in (Bourne shell syntax)

```
MAKEFLAGS="CFLAGS=-O3" R CMD SHLIB *.c
```

It is also possible to set such variables in personal 'Makevars' files, which are read after the local 'Makevars' and the system makefiles. See ⟨undefined⟩ [R-admin], page ⟨undefined⟩, and also ⟨undefined⟩ [R-admin], page ⟨undefined⟩.

Note that as R CMD SHLIB uses Make, it will not remake a shared object just because the flags have changed, and if test.c and test.f both exist in the current directory

```
R CMD SHLIB test.f
```

will compile 'test.c'!

If the 'src' subdirectory of an add-on package contains source code with one of the extensions listed above or a file 'Makevars' but **not** a file Makefile, R CMD INSTALL creates a shared object (for loading into R in the .First.lib or .onLoad function of the package) using the R CMD SHLIB mechanism. If file 'Makevars' exists it is read first, then the system makefile and then any personal 'Makevars' files.

If the 'src' subdirectory of package contains a file 'Makefile', this is used in place of the R CMD SHLIB mechanism. make is called with makefiles 'R_HOME/etc*R_ARCH*/Makeconf'[1], 'src/Makefile' and any personal 'Makevars' files (in that order). The first target found in 'src/Makefile' is used.

It is better to make use of a Makevars file rather than a Makefile: the latter should be needed only exceptionally.

Note that whereas R CMD INSTALL makes use of a 'Makefile', R CMD SHLIB does not. The file must be named 'Makefile', not for example 'makefile' nor 'GNUmakefile'.

Under Windows[2] the same commands work, but 'Makevars.win' will be used in preference to 'Makevars', and only 'src/Makefile.win' will be used by R CMD INSTALL with 'src/Makefile' being ignored. For details of building DLLs with a variety of compilers, see file 'README.packages' and http://www.stats.uwo.ca/faculty/murdoch/software/compilingDLLs/ .

Under Windows you can supply an exports file called '*dllname*-win.def': otherwise all entry points in objects (but not libraries) supplied to R CMD SHLIB will be exported from the DLL. An example is 'stats-win.def' for the **stats** package.

5.6 Interfacing C++ code

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

[1] or the version specific to a sub-architecture

[2] The files in the R binary Windows distribution for installing source packages need to be installed.

```
// X.hh

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

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

```
// X.cc

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

static Y y;

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

} // extern "C"
```

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 system, one can simply use

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

to create the shared object, 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 6.5 [Printing], page 92) for all I/O if at all possible.

Most R header files can be included within C++ programs, and they should **not** be included within an `extern "C"` block (as they include C++ system headers). It may not be possible to include some R headers as they in turn include C header files that may cause conflicts—if this happens, define 'NO_C_HEADERS' before including the R headers, and include the appropriate headers yourself.

5.7 Fortran I/O

We have already warned against the use of C++ iostreams not least because output is not guaranteed to appear on the R console, and this warning applies equally to Fortran (77 or 9x) output to units `*` and `6`. See Section 6.5.1 [Printing from FORTRAN], page 92, which describes workarounds.

In the past most Fortran compilers implemented I/O on top of the C I/O system and so the two interworked successfully. This was true of `g77`, but it is less true of `gfortran` as used in `gcc 4.y.x`. In particular, any package that makes use of Fortran I/O will when compiled on Windows interfere with C I/O: when the Fortran I/O is initialized (typically when the package is loaded) the C `stdout` and `stderr` are switched to LF line endings. (Function `La_Init` in file 'src/main/lapack.c' shows how to mitigate this.) Even worse, prior to R 2.6.2 using Fortran output when running under the Windows GUI console (`Rgui`) would hang the R session. This is now avoided by ensuring that the Fortran output is written to a file ('fort.6' in the working directory).

5.8 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 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 pairlist 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 S4-compatibility macros defined in '`Rdefines.h`'. See Section 5.9 [Interface functions .Call and .External], page 77 for details on `.Call` and `.External`.

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 relatively 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. Neither `.Call` nor `.External` copy their arguments. You should treat arguments you receive through these interfaces as read-only.

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[3] subset of these is defined in the header file '`Rinternals.h`' in the directory '*`R_INCLUDE_DIR`*' (default '*`R_HOME`*`/include`') that should be available on any R installation.

Another 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 to be preferred if the code is intended to be shared with S. However, it is less well documented and even less tested. Note too that some idiomatic S4 constructions with these macros (such as assigning elements of character vectors or lists) are invalid in R.

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.

[3] see Chapter 6 [The R API], page 89: note that these are not all part of the API.

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*[4], 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 and in section "R Internal Structures" in *R Internals*, 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.

5.8.1 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, some or all of the allocated memory not being used is freed.

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. 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 during garbage collection. 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`, for example all elements of a protected list are automatically protected.

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 '`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);
```

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

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

```
#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);
```

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 example, 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 removed. It is usually wise to err on the side of caution and assume that any of the R macros and functions might remove 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 '`src/main/plot3d.c`').

Sometimes an object is changed (for example duplicated, coerced or grown) yet the current value needs to be protected. For these cases `PROTECT_WITH_INDEX` saves an index of the protection location that can be used to replace the protected value using `REPROTECT`. For example (from the internal code for `optim`)

```
PROTECT_INDEX ipx;

....
PROTECT_WITH_INDEX(s = eval(OS->R_fcall, OS->R_env), &ipx);
REPROTECT(s = coerceVector(s, REALSXP), ipx);
```

5.8.2 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 equivalent `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 6.1 [Memory allocation], page 89. 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.

5.8.3 Details of R types

Users of the '`Rinternals.h`' macros will need to know how the R types are known internally: if the '`Rdefines.h`' macros are used then S4-compatible names 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)
`LISTSXP`	"dotted-pair" list
`DOTSXP`	a '`...`' object
`NILSXP`	NULL
`SYMSXP`	name/symbol
`CLOSXP`	function or function closure
`ENVSXP`	environment

Among the important internal `SEXPTYPE`s are `LANGSXP`, `CHARSXP`, `PROMSXP`, etc. (**Note**: although it is possible to return objects of internal types, it is unsafe to do so as assumptions are made about how they are handled which may be violated at user-level evaluation.) More details are given in section "R Internal Structures" in *R Internals*.

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 still protected but now unused.

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

Note that these coercion functions are *not* the same as calling `as.numeric` (and so on) in R code, as they do not dispatch on the class of the object. Thus it is normally preferable to do the coercion in the calling R code.

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.

5.8.4 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)
{
   storage.mode(x) <- storage.mode(y) <- "double"
   .Call("out", x, y)
}
```

where we expect `x` and `y` to be numeric vectors (possibly integer), 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, *rx = REAL(x), *ry = REAL(y), *rans;
  SEXP ans;

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

Note the way `REAL` is used: as it is a function call it can be considerably faster to store the result and index that.

However, 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)
{
  R_len_t i, j, nx, ny;
  double tmp, *rx = REAL(x), *ry = REAL(y), *rans;
  SEXP ans, dim, dimnames;

  nx = length(x); ny = length(y);
  PROTECT(ans = allocVector(REALSXP, nx*ny));
  rans = REAL(ans);
  for(i = 0; i < nx; i++) {
    tmp = rx[i];
    for(j = 0; j < ny; j++)
      rans[i + nx*j] = tmp * ry[j];
  }
```

```
    PROTECT(dim = allocVector(INTSXP, 2));
    INTEGER(dim)[0] = nx; INTEGER(dim)[1] = ny;
    setAttrib(ans, R_DimSymbol, dim);

    PROTECT(dimnames = allocVector(VECSXP, 2));
    SET_VECTOR_ELT(dimnames, 0, getAttrib(x, R_NamesSymbol));
    SET_VECTOR_ELT(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 the default methods of `names<-`, `dim<-` and `dimnames<-` (for vectors and arrays), 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)[0] = 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.

5.8.5 Classes

In R the (S3) 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));
    SET_STRING_ELT(class, 0, mkChar("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 function `mkChar`.

5.8.6 Handling lists

Some care is needed with lists, as R moved early on from using LISP-like lists (now called "pairlists") to S-like generic vectors. As a result, the appropriate test for an 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_ELT(a, 1))[0];
```

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

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

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

  for (i = 0; i < length(list); i++)
    if(strcmp(CHAR(STRING_ELT(names, i)), str) == 0) {
      elmt = VECTOR_ELT(list, i);
      break;
    }
  return elmt;
}
```

and enables us to say

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

5.8.7 Handling character data

R character vectors are stored as `STRSXP`s, a vector type like `VECSXP` where every element is of type `CHARSXP`. The `CHARSXP` elements of `STRSXP`s are accessed using `STRING_ELT` and `SET_STRING_ELT`.

As of R 2.6.0, `CHARSXP`s are read-only objects and must never be modified. In particular, the C-style string contained in a `CHARSXP` should be treated as read-only and for this reason the `CHAR` function used to access the character data of a `CHARSXP` returns `(const char *)` (this also allows compilers to issue warnings about improper use). Since `CHARSXP`s are immutable, the same `CHARSXP` can be shared by any `STRSXP` needing an element representing the same string. As of R 2.6.0, R maintains a global cache of `CHARSXP`s so that there is only ever one `CHARSXP` representing a given string in memory.

You can obtain a `CHARSXP` by calling `mkChar` and providing a nul-terminated C-style string. This function will return a pre-existing `CHARSXP` if one with a matching string already exists, otherwise it will create a new one and add it to the cache before returning it to you.

Currently, it is still possible to create `CHARSXP`s using `allocVector` or `allocString`; `CHARSXP`s created in this way will not be captured by the global `CHARSXP` cache and this should be avoided. In the future, all `CHARSXP`s will be captured by the cache and this will allow further optimizations, for example, replacing calls to `strcmp` with pointer comparisons. A helper macro, CallocCharBuf, can be used to obtain a temporary character buffer for in-place string manipulation: this memory must be released using Free.

5.8.8 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_ELT(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 variables. defineVar creates a new binding or changes the value of an existing binding in the specified environment frame; it is the analogue of assign(symbol, value, envir = rho, inherits = FALSE), but unlike assign, defineVar does not make a copy of the object value.[5] setVar searches for an existing binding for symbol in rho or its enclosing environments. If a binding is found, its value is changed to value. Otherwise, a new binding with the specified value is created in the global environment. This corresponds to assign(symbol, value, envir = rho, inherits = TRUE).

5.8.9 Some convenience functions

Some operations are done so frequently that there are convenience functions to handle them. Suppose we wanted to pass a single logical argument ignore_quotes: we could use

```
int ign;

ign = asLogical(ignore_quotes);
if(ign == NA_LOGICAL) error("'ignore_quotes' must be TRUE or FALSE");
```

which will do any coercion needed (at least from a vector argument), and return NA_LOGICAL if the value passed was NA or coercion failed. There are also asInteger, asReal and asComplex. The function asChar returns a CHARSXP. All of these functions ignore any elements of an input vector after the first.

To return a length-one real vector we can use

```
double x;

...
```

[5] You can assign a *copy* of the object in the environment frame rho using defineVar(symbol, duplicate(value), rho)).

```
return ScalarReal(x);
```

and there are versions of this for all the atomic vector types (those for a length-one character vector being `ScalarString` with argument a `CHARSXP` and `mkString` with argument `const char *`).

Some of the `isXXXX` functions differ from their apparent R-level counterparts: for example `isVector` is true for any atomic vector type (`isVectorAtomic`) and for lists and expressions (`isVectorList`) (with no check on attributes). `isMatrix` is a test of a length-2 `"dim"` attribute.

There are a series of small macros/functions to help construct pairlists and language objects (whose internal structures just differ by `SEXPTYPE`. Function `CONS(u, v)` is the basic building block: is constructs a pairlist from `u` followed by `v` (which is a pairlist or `R_NilValue`). `LCONS` is a variant that constructs a language object. Functions `list1` to `list4` construct a pairlist from one to four items, and `lang1` to `lang4` do the same for a language object (a function to call plus zero to three arguments). Function `elt` and `lastElt` find the *i*th element and the last element of a pairlist, and `nthcdr` returns a pointer to the *n*th position in the pairlist (whose `CAR` is the *n*th item).

Functions `str2type` and `type2str` map R length-one character strings to and from `SEXPTYPE` numbers, and `type2char` maps numbers to C character strings.

5.8.10 Named objects and copying

When assignments are done in R such as

```
x <- 1:10
y <- x
```

the named object is not necessarily copied, so after those two assignments `y` and `x` are bound to the same `SEXPREC` (the structure a `SEXP` points to). This means that any code which alters one of them has to make a copy before modifying the copy if the usual R semantics are to apply. Note that whereas `.C` and `.Fortran` do copy their arguments (unless the dangerous `dup = FALSE` is used), `.Call` and `.External` do not. So `duplicate` is commonly called on arguments to `.Call` before modifying them.

However, at least some of this copying is unneeded. In the first assignment shown, `x <- 1:10`, R first creates an object with value `1:10` and then assigns it to `x` but if `x` is modified no copy is necessary as the temporary object with value `1:10` cannot be referred to again. R distinguishes between named and unnamed objects *via* a field in a `SEXPREC` that can be accessed via the macros `NAMED` and `SET_NAMED`. This can take values

`0` The object is not bound to any symbol

`1` The object has been bound to exactly one symbol

`2` The object has potentially been bound to two or more symbols, and one should act as if another variable is currently bound to this value.

Note the past tenses: R does not do full reference counting and there may currently be fewer bindings.

It is safe to modify the value of any `SEXP` for which `NAMED(foo)` is zero, and if `NAMED(foo)` is two, the value should be duplicated (via a call to `duplicate`) before any modification. Note that it is the responsibility of the author of the code making the modification to do the duplication, even if it is `x` whose value is being modified after `y <- x`.

The case `NAMED(foo) == 1` allows some optimization, but it can be ignored (and duplication done whenever `NAMED(foo) > 0`). (This optimization is not currently usable in user code.) It is intended for use within assignment functions. Suppose we used

```
x <- 1:10
foo(x) <- 3
```

which is computed as

```
x <- 1:10
x <- "foo<-"(x, 3)
```

Then inside `"foo<-"` the object pointing to the current value of `x` will have `NAMED(foo)` as one, and it would be safe to modify it as the only symbol bound to it is `x` and that will be rebound immediately. (Provided the remaining code in `"foo<-"` make no reference to `x`, and no one is going to attempt a direct call such as `y <- "foo<-"(x)`.)

Currently all arguments to a `.Call` call will have `NAMED` set to 2, and so users must assume that they need to be duplicated before alteration.

5.9 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.

5.9.1 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)
{
  R_len_t 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.

5.9.2 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;
  const char *name;
  SEXP el;

  args = CDR(args); /* skip 'name' */
  for(i = 0; args != R_NilValue; i++, args = CDR(args)) {
    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_ELT(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

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

The (very similar) C code is in the scripts.

5.9.3 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 NAs using macros defined in 'R_exts/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 doubles to the special values.

5.10 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)
{
  R_len_t 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_ELT(list, i), rho);
    SET_VECTOR_ELT(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)
{
  R_len_t 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++) {
    SETCADR(R_fcall, VECTOR_ELT(list, i));
    SET_VECTOR_ELT(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 pairlist of two elements, but this will only be clear to those with a knowledge of a LISP-like language.

As a more comprehensive example of constructing an R call in C code and evaluating, consider the following fragment of `printAttributes` in ‘`src/main/print.c`’.

```
    /* Need to construct a call to
       print(CAR(a), digits=digits)
       based on the R_print structure, then eval(call, env).
       See do_docall for the template for this sort of thing.
    */
    SEXP s, t;
    PROTECT(t = s = allocList(3));
```

```
    SET_TYPEOF(s, LANGSXP);
    CAR(t) = install("print"); t = CDR(t);
    CAR(t) = CAR(a); t = CDR(t);
    CAR(t) = allocVector(INTSXP, 1);
    INTEGER(CAR(t))[0] = digits;
    SET_TAG(t, install("digits"));
    eval(s, env);
    UNPROTECT(1);
```

At this point `CAR(a)` is the R object to be printed, the current attribute. There are three steps: the call is constructed as a pairlist of length 3, the list is filled in, and the expression represented by the pairlist is evaluated.

A pairlist is quite distinct from a generic vector list, the only user-visible form of list in R. A pairlist is a linked list (with `CDR(t)` computing the next entry), with items (accessed by `CAR(t)`) and names or tags (set by `SET_TAG`). In this call there are to be three items, a symbol (pointing to the function to be called) and two argument values, the first unnamed and the second named. Setting the type to `LANGSXP` makes this a call which can be evaluated.

5.10.1 Zero-finding

In this section we re-work the example of `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`.

5.10.2 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), *rgr, *rans;
  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)));
  rgr = REAL(gradient); rans = REAL(ans);

  for(i = 0, start = 0; i < LENGTH(theta); i++, start += LENGTH(ans)) {
    PROTECT(par = findVar(install(CHAR(STRING_ELT(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++)
      rgr[j + start] = (REAL(ans1)[j] - rans[j])/delta;
    REAL(par)[0] = tt;
    UNPROTECT(2); /* par, ans1 */
  }

  PROTECT(dimnames = allocVector(VECSXP, 2));
  SET_VECTOR_ELT(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_ELT(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_ELT(theta, i)` accesses the `i`'th element of the `STRSXP theta`. Macro `CHAR()` extracts the actual character representation[6] 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++)
        rgr[j + start] = (REAL(ans1)[j] - rans[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));
    SET_VECTOR_ELT(dimnames, 1, theta);
    dimnamesgets(gradient, dimnames);
    setAttrib(ans, install("gradient"), gradient);
    UNPROTECT(3);
    return ans;
  }
```

[6] see Section 5.14 [Character encoding issues], page 88 for why this might not be what is required.

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`.

5.11 Parsing R code from C

Suppose an R extension want to accept an R expression from the user and evaluate it. The previous section covered evaluation, but the expression will be entered as text and needs to be parsed first. A small part of R's parse interface is declared in header file 'R_ext/Parse.h'[7].

An example of the usage can be found in the (example) Windows package **windlgs** included in the R source tree. The essential part is

```
#include <R.h>
#include <Rinternals.h>
#include <R_ext/Parse.h>

SEXP menu_ttest3()
{
    char cmd[256];
    SEXP cmdSexp, cmdexpr, ans = R_NilValue;
    int i;
    ParseStatus status;
   ...
    if(done == 1) {
        PROTECT(cmdSexp = allocVector(STRSXP, 1));
        SET_STRING_ELT(cmdSexp, 0, mkChar(cmd));
        cmdexpr = PROTECT(R_ParseVector(cmdSexp, -1, &status, R_NilValue));
        if (status != PARSE_OK) {
            UNPROTECT(2);
            error("invalid call %s", cmd);
        }
        /* Loop is needed here as EXPSEXP will be of length > 1 */
        for(i = 0; i < length(cmdexpr); i++)
            ans = eval(VECTOR_ELT(cmdexpr, i), R_GlobalEnv);
        UNPROTECT(2);
    }
    return ans;
}
```

Note that a single line of text may give rise to more than one R expression.

`R_ParseVector` is essentially the code used to implement `parse(text=)` at R level. The first argument is a character vector (corresponding to `text`) and the second the maximal number of expressions to parse (corresponding to `n`). The third argument is a pointer to a variable of an enumeration type, and it is normal (as `parse` does) to regard all values other than `PARSE_OK` as an error. Other values which might be returned are `PARSE_INCOMPLETE` (an incomplete expression was found) and `PARSE_ERROR` (a syntax error), in both cases the value returned being `R_NilValue`. The fourth argument is a `srcfile` object or the R `NULL` object (as in the example above). In the former case a `srcref` attribute would be attached to the result, containing a list of `srcref` objects of the same length as the expression, to allow it to be echoed with its original formatting.

[7] This is only guaranteed to show the current interface: it is liable to change.

5.12 External pointers and weak references

The SEXPTYPEs EXTPTRSXP and WEAKREFSXP can be encountered at R level, but are created in C code.

External pointer SEXPs are intended to handle references to C structures such as 'handles', and are used for this purpose in package **RODBC** for example. They are unusual in their copying semantics in that when an R object is copied, the external pointer object is not duplicated. (For this reason external pointers should only be used as part of an object with normal semantics, for example an attribute or an element of a list.)

An external pointer is created by

```
SEXP R_MakeExternalPtr(void *p, SEXP tag, SEXP prot);
```

where p is the pointer (and hence this cannot portably be a function pointer), and tag and prot are references to ordinary R objects which will remain in existence (be protected from garbage collection) for the lifetime of the external pointer object. A useful convention is to use the tag field for some form of type identification and the prot field for protecting the memory that the external pointer represents, if that memory is allocated from the R heap. Both tag and prot can be R_NilValue, and often are.

The elements of an external pointer can be accessed and set *via*

```
void *R_ExternalPtrAddr(SEXP s);
SEXP R_ExternalPtrTag(SEXP s);
SEXP R_ExternalPtrProtected(SEXP s);
void R_ClearExternalPtr(SEXP s);
void R_SetExternalPtrAddr(SEXP s, void *p);
void R_SetExternalPtrTag(SEXP s, SEXP tag);
void R_SetExternalPtrProtected(SEXP s, SEXP p);
```

Clearing a pointer sets its value to the C NULL pointer.

An external pointer object can have a *finalizer*, a piece of code to be run when the object is garbage collected. This can be R code or C code, and the various interfaces are

```
void R_RegisterFinalizer(SEXP s, SEXP fun);
void R_RegisterFinalizerEx(SEXP s, SEXP fun, Rboolean onexit);

typedef void (*R_CFinalizer_t)(SEXP);
void R_RegisterCFinalizer(SEXP s, R_CFinalizer_t fun);
void R_RegisterCFinalizerEx(SEXP s, R_CFinalizer_t fun, Rboolean onexit);
```

The R function indicated by fun should be a function of a single argument, the object to be finalized. R does not perform a garbage collection when shutting down, and the onexit argument of the extended forms can be used to ask that the finalizer be run during a normal shutdown of the R session. It is suggested that it is good practice to clear the pointer on finalization.

The only R level function for interacting with external pointers is reg.finalizer which can be used to set a finalizer.

It is probably not a good idea to allow an external pointer to be saved and then reloaded, but if this happens the pointer will be set to the C NULL pointer.

Weak references are used to allow the programmer to maintain information on entities without preventing the garbage collection of the entities once they become unreachable.

A weak reference contains a key and a value. The value is reachable is if it either reachable directly or via weak references with reachable keys. Once a value is determined to be unreachable during garbage collection, the key and value are set to R_NilValue and the finalizer will be run later in the garbage collection.

Weak reference objects are created by one of

```
SEXP R_MakeWeakRef(SEXP key, SEXP val, SEXP fin, Rboolean onexit);
SEXP R_MakeWeakRefC(SEXP key, SEXP val, R_CFinalizer_t fin,
                    Rboolean onexit);
```

where the R or C finalizer are specified in exactly the same way as for an external pointer object (whose finalization interface is implemented via weak references).

The parts can be accessed *via*

```
SEXP R_WeakRefKey(SEXP w);
SEXP R_WeakRefValue(SEXP w);
void R_RunWeakRefFinalizer(SEXP w);
```

A toy example of the use of weak references can be found at www.stat.uiowa.edu/~luke/R/references/weakfinex.html, but that is used to add finalizers to external pointers which can now be done more directly.

5.13 Vector accessor functions

The vector accessors like `REAL` and `INTEGER` and `VECTOR_ELT` are *functions* when used in R extensions. (For efficiency they are macros when used in the R source code, apart from `SET_STRING_ELT` and `SET_VECTOR_ELT` which are always functions.)

The accessor functions check that they are being used on an appropriate type of `SEXP`. By default a certain amount of misuse is allowed where the internal representation is the same: for example `LOGICAL` can be used on a `INTSXP` and `SET_VECTOR_ELT` on a `STRSXP`. Strict checking can be enabled by compiling R (specifically '`src/main/memory.c`') with '`USE_TYPE_CHECKING_STRICT`' defined (e.g. in as the configure variable '`DEFS`' on a Unix-alike).

If efficiency is essential, the macro versions of the accessors can be obtained by defining '`USE_RINTERNALS`' before including '`Rinternals.h`'. If you find it necessary to do so, please do test that your code compiled without '`USE_RINTERNALS`' defined, as this provides a stricter test that the accessors have been used correctly.

5.14 Character encoding issues

As from R 2.5.0 `CHARSXP`s can be marked as coming from a known encoding (Latin-1 or UTF-8). This is mainly intended for human-readable output, and most packages can just treat such `CHARSXP`s as a whole. However, if they need to be interpreted as characters or output at C level then it would normally be correct to ensure that they are converted to the encoding of the currrent locale: this can be done by accessing the data in the `CHARSXP` by `translateChar` rather than by `CHAR`. If re-encoding is needed this allocates memory with `R_alloc` which thus persists to the end of the `.Call`/`.External` call unless `vmaxset` is used.

6 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 several other header files from the directory '*R_INCLUDE_DIR*/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 extra some compatibility definitions (for example the S_complex type from S).

The defines used for compatibility with S sometimes causes conflicts (notably with Windows headers), and the known problematic defines can be removed by defining STRICT_R_HEADERS.

Most of these header files, including all those included by 'R.h', can be used from C++ code. Some others need to be included within an extern "C" declaration, and for clarity this is advisable for all R header files.

> **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.

This remapping can cause problems[1], and can be eliminated by defining R_NO_REMAP and prepending Rf_ to *all* the function names used from 'Rinternals.h' and 'R_ext/Error.h'.

We can classify the entry points as

- *API* Entry points which are documented in this manual and declared in an installed header file. These can be used in distributed packages and will only be changed after deprecation.
- *public* Entry points declared in an installed header file that are exported on all R platforms but are not documented and subject to change without notice.
- *private* Entry points that are used when building R and exported on all R platforms but are not declared in the installed header files. Do not use these in distributed code.
- *hidden* Entry points that are where possible (Windows and some modern Unix compilers/loaders when using R as a shared library) not exported.

6.1 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).

6.1.1 Transient storage allocation

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

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

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

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

(size_t is defined in 'stddef.h' which the header defining R_alloc includes.)

There is a similar call, S_alloc (for compatibility with older versions of S) which zeroes the memory allocated,

[1] Known problems are redefining error, length, vector and warning

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

and

```
char *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.

For compatibility with current versions of S, header 'S.h' (only) defines wrapper macros equivalent to

```
type* Salloc(long n, int type)
type* Srealloc(char *p, long new, long old, int type)
```

This memory is taken from the heap, 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.

Note that this memory will be freed on error or user interrupt (if allowed: see Section 6.12 [Allowing interrupts], page 101).

Note that although *n* is `long`, there are limits imposed by R's internal allocation mechanism. These will only come into play on 64-bit systems, where the current limit for *n* is just under 16Gb.

6.1.2 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 during allocation 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.)

Users should arrange to `Free` this memory when no longer needed, including on error or user interrupt. This can often be done most conveniently from an `on.exit` action in the calling R function – see `pwilcox` for an example.

Do not assume that memory allocated by `Calloc`/`Realloc` comes from the same pool as used by `malloc`: in particular do not use `free` or `strdup` with it.

These entry points need to be prefixed by `R_` if `STRICT_R_HEADERS` has been defined.

6.2 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. (Don't do this if the string contains '%' or might otherwise be interpreted as a format.)

If `STRICT_R_HEADERS` is not defined 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.

6.2.1 Error handling from FORTRAN

There are two interface function provided to call `error` and `warning` from FORTRAN code, in each case with a simple character string argument. They are defined as

```
subroutine rexit(message)
subroutine rwarn(message)
```

Messages of more than 255 characters are truncated, with a warning.

6.3 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 '`Rmath.h`'; See Section 6.7.1 [Distribution functions], page 93. Those calls generate a single variate and should also be enclosed in calls to `GetRNGstate` and `PutRNGstate`.

In addition, there is an interface (defined in header '`R_ext/Applic.h`') to the generation of random 2-dimensional tables with given row and column totals using Patefield's algorithm.

`void rcont2` (*int** `nrow`, *int** `ncol`, *int** `nrowt`, *int** `ncolt`, *int** `ntotal`, *double** `fact`, *int** `jwork`, *int** `matrix`) [Function]

Here, *nrow* and *ncol* give the numbers *nr* and *nc* of rows and columns and *nrowt* and *ncolt* the corresponding row and column totals, respectively, *ntotal* gives the sum of the row (or columns) totals, *jwork* is a workspace of length *nc*, and on output *matrix* a contains the $nr * nc$ generated random counts in the usual column-major order.

6.4 Missing and IEEE special values

A set of functions is provided to test for `NA`, `Inf`, `-Inf` 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 via function `R_IsNaN` which is true for `NaN` but not `NA`.

Do use `R_FINITE` rather than `isfinite` or `finite`; the latter is often mendacious and `isfinite` is only available on a few platforms, on which `R_FINITE` is a macro expanding to `isfinite`.

Currently in C code `ISNAN` is a macro calling `isnan`. (Since this gives problems on some C++ systems, if the R headers is called from C++ code a function call is used.)

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.

6.5 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 analogues using the `vprintf` interface.

6.5.1 Printing from FORTRAN

On many systems 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 not portable and 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.

6.6 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

On most current platforms these are all the same, but it is unwise to rely on this. Note that names with underscores are not legal in FORTRAN 77, and are not portably handled by the above macros. (Also, all FORTRAN names for use by R are lower case, but this is not enforced by the macros.)

For example, suppose we want to call R's normal random numbers from FORTRAN. We need a C wrapper along the lines of

```
#include <R.h>

void F77_SUB(rndstart)(void) { GetRNGstate(); }
void F77_SUB(rndend)(void) { PutRNGstate(); }
double F77_SUB(normrnd)(void) { return norm_rand(); }
```

to be called from FORTRAN as in

```
      subroutine testit()
      double precision normrnd, x
      call rndstart()
      x = normrnd()
      call dblepr("X was", 5, x, 1)
      call rndend()
      end
```

Note that this is not guaranteed to be portable, for the return conventions might not be compatible between the C and FORTRAN compilers used. (Passing values via arguments is safer.)

The standard packages, for example **stats**, are a rich source of further examples.

6.7 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 files '`R_ext/BLAS.h`', '`R_ext/Lapack.h`' and '`R_ext/Linpack.h`' contains declarations of the BLAS, LAPACK and LINPACK/EISPACK linear algebra functions included 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 '`Rmath.h`' lists many other functions that are available and documented in the following subsections. Many of these are C interfaces to the code behind R functions, so the R function documentation may give further details.

6.7.1 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

TRUE (or 1) for normal use, but can be FALSE (or 0) if the probability of the upper tail is desired or specified.

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.

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

```
- pdist(t, ..., /*lower_tail = */ FALSE, /* give_log = */ TRUE)
```

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

The random-variate generation routine `rnorm` returns one normal variate. See Section 6.3 [Random numbers], page 91, 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-squared	`chisq`	`df`
non-central chi-squared	`nchisq`	`df`, `lambda`
exponential	`exp`	`scale`
F	`f`	`n1`, `n2`
non-central F	`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	`nt`	`df`, `delta`
Studentized range	`tukey` (*)	`rr`, `cc`, `df`
uniform	`unif`	`a`, `b`
Weibull	`weibull`	`shape`, `scale`
Wilcoxon rank sum	`wilcox`	`m`, `n`
Wilcoxon signed rank	`signrank`	`n`

Entries marked with an asterisk only have '`p`' and '`q`' functions available, and none of the non-central distributions have '`r`' functions. After a call to `dwilcox`, `pwilcox` or `qwilcox` the function `wilcox_free()` should be called, and similarly for the signed rank functions.

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

6.7.2 Mathematical functions

`double gammafn (`*double* `x)` [Function]
`double lgammafn (`*double* `x)` [Function]
`double digamma (`*double* `x)` [Function]

`double trigamma (double x)` [Function]
`double tetragamma (double x)` [Function]
`double pentagamma (double x)` [Function]
`double psigamma (double x, double deriv)` [Function]

The Gamma function, its natural logarithm and first four derivatives and the n-th derivative of Psi, the digamma function.

`double beta (double a, double b)` [Function]
`double lbeta (double a, double b)` [Function]

The (complete) Beta function and its natural logarithm.

`double choose (double n, double k)` [Function]
`double lchoose (double n, double k)` [Function]

The number of combinations of *k* items chosen from from *n* and its natural logarithm. *n* and *k* are rounded to the nearest integer.

`double bessel_i (double x, double nu, double expo)` [Function]
`double bessel_j (double x, double nu)` [Function]
`double bessel_k (double x, double nu, double expo)` [Function]
`double bessel_y (double x, double nu)` [Function]

Bessel functions of types I, J, K and Y with index *nu*. For `bessel_i` and `bessel_k` there is the option to return exp(-*x*) I(*x*; *nu*) or exp(*x*) K(*x*; *nu*) if *expo* is 2. (Use `expo == 1` for unscaled values.)

6.7.3 Numerical Utilities

There are a few other numerical utility functions available as entry points.

`double R_pow (double x, double y)` [Function]
`double R_pow_di (double x, int i)` [Function]

`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)` [Function]

`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 (double x)` [Function]

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

This may be provided by your platform, in which case it is not included in '`Rmath.h`', but is (probably) in '`math.h`' which '`Rmath.h`' includes. For backwards compatibility with R versions prior to 1.5.0, the entry point `Rf_log1p` is still provided.

`double log1pmx (double x)` [Function]

Computes `log(1 + x) - x` (*log 1* **p***lus x* **m***inus* **x**), accurately even for small *x*, i.e., $|x| \ll 1$.

`double expm1 (double x)` [Function]

Computes `exp(x) - 1` (*exp x* **m***inus 1*), accurately even for small *x*, i.e., $|x| \ll 1$.

This may be provided by your platform, in which case it is not included in '`Rmath.h`', but is (probably) in '`math.h`' which '`Rmath.h`' includes.

`double lgamma1p (double x)` [Function]

Computes `log(gamma(x + 1))` (*log(gamma(1* **p***lus x))*), accurately even for small *x*, i.e., $0 < x < 0.5$.

`double logspace_add (double logx, double logy)` [Function]
`double logspace_sub (double logx, double logy)` [Function]

Compute the log of a sum or difference from logs of terms, i.e., "x + y" as `log (exp(logx) + exp(logy))` and "x - y" as `log (exp(logx) - exp(logy))`, without causing overflows or throwing away too much accuracy.

`int imax2 (int x, int y)` [Function]
`int imin2 (int x, int y)` [Function]
`double fmax2 (double x, double y)` [Function]
`double fmin2 (double x, double y)` [Function]

Return the larger (`max`) or smaller (`min`) of two integer or double numbers, respectively.

`double sign (double x)` [Function]

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)` [Function]

Performs "transfer of sign" and is defined as $|x| * \mathrm{sign}(y)$.

`double fprec (double x, double digits)` [Function]

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)` [Function]

Returns the value of x rounded to *digits significant* decimal digits.

This is the function used by R's `signif()`.

`double ftrunc (double x)` [Function]

Returns the value of x truncated (to an integer value) towards zero.

6.7.4 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 '`Rmath.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`	π	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(π)	1.1283792
`M_SQRT2`	sqrt(2)	1.4142136
`M_SQRT1_2`	1/sqrt(2)	0.7071068
`M_SQRT_3`	sqrt(3)	1.7320508
`M_SQRT_32`	sqrt(32)	5.6568542
`M_LOG10_2`	log10(2)	0.3010300

`M_2PI`	2π	6.2831853
`M_SQRT_PI`	sqrt(π)	1.7724539
`M_1_SQRT_2PI`	1/sqrt(2π)	0.3989423
`M_SQRT_2dPI`	sqrt($2/\pi$)	0.7978846
`M_LN_SQRT_PI`	ln(sqrt(π))	0.5723649
`M_LN_SQRT_2PI`	ln(sqrt(2π))	0.9189385
`M_LN_SQRT_PId2`	ln(sqrt($\pi/2$))	0.2257914

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

Further, the included header '`R_ext/Boolean.h`' has constants `TRUE` and `FALSE = 0` of type `Rboolean` in order to provide a way of using "logical" variables in C consistently.

6.8 Optimization

The C code underlying `optim` can be accessed directly. The user needs to supply a function to compute the function to be minimized, of the type

```
typedef double optimfn(int n, double *par, void *ex);
```

where the first argument is the number of parameters in the second argument. The third argument is a pointer passed down from the calling routine, normally used to carry auxiliary information.

Some of the methods also require a gradient function

```
typedef void optimgr(int n, double *par, double *gr, void *ex);
```

which passes back the gradient in the `gr` argument. No function is provided for finite-differencing, nor for approximating the Hessian at the result.

The interfaces (defined in header '`R_ext/Applic.h`') are

- Nelder Mead:

```
void nmmin(int n, double *xin, double *x, double *Fmin, optimfn fn,
           int *fail, double abstol, double intol, void *ex,
           double alpha, double beta, double gamma, int trace,
           int *fncount, int maxit);
```

- BFGS:

```
void vmmin(int n, double *x, double *Fmin,
           optimfn fn, optimgr gr, int maxit, int trace,
           int *mask, double abstol, double reltol, int nREPORT,
           void *ex, int *fncount, int *grcount, int *fail);
```

- Conjugate gradients:

```
void cgmin(int n, double *xin, double *x, double *Fmin,
           optimfn fn, optimgr gr, int *fail, double abstol,
           double intol, void *ex, int type, int trace,
           int *fncount, int *grcount, int maxit);
```

- Limited-memory BFGS with bounds:

```
void lbfgsb(int n, int lmm, double *x, double *lower,
            double *upper, int *nbd, double *Fmin, optimfn fn,
            optimgr gr, int *fail, void *ex, double factr,
            double pgtol, int *fncount, int *grcount,
            int maxit, char *msg, int trace, int nREPORT);
```

- Simulated annealing:

```
void samin(int n, double *x, double *Fmin, optimfn fn, int maxit,
           int tmax, double temp, int trace, void *ex);
```

Many of the arguments are common to the various methods. `n` is the number of parameters, `x` or `xin` is the starting parameters on entry and `x` the final parameters on exit, with final value returned in `Fmin`. Most of the other parameters can be found from the help page for `optim`: see the source code '`src/appl/lbfgsb.c`' for the values of `nbd`, which specifies which bounds are to be used.

6.9 Integration

The C code underlying `integrate` can be accessed directly. The user needs to supply a *vectorizing* C function to compute the function to be integrated, of the type

```
typedef void integr_fn(double *x, int n, void *ex);
```

where `x[]` is both input and output and has length `n`, i.e., a C function, say `fn`, of type `integr_fn` must basically do `for(i in 1:n) x[i] := f(x[i], ex)`. The vectorization requirement can be used to speed up the integrand instead of calling it `n` times. Note that in the current implementation built on QUADPACK, `n` will be either 15 or 21. The `ex` argument is a pointer passed down from the calling routine, normally used to carry auxiliary information.

There are interfaces (defined in header '`R_ext/Applic.h`') for definite and for indefinite integrals. 'Indefinite' means that at least one of the integration boundaries is not finite.

- Finite:

```
void Rdqags(integr_fn f, void *ex, double *a, double *b,
            double *epsabs, double *epsrel,
            double *result, double *abserr, int *neval, int *ier,
            int *limit, int *lenw, int *last,
            int *iwork, double *work);
```

- Indefinite:

```
void Rdqagi(integr_fn f, void *ex, double *bound, int *inf,
            double *epsabs, double *epsrel,
            double *result, double *abserr, int *neval, int *ier,
            int *limit, int *lenw, int *last,
            int *iwork, double *work);
```

Only the 3rd and 4th argument differ for the two integrators; for the definite integral, using `Rdqags`, `a` and `b` are the integration interval bounds, whereas for an indefinite integral, using `Rdqagi`, `bound` is the finite bound of the integration (if the integral is not doubly-infinite) and `inf` is a code indicating the kind of integration range,

`inf = 1` corresponds to (bound, +Inf),

`inf = -1` corresponds to (-Inf, bound),

`inf = 2` corresponds to (-Inf, +Inf),

`f` and `ex` define the integrand function, see above; `epsabs` and `epsrel` specify the absolute and relative accuracy requested, `result`, `abserr` and `last` are the output components `value`, `abs.err` and `subdivisions` of the R function integrate, where `neval` gives the number of integrand function evaluations, and the error code `ier` is translated to R's `integrate() $ message`, look at that function definition. `limit` corresponds to `integrate(..., subdivisions = *)`. It seems you should always define the two work arrays and the length of the second one as

```
lenw = 4 * limit;
iwork =   (int *) R_alloc(limit, sizeof(int));
work = (double *) R_alloc(lenw,  sizeof(double));
```

The comments in the source code in '`src/appl/integrate.c`' give more details, particularly about reasons for failure (`ier >= 1`).

6.10 Utility functions

R has a fairly comprehensive set of sort routines which are made available to users' C code. These are declared in header file 'R_ext/Utils.h' (included by 'R.h') and include the following.

void R_isort (*int* x, int n*) [Function]
void R_rsort (*double* x, int n*) [Function]
void R_csort (*Rcomplex* x, int n*) [Function]
void rsort_with_index (*double* x, int* index, int n*) [Function]

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

rsort_with_index sorts on *x*, and applies the same permutation to *index*. NAs are sorted last.

void revsort (*double* x, int* index, int n*) [Function]

Is similar to rsort_with_index but sorts into decreasing order, and NAs are not handled.

void iPsort (*int* x, int n, int k*) [Function]
void rPsort (*double* x, int n, int k*) [Function]
void cPsort (*Rcomplex* x, int n, int k*) [Function]

These 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.

void R_qsort (*double *v, int i, int j*) [Function]
void R_qsort_I (*double *v, int *I, int i, int j*) [Function]
void R_qsort_int (*int *iv, int i, int j*) [Function]
void R_qsort_int_I (*int *iv, int *I, int i, int j*) [Function]

These routines sort *v*[*i*:*j*] or *iv*[*i*:*j*] (using 1-indexing, i.e., *v*[1] is the first element) calling the quicksort algorithm as used by R's sort(v, method = "quick") and documented on the help page for the R function sort. The ..._I() versions also return the sort.index() vector in I. Note that the ordering is *not* stable, so tied values may be permuted.

Note that NAs are not handled (explicitly) and you should use different sorting functions if NAs can be present.

subroutine qsort4 (*double precision v, integer indx, integer ii, integer jj*) [Function]
subroutine qsort3 (*double precision v, integer ii, integer jj*) [Function]

The FORTRAN interface routines for sorting double precision vectors are qsort3 and qsort4, equivalent to R_qsort and R_qsort_I, respectively.

void R_max_col (*double* matrix, int* nr, int* nc, int* maxes, int* ties_meth*) [Function]

Given the *nr* by *nc* matrix matrix in column-major ("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). In the case of ties (multiple maxima), **ties_meth* is an integer code in 1:3 determining the method: 1 = "random", 2 = "first" and 3 = "last". See R's help page ?max.col.

int findInterval (*double* xt, int n, double x, Rboolean rightmost_closed, Rboolean all_inside, int ilo, int* mflag*) [Function]

Given the ordered vector *xt* of length *n*, return the interval or index of *x* in *xt*[], typically $\max(i;\ 1 \le i \le n\ \&\ xt[i] \le x)$ where we use 1-indexing as in R and FORTRAN (but not C). If *rightmost_closed* is true, also returns $n-1$ if x equals $xt[n]$. If *all_inside* is not 0, the result is coerced to lie in 1:(*n*-1) even when *x* is outside the *xt*[] range. On return, **mflag* equals -1 if x < $xt[1]$, $+1$ if x >= $xt[n]$, and 0 otherwise.

The algorithm is particularly fast when *ilo* is set to the last result of `findInterval()` and *x* is a value of a sequence which is increasing or decreasing for subsequent calls.

There is also an `F77_CALL(interv)()` version of `findInterval()` with the same arguments, but all pointers.

The following two functions do *numerical* colorspace conversion from HSV to RGB and back. Note that all colours must be in [0,1].

`void hsv2rgb` (*double* `h`, *double* `s`, *double* `v`, *double* *`r`, *double* *`g`, *double* *`b`) [Function]

`void rgb2hsv` (*double* `r`, *double* `g`, *double* `b`, *double* *`h`, *double* *`s`, *double* *`v`) [Function]

A system-independent interface to produce the name of a temporary file is provided as

`char * R_tmpnam` (*const char* *`prefix`) [Function]

Return a pathname for a temporary file with name beginning with *prefix*. A `NULL` prefix is replaced by `""`.

There is also the internal function used to expand file names in several R functions, and called directly by `path.expand`.

`const char * R_ExpandFileName` (*const char* *`fn`) [Function]

Expand a path name *fn* by replacing a leading tilde by the user's home directory (if defined). The precise meaning is platform-specific; it will usually be taken from the environment variable `HOME` if this is defined.

6.11 Re-encoding

R has its own C-level interface to the encoding conversion capabilities provided by `iconv`, for the following reasons

- These wrapper routines do error-handling when no usable implementation of `iconv` was available at configure time.
- Under Windows they arrange to load the '`iconv.dll`' at first use.
- There are incompatibilities between the declarations in different implementations of `iconv`.

These are declared in header file '`R_ext/Riconv.h`'.

`void *Riconv_open` (*const char* *`to`, *const char* *`from`) [Function]

Set up a pointer to an encoding object to be used to convert between two encodings: `""` indicates the current locale.

`size_t Riconv` (*void* *`cd`, *const char* **`inbuf`, *size_t* *`inbytesleft`, *char* **`outbuf`, *size_t* *`outbytesleft`) [Function]

Convert as much as possible of `inbuf` to `outbuf`. Initially the `int` variables indicate the number of bytes available in the buffers, and they are updated (and the `char` pointers are updated to point to the next free byte in the buffer). The return value is the number of characters converted, or `(size_t)-1` (beware: `size_t` is usually an unsigned type). It should be safe to assume that an error condition sets `errno` to one of `E2BIG` (the output buffer is full), `EILSEQ` (the input cannot be converted, and might be invalid in the encoding specified) or `EINVAL` (the input does not end with a complete multi-byte character).

`int Riconv_close` (*void* * `cd`) [Function]

Free the resources of an encoding object.

6.12 Allowing interrupts

No port of R can be interrupted whilst running long computations in compiled code, so programmers should make provision for the code to be interrupted at suitable points by calling from C

```
#include <R_ext/Utils.h>

void R_CheckUserInterrupt(void);
```

and from FORTRAN

```
subroutine rchkusr()
```

These check if the user has requested an interrupt, and if so branch to R's error handling functions.

Note that it is possible that the code behind one of the entry points defined here if called from your C or FORTRAN code could be interruptible or generate an error and so not return to your code.

6.13 Platform and 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 'Rconfig.h' (included by 'R.h') is used to define platform-specific macros that are mainly for use in other header files. The macro `WORDS_BIGENDIAN` is defined on big-endian systems (e.g. `sparc-sun-solaris2.6`) and not on little-endian systems (such as `i686` under Linux or Windows). It can be useful when manipulating binary files.

Header file 'Rversion.h' (**not** 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(1, 9, 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 '9.0').

6.14 Inlining C functions

The C99 keyword `inline` is recognized by some compilers used to build R whereas others need `__inline__` or do not support inlining. Portable code can be written using the macro `R_INLINE` (defined in file 'Rconfig.h' included by 'R.h'), as for example from package **cluster**

```
#include <R.h>

static R_INLINE int ind_2(int l, int j)
{
...
}
```

Be aware that using inlining with functions in more than one compilation unit is almost impossible to do portably: see http://www.greenend.org.uk/rjk/2003/03/inline.html. All the R configure code has checked is that `R_INLINE` can be used in a single C file with the compiler used to build R. We recommend that packages making extensive use of inlining include their own configure code.

6.15 Using these functions in your own C code

It is possible to build `Mathlib`, the R set of mathematical functions documented in ‘`Rmath.h`’, as a standalone library ‘`libRmath`’ under both Unix and Windows. (This includes the functions documented in Section 6.7 [Numerical analysis subroutines], page 93 as from that header file.)

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

```
#define MATHLIB_STANDALONE
#include <Rmath.h>
```

and link against ‘`-lRmath`’ (and perhaps ‘`-lm`’. 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 dynamic library or DLL you will have to use the one supplied, which is the Marsaglia-multicarry with an entry points

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

to set its seeds and

```
get_seed(unsigned int *, unsigned int *)
```

to read the seeds).

6.16 Organization of header files

The header files which R installs are in directory ‘*`R_INCLUDE_DIR`*’ (default ‘*`R_HOME`*`/include`’). This currently includes

‘`R.h`’	includes many other files
‘`S.h`’	different version for code ported from S
‘`Rinternals.h`’	definitions for using R’s internal structures
‘`Rdefines.h`’	macros for an S-like interface to the above
‘`Rmath.h`’	standalone math library
‘`Rversion.h`’	R version information
‘`Rinterface.h`’	for add-on front-ends (Unix-alikes only)
‘`Rembedded.h`’	for add-on front-ends
‘`R_ext/Applic.h`’	optimization and integration
‘`R_ext/BLAS.h`’	C definitions for BLAS routines
‘`R_ext/Callbacks.h`’	C (and R function) top-level task handlers
‘`R_ext/GetX11Image.h`’	X11Image interface used by package **trkplot**
‘`R_ext/Lapack.h`’	C definitions for some LAPACK routines
‘`R_ext/Linpack.h`’	C definitions for some LINPACK routines, not all of which are included in R
‘`R_ext/Parse.h`’	a small part of R’s parse interface
‘`R_ext/RConvertors.h`’	
‘`R_ext/Rdynload.h`’	needed to register compiled code in packages
‘`R_ext/R-ftp-http.h`’	interface to internal method of `download.file`
‘`R_ext/Riconv.h`’	interface to `iconv`
‘`R_ext/RStartup.h`’	for add-on front-ends
‘`R_ext/eventloop.h`’	for add-on front-ends and for packages that need to share in the R event loops (on all platforms)

The following headers are included by ‘`R.h`’:

‘`Rconfig.h`’	configuration info that is made available

‘R_ext/Arith.h’	handling for NAs, NaNs, Inf/-Inf
‘R_ext/Boolean.h’	TRUE/FALSE type
‘R_ext/Complex.h’	C typedefs for R’s complex
‘R_ext/Constants.h’	constants
‘R_ext/Error.h’	error handling
‘R_ext/Memory.h’	memory allocation
‘R_ext/Print.h’	Rprintf and variations.
‘R_ext/Random.h’	random number generation
‘R_ext/RS.h’	definitions common to ‘R.h’ and ‘S.h’, including F77_CALL etc.
‘R_ext/Utils.h’	sorting and other utilities
‘R_ext/libextern.h’	definitions for exports from ‘R.dll’ on Windows.

The graphics systems are exposed in headers ‘Rdevices.h’ (for writing graphics devices), ‘Rgraphics.h’ and ‘R_ext/Graphics{Base,Device,Engine}.h’. Some entry points from the **stats** package are in ‘R_ext/stats_package.h’ (currently related to the internals of ‘nls’ and ‘nlminb’).

7 Generic functions and methods

R programmers will often want to add methods for existing generic functions, and may want to add new generic functions or make existing functions generic. In this chapter we give guidelines for doing so, with examples of the problems caused by not adhering to them.

This chapter only covers the 'informal' class system copied from S3, and not with the S4 (formal) methods of package **methods**.

The key function for methods is `NextMethod`, which dispatches the next method. It is quite typical for a method function to make a few changes to its arguments, dispatch to the next method, receive the results and modify them a little. An example is

```
t.data.frame <- function(x)
{
    x <- as.matrix(x)
    NextMethod("t")
}
```

Also consider `predict.glm`: it happens that in R for historical reasons it calls `predict.lm` directly, but in principle (and in S originally and currently) it could use `NextMethod`. (`NextMethod` seems under-used in the R sources. Do be aware that there are S/R differences in this area, and the example above works because there is a *next* method, the default method, not that a new method is selected when the class is changed.)

Any method a programmer writes may be invoked from another method by `NextMethod`, *with the arguments appropriate to the previous method*. Further, the programmer cannot predict which method `NextMethod` will pick (it might be one not yet dreamt of), and the end user calling the generic needs to be able to pass arguments to the next method. For this to work

> *A method must have all the arguments of the generic, including* `...` *if the generic does.*

It is a grave misunderstanding to think that a method needs only to accept the arguments it needs. The original S version of `predict.lm` did not have a `...` argument, although `predict` did. It soon became clear that `predict.glm` needed an argument `dispersion` to handle over-dispersion. As `predict.lm` had neither a `dispersion` nor a `...` argument, `NextMethod` could no longer be used. (The legacy, two direct calls to `predict.lm`, lives on in `predict.glm` in R, which is based on the workaround for S3 written by Venables & Ripley.)

Further, the user is entitled to use positional matching when calling the generic, and the arguments to a method called by `UseMethod` are those of the call to the generic. Thus

> *A method must have arguments in exactly the same order as the generic.*

To see the scale of this problem, consider the generic function `scale`, defined as

```
scale <- function (x, center = TRUE, scale = TRUE)
    UseMethod("scale")
```

Suppose an unthinking package writer created methods such as

```
scale.foo <- function(x, scale = FALSE, ...) { }
```

Then for `x` of class `"foo"` the calls

```
scale(x, , TRUE)
scale(x, scale = TRUE)
```

would do most likely do different things, to the justifiable consternation of the end user.

To add a further twist, which default is used when a user calls `scale(x)` in our example? What if

```
scale.bar <- function(x, center, scale = TRUE) NextMethod("scale")
```

and `x` has class `c("bar", "foo")`? It is the default specified in the method that is used, but the default specified in the generic may be the one the user sees. This leads to the recommendation:

If the generic specifies defaults, all methods should use the same defaults.

An easy way to follow these recommendations is to always keep generics simple, e.g.

```
scale <- function(x, ...) UseMethod("scale")
```

Only add parameters and defaults to the generic if they make sense in all possible methods implementing it.

7.1 Adding new generics

When creating a new generic function, bear in mind that its argument list will be the maximal set of arguments for methods, including those written elsewhere years later. So choosing a good set of arguments may well be an important design issue, and there need to be good arguments *not* to include a `...` argument.

If a `...` argument is supplied, some thought should be given to its position in the argument sequence. Arguments which follow `...` must be named in calls to the function, and they must be named in full (partial matching is suppressed after `...`). Formal arguments before `...` can be partially matched, and so may 'swallow' actual arguments intended for `...`. Although it is commonplace to make the `...` argument the last one, that is not always the right choice.

Sometimes package writers want to make generic a function in the base package, and request a change in R. This may be justifiable, but making a function generic with the old definition as the default method does have a small performance cost. It is never necessary, as a package can take over a function in the base package and make it generic by

```
foo <- function(object, ...) UseMethod("foo")
foo.default <- base::foo
```

(If the thus defined default method needs a '`...`' added to its argument list, one can e.g. use `formals(foo.default) <- c(formals(foo.default), alist(... = ))`.)

The same idea can be applied for functions in other packages with name spaces.

8 Linking GUIs and other front-ends to R

There are a number of ways to build front-ends to R: we take this to mean a GUI or other application that has the ability to submit commands to R and perhaps to receive results back (not necessarily in a text format). There are other routes besides those described here, for example the package **Rserve** (from CRAN, see also http://www.rforge.net/Rserve/) and connections to Java in 'SJava' (see http://www.omegahat.org/RSJava/ and 'JRI', part of the **rJava** package on CRAN).

8.1 Embedding R under Unix-alikes

R can be built as a shared library[1] if configured with '--enable-R-shlib'. This shared library can be used to run R from alternative front-end programs. We will assume this has been done for the rest of this section.

The command-line R front-end, '*R_HOME*/bin/exec/R' is one such example, and the unbundled GNOME (see package **gnomeGUI** on CRAN) and MacOS X consoles are others. The source for '*R_HOME*/bin/exec/R' is in file 'src/main/Rmain.c' and is very simple

```
int Rf_initialize_R(int ac, char **av); /* in ../unix/system.c */
void Rf_mainloop();                     /* in main.c */

extern int R_running_as_main_program;   /* in ../unix/system.c */

int main(int ac, char **av)
{
    R_running_as_main_program = 1;
    Rf_initialize_R(ac, av);
    Rf_mainloop(); /* does not return */
    return 0;
}
```

indeed, misleadingly simple. Remember that '*R_HOME*/bin/exec/R' is run from a shell script '*R_HOME*/bin/R' which sets up the environment for the executable, and this is used for

- Setting R_HOME and checking it is valid, as well as the path R_SHARE_DIR and R_DOC_DIR to the installed 'share' and 'doc' directory trees. Also setting R_ARCH if needed.
- Setting LD_LIBRARY_PATH to include the directories used in linking R. This is recorded as the default setting of R_LD_LIBRARY_PATH in the shell script 'R_HOME/etc*R_ARCH*/ldpaths'.
- Processing some of the arguments, for example to run R under a debugger and to launch alternative front-ends to provide GUIs.

The first two of these can be achieved for your front-end by running it *via* R CMD. So, for example

```
R CMD /usr/local/lib/R/bin/exec/R
R CMD exec/R
```

will both work in a standard R installation. (R CMD looks first for executables in '*R_HOME*/bin'.) If you do not want to run your front-end in this way, you need to ensure that R_HOME is set and LD_LIBRARY_PATH is suitable. (The latter might well be, but modern Unix/Linux systems do not normally include '/usr/local/lib' ('/usr/local/lib64' on some architectures), and R does look there for system components.)

The other senses in which this example is too simple are that all the internal defaults are used and that control is handed over to the R main loop. There are a number of small

[1] In the parlance of MacOS X this is a *dynamic* library, and is the normal way to build R on that platform.

examples[2] in the 'tests/Embedding' directory. These make use of Rf_initEmbeddedR in 'src/main/Rembedded.c', and essentially use

```
#include <Rembedded.h>

int main(int ac, char **av)
{
    /* do some setup */
    Rf_initEmbeddedR(argc, argv);
    /* do some more setup */

    /* submit some code to R, which is done interactively via
        run_Rmainloop();

        A possible substitute for a pseudo-console is

        R_ReplDLLinit();
        while(R_ReplDLLdo1() > 0) {
        /* add user actions here if desired */
       }

     */
    Rf_endEmbeddedR(0);
    /* final tidying up after R is shutdown */
    return 0;
}
```

If you don't want to pass R arguments, you can fake an argv array, for example by

```
    char *argv[]= {"REmbeddedPostgres", "--silent"};
    Rf_initEmbeddedR(sizeof(argv)/sizeof(argv[0]), argv);
```

However, to make a GUI we usually do want to run run_Rmainloop after setting up various parts of R to talk to our GUI, and arranging for our GUI callbacks to be called during the R mainloop.

One issue to watch is that on some platforms Rf_initEmbeddedR and Rf_endEmbeddedR change the settings of the FPU (e.g. to allow errors to be trapped and to set extended precision registers).

The standard code sets up a session temporary directory in the usual way, *unless* R_TempDir is set to a non-NULL value before Rf_initEmbeddedR is called. In that case the value is assumed to contain an existing writable directory (no check is done), and it is not cleaned up when R is shut down.

Rf_initEmbeddedR sets R to be in interactive mode: you can set R_Interactive (defined in 'Rinterface.h') subsequently to change this.

8.1.1 Compiling against the R shared library

Suitable flags to compile and link against the R shared library can be found by

```
R CMD config --cppflags
R CMD config --ldflags
```

If R is installed, pkg-config is available and sub-architectures have not be used, alternatives are

[2] but these are not part of the automated test procedures and so little tested.

```
pkg-config --cflags libR
pkg-config --libs libR
```

8.1.2 Setting R callbacks

For Unix-alkes there is a public header file '`Rinterface.h`' that makes it possible to change the standard callbacks used by R in a documented way. This defines pointers (if `R_INTERFACE_PTRS` is defined)

```
extern void (*ptr_R_Suicide)(char *);
extern void (*ptr_R_ShowMessage)(char *);
extern int  (*ptr_R_ReadConsole)(char *, unsigned char *, int, int);
extern void (*ptr_R_WriteConsole)(char *, int);
extern void (*ptr_R_WriteConsoleEx)(char *, int, int);
extern void (*ptr_R_ResetConsole)();
extern void (*ptr_R_FlushConsole)();
extern void (*ptr_R_ClearerrConsole)();
extern void (*ptr_R_Busy)(int);
extern void (*ptr_R_CleanUp)(SA_TYPE, int, int);
extern int  (*ptr_R_ShowFiles)(int, char **, char **, char *,
                               Rboolean, char *);
extern int  (*ptr_R_ChooseFile)(int, char *, int);
extern int  (*ptr_R_EditFile)(char *);
extern void (*ptr_R_loadhistory)(SEXP, SEXP, SEXP, SEXP);
extern void (*ptr_R_savehistory)(SEXP, SEXP, SEXP, SEXP);
extern void (*ptr_R_addhistory)(SEXP, SEXP, SEXP, SEXP);
```

which allow standard R callbacks to be redirected to your GUI. What these do is generally documented in the file '`src/unix/system.txt`'.

`void R_ShowMessage` (*char *`message`*) [Function]

This should display the message, which may have multiple lines: it should be brought to the user's attention immediately.

`void R_Busy` (*int `which`*) [Function]

This function invokes actions (such as change of cursor) when R embarks on an extended computation (*`which`*`=1`) and when such a state terminates (*`which`*`=0`).

`int R_ReadConsole` (*char *`prompt`, unsigned char *`buf`, int `buflen`, int `hist`*) [Function]
`void R_WriteConsole` (*char *`buf`, int `buflen`*) [Function]
`void R_WriteConsoleEx` (*char *`buf`, int `buflen`, int `otype`*) [Function]
`void R_ResetConsole` () [Function]
`void R_FlushConsole` () [Function]
`void R_ClearErrConsole` () [Function]

These functions interact with a console.

`R_ReadConsole` prints the given prompt at the console and then does a `gets(3)`–like operation, transferring up to *buflen* characters into the buffer *buf*. The last two bytes should be set to '`"\n\0"`' to preserve sanity. If *hist* is non-zero, then the line should be added to any command history which is being maintained. The return value is 0 is no input is available and >0 otherwise.

`R_WriteConsoleEx` writes the given buffer to the console, *otype* specifies the output type (regular output or warning/error). Call to `R_WriteConsole(buf, buflen)` is equivalent to `R_WriteConsoleEx(buf, buflen, 0)`. To ensure backward compatibility of the callbacks, `ptr_R_WriteConsoleEx` is used only if `ptr_R_WriteConsole` is set to `NULL`. To ensure that `stdout()` and `stderr()` connections point to the console, set the corresponding files to `NULL` *via*

```
R_Outputfile = NULL;
R_Consolefile = NULL;
```

`R_ResetConsole` is called when the system is reset after an error. `R_FlushConsole` is called to flush any pending output to the system console. `R_ClearerrConsole` clears any errors associated with reading from the console.

`int R_ShowFiles` (*int* `nfile`, *char* `**file`, *char* `**headers`, *char* `*wtitle`, *Rboolean* `del`, *char* `*pager`) [Function]

This function is used to display the contents of files.

`int R_ChooseFile` (*int* `new`, *char* `*buf`, *int* `len`) [Function]

Choose a file and return its name in *buf* of length *len*. Return value is 0 for success, `>` 0 otherwise.

`int R_EditFile` (*char* `*buf`) [Function]

Send a file to an editor window.

`SEXP R_loadhistory` (*SEXP, SEXP, SEXP, SEXP);* [Function]
`SEXP R_savehistory` (*SEXP, SEXP, SEXP, SEXP);* [Function]
`SEXP R_addhistory` (*SEXP, SEXP, SEXP, SEXP);* [Function]

`.Internal` functions for `loadhistory`, `savehistory` and `timestamp`: these are called after checking the number of arguments.

If the console has no history mechanism these can be as simple as

```
SEXP R_loadhistory (SEXP call, SEXP op, SEXP args, SEXP env)
{
    errorcall(call, "loadhistory is not implemented");
    return R_NilValue;
}
SEXP R_savehistory (SEXP call, SEXP op , SEXP args, SEXP env)
{
    errorcall(call, "savehistory is not implemented");
    return R_NilValue;
}
SEXP R_addhistory (SEXP call, SEXP op , SEXP args, SEXP env)
{
    return R_NilValue;
}
```

The `R_addhistory` function should return silently if no history mechanism is present, as a user may be calling `timestamp` purely to write the time stamp to the console.

`void R_Suicide` (*char* `*message`) [Function]

This should abort R as rapidly as possible, displaying the message. A possible implementation is

```
void R_Suicide (char *message)
{
    char  pp[1024];
    snprintf(pp, 1024, "Fatal error: %s\n", s);
    R_ShowMessage(pp);
    R_CleanUp(SA_SUICIDE, 2, 0);
}
```

void R_CleanUp (*SA_TYPE* **saveact**, *int* **status**, *int* **RunLast**) [Function]

This function invokes any actions which occur at system termination. It needs to be quite complex:

```
#include <Rinterface.h>
#include <Rdevices.h>    /* for KillAllDevices */

void R_CleanUp (SA_TYPE saveact, int status, int RunLast)
{
    if(saveact == SA_DEFAULT) saveact = SaveAction;
    if(saveact == SA_SAVEASK) {
       /* ask what to do and set saveact */
    }
    switch (saveact) {
    case SA_SAVE:
        if(runLast) R_dot_Last();
        if(R_DirtyImage) R_SaveGlobalEnv();
        /* save the console history in R_HistoryFile */
        break;
    case SA_NOSAVE:
        if(runLast) R_dot_Last();
        break;
    case SA_SUICIDE:
    default:
        break;
    }

    R_RunExitFinalizers();
    /* clean up after the editor e.g. CleanEd() */

    R_CleanTempDir();

    /* close all the graphics devices */
    if(saveact != SA_SUICIDE) KillAllDevices();
    fpu_setup(FALSE);

    exit(status);
}
```

8.1.3 Registering symbols

An application embedding R needs a different way of registering symbols because it is not a dynamic library loaded by R as would be the case with a package. Therefore R reserves a special `DllInfo` entry for the embedding application such that it can register symbols to be used with `.C`, `.Call` etc. This entry can be obtained by calling `getEmbeddingDllInfo`, so a typical use is

```
DllInfo *info = R_getEmbeddingDllInfo();
R_registerRoutines(info, cMethods, callMethods, NULL, NULL);
```

The native routines defined by `cMethod` and `callMethods` should be present in the embedding application. See Section 5.4 [Registering native routines], page 62 for details on registering symbols in general.

8.1.4 Meshing event loops

One of the most difficult issues in interfacing R to a front-end is the handling of event loops, at least if a single thread is used. R uses events and timers for

- Running X11 windows such as the graphics device and data editor, and interacting with them (e.g., using `locator()`).
- Supporting Tcl/Tk events for the **tcltk** package.
- Preparing input.
- Timing operations, for example for profiling R code and `Sys.sleep()`.
- Interrupts, where permitted.

Specifically, the Unix command-line version of R runs separate event loops for

- Preparing input at the console command-line, in file '`src/unix/sys-unix.c`'.
- Waiting for a response from a socket in the internal functions underlying FTP and HTTP transfers in `download.file()` and for direct socket access, in files '`src/modules/internet/nanoftp.c`', '`src/modules/internet/nanohttp.c`' and '`src/modules/internet/Rsock.c`'
- Mouse and window events when displaying the X11-based dataentry window, in file '`src/modules/X11/dataentry.c`'. This is regarded as *modal*, and no other events are serviced whilst it is active.

There is a protocol for adding event handlers to the first two types of event loops, using types and functions declared in the header '`R_ext/eventloop.h`' and described in comments in file '`src/unix/sys-std.c`'. It is possible to add (or remove) an input handler for events on a particular file descriptor, or to set a polling interval (*via* `R_wait_usec`) and a function to be called periodically via `R_PolledEvents`: the polling mechanism is used by the **tcltk** package.

An alternative front-end needs both to make provision for other R events whilst waiting for input, and to ensure that it is not frozen out during events of the second type. This is not handled very well in the existing examples. The GNOME front-end can run a own handler for polled events by setting

```
extern int (*R_timeout_handler)();
extern long R_timeout_val;

        if (R_timeout_handler && R_timeout_val)
            gtk_timeout_add(R_timeout_val, R_timeout_handler, NULL);
        gtk_main ();
```

whilst it is waiting for console input. This obviously handles events for Gtk windows (such as the graphics device in the **gtkDevice** package), but not X11 events (such as the `X11()` device) or for other event handlers that might have been registered with R. It does not attempt to keep itself alive whilst R is waiting on sockets. The ability to add a polled handler as `R_timeout_handler` is used by the **tcltk** package.

8.1.5 Threading issues

Embedded R is designed to be run in the main thread, and all the testing is done in that context. There is a potential issue with the stack-checking mechanism where threads are involved. This uses two variables declared in '`Rinterface.h`' (if `CSTACK_DEFNS` is defined) as

```
extern uintptr_t R_CStackLimit; /* C stack limit */
extern uintptr_t R_CStackStart; /* Initial stack address */
```

Note that `uintptr_t` is a C99 type for which a substitute is defined in R, so your code needs to define `HAVE_UINTPTR_T` appropriately.

These will be set[3] when `R_initialize_R` is called, to values appropriate to the main thread. Stack-checking can be disabled by seting `R_CStackLimit = (uintptr_t)-1`, but it is better to if possible set appropriate values. (What these are and how to determine them are OS-specific, and the stack size limit may differ for secondary threads. If you have a choice of stack size, at least 8Mb is recommended.)

You may also want to consider how signals are handled: R sets signal handlers for several signals, including `SIGINT`, `SIGSEGV`, `SIGPIPE`, `SIGUSR1` and `SIGUSR2`, but these can all be suppressed by setting the variable `R_SignalHandlers` (declared in '`Rinterface.h`') to `0`.

8.2 Embedding R under Windows

All Windows interfaces to R call entry points in the DLL '`R.dll`', directly or indirectly. Simpler applications may find it easier to use the indirect route via (D)COM.

8.2.1 Using (D)COM

(D)COM is a standard Windows mechanism used for communication between Windows applications. One application (here R) is run as COM server which offers services to clients, here the front-end calling application. The services are described in a 'Type Library' and are (more or less) language-independent, so the calling application can be written in C or C++ or Visual Basic or Perl or Python and so on. The 'D' in (D)COM refers to 'distributed', as the client and server can be running on different machines.

The basic R distribution is not a (D)COM server, but two addons are currently available that interface directly with R and provide a (D)COM server:

- There is a (D)COM server called `StatConnector` written by Thomas Baier available on CRAN (http://cran.r-project.org/other-software.html) which works with '`Rproxy.dll`' (in the R distribution) and '`R.dll`' to support transfer of data to and from R and remote execution of R commands, as well as embedding of an R graphics window. The **rcom** package on CRAN provides a (D)COM server in a running R session.
- Another (D)COM server, `RDCOMServer`, is available from http://www.omegahat.org/. Its philosophy is discussed in http://www.omegahat.org/RDCOMServer/Docs/Paradigm.html and is very different from the purpose of this section.

8.2.2 Calling R.dll directly

The `R` DLL is mainly written in C and has `_cdecl` entry points. Calling it directly will be tricky except from C code (or C++ with a little care).

There is a version of the Unix interface callng

```
int Rf_initEmbeddedR(int ac, char **av);
void Rf_endEmbeddedR(int fatal);
```

which is an entry point in '`R.dll`'. Examples of its use (and a suitable '`Makefile.win`') can be found in the '`tests/Embedding`' directory of the sources. You may need to ensure that '*`R_HOME`*`/bin`' is in your `PATH` so the R DLLs are found.

Examples of calling '`R.dll`' directly are provided in the directory '`src/gnuwin32/front-ends`', including '`Rproxy.dll`' used by `StatConnector` and a simple command-line front end '`rtest.c`' whose code is

```
#define Win32
#include <windows.h>
#include <stdio.h>
#include <Rversion.h>
```

[3] at least on platforms where the values are available, that is having `getrlimit` and on Linux or having `sysctl` supporting `KERN_USRSTACK`, including FreeBSD and MacOS X.

```
#define LibExtern __declspec(dllimport) extern
#include <Rembedded.h>
#include <R_ext/RStartup.h>
/* for askok and askyesnocancel */
#include <graphapp/graphapp.h>

/* for signal-handling code */
#include <psignal.h>

/* simple input, simple output */

/* This version blocks all events: a real one needs to call ProcessEvents
   frequently. See rterm.c and ../system.c for one approach using
   a separate thread for input.
*/
int myReadConsole(char *prompt, char *buf, int len, int addtohistory)
{
    fputs(prompt, stdout);
    fflush(stdout);
    if(fgets(buf, len, stdin)) return 1; else return 0;
}

void myWriteConsole(char *buf, int len)
{
    printf("%s", buf);
}

void myCallBack()
{
    /* called during i/o, eval, graphics in ProcessEvents */
}

void myBusy(int which)
{
    /* set a busy cursor ... if which = 1, unset if which = 0 */
}

static void my_onintr(int sig) { UserBreak = 1; }

int main (int argc, char **argv)
{
    structRstart rp;
    Rstart Rp = &rp;
    char Rversion[25], *RHome;

    sprintf(Rversion, "%s.%s", R_MAJOR, R_MINOR);
    if(strcmp(getDLLVersion(), Rversion) != 0) {
        fprintf(stderr, "Error: R.DLL version does not match\n");
        exit(1);
    }

    R_setStartTime();
    R_DefParams(Rp);
    if((RHome = get_R_HOME()) == NULL) {
         fprintf(stderr, "R_HOME must be set in the environment or Registry\n");
         exit(1);
    }
    Rp->rhome = RHome;
    Rp->home = getRUser();
    Rp->CharacterMode = LinkDLL;
    Rp->ReadConsole = myReadConsole;
    Rp->WriteConsole = myWriteConsole;
    Rp->CallBack = myCallBack;
    Rp->ShowMessage = askok;
    Rp->YesNoCancel = askyesnocancel;
```

```
    Rp->Busy = myBusy;

    Rp->R_Quiet = TRUE;        /* Default is FALSE */
    Rp->R_Interactive = FALSE; /* Default is TRUE */
    Rp->RestoreAction = SA_RESTORE;
    Rp->SaveAction = SA_NOSAVE;
    R_SetParams(Rp);
    R_set_command_line_arguments(argc, argv);

    FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));

    signal(SIGBREAK, my_onintr);
    GA_initapp(0, 0);
    readconsolecfg();
    setup_Rmainloop();
#ifdef SIMPLE_CASE
    run_Rmainloop();
#else
    R_ReplDLLinit();
    while(R_ReplDLLdo1() > 0) {
/* add user actions here if desired */
    }
/* only get here on EOF (not q()) */
#endif
    Rf_endEmbeddedR(0);
    return 0;
}
```

The ideas are

- Check that the front-end and the linked '`R.dll`' match – other front-ends may allow a looser match.
- Find and set the R home directory and the user's home directory. The former may be available from the Windows Registry: it will normally be in `HKEY_LOCAL_MACHINE\Software\R-core\R\InstallPath` and can be set there by running the program '*R_HOME*`\bin\RSetReg.exe`'.
- Define startup conditions and callbacks via the `Rstart` structure. `R_DefParams` sets the defaults, and `R_SetParams` sets updated values.
- Record the command-line arguments used by `R_set_command_line_arguments` for use by the R function `commandArgs()`.
- Set up the signal handler and the basic user interface.
- Run the main R loop, possibly with our actions intermeshed.
- Arrange to clean up.

An underlying theme is the need to keep the GUI 'alive', and this has not been done in this example. The R callback `R_ProcessEvents` needs to be called frequently to ensure that Windows events in R windows are handled expeditiously. Conversely, R needs to allow the GUI code (which is running in the same process) to update itself as needed – two ways are provided to allow this:

- `R_ProcessEvents` calls the callback registered by `Rp->callback`. A version of this is used to run package Tcl/Tk for **tcltk** under Windows, for the code is

```
void R_ProcessEvents(void)
{
    while (peekevent()) doevent(); /* Windows events for GraphApp */
    if (UserBreak) { UserBreak = FALSE; onintr(); }
    R_CallBackHook();
    if(R_tcldo) R_tcldo();
}
```

- The mainloop can be split up to allow the calling application to take some action after each line of input has been dealt with: see the alternative code below `#ifdef SIMPLE_CASE`.

It may be that no R GraphApp windows need to be considered, although these include pagers, the `windows()` graphics device, the R data and script editors and various popups such as `choose.file()` and `select.list()`. It would be possible to replace all of these, but it seems easier to allow GraphApp to handle most of them.

It is possible to run R in a GUI in a single thread (as '`RGui.exe`' shows) but it will normally be easier[4] to use multiple threads.

Note that R's own front ends use a stack size of 10Mb, whereas MinGW executables default to 2Mb, and Visual C++ ones to 1Mb. The latter stack sizes are too small for a number of R applications, so general-purpose front-ends should use a larger stack size.

[4] An attempt to use only threads in the late 1990s failed to work correctly under Windows 95, the predominant version of Windows at that time.

Function and variable index

*

*Riconv_open 100

.

.C 59
.Call 68, 77
.External 68, 78
.First.lib 6
.Fortran 59
.Last.lib 6, 21
.onAttach 21
.onLoad 21
.onUnload 21
.Random.seed 91

\

\acronym 38
\alias 32
\arguments 33
\author 34
\bold 37
\cite 38
\code 37
\command 38
\concept 40
\cr 37
\deqn 39
\describe 38
\description 32
\details 34
\dfn 38
\dontrun 34
\dontshow 34
\dots 40
\dQuote 37
\email 37
\emph 37
\enc 40
\enumerate 38
\env 38
\eqn 39
\examples 34
\file 37
\format 35
\itemize 38
\kbd 37
\keyword 35
\ldots 40
\link 39
\method 33
\name 32
\note 34
\option 38
\pkg 37
\preformatted 37
\R 40
\references 34
\samp 37
\section 37
\seealso 34
\source 36
\sQuote 37
\strong 37
\tabular 38
\title 32
\url 38
\usage 32
\value 34
\var 38

B

bessel_i 95
bessel_j 95
bessel_k 95
bessel_y 95
beta 95
BLAS_LIBS 11
browser 49

C

Calloc 90
CAR 78
CDR 78
cgmin 97
choose 95
cPsort 99

D

debug 52
debugger 51
defineVar 75
digamma 94
dump.frames 51
duplicate 76
dyn.load 61
dyn.unload 61

E

exp_rand 91
expm1 95
export 20
exportClasses 24
exportMethods 24
exportPattern 20

F

FALSE 97
findInterval 99
findVar 75
FLIBS 11
fmax2 96
fmin2 96
fprec 96
Free 90
fround 96

fsign 96
ftrunc 96

G

gammafn 94
gctorture 54
getAttrib 73
GetRNGstate 91

H

hsv2rgb 100

I

imax2 96
imin2 96
import 20
importClassesFrom 24
importFrom 20
importMethodsFrom 24
install 73
iPsort 99
ISNA 80, 91
ISNAN 80, 91

L

LAPACK_LIBS 11
lbeta 95
lbfgsb 97
lchoose 95
lgamma1p 95
lgammafn 94
library.dynam 6, 61
log1p 95
log1pmx 95
logspace_add 96
logspace_sub 96

M

M_E 96
M_PI 96

N

NA_REAL 91
nmmin 97
norm_rand 91

O

OBJECTS 11, 64

P

pentagamma 94, 95
PKG_CFLAGS 64
PKG_CPPFLAGS 64
PKG_CXXFLAGS 64
PKG_FCFLAGS 64
PKG_FFLAGS 64
PKG_LIBS 64
PKG_OBJCFLAGS 64
prompt 35
PROTECT 69
PROTECT_WITH_INDEX 70
psigamma 94, 95
PutRNGstate 91
pythag 95

Q

qsort3 99
qsort4 99

R

R CMD build 16
R CMD check 14
R CMD config 10
R CMD Rd2dvi 42
R CMD Rd2txt 42
R CMD Rdconv 41
R CMD Sd2Rd 42
R CMD SHLIB 64
R CMD Stangle 42
R CMD Sweave 42
R_addhistory 109
R_alloc 89
R_Busy 108
R_ChooseFile 109
R_CleanUp 110
R_ClearErrConsole 108
R_csort 99
R_EditFile 109
R_ExpandFileName 100
R_FINITE 91
R_FlushConsole 108
R_GetCCallable 64
R_INLINE 101
R_IsNaN 91
R_isort 99
R_LIBRARY_DIR 10
R_loadhistory 109
R_max_col 99
R_NegInf 91
R_PACKAGE_DIR 10
R_ParseVector 86
R_PosInf 91
R_pow 95
R_pow_di 95
R_qsort 99
R_qsort_I 99
R_qsort_int 99
R_qsort_int_I 99
R_ReadConsole 108
R_RegisterCCallable 64
R_ResetConsole 108
R_rsort 99
R_savehistory 109
R_ShowFiles 109
R_ShowMessage 108
R_Suicide 109
R_tmpnam 100
R_Version 101

R_WriteConsole 108
R_WriteConsoleEx 108
rcont2 91
Rdqagi 98
Rdqags 98
Realloc 90
recover 52
REprintf 92
REPROTECT 70
REvprintf 92
revsort 99
rgb2hsv 100
Riconv 100
Riconv_close 100
Rprintf 92
Rprof 43, 45
Rprofmem 46
rPsort 99
rsort_with_index 99
Rvprintf 92

S

S_alloc 89
S_realloc 89
S3method 21
SAFE_FFLAGS 11
samin 97
seed_in 91
seed_out 91
setAttrib 73
setVar 75
sign 96
summaryRprof 45
system 59
system.time 59

T

tetragamma 94, 95
trace 53
traceback 50
tracemem 46
translateChar 88
trigamma 94
TRUE 97

U

undebug 53
unif_rand 91
UNPROTECT 69
UNPROTECT_PTR 70
untracemem 46
useDynLib 21

V

vmaxget 89
vmaxset 89
vmmin 97

Concept index

\

\linkS4class 39

A

Allocating storage 70
Attributes 71

B

Bessel functions 95
Beta function 95
Building packages 16

C

C++ code, interfacing 65
Calling C from FORTRAN and vice versa 92
Checking packages 14
Classes 73
cleanup file 2
configure file 2
COPYING file 2
Copying objects 76
CRAN 19
CRAN submission 19
Creating packages 2
Creating shared objects 64
Cross-references in documentation 39
cumulative hazard 94

D

Debugging 55
DESCRIPTION file 3
Details of R types 70
Distribution functions from C 93
Documentation, writing 31
Dynamic loading 61

E

encoding 41
Error handling from C 90
Error handling from FORTRAN 91
Evaluating R expressions from C 80
external pointer 87

F

finalizer 87
Finding variables 75

G

Gamma function 94
Garbage collection 69
Generic functions 104

H

handling character data 74
Handling lists 74
Handling R objects in C 67

I

IEEE special values 80, 91
INDEX file 5
Indices 40
Inspecting R objects when debugging 57
integration 98
Interfaces to compiled code 59, 77
Interfacing C++ code 65
Interrupts 101

L

LICENCE file 2
LICENSE file 2
Lists and tables in documentation 38

M

Marking text in documentation 37
Mathematics in documentation 39
Memory allocation from C 89
Memory use 45
Method functions 104
Missing values 80, 91

N

name spaces 19
Numerical analysis subroutines from C 93
Numerical derivatives 83

O

Operating system access 59
optimization 97

P

Package builder 16
Package bundles 8
Package structure 2
Package subdirectories 6
Packages 2
Parsing R code from C 86
Platform-specific documentation 41
Printing from C 92
Printing from FORTRAN 92
Processing Rd format 41
Profiling 43, 45, 46

R

Random numbers in C 91, 94

Random numbers in FORTRAN 93
Registering native routines 62

S

Setting variables 75
Sort functions from C 99
Submitting to CRAN 19
Sweave 18

T

Tidying R code 43

V

Version information from C 101
vignettes 18

W

weak reference 87

Z

Zero-finding 82