# 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 ...................................... 8
      1.1.3 Package subdirectories .................................. 8
      1.1.4 Package bundles .......................................... 11
      1.1.5 Data in packages ......................................... 11
      1.1.6 Non-R scripts in packages .............................. 12
   1.2 Configure and cleanup ........................................ 13
      1.2.1 Using ‘Makevars’ ......................................... 15
         1.2.1.1 OpenMP support .................................... 18
      1.2.2 Configure example ....................................... 19
      1.2.3 Using F95 code ........................................... 21
   1.3 Checking and building packages .............................. 21
      1.3.1 Checking packages ....................................... 22
      1.3.2 Building packages ....................................... 24
   1.4 Writing package vignettes ................................... 26
      1.4.1 Encodings and vignettes ................................ 28
   1.5 Submitting a package to CRAN ................................ 29
   1.6 Package name spaces .......................................... 31
      1.6.1 Specifying imports and exports ........................ 31
      1.6.2 Registering S3 methods .................................. 32
      1.6.3 Load hooks ................................................ 33
      1.6.4 An example ................................................. 35
      1.6.5 Summary – converting an existing package ............ 36
      1.6.6 Name spaces with S4 classes and methods ............. 36
   1.7 Writing portable packages ..................................... 37
      1.7.1 Encoding issues .......................................... 39
      1.7.2 Binary distribution ...................................... 40
   1.8 Diagnostic messages ........................................... 40
   1.9 Internationalization .......................................... 42
      1.9.1 C-level messages ........................................ 42
      1.9.2 R messages ................................................ 43
      1.9.3 Installing translations ................................... 43
      1.9.4 Makefile support ......................................... 43
   1.10 CITATION files ................................................. 44
   1.11 Package types ................................................ 45
      1.11.1 Frontend ................................................ 45
      1.11.2 Translation .............................................. 45
   1.12 Services ....................................................... 46
# Writing R documentation files

2.1 Rd format .......................................................... 47
  2.1.1 Documenting functions .................................... 47
  2.1.2 Documenting data sets .................................. 48
  2.1.3 Documenting S4 classes and methods .................. 54
  2.1.4 Documenting packages .................................. 54

2.1.2 Documenting data sets .................................. 53

2.1.3 Documenting S4 classes and methods .................. 54

2.1.4 Documenting packages .................................. 54

2.2 Sectioning ...................................................... 55

2.3 Marking text ..................................................... 55

2.4 Lists and tables ................................................. 57

2.5 Cross-references ................................................ 58

2.6 Mathematics ...................................................... 59

2.7 Insertions .......................................................... 60

2.8 Indices ............................................................. 60

2.9 Platform-specific documentation .............................. 61

2.10 Conditional text .................................................. 61

2.11 Dynamic pages ................................................... 61

2.12 User-defined macros ............................................. 62

2.13 Encoding .......................................................... 63

2.14 Processing Rd format ......................................... 64

# Tidying and profiling R code

3.1 Tidying R code ..................................................... 65

3.2 Profiling R code for speed ..................................... 65

3.3 Profiling R code for memory use ................................. 67
  3.3.1 Memory statistics from \texttt{Rprof} ....................... 68
  3.3.2 Tracking memory allocations .............................. 68
  3.3.3 Tracing copies of an object ............................... 68

3.4 Profiling compiled code ........................................... 69
  3.4.1 Linux .......................................................... 69
    3.4.1.1 sprof .............................................. 69
    3.4.1.2 oprofile ............................................ 70
  3.4.2 Solaris ...................................................... 71
  3.4.3 Mac OS X .................................................... 71

# Debugging

4.1 Browsing .......................................................... 72

4.2 Debugging R code .................................................. 73

4.3 Using gettorture and valgrind ................................. 77
  4.3.1 Using gettorture ......................................... 77
  4.3.2 Using valgrind ........................................... 78

4.4 Debugging compiled code ....................................... 79
  4.4.1 Finding entry points in dynamically loaded code .... 81
  4.4.2 Inspecting R objects when debugging ................. 81
5 System and foreign language interfaces...... 84
  5.1 Operating system access ........................................ 84
  5.2 Interface functions .C and .Fortran ............................ 84
  5.3 dyn.load and dyn.unload ...................................... 86
  5.4 Registering native routines .................................... 88
  5.5 Creating shared objects ......................................... 90
  5.6 Interfacing C++ code ............................................ 92
  5.7 Fortran I/O ................................................................ 94
  5.8 Linking to other packages ......................................... 94
    5.8.1 Unix-alikes ...................................................... 94
    5.8.2 Windows .......................................................... 95
  5.9 Handling R objects in C .......................................... 96
    5.9.1 Handling the effects of garbage collection ................ 98
    5.9.2 Allocating storage ............................................. 99
    5.9.3 Details of R types ............................................. 99
    5.9.4 Attributes ....................................................... 101
    5.9.5 Classes .......................................................... 103
    5.9.6 Handling lists .................................................. 103
    5.9.7 Handling character data ..................................... 104
    5.9.8 Finding and setting variables ................................. 104
    5.9.9 Some convenience functions .................................. 105
      5.9.9.1 Semi-internal convenience functions ................... 106
    5.9.10 Named objects and copying ................................ 106
  5.10 Interface functions .Call and .External ....................... 107
    5.10.1 Calling .Call .................................................. 107
    5.10.2 Calling .External ............................................ 108
    5.10.3 Missing and special values ................................ 110
  5.11 Evaluating R expressions from C ................................ 110
    5.11.1 Zero-finding ................................................... 112
    5.11.2 Calculating numerical derivatives .......................... 113
  5.12 Parsing R code from C ............................................ 116
  5.13 External pointers and weak references ......................... 117
    5.13.1 An example ..................................................... 119
  5.14 Vector accessor functions ....................................... 120
  5.15 Character encoding issues ...................................... 120

6 The R API: entry points for C code ........ 121
  6.1 Memory allocation ............................................... 121
    6.1.1 Transient storage allocation ................................ 122
    6.1.2 User-controlled memory .................................... 122
  6.2 Error handling ..................................................... 123
    6.2.1 Error handling from FORTRAN ............................... 123
  6.3 Random number generation ...................................... 123
  6.4 Missing and IEEE special values ................................. 124
  6.5 Printing ............................................................. 124
    6.5.1 Printing from FORTRAN ...................................... 124
  6.6 Calling C from FORTRAN and vice versa ....................... 125
  6.7 Numerical analysis subroutines ................................ 126
6.7.1 Distribution functions ........................................... 126
6.7.2 Mathematical functions ........................................ 127
6.7.3 Numerical Utilities .................................................. 128
6.7.4 Mathematical constants ........................................... 129
6.8 Optimization ............................................................. 130
6.9 Integration ................................................................. 131
6.10 Utility functions .......................................................... 132
6.11 Re-encoding ............................................................... 134
6.12 Allowing interrupts ..................................................... 134
6.13 Platform and version information .................................. 135
6.14 Inlining C functions .................................................... 135
6.15 Controlling visibility .................................................... 136
6.16 Using these functions in your own C code ...................... 136
6.17 Organization of header files ........................................ 137

7  Generic functions and methods ............... 139
7.1  Adding new generics ................................................... 140

8  Linking GUIs and other front-ends to R ... 141
8.1  Embedding R under Unix-alikes ............................... 141
  8.1.1 Compiling against the R library ................................ 143
  8.1.2 Setting R callbacks ............................................... 143
  8.1.3 Registering symbols ............................................. 146
  8.1.4 Meshing event loops ............................................. 146
  8.1.5 Threading issues .................................................. 147
8.2  Embedding R under Windows ..................................... 148
  8.2.1 Using (D)COM ..................................................... 148
  8.2.2 Calling R.dll directly .......................................... 148
  8.2.3 Finding R_HOME ................................................... 151

Function and variable index ......................... 153

Concept index ......................................................... 156
Acknowledgements

The contributions of Saikat DebRoy (who wrote the first draft of a guide to using \texttt{.Call} and \texttt{.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 itself includes about 25 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

```r
?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 Mac OS 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 (but rare): See Section 1.11 [Package types], page 45.

1.1 Package structure

The sources of a R 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, but which should not be empty). The package subdirectory may also contain files `INDEX`, `NAMESPACE`, `configure`, `cleanup`, `LICENSE`, `LICENCE`, `COPYING` and `NEWS`. Other files such as `INSTALL` (for non-standard installation instructions), `README` or `ChangeLog` will be ignored by R, but may be useful to end-users.

The `DESCRIPTION` and `INDEX` files are described in the subsections below. The `NAMESPACE` file is described in the section on Section 1.6 [Package name spaces], page 31.

The optional files `configure` and `cleanup` are (Bourne shell) script files which are, respectively, executed before and (provided that option `--clean` was given) after installation on Unix-alikes, see Section 1.2 [Configure and cleanup], page 13. The analogues on Windows are `configure.win` and `cleanup.win`.

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 and by default on Mac OS X) 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$’, ‘null’, ‘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 (\{}\[].

A source package if possible should not contain binary executable files: they are not portable, and a security risk if they are of the appropriate architecture. R CMD check will warn about them unless they are listed (one filepath per line) in a file ‘BinaryFiles’ at the top level of the package. Note that CRAN will no longer accept submissions containing binary files even if they are listed.

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@R: c(person("Joe", "Developer", 
                 email = "Joe.Developer@some.domain.net"),
                 person("A.", "User", role = "ctb",
                 email = "A.User@wherever.net"))
Author: Joe Developer <Joe.Developer@some.domain.net>, with
        contributions from A. User <A.User@wherever.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
BugReports: http://pkgname.bugtracker.url
```

The format is that of a ‘Debian Control File’ (see the help for ‘\read.dcf’ and http://www.debian.org/doc/debian-policy/ch-controlfields.html: R does not require encoding in UTF-8). Continuation lines (for example, for descriptions longer than

---

1 false positives are possible, but only a handful have been seen so far.
one line) start with a space or tab. The ‘Package’, ‘Version’, ‘License’, ‘Description’, ‘Title’, ‘Author’, and ‘Maintainer’ fields are mandatory; all other fields are optional.

For maximal portability, the ‘DESCRIPTION’ file should be written entirely in ASCII — if this is not possible it must contain an ‘Encoding’ field (see below).

The mandatory ‘Package’ field gives the name of the package. This should contain only letters, numbers and dot, and start with a letter. To be portable to Windows, it must not end in a dot. (Translation packages are allowed names of the form ‘Translation-ll’.) Do not use ‘R’ as a package name: that has a special meaning in package dependency analysis. Other single-character names are deprecated and will be disallowed as from R 2.14.0, as will names ending in a dot.

The mandatory ‘Version’ field gives the version of the package. This 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’.

The mandatory ‘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 sub-directory ‘share/licenses’ of the R source or home directory.
• The names of abbreviations of free or open source software (FOSS, e.g., http://en.wikipedia.org/wiki/FOSS) licenses as contained in 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 are no restrictions on distribution or use other than those imposed by relevant laws (including copyright laws).

If a package license extends a base FOSS license (e.g., using GPL-3 or AGPL-3 with an attribution clause), the extension should be placed in file ‘LICENSE’ (or ‘LICENCE’), and the string ‘+ file LICENSE’ (or ‘+ file LICENCE’, respectively) should be appended to the corresponding individual license specification.

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
License: Artistic-1.0 | AGPL-3 + file LICENSE

Please note in particular that “Public domain” is not a valid license, since it is not recognized in some jurisdictions.

It is very important that you include this license information! Otherwise, it may not even be legally correct for others to distribute copies of the package. Do not use the ‘License’ field for copyright information: if needed, use a ‘Copyright’ field.

Please ensure that the license you choose also covers any dependencies (including system dependencies) of your package: it is particularly important that any restrictions on the use of such dependencies are evident to people reading your ‘DESCRIPTION’ file.

The mandatory ‘Description’ field should give a comprehensive description of what the package does. One can use several (complete) sentences, but only one paragraph.

The mandatory ‘Title’ field should give a short description of the package. Some package listings may truncate the title to 65 characters. It should be capitalized, not use any markup, not have any continuation lines, and not end in a period.

The mandatory ‘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 ‘Author@R’ field can be used to provide a refined, machine-readable description of the package “authors” (in particular specifying their precise roles), via suitable R code (see ?citation for more information). Auto-generated citation information takes advantage of this specification, and eventually, Author and Maintainer fields will be auto-generated from it if needed.

The mandatory ‘Maintainer’ field should give a single name with a valid (RFC 2822) email address in angle brackets (for sending bug reports etc.). It should not end in a period or comma. For a public package it should be a person, not a mailing list and not a corporate entity: do ensure that it is valid and will remain valid for the lifetime of the package.

The ‘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 8601 standard.

The ‘Depends’ field gives a comma-separated list of package names which this package depends on. The package name may be optionally followed by a comment in parentheses. The comment should contain a comparison operator\(^2\), whitespace and a valid version number. 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 2.11.0 or later, include ‘R (>= 2.11.0)’ 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, possibly amplified in 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 requirements) before the current package, both when library is called and when preparing for lazy-loading.

---

\(^2\) only ‘\(>=\)’ is supported for package versions by install.packages.
A package (or ‘R’) can appear more than once in the ‘Depends’, but only the first occurrence will be used in versions of R prior to 2.7.0: these are now very unlikely to be encountered.

The ‘Imports’ field lists packages whose name spaces are imported from (as specified in the ‘NAMESPACE’ file) 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 that are used, 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). Packages declared in the ‘Depends’ field should not also be in the ‘Imports’ field. Version requirements can be specified, but will not be checked when the name space is loaded (whereas they are checked by R CMD check).

The ‘Suggests’ field uses the same syntax as ‘Depends’ and lists packages that are not necessarily needed. This includes packages used only in examples, tests or vignettes (see Section 1.4 [Writing package vignettes], page 26), 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 use foo unless one wants to execute all the examples/tests/vignettes: it is useful to have bar, but not necessary. Version requirements can be specified, and will be used by R CMD check.

Finally, the ‘Enhances’ field lists packages “enhanced” by the package at hand, e.g., by providing methods for classes from these packages. Version requirements can be specified, but are currently not used.

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 and not in the ‘Depends’ field.
- Packages that need to be attached to successfully load the package using library(pkgname) must be listed in the ‘Depends’ field, only.
- 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’. Packages used to run examples or tests conditionally (e.g. via if(require(pkgname))) should be listed in ‘Suggests’ or ‘Enhances’. (This allows checkers to ensure that all the packages needed for a complete check are installed.)

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.

Version dependencies in the ‘Depends’ field are used by library when it loads the package, and install.packages checks versions for the ‘Imports’ and (for dependencies = TRUE) ‘Suggests’ fields.

The ‘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 in CRAN package listings.

The ‘BugReports’ field may contain a single URL to which bug reports about the package should be submitted. This URL will be used by bug.reports instead of sending an email to the maintainer.
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 ‘Collate’ field can be used for controlling the collation order for the R code files in a package when these are processed for package installation. The default is to collate 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 8) as a whitespace separated list of file paths relative to the ‘R’ subdirectory. Paths containing white space or quotes need to be quoted. An OS-specific collation field (‘Collate.unix’ or ‘Collate.windows’) will be used instead of ‘Collate’.

The ‘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 ‘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. (Setting to any other value is deprecated, and it is unused as from R 2.13.0: but it might still be needed if the package can be installed under earlier versions of R.)

The ‘BuildVignettes’ field can be set to a false value to stop R CMD build from attempting to rebuild the vignettes, as well as preventing R CMD check from testing this. This should only be used exceptionally, for example if the PDFs need large figures which are not part of the package sources.

If the ‘DESCRIPTION’ file is not entirely in ASCII it should contain an ‘Encoding’ field specifying an encoding. This is used as the encoding of the ‘DESCRIPTION’ file itself and of the ‘R’ and ‘NAMESPACE’ files, and as the default encoding of ‘.Rd’ files. The examples are assumed to be in this encoding when running R CMD check, and it is used for the encoding of the CITATION file. 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 ‘OS_type’ field specifies the OS(es) for which the package is intended. If present, it should be one of unix or windows, and indicates that the package can only be installed on a platform with ‘.Platform$OS.type’ having that value.

The ‘Type’ field specifies the type of the package: see Section 1.11 [Package types], page 45.

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

One can add subject classifications for the content of the package using the fields ‘Classification/ACM’ (using the Computing Classification System of the Association for Computing Machinery, http://www.acm.org/class/), ‘Classification/JEL’ (the Journal of Economic Literature Classification System, http://www.aeaweb.org/journal/jel_class_system.html), or ‘Classification/MSC’ (the Mathematics Subject Classification of the American Mathematical Society, http://www.ams.org/msc/). The subject classifications should be comma-separated lists of the respective classification codes, e.g., ‘Classification/ACM: G.4, H.2.8, I.5.1’.
Finally, an ‘Language’ field can be used to indicate if the package documentation is not in English: this should be a comma-separated list of standard (not private use or grandfath-er) IETF language tags as currently defined by RFC 5646 (http://tools.ietf.org/html/rfc5646, see also http://en.wikipedia.org/wiki/IETF_language_tag), i.e., use language subtags which in essence are 2-letter ISO 639-1 (http://en.wikipedia.org/wiki/ISO_639-1) or 3-letter ISO 639-3 (http://en.wikipedia.org/wiki/ISO_639-3) language codes.

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 tools::Rdindex()) when installing from source.

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 54) and/or a vignette (see Section 1.4 [Writing package vignettes], page 26).

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 name spaces 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: please use suitable compression as suggested by tools::resaveRdaFiles) 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 fail when the package is installed. Any byte will be allowed in a quoted character string (but \uxxxx escapes should not be used unless the package depends on R (>= 2.10)): however,

---

3 This is true for OSes which implement the ‘C’ locale: Windows’ idea of the ‘C’ locale uses the WinAnsi charset.

4 It is good practice to encode them as octal or hex escape sequences.
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 33, 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. 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\(^5\) they must be entirely ASCII and not contain `%`. See Chapter 2 [Writing R documentation files], page 47, for more information. 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 31).

Having a `man` directory containing no documentation files may give an installation error.

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 a 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 make variables and rules for this (determined when R is configured and recorded in `R_HOME/etcR_ARCH/Makeconf`), providing support for C, C++, FORTRAN 77, Fortran 9x\(^6\), Objective C and Objective C++\(^7\) with associated extensions ‘.c’, ‘.cc’ or ‘.cpp’, ‘.f’, ‘.f90’ or ‘.f95’, ‘.m’, and ‘.mm’ or ‘.M’, respectively. We recommend using ‘.h’ for headers, also for C++\(^8\) or Fortran 9x include files. (Use of extension ‘.C’ for C++ is no longer supported.)

It is not portable (and may not be possible at all) to mix all these languages in a single package, and we do not support using both C++ and Fortran 9x. Because R itself uses it,

---

\(^5\) More precisely, they can contain the English alphanumeric characters and the symbols ‘$ _ . + ! ’ ( ), ; = $’.  
\(^6\) 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.  
\(^7\) either or both of which may not be supported on particular platforms  
\(^8\) Using ‘.hpp’, although somewhat popular, is not guaranteed to be portable.
we know that C and FORTRAN 77 can be used together and mixing C and C++ seems to
be widely successful.

If your code needs to depend on the platform there are certain defines which can
be used in C or C++. On all Windows builds (even 64-bit ones) ‘WIN32’ will be defined:
on 64-bit Windows builds also ‘WIN64’, and on Mac OS X ‘__APPLE__’ and ‘__APPLE_CC__’
are defined.

The default rules can be tweaked by setting macros\(^9\) in a file ‘src/Makevars’ (see
Section 1.2.1 [Using Makevars], page 15). 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 plat-
forms. If it has any targets at all, it should have an appropriate first target named ‘all’ and
a (possibly empty) target ‘clean’ which removes all files generated by running 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. Some make programs require makefiles to have a
complete final line, including a newline.

A few packages use the ‘src’ directory for purposes other than making a shared ob-
ject (e.g. to create executables). Such packages should have files ‘src/Makefile’ and
‘src/Makefile.win’ (unless intended for only Unix-alikes or only Windows).

The ‘data’ subdirectory is for data files: See Section 1.1.5 [Data in packages], page 11.

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 to achieve this. 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’,
of the ‘inst’ happens after ‘src’ is built so its ‘Makefile’ can create files to be installed.
Prior to R 2.12.2, the files were installed on POSIX platforms with the permissions in the
package sources, so care should be taken to ensure these are not too restrictive: R CMD build
will make suitable adjustments. To exclude files from being installed, one can specify a list
of exclude patterns in file ‘.Rinstignore’ in the top-level source directory. These patterns
should be Perl-like regular expressions (see the help for regexp in R for the precise details),
one per line, to be matched\(^10\) against the file and directory names.

Note that with the exceptions of ‘INDEX’, ‘LICENSE’/‘LICENCE’, ‘COPYING’ and ‘NEWS’,
information files at the top level of the package will not be installed and so not be known
to users of Windows and Mac OS 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’. Note that if the named exceptions also occur
in ‘inst’, the versions in ‘inst’ will be that seen in the installed package.

\(^9\) the POSIX terminology, called ‘make variables’ by GNU make.
\(^10\) case-insensitively on Windows.
Chapter 1: Creating R packages

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).

If ‘tests’ has a subdirectory ‘Examples’ containing a file pkg-Ex.Rout.save, this is compared to the output file for running the examples when the latter are checked.

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. NB: only files (and not directories) under ‘exec’ are installed, and they are all marked as executable (mode 755, moderated by ‘umask’) on POSIX platforms.

Subdirectory ‘po’ is used for files related to localization: see Section 1.9 [Internationalization], page 42.

1.1.4 Package bundles

Support for package bundles was removed in R 2.11.0.

1.1.5 Data in packages

The ‘data’ subdirectory is for data files, either to be made available via lazy-loading or for loading using data(). (The choice is made by the ‘LazyData’ field in the ‘DESCRIPTION’ file.) It should not be used for other data files needed by the package, and the convention has grown up to use directory ‘inst/extdata’ for such files.

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, and note that ‘.csv’ is not the standard CSV format), or save() images (‘.RData’ or ‘.rda’). Note that R code should be “self-sufficient” and not make use of extra functionality provided by the package, so that the data file can also be used without having to load the package.

Images (extensions ‘.RData’ or ‘.rda’) can contain references to the name spaces of packages that were used to create them. Preferably there should be no such references in data files, and in any case they should only be to packages listed in the Depends and Imports, as otherwise it may be impossible to install the package. To check for such references, load all the images into a vanilla R session, and look at the output of loadedNamespaces().

If your data files are large and you are not using ‘LazyData’ you can speed up installation by providing a file ‘datalist’ in the ‘data’ subdirectory. This should have one line per topic

---

11 The best way to generate such a file is to copy the ‘.Rout’ from a successful run of R CMD check. If you want to generate it separately, do run R with options ‘--vanilla --slave’ and with environment variable LANGUAGE=en set to get messages in English.

that `data()` will find, in the format ‘foo’ if `data(foo)` provides ‘foo’, or ‘foo: bar bah’ if `data(foo)` provides ‘bar’ and ‘bah’. R CMD build will automatically add a ‘datalist’ file to ‘data’ directories of over 1Mb, using the function `tools::add_datalist`.

Tables (‘.tab’, ‘.txt’, or ‘.csv’ files) can be compressed by `gzip`, `bzip2` or `xz`, optionally with additional extension ‘.gz’, ‘.bz2’ or ‘.xz’. However, such files can only be used with R 2.10.0 or later, and so the package should have an appropriate ‘`depends`’ entry in its DESCRIPTION file.

If your package is to be distributed, do consider the resource implications of large datasets for your users: they can make packages very slow to download and use up unwelcome amounts of storage space, as well as taking many seconds to load. It is normally best to distribute large datasets as ‘.rda’ images prepared by `save(, compress = TRUE)` (the default): there is no excuse for distributing ASCII saves. Using `bzip2` or `xz` compression will usually reduce the size of both the package tarball and the installed package, in some cases by a factor of two or more. However, such compression can only be used with R 2.10.0 or later, and so the package should have an appropriate ‘`depends`’ entry in its DESCRIPTION file.

Package `tools` has a couple of functions to help with data images: `checkRdaFiles` reports on the way the image was saved, and `resaveRdaFiles` will re-save with a different type of compression, including choosing the best type for that particular image.

Some packages using ‘LazyData’ will benefit from using a form of compression other than `gzip` in the installed lazy-loading database. This can be selected by the ‘--data-compress’ option to R CMD INSTALL or by using the ‘LazyDataCompression’ field in the ‘DESCRIPTION’ file. Useful values are `bzip2`, `xz` and the default, `gzip`. The only way to discover which is best is to try them all and look at the size of the ‘`pkgname/data/Rdata.rdb`’ file.

## 1.1.6 Non-R scripts in packages

Code which needs to be compiled (C, C++, FORTRAN, Fortran 95 . . .) is included in the ‘src’ subdirectory and discussed elsewhere in this document.

Subdirectory ‘exec’ could be used for scripts for interpreters such as the shell (e.g. `arulesSequences`), BUGS, Java, JavaScript, Matlab, Perl (`FEST`), php (`amap`), Python or Tcl, or even R. However, it seems more common to use the ‘inst’ directory, for example ‘AMA/inst/java’, ‘WriteXLS/inst/Perl’, ‘Amelia/inst/tklibs’, ‘CGIwithR/inst/cgi-bin’, ‘NMF/inst/matlab’ and ‘emdbook/inst/BUGS’.

If your package requires one of these interpreters or an extension then this should be declared in the ‘SystemRequirements’ field of its ‘DESCRIPTION’ file. Windows users should be aware that the Tcl extensions ‘BWidget’ and ‘Tktable’ which are included with the R for Windows installer are extensions and do need to be declared. ‘Tktable’ does ship as part of the Tcl/Tk provided on CRAN for Mac OS X, but you will need to tell your users how to make use of it:

```r
> addTclPath('/usr/local/lib/Tktable2.9')
> tclRequire('Tktable')
<Tcl> 2.9
```
1.2 Configure and cleanup

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

If your package needs some system-dependent configuration before installation you can include an executable (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.).

Under a Unix-alike only, an executable (Bourne shell) script ‘cleanup’ is executed as the last thing by `R CMD INSTALL` if 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

```bash
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

```r
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

```r
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 a Unix-alike, 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 '${R_HOME}/bin/R' rather than just 'R' is necessary in order to use the correct version of R when running the script as part of R CMD INSTALL, and the quotes since '${R_HOME}' might contain spaces.)

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.

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

```bash
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 will not be used on Windows systems. If your package is to be made publicly available, please give enough information for a user on a non-Unix-alike 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. Both should be shell scripts to be executed by ash, which is a minimal version of Bourne-style sh.) When 'configure.win' is run the environment variables R_HOME (which uses / as the file separator) and R_ARCH will be set. Use R_ARCH to decide if this is a 64-bit build (its value there is '/x64') and to install DLLs to the correct place (${R_HOME}/libs${R_ARCH}). Use R_ARCH_BIN to find the correct place under the 'bin' directory, e.g. ${R_HOME}/bin${R_ARCH_BIN}/Rscript.exe.

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 environment variable R_PACKAGE_NAME. (Currently the value of R_PACKAGE_DIR is always ${R_LIBRARY_DIR}/${R_PACKAGE_NAME}, but this used not to be the case when versioned installs were allowed. Its main use is in ‘configure.win’ scripts for the installation path of external software’s DLLs.) Note that the value of R_PACKAGE_DIR may contain spaces and other shell-unfriendly characters, and so should be quoted in makefiles and configure scripts.

One of the more tricky tasks can be to find the headers and libraries of external software. One tool which is increasingly available on Unix-alikes (but not Mac OS X) to do this is pkg-config. The ‘configure’ script will need to test for the presence of the command itself (see for example package Cairo), and if present it can be asked if the software is installed, of a suitable version and for compilation/linking flags by e.g.

```
$ pkg-config --exists 'QtCore >= 4.0.0' # check the status
$ pkg-config --modversion QtCore
4.7.1
$ pkg-config --cflags QtCore
-DQT_SHARED -I/usr/include/QtCore
$ pkg-config --libs QtCore
-lQtCore
```

Note that pkg-config --libs gives the information required to link against the default version of that library (usually the dynamic one), and pkg-config --static is needed if the static library is to be used.

Sometimes the name by which the software is known to pkg-config is not what one might expect (e.g. ‘gtk+-2.0’ even for 2.22). To get a complete list use

```
pkg-config --list-all | sort
```

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 options (for example include paths) for C/C++ files via PKG_CPPFLAGS, and additional compiler flags by setting PKG_CFLAGS, PKG_CXXFLAGS, PKG_FFLAGS or PKG_FCFLAGS, for C, C++, FORTRAN or Fortran 9x respectively (see Section 5.5 [Creating shared objects], page 90).

‘Makevars’ can also be used to set flags for the linker, for example ‘-L’ and ‘-l’ options, via PKG_LIBS.

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\textsuperscript{13} which are set whilst configuring the building of R itself and are stored in ‘.R_HOME/etcR_ARCH/Makeconf’. That makefile is included as a ‘Makefile’ after

\textsuperscript{13} in POSIX parlance: GNU make calls these ‘make variables’.
‘Makevars[.win]’, and the macros it defines can be used in macro assignments and make command lines in the latter. These include

**FLIBS**
A macro containing the set of libraries need to link FORTRAN code. This may need to be included in **PKG_LIBS**: it will normally be included automatically if the package contains FORTRAN source files.

**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\(^{14}\), as most BLAS libraries are written at least partially 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**. It 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.]
For portability, 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 `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. It can also be used to control the set of files which are compiled, either by excluding some files in ‘src’ or including some files in subdirectories. For example

```plaintext
OBJECTS = 4dfp/endianio.o 4dfp/Getifh.o R4dfp-object.o
```

Note that ‘Makevars’ should not normally contain targets, as it is included before the default makefile and `make` will call the first target, intended to be all in the default makefile. If you really need to circumvent that, use a suitable (phony) target all before any actual targets in ‘Makevars[.win]’: for example package `fastICA` has

\(^{14}\) at least on Unix-alikes: the Windows build currently resolves such dependencies to a static FORTRAN library when ‘Rblas.dll’ is built.
Chapter 1: Creating R packages

```
PKG_LIBS = @BLAS_LIBS@
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
```

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

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

(since the other macros are all empty on that platform, and R’s internal BLAS is not used).
Note that the first target in ‘Makevars’ will be called, but for back-compatibility it is best named all.

If you want to create and then link to a library, say using code in a subdirectory, use something like

```
.PHONY: all mylibs
all: $(SHLIB)
  $(SHLIB): mylibs
    (cd subdir; make)
```

Be careful to create all the necessary dependencies, as there is a no guarantee that the dependencies of all will be run in a particular order (and some of the CRAN build machines use multiple CPUs and parallel makes).

Note that on Windows it is required that ‘Makevars[.win]’ does create a DLL: this is needed as it is the only reliable way to ensure that building a DLL succeeded. If you want to use the ‘src’ directory for some purpose other than building a DLL, use a ‘Makefile.win’ file.

It is sometimes useful to have a target ‘clean’ in ‘Makevars’ or ‘Makevars.win’: this will be used by R CMD build to clean up (a copy of) the package sources. When it is run by build it will have fewer macros set, in particular not $(SHLIB), nor $(OBJECTS) unless set in the file itself. It would also be possible to add tasks to the target ‘shlib-clean’ which is run by R CMD INSTALL and R CMD SHLIB with options ‘--clean’ and ‘--preclean’.

If you want to run R code in ‘Makevars’, e.g. to find configuration information, please do ensure that you use the correct copy of R or Rscript: there might not be one in the path at all, or it might be the wrong version or architecture. The correct way to do this is via

```
"$(R_HOME)/bin$(R_ARCH_BIN)/Rscript" filename
"$(R_HOME)/bin$(R_ARCH_BIN)/Rscript" -e 'R expression'
```

where $(R_ARCH_BIN) is only needed currently on Windows.

Environment or make variables can be used to select different macros for 32- and 64-bit code, for example (GNU make syntax, allowed on Windows)
ifeq "$(WIN)" "64"
Pkg_LIBS = value for 64-bit Windows
else
Pkg_LIBS = value for 32-bit Windows
endif

On Windows there is normally a choice between linking to an import library or directly
to a DLL. Where possible, the latter is much more reliable: import libraries are tied to a
specific toolchain, and in particular on 64-bit Windows two different conventions have been
commonly used. So for example instead of

    Pkg_LIBS = -L$(XML_DIR)/lib -lxml2

one can use

    Pkg_LIBS = -L$(XML_DIR)/bin -lxml2

since on Windows -lxxx will look in turn for

    libxxx.dll.a
    xxx.dll.a
    libxxx.a
    xxx.lib
    libxxx.dll
    xxx.dll

where the first and second are conventionally import libraries, the third and fourth often
static libraries (with .lib intended for Visual C++), but might be import libraries. See for
example http://sourceware.org/binutils/docs-2.20/ld/WIN32.html#WIN32.

The fly in the ointment is that the DLL might not be named ‘libxxx.dll’, and in fact
on 32-bit Windows there is ‘libxml2.dll’ whereas on 64-bit Windows the DLL is called
‘libxml2-2.dll’. Using import libraries can cover over these differences but can cause
equal difficulties.

If static libraries are available they can save a lot of problems with run-time finding of
dlls, especially when binary packages are to be distributed and even more when these
support both architectures. Where using dlls is unavoidable we normally arrange (via
‘configure.win’) to ship them in the same directory as the package dll.

1.2.1.1 OpenMP support

As from R 2.13.0 there is some support for packages which wish to use OpenMP\(^\text{15}\). The
make macros

    SHLIB_OPENMP_CFLAGS
    SHLIB_OPENMP_CXXFLAGS
    SHLIB_OPENMP_FCFLAGS
    SHLIB_OPENMP_FFLAGS

are available for use in ‘Makevars’,\(^\text{16}\). Include the appropriate macro in PKG_CFLAGS, PKG_
CPPFLAGS and so on, and also in PKG_LIBS. C/C++ code that needs to be conditioned on the

tutorials/openmp/

\(^{16}\) They could also be used with ‘Makevars.win’, but at present the compilers used for R on Windows do
not support pthreads nor OpenMP.
use of OpenMP can be used inside `#ifdef SUPPORT_OPENMP`, a macro defined in the header ‘Rconfig.h’ (see Section 6.13 [Platform and version information], page 135): however the use of OpenMP is most often indicated by `#pragma` statements.

For example, a package with C code written for OpenMP should have in ‘src/Makevars’ the lines

```plaintext
PKG_CFLAGS = $(SHLIB_OPENMP_CFLAGS)
PKG_LIBS = $(SHLIB_OPENMP_CFLAGS)
```

There is nothing to say what version of OpenMP is supported: version 3.0 is supported by recent versions of the main platforms (except Windows, where there is currently no support), but portable packages cannot assume that end users have recent versions.

### 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’).

```plaintext
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

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}, ,
    "ODBC driver manager ${odbc_mgr} not found")
else
    AC_SEARCH_LIBS(SQLTables, odbc odbc32 iodbc, ,
    "no ODBC driver manager found")
fi

dnl for 64-bit ODBC need SQL[UL]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
Chapter 1: Creating R packages

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)

```bash
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 assumes that source files with extension `.f` are FORTRAN 77, and passes them to the compiler specified by `F77`. On most but not all platforms that compiler will accept Fortran 90/95 code: some platforms have a separate Fortran 90/95 compiler and a few (typically older) platforms have no Fortran 90/95 support.

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](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 compiled 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 may get more detailed error messages doing the checks directly.

**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 can set it in the environment: also you can use files¹⁷ `~/.R/check.Renviron` and `~/.R/build.Renviron` to set environment variables when using these utilities.

**Note to Windows users:** `R CMD build` requires you to have installed the Windows toolset (see the “R Installation and Administration” manual) and have it in your path, and `R CMD check` will make use of it if present. You may need to set `TMPDIR` to point to a suitable writable directory with a path not containing spaces – use forward slashes for the separators. Also, the directory needs to be on a case-honouring file system (some network-mounted file systems are not).

---

¹⁷ On systems which use sub-architectures, architecture-specific versions such as `~/.R/check.Renviron.i386` take precedence.
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\(^{18}\) with extension `.tar.gz` or `.tgz`. (Some platforms may allow other forms of compression and extensions `.tar.bz2` and `.tar.xz`.)

This runs a series of checks, including

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-alikes only).
4. The files are checked for binary executables, using a suitable version of `file` if available\(^{19}\). (There may be rare false positives.)
5. 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. (You may need to set `R_LIBS` if dependent packages are in a separate library tree.) One check is that the package name is not that of a standard package, nor one of the defunct standard packages (`cctest`, `eda`, `lqs`, `mle`, `modreg`, `mva`, `nls`, `stepfun` and `ts`). 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.
6. Available index information (in particular, for demos and vignettes) is checked for completeness.
7. 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 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 `.` will be allowed in the `R` directory.

A warning is given for directory names that look like R package check directories – many packages have been submitted to CRAN containing these.
8. 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.
9. 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

---

\(^{18}\) This may require GNU `tar`: the command used can be set with environment variable `TAR`.  
\(^{19}\) A suitable `file.exe` is part of the Windows toolset.
loaded. (Name spaces and packages can be loaded very early in the session, before the
default packages are available, so packages should work then.)

10. The R files are checked for correct calls to library.dynam. 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)

11. The ‘Rd’ files are checked for correct syntax and metadata, including the presence of
the mandatory (\name, \alias, \title and \description) fields. The ‘Rd’ name and
title are checked for being non-empty, and there is a check for missing cross-references
(links).

12. A check is made for missing documentation entries, such as undocumented user-level
objects in the package.

13. Documentation for functions, data sets, and S4 classes is checked for consistency with
the corresponding code.

14. It is checked whether all function arguments given in \usage sections of ‘Rd’ files are
documented in the corresponding \arguments section.

15. 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’
file in the ‘src’ directory, it is checked for portable line endings and the correct use of
‘$(BLAS_LIBS)’.

16. The examples provided by the package’s documentation are run. (see Chapter 2 [Writ-
ing R documentation files], page 47, for information on using \examples to create
executable example code.) If there is a file ‘tests/Examples/pkg-Ex.Rout.save’, the
output of running the examples is compared to that file.

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.

17. If the package sources contain a ‘tests’ directory then the tests specified in that direc-
tory are run. (Typically they will consist of a set of ‘.R’ source files and target output
files ‘.Rout.save’.) Please note that the comparison will be done in the end user’s
locale, so the target output files should be ASCII if at all possible.

18. The code in package vignettes (see Section 1.4 [Writing package vignettes], page 26) is
executed, and the vignettes re-made from their sources as a check of completeness of
the sources (unless there is a ‘Vignettes’ field in the package’s ‘DESCRIPTION’ file with
a false value).

If there is an error in executing the R code in vignette ‘foo’, a log file ‘foo.log’ is cre-
at in the check directory. The vignettes are re-made in a copy of the package sources
in the ‘vign_test’ subdirectory of the check directory, so for further information on
errors look in directory ‘pkgname/vign_test/inst/doc’. (It is only retained if there
are errors.)
Chapter 1: Creating R packages

19. The PDF 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 command-line options. It also allows customization by setting environment variables `_R_CHECK_*_` as described in Section “Tools” in *R Internals*.

   You do need to ensure that the package is checked in a suitable locale if it contains non-ASCII characters. Such packages are likely to fail some of the checks in a C locale, and `R CMD check` will warn if it spots the problem. You should be able to check any package in a UTF-8 locale (if one is available). Beware that although a C locale is rarely used at a console, it may be the default if logging in remotely or for batch jobs.

   **Multiple sub-architectures:** On systems which support multiple sub-architectures (principally Windows and Mac OS X), `R CMD check` will install and check a package which contains compiled code under all available sub-architectures. (Use option `--force-multiarch` to force this for packages without compiled code, which are otherwise only checked under the main sub-architecture.) This will run the loading tests, examples and ‘tests’ directory under each installed sub-architecture in turn, and give an error if any fail. Where environment variables (including `PATH`) need to be set differently for each sub-architecture, these can be set in architecture-specific files such as `R_HOME/etc/i386/Renviron.site`.

   An alternative approach is to use `R CMD check --no-multiarch` to check the primary sub-architecture, and then to use something like `R --arch=x86_64 CMD check --extra-arch` or (Windows) `/path/to/R/bin/x64/Rcmd check --extra-arch` to run for each additional sub-architecture just the checks which differ by sub-architecture.

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 standard 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 relevant 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 final 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-like regular expressions (see the help for `regexp` in R for the precise details), one per line, to be matched case-insensitively on Windows.

   Loading, examples, tests, vignettes
different by sub-architecture.

20  this is needed on Windows to select the appropriate GTK+ and ‘graphviz’ DLLs.
21  loading, examples, tests, vignettes
22  case-insensitively on Windows.
23  including directory names as from R 2.13.0: earlier versions accepted the names of non-empty directories.
addition, directories from source control systems, directories with names ending `.Rcheck`, `Old` or `old` and files ‘GNUMakefile’, ‘Read-and-delete-me’ 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 26) in the package. To do so it installs the current package into a temporary library tree, but any dependent packages need to be installed in an available library tree (see the Note: below).

Similarly, if the ‘.Rd’ documentation files contain any \Sexpr macros (see Section 2.11 [Dynamic pages], page 61), the package will be temporarily installed to execute them. Post-execution binary copies of those pages containing build-time macros will be saved in ‘build/partial.rdb’. If there are any install-time or render-time macros, a ‘.pdf’ version of the package manual will be built and installed in the ‘build/’ subdirectory. (This allows CRAN or other repositories to display the manual even if they are unable to install the package.)

One of the checks that `R CMD build` runs is for empty source directories. These are in most (but not all) cases unintentional, if they are intentional use the option ‘--keep-empty-dirs’.

The ‘--resave-data’ option allows saved images (‘.rda’ and ‘.RData’ files) in the ‘data’ directory to be optimized for size. It will also compress tabular files and convert ‘.R’ files to saved images. It can take values `no`, `gzip` (the default if this option is not supplied, which can be changed by setting the environment variable `_R_BUILD_RESAVE_DATA_`) and `best` (equivalent to giving it without a value), which chooses the most effective compression. Using `best` adds a dependence on R (>= 2.10) to the ‘DESCRIPTION’ file if `bzip2` or `xz` compression is selected for any of the files. If this is thought undesirable, ‘--resave-data=gzip’ (which is the default if that option is not supplied) will do what compression it can with gzip.

The ‘--compact-vignettes’ option will run `tools::compactPDF` over the PDF files in ‘inst/doc’ (and its subdirectories) to losslessly compress them. This is not enabled by default (it can be selected by environment variable `_R_BUILD_COMPACT_VIGNETTES_`) and needs qpdf (http://qpdf.sourceforge.net/) to be available.

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 it is now deprecated in favour of `R CMD INSTALL --build`.

Note that prior to R 2.13.0, `R CMD build` did some cleaning in the supplied source directory, but this was undocumented and is no longer done.

`R CMD build` requires a suitable `tar` program that can produce a compressed tarball: almost certainly one will have been found when R was configured on a Unix-alike (and the

---

24 called ‘CVS’ or `.svn’ or `.arch-ids’ or ‘.bzf’ or ‘.git’ or ‘.hg’.

25 or the package’s description contains ‘BuildVignettes: no’ or similar.
Windows toolset contains one), but if there are problems, set the environment variable TAR to the path to a suitable program or to "internal" if none is available.

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. Documents in ‘inst/doc’ can be in arbitrary format, however we strongly recommend providing them in PDF format, so users on almost all platforms can easily read them. To ensure that they can be accessed from a browser (as an HTML index is provided), the file names should start with an ASCII letter and be comprised entirely of ASCII letters or digits or hyphen or underscore.

A special case are PDF documents with sources in Sweave format, which we call package vignettes. These are normally given the file extension ‘.Rnw’ or ‘.Rtex’, but for historical reasons extensions\(^{26}\) ‘.Snw’ and ‘.Stex’ are also recognized as vignettes. Sweave allows the integration of \LaTeX\ documents: see the Sweave help page in R 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 (except those with option eval=FALSE). The R working directory for all vignette tests in R CMD check is a copy 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 by using calls to system.file(). All other files needed to re-make the vignette PDFs (such as LaTeX style files, BiBTeX input files and files for any figures not created by running the code in the vignette) must in the sources (usually under ‘inst/doc’, but a file ‘.Rinstignore’ can be used to exclude them from the installed package). References to parallel source directories such as

\begin{verbatim}
\graphicspath{{../../../manual/}}
\end{verbatim}

(from RODBC) will work as from R 2.13.0.

R CMD build will automatically\(^{27}\) 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 run and compiled at install time, i.e., the package author can use private R packages, screen snapshots and \LaTeX\ extensions which are only available on his machine.\(^{28}\)

By default R CMD build will run Sweave on all files in Sweave format in ‘inst/doc’ (but not in sub-directories). If no ‘Makefile’ is found in directory ‘inst/doc’, then tools::texi2dvi(pdf = TRUE) is run on all processed vignettes. Whenever a ‘Makefile’ is found, then R CMD build will try to run make after the Sweave runs. The first target in the ‘Makefile’ should take care of both creation of PDF files and cleaning up afterwards (including after Sweave), i.e., delete all files that shall not appear in the final package archive. Note that if the make step runs R it needs to be careful to respect the environment values

\(^{26}\) and to avoid problems with case-insensitive file systems, lower-case versions of all these extensions.
\(^{27}\) unless inhibited by using ‘BuildVignettes: no’ in the ‘DESCRIPTION’ file.
\(^{28}\) provided the conditions of the package’s licence are met: many would see these as incompatible with an Open Source licence.
of R_LIBS and R_HOME\textsuperscript{29}. Finally, if there is a ‘Makefile’ and it has a ‘clean:’ target, make clean is run.

All the usual caveats about including a ‘Makefile’ apply. It must be portable (no GNU extensions) and must work correctly with a parallel make: too many authors have written things like

```r
## BAD EXAMPLE
all: pdf clean


%.pdf: %.tex
texi2dvi --pdf $*

clean:
    rm *.tex ABC-details-*.pdf
```

which will start removing the source files whilst pdflatex is working.

Note that it is pointless (and potentially misleading since the files might be outdated) to include in ‘inst/doc’ R code files which would be generated from vignettes, as these will be re-generated when the package is installed (unless the vignette does not generate any R code, in which case it is also pointless/misleading).

Metadata lines can be placed in the source file, preferably in LaTeX comments in the preamble. One such is a \texttt{\VignetteIndexEntry} of the form

```
%\VignetteIndexEntry{Using Animal}
```

Others you may see are \texttt{\VignettePackage} (currently ignored), \texttt{\VignetteDepends} and \texttt{\VignetteKeyword} (which replaced \texttt{\VignetteKeywords}). These are processed at package installation time to create the saved data frame ‘Meta/vignette.rds’, but only the \texttt{\VignetteIndexEntry} statements are currently used.

At install time an HTML index for all vignettes in the package is automatically created from the \texttt{\VignetteIndexEntry} statements unless a file ‘index.html’ exists in directory ‘inst/doc’. This index is linked from the HTML help index for the package. If you do supply a ‘inst/doc/index.html’ file it should contain relative links only to files under the installed ‘doc’ directory, or perhaps (not really an index) to HTML help files or to the ‘DESCRIPTION’ file.

Sweave/Stangle allows the document to specify the \texttt{split=TRUE} option to create a single R file for each code chunk: this will not work for vignettes where it is assumed that each vignette source generates a single file with the vignette extension replaced by ‘.R’.

Do watch that PDFs are not too large – one in a CRAN package was 72Mb! This is usually caused by the inclusion of overly detailed figures, which will not render well in PDF viewers. Sometimes it is much better to generate fairly high resolution bitmap (PNG, JPEG) figures and include those in the PDF document.

\textsuperscript{29} As from R 2.13.0, R\_HOME/bin is prepended to the \texttt{PATH} so that references to R or \texttt{Rscript} in the ‘Makefile’ do make use of the currently running version of R.
1.4.1 Encodings and vignettes

Vignette PDFs will in general include descriptive text, R input, R output and figures, \LaTeX\ include files and bibliographic references. Any of which may contain non-ASCII characters, the handling of encodings can become very complicated.

The vignette source file should be written in ASCII or contain a declaration of the encoding (see below). This applies even to comments within the source file, since \texttt{Sweave()} processes comments to look for options and metadata lines. It is not assumed that the vignette sources are in the packages’ encoding (and there are packages for which the vignettes have different encodings).

\texttt{Stangle()} will produce an R code file: this is almost always ASCII.

\texttt{Sweave()} will produce a ‘.\texttt{tex}’ file in the current locale’s encoding. That needs to be declared to \LaTeX\ via a line like

\begin{verbatim}
\usepackage[utf8]{inputenc}
\end{verbatim}

\texttt{R CMD check} will warn about any non-ASCII vignettes it finds which do not have such a declaration. The problem is that this cannot be known in advance, so vignette PDFs may only be re-createable on the author’s own machine. \texttt{R CMD check} will report on any non-ASCII vignettes it finds which do not have such a declaration. (It is also possible to use the more recent ‘\texttt{inputenx}’ \LaTeX\ package.)

For R 2.13.0 (at least), how non-ASCII vignettes are handled depends on the locale of the R session.

- In an 8-bit locale the documents and R output are handled byte-by-byte. This may lead to invalid results if the vignette was intended to be in UTF-8 and the R output is in the 8-bit locale.
- In a UTF-8 locale the vignette is translated as if in Latin-1, and the output file translated back to Latin-1. It does not actually need to be in Latin-1 provided all the bytes used\footnote{and on some platforms this will work even with bytes outside Latin-1} in the vignette exist in Latin-1: think of this as a way of temporarily escaping 8-bit characters.
- In any other multi-byte locale \texttt{Sweave()} and \texttt{Stangle()} throw an error.

\texttt{Sweave()} will also parse and evaluate the R code in each chunk. The R output will also be in the current locale, and should be covered by the ‘\texttt{inputenc}’ declaration. One thing people often forget is that the R output may not be ASCII even for ASCII R sources, for many possible reasons. One common one is the use of ‘fancy’ quotes: see the R help on \texttt{sQuote}: note carefully that it is not portable to declare UTF-8 or CP1252 to cover such quotes, as their encoding will depend on the locale used to run \texttt{Sweave()}: this can be circumvented by setting \texttt{options(useFancyQuotes=“UTF-8“)} in the vignette.

The final issue is the encoding of figures – this applies only to PDF figures and not PNG etc. The PDF figures will contain declarations for their encoding, but the \texttt{Sweave()} option \texttt{pdf.encoding} may need to be set appropriately: see the help for the \texttt{pdf()} graphics device.

As a real example of the complexities, consider the \texttt{fortunes} package version ‘1.4-0’. That package did not have a declared encoding, and its vignette was in ASCII. However, the data it displays are read from a UTF-8 CSV file and will be assumed to be in the current encoding, so ‘\texttt{fortunes.tex}’ will be in UTF-8 in any locale. Had \texttt{read.table} been told the data were UTF-8, ‘\texttt{fortunes.tex}’ would have been in the locale’s encoding.
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. (Run from the directory containing ‘mypkg’ as a subdirectory.)

1. Run `R CMD build` to make the release ‘.tar.gz’ file.

2. Run `R CMD check` on the ‘.tar.gz’ file to check that the package will install and will run 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 via personal ‘Makevars’ files. See Section “Customizing package compilation” in *R Installation and Administration*.

   Note that it is particularly important to use ‘-Wall -pedantic’ with C++ code: the GNU C++ compiler has many extensions which are not supported by other compilers, and this will report some of them (such as the misuse of variable-length arrays). If possible, check C++ code on a standards-conformant compiler.

3. Study the output from running your examples, in file ‘pkg.Rcheck/pkg-Ex.Rout’. Often warnings there indicate actual errors, and warnings about your mistakes (which the R developers are warning you that they are working around for you) will just annoy or confuse your users.

   If your package has tests, study their output too.

4. Look for any problems with help file conversions. For example, you should

   • Read through the PDF manual that was produced by `R CMD check` at ‘mypkg.Rcheck/mypkg-manual.pdf’, or produce another copy by `R CMD Rd2pdf mypkg`.
   
   • Look at the rendering of your help pages in text from within R.

   Many aspects of help rendering changed in R 2.10.0, and in particular the interpretation of comment lines (which are rendered as blank lines, so do not put comment lines in the middle of a paragraph of text).

5. Ensure that the package sources are not unnecessarily large. In particular, `R CMD check` will report\(^3\) on installed packages of more than 5Mb, detailing directories of more than 1Mb. It warns about inefficient compression of data if environment variable `_R_CHECK_COMPACT_DATA2_` is set to ‘TRUE’: `R CMD build --resave-data` will compact data as best it can. Watch out for unnecessary files in ‘inst/doc’: for example ‘Rplots.ps’ or ‘Rplots.pdf’ left over from Sweave runs. As a general rule, ‘doc’ directories should not exceed 5Mb, and where ‘data’ directories need to be 10Mb or more, consideration should be given to a separate package containing just the data. (Similarly for external data directories, large ‘jar’ files and other libraries that need to be installed.)

\(^3\) provided a POSIX-compliant du program is found on the system: it is possible that some other du programs will incorrectly report double the actual size. This can be disabled by setting `_R_CHECK_PKG_SIZE_` to a false value.
There are several tools available to reduce the size of PDF files, including Adobe Acrobat (not Reader), Apple’s Preview, some versions of Multivalent, Pdftk (http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/), qpdf (http://qpdf.sourceforge.net/), and Ghostscript (which converts PDF to PDF by ps2pdf options -dAutoRotatePages=/None in.pdf out.pdf)

and suitable options might be

-dPDFSETTINGS=/ebook
-dPDFSETTINGS=/printer

; see http://pages.cs.wisc.edu/~ghost/doc/cvs/Ps2pdf.htm for more such and consider all the options for image downsampling) as well as numerous commercial and shareware Windows programs. Note that these do not all try the same size-reduction strategies, and Acrobat and ps2pdf can be much better at reducing the size of embedded images than some of the others.

Since qpdf is fairly readily available (e.g. it has binaries for Windows and packages in Debian/Ubuntu), there is an option ‘--compact-vignettes’ to R CMD build to run qpdf over PDF files under ‘inst/doc’ and replace them if at least 10Kb and 10% is saved. The full path to the qpdf command can be supplied as environment variable R_QPDF (and is on the CRAN binary of R for Mac OS X). This is done via the function tools::compactPDF which has other options including running Ghostscript.

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. If there are warnings you cannot eliminate (for example because you believe them to be spurious) send an explanatory note with your submission.

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 online.

Note also that for running LATEX, the Debian GNU/Linux CRAN check systems use reasonably recent versions of the Debian TexLive distribution (http://packages.debian.org/de/sid/texlive); for the Windows CRAN server, a reasonably recent version of MikTeX (including all packages available directly for MikTeX) is employed: the Mac OS X builders use a current full version of MacTeX. Developers wanting to have their vignettes use TeX packages or style files not (yet) included in these distributions should add the corresponding style files to the ‘inst/doc’ subdirectory of their package.

Note that CRAN does not accept submissions of precompiled binaries due to security concerns, and does not allow binary executables in packages. Maintainers who need additional software for the Windows binaries of their packages on CRAN have three options

1. To arrange for installation of the package to download the additional software from a URL, as e.g. package Cairo does.

---

32 Select ‘Save as’, and select ‘Reduce file size’ from the ‘Quartz filter’ menu’: this can be accessed in other ways, for example by Automator.
33 for Windows users the simplest way may be to open that URL in Internet Explorer and (depending on the version) follow the instructions to view it as a folder, then copy the submission to the folder.
2. To negotiate with Uwe Ligges to host the additional components on WinBuilder, and write a `configure.win` file to install them. There are many examples, e.g. package `rgdal`.

3. To negotiate with Brian Ripley to host the package on CRAN extras, as was done for package `BRugs`.

Be aware that in all cases license requirements will need to be met so you may need to supply the sources for the additional components (and will if your package has a GPL-like license).

Also be aware that there are both 32- and 64-bit builds of R for Windows with a combined distribution of binary packages, so the CRAN team will be reluctant to support a package that works under just one of the architectures.

### 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. However, such broad patterns are not recommended for production code: it is better to list all exports or use narrowly-defined groups. (As from R 2.13.0 this pattern applies to S4 classes, but did not in earlier versions of R.)

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 name spaces.

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 in 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 (provided it is exported from the name space) when detach is called on the package. There is also a hook .onUnLoad which is called when the name space is unloaded (via a call to unloadNamespace, perhaps called by detach(unload=TRUE)) with argument the full path to the installed package’s directory. .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 allows 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 the routine is invoked. Secondly, this removes any ambiguity in resolving symbols that might be present in several compiled DLLs.

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

\[
\text{useDynLib}(\text{foo}, \text{myRoutine_sym} = \text{myRoutine}, \text{myOtherRoutine})
\]

We could then call that routine from R using the command

\[
\text{.Call(myRoutine_sym, x, y)}
\]

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

\[
\text{.Call(dll$myRoutine, x, y)}
\]

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:

\[
\text{myDLL} = \text{useDynLib}(\text{foo}, \text{myRoutine_sym} = \text{myRoutine}, \text{myOtherRoutine})
\]

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 88), 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`:

\[
\text{R_CMethodDef cMethods[]} = \{
\text{"foo", (DL_FUNC) &foo, 4, \{REALSXP, INTSXP, STRSXP, LGLSXP\}},
\text{"bar_sym", (DL_FUNC) &bar, 0},
\text{NULL, NULL, 0}
\text{};}
\]

\[
\text{R_CallMethodDef callMethods[]} = \{
\text{"R_call_sym", (DL_FUNC) &R_call, 4},
\text{"R_version_sym", (DL_FUNC) &R_version, 0},
\text{NULL, NULL, 0}
\text{};}
\]
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>
"
```

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 \texttt{library(bar)} loads \texttt{bar} and attaches its exports to the search path. Package \texttt{foo} is also loaded but not attached to the search path. A call to \texttt{g} produces

\begin{verbatim}
> g(6)
[1] 1 13
\end{verbatim}

This is consistent with the definitions of \texttt{c} in the two settings: in \texttt{bar} the function \texttt{c} is defined to be equivalent to \texttt{sum}, but in \texttt{foo} the variable \texttt{c} refers to the standard function \texttt{c} in \texttt{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 \texttt{export} directives.
- Identify S3-style method definitions and write corresponding \texttt{S3method} declarations.
- Identify dependencies and replace any \texttt{require} calls by \texttt{import} directives (and make appropriate changes in the \texttt{Depends} and \texttt{Imports} fields of the ‘\texttt{DESCRIPTION}’ file).
- Replace \texttt{.First.lib} functions with \texttt{.onLoad} functions or \texttt{useDynLib} directives.

1.6.6 Name spaces with S4 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 \texttt{Depends: methods} in its ‘\texttt{DESCRIPTION}’ file and any classes and methods which are to be exported need to be declared in the ‘\texttt{NAMESPACE}’ file. For example, the \texttt{stats4} package has

\begin{verbatim}
export(mle)
importFrom("graphics", plot)
importFrom("stats", optim, qchisq)
## For these, we define methods or (AIC, BIC, nobs) an implicit generic:
importFrom("stats", AIC, BIC, coef, confint, logLik, nobs, profile,
    update, vcov)
exportClasses(mle, profile.mle, summary.mle)
## All methods for imported generics:
exportMethods(coef, confint, logLik, plot, profile, summary, show, update, vcov)
## implicit generics which do not have any methods here
export(AIC, BIC, nobs)
\end{verbatim}

All S4 classes to be used outside the package need to be listed in an \texttt{exportClasses} directive. Alternatively, they can be specified using \texttt{exportClassPattern} in the same style as for \texttt{exportPattern} Generics for which S4 methods are defined need to be declared in an \texttt{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 \texttt{base} name space).

\footnote{As from R 2.13.0 this defaults to the same pattern as \texttt{exportPattern}: use something like \texttt{exportClassPattern("\$")} to override this.}
Note that exporting methods on a generic in the name space will also export the generic, and exporting a generic in the name space 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 setting any methods for it (such as AIC in `stats4`), `exports` must be used.

Further, a package using S4 classes and methods defined in another package needs to import them, with directives

```r
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

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

and

```r
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 name space on those methods.

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

### 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.
• If your package has a ‘Makevars’ or ‘Makefile’ file, make sure that you use only portable make features. Such files should be LF-terminated (including the final line of the file) and not make use of GNU extensions. Commonly misused GNU extensions are conditional inclusions (ifeq and the like), ${shell ...} and ${wildcard ...}, and the use of += and :=. Also, the use of $< other than in implicit rules is a GNU extension. Unfortunately makefiles which use GNU extensions often run on other platforms but do not have the intended results.

The use of ${shell ...} can be avoided by using backticks, e.g.

```
PKG_CFLAGS = "gsl-config --cflags"
```

which works in all versions of make known\(^{35}\) to be used with R.

If you really must assume GNU make, declare it in the ‘DESCRIPTON’ file by

```
SystemRequirements: GNU make
```

Since the only viable make for Windows is GNU make, it is permissible to use GNU extensions in files ‘Makevars.win’ or ‘Makefile.win’.

• 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. This is particularly important for C++, where g++ -Wall -pedantic will alert you to the use of GNU extensions which fail to compile on most other C++ compilers. R assumes a C99 compiler as from version 2.12.0, but if you want your package to be portable to earlier versions you should write in C90. (In practice C99 has been available on most platforms since ca 2007 but old versions of gcc were still in use for R 2.11.x.)

If you use FORTRAN 77, 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\(^{36}\)) and typedef your own type to be long or long long, or use another suitable type (such as size_t).

It is not safe to assume that long and pointer types are the same size, and they are not on 64-bit Windows. If you need to convert pointers to and from integers use the C99 integer types intptr_t and uintptr_t.

Note that integer in FORTRAN corresponds to int in C on all R platforms.

• Under no circumstances should your compiled code ever call abort or exit: these terminate the user’s R process, quite possibly including all his unsaved work. One usage that could call abort is the assert macro in C or C++ functions: if you use assert during development, make doubly sure that NDEBUG is defined in the released package (maybe by adding it to PKG_CPPFLAGS).

\(^{35}\) GNU make, BSD make as in FreeBSD and bsdmake on Darwin, AT&T make as implemented on Solaris.

\(^{36}\) but note that long long is not a standard C++ type, and C++ compilers set up for strict checking will reject it.
This applies not only to your own code but to any external software you compile in or link to.

- 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 78 for a tool which can be used to look for this.

- 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
  \[\text{nm -pg mypkg.so} \quad \# \text{ or other extension such as `.s1'}\]

  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 \text{nm -pg}), and to use unusual names, as well as ensuring you have used the \text{PACKAGE} argument that \text{R CMD check} checks for.

- Do not use (hard or symbolic) file links in your package sources. \text{R CMD build} packages the tarball with the ‘-h’ \text{tar} flag which is documented to dereference links so this is not usually a problem, but versions 1.24 and later of GNU \text{tar} dereference some links to hard links which may not be handled correctly by \text{R CMD INSTALL}.

Some notes on issues specific to porting to x64 Windows are available at \url{http://www.stats.ox.ac.uk/~ripley/Win64/W64porting.html}.

### 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 ‘\text{DESCRIPTION}’ file and in ‘.Rd’ files, as discussed elsewhere in 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.

Function \text{showNonASCII} in package \text{tools} can help in finding non-ASCII bytes in files.

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 ‘\text{X11()}’ device needs to find a font that covers the glyphs used, and this will be machine-dependent: you may have Russian or Japanese fonts installed but your users may not. Using ‘\text{postscript}’ or ‘\text{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 ‘\text{encoding}’ argument. UTF-8-encoded text works reasonably well on Windows using the \text{Rgui} console and the \text{windows} family of devices, since they work internally in Unicode (UCS-2).

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 of your machine) can run R in different locales in different sessions, say ‘fr_CA.UTF-8’ one day and ‘fr_CH.iso88591’ the next. Declaring the encoding as either ‘latin1’ or ‘UTF-8’ in the ‘DESCRIPTION’ file will enable this to work.

From R 2.10.0 there is a fairly portable way to have arbitrary text in character strings (only) in your R code, which is to supply them in Unicode as \uxxxx escapes. If there are any characters not in the current encoding the parser will encode the character string as UTF-8 and mark it as such. A variation of this approach that has worked for Latin-1 text since R 2.5.0 is to enter the text with \x escapes and using Encoding(x) <- "latin1" to declare its encoding, but that may not work in future versions of R.

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 environment variable `R_ENCODING_LOCALES`. The default is equivalent to the value

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

(which is appropriate for a system based on glibc) except that if the current locale is UTF-8 when the package code is translated to UTF-8 for syntax checking.

### 1.7.2 Binary distribution

If you want to distribute a binary version of a package on Windows or Mac OS X, there are further checks you need to do to check it is portable: it is all too easy to depend on external software on your own machine that other users will not have.

For Windows, check what other DLLs your package’s DLL depends on (‘imports’ from in the DLL tools’ parlance). A convenient GUI-based tool to do so is ‘Dependency Walker’ ([http://www.dependencywalker.com/](http://www.dependencywalker.com/)) for both 32-bit and 64-bit DLLs – note that this will report as missing links to R’s own DLLs such as ‘R.dll’ and ‘Rblas.dll’. For 32-bit DLLs only, the command-line tool `pedump.exe -i` (in ‘Rtools*.exe’) can be used, and for the brave, the `objdump` tool in the appropriate toolchain will also reveal what DLLs are imported from. If you use a toolchain other than one provided by the R developers or use your own makefiles, watch out in particular for dependencies on the toolchain’s runtime DLLs such as ‘libgfortran’, ‘libstdc++’ and ‘libgcc_s’.

For Mac OS X, using `R CMD otool -L` on the package’s shared objects under ‘libs’ will show what they depend on: watch for any dependencies in ‘/usr/local/lib’, notably ‘libgfortran.2.dylib’.

### 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`

```r
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

```r
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

```c
#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

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

If you want to use different messages for singular and plural forms, you need to add

```c
#ifndef ENABLE_NLS
#define dngettext(pkg, String, StringP, N) (N > 1 ? StringP: String)
#endif
```

and mark strings by

```c
dngettext("pkg", <singular string>, <plural string>, n)
```

(This is only supported from R 2.10.0, so packages which use it need to depend on R (>= 2.10).)

- In the package’s ‘src’ directory run

```c
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`.

1.9.3 Installing translations

Once the template files have been created, translations can be made. Conventional translations have file extension ‘`.po`’ and are placed in the ‘`po`’ subdirectory of the package with a name that is either ‘`ll.po`’ or ‘`R-ll.po`’ for translations of the C and R messages respectively to language with code ‘`ll`’.

See Section “Localization of messages” in `R Installation and Administration`, for details of language codes.

Translations need to be prepared and installed in ‘`inst/po/`’ to be usable once the package is installed. To do this use the appropriate lines of

```
mkdir -p inst/po/ll/LC_MESSAGES
msgfmt -c --statistics -o inst/po/ll/LC_MESSAGES/R-pkg.mo po/R-ll.po
msgfmt -c --statistics -o inst/po/ll/LC_MESSAGES/pkg.mo po/ll.po
```
from the package’s top-level directory. Using ‘`-c`’ does some useful validity checks, and ‘`--statistics`’ notes the coverage.

1.9.4 Makefile support

There is some makefile support in the ‘`po`’ directory of the R sources. To use this to create the template files, use

```
mkdir -p pkgdir/po
```
where ‘`pkgdir`’ is the top-level directory of the package sources. If the package has C source files in its ‘`src`’ directory that are marked for translation, use

```
touch pkgdir/po/pkg.pot
```
to create a dummy template file. Then

```
cd R_BUILD_DIR/po
```
Chapter 1: Creating R packages

make pkg-update PKG=pkg PKDIR=pkgdir

will create a template file of R messages and update any template of C messages. It will also prepare and install a translation for the ‘en@quot’ pseudo-language, which if selected interprets (single and double) quotes in their directional forms in suitable (e.g. UTF-8) locales.

If translations to new languages are added in the ‘pkgdir/po’ directory, running the same make command will check and then install the translations.

If the package sources are updated, the same make command will update the template files, merge the changes into the translation ‘.po’ files and then installed the updated translations. You will often see that merging marks translations as ‘fuzzy’ and this is reported in the coverage statistics. As fuzzy translations are not used, this is an indication that the translation files need human attention.

This support is only for Unix-alikes, and the tools did not work correctly on at least one Mac OS X system.

1.10 CITATION files

An installed file named ‘CITATION’ will be used by the citation() function. (To be installed, it needed to be in the ‘inst’ subdirectory of the package sources.)

The ‘CITATION’ file is parsed as R code (in the package’s declared encoding, or in ASCII if none is declared). If no such file is present, citation auto-generates citation information from the package ‘DESCRIPTION’ metadata, and an example of what that would look like as a ‘CITATION’ file can be seen in recommended package nlme (see below): recommended packages boot, cluster and mgcv have further examples.

A ‘CITATION’ file will contain calls to function bibentry (new style, only works with R 2.12.0 or later), or to the functions citHeader, citEntry and (optionally) citFooter (old style).

Here is that for nlme, re-formatted:

```r
citHeader("To cite package 'nlme' in publications use:"")

## R >= 2.8.0 passes package metadata to citation().
if(!exists("meta") || is.null(meta)) meta <- packageDescription("nlme")
year <- sub(".*(2\[[:digit:]\]{3})-.*", \"\1\", meta$Date)
vers <- paste("R package version", meta$Version)

citEntry(entry="Manual",
          title = "nlme: Linear and Nonlinear Mixed Effects Models",
          author = personList(as.person("Jose Pinheiro"),
                               as.person("Douglas Bates"),
                               as.person("Saikat DebRoy"),
                               as.person("Deepayan Sarkar"),
                               person("R Development Core Team")),
          year = year,
          note = vers,
          )
```
Note the way that information that may need to be updated is picked up from the 'DESCRIPTION' file – it is tempting to hardcode such information, but it normally then gets outdated. See ?bibentry for further details of the information which can be provided.

The 'CITATION' file should itself produce no output when source-d.

1.11 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.

1.11.1 Frontend

This is a rather general mechanism, designed for adding new front-ends such as the former gnomeGUI package (see the 'Archive' area on CRAN). 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.

1.11.2 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
```
1.12 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 (32/64-bit) 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 (R-Forge can be selected by setRepositories and the GUI menus that use it). 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 much of which closely resembles (La)TeX, which can be processed into a variety of formats, including (La)TeX, HTML and plain text. The translation is carried out by functions in the tools package called by the script Rdconv in ‘R_HOME/bin’ and by the installation scripts for packages.

The R distribution contains more than 1300 such files which can be found in the ‘src/library/pkg/man’ directories of the R source tree, where pkg stands for one of the standard packages which are included in the R distribution.

As an example, let us look at a simplified version of ‘src/library/base/man/load.Rd’ which documents the R function load.

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 an optional footer with keyword information. The header is mandatory.
Information is given within a series of sections with standard names (and user-defined sections are also allowed). Unless otherwise specified these should occur only once in an ‘Rd’ file (in any order), and the processing software will retain only the first occurrence of a standard section in the file, with a warning.

See “Guidelines for Rd files” for guidelines for writing documentation in ‘Rd’ format which should be useful for package writers. The R generic function prompt is used to construct a bare-bones ‘Rd’ file ready for manual editing. Methods are defined for documenting functions (which fill in the proper function and argument names) and data frames. There are also functions promptData, promptPackage, promptClass, and promptMethods for other types of ‘Rd’ file.

The general syntax of ‘Rd’ files is summarized below. For a detailed technical discussion of current ‘Rd’ syntax, see “Parsing Rd files”. Note that there have been a number of changes to the ‘Rd’ format over the years, which can be important if a package is intended to be used with earlier versions of R: see earlier versions of this manual if a package is intended to be used with R before 2.10.0.

‘Rd’ files consists of three types of text input. The most common is \LaTeX-like, with the backslash used as a prefix on markup (e.g. \alias), and braces used to indicate arguments (e.g. \{load\}). The least common type of text is verbatim text, where no markup is processed. The third type is R-like, intended for R code, but allowing some embedded macros. Quoted strings within R-like text are handled specially: regular character escapes such as \n may be entered as-is. Only markup starting with \l (e.g. \link) or \v (e.g. \var) will be recognized within quoted strings. The rarely used vertical tab \v must be entered as \\v.

Each macro defines the input type for its argument. For example, the file initially uses \LaTeX-like syntax, and this is also used in the description section, but the usage section uses R-like syntax, and the alias macro uses verbatim syntax. Comments run from a percent symbol % to the end of the line in all types of text (as on the first line of the load example).

Because backslashes, braces and percent symbols have special meaning, to enter them into text sometimes requires escapes using a backslash. In general balanced braces do not need to be escaped, but percent symbols always do. For the complete list of macros and rules for escapes, see “Parsing Rd files”.

### 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 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. To avoid problems with indexing the package manual, it may not contain ’!’ , ’?’ or ’@’. (\LaTeX special characters are allowed, but may not be collated correctly in the index.) There can only be one \name entry in a file.

---

1. e.g. \alias, \keyword and \note sections.
2. 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-insensitive file system.
and it must not contain any markup. Entries in the package manual will be in alphabetic order of the \code{name} entries.

\alias{topic}

The \code{alias} sections 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 \code{topic} can contain spaces, but (for historical reasons) leading and trailing spaces will be stripped. Percent and left brace need to be escaped by a backslash.

There may be several \code{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

\code{Name{Normal}}
\code{Alias{Normal}}
\code{Alias{dnorm}}
\code{Alias{pnorm}}
\code{Alias{qnorm}}
\code{Alias{rnorm}}

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

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

\title{Title}

Title information for the ‘Rd’ file. This should be capitalized and not end in a period; try to limit its length to at most 65 characters for widest compatibility. Since R version 2.12.0 markup has been supported in the text, but use of characters other than English text and punctuation (e.g., ‘<’) may limit portability. There must be one (and only one) \code{title} section in a help file.

\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.) This is mandatory except for package-overview files.

\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 an R-like command. 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 \code{synopsis} for the actual synopsis and show modified synopses in the \code{usage}. Support for \code{synopsis} will be removed eventually.

---

3 in the current locale, and with special treatment for \LaTeX{} special characters and with any ‘pkgname-package’ topic moved to the top of the list.
To indicate that a function can be used in several different ways, depending on the named arguments specified, use section \details. E.g., \texttt{abline.Rd} contains

\begin{verbatim}
\details{
  Typical usages are
  \preformatted{
    abline(a, b, untf = FALSE, \dots)
    \ldots
  }
}
\end{verbatim}

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

For example, \texttt{print.ts.Rd} contains

\begin{verbatim}
\usage{
  \method{print}{ts}(x, calendar, \dots)
}
\end{verbatim}

which will print as

\textbf{Usage:}

\begin{verbatim}
## S3 method for class 'ts':
print(x, calendar, \ldots)
\end{verbatim}

Usage for replacement functions should be given in the style of \texttt{dim(x) <- value} rather than explicitly indicating the name of the replacement function ("\texttt{dim<-}" in the above). Similarly, one can use \texttt{\method{generic}{class}(arglist) <- value} to indicate the usage of an S3 replacement method for the generic replacement function "\texttt{generic<-}" for objects inheriting from class \texttt"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 ("\texttt{\%\%\%}\") follows the above rules, using the appropriate function names. E.g., \texttt{Extract.factor.Rd} contains

\begin{verbatim}
\usage{
  \method{[}{factor}(x, \dots, drop = FALSE)
  \method{[[}{factor}(x, \dots)
  \method{[}{factor}(x, \dots) <- value
}
\end{verbatim}

which will print as
Usage:

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

\S3method is accepted as an alternative to \method.

\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. (Note that there is no whitespace between
the three parts of the entry.) There may be optional text outside the \item entries, for example to give general information about groups of parameters.

\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\footnote{Text between or after list items was discarded prior to R 2.10.0, and is discouraged.} this list
(see for example the help for \code{rle}). Note that \value is implicitly a \describe environment, so that environment should not be used for listing components,
just individual \item{} entries.

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

\note{...}
Use this for a special note you want to have pointed out. Multiple \note
sections are allowed, but might be confusing to the end users.

For example, 'pie.Rd' contains

```r
\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.
   .......
}
```

\footnote{Text between or after list items was discarded prior to R 2.10.0, and is discouraged.}
\author{...}
Information about the author(s) of the ‘Rd’ file. Use \email{} without extra delimiters (such as ‘( )’ or ‘< >’) to specify email addresses, or \url{} or \href{} 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 55, and Section 2.5 [Cross-references], page 58).

\examples{...}
Examples of how to use the function. Code in this section is set in typewriter font without reformatting and is run by \code{example()} unless marked otherwise (see below).

Examples are not only useful for documentation purposes, but also provide test code used for diagnostic checking of R code. By default, text inside \examples{} will be displayed in the output of the help page and run by \code{example()} and by R CMD check. You can use \code{\dontrun{}} for text that should only be shown, but not run, and \code{\dontshow{}} for extra commands for testing that should not be shown to users, but will be run by \code{example()}. (Previously this was called \code{\testonly}, and that is still accepted.)

Text inside \code{\dontrun{}} is verbatim, but the other parts of the \examples{} section are R-like text.

For example,

\begin{verbatim}
x <- runif(10) # Shown and run.
\dontrun{plot(x)} # Only shown.
\dontshow{log(x)} # Only run.
\end{verbatim}

Thus, example code not included in \code{\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). Text included in \code{\dontrun{}} is indicated by comments in the processed help files: it need not be valid R code but the escapes must still be used for \%, \ and unpaired braces as in other verbatim text.

Data needed for making the examples executable can be obtained by random number generation (for example, \code{x <- rnorm(100)}, or by using standard data sets listed by \code{data()} (see ?data for more info).

Finally, there is \code{\donttest}, used (at the beginning of a separate line) to mark code that should be run by \code{examples()} but not by R CMD check. This should be needed only occasionally but can be used for code which might fail in circumstances that are hard to test for, for example in some locales. (Use e.g. \code{capabilities()} to test for features needed in the examples wherever possible, and you can also use \code{try()} or \code{trycatch()}.)

\keyword{key}
There can be zero or more \keyword{} sections per file. Each \keyword{} section should specify a single keyword, preferably one of the standard
keywords as listed in file ‘KEYWORDS’ in the R documentation directory (default ‘R_HOME/doc’). Use e.g. RShowDoc("KEYWORDS") to inspect the standard keywords from within R. There can be more than one \keyword entry if the R object being documented falls into more than one category, or none.

The special keyword ‘internal’ marks a page of internal objects that are not part of the package’s 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, including the alphabetical list of objects in the HTML help system.

help.search() can search by keyword, including user-defined values: however the ‘Search Engine & Keywords’ HTML page accessed via help.start() provides single-click access only to a pre-defined list of keywords.

### 2.1.2 Documenting data sets

The structure of ‘Rd’ files which document R data sets is slightly different. Sections such as \arguments and \value are not needed but 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{
}
\keyword{datasets}
```

This uses the following additional markup commands.

\docType{...}

Indicates the “type” of the documentation object. Always ‘data’ for data sets, and ‘package’ for ‘pkg-package.Rd’ overview files. Documentation for S4 methods and classes uses ‘methods’ (from promptMethods()) and ‘class’ (from promptClass()).

\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 57, for more information.
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 should always be ‘datasets’.

If bar is a data frame, documenting it as a data set can be initiated via prompt(bar). Otherwise, the promptData function may be used.

### 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 make 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.

An S4 replacement method is documented in the same way as an S3 one: see the description of \method in Section 2.1.1 [Documenting functions], page 48.

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 help 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 up to `library(help = pkgname)`. Otherwise (the default) comments will be inserted giving suggestions for content.

Apart from the mandatory `name` and `title` and the `pkgname-package` alias, the only requirement for the package overview 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 26) 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

```latex
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). Whitespace between the first and second braced expressions is not allowed. Markup (e.g. `code{}`) within the section title may cause problems with the latex conversion (depending on the version of macro packages such as ‘hyperref’) and so should be avoided.

The `subsection` macro takes arguments in the same format as `section`, but is used within a section, so it may be used to nest subsections within sections or other subsections. There is no predefined limit on the nesting level, but formatting is not designed for more than 3 levels (i.e. subsections within subsections within sections).

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 amongst themselves as in 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 regarded as stronger (more emphatic).

- `\bold{text}`

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

Portably single or double quote text (without hard-wiring the characters used for quotation marks).

Each of the above commands takes \LaTeX-like input, so other macros may be used within text.

The following logical markup commands are available for indicating specific kinds of text. Except as noted, these take verbatim text input, and so other macros may not be used within them. Some characters will need to be escaped (see Section 2.7 [Insertions], page 60).

\code{text}

Indicate text that is a literal example of a piece of an R program, e.g., a fragment of R code or the name of an R object. Text is entered in R-like syntax, and displayed using typewriter font if possible. Macros \var and \link are interpreted within text.

\preformatted{text}

Indicate text that is a literal example of a piece of a program. Text is displayed using typewriter font if possible. Formatting, e.g. line breaks, is preserved. Due to limitations in \LaTeX as of this writing, this macro may not be nested within other markup macros other than \dQuote and \sQuote, as errors or bad formatting may result.

\kbd{keyboard-characters}

Indicate keyboard input, using slanted typewriter font if possible, so users can distinguish the characters they are supposed to type from computer output. Text is entered verbatim.

\samp{text}

Indicate text that is a literal example of a sequence of characters, entered verbatim. No wrapping or reformatting will occur. Displayed using typewriter font if possible.

\verb{text}

Indicate text that is a literal example of a sequence of characters, with no interpretation of e.g. \var, but which will be included within word-wrapped text. Displayed using typewriter font if possible.

\pkg{package_name}

Indicate the name of an R package. \LaTeX-like.

\file{file_name}

Indicate the name of a file. Text is \LaTeX-like, so backslash needs to be escaped. Displayed using a distinct font if possible.

\email{email_address}

Indicate an electronic mail address. \LaTeX-like, will be rendered as a hyperlink in HTML and PDF conversion. Displayed using typewriter font if possible.
\url{uniform_resource_locator}

Indicate a uniform resource locator (URL) for the World Wide Web. The argument is handled verbatim, and rendered as a hyperlink in HTML and PDF conversion. Displayed using typewriter font if possible.

\href{uniform_resource_locator}{text}

Indicate a hyperlink to the World Wide Web. The first argument is handled verbatim, and is used as the URL in the hyperlink, with the second argument of \LaTeX-like text displayed to the user.

\var{metasyntactic_variable}

Indicate a metasyntactic variable. In some cases this will be rendered distinctly, e.g. in italic, but not in all. \LaTeX-like.

\env{environment_variable}

Indicate an environment variable. Verbatim. Displayed using typewriter font if possible.

\option{option}

Indicate a command-line option. Verbatim. Displayed using typewriter font if possible.

\command{command_name}

Indicate the name of a command. \LaTeX-like, so \var is interpreted. Displayed using typewriter font if possible.

\dfn{term}

Indicate the introductory or defining use of a term. \LaTeX-like.

\cite{reference}

Indicate a reference without a direct cross-reference via \link (see Section 2.5 [Cross-references], page 58), such as the name of a book. \LaTeX-like.

\acronym{acronym}

Indicate an acronym (an abbreviation written in all capital letters), such as GNU. \LaTeX-like.

### 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).

Note that unlike argument lists, \item in these formats is followed by a space and the text (not enclosed in braces). For example

\begin{verbatim}
\enumerate
  \item A database consists of one or more records, each with one or more named fields.
  \item Regular lines start with a non-whitespace character.
\end{verbatim}

\footnote{Currently it is rendered differently only in HTML conversions, and \LaTeX conversion outside \usage and \examples environments.}
\item Records are separated by one or more empty lines.

\itemize and \enumerate commands may be nested.

The \describe command is similar to \itemize but allows initial labels to be specified. Each \item takes two arguments, the label and the body of the item, in exactly the same way as an argument or value \item. \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:

\begin{verbatim}
\begin{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)
\end{tabular}
\end{verbatim}

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). (There is no white-space between \tabular and the first argument, nor between the two arguments.)

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 format.

One main usage of \link is in the \seealso section of the help page, see Section 2.1 [Rd format], page 47.

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 in the source file, \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\(^6\) (in which case the present package
has precedence so this is only needed to refer to other packages). They are currently only
used in HTML help (and ignored for hyperlinks in \LaTeX\ 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). The only reason to use these forms for base and
recommended packages is to force a reference to a package that might be further down the
search path. Because they have been frequently misused, the HTML help system looks for
topic \textit{foo} in package \textit{pkg} if it does not find file ‘\textit{foo.html’}.

\section*{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
\texttt{\textbackslash ebx{latex}{ascii}} and \texttt{\textbackslash ebx{latex}{ascii}} are used. Whereas \texttt{\textbackslash ebx} is used for
“inline” formulae (corresponding to \TeX\’s \$...\$), \texttt{\textbackslash ebx} gives “displayed equations” (as
in \LaTeX\’s \texttt{displaymath} environment, or \TeX\’s \texttt{$...$}). Both arguments are treated as
verbatim text.

Both commands can also be used as \texttt{\textbackslash ebx{latexascii}} (only one argument) which then
is used for both \texttt{latex} and \texttt{ascii}. No whitespace is allowed between command and the first
argument, nor between the first and second arguments.

The following example is from ‘Poisson.Rd’:
\begin{verbatim}
\texttt{\textbackslash ebx{p(x) = \frac{\lambda^x e^{-\lambda}}{x!}}}\% \\
\texttt{p(x) = \lambda^x \exp(-\lambda)/x!}
\end{verbatim}

for \texttt{x = 0, 1, 2, \ldots}.

For the \LaTeX\ manual, this becomes
\begin{verbatim}
 p(x) = \lambda^x e^{-\lambda} \\
\textit{for } x = 0, 1, 2, \ldots
\end{verbatim}

For text on-line help we get
\begin{verbatim}
 p(x) = lambda^x \exp(-lambda)/x! \\
\textit{for } x = 0, 1, 2, \ldots
\end{verbatim}

Greek letters (both cases) will be rendered in HTML if preceded by a backslash, \texttt{\textbackslash dots}
and \texttt{\textbackslash ldots} will be rendered as ellipses and \texttt{\textbackslash sqrt}, \texttt{\textbackslash ge} and \texttt{\textbackslash le} as mathematical symbols.

Note that only basic \LaTeX\ can be used, there being no provision to specify \LaTeX\ style
files such as the AMS extensions.

\footnotemark\footnotetext{a common example in CRAN packages is \texttt{\link[mgcv]{gam}}.}
2.7 Insertions

Use \R for the R system itself. Use \dots for the dots in function argument lists ‘…’, and \ldots for ellipsis dots in ordinary text. These can be followed by {}, and should be unless followed by whitespace.

After an unescaped ‘%’, you can put your own comments regarding the help text. The rest of the line (but not the newline at the end) will be completely disregarded. 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 almost always need to be escaped by ‘\’, and ‘\’ can be used for backslash and needs to be when there two or more adjacent backslashes). In R-like code quoted strings are handled slightly differently; see “Parsing Rd files” for details – in particular braces should not be escaped in quoted strings.

All of ‘% { } \’ should be escaped in \LaTeX-like text.

Text which might need to be represented differently in different encodings should be marked by \enc, e.g. \enc{Jöreskog}{Joreskog} (with no whitespace between the braces) 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 in locales that cannot represent the encoded form). (This is intended to be used for individual words, not whole sentences or paragraphs.)

2.8 Indices

The \alias command (see Section 2.1.1 [Documenting functions], page 48) 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 54). 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. ??Spearman will succeed in finding the help page for the test for association between paired samples using Spearman’s ρ.

(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.

---

7 There is only a fine distinction between \dots and \ldots. It is technically incorrect to use \ldots in code blocks and tools::checkRd will warn about this—on the other hand the current converters treat them the same way in code blocks, and elsewhere apart from the small distinction between the two in \LaTeX.

8 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. Such blocks should not be nested, and should be entirely within a block (that is, between the opening and closing brace of a section or item), or at top-level contain one or more complete sections.

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 Conditional text

Occasionally the best content for one output format is different from the best content for another. For this situation, the \if{format}{text} or \ifelse{format}{text}{alternate} markup is used. Here format is a comma separated list of formats in which the text should be rendered. The alternate will be rendered if the format does not match. Both text and alternate may be any sequence of text and markup.

Currently the following formats are recognized: example, html, latex and text. These select output for the corresponding targets. (Note that example refers to extracted example code rather than the displayed example in some other format.) Also accepted are TRUE (matching all formats) and FALSE (matching no formats). These could be the output of the \Sexpr macro (see Section 2.11 [Dynamic pages], page 61).

The \out{literal} macro would usually be used within the text part of \if{format}{text}. It causes the renderer to output the literal text exactly, with no attempt to escape special characters. For example, use the following to output the markup necessary to display the Greek letter in \LaTeX or HTML, and the text string alpha in other formats:

```
\if{latex}{\out{\alpha}}\ifelse{html}{\out{α}}{alpha}
```

2.11 Dynamic pages

Two new macros supporting dynamically generated man pages were introduced in R 2.10.0, \Sexpr and \RdOpts. These are modelled after Sweave, and are intended to contain executable R expressions in the ‘Rd’ file.

The main argument to \Sexpr must be valid R code that can be executed. It may also take options in square brackets before the main argument. Depending on the options, the code may be executed at package build time, package install time, or man page rendering time.
The options follow the same format as in Sweave, but different options are supported. Currently the allowed options and their defaults are:

- **eval=TRUE** Whether the R code should be evaluated.
- **echo=FALSE** Whether the R code should be echoed. If TRUE, a display will be given in a preformatted block. For example, `\Sexpr{echo=TRUE}{ x <- 1 }` will be displayed as
  
  \[
  > x <- 1
  \]
- **keep.source=TRUE** Whether to keep the author's formatting when displaying the code, or throw it away and use a deparsed version.
- **results=text** How should the results be displayed? The possibilities are:
  - **results=text** Apply `as.character()` to the result of the code, and insert it as a text element.
  - **results=verbatim** Print the results of the code just as if it was executed at the console, and include the printed results verbatim. (Invisible results will not print.)
  - **results=rd** The result is assumed to be a character vector containing markup to be passed to `parse_Rd(fragment=TRUE)`, with the result inserted in place. This could be used to insert computed aliases, for instance.
  - **results=hide** Insert no output.
- **strip.white=TRUE** Remove leading and trailing white space from each line of output if `strip.white=TRUE`. With `strip.white=all`, also remove blank lines.
- **stage=install** Control when this macro is run. Possible values are
  - **stage=build** The macro is run when building a source tarball.
  - **stage=install** The macro is run when installing from source.
  - **stage=render** The macro is run when displaying the help page.

Conditionals such as `#ifdef` (see Section 2.9 [Platform-specific sections], page 61) are applied after the `build` macros but before the `install` macros. In some situations (e.g. installing directly from a source directory without a tarball, or building a binary package) the above description is not literally accurate, but authors can rely on the sequence being `build`, `#ifdef`, `install`, `render`, with all stages executed.

Code is only run once in each stage, so a `\Sexpr{results=rd}` macro can output an `\Sexpr` macro designed for a later stage, but not for the current one or any earlier stage.

- **width, height, fig** These options are currently allowed but ignored.

The `\RdOpts` macro is used to set new defaults for options to apply to following uses of `\Sexpr`.

For more details, see the online document “Parsing Rd files”.

### 2.12 User-defined macros

Two new macros supporting user-defined macros were introduced in R 2.12.0. The `\newcommand` and `\renewcommand` macros allow new macros to be defined within an Rd file. These are similar but not identical to the same-named LaTeX macros.
They each take two arguments which are parsed verbatim. The first is the name of the new macro including the initial backslash, and the second is the macro definition. As in \LaTeX, \texttt{\newcommand} requires that the new macro not have been previously defined, whereas \texttt{\renewcommand} allows existing macros (including all built-in ones) to be replaced.

Also as in \LaTeX, the new macro may be defined to take arguments, and numeric placeholders such as \texttt{#1} are used in the macro definition. However, unlike \LaTeX, the number of arguments is determined automatically from the highest placeholder number seen in the macro definition. For example, a macro definition containing \texttt{#1} and \texttt{#3} (but no other placeholders) will define a three argument macro (whose second argument will be ignored). As in \LaTeX, at most 9 arguments may be defined. If the \# character is followed by a non-digit it will have no special significance. All arguments to user-defined macros will be parsed as verbatim text, and simple text-substitution will be used to replace the placeholders, after which the replacement text will be parsed.

For example, the ‘NEWS.Rd’ file currently uses the definition
\begin{verbatim}
\newcommand{\PR}{\Sexpr[results=rd]{tools:::Rd_expr_PR(#1)}}
\end{verbatim}
which defines \texttt{PR} to be a single argument macro; then code like
\begin{verbatim}
PR{1234}
\end{verbatim}
will expand to
\begin{verbatim}
\Sexpr[results=rd]{tools:::Rd_expr_PR(1234)}
\end{verbatim}
when parsed.

### 2.13 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 an \texttt{\encoding{}} section must be used to specify the encoding if it is not ASCII. (The \texttt{\encoding{}} section must be on a line by itself, and in particular one containing no non-ASCII characters. The encoding declared in the ‘DESCRIPTION’ file will be used if none is declared in the file.) The ‘Rd’ files are converted to UTF-8 before parsing and so the preferred encoding for the files themselves is now UTF-8.

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). (Function \texttt{showNonASCII} in package \texttt{tools} can help in finding non-ASCII bytes in the files.)

For convenience, encoding names ‘latin1’ and ‘latin2’ are always recognized: these and ‘UTF-8’ are likely to work fairly widely. However, this does not mean that all characters in UTF-8 will be recognized, and the coverage of non-Latin characters\footnote{R 2.9.0 added support for UTF-8 Cyrillic characters in \LaTeX, but on some OSes this will need Cyrillic support added to \LaTeX, so environment variable \texttt{_R_CYRILLIC_TEX_} needs to be set to a non-empty value to enable this.} is low.

The \texttt{\enc} command (see Section 2.7 [Insertions], page 60) can be used to provide transliterations which will be used in conversions that do not support the declared encoding.

The \LaTeX{} conversion converts the file to UTF-8 from the declared encoding, and includes
\texttt{\inputencoding{utf8}}

command, and this needs to be matched by a suitable invocation of the \texttt{\usepackage{inputenc}} command. The \text{R} utility \texttt{R CMD Rd2dvi} looks at the converted code and includes the encodings used: it might for example use \texttt{\usepackage[utf8]{inputenc}}

(Use of utf8 as an encoding requires \LaTeX{} dated 2003/12/01 or later. Also, the use of Cyrillic characters in ‘UTF-8’ appears to also need ‘\usepackage[T2A]{fontenc}’, and \texttt{R CMD Rd2dvi} includes this conditionally on the file ‘t2aenc.def’ being present and environment variable \_R\_CYRILLIC\_TEX\_ being set.)

Note that this mechanism works best with Latin letters: the coverage of UTF-8 in \LaTeX{} is quite low.

### 2.14 Processing Rd format

There are several commands to process Rd files from the system command line.

Using \texttt{R CMD Rdconv} one can convert \text{R} documentation format to other formats, or extract the executable examples for run-time testing. The currently supported conversions are to plain text, HTML and \LaTeX{} as well as extraction of the examples.

\texttt{R CMD Rd2dvi} generates DVI (or, if option ‘--pdf’ is given, or it is invoked as \texttt{R CMD Rd2pdf}, PDF) output from documentation in ‘Rd’ files, which can be specified either explicitly or by the path to a directory with the sources of a package. In the latter case, a reference manual for all documented objects in the package is created, including the information in the ‘\texttt{DESCRIPTION}’ files.

\texttt{R CMD Sweave} and \texttt{R CMD Stangle} process ‘Sweave’ documentation files (usually with extension ‘.Snw’ or ‘.Rnw’): \texttt{R CMD Stangle} is use to extract the \text{R} code fragments.

The exact usage and a detailed list of available options for all of these commands can be obtained by running \texttt{R CMD command --help}, e.g., \texttt{R CMD Rdconv --help}. All available commands can be listed using \texttt{R --help} (or \texttt{Rcmd --help} under Windows).

All of these work under Windows. You will need to have installed the 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. By default code entered by users has the source code stored internally, 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 removed from 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. If the original source does not follow the standard format this tidied version can be much easier to read.

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 calling the `removeSource()` function, for example by

   ```r
   myfun <- removeSource(myfun)
   ```

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

```r
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. Warning: comments in your functions will be lost.

The standard format provides a good starting point for further tidying. Although the deparsing cannot do so, we recommend the consistent use of the preferred assignment operator ‘<-’ (rather than ‘=’) for assignment. Many package authors use a version of Emacs (on a Unix-alike 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-alike 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)

---

\(^1\) 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.
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, pp. 225–6).

```r
library(MASS); library(boot)
storm.fm <- nls(Time ~ b*Viscosity/(Wt - c), stormer,
  start = c(b=30.401, 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) # slow enough to profile
Rprof(NULL)

Having run this we can summarize the results by

```
R CMD Rprof boot.out
```
(Function names are not quoted on Windows.) 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.

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

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 \texttt{gc()} with a profiling run without a preceding call to \texttt{gc}.

More detailed analysis of the output can be achieved by the tools in the CRAN packages \texttt{proftools} and \texttt{profr}: in particular these allow 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 \texttt{gc()} shows a division of memory into \texttt{Vcells} used to store the contents of vectors and \texttt{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,

\begin{verbatim}
y <- x + 1
\end{verbatim}

allocates memory for a new vector \texttt{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 \texttt{x$call} will cause the entire object \texttt{x} to be copied. This copying to preserve the call-by-value illusion is usually done by the internal C function \texttt{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 \texttt{Rprof}

The sampling profiler \texttt{Rprof} described in the previous section can be given the option \texttt{memory.profiling=TRUE}. It then writes out 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 \texttt{duplicate}, which is called to copy R objects. \texttt{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 \texttt{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 \texttt{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, \texttt{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

```r
> 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 \texttt{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 \texttt{tracemem} on an object marks it so that a message is printed to standard output when the object is copied via \texttt{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 \texttt{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->0x92a6d90]: $<-.data.frame $<- statistic boot
memtrace[0x92a6d90->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/](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
```

Chapter 3: Tidying and profiling R code 69
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
% (optional, some platforms) opcontrol --callgraph=5
% 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, and `opreport -c` to generate a callgraph (if collection was enabled and the platform supports this).

### 3.4.2 Solaris

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

### 3.4.3 Mac OS X

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

```r
> 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"
Browse[1]> 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.

---

\(^1\) With the exceptions of the commands listed below: an object of such a name can be printed via an explicit call to `print`. 
• **where**

This prints the call stack. For example

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

```r
> 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`)

```r
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)
```
Chapter 4: Debugging

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

```r
> 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

```r
> 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

```r
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, etastart = etastart,
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" "n goedobs" "nobs" "n vars"
[26] "offset" "start" "valid eta" "valid mu" " variance"
[31] "var mu" "w" "weights" "x" "x names"
[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
1.000000 0.9999978 0.99999441 0.99997206 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

```r
> 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()`:

```r
> 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

```r
> 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)

```r
debug: {
## lists the whole function
Browse[1]>
debug: x <- as.matrix(x)
```  
```r
##
Browse[1]>
```
```r
debug: start
[1] -2.235357e-06
```
```r
debug: eta <- drop(x %*% start)
```
```r
Browse[1]>
```
```r
eta
```
```r
 1 2 3 4 5
```
```r
0.000000e+00 -2.235357e-06 -1.117679e-05 -5.588393e-05 -2.794197e-04
```
```r
6 7 8 9
```
```r
-1.397098e-03 -6.985492e-03 -3.492746e-02 -1.746373e-01
```  
```r
Browse[1]>
```
```r
debug: mu <- linkinv(eta <- eta + offset)
```
```r
Browse[1]>
```
```r
mu
```
```r
 1 2 3 4 5 6 7 8 9
```
```r
1.000000 0.9999978 0.9999888 0.9999441 0.9997206 0.9986039 0.9930389 0.9656755
```
```r
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

```r
## 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, unfortu-
```

The reference to ‘memory protection’ is to missing C-level calls to PROTECT/UNPROTECT (see Section 5.9.1 [Garbage Collection], page 98) 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.
Chapter 4: Debugging

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`.

The function `gctorture2` provides more refined control over the GC torture process. Its arguments `step`, `wait` and `inhibit_release` are documented on its help page. Environment variables can also be used to turn on GC torture: `R_GCTORTURE` corresponds to the `step` argument to `gctorture`, `R_GCTORTURE_WAIT` to `wait`, and `R_GCTORTURE_INHIBIT_RELEASE` to `inhibit_release`.

If R is configured with `--enable-strict-barrier` then a variety of tests for the integrity of the write barrier are enabled. In addition tests to help detect protect issues are enabled as well:

- All GCs are full GCs.
- New nodes in small node pages are marked as `NEWSXP` on creation.
- After a GC all free nodes that are not of type `NEWSXP` are marked as type `FREESXP` and their previous type is recorded.
- Most calls to accessor functions check their `SEXP` inputs and `SEXP` outputs and signal an error if a `FREESXP` is found. The address of the node and the old type are included in the error message.

Used with a debugger and with `gctorture` or `gctorture2` this mechanism can be helpful in isolating memory protect problems.

### 4.3.2 Using valgrind

If you have access to Linux on an ‘ix86’, ‘x86_64’, ‘ppc32’ or ‘ppc64’ platform, or Mac OS 10.5.x (‘Leopard’) on ‘i386’ you can use valgrind ([http://www.valgrind.org/](http://www.valgrind.org/), pronounced to rhyme with ‘tinned’) to check for possible problems. To run some examples under valgrind use something like

```r
R -d valgrind --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. Earlier versions (at least) 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 conjunction with **gctorture** can be even more effective (and even slower).

An example of **valgrind** output is

```
==12539== Invalid read of size 4
==12539==  at 0x1CFD6CBE: csc_compTr (Mutils.c:273)
==12539==  by 0x1CE07E1E: tsc_transpose (dtCMatrix.c:25)
==12539==  by 0x80A67A7: do_dotcall (dotcode.c:858)
==12539==  by 0x80CACE2: R_eval (eval.c:400)
==12539==  by 0x80CB5AF: R_execClosure (eval.c:658)
==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: R_eval (eval.c:403)
==12539==  by 0x80CB2F0: Gf_applyClosure (eval.c:573)
==12539==  by 0x80CADCC: R_eval (eval.c:414)
==12539==  by 0x80CA1A3: R_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: R_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 81).

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-compliant 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 0x00397e8a 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
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 \texttt{R -d gdb}.
- Start R.
- At the R prompt, use \texttt{dyn.load} or \texttt{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 \texttt{signal RET}.

Under Windows signals may not be able to be used, and if so the procedure is more complicated. See the rw-FAQ and \url{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 \texttt{PrintValue(SEXP \textit{s})} which uses the normal R printing mechanisms to print the R object pointed to by \textit{s}, or the safer version \texttt{R_PV(SEXP \textit{s})} which will only print ‘objects’.

One way to make use of \texttt{PrintValue} is to insert suitable calls into the code to be debugged.

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

\begin{verbatim}
(gdb) p R_PV(ab)
\end{verbatim}

using the object \texttt{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

\begin{verbatim}
R> DF <- data.frame(a = 1:3, b = 4:6)
\end{verbatim}

By setting a breakpoint at \texttt{do_get} and typing \texttt{get("DF")} at the R prompt, one can find out the address in memory of \texttt{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}
    }
},
$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
Another alternative available from R 2.13.0 on is the \texttt{R\_inspect} function which shows the low-level structure of the objects recursively (addresses differ from the above as this example is created on another machine):

\begin{verbatim}
(gdb) p R_inspect($1)
@100954d18 19 VECSXP g0c2 [OBJ,NAM(2),ATT] (len=2, tl=0)
@100954d50 13 INTSXP g0c2 [NAM(2)] (len=3, tl=0) 1,2,3
@100954d88 13 INTSXP g0c2 [NAM(2)] (len=3, tl=0) 4,5,6
ATTRIB:
@102a70140 02 LISTSXP g0c0 []
TAG: @10083c478 01 SYMSXP g0c0 [MARK,NAM(2),gp=0x4000] "names"
@100954dc0 16 STRSXP g0c2 [NAM(2)] (len=2, tl=0)
@10099df28 09 CHARXP g0c1 [MARK,gp=0x21] "a"
@10095e518 09 CHARXP g0c1 [MARK,gp=0x21] "b"
TAG: @100859e60 01 SYMSXP g0c0 [MARK,NAM(2),gp=0x4000] "row.names"
@102a6f868 13 INTSXP g0c1 [NAM(2)] (len=1, tl=1) -2147483648,-3
TAG: @10083c948 01 SYMSXP g0c0 [MARK,gp=0x4000] "class"
@102a6f838 16 STRSXP g0c1 [NAM(2)] (len=1, tl=1)
@1008c6d48 09 CHARXP g0c2 [MARK,gp=0x21,ATT] "data.frame"
\end{verbatim}

In general the representation of each object follows the format:

\begin{verbatim}
@<address> <type-nr> <type-name> <gc-info> [<flags>] ...
\end{verbatim}

For a more fine-grained control over the depth of the recursion and the output of vectors \texttt{R\_inspect} takes additional two integer parameters: maximum depth and the maximal number of elements that will be printed for scalar vectors. The defaults in \texttt{R\_inspect} are currently -1 (no limit) and 5 respectively.
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-alike 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 86). 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 92).

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"). Use the name you pass to .C or .Fortran rather than the translated symbol name.)

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.

<table>
<thead>
<tr>
<th>R storage mode</th>
<th>C type</th>
<th>FORTRAN type</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>int *</td>
<td>INTEGER</td>
</tr>
<tr>
<td>integer</td>
<td>int *</td>
<td>INTEGER</td>
</tr>
<tr>
<td>double</td>
<td>double *</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>complex</td>
<td>Rcomplex *</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td>character</td>
<td>char **</td>
<td>CHARACTER*255</td>
</tr>
<tr>
<td>raw</td>
<td>unsigned char *</td>
<td>none</td>
</tr>
</tbody>
</table>

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 this is stored in a way compatible
with the C99 double complex type: however, it may not be possible to pass Rcomplex to a C99 function expecting a double complex argument. Nor need it be compatible with a C++ complex type. Moreover, the compatibility can depend on the optimization level set for the compiler.

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 existing C or FORTRAN code.

Logical values are sent as 0 (FALSE), 1 (TRUE) or INT_MIN = -2147483648 (NA, but only if NAOK is true), and the compiled code should return one of these three values. (Non-zero values other than INT_MIN are mapped to TRUE.)

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 and .Fortran 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: 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-alikes including Mac OS X, see Section 5.5 [Creating shared objects], page 90 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
for platform independence. On Unix-alike systems the path supplied to \texttt{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 \textit{via} a call to \texttt{library.dynam} in the \texttt{.First.lib} function of a package. This has the form

\begin{verbatim}
library.dynam("libname", package, lib.loc)
\end{verbatim}

where \texttt{libname} is the object/DLL name \textit{with the extension omitted}. Note that the first argument, \texttt{chname}, should \textbf{not} be \texttt{package} since this will not work if the package is installed under another name.

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

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

\begin{verbatim}
library.dynam("mylib", package, lib.loc)
\end{verbatim}

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

The following example shows templates for the initialization and unload routines for the \texttt{mylib} DLL.

\begin{verbatim}
#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. */
}
\end{verbatim}
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 and registration (see the next section) provide good ways to avoid any ambiguity in which occurrence is meant.

On Unix-alikes the paths used to resolve dynamically linked dependent libraries are fixed (for security reasons) when the process is launched, so `dyn.load` will only look for such libraries in the locations set by the `R` shell script (via `etc/ldpaths`) and in the OS-specific defaults.

Windows allows more control (and less security) over where dependent DLLs are looked for. On all versions this includes the `PATH` environment variable, but with lowest priority: note that it does not include the directory from which the DLL was loaded. On XP and later it is possible\(^1\) to add a single path with quite high priority via the `DLLpath` argument to `dyn.load`. This is (by default) used by `library.dynam` to include the package’s ‘`libs`’ directory in the DLL search path.

### 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 86. `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:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Element Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>.C</td>
<td><code>R_CMethodDef</code></td>
</tr>
<tr>
<td>.Call</td>
<td><code>R_CallMethodDef</code></td>
</tr>
<tr>
<td>.Fortran</td>
<td><code>R_FortranMethodDef</code></td>
</tr>
<tr>
<td>.External</td>
<td><code>R_ExternalMethodDef</code></td>
</tr>
</tbody>
</table>

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

\(^1\) and we provide an emulation on Windows 2000: see ‘`?dyn.load`’. 
SEXP myCall(SEXP a, SEXP b, SEXP c);
we would describe this as

```c
R_CallMethodDef callMethods[] = {
    {"myCall", (DL_FUNC) &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:

<table>
<thead>
<tr>
<th>Type</th>
<th>SEXP Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric</td>
<td>REALSXP</td>
</tr>
<tr>
<td>integer</td>
<td>INTSXP</td>
</tr>
<tr>
<td>logical</td>
<td>LGLSXP</td>
</tr>
<tr>
<td>single</td>
<td>SINGLESXP</td>
</tr>
<tr>
<td>character</td>
<td>STRSXP</td>
</tr>
<tr>
<td>list</td>
<td>VECSXP</td>
</tr>
</tbody>
</table>

Consider a C routine, `myC`, declared as

```c
void myC(double *x, int *n, char **names, int *status);
```
We would register it as

```c
R_CMethodDef cMethods[] = {
    {"myC", (DL_FUNC) &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:

```c
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 ensuring 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.

A CRAN example of the use of this mechanism is package lme4, which links to Matrix.

### 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 or in a site-wide ‘Makevars.site’ file. See Section “Customizing package compilation” in R Installation and Administration.

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 by R CMD INSTALL in place of the R CMD SHLIB mechanism. make is called with makefiles ‘R_HOME/etc/R_ARCH/Makeconf’, ‘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.

Under Windows 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 definitions 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: a CRAN example in package fastICA.

If you feel tempted to read the source code and subvert these mechanisms, please resist. Far too much developer time has been wasted in chasing down errors caused by failures to follow this documentation, and even more by package authors demanding explanations as to why their packages not longer work. In particular, undocumented environment or make variables are not for use by package writers and are subject to change without notice.
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.

```cpp
// 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

```cpp
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 124) 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 C++ versions (such as ‘cmath’) of the appropriate headers yourself before the R headers.
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 124, 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.z. 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 Linking to other packages

It is not in general possible to link a DLL in package packA to a DLL provided by package packB (for the security reasons mentioned in Section 5.3 [dyn.load and dyn.unload], page 86, and also because some platforms distinguish between shared objects and dynamic libraries), but it is on Windows.

Note that there can be tricky versioning issues here, as package packB could be re-installed after package packA — it is desirable that the API provided by package packB remains backwards-compatible.

5.8.1 Unix-alikes

It is possible to link a shared object in package packA to a library provided by package packB under limited circumstances on a Unix-alike OS. There are severe portability issues, so this is not recommended for a distributed package.

This is easiest if packB provides a static library ‘packB/libs/libpackB.a’. (This will need to be compiled with PIC flags on platforms where it matters.) Then as the code from package packB is incorporated when package packA is installed, we only need to find the static library at install time for package packB. The only issue is to find package packB, and for that we can ask R by something like

```
PKGB_PATH='echo 'cat(system.file("libs", .Platform$r_arch, package="packB", mustWork=TRUE))' |
"${R_HOME}/bin/R" --vanilla --slave'
PKG_LIBS="$(PKGB_PATH)/libpackB.a"
```

which will give an empty path component if sub-architectures are not in use (but works on current platforms).

For a dynamic library ‘packB/libs/libpackB.so’ (‘packB/libs/libpackB.dylib’ on Mac OS X) we could use

```
PKGB_PATH='echo 'cat(system.file("libs", .Platform$r_arch, package="packB", mustWork=TRUE))' |
"${R_HOME}/bin/R" --vanilla --slave'
PKG_LIBS=-L"$(PKGB_PATH)" -lpackB
```
This will work for installation, but very likely not when package `packB` is loaded, as the path to package `packB`’s ‘libs’ directory is not in the `ld.so` search path. You can arrange to put it there before R is launched by setting (on some platforms) `LD_RUN_PATH` or `LD_LIBRARY_PATH` or adding to the `ld.so` cache (see `man ldconfig`). On platforms that support it, the path to the dynamic library can be hardcoded at install time (which assumes that the location of package `packB` will not be changed) nor the package updated to a changed API, as Rcpp has done all too often). On systems with the GNU linker (e.g. Linux) and some others (e.g. Mac OS X) this can be done by

```bash
PKGB_PATH='echo 'library(packB); cat(system.file("libs", package="packB"))'' \ 
| "${R_HOME}/bin/R" --vanilla --slave'
PKG_LIBS=-L"$(PKGB_PATH)" -rpath "$({R_HOME}/bin/R)" -lpackB
```

and on some other systems (e.g. Solaris with its native linker) use `-R` rather than `-rpath`. It may be possible to figure out what is required semi-automatically from the result of `R CMD libtool --config` (look for `hardcode`).

Making headers provided by package `packB` available to the code to be compiled in package `packA` can be done by the `LinkingTo` mechanism (see Section 5.4 [Registering native routines], page 88).

### 5.8.2 Windows

Suppose package `packA` wants to make use of compiled code provided by `packB` in DLL `packB/libs/exB.dll`, possibly the package’s DLL `packB/libs/packB.dll`. (This can be extended to linking to more than one package in a similar way.) There are three issues to be addressed:

- Making headers provided by package `packB` available to the code to be compiled in package `packA`.
  
  This is done by the `LinkingTo` mechanism (see Section 5.4 [Registering native routines], page 88).

- preparing `packA.dll` to link to `packB/libs/exB.dll`.
  
  This needs an entry in ‘Makevars.win’ of the form

  ```
  PKG_LIBS= -L<something> -lexB
  ```

  and one possibility is that `<something>` is the path to the installed `pkgB/libs` directory. To find that we need to ask R where it is by something like

  ```
  PKGB_PATH='echo 'library(packB); cat(system.file("libs", package="packB"))'' \ 
  | rterm--vanilla--slave'
  PKG_LIBS=-L"$(PKGB_PATH)" -lexB
  ```

  Another possibility is to use an import library, shipping with package `packA` an exports file `exB.def`. Then ‘Makevars.win’ could contain

  ```
  PKG_LIBS= -L. -lexB
  ```

  ```
  all: $(SHLIB) before
  ```

  before: libexB.dll.a

  

  \(^2\) dyld on Mac OS X, and `DYLD_LIBRARY_PATH`s below.
libexB.dll.a: exB.def

and then installing package `packA` will make and use the import library for `exB.dll`.
(One way to prepare the exports file is to use `pexports.exe`.)

- loading `packA.dll` which depends on `exB.dll`.

If `exB.dll` was used by package `packB` (because it is in fact `packB.dll` or `packB.dll` depends on it) and `packB` has been loaded before `packA`, then nothing more needs to be done as `exB.dll` will already be loaded into the R executable. (This is the most common scenario).

More generally, we can use the `DLLpath` argument to `library.dynam` to ensure that `exB.dll` is found, for example by setting

```r
library.dynam("packA", pkg, lib,
    DLLpath = system.file("libs", package="packB"))
```

Note that `DLLpath` can only set one path, and so for linking to two or more packages you would need to resort to setting `PATH`.

### 5.9 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.

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

```r
.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

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

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

A call to `.External` is almost identical

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

but the C side of the interface is different, having only one argument
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 5.10 [Interface functions .Call and .External], page 107 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.

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

---

3 see Chapter 6 [The R API], page 121: note that these are not all part of the API.
4 SEXP is an acronym for Simple EXPression, common in LISP-like language syntaxes.
needed. As in interpreted R code, much use is made of coercion to force the variant object to the right type.

### 5.9.1 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 `SEXP` in `Rinternals.h`. It contains several things among which are pointers to data blocks and to other `SEXP`s. A `SEXP` is simply a pointer to a `SEXP`.

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 `SEXP`, 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`:

```c
#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`:

```c
#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)

\begin{verbatim}
PROTECT_INDEX ipx;

....

PROTECT_WITH_INDEX(s = eval(OS->R_fcall, OS->R_env), &ipx);
REPROTECT(s = coerceVector(s, REALSXP), ipx);
\end{verbatim}

### 5.9.2 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 \texttt{R_alloc}; see Section 6.1 [Memory allocation], page 121. 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.

### 5.9.3 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 SEXPTYPEs are LANGSXP, CHARSX, 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,

\[
\text{PROTECT(newSexp = coerceVector(oldSexp, REALSXP))}
\]
or

\[
\text{PROTECT(newSexp = AS_NUMERIC(oldSexp))}
\]
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 NAs 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.
5.9.4 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

```r
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

```c
#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.

```c
#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.
5.9.5 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

```c
#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.

5.9.6 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

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

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

```c
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:

```c
/* 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

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

### 5.9.7 Handling character data

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

CHARSXP 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 are immutable, the same CHARSXP can be shared by any STRSXP needing an element representing the same string. R maintains a global cache of CHARSXP 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. The variant `mkCharLen` can be used to create a CHARSXP from part of a buffer and will ensure null-termination.

Currently, it is still possible to create CHARSXP using `allocVector`; CHARSXP created in this way will not be captured by the global CHARSXP cache and this should be avoided. In the future, all CHARSXP 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`.

### 5.9.8 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).`

```c
    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);
        Rprintf("first value is %f
", 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

```c
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`. `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)
```

### 5.9.9 Some convenience functions

Some operations are done so frequently that there are convenience functions to handle them. (All these are provided via the header file ‘Rinternals.h’.)

Suppose we wanted to pass a single logical argument `ignore_quotes`: we could use

```c
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

```c
double x;
...
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 `isXXX` 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 `list5` construct a pairlist from one to five items, and `lang1` to `lang6` do the same for a language object (a function to call plus zero to five arguments). Functions `elt` and `lastElt`

---

5 You can assign a copy of the object in the environment frame `rho` using `defineVar(symbol, duplicate(value), rho)`.
find the $i^{th}$ element and the last element of a pairlist, and \texttt{nthcdr} returns a pointer to the $n^{th}$ position in the pairlist (whose \texttt{CAR} is the $n^{th}$ item).

Functions \texttt{str2type} and \texttt{type2str} map R length-one character strings to and from \texttt{SEXP}, and \texttt{type2char} maps numbers to C character strings.

### 5.9.9.1 Semi-internal convenience functions

There is quite a collection of functions that may be used in your C code if you are willing to adapt to rare “API” changes. These typically contain “work horses” of R counterparts.

Functions \texttt{any\_duplicated} and \texttt{any\_duplicated3} are fast versions of R’s \texttt{any(duplicated(\_)�)}.

Function \texttt{R\_compute\_identical} corresponds to R’s \texttt{identical} function.

### 5.9.10 Named objects and copying

When assignments are done in R such as

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

the named object is not necessarily copied, so after those two assignments \texttt{y} and \texttt{x} are bound to the same \texttt{SEXP} (the structure a \texttt{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 \texttt{.C} and \texttt{.Fortran} do copy their arguments (unless the dangerous \texttt{dup = FALSE} is used), \texttt{.Call} and \texttt{.External} do not. So \texttt{duplicate} is commonly called on arguments to \texttt{.Call} before modifying them.

However, at least some of this copying is unneeded. In the first assignment shown, \texttt{x <- 1:10}, R first creates an object with value \texttt{1:10} and then assigns it to \texttt{x} but if \texttt{x} is modified no copy is necessary as the temporary object with value \texttt{1:10} cannot be referred to again. R distinguishes between named and unnamed objects \textit{via} a field in a \texttt{SEXP} that can be accessed \textit{via} the macros \texttt{NAMED} and \texttt{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 \texttt{SEXP} for which \texttt{NAMED(foo)} is zero, and if \texttt{NAMED(foo)} is two, the value should be duplicated (\textit{via} a call to \texttt{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 \texttt{x} whose value is being modified after \texttt{y <- x}.

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

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

which is computed as
Chapter 5: System and foreign language interfaces

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.

5.10 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.

5.10.1 Calling .Call

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

\[
\text{conv <- function(a, b) .Call("convolve2", a, b)}
\]

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

```r
#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.

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

SEXP convolve2(SEXP a, SEXP b)
{
  Rlen_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.

### 5.10.2 Calling `.External`

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

```r
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.

```c
#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

```c
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

```c
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.

```c
#include <R_ext/PrtUtil.h>

SEXP showArgs(SEXP args)
{
    int i;
    Rcomplex cpl;
    const char *name;
    SEXP el;

    args = CDR(args); /* skip 'name' */
    for(i = 0; args != R_NilValue; i++, args = CDR(args)) {
        name = isNull(TAG(args)) ? "" : CHAR(PRINTNAME(TAG(args)));
        el = CAR(args);
        switch(TYPEOF(el)) {
        case REALSXP:
            Rprintf("[%d] %s %f\n", i+1, name, REAL(el)[0]);
            break;
        case LGLSXP:
        case INTSXP:
            Rprintf("[%d] %s %d\n", i+1, name, INTEGER(el)[0]);
            break;
        case CPLXSXP:
            cpl = COMPLEX(el)[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(...) invisible(.External("showArgs", ...))

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

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

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

5.10.3 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.

5.11 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

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

with C code

```c
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`.

```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
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`.

```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);
SETCAR(t, install("print")); t = CDR(t);
SETCAR(t, CAR(a)); t = CDR(t);
SETCAR(t, ScalarInteger(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.

### 5.11.1 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

```r
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())
}
```

```r
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
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.

5.11.2 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

```r
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
}
```

```r
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

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

with example usage

```r
.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.

```c
#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");
```
The code to handle the arguments is

```c
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

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

We then allocate space for the calculated derivative by

```c
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.

```c
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 */
}
```

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_ELDT(\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.

```c
    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.

```c
    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).

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

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

### 5.12 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

\(^6\) see Section 5.15 [Character encoding issues], page 120 for why this might not be what is required.

\(^7\) This is only guaranteed to show the current interface: it is liable to change.
#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.

5.13 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 \( \text{tag} \) and \( \text{prot} \) 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 \( \text{tag} \) field for some form of type identification and the \( \text{prot} \) field for protecting the memory that the external pointer represents, if that memory is allocated from the R heap. Both \( \text{tag} \) and \( \text{prot} \) can be \text{R NilValue}, and often are.

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

\[
\begin{align*}
\text{void *} & \text{R_ExternalPtrAddr(SEXP s);} \\
\text{SEXP} & \text{R_ExternalPtrTag(SEXP s);} \\
\text{SEXP} & \text{R_ExternalPtrProtected(SEXP s);} \\
\text{void} & \text{R_ClearExternalPtr(SEXP s);} \\
\text{void} & \text{R_SetExternalPtrAddr(SEXP s, void *p);} \\
\text{void} & \text{R_SetExternalPtrTag(SEXP s, SEXP tag);} \\
\text{void} & \text{R_SetExternalPtrProtected(SEXP s, SEXP p);} \\
\end{align*}
\]

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, respectively.

\[
\begin{align*}
\text{void} & \text{R_RegisterFinalizerEx(SEXP s, SEXP fun, Rboolean onexit);} \\
\text{typedef \text{void} (*\text{R CFinalizer\_t})(SEXP);} \\
\text{void} & \text{R_RegisterCFinalizerEx(SEXP s, \text{R CFinalizer\_t} fun, Rboolean onexit);} \\
\end{align*}
\]

The R function indicated by \( \text{fun} \) 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 \( \text{onexit} \) 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 \text{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 \text{R NilValue} and the finalizer will be run later in the garbage collection.

Weak reference objects are created by one of

\[
\begin{align*}
\text{SEXP} & \text{R_MakeWeakRef(SEXP key, SEXP val, SEXP fin, Rboolean onexit);} \\
\text{SEXP} & \text{R_MakeWeakRefC(SEXP key, SEXP val, R\_CFinalizer\_t fin,} \\
& \text{Rboolean onexit);} \\
\end{align*}
\]
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

```c
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. At the time of writing no CRAN or Bioconductor package uses weak references.

### 5.13.1 An example

Package `RODBC` uses external pointers to maintain its channels, connections to databases. There can be several connections open at once, and the status information for each is stored in a C structure (pointed to by `this_handle` in the code extract below) that is returned via an external pointer as part of the RODBC ‘channel’ (as the "handle_ptr" attribute). The external pointer is created by

```c
SEXP ans, ptr;
PROTECT(ans = allocVector(INTSXP, 1));
ptr = R_MakeExternalPtr(thisHandle, install("RODBC_channel"), R_NilValue);
PROTECT(ptr);
R_RegisterCFinalizerEx(ptr, chanFinalizer, TRUE);
...
/* return the channel no */
INTEGER(ans)[0] = nChannels;
/* and the connection string as an attribute */
setAttrib(ans, install("connection.string"), constr);
setAttrib(ans, install("handle_ptr"), ptr);
UNPROTECT(3);
return ans;
```

Note the symbol given to identify the usage of the external pointer, and the use of the finalizer. Since the final argument when registering the finalizer is `TRUE`, the finalizer will be run at the the of the R session (unless it crashes). This is used to close and clean up the connection to the database. The finalizer code is simply

```c
static void chanFinalizer(SEXP ptr)
{
    if(!R_ExternalPtrAddr(ptr)) return;
inRODBCClose(R_ExternalPtrAddr(ptr));
    R_ClearExternalPtr(ptr); /* not really needed */
}
```

Clearing the pointer and checking for a NULL pointer avoids any possibility of attempting to close an already-closed channel.

R’s connections provide another example of using external pointers, in that case purely to be able to use a finalizer to close and destroy the connection if it is no longer in use.
5.14 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.

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.

5.15 Character encoding issues

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 CHARSXPS 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 current locale: this can be done by accessing the data in the CHARSX 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.

There is a similar function translateCharUTF8 which converts to UTF-8: this has the advantage that a faithful translation is almost always possible (whereas only a few languages can be represented in the encoding of the current locale unless that is UTF-8).

There is a public interface to the encoding marked on CHARSXPs via

typedef enum {CE_NATIVE, CE_UTF8, CE_LATIN1, CE_SYMBOL, CE_ANY} cetype_t;

cetype_t getCharCE(SEXP);

SEXP mkCharCE(const char *, cetype_t);

Only the CE_UTF8 and CE_LATIN1 are marked on CHARSXPs (and so Rf_getCharCE will only return one of the first three), and these should only be used on non-ASCII strings. Value CE_SYMBOL is used internally to indicate Adobe Symbol encoding. Value CE_ANY is used to indicate a character string that will not need re-encoding – this is used for character strings known to be in ASCII, and can also be used as an input parameter where the intention is that the string is treated as a series of bytes.

Function

const char *reEnc(const char *, cetype_t ce_in, cetype_t ce_out,
              int subst);

can be used to re-encode character strings: like translateChar it returns a string allocated by R_alloc. This can translate from CE_SYMBOL to CE_UTF8, but not conversely. Argument subst controls what to do with untranslatable characters or invalid input: this is done byte-by-byte with 1 indicates to output hex of the form <a0>, and 2 to replace by ., with any other value causing the byte to produce no output.

There is also

SEXP mkCharLenCE(const char *, int, cetype_t);

to create marked character strings of a given length.
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

```c
#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 some extra 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-alike 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).

\(^1\) Known problems are redefining error, length, vector and warning
6.1.1 Transient storage allocation

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

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

which allocates \( n \) units of \( \text{size} \) bytes each. A typical usage (from package stats) is

```c
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,

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

and

```c
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

```c
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 134).

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

```c
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.

```c
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

```c
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

```c
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

```c
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

```c
GetRNGstate();
```

and after all the required variates have been generated, call

```c
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 126. 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.

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

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. It is defined in ‘R_ext/Print.h’.

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 `REVPprintf` are analogues using the `vprintf` interface.

Because `RVprintf` and `REVPprintf` use a C99 interface, they are only defined by ‘R_ext/Print.h’ in C++ code if the macro `R_USE_C99_IN_CXX` is defined when it is included.

### 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.
Chapter 6: The R API: entry points for C 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:

```c
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

```c
-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 123, 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`) mainly the same as the corresponding R functions of the same name, so the documentation of the R functions can be used. Note that the exponential and gamma distributions are parametrized by `scale` rather than `rate`. 
Chapter 6: The R API: entry points for C code

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.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beta</td>
<td>beta</td>
<td>a, b</td>
</tr>
<tr>
<td>Non-central Beta</td>
<td>nbeta</td>
<td>a, b, ncp</td>
</tr>
<tr>
<td>Binomial</td>
<td>binom</td>
<td>n, p</td>
</tr>
<tr>
<td>Cauchy</td>
<td>cauchy</td>
<td>location, scale</td>
</tr>
<tr>
<td>Chi-squared</td>
<td>chisq</td>
<td>df</td>
</tr>
<tr>
<td>Non-central Chi-squared</td>
<td>nchisq</td>
<td>df, ncp</td>
</tr>
<tr>
<td>Exponential</td>
<td>exp</td>
<td>scale (and not rate)</td>
</tr>
<tr>
<td>F</td>
<td>f</td>
<td>n1, n2</td>
</tr>
<tr>
<td>Non-central F</td>
<td>nf</td>
<td>n1, n2, ncp</td>
</tr>
<tr>
<td>Gamma</td>
<td>gamma</td>
<td>shape, scale</td>
</tr>
<tr>
<td>Geometric</td>
<td>geom</td>
<td>p</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>hyper</td>
<td>NR, NB, n</td>
</tr>
<tr>
<td>Logistic</td>
<td>logis</td>
<td>location, scale</td>
</tr>
<tr>
<td>Lognormal</td>
<td>lnorm</td>
<td>logmean, logsd</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>nbinom</td>
<td>size, prob</td>
</tr>
<tr>
<td>Normal</td>
<td>norm</td>
<td>mu, sigma</td>
</tr>
<tr>
<td>Poisson</td>
<td>pois</td>
<td>lambda</td>
</tr>
<tr>
<td>Student’s t</td>
<td>t</td>
<td>n</td>
</tr>
<tr>
<td>Non-central t</td>
<td>nt</td>
<td>df, delta</td>
</tr>
<tr>
<td>Studentized range</td>
<td>tukey</td>
<td>rr, cc, df</td>
</tr>
<tr>
<td>Uniform</td>
<td>unif</td>
<td>a, b</td>
</tr>
<tr>
<td>Weibull</td>
<td>weibull</td>
<td>shape, scale</td>
</tr>
<tr>
<td>Wilcoxon rank sum</td>
<td>wilcox</td>
<td>m, n</td>
</tr>
<tr>
<td>Wilcoxon signed rank</td>
<td>signrank</td>
<td>n</td>
</tr>
</tbody>
</table>

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.

### 6.7.2 Mathematical functions

```c
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, the natural logarithm of its absolute value and first four derivatives and the n-th derivative of Psi, the digamma function, which is the derivative of lgammafn. In other words, `digamma(x)` is the same as `(psigamma(x,0), trigamma(x) == psigamma(x,1), etc.`
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 the natural logarithm of its absolute value, generalized to arbitrary real n. k is rounded to the nearest integer (with a warning if needed).

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 log1p (double x) [Function]

Computes log(1 + x) (log 1 plus x), accurately even for small x, i.e., |x| ≪ 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 log1pmx (double x) [Function]

Computes log(1 + x) - x (log 1 plus x minus x), accurately even for small x, i.e., |x| ≪ 1.

double expm1 (double x) [Function]

Computes exp(x) - 1 (exp x minus 1), accurately even for small x, i.e., |x| ≪ 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 plus 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.
### 6.7.4 Mathematical constants

R has a set of commonly used mathematical constants encompassing constants usually found in `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).

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition ($\ln = \log$)</th>
<th>round($value$, 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_E</td>
<td>$e$</td>
<td>2.7182818</td>
</tr>
<tr>
<td>M_LOG2E</td>
<td>$\log_2(e)$</td>
<td>1.4426950</td>
</tr>
<tr>
<td>M_LOG10E</td>
<td>$\log_{10}(e)$</td>
<td>0.4342945</td>
</tr>
<tr>
<td>M_LN2</td>
<td>$\ln(2)$</td>
<td>0.6931472</td>
</tr>
<tr>
<td>M_LN10</td>
<td>$\ln(10)$</td>
<td>2.3025851</td>
</tr>
<tr>
<td>M_PI</td>
<td>$\pi$</td>
<td>3.1415927</td>
</tr>
<tr>
<td>M_PI_2</td>
<td>$\pi/2$</td>
<td>1.5707963</td>
</tr>
<tr>
<td>M_PI_4</td>
<td>$\pi/4$</td>
<td>0.7853982</td>
</tr>
<tr>
<td>M_1_PI</td>
<td>$1/\pi$</td>
<td>0.3183099</td>
</tr>
<tr>
<td>M_2_PI</td>
<td>$2/\pi$</td>
<td>0.6366198</td>
</tr>
<tr>
<td>M_2_SQRTPI</td>
<td>$2/\sqrt{\pi}$</td>
<td>1.1283792</td>
</tr>
<tr>
<td>M_SQRT2</td>
<td>$\sqrt{2}$</td>
<td>1.4142136</td>
</tr>
<tr>
<td>M_SQRT1_2</td>
<td>$1/\sqrt{2}$</td>
<td>0.7071068</td>
</tr>
<tr>
<td>M_SQRT_3</td>
<td>$\sqrt{3}$</td>
<td>1.7320508</td>
</tr>
<tr>
<td>M_SQRT_32</td>
<td>$\sqrt{32}$</td>
<td>5.6568542</td>
</tr>
<tr>
<td>M_LOG10_2</td>
<td>$\log_{10}(2)$</td>
<td>0.3010300</td>
</tr>
</tbody>
</table>
Chapter 6: The R API: entry points for C code

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_2PI</td>
<td>$2\pi$</td>
</tr>
<tr>
<td>M_SQRT_PI</td>
<td>$\sqrt{\pi}$</td>
</tr>
<tr>
<td>M_1_SQRT_2PI</td>
<td>$1/\sqrt{2\pi}$</td>
</tr>
<tr>
<td>M_SQRT_2dPI</td>
<td>$\sqrt{2/\pi}$</td>
</tr>
<tr>
<td>M_LN_SQRT_PI</td>
<td>$\ln(\sqrt{\pi})$</td>
</tr>
<tr>
<td>M_LN_SQRT_2PI</td>
<td>$\ln(\sqrt{2\pi})$</td>
</tr>
<tr>
<td>M_LN_SQRT_PId2</td>
<td>$\ln(\sqrt{\pi/2})$</td>
</tr>
</tbody>
</table>

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

```c
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

```c
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:

  ```c
  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:

  ```c
  void vmmn(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:

  ```c
  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:

  ```c
  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 *fnccount, 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 \texttt{optim}: see the source code ‘\texttt{src/appl/lbfgsb.c}’ for the values of \( nbd \), which specifies which bounds are to be used.

### 6.9 Integration

The C code underlying \texttt{integrate} can be accessed directly. The user needs to supply a vectorizing C function to compute the function to be integrated, of the type

\[
\texttt{typedef void integr_fn(double *x, int n, void *ex);}
\]

where \( x[] \) is both input and output and has length \( n \), i.e., a C function, say \( fn \), of type \texttt{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 ‘\texttt{R_ext/Applic.h}’) for definite and for indefinite integrals. ‘Indefinite’ means that at least one of the integration boundaries is not finite.

- Finite:

  \[
  \texttt{void Rdqags(integr_fn f, void *ex, double *a, double *b,}
  \texttt{double *epsabs, double *epsrel,}
  \texttt{double *result, double *abserr, int *neval, int *ier,}
  \texttt{int *limit, int *lenw, int *last,}
  \texttt{int *iwork, double *work);}
  \]

- Indefinite:

  \[
  \texttt{void Rdqagi(integr_fn f, void *ex, double *bound, int *inf,}
  \texttt{double *epsabs, double *epsrel,}
  \texttt{double *result, double *abserr, int *neval, int *ier,}
  \texttt{int *limit, int *lenw, int *last,}
  \texttt{int *iwork, double *work);}
  \]

Only the 3rd and 4th argument differ for the two integrators; for the definite integral, using \texttt{Rdqags}, \( a \) and \( b \) are the integration interval bounds, whereas for an indefinite integral, using \texttt{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,

\[
\text{inf} = 1 \quad \text{corresponds to (bound, +Inf)},
\]

\[
\text{inf} = -1 \quad \text{corresponds to (-Inf, bound)},
\]
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

\[
\text{lenw} = 4 \times \text{limit};
\]
\[
\text{iwork} = (\text{int} *) \text{R_alloc} \left( \text{limit}, \text{sizeof(int)} \right);
\]
\[
\text{work} = (\text{double} *) \text{R_alloc} \left( \text{lenw}, \text{sizeof(double)} \right);
\]

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.

```c
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 `NA`s are not handled (explicitly) and you should use different sorting functions if `NA`s can be present.

```fortran
subroutine qsort4 (double precision v, integer indx, integer ii, integer jj)  
```

The FORTRAN interface routines for sorting double precision vectors are `qsort3` and `qsort4`, equivalent to `R_qsort` and `R_qsort_I`, respectively.

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

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`.

```fortran
int findInterval (double* xt, int n, double x, Rboolean rightmost_closed, Rboolean all_inside, int ilo, int* mflag)  
```

Given the ordered vector `xt` of length `n`, return the interval or index of `x` in `xt[]`, typically `max(i; 1 ≤ i ≤ n & xt[i] ≤ 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].

```fortran
void hsv2rgb (double h, double s, double v, double *r, double *g, double *b)  
void rgb2hsv (double r, double g, double b, double *h, double *s, double *v)  
```

A system-independent interface to produce the name of a temporary file is provided as

```fortran
char * R_tmpnam (const char *prefix)  
```

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`. 

The following two functions do numerical colorspace conversion from HSV to RGB and back. Note that all colours must be in [0,1].
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 because 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 can 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. most OSes on Sparc, PowerPC and PA-RISC hardware, and HP-UX on Itanium) and not on little-endian systems (such as i686 and x86_64 on all OSes, and Linux on Alpha and Itanium). It can be useful when manipulating binary files. The macro SUPPORT_OPENMP is defined on suitable systems as from R 2.13.0, and can be used in conjunction with the SUPPORT_OPENMP_* macros in packages that want to make use of OpenMP.

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 very old versions of R which did not have this macro, use a construction such as

```c
#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 should be recognized by all compilers now used to build R. Portable code which might be used with earlier versions of R can be written using the macro R_INLINE (defined in file ‘Rconfig.h’ included by ‘R.h’), as for example from package cluster

```c
#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, so this usage is for `static` functions as in the example. 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.

---

Chapter 6: The R API: entry points for C code

6.15 Controlling visibility

Header ‘R_ext/Visibility’ has some definitions for controlling the visibility of entry points. These are only effective when ‘HAVE_VISIBILITY_ATTRIBUTE’ is defined – this is checked when R is configured and recorded in header ‘Rconfig.h’ (included by ‘R_ext/Visibility.h’). It is generally defined on modern Unix-alikes with a recent compiler (e.g. gcc4), but not supported on Windows. Minimizing the visibility of symbols in a shared library will both speed up its loading (unlikely to be significant) and reduce the possibility of linking to the wrong entry points of the same name.

C/C++ entry points prefixed by attribute_hidden will not be visible in the shared object. There is no comparable mechanism for FORTRAN entry points, but there is a more comprehensive scheme used by, for example package stats. Most compilers which allow control of visibility will allow control of visibility for all symbols via a flag, and where known the flag is encapsulated in the macros ‘C_VISIBILITY’ and F77_VISIBILITY for C and FORTRAN compilers. These are defined in ‘etc/Makeconf’ and so available for normal compilation of package code. For example, ‘src/Makevars’ could include

```
PKG_CFLAGS=$(C_VISIBILITY)
PKG_FFLAGS=$(F77_VISIBILITY)
```

This would end up with no visible entry points, which would be pointless. However, the effect of the flags can be overridden by using the attribute_visible prefix. A shared object which registers its entry points needs only for have one visible entry point, its initializer, so for example package stats has

```
void attribute_visible R_init_stats(DllInfo *dll)
{
    R_registerRoutines(dll, CEntries, CallEntries, FortEntries, NULL);
    R_useDynamicSymbols(dll, FALSE);
    ...
}
```

The visibility mechanism is not available on Windows, but there is an equally effective way to control which entry points are visible, by supplying a definitions file ‘pkgname/src/pkgname-win.def’: only entry points listed in that file will be visible. Again using stats as an example, it has

```
LIBRARY stats.dll
EXPORTS
R_init_stats
```

6.16 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-alikes and Windows. (This includes the functions documented in Section 6.7 [Numerical analysis subroutines], page 126 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

\[
\text{double unif_rand(void)}
\]

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

\[
\text{set_seed(unsigned int, unsigned int)}
\]
to set its seeds and

\[
\text{get_seed(unsigned int *, unsigned int *)}
\]
to read the seeds).

### 6.17 Organization of header files

The header files which R installs are in directory ‘\texttt{R_INCLUDE_DIR}’ (default ‘\texttt{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/Callblocks.h’ C (and R function) top-level task handlers
- ‘R_ext/GetX11Image.h’ X11Image interface used by package \texttt{tkplot}
- ‘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/RConversions.h’ needed to register compiled code in packages
- ‘R_ext/Rdynload.h’ interface to internal method of \texttt{download.file}
- ‘R_ext/Rfalcon.h’ interface to \texttt{iconv}
- ‘R_ext/RStartup.h’ for add-on front-ends
- ‘R_ext/Visibility.h’ definitions controlling visibility
- ‘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’ \texttt{TRUE/FALSE} type
- ‘R_ext/Complex.h’ C typedefs for R’s complex
- ‘R_ext/Constants.h’ constants
The graphics systems are exposed in headers `R_ext/GraphicsEngine.h`, `R_ext/GraphicDevice.h` (which it includes) and `R_ext/QuartzDevice.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

```r
nextMethod <- 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

```r
scale <- function(x, center = TRUE, scale = TRUE)
  UseMethod("scale")
```

Suppose an unthinking package writer created methods such as

```r
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. Also, it can be built as a static library if configured with ‘--enable-R-static-lib’, and this can be used in a very similar way.

The command-line R front-end, ‘R_HOME/bin/exec/R’ is one such example, and the former GNOME (see package gnomeGUI on CRAN’s ‘Archive’ area) and Mac OS X consoles are others. The source for ‘R_HOME/bin/exec/R’ is in file ‘src/main/Rmain.c’ and is very simple

```c
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/etcR_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

\(^1\) In the parlance of Mac OS X this is a dynamic library, and is the normal way to build R on that platform.
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
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_Rep1DLLInit();
       while(R_Rep1DLLdo1() > 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).

\(^2\) but these are not part of the automated test procedures and so little tested.
Chapter 8: Linking GUIs and other front-ends to R

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.

Note that R expects to be run with the locale category ‘LC_NUMERIC’ set to its default value of C, and so should not be embedded into an application which changes that.

### 8.1.1 Compiling against the R library

Suitable flags to compile and link against the R (shared or static) library can be found by

```bash
R CMD config --cppflags
R CMD config --ldflags
```

If R is installed, `pkg-config` is available and sub-architectures have not been used, alternatives for a shared R library are

```bash
pkg-config --cflags libR
pkg-config --libs libR
```

and for a static R library

```bash
pkg-config --cflags libR
pkg-config --libs --static libR
```

### 8.1.2 Setting R callbacks

For Unix-alikes 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)

```c
extern void (*ptr_R_Suicide)(const char *);
extern void (*ptr_R_ShowMessage)(const char *);
extern int (*ptr_R_ReadConsole)(const char *, unsigned char *, int, int);
extern void (*ptr_R_WriteConsole)(const char *, int);
extern void (*ptr_R_WriteConsoleEx)(const 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, const char **, const char *, Rboolean, const char *);
extern int (*ptr_R_ChooseFile)(int, char *, int);
extern int (*ptr_R_EditFile)(const 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)  
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)  
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 (const char *prompt, unsigned char *buf, int buflen, int hist)  
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 ‘“
\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.

void R_WriteConsole (const char *buf, int buflen)  
void R_WriteConsoleEx (const char *buf, int buflen, int otype)  
void R_ResetConsole ()  
void R_FlushConsole ()  
void R_ClearErrConsole ()  

These functions interact with a console.

int R_ShowFiles (int nfile, const char **file, const char **headers, const char *wtitle, Rboolean del, const char *pager)  
This function is used to display the contents of files.

int R_ChooseFile (int new, char *buf, int len)  
Choose a file and return its name in buf of length len. Return value is 0 for success, > 0 otherwise.

int R_EditFile (const char *buf)  
Send a file to an editor window.

SEXP R_loadhistory (SEXP, SEXP, SEXP, SEXP);  
SEXP R_savehistory (SEXP, SEXP, SEXP, SEXP);  
SEXP R_addhistory (SEXP, SEXP, SEXP, SEXP);  

.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

```c
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.

```c
void R_Suicide (const char *message)  
```

This should abort R as rapidly as possible, displaying the message. A possible implementation is

```c
void R_Suicide (const char *message)  
{  
  char pp[1024];  
  snprintf(pp, 1024, "Fatal error: %s\n", s);  
  R_ShowMessage(pp);  
  R_CleanUp(SA_SUICIDE, 2, 0);  
}
```

```c
void R_CleanUp (SA_TYPE saveact, int status, int RunLast)  
```

This function invokes any actions which occur at system termination. It needs to be quite complex:

```c
#include <Rinterface.h>  
#include <Rembedded.h>  
/* for Rf_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) Rf_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

```c
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 88 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 (for at least the X11 version of Tk).
- Preparing input.
- Timing operations, for example for profiling R code and Sys.sleep().
- Interrupts, where permitted.

Specifically, the Unix-alike 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/...`

- 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

```c
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

```c
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 `Rf_initialize_R` is called, to values appropriate to the main thread. Stack-checking can be disabled by setting `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.)

---

\(^3\) at least on platforms where the values are available, that is having `getrlimit` and on Linux or having `sysctl` supporting `KERN_USERSTACK`, including FreeBSD and Mac OS X.
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 via http://cran.r-project.org/other-software.html or http://sunsite.univie.ac.at/rcom/, which works with package rscproxy 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-alike interface calling

```c
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 a simple command-line front end ‘rtest.c’ whose code is

```c
#define Win32
#include <windows.h>
#include <stdio.h>
#include <Rversion.h>
#define LibExtern __declspec(dllimport) extern
#include <Rembedded.h>
#include <R_ext/RStartup.h>
/* for askok and askyesnocancel */
```
#include <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(const 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(const char *buf, int len)
{
    printf("%s", buf);
}

void myCallBack(void)
{
    /* 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;
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 be in HKEY_LOCAL_MACHINE\Software\R-core\R\InstallPath from an administrative install and HKEY_CURRENT_USER\Software\R-core\R\InstallPath otherwise, if selected during installation (as it is by default).
- 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
  
  ```c
  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 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.

### 8.2.3 Finding R_HOME

Both applications which embed R and those which use a **system** call to invoke R (as **Rscript.exe**, **Rterm.exe** or **R.exe**) need to be able to find the R ‘bin’ directory. The simplest way to do so is the ask the user to set an environment variable **R_HOME** and use that, but naive users may be flummoxed as to how to do so or what value to use.

The R for Windows installers have for a long time allowed the value of **R_HOME** to be recorded in the Windows Registry: this is optional but selected by default. Where is it is recorded has changed over the years to allow for multiple versions of R to be installed at once, and to allow 32- and 64-bit versions of R to be installed on the same machine.

The basic Registry location is **Software\R-core\R**. For an administrative install this is under **HKEY_LOCAL_MACHINE** and on a 64-bit OS **HKEY_LOCAL_MACHINE\Software\R-core\R** is by default redirected for a 32-bit application, so a 32-bit application will see the information for the last 32-bit install, and a 64-bit application that for the last 64-bit install. For a personal install, the information is under **HKEY_CURRENT_USER\Software\R-core\R** which is seen by both 32-bit and 64-bit applications and so records the last install of either architecture. To circumvent this, there are locations **Software\R-core\R32** and **Software\R-core\R64** which always refer to one architecture.

When R is installed and recording is not disabled then two string values are written at that location for keys **InstallPath** and **Current Version**, and these keys are removed when R is uninstalled. To allow information about other installed versions to be retained, there is

---

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.
also a key named something like 2.11.0 or 2.11.0 patched or 2.12.0 Pre-release with a value for InstallPath.

So a comprehensive algorithm to search to R_HOME is something like

- Decide which of personal or administrative installs should have precedence. There are arguments both ways: we find that with roaming profiles that HKEY_CURRENT_USER\Software often gets reverted to an earlier version. Do the following for one or both of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE.

- If the desired architecture is known, look in Software\R-core\R32 or Software\R-core\R64, and if that does not exist or the architecture is immaterial, in Software\R-core\R.

- If key InstallPath exists then this is R_HOME (recorded using backslashes). If it does not, look for version-specific keys like 2.11.0 alpha, pick the latest (which is of itself a complicated algorithm as 2.11.0 patched > 2.11.0 > 2.11.0 alpha > 2.8.1) and use its value for InstallPath.

Prior to R 2.12.0 ‘R.dll’ and the various front-end executables are in ‘R_HOME\bin’, but they are now in ‘R_HOME\bin\i386’ or ‘R_HOME\bin\x64’. So you need to arrange to look first in the architecture-specific subdirectory and then in ‘R_HOME\bin’.
Function and variable index

* ................................. 134
*Riconv_open .......................... 134
.
.C .................................. 84
.Call .................................. 96, 107
.External ............................. 96, 108
.First.lib ............................ 9
.Fortran ............................. 84
.Last.lib ............................ 9, 33
.onAttach ............................ 33
.onLoad .............................. 33
.onUnload ............................ 33
.Random.seed .......................... 123

\acronym ............................ 57
\alias ............................... 49
\arguments ........................... 51
\author .............................. 52
\bold ................................ 55
\cite ................................ 57
\code ................................ 56
\command ............................ 57
\concept ............................. 60
\cr .................................. 55
\deqn ................................ 59
\describe ............................ 58
\description ........................ 49
\details ............................. 51
\dfn ................................ 57
\dontrun ............................. 52
\dontshow ............................ 52
\dots ................................ 60
\dQuote ............................... 56
\email ............................... 56
\emph ................................ 55
\enc ................................ 60
\enumerate ........................... 57
\env .................................. 57
\eqn ................................ 59
\examples ............................ 52
\file ................................ 56
\format ............................... 53
\href ................................ 57
\if ................................... 61
\ifelse ................................ 61
\itemize ................................ 57
\kbd ................................ 56
\keyword ................................ 52
\ldots .................................. 60
\link .................................. 58
\method ................................ 50
\name .................................. 48
\newcommand ........................... 62
\note .................................. 51
\option ................................ 57
\out .................................. 61
\pkg .................................. 56
\preformatted ........................ 56
\R ..................................... 60
\RdOpt ................................ 61
\references ........................... 51
\renewcommand ........................ 62
\SSmethod ............................ 51
\samp .................................. 56
\section ................................ 55
\seealso ................................ 52
\Sexpr ................................ 61
\source ................................ 54
\SQuote ................................ 56
\strong ............................... 55
\tabular .............................. 58
\title .................................. 49
\url .................................. 57
\usage .................................. 49
\value .................................. 51
\var ................................... 57
\verb .................................. 56

B
bessel.i ................................ 128
bessel.j ................................ 128
bessel.K ................................ 128
bessel.y ................................ 128
beta .................................... 128
BLAS_LIBS ................................ 16
browser ................................ 72

C
Call .................................... 122
CAR ..................................... 109
CDR ..................................... 109
cgmin .................................. 130
choose .................................. 128
CITATION ................................ 10, 44
cPsart .................................. 132

D
debug .................................. 76
debugger ................................ 75
defineVar ............................... 105
digamma .................................. 127
dump.frames ............................. 74
duplicate.............................................. 106
dyn.load........................................... 86
dyn.unload......................................... 86

E
exp_rand............................................ 123
expm1............................................. 128
export............................................. 31
exportClasses..................................... 36
exportClassPattern............................... 36
exportMethods.................................... 36
exportPattern.................................... 31, 36

F
FALSE............................................... 130
findInterval....................................... 133
findVar.......................................... 104
FLIBS............................................. 16
fmax2............................................. 129
fmin2............................................. 129
fprec............................................. 129
Free.............................................. 122
fround........................................... 129
fsign.............................................. 129
ftrunc........................................... 129

G
gammafn........................................... 127
gctorture......................................... 77
getAttrib......................................... 102
getCharCE......................................... 120
GetRNGstate....................................... 123

H
 hsv2rgb........................................... 133

I
imax2............................................. 129
imin2............................................. 129
import............................................ 32
importClassesFrom............................... 37
importFrom....................................... 32
importMethodsFrom............................. 37
install.......................................... 102
IPsort........................................... 132
ISNA............................................... 110, 124
ISNAN............................................. 110, 124

L
LAPACK_LIBS....................................... 16
lbeta............................................. 128
1bfgsb........................................... 130
lchoose.......................................... 128
lgamma1p......................................... 128
lgammafn......................................... 127
library.dynam.................................... 9, 87
log1p............................................. 128
loglpmx......................................... 128
logspace_add..................................... 128
logspace_sub..................................... 128

M
M_E................................................ 129
M_PI............................................... 129
mkChar........................................... 104
mkCharCE.......................................... 120
mkCharLen........................................ 104
mkCharLenCE...................................... 120

N
NA_REAL.......................................... 124
nmmin.......................................... 130
norm_rand........................................ 123

O
OBJECTS........................................... 16, 91

P
pentagamma........................................ 127
PKG_CFLAGS....................................... 90
PKG_CPPFLAGS.................................... 90
PKG_CXXFLAGS.................................... 90
PKG_FCFLAGS..................................... 90
PKG_FFLAGS....................................... 90
PKG_LIBS.......................................... 91
PKG_OBJCFLAGS................................... 90
prompt........................................... 48
PROTECT.......................................... 98
PROTECT_WITH_INDEX............................. 99
psigamma.......................................... 127
PutRNGstate...................................... 123

Q
qsort3............................................ 133
qsort4............................................ 133

R
R CMD build...................................... 24
R CMD check...................................... 22
R CMD config.................................... 14
R CMD Rd2dvi..................................... 64
R CMD Rd2pdf..................................... 64
R CMD Rdconv.................................... 64
Function and variable index

R CMD SHLIB ........................................... 90
R CMD Stangle ........................................ 64
R CMD Sweave ......................................... 64
R_addhistory .......................................... 144
R_alloc ............................................... 122
R_BUSY ................................................. 144
R.ChooseFile .......................................... 144
R_CleanUp ............................................. 145
R_ClearErrConsole ..................................... 144
R_csort ............................................... 132
R_EditFile ............................................. 144
R_ExpandFileName ..................................... 134
R_FINITE ............................................... 124
R.FlushConsole ........................................ 144
R_GetCCallable ....................................... 90
RINLINE ............................................... 135
R_IsNaN ............................................... 124
R_isort ............................................... 132
R_LIBRARY_DIR ........................................ 15
R_loadhistory ........................................ 144
R_max_col ............................................ 133
R_NegInf .............................................. 124
R_PACKAGE_DIR ........................................ 15
R_PACKAGE_NAME ....................................... 15
R_ParseVector ........................................ 117
R_PosiNf .............................................. 124
R_pow ................................................. 128
R_pow_di .............................................. 128
R_qsort ................................................ 132
R_qsort.Int ........................................... 132
R_qsort_int .......................................... 132
R_ReadConsole ......................................... 144
R_RegisterCCallable .................................. 90
R_registerRoutines ................................... 88
R_ResetConsole ........................................ 144
R_rsort ................................................. 132
R_savehistory ......................................... 144
R_ShowFiles .......................................... 144
R_ShowMessage ........................................ 144
R_Suicide ............................................. 145
R_tmpnam .............................................. 133
R_Version ............................................. 135
R_WriteConsole ........................................ 144
R_WriteConsoleEx ..................................... 144
rcont2 ............................................... 124
Rdagi .................................................. 131
Rdags .................................................. 131
Realloc ............................................... 122
recover ............................................... 76
reEnc .................................................. 120
REprintf .............................................. 124
REPROTECT ............................................ 99
REvprintf ............................................. 124
revsort ............................................... 132
rgb2hsv ............................................... 133
Riconv ............................................... 134
Riconv_close ......................................... 134
Rprintf ............................................... 124
Rprof .................................................. 65, 68
Rprofmem ............................................. 68
rPsort ............................................... 132
rsort_with_index ..................................... 132
Rvprintf .............................................. 124
S
S_alloc ............................................... 122
S_realloc ............................................ 122
S3method ............................................. 32
SAFE_FFLAGS ......................................... 16
samin ............................................... 131
seed_in ............................................... 123
seed_out ............................................. 123
setAttrib ............................................ 102
setVar ............................................... 105
sign ................................................... 129
summaryRprof ........................................ 68
system ............................................... 84
system.time .......................................... 84
T
tetragamma .......................................... 127
trace .................................................. 77
traceback ............................................. 73
tracemem ............................................. 68
translateChar ......................................... 120
translateCharUTF8 ................................... 120
trigamma ............................................. 127
TRUE .................................................. 130
U
undebug ............................................... 77
unif_rand ............................................. 123
UNPROTECT .......................................... 98
UNPROTECT_PTR ....................................... 99
untracemem .......................................... 68
useDynLib ............................................ 33
V
vmaxget ............................................... 122
vmaxset ............................................... 122
vmmin ............................................... 130
## Concept index

<table>
<thead>
<tr>
<th>\</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkS4class</td>
<td>58</td>
</tr>
</tbody>
</table>

### A

| Allocating storage | 99 |
| Attributes | 101 |

### B

| Bessel functions | 128 |
| Beta function | 128 |
| Building packages | 24 |

### C

| C++ code, interfacing | 92 |
| Calling C from FORTRAN and vice versa | 125 |
| Checking packages | 22 |
| citation | 10, 44 |
| Classes | 103 |
| cleanup file | 2 |
| conditionals | 61 |
| configure file | 2 |
| COPYING file | 2 |
| Copying objects | 106 |
| CRAN | 29 |
| CRAN submission | 29 |
| Creating packages | 2 |
| Creating shared objects | 90 |
| Cross-references in documentation | 58 |
| cumulative hazard | 126 |

### D

| Debugging | 79 |
| DESCRIPTION file | 3 |
| Details of R types | 99 |
| Distribution functions from C | 126 |
| Documentation, writing | 47 |
| Dynamic loading | 86 |
| dynamic pages | 61 |

### E

| encoding | 63 |
| Error handling from C | 123 |
| Error handling from FORTRAN | 123 |
| Evaluating R expressions from C | 110 |
| external pointer | 117 |

### F

| finalizer | 118 |
| Finding variables | 104 |

### G

| Gamma function | 127 |
| Garbage collection | 98 |
| Generic functions | 139 |

### H

| handling character data | 104 |
| Handling lists | 103 |
| Handling R objects in C | 96 |

### I

| IEEE special values | 110, 124 |
| INDEX file | 8 |
| Indices | 60 |
| Inspecting R objects when debugging | 81 |
| integration | 131 |
| Interfaces to compiled code | 84, 107 |
| Interfacing C++ code | 92 |
| Interrupts | 134 |

### L

| LICENCE file | 2 |
| LICENSE file | 2 |
| Lists and tables in documentation | 57 |

### M

| Marking text in documentation | 55 |
| Mathematics in documentation | 59 |
| Memory allocation from C | 121 |
| Memory use | 67 |
| Method functions | 139 |
| Missing values | 110, 124 |

### N

| name spaces | 31 |
| Numerical analysis subroutines from C | 126 |
| Numerical derivatives | 113 |

### O

| OpenMP | 18, 135 |
| Operating system access | 84 |
| optimization | 130 |
P
Package builder ........................................... 24
Package bundles .......................................... 11
Package structure ........................................ 2
Package subdirectories ................................. 8
Packages ..................................................... 2
Parsing R code from C ................................. 116
Platform-specific documentation .................. 61
Printing from C .......................................... 124
Printing from FORTRAN ......................... 124
Processing Rd format ................................. 64
Profiling .................................................... 65, 67, 69

R
Random numbers in C .............................. 123, 126
Random numbers in FORTRAN .............. 125
Registering native routines ....................... 88

S
Setting variables ...................................... 104
Sort functions from C ............................... 132
Submiting to CRAN ................................. 29
Sweave .................................................... 26
T
Tidying R code ......................................... 65
U
user-defined macros .................................. 62
V
Version information from C ...................... 135
vignettes .................................................. 26
Visibility ............................................... 136
W
weak reference ........................................ 118
Z
Zero-finding ............................................. 112