Writing R Extensions

Version 4.5.0 Under development (2024-04-14)

R Core Team

This manual is for R, version 4.5.0 Under development (2024-04-14).

Copyright © 1999–2024 R Core Team

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the R Core Team.

Table of Contents

Acknowledgements ... **1**

1 Creating R packages ... **2**

1.1 Package structure ... 3
1.1.1 The `DESCRIPTION` file ... 4
1.1.2 Licensing ... 8
1.1.3 Package Dependencies ... 10
1.1.3.1 Suggested packages ... 12
1.1.4 The `INDEX` file ... 13
1.1.5 Package subdirectories ... 13
1.1.6 Data in packages ... 17
1.1.7 Non-R scripts in packages ... 19
1.1.8 Specifying URLs ... 20
1.2 Configure and cleanup ... 20
1.2.1 Using `Makevars` ... 25
1.2.1.1 OpenMP support ... 28
1.2.1.2 Using pthreads ... 30
1.2.1.3 Compiling in sub-directories ... 31
1.2.2 Configure example ... 31
1.2.3 Using modern Fortran code ... 33
1.2.4 Using C++ code ... 34
1.2.5 C standards ... 36
1.2.6 Using `cmake` ... 37
1.3 Checking and building packages ... 39
1.3.1 Checking packages ... 39
1.3.2 Building package tarballs ... 43
1.3.3 Building binary packages ... 44
1.4 Writing package vignettes ... 45
1.4.1 Encodings and vignettes ... 47
1.4.2 Non-Sweave vignettes ... 48
1.5 Package namespaces ... 48
1.5.1 Specifying imports and exports ... 49
1.5.2 Registering S3 methods ... 50
1.5.3 Load hooks ... 51
1.5.4 `useDynLib` ... 51
1.5.5 An example ... 53
1.5.6 Namespaces with S4 classes and methods ... 54
1.6 Writing portable packages ... 55
1.6.1 PDF size ... 65
1.6.2 Check timing ... 65
1.6.3 Encoding issues ... 66
1.6.4 Portable C and C++ code ... 67
1.6.4.1 Common symbols ... 73
1.6.4.2 C++17 issues ... 74
1.6.5 Portable Fortran code ... 74
1.6.6 Binary distribution ... 75
1.7 Diagnostic messages ... 76
1.8 Internationalization ... 77

1.8.1 C-level messages ... 77
1.8.2 R messages ... 78
1.8.3 Preparing translations ... 78
1.9 CITATION files ... 79
1.10 Package types ... 79
1.10.1 Frontend ... 79
1.11 Services ... 80

2 Writing R documentation files ... 81
2.1 Rd format ... 81
2.1.1 Documenting functions ... 82
2.1.2 Documenting data sets ... 86
2.1.3 Documenting S4 classes and methods ... 87
2.1.4 Documenting packages ... 88
2.2 Sectioning ... 88
2.3 Marking text ... 89
2.4 Lists and tables ... 91
2.5 Cross-references ... 91
2.6 Mathematics ... 92
2.7 Figures ... 93
2.8 Insertions ... 93
2.9 Indices ... 94
2.10 Platform-specific documentation ... 94
2.11 Conditional text ... 94
2.12 Dynamic pages ... 95
2.13 User-defined macros ... 96
2.14 Encoding ... 97
2.15 Processing documentation files ... 97
2.16 Editing Rd files ... 98

3 Tidying and profiling R code ... 99
3.1 Tidying R code ... 99
3.2 Profiling R code for speed ... 99
3.3 Profiling R code for memory use ... 101
3.3.1 Memory statistics from `Rprof` ... 101
3.3.2 Tracking memory allocations ... 102
3.3.3 Tracing copies of an object ... 102
3.4 Profiling compiled code ... 102
3.4.1 Profiling on Linux ... 103
3.4.1.1 `perf` ... 103
3.4.1.2 `oprofile` and `operf` ... 103
3.4.1.3 `sprof` ... 105
3.4.2 Profiling on macOS ... 105
3.4.3 Profiling on Windows ... 105

4 Debugging ... 106

4.1 Browsing ... 106
4.2 Debugging R code ... 107
4.3 Checking memory access ... 110
4.3.1 Using `gctorture` ... 111
4.3.2 Using Valgrind ... 111
4.3.3 Using the Address Sanitizer ... 113
4.3.3.1 Using the Leak Sanitizer ... 114
4.3.4 Using the Undefined Behaviour Sanitizer ... 115
4.3.5 Other analyses with 'clang' ... 116
4.3.6 Other analyses with 'gcc' ... 117
4.3.7 Using 'Dr. Memory' ... 117
4.3.8 Fortran array bounds checking ... 117
4.4 Debugging compiled code ... 117
4.4.1 Finding entry points in dynamically loaded code ... 119
4.4.2 Inspecting R objects when debugging ... 119
4.4.3 Debugging on macOS ... 121
4.5 Using Link-time Optimization ... 122

5 System and foreign language interfaces ... 124

5.1 Operating system access ... 124
5.2 Interface functions `.C` and `.Fortran` ... 124
5.3 `dyn.load` and `dyn.unload` ... 126
5.4 Registering native routines ... 127
5.4.1 Speed considerations ... 130
5.4.2 Example: converting a package to use registration ... 131
5.4.3 Linking to native routines in other packages ... 134
5.5 Creating shared objects ... 135
5.6 Interfacing C++ code ... 136
5.6.1 External C++ code ... 138
5.7 Fortran I/O ... 138
5.8 Linking to other packages ... 139
5.8.1 Unix-alikes ... 139
5.8.2 Windows ... 140
5.9 Handling R objects in C ... 141
5.9.1 Handling the effects of garbage collection ... 142
5.9.2 Allocating storage ... 144
5.9.3 Details of R types ... 144
5.9.4 Attributes ... 145
5.9.5 Classes ... 147
5.9.6 Handling lists ... 147
5.9.7 Handling character data ... 148
5.9.8 Finding and setting variables ... 148
5.9.9 Some convenience functions ... 149
5.9.9.1 Semi-internal convenience functions ... 150
5.9.10 Named objects and copying ... 150
5.10 Interface functions `.Call` and `.External` ... 152
5.10.1 Calling `.Call` ... 152
5.10.2 Calling `.External` ... 152
5.10.3 Missing and special values ... 154
5.11 Evaluating R expressions from C ... 155

5.11.1 Zero-finding ... 156
5.11.2 Calculating numerical derivatives ... 157
5.12 Parsing R code from C ... 160
5.12.1 Accessing source references ... 161
5.13 External pointers and weak references ... 162
5.13.1 An example ... 163
5.14 Vector accessor functions ... 164
5.15 Character encoding issues ... 164

6 The R API: entry points for C code ... 166
6.1 Memory allocation ... 166
6.1.1 Transient storage allocation ... 166
6.1.2 User-controlled memory ... 167
6.2 Error signaling ... 168
6.2.1 Error signaling from Fortran ... 168
6.3 Random number generation ... 168
6.3.1 Random-number generation from Fortran ... 169
6.4 Missing and IEEE special values ... 170
6.5 Printing ... 170
6.5.1 Printing from Fortran ... 170
6.6 Calling C from Fortran and vice versa ... 171
6.6.1 Fortran character strings ... 172
6.6.2 Fortran LOGICAL ... 174
6.6.3 Passing functions ... 174
6.7 Numerical analysis subroutines ... 175
6.7.1 Distribution functions ... 175
6.7.2 Mathematical functions ... 177
6.7.3 Numerical Utilities ... 177
6.7.4 Mathematical constants ... 179
6.8 Optimization ... 180
6.9 Integration ... 181
6.10 Utility functions ... 182
6.11 Re-encoding ... 184
6.12 Condition handling and cleanup code ... 184
6.13 Allowing interrupts ... 185
6.14 Platform and version information ... 186
6.15 Inlining C functions ... 186
6.16 Controlling visibility ... 187
6.17 Using these functions in your own C code ... 188
6.18 Organization of header files ... 188

7 Generic functions and methods ... 190
7.1 Adding new generics ... 191

8 Linking GUIs and other front-ends to R ... 192
8.1 Embedding R under Unix-alikes ... 192
8.1.1 Compiling against the R library ... 194
8.1.2 Setting R callbacks ... 194
8.1.3 Registering symbols ... 197
8.1.4 Meshing event loops ... 198

8.1.5 Threading issues . . . 198
8.2 Embedding R under Windows . . . 199
8.2.1 Using (D)COM . . . 199
8.2.2 Calling `R.dll` directly . . . 199
8.2.3 Finding R_HOME . . . 202

Function and variable index . . . 204

Concept index . . . 209

Acknowledgements

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

1 Creating R packages

Packages provide a mechanism for loading optional code, data and documentation as needed. The R distribution itself includes about 30 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 `R CMD INSTALL` utility. Otherwise, please look at R's help pages on

```
?library
?INSTALL
```

before reading on.

For packages which contain code to be compiled, a computing environment including a number of tools is assumed; the "R Installation and Administration" manual describes what is needed for each OS.

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*.

Other types of extensions are supported (but rare): See Section 1.10 [Package types], page 79.

Some notes on terminology complete this introduction. These will help with the reading of this manual, and also in describing concepts accurately when asking for help.

A *package* is a directory of files which extend R, a *source package* (the master files of a package), or a tarball containing the files of a source package, or an *installed* package, the result of running `R CMD INSTALL` on a source package. On some platforms (notably macOS and '`x86_64`' Windows) there are also *binary packages*, a zip file or tarball containing the files of an installed package which can be unpacked rather than installing from sources.

A package is **not**[1] a *library*. The latter is used in two senses in R documentation.

- A directory into which packages are installed, e.g. `/usr/lib/R/library`: in that sense it is sometimes referred to as a *library directory* or *library tree* (since the library is a directory which contains packages as directories, which themselves contain directories).
- That used by the operating system, as a shared, dynamic or static library or (especially on Windows) a DLL, where the second L stands for 'library'. Installed packages may contain compiled code in what is known on Unix-alikes as a *shared object* and on Windows as a DLL. The concept of a *shared library* (*dynamic library* on macOS) as a collection of compiled code to which a package might link is also used, especially for R itself on some platforms. On most platforms these concepts are interchangeable (shared objects and DLLs can both be loaded into the R process and be linked against), but macOS distinguishes between shared objects (extension `.so`) and dynamic libraries (extension `.dylib`).

There are a number of well-defined operations on source packages.

- The most common is *installation* which takes a source package and installs it in a library using `R CMD INSTALL` or `install.packages`.
- Source packages can be *built*. This involves taking a source directory and creating a tarball ready for distribution, including cleaning it up and creating PDF/HTML documentation from any *vignettes* it may contain. Source packages (and most often tarballs) can be *checked*, when a test installation is done and tested (including running its examples); also, the contents of the package are tested in various ways for consistency and portability.
- *Compilation* is not a correct term for a package. Installing a source package which contains C, C++ or Fortran code will involve compiling that code. There is also the possibility of 'byte' compiling the R code in a package (using the facilities of package **compiler**): nowadays

[1] although this is a persistent mis-usage. It seems to stem from S, whose analogues of R's packages were officially known as *library sections* and later as *chapters*, but almost always referred to as *libraries*.

this is enabled by default for all packages. So *compiling* a package may come to mean byte-compiling its R code.

- It used to be unambiguous to talk about *loading* an installed package using `library()`, but since the advent of package namespaces this has been less clear: people now often talk about *loading* the package's namespace and then *attaching* the package so it becomes visible on the search path. Function `library` performs both steps, but a package's namespace can be loaded without the package being attached (for example by calls like `splines::ns`).

The concept of *lazy loading* of code or data is mentioned at several points. This is part of the installation, always selected for R code but optional for data. When used the R objects of the package are created at installation time and stored in a database in the `R` directory of the installed package, being loaded into the session at first use. This makes the R session start up faster and use less (virtual) memory. (For technical details, see Section "Lazy loading" in *R Internals*.)

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: current instructions are linked from `https://CRAN.R-project.org/banner.shtml#submitting`.

1.1 Package structure

The sources of an R package consist of a subdirectory containing the files `DESCRIPTION` and `NAMESPACE`, and the subdirectories `R`, `data`, `demo`, `exec`, `inst`, `man`, `po`, `src`, `tests`, `tools` and `vignettes` (some of which can be missing, but which should not be empty). The package subdirectory may also contain files `INDEX`, `configure`, `cleanup`, `LICENSE`, `LICENCE` and `NEWS`. Other files such as `INSTALL` (for non-standard installation instructions), `README/README.md`[2], or `ChangeLog` will be ignored by R, but may be useful to end users. The utility `R CMD build` may add files in a `build` directory (but this should not be used for other purposes).

Except where specifically mentioned,[3] packages should not contain Unix-style 'hidden' files/directories (that is, those whose name starts with a dot).

The `DESCRIPTION` and `INDEX` files are described in the subsections below. The `NAMESPACE` file is described in the section on Section 1.5 [Package namespaces], page 48.

The optional files `configure` and `cleanup` are (Bourne) shell scripts which are, respectively, executed before and (if option `--clean` was given) after installation on Unix-alikes, see Section 1.2 [Configure and cleanup], page 20. The analogues on Windows are `configure.win` and `cleanup.win`. Since R 4.2.0 on Windows, `configure.ucrt` and `cleanup.ucrt` are supported and take precedence over `configure.win` and `cleanup.win`. They can hence be used to provide content specific to UCRT or Rtools42 and newer, if needed, but the support for `.ucrt` files may be removed in future when building packages from source on the older versions of R will no longer be needed, and hence the files may be renamed back to `.win`.

For the conventions for files `NEWS` and `ChangeLog` in the GNU project see `https://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 macOS) are not case-sensitive, to maintain portability it is strongly recommended that case distinctions not be used to distinguish different

[2] This seems to be commonly used for a file in 'markdown' format. Be aware that most users of R will not know that, nor know how to view such a file: platforms such as macOS and Windows do not have a default viewer set in their file associations. The CRAN package web pages render such files in HTML: the converter used expects the file to be encoded in UTF-8.

[3] currently, top-level files `.Rbuildignore` and `.Rinstignore`, and `vignettes/.install_extras`.

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 systems, the ASCII control characters as well as the characters '`"`', '`*`', '`:`', '`/`', '`<`', '`>`', '`?`', '`\`', and '`|`' are not allowed in file names. In addition, files with names '`con`', '`prn`', '`aux`', '`clock$`', '`nul`', '`com1`' to '`com9`', and '`lpt1`' to '`lpt9`' after conversion to lower case and stripping possible "extensions" (e.g., '`lpt5.foo.bar`'), are disallowed. Also, file names in the same directory must not differ only by case (see the previous paragraph). In addition, the basenames of '`.Rd`' files may be used in URLs and so must be ASCII and not contain `%`. For maximal portability filenames should only contain only ASCII characters not excluded already (that is `A-Za-z0-9._!#$%&+,;=@^(){}'[]` — 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 `(){}'[]$~`: `~` is also used as part of '8.3' filenames on Windows. In addition, packages are normally distributed as tarballs, and these have a limit on path lengths: for maximal portability 100 bytes.

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[4] unless they are listed (one filepath per line) in a file `BinaryFiles` at the top level of the package. Note that CRAN will not 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: 2015-01-01
Title: My First Collection of Functions
Authors@R: c(person("Joe", "Developer", role = c("aut", "cre"),
                    email = "Joe.Developer@some.domain.net"),
             person("Pat", "Developer", role = "aut"),
             person("A.", "User", role = "ctb",
                    email = "A.User@whereever.net"))
Author: Joe Developer [aut, cre],
  Pat Developer [aut],
  A. User [ctb]
Maintainer: Joe Developer <Joe.Developer@some.domain.net>
Depends: R (>= 3.1.0), nlme
Suggests: MASS
Description: A (one paragraph) description of what
  the package does and why it may be useful.
License: GPL (>= 2)
URL: https://www.r-project.org, http://www.another.url
BugReports: https://pkgname.bugtracker.url
```

The format is that of a version of a 'Debian Control File' (see the help for '`read.dcf`' and `https://www.debian.org/doc/debian-policy/ch-controlfields.html`: R does not require encoding in UTF-8 and does not support comments starting with '`#`'). Fields start with an ASCII name immediately followed by a colon: the value starts after the colon and a space. Continuation lines (for example, for descriptions longer than one line) start with a space or tab. Field names are case-sensitive: all those used by R are capitalized.

[4] false positives are possible, but only a handful have been seen so far.

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).

Several optional fields take *logical values*: these can be specified as '`yes`', '`true`', '`no`' or '`false`': capitalized values are also accepted.

The '`Package`', '`Version`', '`License`', '`Description`', '`Title`', '`Author`', and '`Maintainer`' fields are mandatory, all other fields are optional. Fields '`Author`' and '`Maintainer`' can be auto-generated from '`Authors@R`', and may be omitted if the latter is provided: however if they are not ASCII we recommend that they are provided.

The mandatory '`Package`' field gives the name of the package. This should contain only (ASCII) letters, numbers and dot, have at least two characters and start with a letter and not end in a dot. If it needs explaining, this should be done in the '`Description`' field (and not the '`Title`' field).

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`'. It is **not** a decimal number, so for example `0.9 < 0.75` since `9 < 75`.

The mandatory '`License`' field is discussed in the next subsection.

The mandatory '`Title`' field should give a *short* description of the package. Some package listings may truncate the title to 65 characters. It should use *title case* (that is, use capitals for the principal words: `tools::toTitleCase` can help you with this), not use any markup, not have any continuation lines, and not end in a period (unless part of ...). Do not repeat the package name: it is often used prefixed by the name. Refer to other packages and external software in single quotes, and to book titles (and similar) in double quotes.

The mandatory '`Description`' field should give a *comprehensive* description of what the package does. One can use several (complete) sentences, but only one paragraph. It should be intelligible to all the intended readership (e.g. for a CRAN package to all CRAN users). It is good practice not to start with the package name, 'This package' or similar. As with the '`Title`' field, double quotes should be used for quotations (including titles of books and articles), and single quotes for non-English usage, including names of other packages and external software. This field should also be used for explaining the package name if necessary. URLs should be enclosed in angle brackets, e.g. '`<https://www.r-project.org>`': see also Section 1.1.8 [Specifying URLs], page 20.

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: for that use '`Authors@R`'). Note that all significant contributors must be included: if you wrote an R wrapper for the work of others included in the `src` directory, you are not the sole (and maybe not even the main) author.

The mandatory '`Maintainer`' field should give a *single* name followed by a *valid* (RFC 2822) email address in angle brackets. It should not end in a period or comma. This field is what is reported by the `maintainer` function and used by `bug.report`. For a CRAN 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.

Note that the *display name* (the part before the address in angle brackets) should be enclosed in double quotes if it contains non-alphanumeric characters such as comma or period. (The current standard, RFC 5322, allows periods but RFC 2822 did not.)

Both '`Author`' and '`Maintainer`' fields can be omitted if a suitable '`Authors@R`' field is given. This field can be used to provide a refined and machine-readable description of the package "authors" (in particular specifying their precise *roles*), *via* suitable R code. It should create

an object of class "person", by either a call to person or a series of calls (one per "author") concatenated by c(): see the example DESCRIPTION file above. The roles can include '"aut"' (author) for full authors, '"cre"' (creator) for the package maintainer, and '"ctb"' (contributor) for other contributors, '"cph"' (copyright holder, which should be the legal name for an institution or corporate body), among others. See ?person for more information. Note that no role is assumed by default. Auto-generated package citation information takes advantage of this specification. The 'Author' and 'Maintainer' fields are auto-generated from it if needed when building[5] or installing.

An optional 'Copyright' field can be used where the copyright holder(s) are not the authors. If necessary, this can refer to an installed file: the convention is to use file inst/COPYRIGHTS.

The optional 'Date' field gives the *release date* of the current version of the package. It is strongly recommended[6] to use the 'yyyy-mm-dd' format conforming to the ISO 8601 standard.

The 'Depends', 'Imports', 'Suggests', 'Enhances', 'LinkingTo' and 'Additional_repositories' fields are discussed in a later subsection.

Dependencies external to the R system should be listed in the 'SystemRequirements' field, possibly amplified in a separate README file. This includes specifying a non-default C++ standard and the need for GNU make.

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. See Section 1.1.8 [Specifying URLs], page 20.

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.report instead of sending an email to the maintainer. A browser is opened for a 'http://' or 'https://' URL. To specify another email address for bug reports, use 'Contact' instead: however bug.report will try to extract an email address (preferably from a 'mailto:' URL or enclosed in angle brackets) from 'BugReports'.

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.

A '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 1.1.5 [Package subdirectories], page 13) 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 in preference to 'Collate'.

The 'LazyData' logical field controls whether the R datasets use lazy-loading. A 'LazyLoad' field was used in versions prior to 2.14.0, but now is ignored.

The 'KeepSource' logical field controls if the package code is sourced using keep.source = TRUE or FALSE: it might be needed exceptionally for a package designed to always be used with keep.source = TRUE.

The 'ByteCompile' logical field controls if the package R code is to be byte-compiled on installation: the default is to byte-compile. This can be overridden by installing with flag --no-byte-compile.

[5] at least if this is done in a locale which matches the package encoding.

[6] and required by CRAN, so checked by R CMD check --as-cran.

The 'UseLTO' logical field is used to indicate if source code in the package[7] is to be compiled with Link-Time Optimization (see Section 4.5 [Using Link-time Optimization], page 122) if R was installed with --enable-lto (default true) or --enable-lto=R (default false) (or on Windows[8] if LTO_OPT is set in MkRules). This can be overridden by the flags --use-LTO and --no-use-LTO. LTO is said to give most size and performance improvements for large and complex (heavily templated) C++ projects.

The 'StagedInstall' logical field controls if package installation is 'staged', that is done to a temporary location and moved to the final location when successfully completed. This field was introduced in R 3.6.0 and it true by default: it is considered to be a temporary measure which may be withdrawn in future.

The 'ZipData' logical field has been ignored since R 2.13.0.

The 'Biarch' logical field is used on Windows to select the INSTALL option --force-biarch for this package. Not currently relevant.

The 'BuildVignettes' logical field can be set to a false value to stop R CMD build from attempting to build the vignettes, as well as preventing[9] R CMD check from testing this. This should only be used exceptionally, for example if the PDFs include large figures which are not part of the package sources (and hence only in packages which do not have an Open Source license).

The 'VignetteBuilder' field names (in a comma-separated list) packages that provide an engine for building vignettes. These may include the current package, or ones listed in 'Depends', 'Suggests' or 'Imports'. The **utils** package is always implicitly appended. See Section 1.4.2 [Non-Sweave vignettes], page 48, for details. Note that if, for example, a vignette has engine 'knitr::rmarkdown', then **knitr** (https://CRAN.R-project.org/package=knitr) provides the engine but both **knitr** and **rmarkdown** (https://CRAN.R-project.org/package=rmarkdown) are needed for using it, so *both* these packages need to be in the 'VignetteBuilder' field and at least suggested (as **rmarkdown** is only suggested by **knitr**, and hence not available automatically along with it). Many packages using **knitr** (https://CRAN.R-project.org/package=knitr) also need the package **formatR** (https://CRAN.R-project.org/package=formatR) which it suggests and so the user package needs to do so too and include this in 'VignetteBuilder'.

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 and 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. If a package has a specified encoding, you should run R CMD build etc in a locale using that encoding.)

The 'NeedsCompilation' field should be set to "yes" if the package contains native code which needs to be compiled, otherwise "no" (when the package could be installed from source on any platform without additional tools). This is used by install.packages(type = "both") in R >= 2.15.2 on platforms where binary packages are the norm: it is normally set by R CMD build or the repository assuming compilation is required if and only if the package has a src directory.

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 1.10 [Package types], page 79.

[7] without a src/Makefile* file.

[8] LTO is not currently supported by the toolchain used on 'aarch64'.

[9] But it is checked for Open Source packages by R CMD check --as-cran.

One can add subject classifications for the content of the package using the fields 'Classification/ACM' or 'Classification/ACM-2012' (using the Computing Classification System of the Association for Computing Machinery, https://www.acm.org/publications/class-2012; the former refers to the 1998 version), 'Classification/JEL' (the Journal of Economic Literature Classification System, https://www.aeaweb.org/econlit/jelCodes.php, or 'Classification/MSC' or 'Classification/MSC-2010' (the Mathematics Subject Classification of the American Mathematical Society, https://mathscinet.ams.org/msc/msc2010.html; the former refers to the 2000 version). 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'.

A '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 grandfathered) IETF language tags as currently defined by RFC 5646 (https://www.rfc-editor.org/rfc/rfc5646, see also https://en.wikipedia.org/wiki/IETF_language_tag), i.e., use language subtags which in essence are 2-letter ISO 639-1 (https://en.wikipedia.org/wiki/ISO_639-1) or 3-letter ISO 639-3 (https://en.wikipedia.org/wiki/ISO_639-3) language codes.

An 'RdMacros' field can be used to hold a comma-separated list of packages from which the current package will import Rd macro definitions. These package should also be listed in 'Imports' (or 'Depends'). The macros in these packages will be imported after the system macros, in the order listed in the 'RdMacros' field, before any macro definitions in the current package are loaded. Macro definitions in individual .Rd files in the man directory are loaded last, and are local to later parts of that file. In case of duplicates, the last loaded definition will be used.[10] Both R CMD Rd2pdf and R CMD Rdconv have an optional flag --RdMacros=pkglist. The option is also a comma-separated list of package names, and has priority over the value given in DESCRIPTION. Packages using Rd macros should depend on R 3.2.0 or later.

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

There is no restriction on the use of other fields not mentioned here (but using other capitalizations of these field names would cause confusion). Fields Note, Contact (for contacting the authors/developers[11]) and MailingList are in common use. Some repositories (including CRAN and R-forge) add their own fields.

1.1.2 Licensing

Licensing for a package which might be distributed is an important but potentially complex subject.

It is very important that you include license information! Otherwise, it may not even be legally correct for others to distribute copies of the package, let alone use it.

The package management tools use the concept of 'free or open source software' (FOSS, e.g., https://en.wikipedia.org/wiki/FOSS) licenses: the idea being that some users of R and its packages want to restrict themselves to such software. Others need to ensure that there are no restrictions stopping them using a package, e.g. forbidding commercial or military use. It is a central tenet of FOSS software that there are no restrictions on users nor usage.

Do not use the 'License' field for information on copyright holders: if needed, use a 'Copyright' field.

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

[10] Duplicate definitions may trigger a warning: see Section 2.13 [User-defined macros], page 96.

[11] bug.report will try to extract an email address from a Contact field if there is no BugReports field.

- One of the "standard" short specifications

  ```
  GPL-2 GPL-3 LGPL-2 LGPL-2.1 LGPL-3 AGPL-3 Artistic-2.0
  BSD_2_clause BSD_3_clause MIT
  ```

  as made available *via* `https://www.R-project.org/Licenses/` and contained in subdirectory `share/licenses` of the R source or home directory.
- The names or abbreviations of other licenses 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 '`-`'.

  Abbreviations `GPL` and `LGPL` are ambiguous and usually[12] taken to mean any version of the license: but it is better not to use them.
- 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).

Multiple licences can be specified separated by '`|`' (surrounded by spaces) in which case the user can choose any of the alternatives.

If a package license *restricts* a base license (where permitted, e.g., using GPL-3 or AGPL-3 with an attribution clause), the additional terms 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 (preferably with the '`+`' surrounded by spaces). Note that several commonly used licenses do not permit restrictions: this includes GPL-2 and hence any specification which includes it.

Examples of standardized specifications include

```
License: GPL-2
License: LGPL (>= 2.0, < 3) | Mozilla Public License
License: GPL-2 | file LICENCE
License: GPL (>= 2) | BSD_3_clause + file LICENSE
License: Artistic-2.0 | AGPL-3 + file LICENSE
```

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

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.

Fields '`License_is_FOSS`' and '`License_restricts_use`' may be added by repositories where information cannot be computed from the name of the license. '`License_is_FOSS: yes`' is used for licenses which are known to be FOSS, and '`License_restricts_use`' can have values '`yes`' or '`no`' if the `LICENSE` file is known to restrict users or usage, or known not to. These are used by, e.g., the `available.packages` filters.

The optional file `LICENSE`/`LICENCE` contains a copy of the license of the package. To avoid any confusion only include such a file if it is referred to in the '`License`' field of the `DESCRIPTION` file.

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

[12] CRAN expands them to e.g. `GPL-2 | GPL-3`.

the copies on `https://www.R-project.org/Licenses/` and included in the R distribution (in directory `share/licenses`). Since files named `LICENSE` or `LICENCE` *will* be installed, do not use these names for standard license files. To include comments about the licensing rather than the body of a license, use a file named something like `LICENSE.note`.

A few "standard" licenses are rather license templates which need additional information to be completed *via* '`+ file LICENSE`' (with the '`+`' surrounded by spaces). Where the additional information is '`COPYRIGHT HOLDER`', this must give the actual legal entities (not something vague like 'Name-of-package authors'): if more than one they should be listed in decreasing order of contribution.

1.1.3 Package Dependencies

The '`Depends`' field gives a comma-separated list of package names which this package depends on. Those packages will be attached before the current package when `library` or `require` is called. Each package name may be optionally followed by a comment in parentheses specifying a version requirement. The comment should contain a comparison operator, whitespace and a valid version number, e.g. '`MASS (>= 3.1-20)`'.

The '`Depends`' field can also specify a dependence on a certain version of R — e.g., if the package works only with R version 4.0.0 or later, include '`R (>= 4.0)`' in the '`Depends`' field. (As here, trailing zeroes can be dropped and it is recommended that they are.) You can also require a certain SVN revision for R-devel or R-patched, e.g. '`R (>= 2.14.0), R (>= r56550)`' requires a version later than R-devel of late July 2011 (including released versions of 2.14.0).

It makes no sense to declare a dependence on `R` without a version specification, nor on the package **base**: this is an R package and package **base** is always available.

A package or '`R`' can appear more than once in the '`Depends`' field, for example to give upper and lower bounds on acceptable versions.

It is inadvisable to use a dependence on R with patch level (the third digit) other than zero. Doing so with packages which others depend on will cause the other packages to become unusable under earlier versions in the series, and e.g. versions 4.x.1 are widely used throughout the Northern Hemisphere academic year.

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. 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.

The '`Imports`' field lists packages whose namespaces are imported from (as specified in the `NAMESPACE` file) but which do not need to be attached. Namespaces 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 and are checked when the namespace is loaded.

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 45), and packages loaded in the body of functions. E.g., suppose an example[13] 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

[13] even one wrapped in `\donttest`.

have **bar**, but not necessary. Version requirements can be specified but should be checked by the code which uses the package.

Finally, the 'Enhances' field lists packages "enhanced" by the package at hand, e.g., by providing methods for classes from these packages, or ways to handle objects from these packages (so several packages have 'Enhances: chron' because they can handle datetime objects from **chron** (https://CRAN.R-project.org/package=chron) even though they prefer R's native datetime functions). Version requirements can be specified, but are currently not used. Such packages cannot be required to check the package: any tests which use them must be conditional on the presence of the package. (If your tests use e.g. a dataset from another package it should be in 'Suggests' and not 'Enhances'.)

The general rules are

- A package should be listed in only one of these fields.
- Packages whose namespace only is needed to load the package using `library(`*pkgname*`)` should be listed in the 'Imports' field and not in the 'Depends' field. Packages listed in `import` or `importFrom` directives in the NAMESPACE file should almost always be in 'Imports' and not 'Depends'.
- Packages that need to be attached to successfully load the package using `library(`*pkgname*`)` must be listed in the 'Depends' field.
- All packages that are needed[14] 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.)
- Packages needed to use datasets from the package should be in 'Imports': this includes those needed to define S4 classes used.

In particular, 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' and 'Imports' fields are used by `library` when it loads the package, and `install.packages` checks versions for the 'Depends', 'Imports' and (for `dependencies = TRUE`) 'Suggests' fields.

It is important that the information in these fields is complete and accurate: it is for example used to compute which packages depend on an updated package and which packages can safely be installed in parallel.

This scheme was developed before all packages had namespaces (R 2.14.0 in October 2011), and good practice changed once that was in place.

Field 'Depends' should nowadays be used rarely, only for packages which are intended to be put on the search path to make their facilities available to the end user (and not to the package itself): for example it makes sense that a user of package **latticeExtra** (https://CRAN.R-project.org/package=latticeExtra) would want the functions of package **lattice** (https://CRAN.R-project.org/package=lattice) made available.

Almost always packages mentioned in 'Depends' should also be imported from in the NAMESPACE file: this ensures that any needed parts of those packages are available when some other package imports the current package.

[14] This includes all packages directly called by `library` and `require` calls, as well as data obtained *via* `data(theirdata, package = "somepkg")` calls: `R CMD check` will warn about all of these. But there are subtler uses which it may not detect: e.g. if package A uses package B and makes use of functionality in package B which uses package C which package B suggests or enhances, then package C needs to be in the 'Suggests' list for package A. Nor will undeclared uses in included files be reported, nor unconditional uses of packages listed under 'Enhances'. `R CMD check --as-cran` will detect more of the subtler uses.

The 'Imports' field should not contain packages which are not imported from (*via* the NAMESPACE file or :: or ::: operators), as all the packages listed in that field need to be installed for the current package to be installed. (This is checked by R CMD check.)

R code in the package should call library or require only exceptionally. Such calls are never needed for packages listed in 'Depends' as they will already be on the search path. It used to be common practice to use require calls for packages listed in 'Suggests' in functions which used their functionality, but nowadays it is better to access such functionality *via* :: calls.

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

```
LinkingTo: link1, link2
```

The 'LinkingTo' field can have a version requirement which is checked at installation.

Specifying a package in 'LinkingTo' suffices if these are C/C++ headers containing source code or static linking is done at installation: the packages do not need to be (and usually should not be) listed in the 'Depends' or 'Imports' fields. This includes CRAN package **BH** (https://CRAN.R-project.org/package=BH) and almost all users of **RcppArmadillo** (https://CRAN.R-project.org/package=RcppArmadillo) and **RcppEigen** (https://CRAN.R-project.org/package=RcppEigen). Note that 'LinkingTo' applies only to installation: if a packages wishes to use headers to compile code in tests or vignettes the package providing them needs to be listed in 'Suggests' or perhaps 'Depends'.

For another use of 'LinkingTo' see Section 5.4.3 [Linking to native routines in other packages], page 134.

The 'Additional_repositories' field is a comma-separated list of repository URLs where the packages named in the other fields may be found. It is currently used by R CMD check to check that the packages can be found, at least as source packages (which can be installed on any platform).

1.1.3.1 Suggested packages

Note that someone wanting to run the examples/tests/vignettes may not have a suggested package available (and it may not even be possible to install it for that platform). The recommendation used to be to make their use conditional *via* if(require("*pkgname*")): this is OK if that conditioning is done in examples/tests/vignettes, although using if(requireNamespace("*pkgname*")) is preferred, if possible.

However, using require for conditioning *in package code* is not good practice as it alters the search path for the rest of the session and relies on functions in that package not being masked by other require or library calls. It is better practice to use code like

```
if (requireNamespace("rgl", quietly = TRUE)) {
   rgl::plot3d(...)
} else {
   ## do something else not involving rgl.
}
```

Note the use of rgl:: as that object would not necessarily be visible (and if it is, it need not be the one from that namespace: plot3d occurs in several other packages). If the intention is to give an error if the suggested package is not available, simply use e.g. rgl::plot3d.

If the conditional code produces print output, function withAutoprint can be useful.

Note that the recommendation to use suggested packages conditionally in tests does also apply to packages used to manage test suites: a notorious example was **testthat** (https://CRAN.R-project.org/package=testthat) which in version 1.0.0 contained illegal C++ code and hence could not be installed on standards-compliant platforms.

Some people have assumed that a 'recommended' package in '`Suggests`' can safely be used unconditionally, but this is not so. (R can be installed without recommended packages, and which packages are 'recommended' may change.)

As noted above, packages in '`Enhances`' *must* be used conditionally and hence objects within them should always be accessed *via* `::`.

On most systems, `R CMD check` can be run with only those packages declared in '`Depends`' and '`Imports`' by setting environment variable `_R_CHECK_DEPENDS_ONLY_=true`, whereas setting `_R_CHECK_SUGGESTS_ONLY_=true` also allows suggested packages, but not those in '`Enhances`' nor those not mentioned in the `DESCRIPTION` file. It is recommended that a package is checked with each of these set, as well as with neither.

WARNING: Be extremely careful if you do things which would be run at installation time depending on whether suggested packages are available or not—this includes top-level code in R code files, `.onLoad` functions and the definitions of S4 classes and methods. The problem is that once a namespace of a suggested package is loaded, references to it may be captured in the installed package (most commonly in S4 methods), but the suggested package may not be available when the installed package is used (which especially for binary packages might be on a different machine). Even worse, the problems might not be confined to your package, for the namespaces of your suggested packages will also be loaded whenever any package which imports yours is installed and so may be captured there.

1.1.4 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.

The file is part of the information given by `library(help = pkgname)`.

Rather than editing this file, it is preferable to put customized information about the package into an overview help page (see Section 2.1.4 [Documenting packages], page 88) and/or a vignette (see Section 1.4 [Writing package vignettes], page 45).

1.1.5 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[15] `.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 functions in the '`Depends`' packages provided that the objects created do not depend on those packages except *via* namespace imports.

Extreme care is needed if top-level computations are made to depend on availability or not of other packages. In particular this applies to `setMethods` and `setClass` calls. Nor should they depend on the availability of external resources such as downloads.

Two exceptions are allowed: if the `R` subdirectory contains a file `sysdata.rda` (a saved image of one or more R objects: please use suitable compression as suggested by

[15] Extensions `.S` and `.s` arise from code originally written for S(-PLUS), but are commonly used for assembler code. Extension `.q` was used for S, which at one time was tentatively called QPE.

`tools::resaveRdaFiles`, and see also the '`SysDataCompression`' `DESCRIPTION` field.) this will be lazy-loaded into the namespace 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, form feed, LF and CR) should be used in code files. Other characters are accepted in comments[16], but then the comments may not be readable in e.g. a UTF-8 locale. Non-ASCII characters in object names will normally[17] fail when the package is installed. Any byte will be allowed in a quoted character string but '`\uxxxx`' escapes should be used for non-ASCII characters. However, 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. See Section 1.5.3 [Load hooks], page 51.

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[18] they must be entirely ASCII and not contain '`%`'. See Chapter 2 [Writing R documentation files], page 81, 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. Note that packages which use internal objects extensively should not export those objects from their namespace, when they do not need to be documented (see Section 1.5 [Package namespaces], page 48).

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

The `man` subdirectory may contain a subdirectory named `macros`; this will contain source for user-defined Rd macros. (See Section 2.13 [User-defined macros], page 96.) These use the Rd format, but may not contain anything but macro definitions, comments and whitespace.

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` (or for use on Windows, with extension `.win` or `.ucrt`). 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*`/etc`*R_ARCH*`/Makeconf`), providing support for C, C++, fixed- or free-form Fortran, Objective C and Objective C++[19] with associated extensions `.c`, `.cc` or `.cpp`, `.f`, `.f90` or `.f95`,[20] `.m`, and `.mm`, respectively. We recommend using `.h` for headers, also for C++[21] or Fortran include files. (Use of extension `.C` for C++ is no longer supported.) Files in the `src` directory should not be hidden (start with a dot), and hidden files will under some versions of R be ignored.

It is not portable (and may not be possible at all) to mix all these languages in a single package. Because R itself uses it, we know that C and fixed-form Fortran can be used together, and mixing C, C++ and Fortran usually work for the platform's native compilers.

---

[16] but they should be in the encoding declared in the `DESCRIPTION` file.

[17] This is true for OSes which implement the '`C`' locale: Windows' idea of the '`C`' locale uses the WinAnsi charset.

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

[19] either or both of which may not be supported on particular platforms. Their main use is on macOS, but unfortunately recent versions of the macOS SDK have removed much of the support for Objective C v1.0 and Objective C++.

[20] This is not accepted by the Intel Fortran compiler.

[21] Using `.hpp` is not guaranteed to be portable.

If your code needs to depend on the platform there are certain defines which can used in C or C++. On all Windows builds (even 64-bit ones) '`_WIN32`' will be defined: on 64-bit Windows builds also '`_WIN64`'. For Windows on ARM, test for '`_M_ARM64`' or both '`_WIN32`' and '`__aarch64__`'. On macOS '`__APPLE__`' is defined[22]; for an 'Apple Silicon' platform, test for both '`__APPLE__`' and '`__arm64__`'.

The default rules can be tweaked by setting macros[23] in a file `src/Makevars` (see Section 1.2.1 [Using Makevars], page 25). Note that this mechanism should be general enough to eliminate the need for a package-specific `src/Makefile`. If such a file is to be distributed, considerable care is needed to make it general enough to work on all R platforms. 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. Since R 4.2.0, `src/Makevars.ucrt` takes precedence over `src/Makevars.win` and `src/Makefile.ucrt` takes precedence over `src/Makefile.win`. `src/Makevars.ucrt` and `src/Makefile.ucrt` will be ignored by earlier versions of R, and hence can be used to provide content specific to UCRT or Rtools42 and newer, but the support for `.ucrt` files may be removed in the future when building packages from source on the older versions of R will no longer be needed, and hence the files may be renamed back to `.win`. 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 object (e.g. to create executables). Such packages should have files `src/Makefile` and `src/Makefile.win` or `src/Makefile.ucrt` (unless intended for only Unix-alikes or only Windows). Note that on Unix such makefiles are included after *R_HOME*`/etc/`*R_ARCH*`/Makeconf` so all the usual R macros and make rules are available – for example C compilation will by default use the C compiler and flags with which R was configured. This also applies on Windows as from R 4.3.0: packages intended to be used with earlier versions should include that file themselves.

The order of inclusion of makefiles for a package which does **not** have a `src/Makefile` file is

Unix-alike	**Windows**
`src/Makevars`	`src/Makevars.ucrt`, `src/Makevars.win`
R_HOME`/etc/`*R_ARCH*`/Makeconf`	*R_HOME*`/etc/`*R_ARCH*`/Makeconf`
`R_MAKEVARS_SITE`, *R_HOME*`/etc/`*R_ARCH*`/Makevars.site`	`R_MAKEVARS_SITE`, *R_HOME*`/etc/`*R_ARCH*`/Makevars.site`
R_HOME`/share/make/shlib.mk`	*R_HOME*`/share/make/winshlib.mk`
`R_MAKEVARS_USER`, `~/.R/Makevars-`*platform*, `~/.R/Makevars`	`R_MAKEVARS_USER`, `~/.R/Makevars.ucrt`, `~/.R/Makevars.win64`, `~/.R/Makevars.win`
For those which do, it is	
R_HOME`/etc/`*R_ARCH*`/Makeconf`	*R_HOME*`/etc/`*R_ARCH*`/Makeconf`
`R_MAKEVARS_SITE`, *R_HOME*`/etc/`*R_ARCH*`/Makevars.site`	`R_MAKEVARS_SITE`, *R_HOME*`/etc/`*R_ARCH*`/Makevars.site`
`src/Makefile`	`src/Makefile.ucrt`, `src/Makefile.win`
`R_MAKEVARS_USER`, `~/.R/Makevars-`*platform*, `~/.R/Makevars`	`R_MAKEVARS_USER`, `~/.R/Makevars.ucrt`, `~/.R/Makevars.win64`, `~/.R/Makevars.win`

Items in capitals are environment variables: those separated by commas are alternatives looked for in the order shown.

[22] There is also '`__APPLE_CC__`', but that indicates a compiler with Apple-specific features not the OS, although for historical reasons is is defined by LLVM `clang`. It is used in `Rinlinedfuns.h`.

[23] the POSIX terminology, called 'make variables' by GNU make.

In very special cases packages may create binary files other than the shared objects/DLLs in the `src` directory. Such files will not be installed in a multi-architecture setting since `R CMD INSTALL --libs-only` is used to merge multiple sub-architectures and it only copies shared objects/DLLs. If a package wants to install other binaries (for example executable programs), it should provide an R script `src/install.libs.R` which will be run as part of the installation in the `src` build directory *instead of* copying the shared objects/DLLs. The script is run in a separate R environment containing the following variables: `R_PACKAGE_NAME` (the name of the package), `R_PACKAGE_SOURCE` (the path to the source directory of the package), `R_PACKAGE_DIR` (the path of the target installation directory of the package), `R_ARCH` (the arch-dependent part of the path, often empty), `SHLIB_EXT` (the extension of shared objects) and `WINDOWS` (`TRUE` on Windows, `FALSE` elsewhere). Something close to the default behavior could be replicated with the following `src/install.libs.R` file:

```
files <- Sys.glob(paste0("*", SHLIB_EXT))
dest <- file.path(R_PACKAGE_DIR, paste0('libs', R_ARCH))
dir.create(dest, recursive = TRUE, showWarnings = FALSE)
file.copy(files, dest, overwrite = TRUE)
if(file.exists("symbols.rds"))
    file.copy("symbols.rds", dest, overwrite = TRUE)
```

On the other hand, executable programs could be installed along the lines of

```
execs <- c("one", "two", "three")
if(WINDOWS) execs <- paste0(execs, ".exe")
if ( any(file.exists(execs)) ) {
  dest <- file.path(R_PACKAGE_DIR,  paste0('bin', R_ARCH))
  dir.create(dest, recursive = TRUE, showWarnings = FALSE)
  file.copy(execs, dest, overwrite = TRUE)
}
```

Note the use of architecture-specific subdirectories of `bin` where needed. (Executables should installed under a `bin` directory and not under `libs`. It is good practice to check that they can be executed as part of the installation script, so a broken package is not installed.)

The `data` subdirectory is for data files: See Section 1.1.6 [Data in packages], page 17.

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 a tab or at least three spaces. (This index file is not generated automatically.) Note that a demo does not have a specified encoding and so should be an ASCII file (see Section 1.6.3 [Encoding issues], page 66). Function `demo()` will use the package encoding if there is one, but this is mainly useful for non-ASCII comments.

The contents of the `inst` subdirectory will be copied recursively to the installation directory. Subdirectories of `inst` should not interfere with those used by R (currently, `R`, `data`, `demo`, `exec`, `libs`, `man`, `help`, `html` and `Meta`, and earlier versions used `latex`, `R-ex`). The copying of the `inst` happens after `src` is built so its `Makefile` can create files to be installed. 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 case-insensitively against the file and directory paths, e.g. `doc/.*[.]png$` will exclude all PNG files in `inst/doc` based on the extension.

Note that with the exceptions of `INDEX`, `LICENSE`/`LICENCE` 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 macOS

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 version in `inst` will be that seen in the installed package.

Things you might like to add to `inst` are a `CITATION` file for use by the `citation` function, and a `NEWS.Rd` file for use by the `news` function. See its help page for the specific format restrictions of the `NEWS.Rd` file.

Another file sometimes needed in `inst` is `AUTHORS` or `COPYRIGHTS` to specify the authors or copyright holders when this is too complex to put in the `DESCRIPTION` file.

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` (or `.r` as from R 3.4.0) 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[24] `.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`*`)`. Note that the package-specific tests are run in a vanilla R session without setting the random-number seed, so tests which use random numbers will need to set the seed to obtain reproducible results (and it can be helpful to do so in all cases, to avoid occasional failures when tests are run).

If directory `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. Reference output should be produced without having the `--timings` option set (and note that `--as-cran` sets it).

If reference output is included for examples, tests or vignettes do make sure that it is fully reproducible, as it will be compared verbatim to that produced in a check run, unless the '`IGNORE_RDIFF`' markup is used. Things which trip up maintainers include displayed version numbers from loading other packages, printing numerical results to an unreproducibly high precision and printing timings. Another trap is small values which are in fact rounding error from zero: consider using `zapsmall`.

Subdirectory `exec` could contain additional executable scripts the package needs, typically scripts for interpreters such as the shell, Perl, or Tcl. NB: only files (and not directories) under `exec` are installed (and those with names starting with a dot are ignored), and they are all marked as executable (mode `755`, moderated by '`umask`') on POSIX platforms. Note too that this is not suitable for executable *programs* since some platforms support multiple architectures using the same installed package directory.

Subdirectory `po` is used for files related to *localization*: see Section 1.8 [Internationalization], page 77.

Subdirectory `tools` is the preferred place for auxiliary files needed during configuration, and also for sources need to re-create scripts (e.g. M4 files for `autoconf`: some prefer to put those in a subdirectory `m4` of `tools`).

1.1.6 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: the default

[24] 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 --no-echo` and with environment variable `LANGUAGE=en` set to get messages in English. Be careful not to use output with the option `--timings` (and note that `--as-cran` sets it).

is not to do so.) 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[25] CSV format), or `save()` images (`.RData` or `.rda`). The files should not be hidden (have names starting with a dot). Note that R code should be if possible "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 or its namespace: it should run as silently as possible and not change the `search()` path by attaching packages or other environments.

Images (extensions `.RData`[26] or `.rda`) can contain references to the namespaces 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` fields, as otherwise it may be impossible to install the package. To check for such references, load all the images into a vanilla R session, run `str()` on all the datasets, and look at the output of `loadedNamespaces()`.

Particular care is needed where a dataset or one of its components is of an S4 class, especially if the class is defined in a different package. First, the package containing the class definition has to be available to do useful things with the dataset, so that package must be listed in `Imports` or `Depends` (even if this gives a check warning about unused imports). Second, the definition of an S4 class can change, and often is unnoticed when in a package with a different author. So it may be wiser to use the `.R` form and use that to create the dataset object when needed (loading package namespaces but not attaching them by using `requireNamespace(`*`pkg`*`, quietly = TRUE)` and using *`pkg`*`::` to refer to objects in the namespace).

If you are not using '`LazyData`' and either your data files are large or e.g., you use `data/foo.R` scripts to produce your data, loading your namespace, you can speed up installation by providing a file `datalist` in the `data` subdirectory. This should have one line per topic that `data()` will find, in the format '`foo`' if `data(foo)` provides '`foo`', or '`foo: bar bah`' if `data(foo)` provides '`bar`' and '`bah`'. `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`.

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). 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.

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.

Many 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`: value `none` is also accepted. 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. A function to do that (quoting sizes in KB) is

```
CheckLazyDataCompression <- function(pkg)
```

[25] e.g. `https://www.rfc-editor.org/rfc/rfc4180`.

[26] People who have trouble with case are advised to use `.rda` as a common error is to refer to `abc.RData` as `abc.Rdata`!

```
{
    pkg_name <- sub("_.*", "", pkg)
    lib <- tempfile(); dir.create(lib)
    zs <- c("gzip", "bzip2", "xz")
    res <- integer(3); names(res) <- zs
    for (z in zs) {
        opts <- c(paste0("--data-compress=", z),
                  "--no-libs", "--no-help", "--no-demo", "--no-exec", "--no-test-load")
        install.packages(pkg, lib, INSTALL_opts = opts, repos = NULL, quiet = TRUE)
        res[z] <- file.size(file.path(lib, pkg_name, "data", "Rdata.rdb"))
    }
    ceiling(res/1024)
}
```

(applied to a source package without any 'LazyDataCompression' field). R CMD check will warn if it finds a *pkgname*/data/Rdata.rdb file of more than 5MB without 'LazyDataCompression' being set. If you see that, run CheckLazyDataCompression() and set the field – to gzip in the unlikely event[27] that is the best choice.

The analogue for sysdata.rda is field 'SysDataCompression': the default is xz for files bigger than 1MB otherwise gzip.

Lazy-loading is not supported for very large datasets (those which when serialized exceed 2GB, the limit for the format on 32-bit platforms).

1.1.7 Non-R scripts in packages

Code which needs to be compiled (C, C++, Fortran . . .) 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, BUGS, JavaScript, Matlab, Perl, PHP (**amap** (https://CRAN.R-project.org/package=amap)), Python or Tcl (**Simile** (https://CRAN.R-project.org/package=Simile)), or even R. However, it seems more common to use the inst directory, for example WriteXLS/inst/Perl, NMF/inst/m-files, RnavGraph/inst/tcl, RProtoBuf/inst/python and emdbook/inst/BUGS and gridSVG/inst/js.

Java code is a special case: except for very small programs, .java files should be byte-compiled (to a .class file) and distributed as part of a .jar file: the conventional location for the .jar file(s) is inst/java. It is desirable (and required under an Open Source license) to make the Java source files available: this is best done in a top-level java directory in the package—the source files should not be installed.

If your package requires one of these interpreters or an extension then this should be declared in the 'SystemRequirements' field of its DESCRIPTION file. (Users of Java most often do so *via* **rJava** (https://CRAN.R-project.org/package=rJava), when depending on/importing that suffices unless there is a version requirement on Java code in the package.)

Windows and Mac users should be aware that the Tcl extensions 'BWidget' and 'Tktable' (which have sometimes been included in the Windows[28] and macOS R installers) *are* extensions and do need to be declared (and that 'Tktable' is less widely available than it used to be, including not in the main repositories for major Linux distributions). 'BWidget' needs to be installed by the user on other OSes. This is fairly easy to do: first find the Tcl search path:

```
library(tcltk)
strsplit(tclvalue('auto_path'), " ")[[1]]
```

[27] For all the CRAN packages tested, either gz or bzip2 provided a very substantial reduction in installed size.

[28] 'BWidget' still is on Windows but 'Tktable' was not in R 4.0.0.

then download the sources from `https://sourceforge.net/projects/tcllib/files/BWidget/` and in a terminal run something like

```
tar xf bwidget-1.9.14.tar.gz
sudo mv bwidget-1.9.14 /usr/local/lib
```

substituting a location on the Tcl search path for `/usr/local/lib` if needed. (If no location on that search path is writeable, you will need to add one each time '`BWidget`' is to be used with `tcltk::addTclPath()`.)

To (silently) test for the presence of '`Tktable`' one can use

```
library(tcltk)
have_tktable <- !isFALSE(suppressWarnings(tclRequire('Tktable')))
```

Installing '`Tktable`' needs a C compiler and the Tk headers (not necessarily installed with Tcl/Tk). At the time of writing the latest sources (from 2008) were available from `https://sourceforge.net/projects/tktable/files/tktable/2.10/Tktable2.10.tar.gz/download`, but needed patching for current Tk (8.6.11, but not 8.6.10) – a patch can be found at `https://www.stats.ox.ac.uk/pub/bdr/Tktable/`. For a system installation of Tk you may need to install '`Tktable`' as '`root`' as on e.g. Fedora all the locations on `auto_path` are owned by '`root`'.

1.1.8 Specifying URLs

URLs in many places in the package documentation will be converted to clickable hyperlinks in at least some of their renderings. So care is needed that their forms are correct and portable.

The full URL should be given, including the scheme (often '`http://`' or '`https://`') and a final '`/`' for references to directories.

Spaces in URLs are not portable and how they are handled does vary by HTTP server and by client. There should be no space in the host part of an '`http://`' URL, and spaces in the remainder should be encoded, with each space replaced by '`%20`'.

Other characters may benefit from being encoded: see the help on `URLencode()`.

The canonical URL for a CRAN package is

`https://cran.r-project.org/package=`*`pkgname`*

and not a version starting '`https://cran.r-project.org/web/packages/`*`pkgname`*'.

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[29] 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

[29] The script should only assume a POSIX-compliant `/bin/sh` – see `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html`. In particular `bash` extensions must not be used, and not all R platforms have a `bash` command, let alone one at `/bin/bash`. All known shells used with R support the use of backticks, but not all support '`$(`*`cmd`*`)`'. However, real-world shells are not fully POSIX-compliant and omissions and idiosyncrasies need to be worked around—which Autoconf will do for you. Arithmetic expansion is a known issue: see `https://www.gnu.org/software/autoconf/manual/autoconf.html#Portable-Shell` for this and others. Some checks can be done by the `checkbashisms` Perl script at `https://sourceforge.net/projects/checkbaskisms/files`, also available in most Linux distributions in a package named either '`devscripts`' or '`devscripts-checkbashisms`': a later version can be extracted from Debian sources such as the most recent `tar.xz` in `https://deb.debian.org/debian/pool/main/d/devscripts/` and has been needed for recent versions of Perl.

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.). Background and useful tips on Autoconf and related tools (including `pkg-config` described below) can be found at `https://autotools.info/`.

A `configure` script is run in an environment which has all the environment variables set for an R session (see *R_HOME*`/etc/Renviron`) plus `R_PACKAGE_NAME` (the name of the package), `R_PACKAGE_DIR` (the path of the target installation directory of the package, a temporary location for staged installs) and `R_ARCH` (the arch-dependent part of the path, often empty).

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.

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 to `FALSE` otherwise, and then substitutes this value into output files (by replacing instances of '`@HAVE_FOO@`' in input files with the value of `HAVE_FOO`). For example, if a function named `bar` is to be made available by linking against library `foo` (i.e., using `-lfoo`), one could use

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

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

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

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

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

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

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

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

You will very likely need to ensure that the same C compiler and compiler flags are used in the `configure` tests as when compiling R or your package. Under a Unix-alike, you can achieve this by including the following fragment early in `configure.ac` (*before* calling `AC_PROG_CC` or anything which calls it)

```
: ${R_HOME=`R RHOME`}
if test -z "${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.)

If your code does load checks (for example, to check for an entry point in a library or to run code) then you will also need

```
LDFLAGS=`"${R_HOME}/bin/R" CMD config LDFLAGS`
```

Packages written with C++ need to pick up the details for the C++ compiler and switch the current language to C++ by something like

```
CXX=`"${R_HOME}/bin/R" CMD config CXX`
if test -z "$CXX"; then
  AC_MSG_ERROR([No C++ compiler is available])
fi
CXXFLAGS=`"${R_HOME}/bin/R" CMD config CXXFLAGS`
CPPFLAGS=`"${R_HOME}/bin/R" CMD config CPPFLAGS`
AC_LANG(C++)
```

The latter is important, as for example C headers may not be available to C++ programs or may not be written to avoid C++ name-mangling. Note that an R installation is not required to have a C++ compiler so '`CXX`' may be empty. If the package specifies a non-default C++ standard, use the `config` variable names (such as `CXX17`) appropriate to the standard, but still set `CXX` and `CXXFLAGS`.

You can use `R CMD config` to get the value of the basic configuration variables, and also the header and library flags necessary for linking a front-end executable program against R, see *`R CMD config --help`* for details. If you do, it is essential that you use both the command and the appropriate flags, so that for example '`CC`' must always be used with '`CFLAGS`' and (for code to be linked into a shared library) '`CPICFLAGS`'. For Fortran, be careful to use '`FC FFLAGS FPICFLAGS`' for fixed-form Fortran and '`FC FCFLAGS FPICFLAGS`' for free-form Fortran.

As from R 4.3.0, variables

```
CC CFLAGS CXX CXXFLAGS CPPFLAGS LDFLAGS FC FCFLAGS
```

are set in the environment (if not already set) when `configure` is called from `R CMD INSTALL`, in case the script forgets to set them as described above. This includes making use of the selected C standard (but not the C++ standard as that is selected at a later stage by `R CMD SHLIB`).

To check for an external BLAS library using the `AX_BLAS` macro from the official Autoconf Macro Archive[30], one can use

```
FC=`"${R_HOME}/bin/R" CMD config FC`
FCLAGS=`"${R_HOME}/bin/R" CMD config FFLAGS`
AC_PROG_FC
FLIBS=`"${R_HOME}/bin/R" CMD config FLIBS`
AX_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 code works on all R platforms.

N.B.: If the `configure` script creates files, e.g. `src/Makevars`, you do need a `cleanup` script to remove them. Otherwise `R CMD build` may ship the files that are created. For example, package **RODBC** (`https://CRAN.R-project.org/package=RODBC`) has

```
#!/bin/sh

rm -f config.* src/Makevars src/config.h
```

[30] `https://www.gnu.org/software/autoconf-archive/ax_blas.html`. If you include macros from that archive you need to arrange for them to be included in the package sources for use by `autoreconf`.

As this example shows, `configure` often creates working files such as `config.log`. If you use a hand-crafted script rather than one created by `autoconf`, it is highly recommended that you log its actions to file `config.log`.

If your configure script needs auxiliary files, it is recommended that you ship them in a `tools` directory (as R itself does).

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 (or `configure.ucrt`) to be used on that platform. (Optionally, there can be a `cleanup.win` script (or `cleanup.ucrt`). Both should be shell scripts to be executed by `ash`, which is a minimal version of Bourne-style `sh`. As from R 4.2.0, `bash` is used. When `configure.win` (or `configure.ucrt`) is run the environment variables `R_HOME` (which uses '`/`' as the file separator), `R_ARCH` and `R_ARCH_BIN` will be set. Use `R_ARCH` to decide if this is a 64-bit build for Intel (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`. If a `configure.win` script does compilation (including calling `R CMD SHLIB`), most of the considerations above apply.

As the scripts on Windows are executed as `sh ./configure.win` and similar, any 'shebang' first line (such as `#! /bin/bash`) is treated as a comment.

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. 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` (or `configure.ucrt`) 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 by default[31] on macOS) to do this is `pkg-config`. The `configure` script will need to test for the presence of the command itself[32] (see for example package **tiff** (`https://CRAN.R-project.org/package=tiff`)), 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 'libtiff-4 >= 4.1.0' --print-errors # check the status
$ pkg-config --modversion libtiff-4
4.3.0
```

[31] but it is available on the machines used to produce the CRAN binary packages: however as Apple does not ship `.pc` files for its system libraries such as `expat`, `libcurl`, `libxml2`, `sqlite3` and '`zlib`', it may well not find information on these. Some substitutes are available from `https://github.com/R-macos/recipes/tree/master/stubs/pkgconfig-darwin` and are installed on the CRAN package builders.

[32] It is not wise to check the version of `pkg-config` as it is sometimes a link to `pkgconf`, a separate project with a different version series.

```
$ pkg-config --cflags libtiff-4
-I/usr/local/include
$ pkg-config --libs libtiff-4
-L/usr/local/lib -ltiff
$ pkg-config --static --libs libtiff-4
-L/usr/local/lib -ltiff -lwebp -llzma -ljpeg -lz
```

Note that `pkg-config --libs` gives the information required to link against the default version[33] of that library (usually the dynamic one), and `pkg-config --static --libs` may be needed if the static library is to be used.

Static libraries are commonly used on macOS and Windows to facilitate bundling external software with binary distributions of packages. This means that portable (source) packages need to allow for this. It is *not* safe to just use `pkg-config --static --libs`, as that will often include further libraries that are not necessarily installed on the user's system (or maybe only the versioned library such as `libjbig.so.2.1` is installed and not `libjbig.so` which would be needed to use `-ljbig` sometimes included in `pkg-config --static --libs libtiff-4`).

Another issue is that `pkg-config --exists` may not be reliable. It checks not only that the 'module' is available but *all* of the dependencies, including those in principle needed for static linking. (XQuartz 2.8.x only distributed dynamic libraries and not some of the `.pc` files needed for `--exists`.)

Sometimes the name by which the software is known to `pkg-config` is not what one might expect (e.g. '`libxml-2.0`' even for 2.9.x). To get a complete list use

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

Some external software provides a `-config` command to do a similar job to `pkg-config`, including

```
curl-config   freetype-config  gdal-config    geos-config
gsl-config    iodbc-config     libpng-config  nc-config
pcre-config   pcre2-config     xml2-config    xslt-config
```

(`curl-config` is for `libcurl` not `curl`. `nc-config` is for `netcdf`.) Most have an option to use static libraries.

N.B. These commands indicate what header paths and libraries are needed, but they do not obviate the need to check that the recipes they give actually work. (This is especially necessary for platforms which use static linking.)

If using Autoconf it is good practice to include all the Autoconf sources in the package (and required for an Open Source package and tested by `R CMD check --as-cran`). This will include the file `configure.ac`[34] in the top-level directory of the package. If extensions written in `m4` are needed, these should be included under the directory `tools` and included from `configure.ac` *via* e.g.,

```
m4_include([tools/ax_pthread.m4])
```

Alternatively, Autoconf can be asked to search all `.m4` files in a directory by including something like[35]

```
AC_CONFIG_MACRO_DIR([tools/m4])
```

One source of such extensions is the 'Autoconf Archive' (`https://www.gnu.org/software/autoconf-archive/`. It is not safe to assume this is installed on users' machines, so the extension should be shipped with the package (taking care to comply with its licence).

[33] but not all projects get this right when only a static library is installed, so it is often necessary to try in turn `pkg-config --libs` and `pkg-config --static --libs`.

[34] a decade ago Autoconf used `configure.in`: this is still accepted but should be renamed and `autoreconf` as used by `R CMD check --as-cran` will report as such.

[35] For those using `autoconf` 2.70 or later there is also `AC_CONFIG_MACRO_DIRS` which allows multiple directories to be specified.

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`.

A `Makevars` file is a makefile and is used as one of several makefiles by `R CMD SHLIB` (which is called by `R CMD INSTALL` to compile code in the `src` directory). It should be written if at all possible in a portable style, in particular (except for `Makevars.win` and `Makevars.ucrt`) without the use of GNU extensions.

The most common use of a `Makevars` file is to set additional preprocessor options (for example include paths and definitions) for C/C++ files *via* `PKG_CPPFLAGS`, and additional compiler flags by setting `PKG_CFLAGS`, `PKG_CXXFLAGS` or `PKG_FFLAGS`, for C, C++ or Fortran respectively (see Section 5.5 [Creating shared objects], page 135).

N.B.: Include paths are preprocessor options, not compiler options, and **must** be set in `PKG_CPPFLAGS` as otherwise platform-specific paths (e.g. '`-I/usr/local/include`') will take precedence. `PKG_CPPFLAGS` should contain '`-I`', '`-D`', '`-U`' and (where supported) '`-include`' and '`-pthread`' options: everything else should be a compiler flag. The order of flags matters, and using '`-I`' in `PKG_CFLAGS` or `PKG_CXXFLAGS` has led to hard-to-debug platform-specific errors.

`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` (and all other `-W` flags: for the Oracle compilers these were used to pass arguments to compiler phases) are all specific to GCC (and compilers such as `clang` which aim to be options-compatible with it).

Also, do not set variables such as `CPPFLAGS`, `CFLAGS` etc.: these should be settable by users (sites) through appropriate personal (site-wide) `Makevars` files. See Section "Customizing package compilation" in *R Installation and Administration*,

There are some macros[36] which are set whilst configuring the building of R itself and are stored in *`R_HOME`*`/etc`*`R_ARCH`*`/Makeconf`. That makefile is included as a `Makefile` *after* `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 in the `src` directory.

`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 nor complex routines. If `BLAS_LIBS` is included, then `FLIBS` also needs to be[37] included following it, as most BLAS libraries are written at least partially in Fortran. However, it can be omitted if the package contains Fortran source code as that will add `FLIBS` to the link line.

`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 the main double-precision LAPACK routines as well as those double-complex LAPACK routines needed to build R, or it may point

[36] in POSIX parlance: GNU `make` calls these 'make variables'.

[37] at least on Unix-alikes: the Windows build currently resolves such dependencies to a static Fortran library when `Rblas.dll` is built.

to an external LAPACK library, or may be empty if an external BLAS library also contains LAPACK.

[`libRlapack` includes all the double-precision LAPACK routines which were current in 2003 and a few more recent ones: a list of which routines are included is in file `src/modules/lapack/README`. Note that an external LAPACK/BLAS library need not do so, as some were 'deprecated' (and not compiled by default) in LAPACK 3.6.0 in late 2015.]

For portability, the macros `BLAS_LIBS` and `FLIBS` should always be included *after* `LAPACK_LIBS` (and in that order).

`SAFE_FFLAGS`

A macro containing flags which are needed to circumvent over-optimization of FORTRAN code: it is might be '`-g -O2 -ffloat-store`' or '`-g -O2 -msse2 -mfpmath=sse`' 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`. 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

```
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** (`https://CRAN.R-project.org/package=fastICA`) used to have

```
PKG_LIBS = @BLAS_LIBS@

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

all: $(SHLIB)

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

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

```
all: $(SHLIB)

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

(since the other macros are all empty on that platform, and R's internal BLAS was 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
```

```
mylibs:
        (cd subdir; $(MAKE))
```

Be careful to create all the necessary dependencies, as there is 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). In particular,

```
all: mylibs
```

does **not** suffice. GNU make does allow the construct

```
.NOTPARALLEL: all
all: mylibs $(SHLIB)
```

but that is not portable. `dmake` and `pmake` allow the similar `.NO_PARALLEL`, also not portable: some variants of `pmake` accept `.NOTPARALLEL` as an alias for `.NO_PARALLEL`.

Note that on Windows it is required that `Makevars[.win, .ucrt]` 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` or `Makefile.ucrt` file.

It is sometimes useful to have a target '`clean`' in `Makevars`, `Makevars.win` or `Makevars.ucrt`: 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`.

Avoid the use of default (also known as 'implicit' rules) in makefiles, as these are `make`-specific. Even when mandated by POSIX – GNU `make` does not comply and this has broken package installation.

An unfortunately common error is to have

```
all: $(SHLIB) clean
```

which asks `make` to clean in parallel with compiling the code. Not only does this lead to hard-to-debug installation errors, it wipes out all the evidence of any error (from a parallel make or not). It is much better to leave cleaning to the end user using the facilities in the previous paragraph.

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 Intel 64-bit code or code for other architectures, for example (GNU `make` syntax, allowed on Windows)

```
ifeq "$(WIN)" "64"
PKG_LIBS = value for 64-bit Intel Windows
else
PKG_LIBS = value for unknown Windows architectures
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 `https://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 was a `libxml2.dll` whereas on one build for 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` or `configure.ucrt`) to ship them in the same directory as the package DLL.

1.2.1.1 OpenMP support

There is some support for packages which wish to use OpenMP[38]. The `make` macros

```
SHLIB_OPENMP_CFLAGS
SHLIB_OPENMP_CXXFLAGS
SHLIB_OPENMP_FFLAGS
```

are available for use in `src/Makevars`, `src/Makevars.win` or `Makevars.ucrt`. Include the appropriate macro in `PKG_CFLAGS`, `PKG_CXXFLAGS` and so on, and also in `PKG_LIBS` (but see below for Fortran). C/C++ code that needs to be conditioned on the use of OpenMP can be used inside `#ifdef _OPENMP`: note that some toolchains used for R (including Apple's for macOS[39] and some others using `clang`[40]) have no OpenMP support at all, not even `omp.h`.

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

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

Note that the macro `SHLIB_OPENMP_CXXFLAGS` applies to the default C++ compiler and not necessarily to the C++17/20/23 compiler: users of the latter should do their own `configure` checks. If you do use your own checks, make sure that OpenMP support is complete by compiling and linking an OpenMP-using program: on some platforms the runtime library is optional and on others that library depends on other optional libraries.

Some care is needed when compilers are from different families which may use different OpenMP runtimes (e.g. `clang` *vs* GCC including `gfortran`, although it is often possible to use the `clang` runtime with GCC but not *vice versa*: however `gfortran` >= 9 may generate calls

[38] `https://www.openmp.org/`, `https://en.wikipedia.org/wiki/OpenMP`, `https://hpc-tutorials.llnl.gov/openmp/`

[39] There are somewhat fragile workarounds: see `https://mac.r-project.org/openmp/`.

[40] Default builds of LLVM `clang` 3.8.0 and later have support for OpenMP, but the `libomp` run-time library may not be installed.

not in the `clang` runtime). For a package with Fortran code using OpenMP the appropriate lines are

```
PKG_FFLAGS = $(SHLIB_OPENMP_FFLAGS)
PKG_LIBS = $(SHLIB_OPENMP_CFLAGS)
```

as the C compiler will be used to link the package code. There are platforms on which this does not work *for some OpenMP-using code* and installation will fail. Since R >= 3.6.2 the best alternative for a package with only Fortran sources using OpenMP is to use

```
USE_FC_TO_LINK =
PKG_FFLAGS = $(SHLIB_OPENMP_FFLAGS)
PKG_LIBS = $(SHLIB_OPENMP_FFLAGS)
```

in `src/Makevars`, `src/Makevars.win` or `Makevars.ucrt`. Note however, that when this is used `$(FLIBS)` should not be included in `PKG_LIBS` since it is for linking Fortran-compiled code by the C compiler.

Common platforms may inline all OpenMP calls and so tolerate the omission of the OpenMP flag from `PKG_LIBS`, but this usually results in an installation failure with a different compiler or compilation flags. So cross-check that e.g. `-fopenmp` appears in the linking line in the installation logs.

It is not portable to use OpenMP with more than one of C, C++ and Fortran in a single package since it is not uncommon that the compilers are of different families.

For portability, any C/C++ code using the `omp_*` functions should include the `omp.h` header: some compilers (but not all) include it when OpenMP mode is switched on (e.g. *via* flag `-fopenmp`).

There is nothing[41] to say what version of OpenMP is supported: version 4.0 (and much of 4.5 or 5.0) is supported by recent versions of the Linux and Windows platforms, but portable packages cannot assume that end users have recent versions. Apple `clang` on macOS has no OpenMP support. `https://www.openmp.org/resources/openmp-compilers-tools/` gives some idea of what compilers support what versions. Note that support for Fortran compilers is often less up-to-date and that page suggests it is unwise to rely on a version later than 3.1. Which introduced a Fortran OpenMP module, so Fortran users of OpenMP should include

```
use omp_lib
```

Rarely, using OpenMP with `clang` on Linux generates calls in `libatomic`, resulting in loading messages like

```
undefined symbol: __atomic_compare_exchange
undefined symbol: __atomic_load
```

The workaround is to link with `-latomic` (having checked it exists).

The performance of OpenMP varies substantially between platforms. The Windows implementation has substantial overheads, so is only beneficial if quite substantial tasks are run in parallel. Also, on Windows new threads are started with the default[42] FPU control word, so computations done on OpenMP threads will not make use of extended-precision arithmetic which is the default for the main process.

Do not include these macros unless your code does make use of OpenMP (possibly for C++ via included external headers): this can result in the OpenMP runtime being linked in, threads being started, ...

Calling any of the R API from threaded code is 'for experts only' and strongly discouraged. Many functions in the R API modify internal R data structures and might corrupt these data

[41] In most implementations the `_OPENMP` macro has value a date which can be mapped to an OpenMP version: for example, value `201307` is the date of version 4.0 (July 2013). However this may be used to denote the latest version which is partially supported, not that which is fully implemented.

[42] Windows default, not MinGW-w64 default.

structures if called simultaneously from multiple threads. Most R API functions can signal errors, which must only happen on the R main thread. Also, external libraries (e.g. LAPACK) may not be thread-safe.

Packages are not standard-alone programs, and an R process could contain more than one OpenMP-enabled package as well as other components (for example, an optimized BLAS) making use of OpenMP. So careful consideration needs to be given to resource usage. OpenMP works with parallel regions, and for most implementations the default is to use as many threads as 'CPUs' for such regions. Parallel regions can be nested, although it is common to use only a single thread below the first level. The correctness of the detected number of 'CPUs' and the assumption that the R process is entitled to use them all are both dubious assumptions. One way to limit resources is to limit the overall number of threads available to OpenMP in the R process: this can be done *via* environment variable `OMP_THREAD_LIMIT`, where implemented.[43] Alternatively, the number of threads per region can be limited by the environment variable `OMP_NUM_THREADS` or API call `omp_set_num_threads`, or, better, for the regions in your code as part of their specification. E.g. R uses[44]

```
#pragma omp parallel for num_threads(nthreads) ...
```

That way you only control your own code and not that of other OpenMP users.

Note that setting environment variables to control OpenMP is implementation-dependent and may need to be done outside the R process or before any use of OpenMP (which might be by another process or R itself). Also, implementation-specific variables such as `KMP_THREAD_LIMIT` might take precedence.

1.2.1.2 Using pthreads

There is no direct support for the POSIX threads (more commonly known as `pthreads`): by the time we considered adding it several packages were using it unconditionally so it seems that nowadays it is universally available on POSIX operating systems.

For reasonably recent versions of `gcc` and `clang` the correct specification is

```
PKG_CPPFLAGS = -pthread
PKG_LIBS = -pthread
```

(and the plural version is also accepted on some systems/versions). For other platforms the specification is

```
PKG_CPPFLAGS = -D_REENTRANT
PKG_LIBS = -lpthread
```

(and note that the library name is singular). This is what `-pthread` does on all known current platforms (although earlier versions of OpenBSD used a different library name).

For a tutorial see `https://hpc-tutorials.llnl.gov/posix/`.

POSIX threads are not normally used on Windows which has its own native concepts of threads: however, recent toolchains do provide the `pthreads` header and library.

The presence of a working `pthreads` implementation cannot be unambiguously determined without testing for yourself: however, that '`_REENTRANT`' is defined in C/C++ code is a good indication.

Note that not all `pthreads` implementations are equivalent as parts are optional (see `https://pubs.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html`): for example, macOS lacks the 'Barriers' option.

[43] Which it was at the time of writing with GCC, Intel and Clang compilers. The count may include the thread running the main process.

[44] Be careful not to declare `nthreads` as `const int`: the Oracle compiler required it to be 'an lvalue'.

See also the comments on thread-safety and performance under OpenMP: on all known R platforms OpenMP is implemented *via* `pthreads` and the known performance issues are in the latter.

1.2.1.3 Compiling in sub-directories

Package authors fairly often want to organize code in sub-directories of `src`, for example if they are including a separate piece of external software to which this is an R interface.

One simple way is simply to set `OBJECTS` to be all the objects that need to be compiled, including in sub-directories. For example, CRAN package **RSiena** (`https://CRAN.R-project.org/package=RSiena`) has

```
SOURCES = $(wildcard data/*.cpp network/*.cpp utils/*.cpp model/*.cpp model/*/*.cpp model/*/*/*.cpp)

OBJECTS = siena07utilities.o siena07internals.o siena07setup.o siena07models.o $(SOURCES:.cpp=.o)
```

One problem with that approach is that unless GNU make extensions are used, the source files need to be listed and kept up-to-date. As in the following from CRAN package **lossDev** (`https://CRAN.R-project.org/package=lossDev`):

```
OBJECTS.samplers = samplers/ExpandableArray.o samplers/Knots.o \
  samplers/RJumpSpline.o samplers/RJumpSplineFactory.o \
  samplers/RealSlicerOV.o samplers/SliceFactoryOV.o samplers/MNorm.o
OBJECTS.distributions = distributions/DSpline.o \
  distributions/DChisqrOV.o distributions/DTOV.o \
  distributions/DNormOV.o distributions/DUnifOV.o distributions/RScalarDist.o
OBJECTS.root = RJump.o

OBJECTS = $(OBJECTS.samplers) $(OBJECTS.distributions) $(OBJECTS.root)
```

Where the subdirectory is self-contained code with a suitable makefile, the best approach is something like

```
PKG_LIBS = -LCsdp/lib -lsdp $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS)

$(SHLIB): Csdp/lib/libsdp.a

Csdp/lib/libsdp.a:
        @(cd Csdp/lib && $(MAKE) libsdp.a \
          CC="$(CC)" CFLAGS="$(CFLAGS) $(CPICFLAGS)" AR="$(AR)" RANLIB="$(RANLIB)")
```

Note the quotes: the macros can contain spaces, e.g. `CC = "gcc -m64 -std=gnu99"`. Several authors have forgotten about parallel makes: the static library in the subdirectory must be made before the shared object (`$(SHLIB)`) and so the latter must depend on the former. Others forget the need[45] for position-independent code.

We really do not recommend using `src/Makefile` instead of `src/Makevars`, and as the example above shows, it is not necessary.

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** (`https://CRAN.R-project.org/package=RODBC`) package.

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

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

dnl A user-specifiable option
odbc_mgr=""
AC_ARG_WITH([odbc-manager],
```

[45] A few OSes (AIX, Windows) do not need special flags for such code, but most do—although compilers will often generate PIC code when not asked to do so.

```
            AC_HELP_STRING([--with-odbc-manager=MGR],
                           [specify the ODBC manager, e.g. odbc or iodbc]),
            [odbc_mgr=$withval])

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

dnl Select an optional include path, from a configure option
dnl or from an environment variable.
AC_ARG_WITH([odbc-include],
            AC_HELP_STRING([--with-odbc-include=INCLUDE_PATH],
                           [the location of ODBC header files]),
            [odbc_include_path=$withval])
RODBC_CPPFLAGS="-I."
if test [ -n "$odbc_include_path" ] ; then
   RODBC_CPPFLAGS="-I. -I${odbc_include_path}"
else
  if test [ -n "${ODBC_INCLUDE}" ] ; then
     RODBC_CPPFLAGS="-I. -I${ODBC_INCLUDE}"
  fi
fi

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

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

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

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

dnl search for a library containing an ODBC function
if test [ -n "${odbc_mgr}" ] ; then
```

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

dnl for 64-bit ODBC need SQL[U]LEN, and it is unclear where they are defined.
AC_CHECK_TYPES([SQLLEN, SQLULEN], , , [# include <sql.h>])
dnl for unixODBC header
AC_CHECK_SIZEOF(long, 4)

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

where `src/Makevars.in` would be simply

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

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

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

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

1.2.3 Using modern Fortran code

R assumes that source files with extension `.f` are fixed-form Fortran 90 (which includes Fortran 77), and passes them to the compiler specified by macro '`FC`'. The Fortran compiler will also accept free-form Fortran 90/95 code with extension `.f90` or (most[46]) `.f95`.

The same compiler is used for both fixed-form and free-form Fortran code (with different file extensions and possibly different flags). Macro `PKG_FFLAGS` can be used for package-specific flags: for the un-encountered case that both are included in a single package and that different flags are needed for the two forms, macro `PKG_FCFLAGS` is also available for free-form Fortran.

The code used to build R allows a 'Fortran 90' compiler to be selected as '`FC`', so platforms might be encountered which only support Fortran 90. However, Fortran 95 is supported on all known platforms.

Most compilers specified by '`FC`' will accept Fortran 2003, 2008 or 2018 code: such code should still use file extension `.f90`. Most current platforms use `gfortran` where you might need to include `-std=f2003`, `-std=f2008` or (from version 8) `-std=f2018` in `PKG_FFLAGS` or `PKG_FCFLAGS`: the default is 'GNU Fortran', currently Fortran 2018 (but Fortran 95 prior to `gfortran` 8) with non-standard extensions. The other compilers in current use (LLVM's `flang-new` and Intel's `ifx`) default to Fortran 2018.

It is good practice to describe a Fortran version requirement in `DESCRIPTION`'s '`SystemRequirements`' field.

Modern versions of Fortran support modules, whereby compiling one source file creates a module file which is then included in others. (Module files typically have a `.mod` extension: they

[46] Intel compilers do not by default but this is worked around when using packages without a `src/Makefile`.

do depend on the compiler used and so should never be included in a package.) This creates a dependence which `make` will not know about and often causes installation with a parallel make to fail. Thus it is necessary to add explicit dependencies to `src/Makevars` to tell `make` the constraints on the order of compilation. For example, if file `iface.f90` creates a module '`iface`' used by files `cmi.f90` and `dmi.f90` then `src/Makevars` needs to contain something like

```
cmi.o dmi.o: iface.o
```

Note that it is not portable (although some platforms do accept it) to define a module of the same name in multiple source files.

1.2.4 Using C++ code

R can be built without a C++ compiler although one is available (but not necessarily installed) on all known R platforms. As from R 4.0.0 a C++ compiler will be selected only if it conforms to the 2011 standard ('C++11'). A minor update[47] ('C++14') was published in December 2014 and was used by default as from R 4.1.0 if supported. Further revisions 'C++17' (in December 2017) and 'C++20' (with many new features in December 2020) have been published since. The next revision, 'C++23', is expected in 2024 and several compilers already have extensive partial support for the final draft.

The default standard for compiling R packages was changed to C++17 in R 4.3.0 if supported, and from R 4.4.0 only a C++17 compiler will be selected as the default C++ compiler.

What standard a C++ compiler aims to support can be hard to determine: the value[48] of `__cplusplus` may help but some compilers use it to denote a standard which is partially supported and some the latest standard which is (almost) fully supported. On a Unix-alike `configure` will try to identify a compiler and flags for each of the standards: this relies heavily on the reported values of `__cplusplus`.

The webpage `https://en.cppreference.com/w/cpp/compiler_support` gives some information on which compilers are known to support recent C++ features.

C++ standards have deprecated and later removed features. Be aware that some current compilers still accept removed features in C++17 mode, such as `std::unary_function` (deprecated in C++11, removed in C++17).

Different versions of R have used different default C++ standards, so for maximal portability a package should specify the standard it requires. In order to specify C++14 code in a package with a `Makevars` file (or `Makevars.win` or `Makevars.ucrt` on Windows) should include the line

```
CXX_STD = CXX14
```

Compilation and linking will then be done with the C++14 compiler (if any). Analogously for other standards (details below). On the other hand, specifying C++11[49] when the code is valid under C++14 or C++17 reduces future portability.

Packages without a `src/Makevars` or `src/Makefile` file may specify a C++ standard for code in the `src` directory by including something like '`C++14`' in the '`SystemRequirements`' field of the `DESCRIPTION` file, e.g.

```
SystemRequirements: C++14
```

If a package does have a `src/Makevars[.win]` file then also setting the make variable '`CXX_STD`' there is recommended, as it allows `R CMD SHLIB` to work correctly in the package's `src` directory.

---

[47] Some changes are linked from `https://isocpp.org/std/standing-documents/sd-6-sg10-feature-test-recommendations`: there were also additional deprecations.

[48] Values `201103L`, `201402L`, `201703L` and `202002L` are most commonly used for C++11, C++14, C++17 and C++20 respectively, but some compilers set `1L`. For C++23 all that can currently be assumed is a value greater than that for C++20: for example `g++` 12 uses `202100L` and `clang++` (LLVM 15, Apple 14) uses `202101L`.

[49] Often historically used to mean 'not C++98'

A requirement of C++17 or later should always be declared in the 'SystemRequirements' field (as well as in src/Makevars or src/Makefile) so this is shown on the package's summary pages on CRAN or similar. This is also good practice for a requirement of C++14. Note that support of C++14 or C++17 is only available from R 3.4.0, so if the package has an R version requirement it needs to take that into account.

Essentially complete C++14 support is available from GCC 5, LLVM clang 3.4 and currently-used versions of Apple clang.

Code needing C++14 features can check for their presence *via* 'SD-6 feature tests'[50]. Such a check could be

```
#include <memory> // header where this is defined
#if defined(__cpp_lib_make_unique) && (__cpp_lib_make_unique >= 201304)
using std::make_unique;
#else
// your emulation
#endif
```

C++17 (as from R 3.4.0), C++20 (as from R 4.0.0) and C++23 (as from R 4.3.0) can be specified in an analogous way (replacing 14 by 17, 20 or 23) but compiler/OS support is platform-dependent. Some C++17 and C++20 support is available with the default builds of R on macOS and Windows as from R 4.0.0. Much of g++'s support for C++17 needs version 7 or later: that is more recent than some still-current Linux distributions but often packages for later compilers are available: for RHEL/Centos 7 look for 'devtoolset'.

Note that C++17 or later 'support' does not mean complete support: use feature tests as well as resources such as https://en.cppreference.com/w/cpp/compiler_support, https://gcc.gnu.org/projects/cxx-status.html and https://clang.llvm.org/cxx_status.html to see if the features you want to use are widely implemented. In particular, for C++23 R's configure script only checks for a compiler claiming to be later than C++20.[51]

Attempts to specify an unknown C++ standard are silently ignored: recent versions of R throw an error for C++98 and for known standards for which no compiler+flags has been detected.

If a package using C++ has a configure script it is essential that the script selects the correct C++ compiler and standard, *via* something like

```
CXX17=`"${R_HOME}/bin/R" CMD config CXX17`
if test -z "$CXX17"; then
  AC_MSG_ERROR([No C++17 compiler is available])
fi
CXX17STD=`"${R_HOME}/bin/R" CMD config CXX17STD`
CXX="${CXX17} ${CXX17STD}"
CXXFLAGS=`"${R_HOME}/bin/R" CMD config CXX17FLAGS`
## for an configure.ac file
AC_LANG(C++)
```

if C++17 was specified, but using

```
CXX=`"${R_HOME}/bin/R" CMD config CXX`
CXXFLAGS=`"${R_HOME}/bin/R" CMD config CXXFLAGS`
## for an configure.ac file
AC_LANG(C++)
```

[50] See https://isocpp.org/std/standing-documents/sd-6-sg10-feature-test-recommendations or https://en.cppreference.com/w/cpp/experimental/feature_test. It seems a reasonable assumption that any compiler promising some C++14 conformance will provide these—e.g. g++ 4.9.x did but 4.8.5 did not.

[51] The __cplusplus value for C++23 is not yet finalized and all but some of the latest compilers use values from earlier drafts.

if no standard was specified.

If you want to compile C++ code in a subdirectory, make sure you pass down the macros to specify the appropriate compiler, e.g. in `src/Makevars`

```
sublibs:
        @(cd libs && $(MAKE) \
           CXX="$(CXX17) $(CXX17STD)" CXXFLAGS="$(CXX17FLAGS) $(CXX17PICFLAGS)")
```

The discussion above is about the standard R ways of compiling C++: it will not apply to packages using `src/Makefile` or building in a subdirectory that do not set the C++ standard. And compilers' default C++ standards varies widely and gets changed frequently by vendors – for example Apple clang up to at least 15 defaults to C++98, LLVM clang 14–15 to C++14, LLVM clang 16–18 to C++17 and `g++` 11–13 to C++17.

For a package with a `src/Makefile` (or a Windows analogue), a non-default C++ compiler can be selected by including something like

```
CXX14 = `"${R_HOME}/bin/R" CMD config CXX14`
CXX14STD = `"${R_HOME}/bin/R" CMD config CXX14STD`
CXX = ${CXX14} ${CXX14STD}
CXXFLAGS = `"${R_HOME}/bin/R" CMD config CXX14FLAGS`
CXXPICFLAGS = `"${R_HOME}/bin/R" CMD config CXX14PICFLAGS`
SHLIB_LD = "${R_HOME}/bin/R" CMD config SHLIB_CXX14LD`
SHLIB_LDFLAGS = "${R_HOME}/bin/R" CMD config SHLIB_CXX14LDFLAGS`
```

and ensuring these values are used in relevant compilations, after checking they are non-empty. A common use of `src/Makefile` is to compile an executable, when likely something like (for example for C++14)

```
CXX14 = `"${R_HOME}/bin/R" CMD config CXX14`
CXX14STD = `"${R_HOME}/bin/R" CMD config CXX14STD`
CXX = ${CXX14} ${CXX14STD}
CXXFLAGS = `"${R_HOME}/bin/R" CMD config CXX14FLAGS`
```

suffices. On Unix (and on Windows from R 4.3.0) this can be simplified to

```
CXX = ${CXX14} ${CXX14STD}
CXXFLAGS = ${CXX14FLAGS}
```

On a Unix-alike C++ compilation defaulted to C++11 from R 3.6.0, to C++14 from R 4.1.0 and to C++17 from R 4.3.0. However, only 'if available', so platforms using very old OSes might have used the previous default. Even older versions of R defaulted to the compiler's default, almost certainly C++98 for compilers of comparable vintage.

On Windows the default was changed from C++98 to C++11 in R 3.6.2, to C++14 in R 4.2.3 and to C++17 in R 4.3.0.

The C++11 standard could be specified as from R 3.1.0 and C++14 or C++17 as from R 3.4.0, for C++20 from R 4.0.0 and for C++23 from R 4.3.0 (although they may not be supported by the compilers in use). C++11 support became mandatory in R 4.0.0 and C++17 support in R 4.4.0.

The `.so`/`.dll` in a package may need to be linked by the C++ compiler if it or any library it links to contains compiled C++ code. Dynamic linking usually brings in the C++ runtime library (commonly `libstdc++` but can be, for example, `libc++`) but static linking (as used for external libraries on Windows and macOS) will not. `R CMD INSTALL` will link with the C++ compiler if there are any top-level C++ files in `src`, but not if these are all in subdirectories. The simplest way to force linking by the C++ compiler is to include an empty C++ file in `src`..

1.2.5 C standards

C has had standards C89/C90, C99, C11, C17 (also known as C18), and C23 is in final draft and expected to be published in early 2024. C11 was a minor change to C99 which introduced some

new features and made others optional, and C17 is a 'bug-fix' update to C11. On the other hand, C23 makes extensive changes, including making `bool`, `true` and `false` reserved words, finally disallowing K&R-style function declarations and clarifying the formerly deprecated meaning of function declarations with an empty parameter list to mean zero parameters. (There are many other additions: see for example `https://en.cppreference.com/w/c/23`.)

The `configure` script in recent versions of R aims to choose a C compiler which supports C11: as the default in recent versions of `gcc`, LLVM `clang` and Apple `clang` is C17, that is what is likely to be chosen. On the other hand, until R 4.3.0 the makefiles for the Windows build specified C99. They now use the compiler default which for the recommended compiler is C17.

Packages may want to either avoid or embrace the changes in C23, and can do so *via* specifying '`USE_Cnn`' for 17, 23, 90 or 99 in the '`SystemRequirements`' field of their `DESCRIPTION` file of a package depending on '`R (>= 4.3.0)`'. Those using a `configure` script should set the corresponding compiler and flags, for example using

```
CC=`"${R_HOME}/bin/R" CMD config CC23`
CFLAGS=`"${R_HOME}/bin/R" CMD config C23FLAGS`
CPPFLAGS=`"${R_HOME}/bin/R" CMD config CPPFLAGS`
LDFLAGS=`"${R_HOME}/bin/R" CMD config LDFLAGS`
```

The (claimed) C standard in use can be checked by the macro `__STDC_VERSION__`. This is undefined in C89/C90 and should have values `199901L`, `201112L` and `201710L` for C99, C11 and C17. As C23 is not yet published there is as yet no definitive value: compilers are currently using `202000L`. C23 has macros similar to C++ 'feature tests' for many of its changes, for example `__STDC_VERSION_LIMITS_H__`.

However, note the 'claimed' as no compiler had 100% conformance, and it is better to use `configure` to test for the feature you want to use than to condition on the value of `__STDC_VERSION__`. In particular, C11 alignment functionality such as `_Alignas` and `aligned_alloc` is not implemented on Windows.

End users can specify a standard by something like `R CMD INSTALL --use-C17`. This overrides the '`SystemRequirements`' field, but not for any `configure` file.

1.2.6 Using `cmake`

Packages often wish to include the sources of other software and compile that for inclusion in their `.so` or `.dll`, which is normally done by including (or unpacking) the sources in a subdirectory of `src`, as considered above.

Further issues arise when the external software uses another build system such as `cmake`, principally to ensure that *all* the settings for compilers, include and load paths *etc* are made. This section has already mentioned the need to set at least some of

```
CC CFLAGS CXX CXXFLAGS CPPFLAGS LDFLAGS
```

`CFLAGS` and `CXXFLAGS` will need to include `CPICFLAGS` and `CXXPICFLAGS` respectively unless (as below) `cmake` is asked to generate PIC code.

Setting these (and more) as environment variables controls the behaviour of `cmake` (`https://cmake.org/cmake/help/latest/manual/cmake-env-variables.7.html#manual:cmake-env-variables(7)`), but it may be desirable to translate these into native settings such as

```
CMAKE_C_COMPILER
CMAKE_C_FLAGS
CMAKE_CXX_COMPILER
CMAKE_CXX_FLAGS
CMAKE_INCLUDE_PATH
```

```
CMAKE_LIBRARY_PATH
CMAKE_SHARED_LINKER_FLAGS_INIT
CMAKE_OSX_DEPLOYMENT_TARGET
```

and it is often necessary to ensure a static library of PIC code is built by

```
-DBUILD_SHARED_LIBS:bool=OFF
-DCMAKE_POSITION_INDEPENDENT_CODE:bool=ON
```

If R is to be detected or used, this must be the build being used for package installation – `"${R_HOME}"/bin/R`.

To fix ideas, consider a package with sources for a library `myLib` under `src/libs`. Two approaches have been used. It is often most convenient to build the external software in a directory other than its sources (particularly during development when the build directory can be removed between builds rather than attempting to clean the sources) – this is illustrated in the first approach.

1. Use the package's `configure` script to create a static library `src/build/libmyLib.a`. This can then be treated in the same way as external software, for example having in `src/Makevars`

   ```
   PKG_CPPFLAGS = -Ilibs/include
   PKG_LIBS = build/libmyLib.a
   ```

   (`-Lbuild -lmyLib` could also be used but this explicit specification avoids any confusion with dynamic libraries of the same name.)

   The `configure` script will need to contain something like (for C code)

   ```
   : ${R_HOME=`R RHOME`}
   if test -z "${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=`"${R_HOME}/bin/R" CMD config CPPFLAGS`
   LDFLAGS=`"${R_HOME}/bin/R" CMD config LDFLAGS`

   cd src
   mkdir build && cd build
   cmake -S ../libs \
     -DCMAKE_BUILD_TYPE=Release \
     -DBUILD_SHARED_LIBS:bool=OFF \
     -DCMAKE_POSITION_INDEPENDENT_CODE:bool=ON
   ${MAKE}
   ```

2. Use `src/Makevars` (or `src/Makevars.win` or `Makevars.ucrt`) to build within the subdirectory. This could be something like (for C code)

   ```
   PKG_CPPFLAGS = -Ilibs/include
   PKG_LIBS = libs/libmyLib.a

   $(SHLIB): mylibs

   mylibs:
   	(cd libs; \
   	  CC="$(CC)" CFLAGS="$(CFLAGS)" \
   	  CPPFLAGS="$(CPPFLAGS)" LDFLAGS="$(LDFLAGS)" \
   ```

```
cmake . \
  -DCMAKE_BUILD_TYPE=Release \
  -DBUILD_SHARED_LIBS:bool=OFF \
  -DCMAKE_POSITION_INDEPENDENT_CODE:bool=ON; \
$(MAKE))
```

the compiler and other settings having been set as Make variables by an R makefile included by `INSTALL` before `src/Makevars`.

A complication is that on macOS `cmake` (where installed) is commonly not on the path but at `/Applications/CMake.app/Contents/bin/cmake`. One way to work around this is for the package's `configure` script to include

```
if test -z "$CMAKE"; then CMAKE="`which cmake`"; fi
if test -z "$CMAKE"; then CMAKE=/Applications/CMake.app/Contents/bin/cmake; fi
if test -f "$CMAKE"; then echo "no 'cmake' command found"; exit 1; fi
```

and for the second approach to substitute `CMAKE` into `src/Makevars`. This also applies to the ancillary command `ctest`, if used.

1.3 Checking and building packages

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

If your package specifies an encoding in its `DESCRIPTION` file, you should run these tools in a locale which makes use of that encoding: they may not work at all or may work incorrectly in other locales (although UTF-8 locales will most likely work).

Note: `R CMD check` and `R CMD build` run R processes with `--vanilla` in which 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 the check and build environment files (as specified by the environment variables `R_CHECK_ENVIRON` and `R_BUILD_ENVIRON`; if unset, files[52] `~/.R/check.Renviron` and `~/.R/build.Renviron` are used) to set environment variables when using these utilities.

Note to Windows users: `R CMD build` may make use of the Windows toolset (see the "R Installation and Administration" manual) if present and in your path, and it is required for packages which need it to install (including those with `configure.win`, `cleanup.win`, `configure.ucrt` or `cleanup.ucrt` scripts or a `src` directory) and e.g. need vignettes built.

You may need to set the environment variable `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).

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 compressed package `tar` archives with extension `.tar.gz`, `.tgz`, `.tar.bz2` or `.tar.xz`.

It is strongly recommended that the final checks are run on a `tar` archive prepared by `R CMD build`.

[52] On systems which use sub-architectures, architecture-specific versions such as `~/.R/check.Renviron.x64` take precedence.

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[53]. (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` in the environment 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 ('`ctest`', '`eda`', '`lqs`', '`mle`', '`modreg`', '`mva`', '`nls`', '`stepfun`' and '`ts`'). Another check is that all packages mentioned in `library` or `require`s or from which the `NAMESPACE` file imports or are called *via* `::` or `:::` are listed (in '`Depends`', '`Imports`', '`Suggests`'): 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 '`.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. It is checked that the namespace can be loaded in an empty session with only the **base** namespace loaded. (Namespaces and packages can be loaded very early in the session, before the default packages are available, so packages should work then.)
10. The R files are checked for correct calls to `library.dynam`. Package startup functions are checked for correct argument lists and (incorrect) calls to functions which modify the search path or inappropriately generate messages. The R code is checked for possible problems using **codetools** (`https://CRAN.R-project.org/package=codetools`). In addition, it is checked whether S3 methods have all the 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 namespace of the package. Any other calls are reported. (The check is generous, and users may want to

[53] A suitable `file.exe` is part of the Windows toolset: it checks for `gfile` if a suitable `file` is not found: the latter is available in the OpenCSW collection for Solaris at `https://www.opencsw.org/`. The source repository is `http://ftp.astron.com/pub/file/`.

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 fields (`\name`, `\alias`, `\title` and `\description`). 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. The `data` directory is checked for non-ASCII characters and for the use of reasonable levels of compression.
16. C, C++ and Fortran source and header files[54] are tested for portable (LF-only) line endings. If there is a `Makefile` or `Makefile.in` or `Makevars` or `Makevars.in` file under the `src` directory, it is checked for portable line endings and the correct use of '`$(BLAS_LIBS)`' and '`$(LAPACK_LIBS)`'

    Compiled code is checked for symbols corresponding to functions which might terminate R or write to `stdout`/`stderr` instead of the console. Note that the latter might give false positives in that the symbols might be pulled in with external libraries and could never be called. Windows[55] users should note that the Fortran and C++ runtime libraries are examples of such external libraries.
17. Some checks are made of the contents of the `inst/doc` directory. These always include checking for files that look like leftovers, and if suitable tools (such as `qpdf`) are available, checking that the PDF documentation is of minimal size.
18. The examples provided by the package's documentation are run. (see Chapter 2 [Writing R documentation files], page 81, 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*.
19. If the package sources contain a `tests` directory then the tests specified in that directory are run. (Typically they will consist of a set of `.R` source files and target output files `.Rout.save`.) 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. (The command line option `--test-dir=foo` may be used to specify tests in a non-standard location. For example, unusually slow tests could be placed in `inst/slowTests` and then `R CMD check --test-dir=inst/slowTests` would be used to run them. Other names that have been suggested are, for example, `inst/testWithOracle` for tests that require Oracle to be installed, `inst/randomTests` for tests which use random values and may occasionally fail by chance, etc.)
20. The R code in package vignettes (see Section 1.4 [Writing package vignettes], page 45) is executed, and the vignettes re-made from their sources as a check of completeness of the sources (unless there is a '`BuildVignettes`' field in the package's `DESCRIPTION` file with a

[54] An exception is made for subdirectories with names starting '`win`' or '`Win`'.

[55] on most other platforms such runtime libraries are dynamic, but static libraries are currently used on Windows because the toolchain is not a standard part of the OS.

false value). If there is a target output file `.Rout.save` in the vignette source directory, the output from running the code in that vignette is compared with the target output file and any differences are reported (but not recorded in the log file). (If the vignette sources are in the deprecated location `inst/doc`, do mark such target output files to not be installed in `.Rinstignore`.)

If there is an error[56] in executing the R code in vignette *`foo.ext`*, a log file *`foo.ext`*`.log` is created 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/vignettes`. (It is only retained if there are errors or if environment variable `_R_CHECK_CLEAN_VIGN_TEST_` is set to a false value.)

21. The PDF version of the package's manual is created (to check that the `Rd` files can be converted successfully). This needs LaTeX and suitable fonts and LaTeX packages to be installed. See Section "Making the manuals" in *R Installation and Administration*.
22. Optionally (including by `R CMD check --as-cran`) the HTML version of the manual is created and checked for compliance with the HTML5 standard. This requires a recent version[57] of 'HTML Tidy', either on the path or at a location specified by environment variable `R_TIDYCMD`. Up-to-date versions can be installed from `http://binaries.html-tidy.org/`.

All these tests are run with collation set to the `C` locale, and for the examples and tests with environment variable `LANGUAGE=en`: this is to minimize differences between platforms.

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*: a set of these customizations similar to those used by CRAN can be selected by the option `--as-cran` (which works best if Internet access is available). Some Windows users may need to set environment variable `R_WIN_NO_JUNCTIONS` to a non-empty value. The test of cyclic declarations[58]in `DESCRIPTION` files needs repositories (including CRAN) set: do this in `~/.Rprofile`, by e.g.

```
options(repos = c(CRAN="https://cran.r-project.org"))
```

One check customization which can be revealing is

```
_R_CHECK_CODETOOLS_PROFILE_="suppressLocalUnused=FALSE"
```

which reports unused local assignments. Not only does this point out computations which are unnecessary because their results are unused, it also can uncover errors. (Two such are to intend to update an object by assigning a value but mistype its name or assign in the wrong scope, for example using `<-` where `<<-` was intended.) This can give false positives, most commonly because of non-standard evaluation for formulae and because the intention is to return objects in the environment of a function for later use.

Complete checking of a package which contains a file `README.md` needs a reasonably current version of `pandoc` installed: see `https://pandoc.org/installing.html`.

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

[56] or if option `--use-valgrind` is used or environment variable `_R_CHECK_ALWAYS_LOG_VIGNETTE_OUTPUT_` is set to a true value or if there are differences from a target output file

[57] for the most comprehensive checking this should be 5.8.0 or later: any for which `tidy --version` does not report a version number will be too old – this includes the 2006 version shipped with macOS.

[58] For example, in early 2014 **gdata** (`https://CRAN.R-project.org/package=gdata`) declared '`Imports: gtools`' and **gtools** (`https://CRAN.R-project.org/package=gtools`) declared '`Imports: gdata`'.

(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.

Often `R CMD check` will need to consult a CRAN repository to check details of uninstalled packages. Normally this defaults to the CRAN main site, but a mirror can be specified by setting environment variables `R_CRAN_WEB` and (rarely needed) `R_CRAN_SRC` to the URL of a CRAN mirror.

It is possible to install a package and then check the installed package. To do so first install the package and keep a log of the installation:

```
R CMD INSTALL -l libdir pkg > pkg.log 2>&1
```

and then use

```
Rdev CMD check -l libdir --install=check:pkg.log pkg
```

(Specifying the library is required: it ensures that the just-installed package is the one checked. If you know for sure only one copy is installed you can use `--install=skip`: this is used for R installation's `make check-recommended`.)

1.3.2 Building package tarballs

Packages may be distributed in source form as "tarballs" (`.tar.gz` files) or in binary form. The source form can be installed on all platforms with suitable tools and is the usual form for Unix-like systems; the binary form is platform-specific, and is the more common distribution form for the macOS and '`x86_64`' Windows platforms.

Using `R CMD build`, the R package builder, one can build R package tarballs from their sources (for example, for subsequent release). It is recommended that packages are built for release by the current release version of R or '`r-patched`', to avoid inadvertently picking up new features of a development version of R.

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 makefiles in a `src` directory 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 against the file and directory names relative to the top-level package source directory. In addition, directories from source control systems[59] or from `eclipse`[60], directories with names `check`, `chm`, or ending `.Rcheck` or `Old` or `old` and files `GNUMakefile`[61], `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[62]. In addition, same-package tarballs (from previous builds) and their binary forms will be excluded from the top-level directory, as well as those files in the `R`, `demo` and `man` directories which are flagged by `R CMD check` as having invalid names.

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

---

[59] called `CVS` or `.svn` or `.arch-ids` or `.bzr` or `.git` (but not files called `.git`) or `.hg`.

[60] called `.metadata`.

[61] which is an error: GNU make uses `GNUmakefile`.

[62] see `tools:::.hidden_file_exclusions` and `tools:::get_exclude_patterns()` for further excluded files and file patterns, respectively.

Unless *R CMD build* is invoked with the `--no-build-vignettes` option (or the package's `DESCRIPTION` contains '`BuildVignettes: no`' or similar), it will attempt to (re)build the vignettes (see Section 1.4 [Writing package vignettes], page 45) 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: at the top of this section).

Similarly, if the `.Rd` documentation files contain any `\Sexpr` macros (see Section 2.12 [Dynamic pages], page 95), 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.) This can be suppressed by the option `--no-manual` or if package's `DESCRIPTION` contains '`BuildManual: no`' or similar.

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` (or set the environment variable `_R_BUILD_KEEP_EMPTY_DIRS_` to '`TRUE`', or have a '`BuildKeepEmpty`' field with a true value in the `DESCRIPTION` file).

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`. A package can control how its data is resaved by supplying a '`BuildResaveData`' field (with one of the values given earlier in this paragraph) in its `DESCRIPTION` file.

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` (`https://qpdf.sourceforge.io/`) 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.

Where a non-POSIX file system is in use which does not utilize execute permissions, some care is needed with permissions. This applies on Windows and to e.g. FAT-formatted drives and SMB-mounted file systems on other OSes. The 'mode' of the file recorded in the tarball will be whatever `file.info()` returns. On Windows this will record only directories as having execute permission and on other OSes it is likely that all files have reported 'mode' `0777`. A particular issue is packages being built on Windows which are intended to contain executable scripts such as `configure` and `cleanup`: `R CMD build` ensures those two are recorded with execute permission.

Directory `build` of the package sources is reserved for use by `R CMD build`: it contains information which may not easily be created when the package is installed, including index information on the vignettes and, rarely, information on the help pages and perhaps a copy of the PDF reference manual (see above).

1.3.3 Building binary packages

Binary packages are compressed copies of installed versions of packages. They contain compiled shared libraries rather than C, C++ or Fortran source code, and the R functions are included in their installed form. The format and filename are platform-specific; for example, a binary package for Windows is usually supplied as a `.zip` file, and for the macOS platform the default binary package file extension is `.tgz`.

The recommended method of building binary packages is to use

```
R CMD INSTALL --build pkg
```

where `pkg` is either the name of a source tarball (in the usual `.tar.gz` format) or the location of the directory of the package source to be built. This operates by first installing the package and then packing the installed binaries into the appropriate binary package file for the particular platform.

By default, `R CMD INSTALL --build` will attempt to install the package into the default library tree for the local installation of R. This has two implications:

- If the installation is successful, it will overwrite any existing installation of the same package.
- The default library tree must have write permission; if not, the package will not install and the binary will not be created.

To prevent changes to the present working installation or to provide an install location with write access, create a suitably located directory with write access and use the `-l` option to build the package in the chosen location. The usage is then

```
R CMD INSTALL -l location --build pkg
```

where `location` is the chosen directory with write access. The package will be installed as a subdirectory of `location`, and the package binary will be created in the current directory.

Other options for `R CMD INSTALL` can be found using `R CMD INSTALL --help`, and platform-specific details for special cases are discussed in the platform-specific FAQs.

Finally, at least one web-based service is available for building binary packages from (checked) source code: WinBuilder (see `https://win-builder.R-project.org/`) is able to build '`x86_64`' Windows binaries. Note that this is intended for developers on other platforms who do not have access to Windows but wish to provide binaries for the Windows platform.

1.4 Writing package vignettes

In addition to the help files in `Rd` format, R packages allow the inclusion of documents in arbitrary other formats. The standard location for these is subdirectory `inst/doc` of a source package, the contents will be copied to subdirectory `doc` when the package is installed. Pointers from package help indices to the installed documents are automatically created. Documents in `inst/doc` can be in arbitrary format, however we strongly recommend 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 is *package vignettes*. Vignettes are documents in PDF or HTML format obtained from plain-text literate source files from which R knows how to extract R code and create output (in PDF/HTML or intermediate LaTeX). Vignette engines do this work, using "tangle" and "weave" functions respectively. Sweave, provided by the R distribution, is the default engine. Other vignette engines besides Sweave are supported; see Section 1.4.2 [Non-Sweave vignettes], page 48.

Package vignettes have their sources in subdirectory `vignettes` of the package sources. Note that the location of the vignette sources only affects `R CMD build` and `R CMD check`: the tarball built by `R CMD build` includes in `inst/doc` the components intended to be installed.

Sweave vignette sources are normally given the file extension `.Rnw` or `.Rtex`, but for historical reasons extensions[63] `.Snw` and `.Stex` are also recognized. Sweave allows the integration of LaTeX documents: see the `Sweave` help page in R and the `Sweave` vignette in package **utils** for details on the source document format.

[63] and to avoid problems with case-insensitive file systems, lower-case versions of all these extensions.

Package vignettes are tested by `R CMD check` by executing all R code chunks they contain (except those marked for non-evaluation, e.g., with option `eval=FALSE` for Sweave). The R working directory for all vignette tests in `R CMD check` is a *copy* of the vignette source directory. Make sure all files needed to run the R code in 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 vignettes (such as LaTeX style files, BibTeX input files and files for any figures not created by running the code in the vignette) must be in the vignette source directory. `R CMD check` will check that vignette production has succeeded by comparing modification times of output files in `inst/doc` with the source in `vignettes`.

`R CMD build` will automatically[64] create the (PDF or HTML versions of the) vignettes in `inst/doc` for distribution with the package sources. By including the vignette outputs in the package sources it is not necessary that these can be re-built at install time, i.e., the package author can use private R packages, screen snapshots and LaTeX extensions which are only available on their machine.[65]

By default `R CMD build` will run `Sweave` on all Sweave vignette source files in `vignettes`. If `Makefile` is found in the vignette source directory, then `R CMD build` will try to run `make` after the `Sweave` runs, otherwise `texi2pdf` is run on each `.tex` file produced.

The first target in the `Makefile` should take care of both creation of PDF/HTML 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 of `R_LIBS` and `R_HOME`[66]. 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), use LF line endings and must work correctly with a parallel `make`: too many authors have written things like

```
## BAD EXAMPLE
all: pdf clean

pdf: ABC-intro.pdf ABC-details.pdf

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

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

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

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

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

Others you may see are `\VignettePackage` (currently ignored), `\VignetteDepends` (a comma-separated list of package names) and `\VignetteKeyword` (which replaced `\VignetteKeywords`). These are processed at package installation time to create the saved data frame `Meta/vignette.rds`. The `\VignetteEngine` statement is described in Section 1.4.2 [Non-Sweave vignettes], page 48. Vignette metadata can be extracted from a source file using `tools::vignetteInfo`.

---

[64] unless inhibited by using '`BuildVignettes: no`' in the `DESCRIPTION` file.

[65] provided the conditions of the package's license are met: many, including CRAN, see the omission of source components as incompatible with an Open Source license.

[66] `R_HOME/bin` is prepended to the `PATH` so that references to `R` or `Rscript` in the `Makefile` do make use of the currently running version of R.

At install time an HTML index for all vignettes in the package is automatically created from the `\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, and be valid HTML as confirmed *via* the W3C Markup Validation Service (`https://validator.w3.org`) or Validator.nu (`https://validator.nu/`).

Sweave/Stangle allows the document to specify the `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.

When `R CMD build` builds the vignettes, it copies these and the vignette sources from directory `vignettes` to `inst/doc`. To install any other files from the `vignettes` directory, include a file `vignettes/.install_extras` which specifies these as Perl-like regular expressions on one or more lines. (See the description of the `.Rinstignore` file for full details.)

1.4.1 Encodings and vignettes

Vignettes will in general include descriptive text, R input, R output and figures, LaTeX include files and bibliographic references. As any of these 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 vignette engines process comments to look for options and metadata lines. When an engine's weave and tangle functions are called on the vignette source, it will be converted to the encoding of the current R session.

`Stangle()` will produce an R code file in the current locale's encoding: for a non-ASCII vignette what that is is recorded in a comment at the top of the file.

`Sweave()` will produce a `.tex` file in the current encoding, or in UTF-8 if that is declared. Non-ASCII encodings need to be declared to LaTeX via a line like

```
\usepackage[utf8]{inputenc}
```

(It is also possible to use the more recent '`inputenx`' LaTeX package.) For files where this line is not needed (e.g. chapters included within the body of a larger document, or non-Sweave vignettes), the encoding may be declared using a comment like

```
%\VignetteEncoding{UTF-8}
```

If the encoding is UTF-8, this can also be declared using the declaration

```
%\SweaveUTF8
```

If no declaration is given in the vignette, it will be assumed to be in the encoding declared for the package. If there is no encoding declared in either place, then it is an error to use non-ASCII characters in the vignette.

In any case, be aware that LaTeX may require the '`usepackage`' declaration.

`Sweave()` will also parse and evaluate the R code in each chunk. The R output will also be in the current locale (or UTF-8 if so declared), and should be covered by the '`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 `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 `Sweave()`: this can be circumvented by setting `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 Sweave option `pdf.encoding` may need to be set appropriately: see the help for the `pdf()` graphics device.

As a real example of the complexities, consider the **fortunes** (`https://CRAN.R-project.org/package=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 `fortunes.tex` will be in UTF-8 in any locale. Had `read.table` been told the data were UTF-8, `fortunes.tex` would have been in the locale's encoding.

1.4.2 Non-Sweave vignettes

Vignettes in formats other than Sweave are supported *via* "vignette engines". For example **knitr** (`https://CRAN.R-project.org/package=knitr`) version 1.1 or later can create `.tex` files from a variation on Sweave format, and `.html` files from a variation on "markdown" format. These engines replace the `Sweave()` function with other functions to convert vignette source files into LaTeX files for processing into `.pdf`, or directly into `.pdf` or `.html` files. The `Stangle()` function is replaced with a function that extracts the R source from a vignette.

R recognizes non-Sweave vignettes using filename extensions specified by the engine. For example, the **knitr** (`https://CRAN.R-project.org/package=knitr`) package supports the extension `.Rmd` (standing for "R markdown"). The user indicates the vignette engine within the vignette source using a `\VignetteEngine` line, for example

```
%\VignetteEngine{knitr::knitr}
```

This specifies the name of a package and an engine to use in place of Sweave in processing the vignette. As `Sweave` is the only engine supplied with the R distribution, the package providing any other engine must be specified in the '`VignetteBuilder`' field of the package `DESCRIPTION` file, and also specified in the '`Suggests`', '`Imports`' or '`Depends`' field (since its namespace must be available to build or check your package). If more than one package is specified as a builder, they will be searched in the order given there. The **utils** package is always implicitly appended to the list of builder packages, but may be included earlier to change the search order.

Note that a package with non-Sweave vignettes should always have a '`VignetteBuilder`' field in the `DESCRIPTION` file, since this is how `R CMD check` recognizes that there are vignettes to be checked: packages listed there are required when the package is checked.

The vignette engine can produce `.tex`, `.pdf`, or `.html` files as output. If it produces `.tex` files, R will call `texi2pdf` to convert them to `.pdf` for display to the user (unless there is a `Makefile` in the `vignettes` directory).

Package writers who would like to supply vignette engines need to register those engines in the package `.onLoad` function. For example, that function could make the call

```
tools::vignetteEngine("knitr", weave = vweave, tangle = vtangle,
                      pattern = "[.]Rmd$", package = "knitr")
```

(The actual registration in **knitr** (`https://CRAN.R-project.org/package=knitr`) is more complicated, because it supports other input formats.) See the `?tools::vignetteEngine` help topic for details on engine registration.

1.5 Package namespaces

R has a namespace management system for code in 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 namespace for a package is specified by the `NAMESPACE` file in the top level package directory. This file contains *namespace directives* describing the imports and exports of the

namespace. 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.

Packages are loaded and attached to the search path by calling `library` or `require`. 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. Thus code in the package can only depend on objects in its own namespace and its imports (including the **base** namespace) being visible[67].

Namespaces 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 namespace mechanism and allows code analysis and compilation tools to accurately identify the definition corresponding to a global variable reference in a function body.

The namespace controls the search strategy for variables used by functions in the package. If not found locally, R searches the package namespace first, then the imports, then the base namespace and then the normal search path (so the base namespace precedes the normal search rather than being at the end of it).

1.5.1 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. (This pattern applies to S4 classes.) Beware of patterns which include names starting with a period: some of these are internal-only variables and should never be exported, e.g. '`.__S3MethodsTable__.`' (and loading excludes known cases).

Packages implicitly import the base namespace. Variables exported from other packages with namespaces 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. Using `importFrom` selectively rather than `import` is good practice and recommended notably when importing from packages with more than a dozen exports and especially from those written by others (so what they export can change in future).

[67] Note that lazy-loaded datasets are *not* in the package's namespace so need to be accessed *via* `::`, e.g. `survival::survexp.us`.

To import every symbol from a package but for a few exceptions, pass the `except` argument to `import`. The directive

```
import(foo, except=c(bar, baz))
```

imports every symbol from **foo** except `bar` and `baz`. The value of `except` should evaluate to something coercible to a character vector, after substituting each symbol for its corresponding string.

It is possible to export variables from a namespace which it has imported from other namespaces: this has to be done explicitly and not *via* `exportPattern`.

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 slightly less efficient than a formal import and also loses the advantage of recording all dependencies in the `NAMESPACE` file (but they still need to be recorded in the `DESCRIPTION` file). 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. However, if **foo** is listed only in '`Suggests`' or '`Enhances`' this also delays the check that it is installed: it is good practice to use such imports conditionally (e.g. *via* `requireNamespace("foo", quietly = TRUE)`).

Using the `foo::f` form will be necessary when a package needs to use a function of the same name from more than one namespace.

Using `foo:::f` instead of `foo::f` allows access to unexported objects. This is generally not recommended, as the existence or semantics of unexported objects may be changed by the package author in routine maintenance.

1.5.2 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, and 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.

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

It is possible to specify a third argument to S3method, the function to be used as the method, for example

```
S3method(print, check_so_symbols, .print.via.format)
```

when `print.check_so_symbols` is not needed.

As from R 3.6.0 one can also use `S3method()` directives to perform *delayed* registration. With

```
if(getRversion() >= "3.6.0") {
    S3method(pkg::gen, cls)
}
```

function `gen.cls` will get registered as an S3 method for class `cls` and generic `gen` from package `pkg` only when the namespace of `pkg` is loaded. This can be employed to deal with situations where the method is not "immediately" needed, and having to pre-load the namespace of `pkg` (and all its strong dependencies) in order to perform immediate registration is considered too onerous.

1.5.3 Load hooks

There are a number of hooks called as packages are loaded, attached, detached, and unloaded. See `help(".onLoad")` for more details.

Since loading and attaching are distinct operations, separate hooks are provided for each. These hook functions are called `.onLoad` and `.onAttach`. They both take arguments[68] `libname` and `pkgname`; they should be defined in the namespace but not exported.

Packages can use a `.onDetach` or `.Last.lib` function (provided the latter is exported from the namespace) when `detach` is called on the package. It is called with a single argument, the full path to the installed package. There is also a hook `.onUnload` which is called when the namespace is unloaded (*via* a call to `unloadNamespace`, perhaps called by `detach(unload = TRUE)`) with argument the full path to the installed package's directory. Functions `.onUnload` and `.onDetach` should be defined in the namespace 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.

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

These hooks are often used incorrectly. People forget to export `.Last.lib`. Compiled code should be loaded in `.onLoad` (or *via* a `useDynLb` directive: see below) and unloaded in `.onUnload`. Do remember that a package's namespace can be loaded without the namespace being attached (e.g. by `pkgname::fun`) and that a package can be detached and re-attached whilst its namespace remains loaded.

It is good practice for these functions to be quiet. Any messages should use `packageStartupMessage` so users (include check scripts) can suppress them if desired.

1.5.4 `useDynLib`

A `NAMESPACE` file can contain one or more `useDynLib` directives which allows shared objects that need to be loaded.[69] The directive

```
useDynLib(foo)
```

registers the shared object `foo`[70] 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.

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 namespace 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)
```

[68] they will be called with two unnamed arguments, in that order.

[69] NB: this will only be read in all versions of R if the package contains R code in a `R` directory.

[70] Note that this is the basename of the shared object, and the appropriate extension (`.so` or `.dll`) will be added.

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 more than one DLL. However, this approach is nowadays deprecated in favour of supplying registration information (see below).

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

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

We could then call that routine from R using the command

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

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

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

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

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

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

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

Then, the R variable `myDLL` is in the package's namespace 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 127), 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 namespace mechanism to create R variables for these symbols. For example, suppose we have the following registration information for a DLL named `myDLL`:

```
static R_NativePrimitiveArgType foo_t[] = {
    REALSXP, INTSXP, STRSXP, LGLSXP
};

static const R_CMethodDef cMethods[] = {
   {"foo", (DL_FUNC) &foo, 4, foo_t},
   {"bar_sym", (DL_FUNC) &bar, 0},
```

```
    {NULL, NULL, 0, NULL}
};

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

Then, the directive in the `NAMESPACE` file

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

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

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** (`https://CRAN.R-project.org/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 namespace.

NB: Using these arguments for a package which does not register native symbols merely slows down the package loading (although many CRAN packages have done so). Once symbols are registered, check that the corresponding R variables are not accidentally exported by a pattern in the `NAMESPACE` file.

1.5.5 An example

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

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

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

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

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

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

and NAMESPACE file

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

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

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

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

1.5.6 Namespaces 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 `Depends: methods`[71] in its `DESCRIPTION` and `import(methods)` or `importFrom(methods, ...)` plus any classes and methods which are to be exported need to be declared in the `NAMESPACE` file. For example, the **stats4** package has

```
export(mle) # exporting methods implicitly exports the generic
importFrom("stats", approx, optim, pchisq, predict, qchisq, qnorm, spline)
## 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)
```

All S4 classes to be used outside the package need to be listed in an `exportClasses` directive. Alternatively, they can be specified using `exportClassPattern`[72] in the same style as for `exportPattern`. To export methods for generics from other packages an `exportMethods` directive can be used.

Note that exporting methods on a generic in the namespace will also export the generic, and exporting a generic in the namespace will also export its methods. If the generic function is not local to this package, either because it was imported as a generic function or because the non-generic version has been made generic solely to add S4 methods to it (as for functions such as `coef` in the example above), it can be declared *via* either or both of `export` or `exportMethods`, but the latter is clearer (and is used in the **stats4** example above). In particular, for primitive functions there is no generic function, so `export` would export the primitive, which makes no

[71] `Imports: methods` may suffice, but package code is little exercised without the **methods** package on the search path and may not be fully robust to this scenario.

[72] This defaults to the same pattern as `exportPattern`: use something like `exportClassPattern("^$")` to override this.

sense. On the other hand, if the generic is local to this package, it is more natural to export the function itself using `export()`, and this *must* be done if an implicit generic is created without setting any methods for it (as is the case for `AIC` in **stats4**).

A non-local generic function is only exported to ensure that calls to the function will dispatch the methods from this package (and that is not done or required when the methods are for primitive functions). For this reason, you do not need to document such implicitly created generic functions, and `undoc` in package **tools** will not report them.

If a package uses S4 classes and methods exported from another package, but does not import the entire namespace of the other package[73], it needs to import the classes and methods explicitly, with directives

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

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

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

and

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

respectively.

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

It is important if you export S4 methods that the corresponding generics are available. You may for example need to import `coef` from **stats** to make visible a function to be converted into its implicit generic. But it is better practice to make use of the generics exported by **stats4** as this enables multiple packages to unambiguously set methods on those generics.

1.6 Writing portable packages

This section contains advice on writing packages to be used on multiple platforms or for distribution (for example to be submitted to a package repository such as CRAN).

Portable packages should have simple file names: use only alphanumeric ASCII characters and period (`.`), and avoid those names not allowed under Windows (see Section 1.1 [Package structure], page 3).

Many of the graphics devices are platform-specific: even `X11()` (aka `x11()`) which although emulated on Windows may not be available on a Unix-alike (and is not the preferred screen

[73] if it does, there will be opaque warnings about replacing imports if the classes/methods are also imported.

device on OS X). It is rarely necessary for package code or examples to open a new device, but if essential,[74] use `dev.new()`.

Use `R CMD build` to make the release `.tar.gz` file.

`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` or `configure.ucrt` script to be used on Windows (an empty `configure.win` file if no actions are needed).
- If your package has a `Makevars` or `Makefile` file, make sure that you use only portable make features. Such files should be LF-terminated[75] (including the final line of the file) and not make use of GNU extensions. (The POSIX specification is available at `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/make.html`; anything not documented there should be regarded as an extension to be avoided. Further advice can be found at `https://www.gnu.org/software/autoconf/manual/autoconf.html#Portable-Make`.) Commonly misused GNU extensions are conditional inclusions (`ifeq` and the like), `${shell ...}`, `${wildcard ...}` and similar, and the use of `+=`[76] and `:=`. Also, the use of `$<` other than in implicit rules is a GNU extension, as is the `$^` macro. As is the use of `.PHONY` (some other makes ignore it). Unfortunately makefiles which use GNU extensions often run on other platforms but do not have the intended results.

  Note that the `-C` flag for `make` is not included in the POSIX specification and is not implemented by some of the `make`s which have been used with R. However, it is more commonly implemented (e.g. by FreeBSD `make`) than the GNU-specific `--directory=`.

  You should not rely on built-in/default `make` rules, even when specified by POSIX, as some `make`s do not have the POSIX ones and others have altered them.

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

  ```
  PKG_CPPFLAGS = `gsl-config --cflags`
  ```

  which works in all versions of `make` known[77] to be used with R.

  If you really must require GNU make, declare it in the `DESCRIPTION` file by

  ```
  SystemRequirements: GNU make
  ```

  and ensure that you use the value of environment variable `MAKE` (and not just `make`) in your scripts. (On some platforms GNU make is available under a name such as `gmake`, and there `SystemRequirements` is used to set `MAKE`.)

  If you only need GNU make for parts of the package which are rarely needed (for example to create bibliography files under `vignettes`), use a file called `GNUmakefile` rather than `Makefile` as GNU make (only) will use the former.

  macOS has used GNU make for many years (it previously used BSD make), but the version has been frozen at 3.81 (from 2006).

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

---

[74] People use `dev.new()` to open a device at a particular size: that is not portable but using `dev.new(noRStudioGD = TRUE)` helps.

[75] Solaris `make` did not accept CRLF-terminated Makefiles; Solaris warned about and some other `make`s ignore incomplete final lines.

[76] This was apparently introduced in SunOS 4, and is available elsewhere *provided* it is surrounded by spaces.

[77] GNU make, BSD make and other variants of `pmake` in FreeBSD, NetBSD and formerly in macOS, and formerly AT&T make as implemented on Solaris and 'Distributed Make' (`dmake`), part of Oracle Developer Studio and available in other versions including from Apache OpenOffice.

- If you use `src/Makevars` to compile code in a subdirectory, ensure that you have followed all the advice above. In particular
  - Anticipate a parallel `make`. See Section 1.2.1 [Using Makevars], page 25.
  - Pass macros down to the makefile in the subdirectory, including **all** the needed compiler flags (including PIC and visibility flags). If they are used (even by a default rule) in the subdirectory's Makefile, this includes macros '`AR`' and '`RANLIB`'. See Section 1.2.1.3 [Compiling in sub-directories], page 31, which has a C example. A C++ example:

    ```
    pkg/libpkg.a:
            (cd pkg && $(MAKE) -f make_pkg libpkg.a \
             CXX="$(CXX)" CXXFLAGS="$(CXXFLAGS) $(CXXPICFLAGS) $(C_VISIBILITY)" \
             AR="$(AR)" RANLIB="$(RANLIB)")
    ```

  - Ensure that cleanup will be performed by `R CMD build`, for example in a `cleanup` script or a '`clean`' target.
- If your package uses a `src/Makefile` file to compile code to be linked into R, ensure that it uses exactly the same compiler and flag settings that R uses when compiling such code: people often forget '`PIC`' flags. If `R CMD config` is used, this needs something like (for C++)

  ```
  RBIN = `"${R_HOME}/bin/R"`
  CXX = `"${RBIN}" CMD config CXX`
  CXXFLAGS = `"${RBIN}" CMD config CXXFLAGS` `"${RBIN}" CMD config CXXPICFLAGS`
  ```

- Names of source files including `=` (such as `src/complex_Sig=gen.c`) will confuse some `make` programs and should be avoided.
- Bash extensions also need to be avoided in shell scripts, including expressions in Makefiles (which are passed to the shell for processing). Some R platforms use strict[78] Bourne shells: an earlier R toolset on Windows[79] and some Unix-alike OSes use `ash` (`https://en.wikipedia.org/wiki/Almquist_shell`, a 'lightweight shell with few builtins) or derivatives such as `dash`. Beware of assuming that all the POSIX command-line utilities are available, especially on Windows where only a subset (which has changed by version of `Rtools`) is provided for use with R. One particular issue is the use of `echo`, for which two behaviours are allowed (`https://pubs.opengroup.org/onlinepubs/9699919799/utilities/echo.html`) and both have occurred as defaults on R platforms: portable applications should use neither `-n` (as the first argument) nor escape sequences. The recommended replacement for `echo -n` is the command `printf`. Another common issue is the construction

  ```
  export FOO=value
  ```

  which is `bash`-specific (first set the variable then export it by name).

  Using `test -e` (or `[ -e ]`) in shell scripts is not fully portable[80]: `-f` is normally what is intended. Flags `-a` and `-o` are nowadays declared obsolescent by POSIX and should not be used.

  Use of 'brace expansion', e.g.,

  ```
  rm -f src/*.{o,so,d}
  ```

  is not portable.

  The `-o` flag for `set` in shell scripts is optional in POSIX and not supported on all the platforms R is used on.

  The variable '`OSTYPE`' is shell-specific and its values are rather unpredictable and may include a version such as '`darwin19.0`': `` `uname` `` is often what is intended (with common values '`Darwin`', '`Linux`' and '`SunOS`').

[78] For example, `test` options `-a` and `-e` are not portable, and not supported in the AT&T Bourne shell used on Solaris 10/11, even though they are in the POSIX standard. Nor did Solaris support '`$(cmd)`'.

[79] as from R 4.0.0 the default is `bash`.

[80] it was not in the Bourne shell, and was not supported by Solaris 10.

On macOS which shell `/bin/sh` invokes is user- and platform-dependent: it might be `bash` version 3.2, `dash` or `zsh` (for new accounts it is `zsh`, for accounts ported from Mojave or earlier it is usually `bash`).

- R is not built by default as a shared library on non-Windows platforms (although it commonly is on macOS to support the GUI), so there need not be a file `libR.so` nor `libR.dylib`. Users of `cmake` or `rust` have all too frequently assumed otherwise, so do ensure your package is checked under a vanilla R build. See Section "Configuration options" in *R Installation and Administration*.
- Make use of the abilities of your compilers to check the standards-conformance of your code. For example, `gcc`, `clang` and `gfortran`[81] 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 some of the GNU extensions which fail to compile on most other C++ compilers. If R was not configured accordingly, one can achieve this *via* personal `Makevars` files. See Section "Customizing package compilation" in *R Installation and Administration*,

  Portable C++ code needs to follow all of the 2011, 2014 and 2017 standards (including not using deprecated/removed features) or to specify C+11/14/17/20/23 where available (which is not the case on all R platforms). Currently C++20 support is patchy across R platforms.

  If using Fortran with the GNU compiler, use the flags `-std=f95 -Wall -pedantic` which reject most GNU extensions and features from later standards. (Although R only requires Fortran 90, `gfortran` does not have a way to specify that standard.) Also consider `-std=f2008` as some recent compilers have Fortran 2008 or even 2018 as the minimum supported standard.

  As from macOS 11 (late 2020), its C compiler sets the flag `-Werror=implicit-function-declaration` by default which forces stricter conformance to C99. This can be used on other platforms with `gcc` or `clang`. If your package has a (`autoconf`-generated) `configure script`, try installing it whilst using this flag, and read through the `config.log` file — compilation warnings and errors can lead to features which are present not being detected. (If possible do this on several platforms.)
- `R CMD check` performs some checks for non-portable compiler/linker flags in `src/Makevars`. However, it cannot check the meaning of such flags, and some are commonly accepted but with compiler-specific meanings. There are other non-portable flags which are not checked, nor are `src/Makefile` files and makefiles in sub-directories. As a comment in the code says

  > It is hard to think of anything apart from `-I*` and `-D*` that is safe for general use ...

  although `-pthread` is pretty close to portable. (Option `-U` is portable but little use on the command line as it will only cancel built-in defines (not portable) and those defined earlier on the command line (R does not use any).)

  The GNU option `-pipe` used to be widely accepted by C/C++/Fortran compilers, but has been removed in `flang-new` 18. In any case, it should not be used in distributed code as it may lead to excessive memory use.

  People have used `configure` to customize `src/Makevars`, including for specific compilers. This is unsafe for several reasons. First, unintended compilers might meet the check—for example, several compilers other than GCC identify themselves as 'GCC' whilst being only partially conformant. Second, future versions of compilers may behave differently (including updates to quite old series) so for example `-Werror` (and specializations) can make a package non-installable under a future version. Third, using flags to suppress diagnostic messages

[81] `https://fortranwiki.org/fortran/show/Modernizing+Old+Fortran` may help explain some of the warnings from `gfortran -Wall -pedantic`.

can hide important information for debugging on a platform not tested by the package maintainer. (`R CMD check` can optionally report on unsafe flags which were used.)

Avoid the use of `-march` and especially `-march=native`. This allows the compiler to generate code that will only run on a particular class of CPUs (that of the compiling machine for '`native`'). People assume this is a 'minimum' CPU specification, but that is not how it is documented for `gcc` (it is accepted by `clang` but apparently it is undocumented what precisely it does, and it can be accepted and may be ignored for other compilers). (For personal use `-mtune` is safer, but still not portable enough to be used in a public package.) Not even `gcc` supports '`native`' for all CPUs, and it can do surprising things if it finds a CPU released later than its version.

- Do be very careful with passing arguments between R, C and Fortran code. In particular, `long` in C will be 32-bit on some R platforms (including 64-bit Windows), but 64-bit on most 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/C++11 type such as `int_fast64_t` (and failing that, `long long`) and typedef your own type, or use another suitable type (such as `size_t`, but beware that is unsigned and `ssize_t` is not portable).

  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/C++11 integer types `intptr_t` and `uintptr_t` (in the headers `<stdint.h>` and `<cstdint>`: they are not required to be implemented by the standards but are used in C code by R itself).

  Note that `integer` in Fortran corresponds to `int` in C on all R platforms. There is no such guarantee for Fortran `logical`, and recent `gfortran` maps it to `int_least32_t` on most platforms.
- Under no circumstances should your compiled code ever call `abort` or `exit`[82]: these terminate the user's R process, quite possibly losing all unsaved work. One usage that could call `abort` is the `assert` macro in C or C++ functions, which should never be active in production code. The normal way to ensure that is to define the macro `NDEBUG`, and `R CMD INSTALL` does so as part of the compilation flags. Beware of including headers (including from other packages) which could undefine it, now or in future versions. If you wish to use `assert` during development. you can include `-UNDEBUG` in `PKG_CPPFLAGS` or `#undef` it in your headers or code files. Note that your own `src/Makefile` or makefiles in sub-directories may also need to define `NDEBUG`.

  This applies not only to your own code but to any external software you compile in or link to.
- Compiled code should not write to `stdout` or `stderr` and C++ and Fortran I/O should not be used. As with the previous item such calls may come from external software and may never be called, but package authors are often mistaken about that.
- Compiled code should not call the system random number generators such as `rand`, `drand48` and `random`[83], but rather use the interfaces to R's RNGs described in Section 6.3 [Random numbers], page 168. In particular, if more than one package initializes a system RNG (e.g. *via* `srand`), they will interfere with each other. This applies also to Fortran 90's `random_number` and `random_seed`, and Fortran 2018's `random_init`. And to GNU Fortran's `rand`, `irand` and `srand`. Except for `drand48`, what PRNG these functions use is implementation-dependent.

  Nor should the C++11 random number library be used nor any other third-party random number generators such as those in GSL.

[82] or where supported the variants `_Exit` and `_exit`.

[83] This and `srandom` are in any case not portable. They are in POSIX but not in the C99 standard, and not available on Windows.

- Use of `sprintf` and `vsprintf` is regarded as a potential security risk and warned about on some platforms.[84] `R CMD check` reports if any calls are found.
- 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 [Checking memory access], page 110, for tools 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

  ```
  nm -pg mypkg.so
  ```

  and checking if any of the symbols marked `U` is unexpected is a good way to avoid this.
- Linkers have a lot of freedom in how to resolve entry points in dynamically-loaded code, so the results may differ by platform. One area that has caused grief is packages including copies of standard system software such as `libz` (especially those already linked into R). In the case in point, entry point `gzgets` was sometimes resolved against the old version compiled into the package, sometimes against the copy compiled into R and sometimes against the system dynamic library. The only safe solution is to rename the entry points in the copy in the package. We have even seen problems with entry point name `myprintf`, which is a system entry point[85] on some Linux systems.

  A related issue is the naming of libraries built as part of the package installation. macOS and Windows have case-insensitive file systems, so using

  ```
  -L. -lLZ4
  ```

  in `PKG_LIBS` will match `liblz4`. And `-L.` only appends to the list of searched locations, and `liblz4` might be found in an earlier-searched location (and has been). The only safe way is to give an explicit path, for example

  ```
  ./libLZ4.a
  ```
- Conflicts between symbols in DLLs are handled in very platform-specific ways. Good ways to avoid trouble are to make as many symbols as possible static (check with `nm -pg`), and to use names which are clearly tied to your package (which also helps users if anything does go wrong). Note that symbol names starting with `R_` are regarded as part of R's namespace and should not be used in packages.
- It is good practice for DLLs to register their symbols (see Section 5.4 [Registering native routines], page 127), restrict visibility (see Section 6.16 [Controlling visibility], page 187) and not allow symbol search (see Section 5.4 [Registering native routines], page 127). It should be possible for a DLL to have only one visible symbol, `R_init_`*`pkgname`*, on suitable platforms[86], which would completely avoid symbol conflicts.
- It is not portable to call compiled code in R or other packages *via* `.Internal`, `.C`, `.Fortran`, `.Call` or `.External`, since such interfaces are subject to change without notice and will probably result in your code terminating the R process.
- Do not use (hard or symbolic) file links in your package sources. Where possible `R CMD build` will replace them by copies.
- If you do not yourself have a Windows system, consider submitting your source package to WinBuilder (`https://win-builder.r-project.org/`) before distribution. If you need to check on an M1 Mac, there is a check service at `https://mac.r-project.org/macbuilder/submit.html`.

[84] including macOS as from version 13.

[85] in `libselinux`.

[86] At least Linux and Windows, but not macOS.

- It is bad practice for package code to alter the search path using `library`, `require` or `attach` and this often does not work as intended. For alternatives, see Section 1.1.3.1 [Suggested packages], page 12, and `with()`.
- Examples can be run interactively *via* `example` as well as in batch mode when checking. So they should behave appropriately in both scenarios, conditioning by `interactive()` the parts which need an operator or observer. For instance, progress bars[87] are only appropriate in interactive use, as is displaying help pages or calling `View()` (see below).
- Be careful with the order of entries in macros such as `PKG_LIBS`. Some linkers will re-order the entries, and behaviour can differ between dynamic and static libraries. Generally `-L` options should precede[88] the libraries (typically specified by `-l` options) to be found from those directories, and libraries are searched once in the order they are specified. Not all linkers allow a space after `-L` .
- Care is needed with the use of `LinkingTo`. This puts one or more directories on the include search path ahead of system headers but (prior to R 3.4.0) after those specified in the `CPPFLAGS` macro of the R build (which normally includes `-I/usr/local/include`, but most platforms ignore that and include it with the system headers).

  Any confusion would be avoided by having `LinkingTo` headers in a directory named after the package. In any case, name conflicts of headers and directories under package `include` directories should be avoided, both between packages and between a package and system and third-party software.
- The `ar` utility is often used in makefiles to make static libraries. Its modifier `u` is defined by POSIX but is disabled in GNU `ar` on some Linux distributions which use 'deterministic mode'. The safest way to make a static library is to first remove any existing file of that name then use `$(AR) -cr` and then `$(RANLIB)` if needed (which is system-dependent: on most systems[89] `ar` always maintains a symbol table). The POSIX standard says options should be preceded by a hyphen (as in `-cr`), although most OSes accept them without. Note that on some systems `ar -cr` must have at least one file specified.

  The `s` modifier (to replace a separate call to `ranlib`) is required by X/OPEN but not POSIX, so `ar -crs` is not portable.

  For portability the `AR` and `RANLIB` macros should always be used – some builds require wrappers such as `gcc-ar` or extra arguments to specify plugins.
- The `strip` utility is platform-specific (and CRAN prohibits removing debug symbols). For example the options `--strip-debug` and `--strip-unneeded` of the GNU version are not supported on macOS: the POSIX standard for `strip` does not mention any options, and what calling it without options does is platform-dependent. Stripping a `.so` file could even prevent it being dynamically loaded into R on an untested platform.

  `ld -S` invokes `strip --strip-debug` for GNU `ld` (and similarly on macOS) but is not portable: in particular on Solaris it did something completely different and took an argument.
- Some people have a need to set a locale. Locale names are not portable, and e.g. '`fr_FR.utf8`' is commonly used on Linux but not accepted on macOS. '`fr_FR.UTF-8`' is more portable, being accepted on recent Linux, AIX, FreeBSD, macOS and Solaris (at least). However, some Linux distributions micro-package, so locales defined by **glibc** (including these examples) may not be installed.
- Avoid spaces in file names, not least as they can cause difficulties for external tools. An example was a package with a **knitr** (`https://CRAN.R-project.org/package=knitr`) vi-

[87] except perhaps the simplest kind as used by `download.file()` in non-interactive use.

[88] Whereas the GNU linker reorders so `-L` options are processed first, the Solaris one did not.

[89] some versions of macOS did not.

gnette that used spaces in plot names: this caused some older versions of `pandoc` to fail with a baffling error message.

Non-ASCII filenames can also cause problems (particularly in non-UTF-8 locales).

- Take care in naming LaTeX macros (also known as 'commands') in vignette sources: if these are also defined in a future version of one of the LaTeX packages used there will be a fatal error. One instance in 2021 was package '`hyperref`' newly defining '`\C`', '`\F`', '`\G`', '`\U`' and '`\textapprox`'. If you are confident that your definitions will be the only ones relevant you can use '`\renewcommand`' but it is better to use names clearly associated with your package.
- Make sure that any version requirement for Java code is both declared in the '`SystemRequirements`' field[90] and tested at runtime (not least as the Java installation when the package is installed might not be the same as when the package is run and will not be for binary packages).

  When specifying a minimum Java version please use the official version names, which are (confusingly)

  ```
  1.1 1.2 1.3 1.4 5.0 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
  ```

  and as from 2018 a year.month scheme such as '`18.9`' is also in use. Fortunately only the integer values are likely to be relevant. If at all possible, use one of the LTS versions (8, 11, 17, 21 ...) as the minimum version. The preferred form of version specification is

  ```
  SystemRequirements: Java (>= 11)
  ```

  A suitable test for Java at least version 8 for packages using **rJava** (`https://CRAN.R-project.org/package=rJava`) would be something like

  ```
  .jinit()
  jv <- .jcall("java/lang/System", "S", "getProperty", "java.runtime.version")
  if(substr(jv, 1L, 2L) == "1.") {
    jvn <- as.numeric(paste0(strsplit(jv, "[.]")[[1L]][1:2], collapse = "."))
    if(jvn < 1.8) stop("Java >= 8 is needed for this package but not available")
  }
  ```

  Java 9 changed the format of this string (which used to be something like '`1.8.0_292-b10`'); Java 11 gave `jv` as '`11+28`' whereas Java 11.0.11 gave '`11.0.11+9`'. (`https://openjdk.org:443/jeps/322` details the current scheme. Note that it is necessary to allow for pre-releases like '`11-ea+22`'.)

  Note too that the compiler used to produce a `jar` can impose a minimum Java version, often resulting in an arcane message like

  ```
  java.lang.UnsupportedClassVersionError: ... Unsupported major.minor version 52.0
  ```

  (Where `https://en.wikipedia.org/wiki/Java_class_file` maps class-file version numbers to Java versions.) Compile with something like `javac -target 11` to ensure this is avoided. Note this also applies to packages distributing (or even downloading) compiled Java code produced by others, so their requirements need to be checked (they are often not documented accurately) and accounted for. It should be possible to check the class-file version *via* command-line utility `javap`, if necessary after extracting the `.class` files from a `.jar` archive. For example,

  ```
  jar xvf some.jar
  javap -verbose path/to/some.class | grep major
  ```

  Some packages have stated a requirement on a particular JDK, but a package should only be requiring a JRE unless providing its own Java interface.

[90] If a Java interpreter is required directly (not *via* **rJava** (`https://CRAN.R-project.org/package=rJava`)) this must be declared and its presence tested like any other external command.

Java 8 is still in widespread use (and may remain so because of licence changes and support on older OSes: OpenJDK has security support until March 2026). On the other hand, newer platforms may only have support for recent versions of Java: for '`arm64`' macOS the first officially supported version was 17.

- A package with a hard-to-satisfy system requirement is by definition not portable, annoyingly so if this is not declared in the '`SystemRequirements`' field. The most common example is the use of `pandoc`, which is only available for a very limited range of platforms (and has onerous requirements to install from source) and has capabilities[91] that vary by build but are not documented. Several recent versions of `pandoc` for macOS did not work on R's then target of High Sierra (and this too was undocumented). Another example is the Rust compilation system (`cargo` and `rustc`).

  Usage of external commands should always be conditional on a test for presence (perhaps using `Sys.which`), as well as declared in the '`SystemRequirements`' field. A package should pass its checks without warnings nor errors without the external command being present.

  An external command can be a (possibly optional) requirement for an imported or suggested package but needed for examples, tests or vignettes in the package itself. Such usages should always be declared and conditional.

  Interpreters for scripting languages such as Perl, Python and Ruby need to be declared as system requirements and used conditionally: for example macOS 10.16 was announced not to have them (but released as macOS 11 with them); later it was announced that macOS 12.3 does not have Python 2 and only a minimal install of Python 3 is included. Python 2 has passed end-of-life and been removed from many major distributions. Support for Rust or Go cannot be assumed.

  Command `cmake` is not commonly installed, and where it is, it might not be on the path. In particular, the most common location on macOS is `/Applications/CMake.app/Contents/bin/cmake` and that should be looked for if `cmake` is not found on the path.

- Be sure to use portable encoding names: none of `utf8`, `mac` and `macroman` is portable. See the help for `file` for more details.

- Do not invoke R by plain `R`, `Rscript` or (on Windows) `Rterm` in your examples, tests, vignettes, makefiles or other scripts. As pointed out in several places earlier in this manual, use something like

  ```
  "$(R_HOME)/bin/Rscript"
  "$(R_HOME)/bin$(R_ARCH_BIN)/Rterm"
  ```

  with appropriate quotes (as, although not recommended, `R_HOME` can contain spaces).

- Do not use `R_HOME` in makefiles except when passing them to the shell. Specifically, do not use `R_HOME` in the argument to `include`, as `R_HOME` can contain spaces. Quoting the argument to `include` does not help. A portable and the recommended way to avoid the problem of spaces in `${R_HOME}` is using option `-f` of `make`. This is easy to do with recursive invocation of `make`, which is also the only usual situation when `R_HOME` is needed in the argument for `include`.

  ```
  $(MAKE) -f "${R_HOME}/etc${R_ARCH}/Makeconf" -f Makefile.inner
  ```

- If distributing datasets involving date-times, consider if a time zone needs to be specified. The most portable way to distribute date-times is as objects of class `"POSIXct"` and as these record the time in UTC, the time represented is independent of the time zone: but how it is printed may not be. Objects of class `"POSIXlt"` should have a `"tzone"` attribute. Dates (e.g, birthdays) are conventionally considered independently of time zone.

[91] For example, the ability to handle '`https://`' URLs.

- If at all possible avoid any Internet access during package installation. Installation and use may well be on different machines/accounts and those allowed to install software may have no Internet access, and being self-contained helps ensure long-term reproducibility.

Do be careful in what your tests (and examples) actually test. Bad practice seen in distributed packages include:

- It is not reasonable to test the time taken by a command: you cannot know how fast or how heavily loaded an R platform might be. At best you can test a ratio of times, and even that is fraught with difficulties and not advisable: for example, the garbage collector may trigger at unpredictable times following heuristics that may change without notice.
- Do not test the exact format of R messages (from R itself or from other packages): They change, and they can be translated.

  Packages have even tested the exact format of system error messages, which are platform-dependent and perhaps locale-dependent. For example, in late 2021 `libcurl` changed its warning/error messages, including when URLs are not found.
- Do not test for the absence of warnings (something users of **testthat** (`https://CRAN.R-project.org/package=testthat`) are fond of). Future changes in either R or packages you make use of can create new warnings, and your tests should not make these into errors. (Some deprecation notices may be intended to remain as warnings for a long time.)
- If you use functions such as `View`, remember that in testing there is no one to look at the output. It is better to use something like one of

  ```
  if(interactive()) View(obj) else print(head(obj))
  if(interactive()) View(obj) else str(obj)
  ```
- Be careful when comparing file paths. There can be multiple paths to a single file, and some of these can be very long character strings. If possible canonicalize paths before comparisons, but study `?normalizePath` to be aware of the pitfalls.
- Only test the accuracy of results if you have done a formal error analysis. Things such as checking that probabilities numerically sum to one are silly: numerical tests should always have a tolerance. That the tests on your platform achieve a particular tolerance says little about other platforms. R is configured by default to make use of long doubles where available, but they may not be available or be too slow for routine use. Most R platforms use '`ix86`' or '`x86_64`' CPUs: these may use extended precision registers on some but not all of their FPU instructions. Thus the achieved precision can depend on the compiler version and optimization flags—our experience is that 32-bit builds tend to be less precise than 64-bit ones. But not all platforms use those CPUs, and not all[92] which use them configure them to allow the use of extended precision. In particular, current ARM CPUs do not have extended precision nor long doubles, and `clang` currently has long double the same as double on all ARM CPUs. On the other hand some CPUs have higher-precision modes which may be used for `long double`, notably 64-bit PowerPC and Sparc.

  If you must try to establish a tolerance empirically, configure and build R with `--disable-long-double` and use appropriate compiler flags (such as `-ffloat-store` and `-fexcess-precision=standard` for `gcc`, depending on the CPU type[93]) to mitigate the effects of extended-precision calculations. The platform most often seen to give different numerical results is '`arm64`' macOS, so be sure to include that in any empirical determination.

  Tests which involve random inputs or non-deterministic algorithms should normally set a seed or be tested for many seeds.
- Tests should use `options(warn = 1)` as reporting

  ```
  There were 22 warnings (use warnings() to see them)
  ```

[92] Not doing so is the default on Windows, overridden for the R executables.

[93] These are not needed for the default compiler settings on '`x86_64`' but are likely to be needed on '`ix86`'.

is pointless, especially for automated checking systems.

- If your package uses dates/times, ensure that it works in all timezones, especially those near boundaries (problems have most often be seen in '`Europe/London`' (zero offset in Winter) and '`Pacific/Auckland`', near enough the International Date line) and with offsets not in whole hours (Adelaide, Chatham Islands, ...). More extreme examples are '`Africa/Conakry`' (permanent UTC), '`Asia/Calcutta`' (no DST, permanent half-hour offset) and '`Pacific/Kiritimati`'(no DST, more than 12 hours ahead of UTC).

1.6.1 PDF size

There are a several tools available to reduce the size of PDF files: often the size can be reduced substantially with no or minimal loss in quality. Not only do large files take up space: they can stress the PDF viewer and take many minutes to print (if they can be printed at all).

`qpdf` (`https://qpdf.sourceforge.io/`) can compress losslessly. It is fairly readily available (e.g. it is included in `rtools`, has packages in Debian/Ubuntu/Fedora, and is installed as part of the CRAN macOS distribution of R). `R CMD build` has an option 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 macOS). It seems MiKTeX does not use PDF object compression and so `qpdf` can reduce considerably the sizes of files it outputs: MiKTeX's defaults can be overridden by code in the preamble of an Sweave or LaTeX file — see how this is done for the R reference manual at `https://svn.r-project.org/R/trunk/doc/manual/refman.top`.

Other tools can reduce the size of PDFs containing bitmap images at excessively high resolution. These are often best re-generated (for example `Sweave` defaults to 300 ppi, and 100–150 is more appropriate for a package manual). These tools include Adobe Acrobat (not Reader), Apple's Preview[94] and Ghostscript (which converts PDF to PDF by

```
ps2pdf options -dAutoRotatePages=/None -dPrinted=false in.pdf out.pdf
```

and suitable options might be

```
-dPDFSETTINGS=/ebook
-dPDFSETTINGS=/screen
```

See `https://ghostscript.readthedocs.io/en/latest/VectorDevices.html` for more such and consider all the options for image downsampling). There have been examples in CRAN packages for which current versions of Ghostscript produced much bigger reductions than earlier ones (e.g. at the upgrades from `9.50` to `9.52`, from `9.55` to `9.56` and then to `10.00.0`).

We come across occasionally large PDF files containing excessively complicated figures using PDF vector graphics: such figures are often best redesigned or failing that, output as PNG files.

Option `--compact-vignettes` to `R CMD build` defaults to value '`qpdf`': use '`both`' to try harder to reduce the size, provided you have Ghostscript available (see the help for `tools::compactPDF`).

1.6.2 Check timing

There are several ways to find out where time is being spent in the check process. Start by setting the environment variable `_R_CHECK_TIMINGS_` to '`0`'. This will report the total CPU times (not Windows) and elapsed times for installation and running examples, tests and vignettes, under each sub-architecture if appropriate. For tests and vignettes, it reports the time for each as well as the total.

Setting `_R_CHECK_TIMINGS_` to a positive value sets a threshold (in seconds elapsed time) for reporting timings.

[94] Select 'Save as', and select 'Reduce file size' from the 'Quartz filter' menu': this can be accessed in other ways, for example by Automator.

If you need to look in more detail at the timings for examples, use option `--timings` to `R CMD check` (this is set by `--as-cran`). This adds a summary to the check output for all the examples with CPU or elapsed time of more than 5 seconds. It produces a file *mypkg*`.Rcheck/`*mypkg*`-Ex.timings` containing timings for each help file: it is a tab-delimited file which can be read into R for further analysis.

Timings for the tests and vignette runs are given at the bottom of the corresponding log file: note that log files for successful vignette runs are only retained if environment variable `_R_CHECK_ALWAYS_LOG_VIGNETTE_OUTPUT_` is set to a true value.

1.6.3 Encoding issues

The issues in this subsection have been much alleviated by the change in R 4.2.0 to running the Windows port of R in a UTF-8 locale where available. However, Windows users might be running an earlier version of R on an earlier version of Windows which does not support UTF-8 locales.

Care is needed if your package contains non-ASCII text, and in particular if it is intended to be used in more than one locale. It is possible to mark the encoding used in the `DESCRIPTION` file and in `.Rd` files, as discussed elsewhere in this manual.

First, consider carefully if you really need non-ASCII text. Some 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.[95]. Other characters may not be rendered at all, rendered incorrectly, or cause your R code to give an error. For `.Rd` documentation, marking the encoding and including ASCII transliterations is likely to do a reasonable job. The set of characters which is commonly supported is wider than it used to be around 2000, but non-Latin alphabets (Greek, Russian, Georgian, ...) are still often problematic and those with double-width characters (Chinese, Japanese, Korean, emoji) often need specialist fonts to render correctly.

Several CRAN packages have messages in their R code in French (and a few in German). A better way to tackle this is to use the internationalization facilities discussed elsewhere in this manual.

Function `showNonASCIIfile` in package **tools** can help in finding non-ASCII bytes in files.

There is a portable way to have arbitrary text in character strings (only) in your R code, which is to supply them in Unicode as '`\uxxxx`' escapes (or, rarely needed except for emojis, '`\Uxxxxxxxx`' 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. This applies also to character strings in datasets: they can be prepared using '`\uxxxx`' escapes or encoded in UTF-8 in a UTF-8 locale, or even converted to UTF-8 *via* `iconv()`. If you do this, make sure you have '`R (>= 2.10)`' (or later) in the '`Depends`' field of the `DESCRIPTION` file.

R sessions running in non-UTF-8 locales will if possible re-encode such strings for display (and this is done by `RGui` on older versions of Windows, for example). Suitable fonts will need to be selected or made available[96] both for the console/terminal and graphics devices such as '`X11()`' and '`windows()`'. Using '`postscript`' or '`pdf`' will choose a default 8-bit encoding depending on the language of the UTF-8 locale, and your users would need to be told how to select the '`encoding`' argument.

Note that the previous two paragraphs only apply to character strings in R code. Non-ASCII characters are particularly prevalent in comments (in the R code of the package, in examples, tests, vignettes and even in the `NAMESPACE` file) but should be avoided there. Most commonly

[95] except perhaps some special characters such as backslash and hash which may be taken over for currency symbols.

[96] Typically on a Unix-alike this is done by telling `fontconfig` where to find suitable fonts to select glyphs from.

people use the Windows extensions to Latin-1 (often directional single and double quotes, ellipsis, bullet and en and em dashes) which are not supported in strict Latin-1 locales nor in CJK locales on Windows. A surprisingly common misuse is to use a right quote in '`don't`' instead of the correct apostrophe.

Datasets can include marked UTF-8 or Latin-1 character strings. As R is nowadays unlikely to be run in a Latin-1 or Windows' CP1252 locale, for performance reasons these should be converted to UTF-8.

If you want to run `R CMD check` on a Unix-alike over a package that sets a package encoding in its `DESCRIPTION` file *and do not use a UTF-8 locale* 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`: macOS requires `latin9=fr_FR.ISO8859-15`) except that if the current locale is UTF-8 then the package code is translated to UTF-8 for syntax checking, so it is strongly recommended to check in a UTF-8 locale.

1.6.4 Portable C and C++ code

Writing portable C and C++ code is mainly a matter of observing the standards (C99, C++14 or where declared C++11/17/20) and testing that extensions (such as POSIX functions) are supported. Do make maximal use of your compiler diagnostics — this typically means using flags `-Wall` and `-pedantic` for both C and C++ and additionally `-Werror=implicit-function-declaration` and `-Wstrict-prototypes` for C (on some platforms and compiler versions) these are part of `-Wall` or `-pedantic`).

C++ standards: From version 3.6.0 (3.6.2 on Windows), R defaulted to C++11 where available[97]; from R 4.1.0 to C++14 and from R 4.3.0 to C++17 (where available). However, in earlier versions the default standard was that of the compiler used, often C++98 or C++14, and the default is likely to change in future. For maximal portability a package should either specify a standard (see Section 1.2.4 [Using C++ code], page 34) or be tested under all of C++11, C++98, C++14 and C++17. (Specifying C++14 or later will limit portability.)

Note that the 'TR1' C++ extensions are not part of any of these standards and the `<tr1/name>` headers are not supplied by some of the compilers used for R, including on macOS. (Use the C++11 versions instead.)

A common error is to assume recent versions of compilers or OSes. In production environments 'long term support' versions of OSes may be in use for many years,[98] and their compilers may not be updated during that time. For example, GCC 4.8 was still in use in 2022 and could be (in RHEL 7) until 2028: that supports neither C++14 nor C++17.

The POSIX standards only require recently-defined functions to be declared if certain macros are defined with large enough values, and on some compiler/OS combinations[99] they are not declared otherwise. So you may need to include something like one of

```
#define _XOPEN_SOURCE 600
```

or

```
#ifdef __GLIBC__
# define _POSIX_C_SOURCE 200809L
```

[97] which it is on all known platforms, and is required as from R 4.0.0

[98] Ubuntu provides 5 years of support (but people were running 14.04 after 7 years) and RHEL provides 10 years full support and up to 14 with extended support.

[99] This is seen on Linux, Solaris and FreeBSD, although each has other ways to turn on all extensions, e.g. defining `_GNU_SOURCE`, `__EXTENSIONS__` or `_BSD_SOURCE`: the GCC compilers by default define `_GNU_SOURCE` unless a strict standard such as `-std=c99` is used. On macOS extensions are declared unless one of these macros is given too small a value.

```
#endif
```

before *any* headers. (`strdup`, `strncasecmp` and `strnlen` are such functions – there were several older platforms which did not have the POSIX 2008 function `strnlen`.)

However, some common errors are worth pointing out here. It can be helpful to look up functions at `https://cplusplus.com/reference/` or `https://en.cppreference.com/w/` and compare what is defined in the various standards.

More care is needed for functions such as `mallinfo` which are not specified by any of these standards—hopefully the `man` page on your system will tell you so. Searching online for such pages for various OSes (preferably at least Linux and macOS, and the FreeBSD manual pages at `https://man.freebsd.org/cgi/man.cgi` allow you to select many OSes) should reveal useful information but a `configure` script is likely to be needed to check availability and functionality.

Both the compiler and OS (*via* system header files, which may differ by architecture even for nominally the same OS) affect the compilability of C/C++ code. Compilers from the GCC, LLVM (`clang` and `flang`) Intel and Oracle Developer Studio suites have been used with R, and both LLVM `clang` and Oracle have more than one implementation of C++ headers and library. The range of possibilities makes comprehensive empirical checking impossible, and regrettably compilers are patchy at best on warning about non-standard code.

- Mathematical functions such as `sqrt` are defined in C++11 for floating-point arguments: `float`, `double`, `long double` and possibly more. The standard specifies what happens with an argument of integer type but this is not always implemented, resulting in a report of 'overloading ambiguity': this was commonly seen on Solaris, but for `pow` also seen on macOS and other platforms using `clang++`.

  A not-uncommonly-seen problem is to mistakenly call `floor(x/y)` or `ceil(x/y)` for `int` arguments `x` and `y`. Since `x/y` does integer division, the result is of type `int` and 'overloading ambiguity' may be reported. Some people have (pointlessly) called `floor` and `ceil` on arguments of integer type, which may have an 'overloading ambiguity'.

  A surprising common misuse is things like `pow(10, -3)`: this should be the constant `1e-3`. Note that there are constants such as `M_SQRT2` defined *via* `Rmath.h`[100] for `sqrt(2.0)`, frequently mis-coded as `sqrt(2)`.
- Function `fabs` is defined only for floating-point types, except in C++11 and later which have overloads for `std::fabs` in `<cmath>` for integer types. Function `abs` is defined in C99's `<stdlib.h>` for `int` and in C++'s `<cstdlib>` for integer types, overloaded in `<cmath>` for floating-point types. C++11 has additional overloads for `std::abs` in `<cmath>` for integer types. The effect of calling `abs` with a floating-point type is implementation-specific: it may truncate to an integer. For clarity and to avoid compiler warnings, use `abs` for integer types and `fabs` for double values, and when using C++ include `<cmath>` and use the `std::` prefix.
- It is an error (and make little sense, although has been seen) to call macros/functions `isnan`, `isinf` and `isfinite` for integer arguments: a few compilers give a compilation error. Function `finite` is obsolete, and some compilers will warn about its use[101].
- The GNU C/C++ compilers support a large number of non-portable extensions. For example, `INFINITY` (which is a *float* value in C99 and C++11), for which R provides the portable double value `R_PosInf` (and `R_NegInf` for `-INFINITY`). And `NAN`[102] is just one NaN *float* value: for use with R, `NA_REAL` is often what is intended, but `R_NaN` is also available.

  Some (but not all) extensions are listed at `https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html` and `https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Extensions.html`.

[100] often taken from the toolchain's headers.

[101] at the time of writing '`arm64`' macOS both warned and did not supply a prototype in `math.h` which resulted in a compilation error.

[102] also part of C++11 and later.

Other GNU extensions which have bitten package writers are the use of non-portable characters such as '$' in identifiers and use of C++ headers under `ext`.

item Including C-style headers in C++ code is not portable. Including the legacy header[103] `math.h` in C++ code may conflict with `cmath` which may be included by other headers. In C++11, functions like `sqrt` and `isnan` are defined for `double` arguments in `math.h` and for a range of types including `double` in `cmath`. Similar issues have been seen for `stdlib.h` and `cstdlib`. Including the C++ header first used to be a sufficient workaround but for some 2016 compilers only one could be included.

- Be careful to include the headers which define the functions you use. Some compilers/OSes include other system headers in their headers which are not required by the standards, and so code may compile on such systems and not on others. (A prominent example is the C++ header `<random>` which is indirectly included by `<algorithm>` by `g++`. Another issue is the C header `<time.h>` which is included by other headers on Linux and Windows but not macOS.) `g++` 11 often needs explicit inclusion of the C++ headers `<limits>` (for `numeric_limits`) or `<exception>` (for `set_terminate` and similar), whereas earlier versions included these in other headers. `g++` 13 requires the explicit inclusion of `<cstdint>` for types such as `uint32_t` which was previously included implicitly. (For more such, see `https://gcc.gnu.org/gcc-13/porting_to.html`.)

  Note that `malloc`, `calloc`, `realloc` and `free` are defined by C99 in the header `stdlib.h` and (in the `std::` namespace) by C++ header `cstdlib`. Some earlier implementations used a header `malloc.h`, but that is not portable and does not exist on macOS.

  This also applies to types such as `ssize_t`. The POSIX standards say that is declared in headers `unistd.h` and `sys/types.h`, and the latter is often included indirectly by other headers on some but not all systems.

  Similarly for constants: for example `SIZE_MAX` is defined in `stdint.h` alongside `size_t`.

- Some headers are not portable: we have just mentioned `malloc.h` and often CRAN submissions attempt to use `endian.h`. The latter is a `glibc` extension: some OSes have `machine/endian.h` or `sys/endian.h` but some have neither.

- Use `#include "my.h"` not `#include <my.h>` for headers in your package. The second form is intended for system headers and the search order for such headers is platform-dependent (and may not include the current directory). For extra safety, name headers in a way that cannot be confused with a system header so not, for example, `types.h`.

- For C++ code, be careful to specify namespaces where needed. Many functions are defined by the standards to be in the `std` namespace, but `g++` puts many such also in the C++ main namespace. One way to do so is to use declarations such as

  ```
  using std::floor;
  ```

  but it is usually preferable to use explicit namespace prefixes in the code.

  Examples seen in CRAN packages include

  ```
  abs acos atan bind calloc ceil div exp fabs floor fmod free log malloc
  memcpy memset pow printf qsort round sin sprintf sqrt strcmp strcpy
  strerror strlen strncmp strtol tan trunc
  ```

  This problem is less common than it used to be, but in 2019 LLVM `clang` did not have `bind` in the main namespace. Also seen has been type `size_t` defined only in the `std` namespace.

  NB: These functions are only guaranteed to be in the `std` namespace if the correct C++ header is included, e.g. `<cmath>` rather than `<math.h>`.

[103] which often is the same as the header included by the C compiler, but some compilers have wrappers for some of the C headers.

If you define functions in C++ which are inspired by later standards, put them in a namespace and refer to them using the namespace. We have seen conflicts with `std::make_unique` from C++14 and `std::byte`, `std::data`, `std::sample` and `std::size` from C++17.

- In C++ code

  ```
  using namesapace std;
  ```

  is not good practice, and has caused platform-dependent errors if included before headers, especially system headers (which may be included by other headers). The best practice is to use explicit `std::` prefixes for all functions declared by the C++ standard to be in that namespace.

- Some C++ compilers refuse to compile constructs such as

  ```
  if(ptr > 0) { ....}
  ```

  which compares a pointer to the integer `0`. This could just use `if(ptr)` (pointer addresses cannot be negative) but if needed pointers can be tested against `nullptr` (C++11) or `NULL`.

- Macros defined by the compiler/OS can cause problems. Identifiers starting with an underscore followed by an upper-case letter or another underscore are reserved for system macros and should not be used in portable code (including not as guards in C/C++ headers). Other macros, typically upper-case, may be defined by the compiler or system headers and can cause problems. Some of these can be avoided by defining `_POSIX_C_SOURCE` before including any system headers, but it is better to only use all-upper-case names which have a unique prefix such as the package name.

- `typedef`s in OS headers can conflict with those in the package: examples have included `ulong`, `index_t`, `single` and `thread`. (Note that these may conflict with other uses as identifiers, e.g. defining a C++ function called `single`.) The POSIX standard reserves (in §2.2.2) all identifiers ending in `_t`.

- Some compilers do not allow a space between `-D` and the macro to be defined. Similarly for `-U`.

- If you use OpenMP, check carefully that you have followed the advice in the subsection on Section 1.2.1.1 [OpenMP support], page 28. In particular, any use of OpenMP in C/C++ code will need to use

  ```
  #ifdef _OPENMP
  # include <omp.h>
  #endif
  ```

  Any use of OpenMP functions, e.g. `omp_set_num_threads`, also needs to be conditioned. To avoid incessant warnings such as

  ```
  warning: ignoring #pragma omp parallel [-Wunknown-pragmas]
  ```

  uses of such pragmas should also be conditioned (or commented out if they are used in code in a package not enabling OpenMP on any platform).

  Do not hardcode `-lgomp`: not only is that specific to the GCC family of compilers, using the correct linker flag often sets up the run-time path to the library.

- Package authors commonly assume things are part of C/C++ when they are not: the most common example is POSIX[104] function `strdup`. The most common C library on Linux, `glibc`, will hide the declarations of such extensions unless a 'feature-test macro' is defined **before** (almost) any system header is included. So for `strdup` you need

  ```
  #define _POSIX_C_SOURCE 200809L
  ...
  #include <string.h>
  ```

[104] Although this is expected to be part of C23, full support of that is years away.

```
...
strdup call(s)
```

where the appropriate value can be found by `man strdup` on Linux. (Use of `strncasecmp` is similar.)

However, modes of `gcc` with 'GNU EXTENSIONS' (which are the default, either `-std=gnu99` or `-std=gnu11`) declare enough macros to ensure that missing declarations are rarely seen.

This applies also to constants such as `M_PI` and `M_LN2`, which are part of the X/Open standard: to use these define `_XOPEN_SOURCE` before including any headers, or include the R header `Rmath.h`.

- Using `alloca` portably is tricky: it is neither an ISO C/C++ nor a POSIX function. An adequately portable preamble is

```
#ifdef __GNUC__
/* Includes GCC, clang and Intel compilers */
# undef alloca
# define alloca(x) __builtin_alloca((x))
#elif defined(__sun) || defined(_AIX)
/* this was necessary (and sufficient) for Solaris 10 and AIX 6: */
# include <alloca.h>
#endif
```

- Compiler writers feel free to implement features from later standards than the one specified, so for example they may implement or warn on C++14/17/20 features when C++11 is specified. Portable code will not use such features – it can be hard to know what they are but the most common warnings are

```
'register' storage class specifier is deprecated and incompatible with C++17

ISO C++11 does not allow conversion from string literal to 'char *'
```

(where conversion should be to `const char *`). Keyword `register` was not mentioned in C++98, deprecated in C++11 and removed in C++17.

There are quite a lot of other C++98 features deprecated in C++11 and removed in C++17, and LLVM `clang` 9 and later warn about them (and as from version 16 they have been removed). Examples include `bind1st`/`bind2nd` (use `std::bind` or lambdas[105]) `std::auto_ptr` (replaced by `std::unique_ptr`), `std::mem_fun_ref` and `std::ptr_fun`.

- Later versions of standards may add reserved words: for example `bool`, `false` and `true` became keywords in C23 and are no longer available as variable names. As noted above, C++17 uses `byte`, `data`, `sample` and `size`.

  So avoid common words and keywords from other programming languages.

- Be careful about including C headers in C++ code. Issues include
  - Use of the `register` storage class specifier (see the previous but one item).
  - The C99 keyword `restrict` is not part of[106] any C++ standard and is rejected by some C++ compilers.
  - Inclusion by such headers of C-style headers such as `math.h` (see above).

  The most portable way to interface to other software with a C API is to use C code (which can normally be mixed with C++ code in a package).

[105] `https://stackoverflow.com/questions/32739018/a-replacement-for-stdbind2nd`

[106] it is allowed but ignored in system headers.

- `reinterpret_cast` in C++ is not safe for pointers: for example the types may have different alignment requirements. Use `memcpy` to copy the contents to a fresh variable of the destination type.
- Avoid platform-specific code if at all possible, but if you need to test for a platform ensure that all platforms are covered. For example, `__unix__` is not defined on all Unix-alikes, in particular not on macOS. A reasonably portable way to condition code for a Unix-alike is

  ```
  #if defined (__unix__) || (defined (__APPLE__) && defined (__MACH__))
  #endif
  ```

  but

  ```
  #ifdef _WIN32
  // Windows-specific code
  # if defined(_M_ARM64) || defined(__aarch64__)
  // for ARM
  # else
  // for Intel
  # endif
  #else
  // Unix-alike code
  #endif
  ```

  would be better. For a Unix-alike it is much better to use `configure` to test for the functionality needed than make assumptions about OSes (and people all too frequently forget R is used on platforms other than Linux, Windows and macOS — and some forget macOS).
- Headers in subdirectories are often not portable. For C++, this includes `bits/`, `tr1/` and `tr2/`, none of which exist on macOS (and `ext/` exists there but with different content from `g++`-based platforms). Header `bits/stdc++.h` is both not portable and not recommended for end-user code even on platforms which include it.
- Be careful if using `malloc` or `calloc`. First, their return value must always be checked to see if the allocation succeeded – it is almost always easier to use R's `R_Calloc`, which does check. Second, the first argument is of type `size_t` and recent compilers are warning about passing signed arguments (which could get promoted to ridiculously large values).
- For C code, consider using the flag `-Wstrict-prototypes` which is supported by `gcc` and LLVM and Apple `clang`. This has found quite a number of errors where functions have been declared without arguments and is likely to become the default in future compilers. (It already is for LLVM `clang` in C23 mode.) Note that using `f()` for a function without any parameters was deprecated in C99 and C11, but it expected to be non-deprecated in C23. However, `f(void)` is supported by all standards and avoids any uncertainty.

  LLVM `clang` has a separate warning `-Wdeprecated-non-prototype` which is enabled by `-Wstrict-prototypes`. This warns on K&R-style usage, which will not be accepted in C23.
- Several C entry points are warned against in their `man` pages on most systems, often in very strong terms such as '**Do not use these functions**'. macOS has started to warn[107] if these are used for `sprintf`, `vsprintf`, `gets`, `mktemp`, `tempmam` and `tmpnam`. It is highly recommended that you use safer alternatives (on any platform) but the warning can be avoided by defining '`_POSIX_C_SOURCE`' to for example '`200809L`' before including the (C or C++) header which defines them. (However, this may hide other extensions.)
- Compilers may interpret comments in source code, so it is necessary to remove any intended

[107] when using the macOS 13 SDK with a deployment target of macOS 13.

for a compiler to interpret. The main example has been comments for Visual Fortran (as the Intel Fortran compiler has been known on Windows[108]) like

```
!DEC$ ATTRIBUTES DLLEXPORT,C,REFERENCE,ALIAS:'kdenestmlcvb' :: kdenestmlcvb
```

which are interpreted by Intel Fortran on all platforms (and are inappropriate for use with R on Windows). `gfortran` has similar forms starting with `!GCC$`.

- The C++ `new` operator takes argument `std::size_t size`, which is unsigned. Using a signed integer type such as `int` may lead to compiler warnings such as

```
warning: argument 1 value '18446744073709551615' exceeds maximum object
  size 9223372036854775807  [-Walloc-size-larger-than=]
```

(especially if LTO is used). So don't do that!

- Some authors feel the need to print (using `Rprintf` or similar) vector lengths or indices which are of type `R_xlen_t`. That may be a 32-bit or (most commonly) 64-bit type but which integer type it is mapped to is platform-specific. The safest way is to cast the length to double and use a double format. So one could use something like

```
SEXP Robj; R_xlen_t nelem;
error("Actual: %0.f; Expected %0.f\n", (double) XLENGTH(Robj), (double) nelem);
```

(This could print to full precision, lengths well beyond the address space limits of current OSes, let alone practical limits.)

If you do want to use an integer format, be aware that `R_xlen_t` is implemented by the `int`, `long` or `long long` type on current platforms and even on 64-bit ones need not be the same type as `int64_t`. So the values will need to be cast to the type assumed by the format (and `%lld` was not supported on Windows until R 4.2.0).

Some additional information for C++ is available at `https://journal.r-project.org/archive/2011-2/RJournal_2011-2_Plummer.pdf` by Martyn Plummer.

1.6.4.1 Common symbols

Most OSes (including all those commonly used for R) have the concept of 'tentative definitions' where global C variables are defined without an initializer. Traditionally the linker resolves all tentative definitions of the same variable in different object files to the same object, or to a non-tentative definition. However, `gcc` 10[109] and LLVM `clang` 11[110] changed their default so that tentative definitions cannot be merged and the linker will give an error if the same variable is defined in more than one object file. To avoid this, all but one of the C source files should declare the variable `extern` — which means that any such variables included in header files need to be declared `extern`. A commonly used idiom (including by R itself) is to define all global variables as `extern` in a header, say `globals.h` (and nowhere else), and then in one (and one only) source file use

```
#define extern
# include "globals.h"
#undef extern
```

A cleaner approach is not to have global variables at all, but to place in a single file common variables (declared `static`) followed by all the functions which make use of them: this may result in more efficient code.

The 'modern' behaviour can be seen[111] by using compiler flag `-fno-common` as part of '`CFLAGS`' in earlier versions of `gcc` and `clang`.

[108] and at one time as DEC Fortran, hence the `DEC`.

[109] see `https://gcc.gnu.org/gcc-10/porting_to.html`.

[110] See `https://prereleases.llvm.org/11.0.0/rc2/tools/clang/docs/ReleaseNotes.html#modified-compiler-flags`.

[111] In principle this could depend on the OS, but has been checked on Linux and macOS.

`-fno-common` is said to be particularly beneficial for ARM CPUs.

This is not pertinent to C++ which does not permit tentative definitions.

1.6.4.2 C++17 issues

R 4.3.0 and later default to C++17 when compiling C++, and that finally removed many C++98 features which were deprecated as long ago as C++11. Compiler/runtime authors have been slow to remove these, but LLVM `clang` with its `libc++` runtime library finally started to do so in 2023 – some others warn but some do not.

The principal offender is the 'Boost' collection of C++ headers and libraries. There are two little-documented ways to work around aspects of its outdated code. One is to add

```
-D_HAS_AUTO_PTR_ETC=0
```

to `PKG_CPPLAGS` in `src/Makevars`, `src/Makevars.win` and `src/Makevars.ucrt`. This covers the removal of

```
std::auto_ptr
std::unary_function
std::binary_function
std::random_shuffle
std::binder1st
std::binder2nd
```

with most issues seen with code that includes `boost/functional.hpp`, usually indirectly.

A rarer issue is the use of illegal values for `enum` types, usually negative ones such as

```
BOOST_MPL_AUX_STATIC_CAST(AUX_WRAPPER_VALUE_TYPE, (value - 1));
```

in `boost/mpl/aux_/integral_wrapper.hpp`. Adding

```
-Wno-error=enum-constexpr-conversion
```

to `PKG_CXXFLAGS` will allow this, but that flag is only accepted by recent versions of LLVM `clang` (and will not be in future) so needs a `configure` test.

Pre=built versions of current `clang`/`libc++` are usually available from `https://github.com/llvm/llvm-project/releases` for a wide range of platforms (but the Windows builds there are not compatible with `Rtools` and the macOS ones are unsigned). To select `libc++` add `-stdlib=libc++` to `CXX`, for example by having

```
CXX="/path/to/clang/clang++ -std=gnu++17 -stdlib=libc++"
```

in `~/.R/Makevars`.

Another build for Windows which may be sufficiently compatible with `Rtools` can be found at `https://github.com/mstorsjo/llvm-mingw`: this uses `libc++`.

1.6.5 Portable Fortran code

For many years almost all known R platforms used `gfortran` as their Fortran compiler, but now there are LLVM and 'classic' `flang` and the Intel compilers `ifort`[112] and `ifx` are now free-of-change.

There is still a lot of Fortran code in CRAN packages which predates Fortran 77. Modern Fortran compilers are being written to target a minimum standard of Fortran 2018. and it is desirable that Fortran code in packages complies with that standard. For `gfortran` this can be checked by adding `-std=f2018` to `FFLAGS`. The most commonly seen issues are

- The use of `DFLOAT`, which was superseded by `DBLE` in Fortran 77. Also, use of `DCMPLX`, `DCONJG`, `DIMAG` and similar.

[112] discontinued in 2023.

- The use of GNU Fortran extensions. Some are listed at `https://gcc.gnu.org/onlinedocs/gfortran/Extensions-implemented-in-GNU-Fortran.html`. Others which have caused problems include `etime`, `getpid`, `isnan`[113] and `sizeof`.

  One that frequently catches package writers is that it allows out-of-order declarations: in standard-conformant Fortran variables must be declared (explicitly or implicitly) before use in other declarations such as dimensions.

Unfortunately this flags extensions such as `DOUBLE COMPLEX` and `COMPLEX*16`. R has tested that `DOUBLE COMPLEX` works and so is preferred to `COMPLEX*16`. (One can also use something like `COMPLEX(KIND=KIND(0.0D0))`.)

GNU Fortran 10 and later give a compilation error for the previously widespread practice of passing a Fortran array element where an array is expected, or a scalar instead of a length-one array. See `https://gcc.gnu.org/gcc-10/porting_to.html`. As do the Intel Fortran compilers, and they can be stricter.

The use of `IMPLICIT NONE` is highly recommended – Intel compilers with `-warn` will warn on variables without an explicit type.

Common non-portable constructions include

- The use of Fortran types such as `REAL(KIND=8)` is very far from portable. According to the standards this merely enumerates different supported types, so `DOUBLE PRECISION` might be `REAL(KIND=3)` (and is on an actual compiler). Even if for a particular compiler the value indicates the size in bytes, which values are supported is platform-specific — for example `gfortran` supports values of 4 and 8 on all current platforms and 10 and 16 on a few (but not for example on all '`arm`' CPUs).

  The same applies to `INTEGER(KIND=4)` and `COMPLEX(KIND=16)`.

  Many uses of integer and real variable in Fortran code in packages will interwork with C (for example `.Fortran` is written in C), and R has checked that `INTEGER` and `DOUBLE PRECISION` correspond to the C types `int` and `double`. To make this explicit, from Fortran 2003 one can use the named constants `c_int`, `c_double` and `c_double_complex` from module `iso_c_binding`.
- The Intel compilers only recognize the extensions `.f` (fixed-form) and `.f90` (free-form) and not `.f95`. `R CMD INSTALL` works around this for packages without a `src/Makefile`.
- Use of extensions `.F` and `.F90` to indicate source code to be preprocessed: the preprocessor used is compiler-specific and may or may not be `cpp`.
- Fixed form Fortran (with extension `.f`) should only use 72 columns, and free-form at most 132 columns. This includes trailing comments. Over-long lines may be silently truncated or give a warning. %% The Intel compilers warn.

1.6.6 Binary distribution

If you want to distribute a binary version of a package on Windows or macOS, 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' (`https://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`. The command-line tool `objdump` in the appropriate toolchain will also reveal what DLLs are imported from. If you use a toolchain

[113] There is a portable way to do this in Fortran 2003 (`ieee_is_nan()` in module `ieee_arithmetic`), but that was not supported in the versions 4.x of GNU Fortran. A pretty robust alternative is to test `if(my_var /= my_var)`.

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 macOS, using `R CMD otool -L` on the package's shared object(s) in the `libs` directory will show what they depend on: watch for any dependencies in `/usr/local/lib` or `/usr/local/gfortran/lib`, notably `libgfortran.?.dylib` and `libquadmath.0.dylib`. (For ways to fix these, see Section "Building binary-packages" in *R Installation and Administration*.)

Many people (including the CRAN package repository) will not accept source packages containing binary files as the latter are a security risk. If you want to distribute a source package which needs external software on Windows or macOS, options include

- To arrange for installation of the package to download the additional software from a URL, as e.g. package **Cairo** (`https://CRAN.R-project.org/package=Cairo`) used to.
- To negotiate with Tomas Kalibera to include Windows software in `Rtools` or with Simon Urbanek to include macOS software in his 'recipes' system.
- (For CRAN.) To negotiate with Uwe Ligges to host the additional components on Win-Builder, and write a `configure.win` file to install them.

Be aware that license requirements you may require you to supply the sources for the additional components (and will if your package has a GPL-like license).

1.7 Diagnostic messages

Diagnostic messages can be made available for translation, so 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".
- 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 or a class, 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).

  In R messages it is also possible to use `sQuote` or `dQuote` as in

  ```
  stop(gettextf("object must be of class %s or %s",
                dQuote("manova"), dQuote("maov")),
       domain = NA)
  ```
- Occasionally messages need to be singular or plural (and in other languages there may be no such concept or several plural forms – Slovenian has four). So avoid constructions such as was once used in `library`

  ```
  if((length(nopkgs) > 0) && !missing(lib.loc)) {
  ```

```
        if(length(nopkgs) > 1)
            warning("libraries ",
                    paste(sQuote(nopkgs), collapse = ", "),
                    " contain no packages")
        else
            warning("library ", paste(sQuote(nopkgs)),
                    " contains no package")
    }
```

and was replaced by

```
    if((length(nopkgs) > 0) && !missing(lib.loc)) {
        pkglist <- paste(sQuote(nopkgs), collapse = ", ")
        msg <- sprintf(ngettext(length(nopkgs),
                                "library %s contains no packages",
                                "libraries %s contain no packages",
                                domain = "R-base"),
                       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'.

1.8 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: '`x86_64`' Windows users can find pre-compiled binaries at `https://www.stats.ox.ac.uk/pub/Rtools/goodies/gettext-tools.zip`.

1.8.1 C-level messages

The process of enabling translations is

- In a header file that will be included in all the C (or C++ or Objective C/C++) files containing messages that should be translated, declare

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

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

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

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

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

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

and mark strings by

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

- In the package's `src` directory run

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

The file `src/pkg.pot` is the template file, and conventionally this is shipped as `po/pkg.pot`.

1.8.2 R messages

Mechanisms are also available to support the automatic translation of R `stop`, `warning` and `message` messages. 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.

The R function `ngettext` provides an interface to the C function of the same name: see example in the previous section. It is safest to use `domain="R-pkg"` explicitly in calls to `ngettext`, and necessary for earlier versions of R unless they are calls directly from a function in the package.

1.8.3 Preparing 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.

There is an R function, `update_pkg_po` in package **tools**, to automate much of the maintenance of message translations. See its help for what it does in detail.

If this is called on a package with no existing translations, it creates the directory *pkgdir*`/po`, creates a template file of R messages, *pkgdir*`/po/R-`*pkg*`.pot`, within it, creates the '`en@quot`' translation and installs that. (The '`en@quot`' pseudo-language interprets quotes in their directional forms in suitable (e.g. UTF-8) locales.)

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 call `update_pkg_po` again (this can also be done before it is called for the first time).

When translations to new languages are added in the *pkgdir*`/po` directory, running the same command will check and then install the translations.

If the package sources are updated, the same 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.

The merged translations are run through `tools::checkPofile` to check that C-style formats are used correctly: if not the mismatches are reported and the broken translations are not installed.

This function needs the GNU `gettext-tools` installed and on the path: see its help page.

1.9 CITATION files

An installed file named `CITATION` will be used by the `citation()` function. (It should 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). It will contain calls to function `bibentry`. Here is that for **nlme** (`https://CRAN.R-project.org/package=nlme`):

```
## R package reference generated from DESCRIPTION metadata
citation(auto = meta)

## NLME book
bibentry(bibtype = "Book",
         title = "Mixed-Effects Models in S and S-PLUS",
         author = c(person(c("José", "C."), "Pinheiro"),
                    person(c("Douglas", "M."), "Bates")),
         year = "2000", publisher = "Springer", address = "New York",
         doi = "10.1007/b98882")
```

Note how the first call auto-generates citation information from object `meta`, a parsed version of the `DESCRIPTION` file – it is tempting to hardcode such information, but it normally then gets outdated. How the first entry would look like as a `bibentry` call can be seen from `print(citation("`*pkgname*`", auto = TRUE), style = "R")` for any installed package. Auto-generated information is returned by default if no `CITATION` file is present.

See `?bibentry` for further details of the information which can be provided. In case a bibentry contains LaTeX markup (e.g., for accented characters or mathematical symbols), it may be necessary to provide a text representation to be used for printing *via* the `textVersion` argument to `bibentry`. E.g., earlier versions of **nlme** (`https://CRAN.R-project.org/package=nlme`) additionally used something like

```
textVersion =
paste0("Jose Pinheiro, Douglas Bates, Saikat DebRoy, ",
       "Deepayan Sarkar and the R Core Team (",
       sub("-.*", "", meta$Date),
       "). nlme: Linear and Nonlinear Mixed Effects Models. ",
       sprintf("R package version %s", meta$Version), ".")
```

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

It is desirable (and essential for CRAN) that the `CITATION` file does not contain calls to functions such as `packageDescription` which assume the package is installed in a library tree on the package search path.

1.10 Package types

The `DESCRIPTION` file has an optional field `Type` which if missing is assumed to be '`Package`', the sort of extension discussed so far in this chapter. Currently one other type is recognized; there used also to be a '`Translation`' type.

1.10.1 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.

Many packages of this type need write permission for the R installation directory.

1.11 Services

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

win-builder.r-project.org (`https://win-builder.r-project.org`) offers the automated preparation of ('`x86_64`') Windows binaries from well-tested source packages.

R-Forge (R-Forge.r-project.org (`https://R-Forge.r-project.org`)) and RForge (www.rforge.net (`https://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` (they 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 LATEX, 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 1400 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`.

```
% File src/library/base/man/load.Rd
\name{load}
\alias{load}
\title{Reload Saved Datasets}
\description{
  Reload datasets written with the function \code{save}.
}
\usage{
load(file, envir = parent.frame(), verbose = FALSE)
}
\arguments{
  \item{file}{a (readable binary-mode) \link{connection}
    or a character string giving the name of the file to load
    (when \link{tilde expansion} is done).}
  \item{envir}{the environment where the data should be loaded.}
  \item{verbose}{should item names be printed during loading?}
}
\value{
  A character vector of the names of objects created, invisibly.
}
\seealso{
  \code{\link{save}}.
}
\examples{
## save all data
save(list = ls(all.names = TRUE), file = "all.RData")

## restore the saved values to the current environment
load("all.RData")
}
\keyword{file}
```

An `Rd` file consists of three parts. The header gives basic information about the name of the file, the topics documented, a title, a short textual description and R usage information for the objects documented. The body gives further information (for example, on the function's arguments and return value, as in the above example). Finally, there is 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[1] these should occur only once in an `Rd` file (in any

[1] e.g. `\alias`, `\keyword` and `\note` sections.

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" (`https://developer.r-project.org/Rds.html`) 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` files.

The general syntax of `Rd` files is summarized below. For a detailed technical discussion of current `Rd` syntax, see "Parsing Rd files" (`https://developer.r-project.org/parseRd.pdf`).

`Rd` files consist of four 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 other than the comment marker (`%`) is processed. There is also a rare variant of 'verbatim' text (used in `\eqn`, `\deqn`, `\figure`, and `\newcommand`) where comment markers need not be escaped. The final 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 except the rare 'verbatim' variant (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, except in the 'verbatim' variant. For the complete list of macros and rules for escapes, see "Parsing Rd files" (`https://developer.r-project.org/parseRd.pdf`).

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[2] 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 '`!`' '`|`' nor '`@`'. (LaTeX special characters are allowed, but may not be collated correctly in the index.) There can only be one `\name` entry in a file, and it must not contain any markup and should only contain printable ASCII characters. Entries in the package manual will be in alphabetic[3] order of the `\name` entries.

`\alias{`*topic*`}`

The `\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 *topic* can contain spaces, but (for historical reasons) leading and

[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.

[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.

trailing spaces will be stripped. Percent and left brace need to be escaped by a backslash.

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

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

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

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

`\title{`*`Title`*`}`

Title information for the `Rd` file. This should be capitalized and not end in a period; try to limit its length to at most 65 characters for widest compatibility.

Markup is 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) `\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).

To indicate that a function can be used in several different ways, depending on the named arguments specified, use section `\details`. E.g., `abline.Rd` contains

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

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

For example, `print.ts.Rd` contains

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

which will print as

```
Usage:

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

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

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

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

which will print as

```
Usage:

     ## 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.) Arguments can also be described jointly by separating their names with commas (and optional whitespace) in the `\item` label. 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. There may be optional text outside the `\item` entries (see for example the joint help for `rle` and `inverse.rle`, or the sets of items in `l10n_info`). Note that `\value` is implicitly a `\describe` environ-

ment, so that environment should not be used for listing components, just individual `\item{}{}` entries.[4]

`\references{...}`
: A section with references to the literature. Use `\url{}` or `\href{}{}` for web pointers, and `\doi{}` for DOIs (this needs R >= 3.3, see Section 2.13 [User-defined macros], page 96, for more info).

`\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

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

`\author{...}`
: Information about the author(s) of the `Rd` file. Use `\email{}` without extra delimiters (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 89, and Section 2.5 [Cross-references], page 91).

`\examples{...}`
: Examples of how to use the function. Code in this section is set in typewriter font without reformatting and is run by `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 `example()` and by `R CMD check`. You can use `\dontrun{}` for text that should only be shown, but not run, and `\dontshow{}` for extra commands for testing that should not be shown to users, but will be run by `example()`. (Previously this was called `\testonly`, and that is still accepted.)

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

  For example,

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

  Thus, example code not included in `\dontrun` must be executable! In addition, it should not use any system-specific features or require special facilities (such as Internet access or write permission to specific directories). Text included in `\dontrun` is indicated by comments in the processed help files: it need not be valid R code but

[4] `\describe` can still be used for more general lists, including when `\item` labels need special markup such as `\var` for metasyntactic variables, see Section 2.3 [Marking text], page 89.

the escapes must still be used for `%`, `\` and unpaired braces as in other 'verbatim' text.

Example code must be capable of being run by `example`, which uses `source`. This means that it should not access `stdin`, e.g. to `scan()` data from the example file.

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

Finally, there is `\donttest`, used (at the beginning of a separate line) to mark code that should be run by `example()` but not by `R CMD check` (by default: the option `--run-donttest` can be used). 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. `capabilities()` or `nzchar(Sys.which("someprogram"))` to test for features needed in the examples wherever possible, and you can also use `try()` or `tryCatch()`. Use `interactive()` to condition examples which need someone to interact with.) Note that code included in `\donttest` must be correct R code, and any packages used should be declared in the `DESCRIPTION` file. It is good practice to include a comment in the `\donttest` section explaining why it is needed.

Output from code marked with `\dontdiff` (requires R >= 4.4.0) or between comment lines

```
## IGNORE_RDIFF_BEGIN
## IGNORE_RDIFF_END
```

is ignored when comparing check output to reference output (a *pkg*`-Ex.Rout.save` file). The comment-based markup can also be used for scripts under `tests`.

`\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.

Do strongly consider using `\concept` (see Section 2.9 [Indices], page 94) instead of `\keyword` if you are about to use more than very few non-standard keywords.

The special keyword '`internal`' marks a page of internal topics (typically, objects that are not part of the package's API). If the help page for topic `foo` has keyword '`internal`', then `help(foo)` gives this help page, but `foo` is excluded from several topic indices, including the alphabetical list of topics 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{
  McNeil, D. R. (1977) \emph{Interactive Data Analysis}.
  New York: Wiley.
}
\keyword{datasets}
```

This uses the following additional markup commands.

`\docType{...}`
: Indicates the "type" of the documentation object. Always '`data`' for data sets, 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 91, for more information.

`\source{...}`
: Details of the original source (a reference or URL, see Section 1.1.8 [Specifying URLs], page 20). 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 82.

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 only basic information from the `DESCRIPTION` file. 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 45) and referenced from this page, or into individual man pages for the functions, datasets, or classes.

2.2 Sectioning

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

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

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

For consistency with the pre-assigned sections, the section name (the first argument to `\section`) should be capitalized (but not all upper case) and not end in a period. 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 where 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 2.8 [Insertions], page 93).

`\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 where 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 where possible. Formatting, e.g. line breaks, is preserved. (Note that this includes a line break after the initial {, so typically text should start on the same line as the command.)

  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', to be included within word-wrapped text. Displayed within single quotation marks and using `typewriter` font where possible.

`\verb{`*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 where 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 where 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 where possible.

`\url{`*uniform_resource_locator*`}`
: Indicate a uniform resource locator (URL) for the World Wide Web. The argument is handled as 'verbatim' text (with percent and braces escaped by backslash), and rendered as a hyperlink in HTML and PDF conversion. Line feeds are removed, and leading and trailing whitespace[5] is removed. See Section 1.1.8 [Specifying URLs], page 20.

  Displayed using `typewriter` font where possible.

`\href{`*uniform_resource_locator*`}{`*text*`}`
: Indicate a hyperlink to the World Wide Web. The first argument is handled as 'verbatim' text (with percent and braces escaped by backslash) and is used as the URL in the hyperlink, with the second argument of LaTeX-like text displayed to the user. Line feeds are removed from the first argument, and leading and trailing whitespace is removed.

  Note that RFC3986-encoded URLs (e.g. using '`%28VS.85%29`' in place of '`(VS.85)`') may not work correctly in versions of R before 3.1.3 and are best avoided—use `URLdecode()` to decode them.

`\var{`*metasyntactic_variable*`}`
: Indicate a metasyntactic variable. In most cases this will be rendered distinctly, e.g. in italic (PDF/HTML) or wrapped in '`<...>`' (text), but not in all[6]. LaTeX-like.

`\env{`*environment_variable*`}`
: Indicate an environment variable. 'Verbatim'. Displayed using `typewriter` font where possible

`\option{`*option*`}`
: Indicate a command-line option. 'Verbatim'. Displayed using `typewriter` font where possible.

`\command{`*command_name*`}`
: Indicate the name of a command. LaTeX-like, so `\var` is interpreted. Displayed using `typewriter` font where 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 91), 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.

---

[5] as defined by the R function `trimws`.

[6] Currently it is rendered differently in HTML conversions, and in LaTeX and text conversion outside '`\usage`' and '`\examples`' environments.

`\abbr{`*abbr*`}`
: Indicates an abbreviation. 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

```
\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.
  \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.

Using these without any `\item`s may cause problems with some conversions and makes little sense.

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

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

There must be the same number of fields on each line as there are alignments in the first argument, and they must be non-empty (but can contain only spaces). (There is no whitespace 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 81.

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 with an optional argument, specified as `\link[pkg]{foo}` and `\link[pkg:bar]{foo}`, to link to topics *foo* and *bar* respectively in the package *pkg*. They are currently only used in HTML help (and ignored for hyperlinks in LaTeX conversions of help pages). One should be careful about topics containing special characters (such as arithmetic operators) as they may result in unresolvable links, and preferably use a safer alias in the same help page.

Historically (before R version 4.1.0), links of the form `\link[pkg]{foo}` and `\link[pkg:bar]{foo}` used to be interpreted as links to *files* `foo.html` and `bar.html` in package *pkg*, respectively. For this reason, the HTML help system looks for file `foo.html` in package *pkg* if it does not find topic *foo*, and then searches for the topic in other installed packages. To test that links work both with both old and new systems, the pre-4.1.0 behaviour can be restored by setting the environment variable `_R_HELP_LINKS_TO_TOPICS_=false`.

Packages referred to by these 'other forms' should be declared in the `DESCRIPTION` file, in the '`Depends`', '`Imports`', '`Suggests`' or '`Enhances`' fields.

2.6 Mathematics

Mathematical formulae should be set beautifully for printed documentation and in KaTeX/MathJax-enhanced HTML help (as from R 4.2.0) yet we still want something useful for plain-text (and legacy HTML) help. To this end, the two commands `\eqn{latex}{ascii}` and `\deqn{latex}{ascii}` are used. Whereas `\eqn` is used for "inline" formulae (corresponding to TeX's `$...$`), `\deqn` gives "displayed equations" (as in LaTeX's `displaymath` environment, or TeX's `$$...$$`). Both arguments are treated as 'verbatim' text.

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

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

For the LaTeX manual and in enhanced HTML help, this becomes

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

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

In plain-text help we get

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

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

In legacy HTML help, Greek letters (both cases) will be rendered if preceded by a backslash, `\dots` and `\ldots` will be rendered as ellipses and `\sqrt`, `\ge` and `\le` as mathematical symbols.

Note that only basic LaTeX can be used, there being no provision to specify LaTeX style files, but AMS extensions are supported as from R 4.2.2.

2.7 Figures

To include figures in help pages, use the `\figure` markup. There are three forms.

The two commonly used simple forms are `\figure{`*filename*`}` and `\figure{`*filename*`}{`*alternate text*`}`. This will include a copy of the figure in either HTML or LaTeX output. In text output, the alternate text will be displayed instead. (When the second argument is omitted, the filename will be used.) Both the filename and the alternate text will be parsed verbatim, and should not include special characters that are significant in HTML or LaTeX.

The expert form is `\figure{`*filename*`}{options: `*string*`}`. (The word '`options:`' must be typed exactly as shown and followed by at least one space.) In this form, the *string* is copied into the HTML `img` tag as attributes following the `src` attribute, or into the second argument of the `\Figure` macro in LaTeX, which by default is used as options to an `\includegraphics` call. As it is unlikely that any single string would suffice for both display modes, the expert form would normally be wrapped in conditionals. It is up to the author to make sure that legal HTML/LaTeX is used. For example, to include a logo in both HTML (using the simple form) and LaTeX (using the expert form), the following could be used:

```
\if{html}{\figure{Rlogo.svg}{options: width=100 alt="R logo"}}
\if{latex}{\figure{Rlogo.pdf}{options: width=0.5in}}
```

The files containing the figures should be stored in the directory `man/figures`. Files with extensions `.jpg`, `.jpeg`, `.pdf`, `.png` and `.svg` from that directory will be copied to the `help/figures` directory at install time. (Figures in PDF format will not display in most HTML browsers, but might be the best choice in reference manuals.) Specify the filename relative to `man/figures` in the `\figure` directive.

2.8 Insertions

Use `\R` for the R system itself. The `\dots` macro is a historical alternative to using literal '`...`' for the dots in function argument lists; use `\ldots` for ellipsis dots in ordinary text.[7] These macros 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[8] *almost always* need to be escaped by '`\`', and '`\\`' can be used for backslash and needs to be when there are two or more adjacent backslashes. In R-like code quoted strings are handled slightly differently; see "Parsing Rd files" (`https://developer.r-project.org/parseRd.pdf`) 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

[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.

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.)

2.9 Indices

The `\alias` command (see Section 2.1.1 [Documenting functions], page 82) 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 87). 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.)

Each `\concept` entry should give a *single* index term (word or phrase), and not use any Rd markup.

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

2.10 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.

2.11 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 2.12 [Dynamic pages], page 95).

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:

```
\ifelse{latex}{\out{$\alpha$}}{\ifelse{html}{\out{&alpha;}}{alpha}}
```

2.12 Dynamic pages

Two macros supporting dynamically generated man pages are `\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` *and* `results=verbatim`, a display will be given in a preformatted block. For example, `\Sexpr[echo=TRUE,results=verbatim]{ 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()`, with the result inserted in place. This could be used to insert computed aliases, for instance. `parse_Rd()` is called first with `fragment = FALSE` to allow a single Rd section macro to be inserted. If that fails, it is called again with `fragment = TRUE`, the older behavior.
  - `results=hide` Insert no output.
- `strip.white=true` Remove leading and trailing blank lines in verbatim output if `strip.white=true` (or `TRUE`). With `strip.white=all`, remove all 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 2.10 [Platform-specific sections], page 94) 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" (`https://developer.r-project.org/parseRd.pdf`).

2.13 User-defined macros

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, `\newcommand` requires that the new macro not have been previously defined, whereas `\renewcommand` allows existing macros (including all built-in ones) to be replaced. (This test is disabled by default, but may be enabled by setting the environment variable `_R_WARN_DUPLICATE_RD_MACROS_` to a true value.)

Also as in LaTeX, the new macro may be defined to take arguments, and numeric placeholders such as `#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 `#1` and `#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 place-holders, after which the replacement text will be parsed.

A number of macros are defined in the file `share/Rd/macros/system.Rd` of the R source or home directory, and these will normally be available in all `.Rd` files. For example, that file contains the definition

```
\newcommand{\PR}{\Sexpr[results=rd]{tools:::Rd_expr_PR(#1)}}
```

which defines `\PR` to be a single argument macro; then code (typically used in the `NEWS.Rd` file) like

```
\PR{1234}
```

will expand to

```
\Sexpr[results=rd]{tools:::Rd_expr_PR(1234)}
```

when parsed.

Some macros that might be of general use are:

`\CRANpkg{`*pkg*`}`
: A package on CRAN

`\sspace` A single space (used after a period that does not end a sentence).

`\doi{`*numbers*`}`
: A digital object identifier (DOI).

See the `system.Rd` file in `share/Rd/macros` for more details and macro definitions, including macros `\packageTitle`, `\packageDescription`, `\packageAuthor`, `\packageMaintainer`, `\packageDESCRIPTION` and `\packageIndices`.

Packages may also define their own common macros; these would be stored in an `.Rd` file in `man/macros` in the package source and will be installed into `help/macros` when the package is installed. A package may also use the macros from a different package by listing the other package in the '`RdMacros`' field in the `DESCRIPTION` file.

2.14 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 `\encoding{}` section must be used to specify the encoding if it is not ASCII. (The `\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 `showNonASCIIfile` in package **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[9] is fairly low. Using LaTeX `inputenx` (see `?Rd2pdf` in R) will give greater coverage of UTF-8.

The `\enc` command (see Section 2.8 [Insertions], page 93) 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 a

```
\inputencoding{utf8}
```

command, and this needs to be matched by a suitable invocation of the `\usepackage{inputenc}` command. The R utility `R CMD Rd2pdf` looks at the converted code and includes the encodings used: it might for example use

```
\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 `R CMD Rd2pdf` 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.

2.15 Processing documentation files

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

Using `R CMD Rdconv` one can convert 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.

`R CMD Rd2pdf` generates 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 `DESCRIPTION` files.

[9] 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 `_R_CYRILLIC_TEX_` may need to be set to a non-empty value to enable this.

`R CMD Sweave` and `R CMD Stangle` process vignette-like documentation files (e.g. Sweave vignettes with extension '`.Snw`' or '`.Rnw`', or other non-Sweave vignettes). `R CMD Stangle` is used to extract the R code fragments.

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

All of these work under Windows. You may need to have installed the the tools to build packages from source as described in the "R Installation and Administration" manual, although typically all that is needed is a LaTeX installation.

2.16 Editing Rd files

It can be very helpful to prepare `.Rd` files using a editor which knows about their syntax and will highlight commands, indent to show the structure and detect mis-matched braces, and so on.

The system most commonly used for this is some version of `Emacs` (including `XEmacs`) with the ESS package (`https://ESS.R-project.org/`: it is often is installed with `Emacs` but may need to be loaded, or even installed, separately).

Another is the Eclipse IDE with the Stat-ET plugin (`https://projects.eclipse.org/projects/science.statet`), and (on Windows only) Tinn-R (`https://sourceforge.net/projects/tinn-r/`).

People have also used LaTeX mode in a editor, as `.Rd` files are rather similar to LaTeX files.

Some R front-ends provide editing support for `.Rd` files, for example RStudio (`https://posit.co/`).

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

```
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

```
source("myfuns.R", keep.source = FALSE)
dump(ls(all.names = 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) which line in 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.

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

[2] For Unix-alikes by default these are intervals of CPU time, and for Windows of elapsed ('wall-clock') time. As from R 4.4.0, elapsed time is optional on Unix-alikes

As an example, consider the following code (from Venables & Ripley, 2002, pp. 225–6).

```
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

Each sample represents 0.02 seconds.
Total run time: 22.52 seconds.

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

   %       total       %        self
 total    seconds     self    seconds    name
 100.0     25.22       0.2      0.04     "boot"
  99.8     25.18       0.6      0.16     "statistic"
  96.3     24.30       4.0      1.02     "nls"
  33.9      8.56       2.2      0.56     "<Anonymous>"
  32.4      8.18       1.4      0.36     "eval"
  31.8      8.02       1.4      0.34     ".Call"
  28.6      7.22       0.0      0.00     "eval.parent"
  28.5      7.18       0.3      0.08     "model.frame"
  28.1      7.10       3.5      0.88     "model.frame.default"
  17.4      4.38       0.7      0.18     "sapply"
  15.0      3.78       3.2      0.80     "nlsModel"
  12.5      3.16       1.8      0.46     "lapply"
  12.3      3.10       2.7      0.68     "assign"
 ...

   %        self        %      total
  self    seconds     total   seconds    name
   5.7      1.44       7.5      1.88     "inherits"
   4.0      1.02      96.3     24.30     "nls"
   3.6      0.92       3.6      0.92     "$"
   3.5      0.88      28.1      7.10     "model.frame.default"
   3.2      0.80      15.0      3.78     "nlsModel"
   2.8      0.70       9.8      2.46     "qr.coef"
   2.7      0.68      12.3      3.10     "assign"
   2.5      0.64       2.5      0.64     ".Fortran"
   2.5      0.62       7.1      1.80     "qr.default"
   2.2      0.56      33.9      8.56     "<Anonymous>"
   2.1      0.54       5.9      1.48     "unlist"
   2.1      0.52       7.9      2.00     "FUN"
  ...
```

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 `gc()` with a profiling run without a preceding call to `gc`.

More detailed analysis of the output can be achieved by the tools in the CRAN packages **proftools** (`https://CRAN.R-project.org/package=proftools`) and **profr** (`https://CRAN.R-project.org/package=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. The second and third require R to have been compiled with `--enable-memory-profiling`, which is not the default, but is currently used for the macOS and Windows binary distributions. All can be misleading, for different reasons.

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

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

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

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

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

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

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

3.3.1 Memory statistics from `Rprof`

The sampling profiler `Rprof` described in the previous section can be given the option `memory.profiling=TRUE`. It then writes 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 `duplicate`, which is called to copy R objects. `summaryRprof` provides summaries of this information. The main reason that this can be misleading is that the memory use is attributed to the function running at the end of the sampling interval. A second reason is that garbage collection can make the amount of memory in use decrease, so a function appears to use little memory. Running under `gctorture` helps with both problems: it

slows down the code to effectively increase the sampling frequency and it makes each garbage collection release a smaller amount of memory.

3.3.2 Tracking memory allocations

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

Running the example from the previous section with

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

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

3.3.3 Tracing copies of an object

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

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

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

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

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

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

3.4 Profiling compiled code

Profiling compiled code is highly system-specific, but this section contains some hints gleaned from various R users. Some methods need to be different for a compiled executable and for dynamic/shared libraries/objects as used by R packages.

This chapter is based on reports from users and the information may not be current.

3.4.1 Profiling on Linux

Options include using `sprof` for a shared object, and `oprofile` (see `https://oprofile.sourceforge.io/news/`) and `perf` (see `https://perf.wiki.kernel.org/index.php/Tutorial`) for any executable or shared object. These seem less widely supplied than they used to be. There is also 'Google Performance Tools', also known as **gperftools** or **google-perftools**.

All of these work best when R and any packages have been built with debugging symbols.

3.4.1.1 `perf`

This seems the most widely distributed tool. Here is an example on `x86_64` Linux using R 4.3.1 built with LTO.

At its simplest

```
perf record R -f tests/Examples/stats-Ex.R
perf report --sort=dso
perf report --sort=srcfile
rm perf.data*
```

The first report is

```
  75.67%  R
   9.25%  libc.so.6
   4.87%  [unknown]
   3.75%  libz.so.1.2.11
   3.37%  stats.so
   1.17%  libm.so.6
   0.63%  libtirpc.so.3.0.0
   0.41%  graphics.so
   0.30%  grDevices.so
   0.20%  libRblas.so
   0.09%  libpcre2-8.so.0.11.0
   0.07%  methods.so
   ...
```

which shows which shared libraries (DSOs) the time was spent in.

`perf annotate` can be used on an application built with GCC and `-ggdb`: it interleaves disassembled and source code.

3.4.1.2 `oprofile` and `operf`

The `oprofile` project has two modes of operation. Since version 0.9.8 (August 2012), the preferred mode is to use `operf`, so we discuss only that.

Let us look at the **boot** example from §3.2 on `x86_64` Linux using R 4.3.1.

This can be run under `operf` and analysed by commands like

```
operf R -f boot.R
opreport
opreport -l /path/to/R_HOME/bin/exec/R
opreport -l /path/to/R_HOME/library/stats/src/stats.so
opannotate --source /path/to/R_HOME/bin/exec/R
```

The first line had to be done as root.

The first report shows in which library (etc) the time was spent:

```
CPU_CLK_UNHALT...|
  samples|      %|
------------------
```

```
278341 91.9947 R
 18290  6.0450 libc.so.6
  2277  0.7526 kallsyms
  1426  0.4713 stats.so
   739  0.2442 libRblas.so
   554  0.1831 libz.so.1.2.11
   373  0.1233 libm.so.6
   352  0.1163 libtirpc.so.3.0.0
   153  0.0506 ld-linux-x86-64.so.2
    12  0.0040 methods.so
```

(`kallsyms` is the kernel.)

The rest of the output is voluminous, and only extracts are shown.

Most of the time within R is spent in

```
samples  %        image name symbol name
52955    19.0574  R                      bcEval.lto_priv.0
16484     5.9322  R                      Rf_allocVector3
14224     5.1189  R                      Rf_findVarInFrame3
12581     4.5276  R                      CONS_NR
8289      2.9830  R                      Rf_matchArgs_NR
8034      2.8913  R                      Rf_cons
7114      2.5602  R                      R_gc_internal.lto_priv.0
6552      2.3579  R                      Rf_eval
5969      2.1481  R                      VECTOR_ELT
5684      2.0456  R                      Rf_applyClosure
5497      1.9783  R                      findVarLocInFrame.part.0.lto_priv.0
4827      1.7371  R                      Rf_mkPROMISE
4609      1.6587  R                      Rf_install
4317      1.5536  R                      Rf_findFun3
4035      1.4521  R                      getvar.lto_priv.0
3491      1.2563  R                      SETCAR
3179      1.1441  R                      Rf_defineVar
2892      1.0408  R                      duplicate1.lto_priv.0
```

and in `stats.so`

```
samples  %        image name             symbol name
285      24.4845  stats.so               termsform
284      24.3986  stats.so               numeric_deriv
213      18.2990  stats.so               modelframe
114       9.7938  stats.so               nls_iter
55        4.7251  stats.so               ExtractVars
47        4.0378  stats.so               EncodeVars
37        3.1787  stats.so               getListElement
32        2.7491  stats.so               TrimRepeats
25        2.1478  stats.so               InstallVar
20        1.7182  stats.so               MatchVar
20        1.7182  stats.so               isZeroOne
15        1.2887  stats.so               ConvInfoMsg.isra.0
```

The profiling data is by default stored in sub-directory `oprofile_data` of the current directory, which can be removed at the end of the session.

3.4.1.3 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 -f boot.R
% sprof /path/to/R_HOME/library/stats/libs/stats.so \
  /var/tmp/path/to/R_HOME/library/stats/libs/stats.so.profile

Flat profile:

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

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

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

3.4.2 Profiling on macOS

Developers have recommended `Instruments` (part of `Xcode`, see `https://help.apple.com/instruments/mac/current/`), This had a command-line version prior to macOS 12.

3.4.3 Profiling on Windows

`Very Sleepy` (`https://github.com/VerySleepy/verysleepy`) has been used on '`x86_64`' Windows. There were problems with accessing the debug information, but the best results which included function names were obtained by attaching the profiler to an existing `Rterm` process, either via GUI or using `/a:` (PID obtained via `Sys.getpid()`).

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.

4.1 Browsing

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

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

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

- `n`

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

- `c`

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

- `where`

  This prints the call stack. For example

  ```
  > summary(women)
  Called from: summary.data.frame(women)
  Browse[1]> where
  where 1: summary.data.frame(women)
  where 2: summary(women)

  Browse[1]>
  ```

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

- `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. Errors are often reported to the R mailing lists as being in some package when `traceback()` would show that the error was being reported by some other package or base R. Here is an example from the regression suite.

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

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

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

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

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

```
> traceback()
9: gm(a, b, x)
8: .Call(R_numeric_deriv, expr, theta, rho, dir)
7: numericDeriv(form[[3]], names(ind), env)
6: getRHS()
5: assign("rhs", getRHS(), envir = thisEnv)
4: assign("resid", .swts * (lhs - assign("rhs", getRHS(), envir = thisEnv)),
       envir = thisEnv)
3: function (newPars)
   {
       setPars(newPars)
       assign("resid", .swts * (lhs - assign("rhs", getRHS(), envir = thisEnv)),
           envir = thisEnv)
       assign("dev", sum(resid^2), envir = thisEnv)
       assign("QR", qr(.swts * attr(rhs, "gradient")), envir = thisEnv)
       return(QR$rank < min(dim(QR$qr)))
   }(c(-0.00760232418963883, 1.00119632515036))
```

```
2: .Call(R_nls_iter, m, ctrl, trace)
1: nls(yeps ~ gm(a, b, x), start = list(a = 0.12345, b = 0.54321))
```

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

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

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

The error message

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

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

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

and re-run the example showing the error.

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

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

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

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

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

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

```
Enter an environment number, or 0 to exit  Selection: 2
Browsing in the environment with call:
   glm.fit(x = X, y = Y, weights = weights, start = start, etas
Called from: debugger.look(ind)
Browse[1]> ls()
```

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

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

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

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

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

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

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

Enter a frame number, or 0 to exit

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

Selection:
```

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

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

```
> debug(glm.fit)
> glm(resp ~ 0 + predictor, family = binomial(link ="log"))
debugging in: glm.fit(x = X, y = Y, weights = weights, start = start, etastart = etastart,
    mustart = mustart, offset = offset, family = family, control = control,
```

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

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

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

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

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

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

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

4.3 Checking memory access

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).

Note that memory access errors may be seen with LAPACK, BLAS, OpenMP and Java-using packages: some at least of these seem to be intentional, and some are related to passing characters to Fortran.

Some of these tools can detect mismatched allocation and deallocation. C++ programmers should note that memory allocated by `new []` must be freed by `delete []`, other uses of `new` by `delete`, and memory allocated by `malloc`, `calloc` and `realloc` by `free`. Some platforms will tolerate mismatches (perhaps with memory leaks) but others will segfault.

4.3.1 Using `gctorture`

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

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

The reference to 'memory protection' is to missing C-level calls to `PROTECT`/`UNPROTECT` (see Section 5.9.1 [Garbage Collection], page 142) which if missing allow R objects to be garbage-collected when they are still in use. But it can also help with other memory-related errors.

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

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

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 at the start of the R session to turn on GC torture: `R_GCTORTURE` corresponds to the `step` argument to `gctorture2`, `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:

- 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.

`R CMD check --use-gct` can be set to use `gctorture2(`*`n`*`)` rather than `gctorture(TRUE)` by setting environment variable `_R_CHECK_GCT_N_` to a positive integer value to be used as *`n`*.

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 a common CPU type or supported versions of FreeBSD or Solaris[2] you can use `valgrind` (`https://valgrind.org/`, pronounced to rhyme with 'tinned') to check for possible problems. To run some examples under `valgrind` use something like

```
R -d valgrind --vanilla < mypkg-Ex.R
```

[2] The macOS support is for long-obsolete versions.

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

where `mypkg-Ex.R` is a set of examples, e.g. the file created in `mypkg.Rcheck` by `R CMD check`. Occasionally this reports memory reads of 'uninitialised values' that are the result of compiler optimization, so can be worth checking under an unoptimized compile: for maximal information use a build with debugging symbols. 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 much slower than that. Several versions of `valgrind` were not happy with some optimized BLAS libraries that use CPU-specific instructions so you may need to build a version of R specifically to use with `valgrind`.

On platforms where `valgrind` and its headers[3] are installed 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 anything. Level 1 will detect some uses[4] of uninitialised memory and has little impact on speed (compared to level 0). Level 2 will detect many other memory-use bugs[5] but make 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 0x1CDF6CBE: csc_compTr (Mutils.c:273)
==12539==    by 0x1CE07E1E: tsc_transpose (dtCMatrix.c:25)
==12539==    by 0x80A67A7: do_dotcall (dotcode.c:858)
==12539==    by 0x80CACE2: Rf_eval (eval.c:400)
==12539==    by 0x80CB5AF: R_execClosure (eval.c:658)
==12539==    by 0x80CB98E: R_execMethod (eval.c:760)
==12539==    by 0x1B93DEFA: R_standardGeneric (methods_list_dispatch.c:624)
==12539==    by 0x810262E: do_standardGeneric (objects.c:1012)
==12539==    by 0x80CAD23: Rf_eval (eval.c:403)
==12539==    by 0x80CB2F0: Rf_applyClosure (eval.c:573)
==12539==    by 0x80CADCC: Rf_eval (eval.c:414)
==12539==    by 0x80CAA03: Rf_eval (eval.c:362)
==12539==  Address 0x1C0D2EA8 is 280 bytes inside a block of size 1996 alloc'd
==12539==    at 0x1B9008D1: malloc (vg_replace_malloc.c:149)
==12539==    by 0x80F1B34: GetNewPage (memory.c:610)
==12539==    by 0x80F7515: Rf_allocVector (memory.c:1915)
...
```

This example is from an instrumented version of R, while tracking down a bug in the **Matrix** (`https://CRAN.R-project.org/package=Matrix`) package in 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.

`valgrind` is good at spotting the use of uninitialized values: use option `--track-origins=yes` to show where these originated from. What it cannot detect is the misuse of arrays allocated on the stack: this includes C automatic variables and some[6] Fortran arrays.

[3] in some distributions packaged separately, for example as `valgrind-devel`.

[4] Those in some numeric, logical, integer, raw, complex vectors and in memory allocated by `R_alloc`.

[5] including using the data sections of R vectors after they are freed.

[6] small fixed-size arrays by default in `gfortran`, for example.

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

```
--leak-check=full
--track-origins=yes
```

or setting the environment variable `VALGRIND_OPTS`. As from R 4.2.0, `--use-valgrind` also uses `valgrind` when re-building the vignettes.

This section has described the use of `memtest`, the default (and most useful) of `valgrind`'s tools. There are others described in its documentation: `helgrind` can be useful for threaded programs.

4.3.3 Using the Address Sanitizer

`AddressSanitizer` ('ASan') is a tool with similar aims to the memory checker in `valgrind`. It is available with suitable builds[7] of `gcc` and `clang` on common Linux and macOS platforms. See `https://clang.llvm.org/docs/UsersManual.html#controlling-code-generation`, `https://clang.llvm.org/docs/AddressSanitizer.html` and `https://github.com/google/sanitizers`.

More thorough checks of C++ code are done if the C++ library has been 'annotated': at the time of writing this applied to `std::vector` in `libc++` for use with `clang` and gives rise to '`container-overflow`'[8] reports.

It requires code to have been compiled *and linked* with `-fsanitize=address` and compiling with `-fno-omit-frame-pointer` will give more legible reports. It has a runtime penalty of 2–3x, extended compilation times and uses substantially more memory, often 1–2GB, at run time. On 64-bit platforms it reserves (but does not allocate) 16–20TB of virtual memory: restrictive shell settings can cause problems. It can be helpful to increase the stack size, for example to 40MB.

By comparison with `valgrind`, ASan can detect misuse of stack and global variables but not the use of uninitialized memory.

Recent versions return symbolic addresses for the location of the error provided `llvm-symbolizer`[9] is on the path: if it is available but not on the path or has been renamed[10], one can use an environment variable, e.g.

```
ASAN_SYMBOLIZER_PATH=/path/to/llvm-symbolizer
```

An alternative is to pipe the output through `asan_symbolize.py`[11] and perhaps then (for compiled C++ code) `c++filt`. (On macOS, you may need to run `dsymutil` to get line-number reports.)

The simplest way to make use of this is to build a version of R with something like

```
CC="gcc -std=gnu99 -fsanitize=address"
CFLAGS="-fno-omit-frame-pointer -g -O2 -Wall -pedantic -mtune=native"
```

[7] currently on '`x86_64`'/'`ix86`' Linux and FreeBSD, with some support for Intel macOS but not with the toolchain normally used with R. (There is a faster variant, HWASAN, for '`aarch64`' only.) On some platforms the runtime library, **libasan**, needs to be installed separately, and for checking C++ you may also need **libubsan**.

[8] see `https://llvm.org/devmtg/2014-04/PDFs/LightningTalks/EuroLLVM%202014%20--%20container%20overflow.pdf`.

[9] part of the LLVM project and distributed in `llvm` RPMs and `.deb`s on Linux. It is not currently shipped by Apple.

[10] as Ubuntu has been said to do.

[11] installed on some Linux systems as `asan_symbolize`, and obtainable from `https://github.com/llvm/llvm-project/blob/main/compiler-rt/lib/asan/scripts/asan_symbolize.py`: it makes use of `llvm-symbolizer` if available.

which will ensure that the `libasan` run-time library is compiled into the R executable. However this check can be enabled on a per-package basis by using a `~/.R/Makevars` file like

```
CC = gcc -std=gnu99 -fsanitize=address -fno-omit-frame-pointer
CXX = g++ -fsanitize=address -fno-omit-frame-pointer
FC = gfortran -fsanitize=address
```

(Note that `-fsanitize=address` has to be part of the compiler specification to ensure it is used for linking. These settings will not be honoured by packages which ignore `~/.R/Makevars`.) It will be necessary to build R with

```
MAIN_LDFLAGS = -fsanitize=address
```

to link the runtime libraries into the R executable if it was not specified as part of '`CC`' when R was built. (For some builds without OpenMP, `-pthread` is also required.)

For options available *via* the environment variable `ASAN_OPTIONS` see `https://github.com/google/sanitizers/wiki/AddressSanitizerFlags`. With `gcc` additional control is available *via* the `--param` flag: see its `man` page.

For more detailed information on an error, R can be run under a debugger with a breakpoint set before the address sanitizer report is produced: for `gdb` or `lldb` you could use

```
break __asan_report_error
```

(See `https://github.com/google/sanitizers/wiki/AddressSanitizerAndDebugger`.)

More recent versions[12] added the flag `-fsanitize-address-use-after-scope`: see `https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterScope`.

One of the checks done by ASan is that `malloc/free` and in C++ `new/delete` and `new[]/delete[]` are used consistently (rather than say `free` being used to deallocate memory allocated by `new[]`). This matters on some systems but not all: unfortunately on some of those where it does not matter, system libraries[13] are not consistent. The check can be suppressed by including '`alloc_dealloc_mismatch=0`' in `ASAN_OPTIONS`.

ASan also checks system calls and sometimes reports can refer to problems in the system software and not the package nor R. A couple of reports have been of 'heap-use-after-free' errors in the X11 libraries called from Tcl/Tk.

4.3.3.1 Using the Leak Sanitizer

For `x86_64` Linux there is a leak sanitizer, 'LSan': see `https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer`. This is available on recent versions of `gcc` and `clang`, and where available is compiled in as part of ASan.

One way to invoke this from an ASan-enabled build is by the environment variable

```
ASAN_OPTIONS='detect_leaks=1'
```

However, this was made the default as from `clang` 3.5 and `gcc` 5.1.0.

When LSan is enabled, leaks give the process a failure error status (by default `23`). For an R package this means the R process, and as the parser retains some memory to the end of the process, if R itself was built against ASan all runs will have a failure error status (which may include running R as part of building R itself).

To disable this, allocation-mismatch checking and some strict C++ checking use

```
setenv ASAN_OPTIONS 'alloc_dealloc_mismatch=0:detect_leaks=0:detect_odr_violation=0'
```

LSan also has a 'stand-alone' mode where it is compiled in using `-fsanitize=leak` and avoids the run-time overhead of ASan.

---

[12] including `gcc` 7.1 and `clang` 4.0.0: for `gcc` it is implied by `-fsanitize=address`.

[13] for example, X11/GL libraries on Linux, seen when checking package **rgl** (`https://CRAN.R-project.org/package=rgl`) and some others using it—a workaround is to set environment variable `RGL_USE_NULL=true`.

4.3.4 Using the Undefined Behaviour Sanitizer

'Undefined behaviour' is where the language standard does not require particular behaviour from the compiler. Examples include division by zero (where for doubles R requires the ISO/IEC 60559 behaviour but C/C++ do not), use of zero-length arrays, shifts too far for signed types (e.g. `int x, y; y = x << 31;`), out-of-range coercion, invalid C++ casts and mis-alignment. Not uncommon examples of out-of-range coercion in R packages are attempts to coerce a `NaN` or infinity to type `int` or `NA_INTEGER` to an unsigned type such as `size_t`. Also common is `y[x - 1]` forgetting that `x` might be `NA_INTEGER`.

'UBSanitizer' is a tool for C/C++ source code selected by `-fsanitize=undefined` in suitable builds[14] of `clang` and GCC. Its (main) runtime library is linked into each package's DLL, so it is less often needed to be included in `MAIN_LDFLAGS`. Platforms supported by `clang` are listed at `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#supported-platforms`: CRAN uses it for C/C++ with both GCC and `clang` on '`x86_64`' Linux: the two toolchains often highlight different things with more reports from `clang` than GCC.

This sanitizer may be combined with the Address Sanitizer by `-fsanitize=undefined,address` (where both are supported, and we have seen library conflicts for `clang` 17 and later).

Finer control of what is checked can be achieved by other options.

For `clang` see `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks`. The current set is (on a single line):

```
-fsanitize=alignment,bool,bounds,builtin,enum,float-cast-overflow,
float-divide-by-zero,function,implicit-unsigned-integer-truncation,
implicit-signed-integer-truncation,implicit-integer-sign-change,
integer-divide-by-zero,nonnull-attribute,null,nullability-arg,
nullability-assign,nullability-return,object-size,
pointer-overflow,return,returns-nonnull-attribute,shift,
signed-integer-overflow,unreachable,unsigned-integer-overflow,
unsigned-shift-base,vla-bound,vptr
```

(plus the more specific versions `array-bounsds`, `local-bounds`, `shift-base` and `shift-exponent`), or use something like

```
-fsanitize=undefined -fno-sanitize=float-divide-by-zero
```

where in recent versions `-fno-sanitize=float-divide-by-zero` is the default.

Options `return` and `vptr` apply only to C++: to use `vptr` its run-time library needs to be linked into the main R executable by building the latter with something like

```
MAIN_LD="clang++ -fsanitize=undefined"
```

Option `float-divide-by-zero` is undesirable for use with R which allow such divisions as part of IEC 60559 arithmetic, and in versions of `clang` since June 2019 it is no longer part of `-fsanitize=undefined`.

There are also groups of options `implicit-integer-truncation`, `mplicit-integer-arithmetic-value-change`, `implicit-conversion`, `integer` and `nullability`.

For GCC see `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html` (or the manual for your version of GCC, installed or *via* `https://gcc.gnu.org/onlinedocs/`: look for 'Program Instrumentation Options') for the options supported by GCC: versions 13.x supported

```
-fsanitize=alignment,bool,bounds,builtin,enum,integer-divide-by-zero,
nonnull-attribute,null,object-size,pointer-overflow,return,
returns-nonnull-attribute,shift,signed-integer-overflow,
```

[14] On some platforms the runtime library, **libubsan**, needs to be installed separately.

```
unreachable,vla-bound,vptr
```

plus the more specific versions `shift-base` and `shift-exponent` and non-default options

```
bounds-strict,float-cast-overflow,float-divide-by-zero
```

where `float-divide-by-zero` is not desirable for R uses and `bounds-strict` is an extension of `bounds`.

Other useful flags include

```
-no-fsanitize-recover
```

which causes the first report to be fatal (it always is for the `unreachable` and `return` suboptions). For more detailed information on where the runtime error occurs, using

```
setenv UBSAN_OPTIONS 'print_stacktrace=1'
```

will include a traceback in the report. Beyond that, R can be run under a debugger with a breakpoint set before the sanitizer report is produced: for `gdb` or `lldb` you could use

```
break __ubsan_handle_float_cast_overflow
break __ubsan_handle_float_cast_overflow_abort
```

or similar (there are handlers for each type of undefined behaviour).

There are also the compiler flags `-fcatch-undefined-behavior` and `-ftrapv`, said to be more reliable in `clang` than `gcc`.

For more details on the topic see `https://blog.regehr.org/archives/213` and `https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html` (which has 3 parts).

It may or may not be possible to build R itself with `-fsanitize=undefined`: problems have in the past been seen with OpenMP-using code with `gcc` but there has been success with `clang` up to version 16.. However, problems have been seen with `clang` 17 and later, including missing entry points and R builds hanging. What has succeeded is to use UBSAN just for the package under test (and not in combination with ASAN). To do so, check with an unaltered R, using a custom `Makevars` file something like

```
CC = clang -fsanitize=undefined -fno-sanitize=float-divide-by-zero -fno-omit-frame-poi
CXX = clang++ -fsanitize=undefined -fno-sanitize=float-divide-by-zero -fno-omit-frame-p

UBSAN_DIR = /path/to/LLVM18/lib/clang/18/lib/x86_64-unknown-linux-gnu
SAN_LIBS = $(UBSAN_DIR)/libclang_rt.ubsan_standalone.a $(UBSAN_DIR)/libclang_rt.ubsan_s
```

which links the UBSAN libraries statically into the package-under-test's DSO. It is also possible to use the dynamic library *via*

```
SAN_LIBS = -L$(UBSAN_DIR) -Wl,-rpath,$(UBSAN_DIR) -lclang_rt.ubsan_standalone
```

provided `UBSAN_DIR` is added to the runtime library path (as shown or using `LD_LIBRARY_PATH`). **N.B.**: The details, especially the paths used, have changed several times recently.

4.3.5 Other analyses with 'clang'

Recent versions of `clang` on '`x86_64`' Linux have 'ThreadSanitizer' (`https://github.com/google/sanitizers/wiki#threadsanitizer`), a 'data race detector for C/C++ programs', and 'MemorySanitizer' (`https://clang.llvm.org/docs/MemorySanitizer.html`, `https://github.com/google/sanitizers`) for the detection of uninitialized memory. Both are based on and provide similar functionality to tools in `valgrind`.

`clang` has a 'Static Analyzer' which can be run on the source files during compilation: see `https://clang-analyzer.llvm.org/`.

4.3.6 Other analyses with 'gcc'

GCC 10 introduced a new flag `-fanalyzer` which does static analysis during compilation, currently for C code. It is regarded as *experimental* and it may slow down computation considerably when problems are found (and use many GB of resident memory). There is some overlap with problems detected by the Undefined Behaviour sanitizer, but some issues are only reported by this tool and as it is a static analysis, it does not rely on code paths being exercised.

See `https://gcc.gnu.org/onlinedocs/gcc-10.1.0/gcc/Static-Analyzer-Options.html` (or the documentation for your version of `gcc` if later) and `https://developers.redhat.com/blog/2020/03/26/static-analysis-in-gcc-10`

4.3.7 Using 'Dr. Memory'

'Dr. Memory' from `https://drmemory.org/` is a memory checker for (currently) Windows, Linux and macOS with similar aims to `valgrind`. It works with unmodified executables[15] and detects memory access errors, uninitialized reads and memory leaks.

4.3.8 Fortran array bounds checking

Most of the Fortran compilers used with R allow code to be compiled with checking of array bounds: for example `gfortran` has option `-fbounds-check`. This will give an error when the upper or lower bound is exceeded, e.g.

```
At line 97 of file .../src/appl/dqrdc2.f
Fortran runtime error: Index '1' of dimension 1 of array 'x' above upper bound of 0
```

One does need to be aware that lazy programmers often specify Fortran dimensions as `1` rather than `*` or a real bound and these will be reported (as may `*` dimensions)

It is easy to arrange to use this check on just the code in your package: add to `~/.R/Makevars` something like (for `gfortran`)

```
FFLAGS = -g -O2 -mtune=native -fbounds-check
```

when you run `R CMD check`.

This may report errors with the way that Fortran character variables are passed, particularly when Fortran subroutines are called from C code and character lengths are not passed (see Section 6.6.1 [Fortran character strings], page 172).

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 other debuggers such as `lldb` (`https://lldb.llvm.org/`, used on macOS) and Sun's `dbx`: some debuggers have graphical front-ends available.

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))
```

[15] but works better if inlining and frame pointer optimizations are disabled.

```
 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 options are the first or third. (Note that a core dump is only produced where enabled: a common default in a shell is to limit its size to 0, thereby disabling it.)

A fairly common cause of such crashes is a package which uses `.C` or `.Fortran` and writes beyond (at either end) one of the arguments it is passed. There is a good way to detect this: using `options(CBoundsCheck = TRUE)` (which can be selected *via* the environment variable `R_C_BOUNDS_CHECK=yes`) changes the way `.C` and `.Fortran` work to check if the compiled code writes in the 64 bytes at either end of an argument.

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 R can safely catch that and return to the top-level prompt, so one gets something like

```
> .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. Further, the size of the stack is set when R is compiled on Windows, whereas on POSIX OSes it can be set in the shell from which R is launched.

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

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

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

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

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

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

```
#5  0xb7d6c0a5 in do_random2 (call=0x94bf7d4, op=0x92580e8, args=0x9698f98,
    rho=0x9698f28) at /users/ripley/R/svn/R-devel/src/main/random.c:183
...
```

In many cases it is possible to attach a debugger to a running process: this is helpful if an alternative front-end is in use or to investigate a task that seems to be taking far too long. This is done by something like

```
gdb -p pid
```

where *pid* is the id of the R executable or front-end process and can be found from within a running R process by calling `Sys.getpid()` or from a process monitor. This stops the process so its state can be examined: use `continue` to resume execution.

Some "tricks" worth knowing follow:

4.4.1 Finding entry points in dynamically loaded code

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

- Call the debugger on the R executable, for example by *R -d gdb*.
- Start R.
- At the R prompt, use `dyn.load` or `library` to load your shared object.
- Send an interrupt signal. This will put you back to the debugger prompt.
- Set the breakpoints in your code.
- Continue execution of R by typing *signal 0*RET.

Under Windows signals may not be able to be used, and if so the procedure is more complicated. See the rw-FAQ.

4.4.2 Inspecting R objects when debugging

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

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

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

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

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

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

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

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

```
Value returned is $1 = (SEXPREC *) 0x40583e1c
(gdb) p *$1
$2 = {
  sxpinfo = {type = 19, obj = 1, named = 1, gp = 0,
    mark = 0, debug = 0, trace = 0, = 0},
  attrib = 0x40583e80,
  u = {
    vecsxp = {
      length = 2,
      type = {c = 0x40634700 "0>X@D>X@0>X@", i = 0x40634700,
        f = 0x40634700, z = 0x40634700, s = 0x40634700},
      truelength = 1075851272,
    },
    primsxp = {offset = 2},
    symsxp = {pname = 0x2, value = 0x40634700, internal = 0x40203008},
    listsxp = {carval = 0x2, cdrval = 0x40634700, tagval = 0x40203008},
    envsxp = {frame = 0x2, enclos = 0x40634700},
    closxp = {formals = 0x2, body = 0x40634700, env = 0x40203008},
    promsxp = {value = 0x2, expr = 0x40634700, env = 0x40203008}
  }
}
```

(Debugger output reformatted for better legibility).

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

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

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

$class
[1] "data.frame"

$3 = void
```

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

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

Another alternative is the `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):

```
(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 CHARSXP g0c1 [MARK,gp=0x21] "a"
      @10095e518 09 CHARSXP g0c1 [MARK,gp=0x21] "b"
    TAG: @100859e60 01 SYMSXP g0c0 [MARK,NAM(2),gp=0x4000] "row.names"
    @102a6f868 13 INTSXP g0c1 [NAM(2)] (len=2, 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 CHARSXP g0c2 [MARK,gp=0x21,ATT] "data.frame"
```

In general the representation of each object follows the format:

```
@<address> <type-nr> <type-name> <gc-info> [<flags>] ...
```

For a more fine-grained control over the depth of the recursion and the output of vectors `R_inspect3` takes additional two character() parameters: maximum depth and the maximal number of elements that will be printed for scalar vectors. The defaults in `R_inspect` are currently -1 (no limit) and 5 respectively.

4.4.3 Debugging on macOS

To debug code in a package it is easiest to unpack it in a directory and install it with

```
R CMD INSTALL --dsym pkgname
```

as macOS does not store debugging symbols in the `.so` file. (It is not necessary to have R built with debugging symbols, although compiling the package should be done including `-g` in `CFLAGS` / `CXXFLAGS` / `FFLAGS` / `FCFLAGS` as appropriate.)

Security measures may prevent running a CRAN binary distribution of R under `lldb` or attaching this as a debugger (`https://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html#I-cannot-attach-debugger-to-R`), although both were possible on High Sierra and are again from R 4.2.0. This can also affect locally compiled builds, where attaching to an interactive R session under Big Sur or Monterey worked in 2022 after giving administrator permission *via* a popup-up. (To debug in what Apple deems a non-interactive session, e.g. logged in remotely, see `man DevToolsSecurity`.)

Debugging a local build of R on macOS can raise additional hurdles as environment variables such as `DYLD_FALLBACK_LIBRARY_PATH` are not usually passed through[16] the `lldb` process, resulting in messages like

```
R -d lldb
...
(lldb) run
Process 16828 launched: '/path/to/bin/exec/R' (x86_64)
dyld: Library not loaded: libR.dylib
  Referenced from: /path/to/bin/exec/R
```

A quick workaround is to symlink the dylibs under `R_HOME/lib` to somewhere where they will be found such as the current working directory. It would be possible to do as the distribution

[16] By default as a security measure: see `man dyld`.

does[17] and use `install_name_tool`, but that would have to be done for all the dylibs including those in packages.

It may be simplest to attach the debugger to a running process (see above). Specifically, run R and when it is at the prompt just before a command that is to be debugged, at a terminal

```
ps -ef | grep exec/R
# identify the PID pid for the next command: it is the second item
lldb -p pid
(lldb) continue
```

and then return to the R console.

For non-interactive use, one may need `lldb --batch`.

4.5 Using Link-time Optimization

Where supported, *link time optimization* provides a comprehensive way to check the consistency of calls between Fortran files or between C and Fortran. Use this *via* `R CMD INSTALL --use-LTO` (but that does not apply if there is a `src/Makefile` file or a Windows analogue).

To set up support on a Unix-alike, see Section "Link-Time Optimization" in *R Installation and Administration*. On Linux using GCC without building R with LTO support, it should suffice to set

```
LTO_OPT = -flto
LTO_FC_OPT = -flto
AR = gcc-ar
NM = gcc-nm
```

in a personal (or site) `Makevars` file: See Section "Customizing package compilation" in *R Installation and Administration*,

For Windows, first edit file `etc/${R_ARCH}/Makeconf` to give `LTO_OPT` the value `-flto` or do so in a personal/site `Makevars` file; see also file `src/gnuwin32/README.compilation` in the sources.

For example:

```
boot.f:61: warning: type of 'ddot' does not match original declaration [-Wlto-type-misr
        y(j,i)=ddot(p,x(j,1),n,b(1,j,i),1)
crq.f:1023: note: return value type mismatch
```

where the package author forgot to declare

```
      double precision ddot
      external ddot
```

in `boot.f`. That package had its own copy of `ddot`: to detect misuse of the one in R's BLAS library would have needed R configured with `--enable-lto=check`.

Further examples:

```
rkpk2.f:77:5: warning: type of 'dstup' does not match original declaration [-Wlto-type-
      *info, wk)
rkpk1.f:2565:5: note: type mismatch in parameter 14
       subroutine dstup (s, lds, nobs, nnull, qraux, jpvt, y, q, ldqr,
rkpk1.f:2565:5: note: 'dstup' was previously declared here
```

where the fourteenth argument `dum` was missing in the call.

```
reg.f:78:33: warning: type of 'dqrdc' does not match original declaration [-Wlto-type-m
       call dqrdc (sr, nobs, nobs, nnull, wk, dum, dum, 0)
```

[17] See `https://svn.r-project.org/R-dev-web/trunk/CRAN/QA/Simon/R-build/fixpathR`: '`@executable_path`' could be used rather than absolute paths.

```
dstup.f:20: note: 'dqrdc' was previously declared here
       call dqrdc (s, lds, nobs, nnull, qraux, jpvt, work, 1)
```

`dqrdc` is a LINPACK routine from R, `jpvt` is an integer array and `work` is a double precision one so `dum` cannot match both. (If `--enable-lto=check` had been used the comparison would have been with the definition in R.)

For Fortran files all in the package, most inconsistencies can be detected by concatenating the Fortran files and compiling the result, sometimes with clearer diagnostics than provided by LTO. For our last two examples this gives

```
all.f:2966:72:

      *info, work1)
                                                                        1
Warning: Missing actual argument for argument 'dum' at (1)
```

and

```
all.f:1663:72:

      *ipvtwk), wk(ikwk), wk(iwork1), wk(iwork2), info)
                                                                      1
Warning: Type mismatch in argument 'jpvt' at (1); passed REAL(8) to INTEGER(4)
```

On a Unix-alike for a package with a `src/Makefile` file, LTO can be enabled by including suitable flags in that file, for example

```
LTO = $(LTO_OPT)
LTO_FC = $(LTO_FC_OPT)
```

and ensuring these are used for compilation, for example as part of `CFLAGS`, `CXXFLAGS` or `FCFLAGS`. If `R CMD SHLIB` is used for compilation, add `--use-LTO` to its call.

On Windows for a package with a `src/Makefile.ucrt` or `src/Makefile.win` file which includes '`"${R_HOME}/etc${R_ARCH}/Makeconf"`', include

```
LTO = $(LTO_OPT)
```

or to always use LTO however R was built,

```
LTO = -flto
```

5 System and foreign language interfaces

5.1 Operating system access

Access to operating system functions is *via* the R functions `system` and `system2`. 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 to `system` will be `internal` which if true will collect the output of the command into an R character vector.

On POSIX-compliant OSes these commands pass a command-line to a shell: Windows is not POSIX-compliant and there is a separate function `shell` to do so.

The function `system.time` is available for timing. Timing on child processes is only available on Unix-alikes, and may not be reliable there.

5.2 Interface functions `.C` and `.Fortran`

These two functions provide an 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 126). They are primarily intended for compiled C and Fortran code respectively, but the `.C` function can be used with other languages which can generate C interfaces, for example C++ (see Section 5.6 [Interfacing C++ code], page 136).

The first argument to each function is a character string specifying the symbol name as known[1] 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 atomic vectors and the types of arguments to a C function or Fortran subroutine.

R storage mode	C type	Fortran type
`logical`	`int *`	`INTEGER`
`integer`	`int *`	`INTEGER`
`double`	`double *`	`DOUBLE PRECISION`
`complex`	`Rcomplex *`	`DOUBLE COMPLEX`
`character`	`char **`	`CHARACTER(255)`
`raw`	`unsigned char *`	none

On all R platforms `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, including '`x86_64`' Windows). 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`.[2] (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

[1] possibly after some platform-specific translation, e.g. adding leading or trailing underscores.

[2] This is currently included by `R.h` but may not be in future, so it should be included by code needing the type.

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 of fixed length can be passed to or from Fortran (the length is not passed), and the success of this is compiler-dependent: its use was formally deprecated in 2019. Other R objects can be passed to `.C`, but it is much better to use one of the other interfaces.

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`.) Note that the use of `int *` for Fortran logical is not guaranteed to be portable (although people have gotten away with it for many years): it is better to pass integers and convert to/from Fortran logical in a Fortran wrapper.

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 `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 namespace, the shared object loaded by the first (if any) `useDynLib` directive will be used.)

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 nab = *na + *nb - 1;

    for(int i = 0; i < nab; i++)
        ab[i] = 0.0;
    for(int i = 0; i < *na; i++)
        for(int 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++). 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[3] 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, is deprecated 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 (including on their options). Often what is being passed to Fortran is one of a small set of possible values (a factor in R terms) which could alternatively be passed as an integer code: similarly Fortran code that wants to generate diagnostic messages could pass an integer code to a C or R wrapper which would convert it to a character string.

It is possible to pass some R objects other than atomic vectors *via* `.C`, but this is only supported for historical compatibility: use the `.Call` or `.External` interfaces for such objects. Any C/C++ code that includes `Rinternals.h` should be called *via* `.Call` or `.External`.

`.Fortran` is primarily intended for Fortran 77 code, and long precedes any support for 'modern' Fortran. Nowadays implementations of Fortran support the Fortran 2003 module `iso_c_binding`, a better way to interface modern Fortran code to R is to use `.C` and write a C interface using `use iso_c_binding`.

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

Compiled code to be used with R is loaded as a shared object (Unix-alikes including macOS, see Section 5.5 [Creating shared objects], page 135, 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 and is not safe in general, but it is needed to allow the DLL to be re-built on some platforms, including Windows. Unloading a DLL and then re-loading a DLL of the same name may not work: Solaris used the first version loaded. A DLL that registers C finalizers, but fails to unregister them when unloaded, may cause R to crash after unloading.

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

```
file.path(path1, path2, paste0("mylib", .Platform$dynlib.ext))
```

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

Loading is most often done automatically based on the `useDynLib()` declaration in the `NAMESPACE` file, but may be done explicitly *via* a call to `library.dynam`. This has the form

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

where `libname` is the object/DLL name *with the extension omitted*. Note that the first argument, `chname`, should **not** be `package` since this will not work if the package is installed under another name.

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

[3] Note that this is then not checked for over-runs by option `CBoundsCheck = TRUE`.

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

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

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

Note that there are some implicit restrictions on this mechanism as the basename of the DLL needs to be both a valid file name and valid as part of a C entry point (e.g. it cannot contain '`.`'): for portable code it is best to confine DLL names to be ASCII alphanumeric plus underscore. If entry point `R_init_lib` is not found it is also looked for with '`.`' replaced by '`_`'.

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

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

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

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

If a shared object/DLL is loaded more than once the most recent version is used.[4] More generally, if the same symbol name appears in several shared objects, 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. It is possible 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/x64` directory (on Intel) in the DLL search path.

5.4 Registering native routines

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

[4] Strictly this is OS-specific, but no exceptions have been seen for many years.

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 in all[5] loaded DLLs and the R executable or libraries it is linked to. 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.

Registering routines has two main advantages: it provides a faster[6] way to find the address of the entry point *via* tables stored in the DLL at compilation time, and it provides a run-time check that the entry point is called with the right number of arguments and, optionally, the right argument types.

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 126. `R_registerRoutines` takes 5 arguments. The first is the `DllInfo` object passed by R to the initialization routine. This is where R stores the information about the methods. The remaining 4 arguments are arrays describing the routines for each of the 4 different interfaces: `.C`, `.Call`, `.Fortran` and `.External`. Each argument is a `NULL`-terminated array of the element types given in the following table:

`.C`	`R_CMethodDef`
`.Call`	`R_CallMethodDef`
`.Fortran`	`R_FortranMethodDef`
`.External`	`R_ExternalMethodDef`

Currently, the `R_ExternalMethodDef` type 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 to be passed from R. For example, if we had a routine named `myCall` defined as

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

we would describe this as

```
static const R_CallMethodDef callMethods[]  = {
  {"myCall", (DL_FUNC) &myCall, 3},
  {NULL, NULL, 0}
};
```

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

Routines for use with the `.C` and `.Fortran` interfaces are described with similar data structures, which have one optional additional field for describing the type of each argument. If specified, this field should be an array with the `SEXP` types describing the expected type of each argument of the routine. (Technically, the elements of the types array are of type `R_NativePrimitiveArgType` which is just an unsigned integer.) The R types and corresponding type identifiers are provided in the following table:

`numeric`	`REALSXP`
`integer`	`INTSXP`
`logical`	`LGLSXP`
`single`	`SINGLESXP`

[5] For calls from within a namespace the search is confined to the DLL loaded for that package.

[6] For unregistered entry points the OS's `dlsym` routine is used to find addresses. Its performance varies considerably by OS and even in the best case it will need to search a much larger symbol table than, say, the table of `.Call` entry points.

```
character   STRSXP
list        VECSXP
```

Consider a C routine, `myC`, declared as

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

We would register it as

```
static R_NativePrimitiveArgType myC_type[] = {
    REALSXP, INTSXP, STRSXP, LGLSXP
};

static const R_CMethodDef cMethods[] = {
   {"myC", (DL_FUNC) &myC, 4, myC_type},
   {NULL, NULL, 0, NULL}
};
```

If registering types, check carefully that the number of types matches the number of arguments: as the type array (here `myC_type`) is passed as a pointer in C, the registration mechanism cannot check this for you.

Note that `.Fortran` entry points are mapped to lowercase, so registration should use lowercase only.

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

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

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

When R unloads a shared object/DLL, its registrations are removed. There is no other facility for unregistering a symbol.

Examples of registering routines can be found in the different packages in the R source tree (e.g., **stats** and **graphics**). Also, there is a brief, high-level introduction in *R News* (volume 1/3, September 2001, pages 20–23, `https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf`).

Once routines are registered, they can be referred to as R objects if this is arranged in the `useDynLib` call in the package's `NAMESPACE` file (see Section 1.5.4 [useDynLib], page 51). So for example the **stats** package has

```
# Refer to all C/Fortran routines by their name prefixed by C_
useDynLib(stats, .registration = TRUE, .fixes = "C_")
```

in its `NAMESPACE` file, and then `ansari.test`'s default methods can contain

```
pansari <- function(q, m, n)
    .C(C_pansari, as.integer(length(q)), p = as.double(q),
       as.integer(m), as.integer(n))$p
```

This avoids the overhead of looking up an entry point each time it is used, and ensures that the entry point in the package is the one used (without a `PACKAGE = "pkg"` argument).

`R_init_` routines are often of the form

```
void attribute_visible R_init_mypkg(DllInfo *dll)
```

```
{
    R_registerRoutines(dll, CEntries, CallEntries, FortEntries,
                       ExternalEntries);
    R_useDynamicSymbols(dll, FALSE);
    R_forceSymbols(dll, TRUE);
...
}
```

The `R_useDynamicSymbols` call says the DLL is not to be searched for entry points specified by character strings so `.C` etc calls will only find registered symbols: the `R_forceSymbols` call only allows `.C` etc calls which specify entry points by R objects such as `C_pansari` (and not by character strings). Each provides some protection against accidentally finding your entry points when people supply a character string without a package, and avoids slowing down such searches. (For the visibility attribute see Section 6.16 [Controlling visibility], page 187.)

In more detail, if a package `mypkg` contains entry points `reg` and `unreg` and the first is registered as a 0-argument `.Call` routine, we could use (from code in the package)

```
.Call("reg")
.Call("unreg")
```

Without or with registration, these will both work. If `R_init_mypkg` calls `R_useDynamicSymbols(dll, FALSE)`, only the first will work. If in addition to registration the `NAMESPACE` file contains

```
useDynLib(mypkg, .registration = TRUE, .fixes = "C_")
```

then we can call `.Call(C_reg)`. Finally, if `R_init_mypkg` also calls `R_forceSymbols(dll, TRUE)`, only `.Call(C_reg)` will work (and not `.Call("reg")`). This is usually what we want: it ensures that all of our own `.Call` calls go directly to the intended code in our package and that no one else accidentally finds our entry points. (Should someone need to call our code from outside the package, for example for debugging, they can use `.Call(mypkg:::C_reg)`.)

5.4.1 Speed considerations

Sometimes registering native routines or using a `PACKAGE` argument can make a large difference. The results can depend quite markedly on the OS (and even if it is 32- or 64-bit), on the version of R and what else is loaded into R at the time.

To fix ideas, first consider `x86_64` OS 10.7 and R 2.15.2. A simple `.Call` function might be

```
foo <- function(x) .Call("foo", x)
```

with C code

```
#include <Rinternals.h>

SEXP foo(SEXP x)
{
    return x;
}
```

If we compile with by `R CMD SHLIB foo.c`, load the code by `dyn.load("foo.so")` and run `foo(pi)` it took around 22 microseconds (us). Specifying the DLL by

```
foo2 <- function(x) .Call("foo", x, PACKAGE = "foo")
```

reduced the time to 1.7 us.

Now consider making these functions part of a package whose `NAMESPACE` file uses `useDynlib(foo)`. This immediately reduces the running time as `"foo"` will be preferentially looked for `foo.dll`. Without specifying `PACKAGE` it took about 5 us (it needs to fathom out the appropriate DLL each time it is invoked but it does not need to search all DLLs), and with the `PACKAGE` argument it is again about 1.7 us.

Next suppose the package has registered the native routine `foo`. Then `foo()` still has to find the appropriate DLL but can get to the entry point in the DLL faster, in about 4.2 us. And `foo2()` now takes about 1 us. If we register the symbols in the `NAMESPACE` file and use

```
foo3 <- function(x) .Call(C_foo, x)
```

then the address for the native routine is looked up just once when the package is loaded, and `foo3(pi)` takes about 0.8 us.

Versions using `.C()` rather than `.Call()` took about 0.2 us longer.

These are all quite small differences, but C routines are not uncommonly invoked millions of times for run times of a few microseconds each, and those doing such things may wish to be aware of the differences.

On Linux and Solaris there is a smaller overhead in looking up symbols.

Symbol lookup on Windows used to be far slower, so R maintains a small cache. If the cache is currently empty enough that the symbol can be stored in the cache then the performance is similar to Linux and Solaris: if not it may be slower. R's own code always uses registered symbols and so these never contribute to the cache: however many other packages do rely on symbol lookup.

In more recent versions of R all the standard packages register native symbols and do not allow symbol search, so in a new session `foo()` can only look in `foo.so` and may be as fast as `foo2()`. This will no longer apply when many contributed packages are loaded, and generally those last loaded are searched first. For example, consider R 3.3.2 on x86_64 Linux. In an empty R session, both `foo()` and `foo2()` took about 0.75 us; however after packages **igraph** (`https://CRAN.R-project.org/package=igraph`) and **spatstat** (`https://CRAN.R-project.org/package=spatstat`) had been loaded (which loaded another 12 DLLs), `foo()` took 3.6 us but `foo2()` still took about 0.80 us. Using registration in a package reduced this to 0.55 us and `foo3()` took 0.40 us, times which were unchanged when further packages were loaded.

5.4.2 Example: converting a package to use registration

The **splines** package was converted to use symbol registration in 2001, but we can use it as an example[7] of what needs to be done for a small package.

- Find the relevant entry points. This is somewhat OS-specific, but something like the following should be possible at the OS command-line

  ```
  nm -g /path/to/splines.so | grep " T "
  0000000000002670 T _spline_basis
  0000000000001ec0 T _spline_value
  ```

  This indicates that there are two relevant entry points. (They may or may not have a leading underscore, as here. Fortran entry points will have a trailing underscore.) Check in the R code that they are called by the package and how: in this case they are used by `.Call`.

  Alternatively, examine the package's R code for all `.C`, `.Fortran`, `.Call` and `.External` calls.

- Construct the registration table. First write skeleton registration code, conventionally in file `src/init.c` (or at the end of the only C source file in the package: if included in a C++ file the '`R_init`' function would need to be declared `extern "C"`):

[7] Because it is a standard package, one would need to rename it before attempting to reproduce the account here.

```
#include <stdlib.h> // for NULL
#include <R_ext/Rdynload.h>

#define CALLDEF(name, n)  {#name, (DL_FUNC) &name, n}

static const R_CallMethodDef R_CallDef[] = {
   CALLDEF(spline_basis, ?),
   CALLDEF(spline_value, ?),
   {NULL, NULL, 0}
};

void R_init_splines(DllInfo *dll)
{
    R_registerRoutines(dll, NULL, R_CallDef, NULL, NULL);
}
```

and then replace the `?` in the skeleton with the actual numbers of arguments. You will need to add declarations (also known as 'prototypes') of the functions unless appending to the only C source file. Some packages will already have these in a header file, or you could create one and include it in `init.c`, for example `splines.h` containing

```
#include <Rinternals.h> // for SEXP
extern SEXP spline_basis(SEXP knots, SEXP order, SEXP xvals, SEXP derivs);
extern SEXP spline_value(SEXP knots, SEXP coeff, SEXP order, SEXP x, SEXP deriv);
```

Tools are available to extract declarations, at least for C and C++ code: see the help file for `package_native_routine_registration_skeleton` in package **tools**. Here we could have used

```
cproto -I/path/to/R/include -e splines.c
```

For examples of registering other types of calls, see packages **graphics** and **stats**. In particular, when registering entry points for `.Fortran` one needs declarations as if called from C, such as

```
#include <R_ext/RS.h>
void F77_NAME(supsmu)(int *n, double *x, double *y,
                      double *w, int *iper, double *span, double *alpha,
                      double *smo, double *sc, double *edf);
```

`gfortran` 8.4, 9.2 and later can help generate such prototypes with its flag `-fc-prototypes-external` (although one will need to replace the hard-coded trailing underscore with the `F77_NAME` macro).

One can get away with inaccurate argument lists in the declarations: it is easy to specify the arguments for `.Call` (all `SEXP`) and `.External` (one `SEXP`) and as the arguments for `.C` and `.Fortran` are all pointers, specifying them as `void *` suffices. (For most platforms one can omit all the arguments, although link-time optimization will warn, as will compilers set up to warn on strict prototypes – and C23 will require correct arguments.)

Using `-fc-prototypes-external` will give a prototype using `int_least32_t *lgl` for Fortran `LOGICAL LGL`, but this is not portable and traditionally it has been assumed that the C/C++ equivalent was `int *lgl`. If adding a declaration just to register a `.Fortran` call, the most portable version is `void *lgl`.

- (Optional but highly recommended.) Restrict `.Call` etc to use the symbols you chose to register by editing `src/init.c` to contain

```
void R_init_splines(DllInfo *dll)
{
    R_registerRoutines(dll, NULL, R_CallDef, NULL, NULL);
    R_useDynamicSymbols(dll, FALSE);
}
```

A skeleton for the steps so far can be made using `package_native_routine_registration_skeleton` in package **tools**. This will optionally create declarations based on the usage in the R code.

The remaining steps are optional but recommended.

- Edit the `NAMESPACE` file to create R objects for the registered symbols:

  ```
  useDynLib(splines, .registration = TRUE, .fixes = "C_")
  ```

- Find all the relevant calls in the R code and edit them to use the R objects. This entailed changing the lines

  ```
  temp <- .Call("spline_basis", knots, ord, x, derivs, PACKAGE = "splines")
  y[accept] <- .Call("spline_value", knots, coeff, ord, x[accept], deriv, PACKAGE = "splines")
  y = .Call("spline_value", knots, coef(object), ord, x, deriv, PACKAGE = "splines")
  ```

  to

  ```
  temp <- .Call(C_spline_basis, knots, ord, x, derivs)
  y[accept] <- .Call(C_spline_value, knots, coeff, ord, x[accept], deriv)
  y = .Call(C_spline_value, knots, coef(object), ord, x, deriv)
  ```

  Check that there is no `exportPattern` directive which unintentionally exports the newly created R objects.

- Restrict `.Call` to use the R symbols by editing `src/init.c` to contain

  ```
  void R_init_splines(DllInfo *dll)
  {
      R_registerRoutines(dll, NULL, R_CallDef, NULL, NULL);
      R_useDynamicSymbols(dll, FALSE);
      R_forceSymbols(dll, TRUE);
  }
  ```

- Consider visibility. On some OSes we can hide entry points from the loader, which precludes any possible name clashes and calling them accidentally (usually with incorrect arguments and crashing the R process). If we repeat the first step we now see

  ```
  nm -g /path/to/splines.so | grep " T "
  0000000000002e00 T _R_init_splines
  00000000000025e0 T _spline_basis
  0000000000001e20 T _spline_value
  ```

  If there were any entry points not intended to be used by the package we should try to avoid exporting them, for example by making them `static`. Now that the two relevant entry points are only accessed *via* the registration table, we can hide them. There are two ways to do so on some Unix-alikes. We can hide individual entry points *via*

  ```
  #include <R_ext/Visibility.h>

  SEXP attribute_hidden
  spline_basis(SEXP knots, SEXP order, SEXP xvals, SEXP derivs)
  ...

  SEXP attribute_hidden
  spline_value(SEXP knots, SEXP coeff, SEXP order, SEXP x, SEXP deriv)
  ...
  ```

Alternatively, we can change the default visibility for all C symbols by including

```
PKG_CFLAGS = $(C_VISIBILITY)
```

in `src/Makevars`, and then we need to allow registration by declaring `R_init_splines` to be visible:

```
#include <R_ext/Visibility.h>

void attribute_visible
R_init_splines(DllInfo *dll)
...
```

See Section 6.16 [Controlling visibility], page 187, for more details, including using Fortran code and ways to restrict visibility on Windows.

- We end up with a file `src/init.c` containing

```
#include <stdlib.h>
#include <R_ext/Rdynload.h>
#include <R_ext/Visibility.h>  // optional

#include "splines.h"

#define CALLDEF(name, n)  {#name, (DL_FUNC) &name, n}

static const R_CallMethodDef R_CallDef[] = {
    CALLDEF(spline_basis, 4),
    CALLDEF(spline_value, 5),
    {NULL, NULL, 0}
};

void
attribute_visible  // optional
R_init_splines(DllInfo *dll)
{
    R_registerRoutines(dll, NULL, R_CallDef, NULL, NULL);
    R_useDynamicSymbols(dll, FALSE);
    R_forceSymbols(dll, TRUE);
}
```

5.4.3 Linking to native routines in other packages

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. The interface consists of two routines declared in header `R_ext/Rdynload.h` 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");
```

As the type DL_FUNC is only appropriate for functions with no arguments, other users will need to cast to an appropriate type. For example

```
typedef SEXP (*na_omit_xts_func) (SEXP x);
...
  na_omit_xts_func fun = (na_omit_xts_func) R_GetCCallable("xts", "na_omit_xts");
  return fun(x);
```

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

It must specify[8] 'Imports' or 'Depends' of those packages, for they have to be loaded[9] prior to this one (so the path to their compiled code has been registered).

CRAN examples of the use of this mechanism include **coxme** (https://CRAN.R-project.org/package=coxme) linking to **bdsmatrix** (https://CRAN.R-project.org/package=bdsmatrix) and **xts** (https://CRAN.R-project.org/package=xts) linking to **zoo** (https://CRAN.R-project.org/package=zoo).

NB: this mechanism is fragile, as changes to the interface provided by **packA** have to be recognised by **packB**. The consequences of not doing so have included serious corruption to the memory pool of the R session. Either **packB** has to depend on the exact version of **packA** or there needs to be a mechanism for **packB** to test at runtime the version of **packA** it is linked to matches that it was compiled against.

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, Objective C or Objective C++ (with extensions .c, .cc or .cpp, .f (fixed-form Fortran), .f90 or .f95 (free-form), .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. Note that files intended for the Fortran pre-processor with extension .F are not accepted.

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/C++ preprocessor, mainly '-I', '-D' and '-U' flags), PKG_CFLAGS, PKG_CXXFLAGS, PKG_FFLAGS, PKG_OBJCFLAGS, and PKG_OBJCXXFLAGS (for the C, C++, Fortran, Objective C, 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 take precedence over 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 that are already set (for example in file etc*R_ARCH*/Makeconf) 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
```

[8] whether or not 'LinkingTo' is used.

[9] so there needs to be a corresponding import or importFrom entry in the NAMESPACE file.

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 through `useDynlib` in the `NAMESPACE`, or in the `.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. Since R 4.2.0, `Makevars.ucrt` will be used in preference to `Makevars.win` and `src/Makefile.ucrt` will be used in preference to `src/Makefile.win`. For past experiences of building DLLs with a variety of compilers, see file '`README.packages`'. 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** (`https://CRAN.R-project.org/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 no 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.h` and `X.cpp`, and implementing the two classes `X` and `Y` which we want to use in R.

```
// X.h

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

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

```
// X.cpp

#include <R.h>
#include "X.h"

static Y y;

X::X()  { REprintf("constructor X\n"); }
X::~X() { REprintf("destructor X\n");  }
Y::Y()  { REprintf("constructor Y\n"); }
Y::~Y() { REprintf("destructor Y\n");  }
```

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

```
extern "C" {

}
```

For example,

```
// X_main.cpp:

#include "X.h"

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.cpp X_main.cpp
```

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

```
R version 2.14.1 Patched (2012-01-16 r58124)
Copyright (C) 2012 The R Foundation for Statistical Computing
...
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 Windows.

Earlier versions of this example used C++ iostreams: this 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 170) for all I/O if at all possible. Examples have been seen where merely loading a DLL that contained calls to C++ I/O upset R's own C I/O (for example by resetting buffers on open files).

Most R header files can be included within C++ programs but they should **not** be included within an `extern "C"` block (as they include system headers[10]).

5.6.1 External C++ code

Quite a lot of external C++ software is header-only (e.g. most of the Boost 'libraries' including all those supplied by package **BH** (`https://CRAN.R-project.org/package=BH`), and most of Armadillo as supplied by package **RcppArmadillo** (`https://CRAN.R-project.org/package=RcppArmadillo`)) and so is compiled when an R package which uses it is installed. This causes few problems.

A small number of external libraries used in R packages have a C++ interface to a library of compiled code, e.g. packages **sf** (`https://CRAN.R-project.org/package=sf`) and **rjags** (`https://CRAN.R-project.org/package=rjags`). This raises many more problems! The C++ interface uses name-mangling and the ABI[11] may depend on the compiler, version and even C++ defines[12], so requires the package C++ code to be compiled in exactly the same way as the library (and what that was is often undocumented).

Even fewer external libraries use C++ internally but present a C interface, such as **rgeos** (`https://CRAN.R-project.org/package=rgeos`). These require the C++ runtime library to be linked into the package's shared object/DLL, and this is best done by including a dummy C++ file in the package sources.

There is a trend to link to the C++ interfaces offered by C software such as **hdf5**, **pcre** and **ImageMagick**. Their C interfaces are much preferred for portability (and can be used from C++ code). Also, the C++ interfaces are often optional in the software build or packaged separately and so users installing from package sources are less likely to already have them installed.

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 output to units `*` and `6`. See Section 6.5.1 [Printing from Fortran], page 170, 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`

[10] Even including C system headers in such a block has caused compilation errors.

[11] `https://en.wikipedia.org/wiki/Application_binary_interface`.

[12] For example, '`_GLIBCXX_USE_CXX11_ABI`' in `g++` 5.1 and later: `https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html`.

4 and later. 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 support code is initialized (typically when the package is loaded) the C `stdout` and `stderr` are switched to LF line endings. (Function `init` in file `src/modules/lapack/init_win.c` shows how to mitigate this. In a package this would look something like

```
#ifdef _WIN32
# include <fcntl.h>
#endif

void R_init_mypkgname(DllInfo *dll)
{
    // Native symbol registration calls

#ifdef _WIN32
    // gfortran I/O initialization sets these to _O_BINARY
    setmode(1, _O_TEXT); /* stdout */
    setmode(2, _O_TEXT); /* stderr */
#endif
}
```

in the file used for native symbol registration.)

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 126, 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.

Shipping a static library in package **packB** for other packages to link to avoids most of the difficulties.

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/lib/libpackB.a`. (Note using directory `lib` rather than `libs` is conventional, and architecture-specific sub-directories may be needed and are assumed in the sample code below. The code in the static library 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 **packA**. The only issue is to find package **packB**, and for that we can ask R by something like (long lines broken for display here)

```
PKGB_PATH=`echo 'library(packB);
  cat(system.file("lib",  package="packB", mustWork=TRUE))' \
 | "${R_HOME}/bin/R" --vanilla --no-echo`
PKG_LIBS="$(PKGB_PATH)$(R_ARCH)/libpackB.a"
```

For a dynamic library `packB/lib/libpackB.so` (`packB/lib/libpackB.dylib` on macOS: note that you cannot link to a shared object, `.so`, on that platform) we could use

```
PKGB_PATH=`echo 'library(packB);
```

```
  cat(system.file("lib", package="packB", mustWork=TRUE))' \
 | "${R_HOME}/bin/R" --vanilla --no-echo`
PKG_LIBS=-L"$(PKGB_PATH)$(R_ARCH)" -lpackB
```

This will work for installation, but very likely not when package `packB` is loaded, as the path to package **packB**'s `lib` directory is not in the `ld.so`[13] 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 directory containing 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). On systems with the `gcc` or `clang` and the GNU linker (e.g. Linux) and some others this can be done by e.g.

```
PKGB_PATH=`echo 'library(packB);
  cat(system.file("lib", package="packB", mustWork=TRUE)))' \
 | "${R_HOME}/bin/R" --vanilla --no-echo`
PKG_LIBS=-L"$(PKGB_PATH)$(R_ARCH)" -Wl,-rpath,"$(PKGB_PATH)$(R_ARCH)" -lpackB
```

Some other systems (e.g. Solaris with its native linker) use `-Rdir` rather than `-rpath,dir` (and this is accepted by the compiler as well as the linker).

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 127).

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 127).

- preparing `packA.dll` to link to `packB/libs/exB.dll`.

  This needs an entry in `Makevars.win` or `Makevars.ucrt` 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", mustWork=TRUE))' \
   | rterm --vanilla --no-echo`
  PKG_LIBS= -L"$(PKGB_PATH)$(R_ARCH)" -lexB
  ```

  Another possibility is to use an import library, shipping with package **packA** an exports file `exB.def`. Then `Makevars.win` (or `Makevars.ucrt`) could contain

  ```
  PKG_LIBS= -L. -lexB

  all: $(SHLIB) before
  ```

[13] `dyld` on macOS, and `DYLD_LIBRARY_PATHS` below.

```
before: libexB.dll.a
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

```
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 environment variable `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. However, if a user wants to write C code using internal R data structures, then that can be done using the `.Call` and `.External` functions. The syntax for the calling function in R in each case is similar to that of `.C`, but the two functions have different C interfaces. Generally the `.Call` interface is simpler to use, but `.External` is a little more general.

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

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

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

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

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

A call to `.External` is almost identical

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

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

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

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

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

In each case the R objects are available for manipulation *via* a set of functions and macros defined in the header file `Rinternals.h` or some S-compatibility macros[14] See Section 5.10 [Interface functions .Call and .External], page 152, for details on `.Call` and `.External`.

[14] That is, similar to those defined in S version 4 from the 1990s: these are not kept up to date and are not recommended for new projects.

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 involving only atomic vectors and requiring no call to R, `.C` suffices. A great deal of useful code was written using just `.C` before `.Call` and `.External` were available. These 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.

To handle R objects from within C code we use the macros and functions that have been used to implement the core parts of R. A public[15] 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.

A substantial amount of R, including the standard packages, 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": 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*[16], which is a pointer to a structure with typedef `SEXPREC`. Think of this structure as a *variant type* that can handle all the usual types of R objects, that is vectors of various modes, functions, environments, language objects and so on. The details are given later in this section and in Section "R Internal Structures" in *R Internals*, but for most purposes the programmer does not need to know them. Think rather of a model such as that used by Visual Basic, in which R objects are handed around in C code (as they are in interpreted R code) as the variant type, and the appropriate part is extracted for, for example, numerical calculations, only when it is needed. As in interpreted R code, much use is made of coercion to force the variant object to the right type.

5.9.1 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 or marked as re-usable.

The R object types are represented by a C structure defined by a typedef `SEXPREC` in `Rinternals.h`. It contains several things among which are pointers to data blocks and to other `SEXPREC`s. A `SEXP` is simply a pointer to a `SEXPREC`.

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

Protecting an R object automatically protects all the R objects pointed to in the corresponding `SEXPREC`, for example all elements of a protected list are automatically protected.

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

[15] see Chapter 6 [The R API], page 166: note that these are not all part of the API.

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

Here is a small example of creating an R numeric vector in C code:

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

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

Now, the reader may ask how the R object could possibly get removed during those manipulations, as it is just our C code that is running. As it happens, we can do without the protection in this 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.

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 macro was introduced for use in the parser, where the code interfacing with the R heap is generated and the generator cannot be configured to insert proper calls to `PROTECT` and `UNPROTECT`. However, `UNPROTECT_PTR` is dangerous to use in combination with `UNPROTECT` when the same object has been protected multiple times. It has been superseded by multi-set based functions `R_PreserveInMSet` and `R_ReleaseFromMSet`, which protect objects in a multi-set created by `R_NewPreciousMSet` and typically itself protected using `PROTECT`. These functions should not be needed outside parsers.

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

```
    PROTECT_INDEX ipx;

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

Note that it is dangerous to mix `UNPROTECT_PTR` also with `PROTECT_WITH_INDEX`, as the former changes the protection locations of objects that were protected after the one being unprotected.

There is another way to avoid the effects of garbage collection: a call to `R_PreserveObject` adds an object to an internal list of objects not to be collects, and a subsequent call to `R_ReleaseObject` removes it from that list. This provides a way for objects which are not returned as part of R objects to be protected across calls to compiled code: on the other hand it becomes the user's responsibility to release them when they are no longer needed (and this often requires the use of a finalizer). It is less efficient than the normal protection mechanism, and should be used sparingly.

For functions from packages as well as R to safely co-operate in protecting objects, certain rules have to be followed:

- Pointer-protection balance. Calls to `PROTECT` and `UNPROTECT` should balance in each function. A function may only call `UNPROTECT` or `REPROTECT` on objects it has itself protected. Note that the pointer protection stack balance is restored automatically on non-local transfer of control (See Section 6.12 [Condition handling and cleanup code], page 184.), as if a call to `UNPROTECT` was invoked with the right argument.
- Caller protection. It is the responsibility of the caller that all arguments passed to a function are protected and will stay protected for the whole execution of the callee. Typically this is achieved by `PROTECT` and `UNPROTECT` calls.
- Protecting return values. Any R objects returned from a function are unprotected (the callee must maintain pointer-protection balance), and hence should be protected immediately by the caller. To be safe against future code changes, assume that any R object returned from any function may need protection. Note that even when conceptually returning an existing protected object, that object may be duplicated.
- All functions/macros allocate. To be safe against future code changes, assume that any function or macro may allocate and hence garbage collector may run and destroy unprotected objects.

It is always safe and recommended to follow those rules. In fact, several R functions and macros protect their own arguments and some functions do not allocate or do not allocate when used in a certain way, but that is subject to change, so relying on that may be fragile. `PROTECT` and `PROTECT_WITH_INDEX` can be safely called with unprotected arguments and `UNPROTECT` does not allocate.

5.9.2 Allocating storage

For many purposes it is sufficient to allocate R objects and manipulate those. There are quite a few `allocXxx` functions defined in `Rinternals.h`—you may want to explore them.

One that is commonly used is `allocVector`, the C-level equivalent of R-level `vector()` and its wrappers such as `integer()` and `character()`. One distinction is that whereas the R functions always initialize the elements of the vector, `allocVector` only does so for lists, expressions and character vectors (the cases where the elements are themselves R objects).

If storage is required for C objects during the calculations this is best allocating by calling `R_alloc`; see Section 6.1 [Memory allocation], page 166. 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. 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`	pairlist
`DOTSXP`	a '`...`' object
`NILSXP`	NULL

SYMSXP	name/symbol
CLOSXP	function or function closure
ENVSXP	environment

Among the important internal SEXPTYPEs are LANGSXP, CHARSXP, PROMSXP, etc. (**N.B.**: 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. Other such functions declared in the header file `Rinternals.h` include `isNull`, `isSymbol`, `isLogical`, `isComplex`, `isExpression`, and `isEnvironment`. All of these take a `SEXP` as argument and return 1 or 0 to indicate *TRUE* or *FALSE*.

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

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

Protection is needed as a new *object* is created; the object formerly pointed to by the `SEXP` is still protected but now unused.[17]

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

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

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

5.9.4 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

```
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

[17] If no coercion was required, `coerceVector` would have passed the old object through unchanged.

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

SEXP out(SEXP x, SEXP y)
{
    int nx = length(x), ny = length(y);
    SEXP ans = PROTECT(allocMatrix(REALSXP, nx, ny));
    double *rx = REAL(x), *ry = REAL(y), *rans = REAL(ans);
    for(int i = 0; i < nx; i++) {
        double tmp = rx[i];
        for(int 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. We can use

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

SEXP out(SEXP x, SEXP y)
{
    int nx = length(x), ny = length(y);
    SEXP ans = PROTECT(allocMatrix(REALSXP, nx, ny));
    double *rx = REAL(x), *ry = REAL(y), *rans = REAL(ans);

    for(int i = 0; i < nx; i++) {
      double tmp = rx[i];
      for(int j = 0; j < ny; j++)
        rans[i + nx*j] = tmp * ry[j];
    }

    SEXP dimnames = PROTECT(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(2);
    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 `GetColNames`, `GetRowNames`, `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;
version = PROTECT(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. However, the lookup takes a non-trivial amount of time, so consider code such as

```
static SEXP VerSymbol = NULL;
...
    if (VerSymbol == NULL) VerSymbol = install("version");
```

if it is to be done frequently.

This example can be simplified by another convenience function:

```
SEXP version = PROTECT(ScalarReal(3.0));
setAttrib(ans, install("version"), version);
UNPROTECT(1);
```

5.9.5 Classes

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

```
#include <R.h>
#include <Rinternals.h>
      ....
    SEXP ans, dim, dimnames, class;
      ....
    class = PROTECT(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

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

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

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

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

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

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

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

and enables us to say

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

5.9.7 Handling character data

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

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

You can obtain a `CHARSXP` by calling `mkChar` and providing a NUL-terminated C-style string. This function will return a pre-existing `CHARSXP` if one with a matching string already exists, otherwise it will create a new one and add it to the cache before returning it to you. The variant `mkCharLen` can be used to create a `CHARSXP` from part of a buffer and will ensure null-termination.

Note that R character strings are restricted to `2^31 - 1` bytes, and hence so should the input to `mkChar` be (C allows longer strings on 64-bit platforms).

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)`.

```
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(installChar(STRING_ELT(name, 0)), rho);
    Rprintf("first value is %f\n", REAL(ans)[0]);
    return R_NilValue;
}
```

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

Similar functions with syntax

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

can be used to assign values to R variables. `defineVar` creates a new binding or changes the value of an existing binding in the specified environment frame; it is the analogue of `assign(symbol, value, envir = rho, inherits = FALSE)`, but unlike `assign`, `defineVar` does not make a copy of the object `value`.[18] `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)`.

At times it may also be useful to create a new environment frame in C code. `R_NewEnv` is a C version of the R function `new.env`:

```
SEXP R_NewEnv(SEXP enclos, int hash, int size)
```

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

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

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

To return a length-one real vector we can use

```
double x;

...
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 *`).

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

```
SEXP ScalarReal(double);
SEXP ScalarInteger(int);
SEXP ScalarLogical(int)
SEXP ScalarRaw(Rbyte);
SEXP ScalarComplex(Rcomplex);
SEXP ScalarString(SEXP);
SEXP mkString(const char *);
```

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

```
Rboolean isVector(SEXP);
Rboolean isVectorAtomic(SEXP);
Rboolean isVectorList(SEXP);
Rboolean isMatrix(SEXP);
Rboolean isPairList(SEXP);
Rboolean isPrimitive(SEXP);
Rboolean isTs(SEXP);
Rboolean isNumeric(SEXP);
Rboolean isArray(SEXP);
Rboolean isFactor(SEXP);
Rboolean isObject(SEXP);
Rboolean isFunction(SEXP);
Rboolean isLanguage(SEXP);
Rboolean isNewList(SEXP);
Rboolean isList(SEXP);
Rboolean isOrdered(SEXP);
Rboolean isUnordered(SEXP);
```

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: it 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 `list6` construct a pairlist from one to six 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` find the i-th element and the last element of a pairlist, and `nthcdr` returns a pointer to the n-th position in the pairlist (whose `CAR` is the n-th item).

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

5.9.9.1 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 "workhorses" of their R counterparts.

Functions `any_duplicated` and `any_duplicated3` are fast versions of R's `any(duplicated(.))`.

Function `R_compute_identical` corresponds to R's `identical` function.

5.9.10 Named objects and copying

[The `NAMED` mechanism has been replaced by reference counting.]

When assignments are done in R such as

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

```
y <- x
```

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

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

`0`	The object is not bound to any symbol
`1`	The object has been bound to exactly one symbol
`>= 2`	The object has potentially been bound to two or more symbols, and one should act as if another variable is currently bound to this value. The maximal value is `NAMEDMAX`.

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

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

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

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

which is computed as

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

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

This mechanism was replaced in R 4.0.0. To support future changes, package code should use the macros `MAYBE_REFERENCED`, `MAYBE_SHARED`, and `MARK_NOT_MUTABLE`. These currently correspond to

```
MAYBE_REFERENCED(x)
          NAMED(x) > 0

MAYBE_SHARED(x)
          NAMED(x) > 1

MARK_NOT_MUTABLE(x)
          SET_NAMED(x, NAMEDMAX)
```

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 R's `.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`. The calling function in R is

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

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

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

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

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

5.10.2 Calling `.External`

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

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

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

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

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

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

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

```
first = CADR(args);
second = CADDR(args);
third = CADDDR(args);
fourth = CAD4R(args);
fifth  = CAD5R(args);
```

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

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

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

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

```
SEXP showArgs(SEXP args)
{
    args = CDR(args); /* skip 'name' */
    for(int i = 0; args != R_NilValue; i++, args = CDR(args)) {
        const char *name =
            isNull(TAG(args)) ? "" : CHAR(PRINTNAME(TAG(args)));
        SEXP el = CAR(args);
        if (length(el) == 0) {
            Rprintf("[%d] '%s' R type, length 0\n", i+1, name);
           continue;
        }
        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:
            {
                Rcomplex 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(el, 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.

Additional functions for accessing pairlist components are `CAAR`, `CDAR`, `CDDR`, and `CDDDR`. These components can be modified with `SETCAR`, `SETCDR`, `SETCADR`, `SETCADDR`, `SETCADDDR`, and `SETCAD4R`.

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 IEC 60559 arithmetic will handle them correctly. In the current implementation this is also true of `NA` as it is a type of `NaN`, but it is unwise to rely on such details. Thus we will re-write the code to handle `NA`s using macros defined in `R_ext/Arith.h` included by `R.h`.

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

```
    ...
  for(int i = 0; i < na; i++)
    for(int 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` and `R_NegInf` can be used to set `double`s to the special values.

5.11 Evaluating R expressions from C

The main function we will use is

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

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

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

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

with C code

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

    if(!isNewList(list)) error("'list' must be a list");
    if(!isEnvironment(rho)) error("'rho' should be an environment");
    ans = PROTECT(allocVector(VECSXP, n));
    for(int i = 0; i < n; i++) {
        defineVar(install("x"), VECTOR_ELT(list, i), rho);
        SET_VECTOR_ELT(ans, i, eval(expr, rho));
    }
    setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
    UNPROTECT(1);
    return ans;
}
```

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

```
SEXP lapply2(SEXP list, SEXP fn, SEXP rho)
{
    int 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");
    R_fcall = PROTECT(lang2(fn, R_NilValue));
    ans = PROTECT(allocVector(VECSXP, n));
    for(int i = 0; i < n; i++) {
        SETCADR(R_fcall, VECTOR_ELT(list, i));
        SET_VECTOR_ELT(ans, i, eval(R_fcall, rho));
    }
    setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
    UNPROTECT(2);
    return ans;
}
```

used by

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

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

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

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

Customarily, the evaluation environment is passed from the calling R code (see `rho` above). In special cases it is possible that the C code may need to obtain the current evaluation environment which can be done via `R_GetCurrentEnv()` function.

5.11.1 Zero-finding

In this section we re-work the example of Becker, Chambers & Wilks (1988, pp.˜205–10) on finding a zero of a univariate function. The R code and an example are

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

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

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

```
SEXP mkans(double x)
{
    // no need for PROTECT() here, as REAL(.) does not allocate:
    SEXP ans = allocVector(REALSXP, 1);
    REAL(ans)[0] = x;
    return ans;
}

double feval(double x, SEXP f, SEXP rho)
{
    // a version with (too) much PROTECT()ion .. "better safe than sorry"
    SEXP symbol, value;
    PROTECT(symbol = install("x"));
    PROTECT(value = mkans(x));
    defineVar(symbol, value, rho);
    UNPROTECT(2);
    return(REAL(eval(f, rho))[0]);
}

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

    if(tol <= 0.0) error("non-positive tol value");
    f0 = feval(x0, f, rho); f1 = feval(x1, f, rho);
    if(f0 == 0.0) return mkans(x0);
    if(f1 == 0.0) return mkans(x1);
    if(f0*f1 > 0.0) error("x[0] and x[1] have the same sign");

    for(;;) {
        xc = 0.5*(x0+x1);
        if(fabs(x0-x1) < tol) return  mkans(xc);
        fc = feval(xc, f, rho);
        if(fc == 0) return  mkans(xc);
        if(f0*fc > 0.0) {
            x0 = xc; f0 = fc;
        } else {
            x1 = xc; f1 = fc;
        }
    }
}
```

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

```
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 * max(1, abs(old))
        assign(theta[i], old+delta, envir=rho)
        ans1 <- eval(substitute(expr), rho)
        assign(theta[i], old, envir=rho)
        grad[, i] <- (ans1 - ans)/delta
    }
    attr(ans, "gradient") <- grad
    ans
}
omega <- 1:5; x <- 1; y <- 2
numeric.deriv(sin(omega*x*y), c("x", "y"))
```

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

For the compiled version the call from R will be

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

with example usage

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

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

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

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

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

    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");

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

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

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

The code to handle the arguments is

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

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

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

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

We then allocate space for the calculated derivative by

```
    gradient = PROTECT(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. (Note that `LENGTH` is intended to be used for vectors: `length` is more generally applicable.)

```
    for(i = 0, start = 0; i < LENGTH(theta); i++, start += LENGTH(ans)) {
        par = PROTECT(findVar(installChar(STRING_ELT(theta, i)), rho));
```

Here, we are entering a for loop. We loop through each of the variables. In the `for` loop, we first create a symbol corresponding to the `i`-th element of the `STRSXP theta`. Here, `STRING_ELT(theta, i)` accesses the `i`-th element of the `STRSXP theta`. Macro `CHAR()` extracts the actual character representation[19] of it: it returns a pointer. We then install the name and use `findVar` to find its value.

[19] see Section 5.15 [Character encoding issues], page 164, for why this might not be what is required.

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

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

```
        for(int 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).

```
    dimnames = PROTECT(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`[20].

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

[20] 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;
    ParseStatus status;
   ...
    if(done == 1) {
        cmdSexp = PROTECT(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(int 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 length one character vector to be used as a filename in error messages, a `srcfile` object or the R `NULL` object (as in the example above). If a `srcfile` object was used, 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.

5.12.1 Accessing source references

The source references added by the parser are recorded by R's evaluator as it evaluates code. Two functions make these available to debuggers running C code:

```
SEXP R_GetCurrentSrcref(int skip);
```

This function checks `R_Srcref` and the current evaluation stack for entries that contain source reference information. The `skip` argument tells how many source references to skip before returning the `SEXP` of the `srcref` object, counting from the top of the stack. If `skip < 0`, `abs(skip)` locations are counted up from the bottom of the stack. If too few or no source references are found, `NULL` is returned.

```
SEXP R_GetSrcFilename(SEXP srcref);
```

This function extracts the filename from the source reference for display, returning a length 1 character vector containing the filename. If no name is found, `""` is returned.

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** (https://CRAN.R-project.org/package=RODBC) for example. They are unusual in their copying semantics in that when an R object is copied, the external pointer object is not duplicated. (For this reason external pointers should only be used as part of an object with normal semantics, for example an attribute or an element of a list.)

An external pointer is created by

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

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

An alternative way to create an external pointer from a function pointer is

```
typedef void * (*R_DL_FUNC)();
SEXP R_MakeExternalPtrFn(R_DL_FUNC p, SEXP tag, SEXP prot);
```

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

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

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

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

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

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

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

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

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

Finalizers can be run at many places in the code base and much of it, including the R interpreter, is not re-entrant. So great care is needed in choosing the code to be run in a

finalizer. Finalizers are marked to be run at garbage collection but only run at a somewhat safe point thereafter.

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

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

Weak reference objects are created by one of

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

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

The parts can be accessed *via*

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

A toy example of the use of weak references can be found at `https://homepage.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** (`https://CRAN.R-project.org/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 `thisHandle` 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

```
SEXP ans, ptr;
ans = PROTECT(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 end 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

```
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 is use.

5.14 Vector accessor functions

The vector accessors like `REAL`, `INTEGER`, `LOGICAL`, `RAW`, `COMPLEX`, and `VECTOR_ELT` are *functions* when used in R extensions. (For efficiency they may be macros or inline functions 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`.

Formerly it was possible for packages to obtain internal versions of some accessors by defining '`USE_RINTERNALS`' before including `Rinternals.h`. This is no longer the case. Defining '`USE_RINTERNALS`' now has no effect.

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 `CHARSXP`s as a whole. However, if they need to be interpreted as characters or output at C level then it would normally be correct to ensure that they are converted to the encoding of the current locale: this can be done by accessing the data in the `CHARSXP` by `translateChar` rather than by `CHAR`. If re-encoding is needed this allocates memory with `R_alloc` which thus persists to the end of the `.Call`/`.External` call unless `vmaxset` is used (see Section 6.1.1 [Transient storage allocation], page 166).

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).

Both `translateChar` and `translateCharUTF8` will translate any input, using escapes such as '`<A9>`' and '`<U+0093>`' to represent untranslatable parts of the input.

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

```
typedef enum {CE_NATIVE, CE_UTF8, CE_LATIN1, CE_BYTES, CE_SYMBOL, CE_ANY} cetype_t;
cetype_t getCharCE(SEXP);
SEXP mkCharCE(const char *, cetype_t);
```

Only `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_BYTES` is used to make `CHARSXP`s which should be regarded as a set of bytes and not translated. 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. (See the comments under `mkChar` about the length of input allowed.)

Function

```
const char *reEnc(const char *x, 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

```
#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.

Most of these header files, including all those included by `R.h`, can be used from C++ code.

Note: Because R re-maps many of its external names to avoid clashes with system or 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` (before including any R headers) and prepending '`Rf_`' to *all* the function names used from `Rinternals.h` and `R_ext/Error.h`. These problems can usually be avoided by including other headers (such as system headers and those for external software used by the package) before any R headers. (Headers from other packages may include R headers directly or *via* inclusion from further packages, and may define `R_NO_REMAP` with or without including `Rinternals.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).

These functions are declared in header `R_ext/RS.h` which is included by `R.h`.

6.1.1 Transient storage allocation

Here R will reclaim the memory at the end of the call to `.C`, `.Call` or `.External`. Use

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

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

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

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

[1] Known problems have been defining `LENGTH`, `error`, `length`, `match`, `vector` and `warning`: whether these matter depends on the OS and toolchain, with many problem reports involving `clang++`.

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

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

and

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

which (for *new* > *old*) changes the allocation size from *old* to *new* units, and zeroes the additional units. NB: these calls are best avoided as `long` is insufficient for large memory allocations on 64-bit Windows (where it is limited to 2^31-1 bytes).

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 subsequently clearing memory allocated by a call to `vmaxset`. An example might be

```
void *vmax = vmaxget()
// a loop involving the use of R_alloc at each iteration
vmaxset(vmax)
```

This is only recommended for experts.

Note that this memory will be freed on error or user interrupt (if allowed: see Section 6.13 [Allowing interrupts], page 185).

The memory returned is only guaranteed to be aligned as required for `double` pointers: take precautions if casting to a pointer which needs more. There is also

```
long double *R_allocLD(size_t n)
```

which is guaranteed to have the 16-byte alignment needed for `long double` pointers on some platforms.

These functions should only be used in code called by `.C` etc, never from front-ends. They are not thread-safe.

6.1.2 User-controlled memory

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

The interface macros are

```
type* R_Calloc(size_t n, type)
type* R_Realloc(any *p, size_t n, type)
void R_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 return the memory has been successfully allocated or freed. `R_Free` will set the pointer *p* to `NULL`. (Some but not all versions of S did so.)

Users should arrange to `R_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 `R_Calloc`/`R_Realloc` comes from the same pool as used by `malloc`:[2] in particular do not use `free` or `strdup` with it.

Memory obtained by these macros should be aligned in the same way as `malloc`, that is 'suitably aligned for any kind of variable'.

[2] That was not the case on Windows prior to R 4.2.0.

Historically the macros `Calloc`, `Free` and `Realloc` were used, and these remain available unless `STRICT_R_HEADERS` was defined prior to the inclusion of the header.

```
char * CallocCharBuf(size_t n)
void * Memcpy(q, p, n)
void * Memzero(p, n)
```

`CallocCharBuf(n)` is shorthand for `R_Calloc(n+1, char)` to allow for the `nul` terminator. `Memcpy` and `Memzero` take `n` items from array `p` and copy them to array `q` or zero them respectively.

6.2 Error signaling

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

```
void error(const char * format, ...);
void warning(const char * format, ...);
void errorcall(SEXP call, const char * format, ...);
void warningcall(SEXP call, const char * format, ...);
void warningcall_immediate(SEXP call, 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.)

These are defined in header `R_ext/Error.h` included by `R.h`.

6.2.1 Error signaling from Fortran

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

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

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

6.3 Random number generation

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

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

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

```
GetRNGstate();
```

and after all the required variates have been generated, call

```
PutRNGstate();
```

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

These are defined in header `R_ext/Random.h`.

The random number generator is private to R; there is no way to select the kind of RNG nor 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 175. Those calls should also be preceded and followed by calls to `GetRNGstate` and `PutRNGstate`.

6.3.1 Random-number generation from Fortran

It was explained earlier that Fortran random-number generators should not be used in R packages, not least as packages cannot safely initialize them. Rather a package should call R's built in generators: one way to do so is to use C wrappers like

```
#include <R_ext/RS.h>
#include <R_ext/Random.h>

void F77_SUB(getRNGseed)(void) {
    GetRNGstate();
}
void F77_SUB(putRNGseed)(void) {
    PutRNGstate();
}
double F77_SUB(unifRand)(void) {
    return(unif_rand());
}
```

called from Fortran code like

```
      ...
      double precision X
      call getRNGseed()
      X = unifRand()
      ...
      call putRNGseed()
```

Alternatively one could use Fortran 2003's `iso_c_binding` module by something like (fixed-form Fortran 90 code):

```
      module rngfuncs
        use iso_c_binding
        interface
          double precision
     *      function unifRand() bind(C, name = "unif_rand")
          end function unifRand

          subroutine getRNGseed() bind(C, name = "GetRNGstate")
          end subroutine getRNGseed

          subroutine putRNGseed() bind(C, name = "PutRNGstate")
          end subroutine putRNGseed
        end interface
      end module

      subroutine testit
      use rngfuncs
      double precision X
      call getRNGseed()
      X = unifRand()
      print *, X
      call putRNGSeed()
      end
```

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 some 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.

These are defined in header `R_ext/Arith.h` included by `R.h`.

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, and can be re-directed by `sink`). 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 `REvprintf` are analogues using the `vprintf` interface. Because that is a C99[3] interface, they are only defined by `R_ext/Print.h` in C++ code if the macro `R_USE_C99_IN_CXX` is defined before it is included or (as from R 4.0.0) a C++11 compiler is used.

Another circumstance when it may be important to use these functions is when using parallel computation on a cluster of computational nodes, as their output will be re-directed/logged appropriately.

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 may be invisible on GUI interfaces. They are not portable and best avoided.

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

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

and from R 4.0.0,

```
subroutine labelpr(label, nchar)
subroutine dblepr1(label, nchar, var)
subroutine realpr1(label, nchar, var)
subroutine intpr1 (label, nchar, var)
```

[3] also part of C++11.

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), *data* is an array of length at least *ndata* of the appropriate type (`double precision`, `real` and `integer` respectively) and *var* is a (scalar) variable. These routines print the label on one line and then print *data* or *var* as if it were an R vector on subsequent line(s). Note that some compilers will give an error or warning unless *data* is an array: others will accept a scalar when *ndata* has value one or zero. **NB:** There is no check on the type of *data* or *var*, so using `real` (including a real constant) instead of `double precision` will give incorrect answers.

`intpr` works with zero *ndata* so can be used to print a label in earlier versions of R.

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[4] 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 containing underscores were 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.)

[4] The '`F77_`' in the names is historical and dates back to usage in S.

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

Where supported, *link time optimization* provides a reliable way to check the consistency of calls to C from Fortran or *vice versa*. See Section 4.5 [Using Link-time Optimization], page 122. One place where this occurs is the registration of `.Fortran` calls in C code (see Section 5.4 [Registering native routines], page 127). For example

```
init.c:10:13: warning: type of 'vsom_' does not match original
 declaration [-Wlto-type-mismatch]
  extern void F77_NAME(vsom)(void *, void *, void *, void *,
    void *, void *, void *, void *, void *);
vsom.f90:20:33: note: type mismatch in parameter 9
   subroutine vsom(neurons,dt,dtrows,dtcols,xdim,ydim,alpha,train)
vsom.f90:20:33: note: 'vsom' was previously declared here
```

shows that a subroutine has been registered with 9 arguments (as that is what the `.Fortran` call used) but only has 8.

6.6.1 Fortran character strings

Passing character strings from C to Fortran or *vice versa* is not portable, but can be done with care. The internal representations are different: a character array in C (or C++) is NUL-terminated so its length can be computed by `strlen`. Fortran character arrays are typically stored as an array of bytes and a length. This matters when passing strings from C to Fortran or *vice versa*: in many cases one has been able to get away with passing the string but not the length. However, in 2019 this changed for `gfortran`, starting with version 9 but backported to versions 7 and 8. Several months later, `gfortran` 9.2 introduced an option

```
-ftail-call-workaround
```

and made it the current default but said it might be withdrawn in future.

Suppose we want a function to report a message from Fortran to R's console (one could use `labelpr`, or `intpr` with dummy data, but this might be the basis of a custom reporting function). Suppose the equivalent in Fortran would be

```
      subroutine rmsg(msg)
      character*(*) msg
      print *,msg
      end
```

in file `rmsg.f`. Using `gfortran` 9.2 and later we can extract the C view by

```
gfortran -c -fc-prototypes-external rmsg.f
```

which gives

```
void rmsg_ (char *msg, size_t msg_len);
```

(where `size_t` applies to version 8 and later). We could re-write that portably in C as

```
#ifndef USE_FC_LEN_T
# define USE_FC_LEN_T
#endif
#include <Rconfig.h> // included by R.h, so define USE_FC_LEN_T early

void F77_NAME(rmsg)(char *msg, FC_LEN_T msg_len)
{
    char cmsg[msg_len+1];
    strncpy(cmsg, msg, msg_len);
    cmsg[msg_len] = '\0'; // nul-terminate the string, to be sure
    // do something with 'cmsg'
}
```

in code depending on `R(>= 3.6.2)`. For earlier versions of R we could just assume that `msg` is NUL-terminated (not guaranteed, but people have been getting away with it for many years), so the complete C side might be

```
#ifndef USE_FC_LEN_T
# define USE_FC_LEN_T
#endif
#include <Rconfig.h>

#ifdef FC_LEN_T
void F77_NAME(rmsg)(char *msg, FC_LEN_T msg_len)
{
    char cmsg[msg_len+1];
    strncpy(cmsg, msg, msg_len);
    cmsg[msg_len] = '\0';
    // do something with 'cmsg'
}
#else
void F77_NAME(rmsg)(char *msg)
{
    // do something with 'msg'
}
#endif
```

(`USE_FC_LEN_T` is the default as from R 4.3.0.)

An alternative is to use Fortran 2003 features to set up the Fortran routine to pass a C-compatible character string. We could use something like

```
module cfuncs
  use iso_c_binding, only: c_char, c_null_char
  interface
    subroutine cmsg(msg) bind(C, name = 'cmsg')
      use iso_c_binding, only: c_char
      character(kind = c_char):: msg(*)
    end subroutine cmsg
  end interface
end module

subroutine rmsg(msg)
  use cfuncs
  character(*) msg
  call cmsg(msg//c_null_char) ! need to concatenate a nul terminator
end subroutine rmsg
```

where the C side is simply

```
void cmsg(const char *msg)
{
    // do something with nul-terminated string 'msg'
}
```

If you use `bind` to a C function as here, the only way to check that the bound definition is correct is to compile the package with LTO (which requires compatible C and Fortran compilers, usually `gcc` and `gfortran`).

Passing a variable-length string from C to Fortran is trickier, but `https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2023-0/`

`bind-c.html` provides a recipe. However, all the uses in BLAS and LAPACK are of a single character, and for these we can write a wrapper in Fortran along the lines of

```
      subroutine c_dgemm(transa, transb, m, n, k, alpha,
     +     a, lda, b, ldb, beta, c, ldc)
     +     bind(C, name = 'Cdgemm')
        use iso_c_binding, only : c_char, c_int, c_double
        character(c_char), intent(in) :: transa, transb
        integer(c_int), intent(in) :: m, n, k, lda, ldb, ldc
        real(c_double), intent(in) :: alpha, beta, a(lda, *), b(ldb, *)
        real(c_double), intent(out) ::  c(ldc, *)
        call dgemm(transa, transb, m, n, k, alpha,
     +             a, lda, b, ldb, beta, c, ldc)
      end subroutine c_dgemm
```

which is then called from C with declaration

```
void
Cdgemm(const char *transa, const char *transb, const int *m,
       const int *n, const int *k, const double *alpha,
       const double *a, const int *lda, const double *b, const int *ldb,
       const double *beta, double *c, const int *ldc);
```

Alternatively, do as R does as from version 3.6.2 and pass the character length(s) from C to Fortran. A portable way to do this is

```
// before any R headers, or define in PKG_CPPFLAGS
#ifndef  USE_FC_LEN_T
# define USE_FC_LEN_T
#endif
#include <Rconfig.h>
#include <R_ext/BLAS.h>
#ifndef FCONE
# define FCONE
#endif
...
        F77_CALL(dgemm)("N", "T", &nrx, &ncy, &ncx, &one, x,
                        &nrx, y, &nry, &zero, z, &nrx FCONE FCONE);
```

(Note there is no comma before or between the `FCONE` invocations.) It is strongly recommended that packages which call from C/C++ BLAS/LAPACK routines with character arguments adopt this approach: packages not using will fail to install as from R 4.3.0.

6.6.2 Fortran LOGICAL

Passing Fortran LOGICAL variables to/from C/C++ is potentially compiler-dependent. Fortran compilers have long used a 32-bit integer type so it is pretty portable to use `int *` on the C/C++ side. However, recent versions of `gfortran` *via* the option `-fc-prototypes-external` say the C equivalent is `int_least32_t *`: 'Link-Time Optimization' will report `int *` as a mismatch. It is possible to use `iso_c_binding` in Fortran 2003 to map LOGICAL variables to the C99 type `_Bool`, but it is usually simpler to pass integers to and fro.

6.6.3 Passing functions

A number of packages call C functions passed as arguments to Fortran code along the lines of

```
c         subroutine fcn(m,n,x,fvec,iflag)
c         integer m,n,iflag
c         double precision x(n),fvec(m)
```

```
...
      subroutine lmdif(fcn, ...
```

where the C declaration and call are

```
void fcn_lmdif(int *m, int *n, double *par, double *fvec, int *iflag);

void F77_NAME(lmdif)(void (*fcn_lmdif)(int *m, int *n, double *par,
                                       double *fvec, int *iflag), ...

F77_CALL(lmdif)(&fcn_lmdif, ...
```

This works on most platforms but depends on the C and Fortran compilers agreeing on calling conventions: this have been seen to fail. The most portable solution seems to be to convert the Fortran code to C, perhaps using `f2c`.

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 linear algebra functions included in R. These are expressed as calls to Fortran subroutines, and they will also be usable from users' Fortran code. Although not part of the official API, this set of subroutines is unlikely to change (but might be supplemented).

The header file `Rmath.h` lists many other functions that are available and documented in the following subsections. Many of these are C interfaces to the code behind R functions, so the R function documentation may give further details.

6.7.1 Distribution functions

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

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

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

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

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

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

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

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

The random-variate generation routine `rnorm` returns one normal variate. See Section 6.3 [Random numbers], page 168, 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`.

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

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

Entries marked with an asterisk only have '`p`' and '`q`' functions available, and none of the non-central distributions have '`r`' functions.

(If remapping is suppressed, the Normal distribution names are `Rf_dnorm4`, `Rf_pnorm5` and `Rf_qnorm5`.)

Additionally, a *multivariate* RNG for the multinomial distribution is

```
void rmultinom(int n, double* prob, int K, int* rN)
```

where `K = length(prob)`, $\pi :=$ `prob[]`, $\sum_{j=1}^{K} \pi_j = 1$ and `rN` must point to a length-`K` integer vector $n_1 n_2 \dots n_K$ where each entry $n_j =$`rN[j]` is "filled" by a random binomial from $Bin(n; \pi_j)$, constrained to $\sum_{j=1}^{K} n_j = n$.

After calls to `dwilcox`, `pwilcox` or `qwilcox` the function `wilcox_free()` should be called, and similarly `signrank_free()` for the signed rank functions. Since `wilcox_free()` and `signrank_free()` were only added to `Rmath.h` in R 4.2.0, their use requires something like

```
#include "Rmath.h"
#include "Rversion.h"

#if R_VERSION < R_Version(4, 2, 0)
extern void wilcox_free(void);
extern void signrank_free(void);
#endif
```

For the negative binomial distribution ('nbinom'), in addition to the (size, prob) parametrization, the alternative (size, mu) parametrization is provided as well by functions '[dpqr]nbinom_mu()', see *?NegBinomial* in R.

Functions dpois_raw(x, *) and dbinom_raw(x, *) are versions of the Poisson and binomial probability mass functions which work continuously in x, whereas dbinom(x,*) and dpois(x,*) only return non zero values for integer x.

```
double dbinom_raw(double x, double n, double p, double q, int give_log)
double dpois_raw (double x, double lambda, int give_log)
```

Note that dbinom_raw() returns both p and $q = 1 - p$ which may be advantageous when one of them is close to 1.

6.7.2 Mathematical functions

`double gammafn (double x)` [Function]
`double lgammafn (double x)` [Function]
`double digamma (double x)` [Function]
`double trigamma (double x)` [Function]
`double tetragamma (double x)` [Function]
`double pentagamma (double x)` [Function]
`double psigamma (double x, double deriv)` [Function]
`void dpsifn (double x, int n, int kode, int m, double* ans, int* nz, int* ierr)` [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. The underlying workhorse, dpsifn(), is useful, e.g., when several derivatives of $\log\Gamma$ =lgammafn are desired. It computes and returns in ans[] the length-m sequence $(-1)^{k+1}/\Gamma(k+1) * \psi^{(k)}(x)$ for $k = n \ldots n+m-1$, where $\psi^{(k)}(x)$ is the k-th derivative of $\psi(x)$, i.e., psigamma(x,k). For more details, see the comments in src/nmath/polygamma.c.

`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 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]

`double pow1p (double x, double y)` [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`.

`pow1p(x, y)` computes `(1 + x)^y`, accurately even for small x, i.e., $|x| \ll 1$.

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

This should 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 (except under C++, so it may not be declared for C++98).

`double log1pmx (double x)` [Function]
Computes `log(1 + x) - x` (*log 1* **p***lus x* **m***inus* **x**), accurately even for small x, i.e., $|x| \ll 1$.

`double log1pexp (double x)` [Function]
Computes `log(1 + exp(x))` (*log 1* **p***lus* **exp**), accurately, notably for large x, e.g., $x > 720$.

`double log1mexp (double x)` [Function]
Computes `log(1 - exp(-x))` (*log 1* **m***inus* **exp**), accurately, carefully for two regions of x, optimally cutting off at $\log 2$ ($= 0.693147..$), using `((-x) > -M_LN2 ? log(-expm1(-x)) : log1p(-exp(-x)))`.

`double expm1 (double x)` [Function]
Computes `exp(x) - 1` (*exp x* **m***inus 1*), accurately even for small x, i.e., $|x| \ll 1$.

This should 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 (except under C++, so it may not be declared for C++98).

`double lgamma1p (double x)` [Function]
Computes `log(gamma(x + 1))` (*log(gamma(1* **p***lus x))*), accurately even for small x, i.e., $0 < x < 0.5$.

`double cospi (double x)` [Function]
Computes `cos(pi * x)` (where `pi` is 3.14159...), accurately, notably for half integer x.

This might be provided by your platform[5], in which case it is not included in `Rmath.h`, but is in `math.h` which `Rmath.h` includes. (Ensure that neither `math.h` nor `cmath` is included before `Rmath.h` or define

```
#define __STDC_WANT_IEC_60559_FUNCS_EXT__ 1
```

before the first inclusion.)

`double sinpi (double x)` [Function]
Computes `sin(pi * x)` accurately, notably for (half) integer x.

This might be provided by your platform, in which case it is not included in `Rmath.h`, but is in `math.h` which `Rmath.h` includes (but see the comments for `cospi`).

`double Rtanpi (double x)` [Function]
Computes `tan(pi * x)` accurately, notably for integer x, giving *NaN* for half integer x and exactly +1 or -1 for (non half) quarter integers.

[5] It is an optional C11 extension.

`double tanpi (double x)` [Function]
Computes `tan(pi * x)` accurately for integer x with possibly platform dependent behavior for half (and quarter) integers. This might be provided by your platform, in which case it is not included in `Rmath.h`, but is in `math.h` which `Rmath.h` includes (but see the comments for `cospi`).

`double logspace_add (double logx, double logy)` [Function]
`double logspace_sub (double logx, double logy)` [Function]
`double logspace_sum (const double* logx, int n)` [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))`, and "sum_i x[i]" as `log (sum[i = 1:n exp(logx[i])] )` without causing unnecessary overflows or throwing away too much accuracy.

`int imax2 (int x, int y)` [Function]
`int imin2 (int x, int y)` [Function]
`double fmax2 (double x, double y)` [Function]
`double fmin2 (double x, double y)` [Function]
Return the larger (`max`) or smaller (`min`) of two integer or double numbers, respectively. Note that `fmax2` and `fmin2` differ from C99/C++11's `fmax` and `fmin` when one of the arguments is a `NaN`: these versions return `NaN`.

`double sign (double x)` [Function]
Compute the *signum* function, where sign(x) is 1, 0, or -1, when x is positive, 0, or negative, respectively, and `NaN` if `x` is a `NaN`.

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

`double fprec (double x, double digits)` [Function]
Returns the value of x rounded to *digits* decimal digits (after the decimal point).

This is the function used by R's `signif()`.

`double fround (double x, double digits)` [Function]
Returns the value of x rounded to *digits significant* decimal digits.

This is the function used by R's `round()`. (Note that C99/C++11 provide a `round` function but C++98 need not.)

`double ftrunc (double x)` [Function]
Returns the value of x truncated (to an integer value) towards zero.

6.7.4 Mathematical constants

R has a set of commonly used mathematical constants encompassing constants defined by POSIX and usually found in headers `math.h` and `cmath`, as well as further ones that are used in statistical computations. These are defined to (at least) 30 digits accuracy in `Rmath.h`. The following definitions use `ln(x)` for the natural logarithm (`log(x)` in R).

Name	**Definition** (`ln` = `log`)	**round**(*value*, **7**)
`M_E`	e	2.7182818
`M_LOG2E`	$\log2(e)$	1.4426950
`M_LOG10E`	$\log10(e)$	0.4342945
`M_LN2`	$\ln(2)$	0.6931472
`M_LN10`	$\ln(10)$	2.3025851
`M_PI`	π	3.1415927

M_PI_2	$\pi/2$	1.5707963
M_PI_4	$\pi/4$	0.7853982
M_1_PI	$1/\pi$	0.3183099
M_2_PI	$2/\pi$	0.6366198
M_2_SQRTPI	$2/\mathrm{sqrt}(\pi)$	1.1283792
M_SQRT2	$\mathrm{sqrt}(2)$	1.4142136
M_SQRT1_2	$1/\mathrm{sqrt}(2)$	0.7071068
M_SQRT_3	$\mathrm{sqrt}(3)$	1.7320508
M_SQRT_32	$\mathrm{sqrt}(32)$	5.6568542
M_LOG10_2	$\log10(2)$	0.3010300
M_2PI	2π	6.2831853
M_SQRT_PI	$\mathrm{sqrt}(\pi)$	1.7724539
M_1_SQRT_2PI	$1/\mathrm{sqrt}(2\pi)$	0.3989423
M_SQRT_2dPI	$\mathrm{sqrt}(2/\pi)$	0.7978846
M_LN_SQRT_PI	$\ln(\mathrm{sqrt}(\pi))$	0.5723649
M_LN_SQRT_2PI	$\ln(\mathrm{sqrt}(2\pi))$	0.9189385
M_LN_SQRT_PId2	$\ln(\mathrm{sqrt}(\pi/2))$	0.2257914

There are a set of constants (`PI`, `DOUBLE_EPS`) (and so on) defined (unless `STRICT_R_HEADERS` is defined) in the included header `R_ext/Constants.h`, mainly for compatibility with S. All but `PI` are deprecated and should be replaced by the C99/C++11 versions used in that file.

Further, the included header `R_ext/Boolean.h` has enumeration constants `TRUE` and `FALSE` of type `Rboolean` in order to provide a way of using "logical" variables in C consistently. This can conflict with other software: for example it conflicts with the headers in IJG's `jpeg-9` (but not earlier versions).

6.8 Optimization

The C code underlying `optim` can be accessed directly. The user needs to supply a function to compute the function to be minimized, of the type

```
typedef double optimfn(int n, double *par, void *ex);
```

where the first argument is the number of parameters in the second argument. The third argument is a pointer passed down from the calling routine, normally used to carry auxiliary information.

Some of the methods also require a gradient function

```
typedef void optimgr(int n, double *par, double *gr, void *ex);
```

which passes back the gradient in the `gr` argument. No function is provided for finite-differencing, nor for approximating the Hessian at the result.

The interfaces (defined in header `R_ext/Applic.h`) are

- Nelder Mead:

```
void nmmin(int n, double *xin, double *x, double *Fmin, optimfn fn,
           int *fail, double abstol, double intol, void *ex,
           double alpha, double beta, double gamma, int trace,
           int *fncount, int maxit);
```

- BFGS:

```
void vmmin(int n, double *x, double *Fmin,
           optimfn fn, optimgr gr, int maxit, int trace,
           int *mask, double abstol, double reltol, int nREPORT,
           void *ex, int *fncount, int *grcount, int *fail);
```

- Conjugate gradients:

```
void cgmin(int n, double *xin, double *x, double *Fmin,
           optimfn fn, optimgr gr, int *fail, double abstol,
           double intol, void *ex, int type, int trace,
           int *fncount, int *grcount, int maxit);
```

- Limited-memory BFGS with bounds:

```
void lbfgsb(int n, int lmm, double *x, double *lower,
            double *upper, int *nbd, double *Fmin, optimfn fn,
            optimgr gr, int *fail, void *ex, double factr,
            double pgtol, int *fncount, int *grcount,
            int maxit, char *msg, int trace, int nREPORT);
```

- Simulated annealing:

```
void samin(int n, double *x, double *Fmin, optimfn fn, int maxit,
           int tmax, double temp, int trace, void *ex);
```

Many of the arguments are common to the various methods. `n` is the number of parameters, `x` or `xin` is the starting parameters on entry and `x` the final parameters on exit, with final value returned in `Fmin`. Most of the other parameters can be found from the help page for `optim`: see the source code `src/appl/lbfgsb.c` for the values of `nbd`, which specifies which bounds are to be used.

6.9 Integration

The C code underlying `integrate` can be accessed directly. The user needs to supply a *vectorizing* C function to compute the function to be integrated, of the type

```
typedef void integr_fn(double *x, int n, void *ex);
```

where `x[]` is both input and output and has length `n`, i.e., a C function, say `fn`, of type `integr_fn` must basically do `for(i in 1:n) x[i] := f(x[i], ex)`. The vectorization requirement can be used to speed up the integrand instead of calling it `n` times. Note that in the current implementation built on QUADPACK, `n` will be either 15 or 21. The `ex` argument is a pointer passed down from the calling routine, normally used to carry auxiliary information.

There are interfaces (defined in header `R_ext/Applic.h`) for integrals over finite and infinite intervals (or "ranges" or "integration boundaries").

- Finite:

```
void Rdqags(integr_fn f, void *ex, double *a, double *b,
            double *epsabs, double *epsrel,
            double *result, double *abserr, int *neval, int *ier,
            int *limit, int *lenw, int *last,
            int *iwork, double *work);
```

- Infinite:

```
void Rdqagi(integr_fn f, void *ex, double *bound, int *inf,
            double *epsabs, double *epsrel,
            double *result, double *abserr, int *neval, int *ier,
            int *limit, int *lenw, int *last,
            int *iwork, double *work);
```

Only the 3rd and 4th argument differ for the two integrators; for the finite range integral using `Rdqags`, `a` and `b` are the integration interval bounds, whereas for an infinite range integral using `Rdqagi`, `bound` is the finite bound of the integration (if the integral is not doubly-infinite) and `inf` is a code indicating the kind of integration range,

`inf = 1` corresponds to (bound, +Inf),

`inf = -1` corresponds to (-Inf, bound),

`inf = 2` corresponds to (-Inf, +Inf),

`f` and `ex` define the integrand function, see above; `epsabs` and `epsrel` specify the absolute and relative accuracy requested, `result`, `abserr` and `last` are the output components `value`, `abs.err` and `subdivisions` of the R function integrate, where `neval` gives the number of integrand function evaluations, and the error code `ier` is translated to R's `integrate() $ message`, look at that function definition. `limit` corresponds to `integrate(..., subdivisions = *)`. It seems you should always define the two work arrays and the length of the second one as

```
lenw = 4 * limit;
iwork =   (int *) R_alloc(limit, sizeof(int));
work = (double *) R_alloc(lenw,  sizeof(double));
```

The comments in the source code in `src/appl/integrate.c` give more details, particularly about reasons for failure (`ier >= 1`).

6.10 Utility functions

R has a fairly comprehensive set of sort routines which are made available to users' C code. The following is declared in header file `Rinternals.h`.

`void R_orderVector (int* indx, int n, SEXP arglist, Rboolean nalast, Rboolean decreasing)` [Function]

`void R_orderVector1 (int* indx, int n, SEXP x, Rboolean nalast, Rboolean decreasing)` [Function]

`R_orderVector()` corresponds to R's `order(..., na.last, decreasing)`. More specifically, `indx <- order(x, y, na.last, decreasing)` corresponds to `R_orderVector(indx, n, Rf_lang2(x, y), nalast, decreasing)` and for three vectors, `Rf_lang3(x,y,z)` is used as *arglist*.

Both `R_orderVector` and `R_orderVector1` assume the vector `indx` to be allocated to length $\geq n$. On return, `indx[]` contains a permutation of `0:(n-1)`, i.e., 0-based C indices (and not 1-based R indices, as R's `order()`).

When ordering only one vector, `R_orderVector1` is faster and corresponds (but is 0-based) to R's `indx <- order(x, na.last, decreasing)`. It was added in R 3.3.0.

All other sort routines are declared in header file `R_ext/Utils.h` (included by `R.h`) and include the following.

`void R_isort (int* x, int n)` [Function]

`void R_rsort (double* x, int n)` [Function]

`void R_csort (Rcomplex* x, int n)` [Function]

`void rsort_with_index (double* x, int* index, int n)` [Function]

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

`rsort_with_index` sorts on *x*, and applies the same permutation to *index*. `NA`s are sorted last.

`void revsort (double* x, int* index, int n)` [Function]

Is similar to `rsort_with_index` but sorts into decreasing order, and `NA`s 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, size_t i, size_t j)` [Function]
`void R_qsort_I (double *v, int *I, int i, int j)` [Function]
`void R_qsort_int (int *iv, size_t i, size_t 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.

`subroutine qsort4 (double precision v, integer indx, integer ii, integer jj)` [Function]
`subroutine qsort3 (double precision v, integer ii, integer jj)` [Function]
The Fortran interface routines for sorting double precision vectors are `qsort3` and `qsort4`, equivalent to `R_qsort` and `R_qsort_I`, respectively.

`void R_max_col (double* matrix, int* nr, int* nc, int* maxes, int* ties_meth)` [Function]
Given the *nr* by *nc* matrix `matrix` in column-major ("Fortran") order, `R_max_col()` returns in `maxes[i-1]` the column number of the maximal element in the i-th row (the same as R's `max.col()` function). In the case of ties (multiple maxima), `*ties_meth` is an integer code in `1:3` determining the method: 1 = "random", 2 = "first" and 3 = "last". See R's help page `?max.col`.

`int findInterval (double* xt, int n, double x, Rboolean rightmost_closed, Rboolean all_inside, int ilo, int* mflag)` [Function]
`int findInterval2(double* xt, int n, double x, Rboolean rightmost_closed, Rboolean all_inside, Rboolean left_open, int ilo, int* mflag)` [Function]
Given the ordered vector xt of length n, return the interval or index of x in `xt[]`, typically $\max(i;\ 1 \le i \le n\ \&\ xt[i] \le x)$ where we use 1-indexing as in R and Fortran (but not C). If *rightmost_closed* is true, also returns $n-1$ if x equals $xt[n]$. If *all_inside* is not 0, the result is coerced to lie in `1:(n-1)` even when x is outside the $xt[]$ range. On return, `*mflag` equals -1 if x `<` $xt[1]$, $+1$ if x `>=` $xt[n]$, and 0 otherwise.

The algorithm is particularly fast when *ilo* is set to the last result of `findInterval()` and x is a value of a sequence which is increasing or decreasing for subsequent calls.

`findInterval2()` is a generalization of `findInterval()`, with an extra `Rboolean` argument *left_open*. Setting `left_open = TRUE` basically replaces all left-closed right-open intervals $[s$ by left-open ones (s, see the help page of R function `findInterval` for details.

There is also an `F77_CALL(interv)()` version of `findInterval()` with the same arguments, but all pointers.

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

`char * R_tmpnam (const char *prefix, const char *tmpdir)` [Function]
`char * R_tmpnam2 (const char *prefix, const char *tmpdir, const char *fileext)` [Function]

`void R_free_tmpnam (char *name)` [Function]
Return a pathname for a temporary file with name beginning with *prefix* and ending with *fileext* in directory *tmpdir*. A `NULL` prefix or extension is replaced by `""`. Note that the return value is dynamically allocated and should be freed using `R_free_tmpnam` when no longer needed (unlike the system call `tmpnam`). Freeing the result using `free` is no longer recommended.

There is also the internal function used to expand file names in several R functions, and called directly by `path.expand`.

`const char * R_ExpandFileName (const char *fn)` [Function]
Expand a path name *fn* by replacing a leading tilde by the user's home directory (if defined). The precise meaning is platform-specific; it will usually be taken from the environment variable `HOME` if this is defined.

For historical reasons there are Fortran interfaces to functions `D1MACH` and `I1MACH`. These can be called from C code as e.g. `F77_CALL(d1mach)(4)`. Note that these are emulations of the original functions by Fox, Hall and Schryer on Netlib at `https://netlib.org/slatec/src/` for IEC 60559 arithmetic (required by R).

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 `size_t` 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 Condition handling and cleanup code

Three functions are available for establishing condition handlers from within C code:

```
#include <Rinternals.h>

SEXP R_tryCatchError(SEXP (*fun)(void *data), void *data,
     SEXP (*hndlr)(SEXP cond, void *hdata), void *hdata);

SEXP R_tryCatch(SEXP (*fun)(void *data), void *data,
SEXP,
SEXP (*hndlr)(SEXP cond, void *hdata), void *hdata,
```

```
void (*clean)(void *cdata), void *cdata);
SEXP R_withCallingErrorHandler(SEXP (*fun)(void *data), void *data,
       SEXP (*hndlr)(SEXP cond, void *hdata), void *hdata)
```

`R_tryCatchError` establishes an exiting handler for conditions inheriting form class `error`.

`R_tryCatch` can be used to establish a handler for other conditions and to register a cleanup action. The conditions to be handled are specified as a character vector (`STRSXP`). A `NULL` pointer can be passed as `fun` or `clean` if condition handling or cleanup are not needed.

These are currently implemented using the R-level `tryCatch` mechanism so are subject to some overhead.

`R_withCallingErrorHandler` establishes a calling handler for conditions inheriting form class `error`. It establishes the handler without calling back into R and will therefore be more efficient.

The function `R_UnwindProtect` can be used to ensure that a cleanup action takes place on ordinary return as well as on a non-local transfer of control, which R implements as a `longjmp`.

```
SEXP R_UnwindProtect(SEXP (*fun)(void *data), void *data,
                     void (*clean)(void *data, Rboolean jump), void *cdata,
                     SEXP cont);
```

`R_UnwindProtect` can be used in two ways. The simper usage, suitable for use in C code, passes `NULL` for the `cont` argument. `R_UnwindProtect` will call `fun(data)`. If `fun` returns a value, then `R_UnwindProtect` calls `clean(cleandata, FALSE)` before returning the value returned by `fun`. If `fun` executes a non-local transfer of control, then `clean(cleandata, TRUE)` is called, and the non-local transfer of control is resumed.

The second use pattern, suitable to support C++ stack unwinding, uses two additional functions:

```
SEXP R_MakeUnwindCont();
NORET void R_ContinueUnwind(SEXP cont);
```

`R_MakeUnwindCont` allocates a *continuation token* `cont` to pass to `R_UnwindProtect`. This token should be protected with `PROTECT` before calling `R_UnwindProtect`. When the `clean` function is called with `jump == TRUE`, indicating that R is executing a non-local transfer of control, it can throw a C++ exception to a C++ `catch` outside the C++ code to be unwound, and then use the continuation token in the a call `R_ContinueUnwind(cont)` to resume the non-local transfer of control within R.

6.13 Allowing interrupts

No part 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 signaling 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.

6.14 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[6] systems (e.g. most OSes on Sparc and PowerPC hardware) and not on little-endian systems (nowadays all the commoner R platforms). It can be useful when manipulating binary files. NB: these macros apply only to the C compiler used to build R, not necessarily to another C or C++ compiler.

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

```
#if defined(R_VERSION) && R_VERSION >= R_Version(3, 1, 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 patch level (as in '`2.2`').

Packages which use `alloca` need to ensure it is defined: as it is part of neither C nor POSIX there is no standard way to do so. One can use

```
#include <Rconfig.h> // for HAVE_ALLOCA_H
#ifdef __GNUC__
// this covers gcc, clang, icc
# undef alloca
# define alloca(x) __builtin_alloca((x))
#elif defined(HAVE_ALLOCA_H)
// needed for native compilers on Solaris and AIX
# include <alloca.h>
#endif
```

(and this should be included before standard C headers such as `stdlib.h`, since on some platforms these include `malloc.h` which may have a conflicting definition), which suffices for known R platforms.

6.15 Inlining C functions

The C99 keyword `inline` should be recognized by all compilers nowadays 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** (`https://CRAN.R-project.org/package=cluster`)

```
#include <R.h>

static R_INLINE int ind_2(int l, int j)
{
...
}
```

Be aware that using inlining with functions in more than one compilation unit is almost impossible to do portably, see `https://www.greenend.org.uk/rjk/tech/inline.html`, so this

[6] `https://en.wikipedia.org/wiki/Endianness`.

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.

6.16 Controlling visibility

Header `R_ext/Visibility.h` 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 often defined on modern Unix-alikes with a recent compiler[7], but not supported on macOS nor 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 other 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`', '`CXX_VISIBILITY`'[8] and '`F_VISIBILITY`' for C, C++ and Fortran compilers.[9] These are defined in `etc/Makeconf` and so available for normal compilation of package code. For example, `src/Makevars` could include some of

```
PKG_CFLAGS=$(C_VISIBILITY)
PKG_CXXFLAGS=$(CXX_VISIBILITY)
PKG_FFLAGS=$(F_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);
...
}
```

Because the '`C_VISIBILITY`' mechanism is only useful in conjunction with `attribute_visible`, it is not enabled unless '`HAVE_VISIBILITY_ATTRIBUTE`' is defined. The usual visibility flag is `-fvisibility=hidden`: some compilers also support `-fvisibility-inlines-hidden` which can be used by overriding '`C_VISIBILITY`' and '`CXX_VISIBILITY`' in `config.site` when building R, or editing `etc/Makeconf` in the R installation.

Note that `configure` only checks that visibility attributes and flags are accepted, not that they actually hide symbols.

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 *pkgnme*`/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
```

[7] It is defined by the Intel compilers, but also hides unsatisfied references and so cannot be used with R. It was not supported by the AIX nor Solaris compilers.

[8] This applies to the compiler for the default C++ dialect (currently C++11) and not necessarily to other dialects.

[9] In some cases Fortran compilers accept the flag but do not actually hide their symbols.

6.17 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 175, as from that header file.)

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

```
#define MATHLIB_STANDALONE
#include <Rmath.h>
```

and link against '`-lRmath`' (and perhaps '`-lm`'). There is an example file `test.c`.

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

```
double unif_rand(void)
```

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

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

to set its seeds and

```
get_seed(unsigned int *, unsigned int *)
```

to read the seeds).

6.18 Organization of header files

The header files which R installs are in directory *`R_INCLUDE_DIR`* (default *`R_HOME`*`/include`). This currently includes

`R.h`	includes many other files
`Rinternals.h`	definitions for using R's internal structures
`Rdefines.h`	macros for an S-like interface to the above (no longer maintained)
`Rmath.h`	standalone math library
`Rversion.h`	R version information
`Rinterface.h`	for add-on front-ends (Unix-alikes only)
`Rembedded.h`	for add-on front-ends
`R_ext/Applic.h`	optimization and integration
`R_ext/BLAS.h`	C definitions for BLAS routines
`R_ext/Callbacks.h`	C (and R function) top-level task handlers
`R_ext/GetX11Image.h`	X11Image interface used by package **trkplot**
`R_ext/Lapack.h`	C definitions for some LAPACK routines
`R_ext/Linpack.h`	C definitions for some LINPACK routines, not all of which are included in R
`R_ext/Parse.h`	a small part of R's parse interface: not part of the stable API.
`R_ext/RStartup.h`	for add-on front-ends
`R_ext/Rdynload.h`	needed to register compiled code in packages
`R_ext/Riconv.h`	interface to `iconv`
`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 (not Windows)

The following headers are included by `R.h`:

`Rconfig.h`	configuration info that is made available
`R_ext/Arith.h`	handling for `NA`s, `NaN`s, `Inf`/`-Inf`
`R_ext/Boolean.h`	`TRUE`/`FALSE` type
`R_ext/Complex.h`	C typedefs for R's `complex`
`R_ext/Constants.h`	constants
`R_ext/Error.h`	error signaling
`R_ext/Memory.h`	memory allocation
`R_ext/Print.h`	`Rprintf` and variations.
`R_ext/RS.h`	definitions common to `R.h` and the former `S.h`, including `F77_CALL` etc.
`R_ext/Random.h`	random number generation
`R_ext/Utils.h`	sorting and other utilities
`R_ext/libextern.h`	definitions for exports from `R.dll` on Windows.

The graphics systems are exposed in headers `R_ext/GraphicsEngine.h`, `R_ext/GraphicsDevice.h` (which it includes) and `R_ext/QuartzDevice.h`. Facilities for defining custom connection implementations are provided in `R_ext/Connections.h`, but make sure you consult the file before use.

Let us re-iterate the advice to include in C++ code system headers before the R header files, especially `Rinternals.h` (included by `Rdefines.h`) and `Rmath.h`, which redefine names which may be used in system headers, or (preferably) to define `R_NO_REMAP`. Setting the environment variable `_R_CXX_USE_NO_REMAP_` to a true value allows the need for this to be tested, as it causes `R CMD INSTALL` to compile C++ code defining `R_NO_REMAP`. `_R_CXX_USE_NO_REMAP_` is set by `R CMD check --as-cran` and is planned to become the default in future.

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**.

First, a *caveat*: a function named `gen.cl` will be invoked by the generic `gen` for class `cl`, so do not name functions in this style unless they are intended to be methods.

The key function for methods is `NextMethod`, which dispatches the next method. It is quite typical for a method function to make a few changes to its arguments, dispatch to the next method, receive the results and modify them a little. An example is

```
t.data.frame <- function(x)
{
    x <- as.matrix(x)
    NextMethod("t")
}
```

Note that the example above works because there is a *next* method, the default method, not that a new method is selected when the class is changed.

Any method a programmer writes may be invoked from another method by `NextMethod`, *with the arguments appropriate to the previous method.* Further, the programmer cannot predict which method `NextMethod` will pick (it might be one not yet dreamt of), and the end user calling the generic needs to be able to pass arguments to the next method. For this to work

> *A method must have all the arguments of the generic, including* `...` *if the generic does.*

It is a grave misunderstanding to think that a method needs only to accept the arguments it needs. The original S version of `predict.lm` did not have a `...` argument, although `predict` did. It soon became clear that `predict.glm` needed an argument `dispersion` to handle over-dispersion. As `predict.lm` had neither a `dispersion` nor a `...` argument, `NextMethod` could no longer be used. (The legacy, two direct calls to `predict.lm`, lives on in `predict.glm` in R, which is based on the workaround for S3 written by Venables & Ripley.)

Further, the user is entitled to use positional matching when calling the generic, and the arguments to a method called by `UseMethod` are those of the call to the generic. Thus

> *A method must have arguments in exactly the same order as the generic.*

To see the scale of this problem, consider the generic function `scale`, defined as

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

Suppose an unthinking package writer created methods such as

```
scale.foo <- function(x, scale = FALSE, ...) { }
```

Then for `x` of class `"foo"` the calls

```
scale(x, , TRUE)
scale(x, scale = TRUE)
```

would 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 something like

```
foo <- function(object, ...) UseMethod("foo")
foo.default <- function(object, ...) base::foo(object)
```

Earlier versions of this manual suggested assigning `foo.default <- base::foo`. This is **not** a good idea, as it captures the base function at the time of installation and it might be changed as R is patched or updated.

The same idea can be applied for functions in other packages.

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** (`https://CRAN.R-project.org/package=Rserve`) (from CRAN, see also `https://www.rforge.net/Rserve/`) and connections to Java in '`JRI`' (part of the **rJava** (`https://CRAN.R-project.org/package=rJava`) package on CRAN).

Note that the APIs described in this chapter are only intended to be used in an alternative front-end: they are not part of the API made available for R packages and can be dangerous to use in a conventional package (although packages may contain alternative front-ends). Conversely some of the functions from the API (such as `R_alloc`) should not be used in front-ends.

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 that can be used in a very similar way (at least on Linux: on other platforms one needs to ensure that all the symbols exported by `libR.a` are linked into the front-end).

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 macOS consoles are others. The source for *R_HOME*`/bin/exec/R` is in file `src/main/Rmain.c` and is very simple

```
int Rf_initialize_R(int ac, char **av); /* in ../unix/system.c */
void Rf_mainloop();                     /* in main.c */

extern int R_running_as_main_program;   /* in ../unix/system.c */

int main(int ac, char **av)
{
    R_running_as_main_program = 1;
    Rf_initialize_R(ac, av);
    Rf_mainloop(); /* does not return */
    return 0;
}
```

indeed, misleadingly simple. Remember that *R_HOME*`/bin/exec/R` is run from a shell script *R_HOME*`/bin/R` which sets up the environment for the executable, and this is used for

- Setting `R_HOME` and checking it is valid, as well as the path `R_SHARE_DIR` and `R_DOC_DIR` to the installed `share` and `doc` directory trees. Also setting `R_ARCH` if needed.
- Setting `LD_LIBRARY_PATH` to include the directories used in linking R. This is recorded as the default setting of `R_LD_LIBRARY_PATH` in the shell script *R_HOME*`/etc`*R_ARCH*`/ldpaths`.
- Processing some of the arguments, for example to run R under a debugger and to launch alternative front-ends to provide GUIs.

The first two of these can be achieved for your front-end by running it *via* `R CMD`. So, for example

```
R CMD /usr/local/lib/R/bin/exec/R
R CMD exec/R
```

will both work in a standard R installation. (`R CMD` looks first for executables in *R_HOME*`/bin`. These command-lines need modification if a sub-architecture is in use.) If you do not want to

[1] In the parlance of macOS this is a *dynamic* library, and is the normal way to build R on that platform.

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_ReplDLLinit();
        while(R_ReplDLLdo1() > 0) {
        /* add user actions here if desired */
       }

     */
    Rf_endEmbeddedR(0);
    /* final tidying up after R is shutdown */
    return 0;
}
```

If you do not 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 make use of extended precision registers).

The standard code sets up a session temporary directory in the usual way, *unless* `R_TempDir` is set to a non-NULL value before `Rf_initEmbeddedR` is called. In that case the value is assumed to contain an existing writable directory, 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.

[2] but these are not part of the automated test procedures and so little tested.

It is the user's responsibility to attempt to initialize only once. To protect the R interpreter, `Rf_initialize_R` will exit the process if re-initialization is attempted.

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

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

(These apply only to an uninstalled copy or a standard install.)

If R is installed, `pkg-config` is available and neither sub-architectures nor a macOS framework have been used, alternatives for a shared R library are

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

and for a static R library

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

(This may work for an installed OS framework if `pkg-config` is taught where to look for `libR.pc`: it is installed inside the framework.)

However, a more comprehensive way is to set up a `Makefile` to compile the front-end. Suppose file `myfe.c` is to be compiled to `myfe`. A suitable `Makefile` might be

```
## WARNING: does not work when ${R_HOME} contains spaces
include ${R_HOME}/etc${R_ARCH}/Makeconf
all: myfe

## The following is not needed, but avoids PIC flags.
myfe.o: myfe.c
        $(CC) $(ALL_CPPFLAGS) $(CFLAGS) -c myfe.c -o $@

## replace $(LIBR) $(LIBS) by $(STATIC_LIBR) if R was build with a static libR
myfe: myfe.o
        $(MAIN_LINK) -o $@ myfe.o $(LIBR) $(LIBS)
```

invoked as

```
R CMD make
R CMD myfe
```

Even though not recommended, `${R_HOME}` may contain spaces. In that case, it cannot be passed as an argument to `include` in the makefile. Instead, one can instruct `make` using the `-f` option to include `Makeconf`, for example *via* recursive invocation of `make`, see Section 1.6 [Writing portable packages], page 55.

```
all:
	$(MAKE) -f "${R_HOME}/etc${R_ARCH}/Makeconf" -f Makefile.inner
```

Additional flags which `$(MAIN_LINK)` includes are, amongst others, those to select OpenMP and `--export-dynamic` for the GNU linker on some platforms. In principle `$(LIBS)` is not needed when using a shared R library as `libR` is linked against those libraries, but some platforms need the executable also linked against them.

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)

```
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 **,
                               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);
extern int  (*ptr_R_EditFiles)(int, const char **, const char **, const char *);
extern SEXP (*ptr_do_selectlist)(SEXP, SEXP, SEXP, SEXP);
extern SEXP (*ptr_do_dataentry)(SEXP, SEXP, SEXP, SEXP);
extern SEXP (*ptr_do_dataviewer)(SEXP, SEXP, SEXP, SEXP);
extern void (*ptr_R_ProcessEvents)();
```

which allow standard R callbacks to be redirected to your GUI. What these do is generally documented in the file `src/unix/system.txt`.

`void R_ShowMessage (char *message)` [Function]

This should display the message, which may have multiple lines: it should be brought to the user's attention immediately.

`void R_Busy (int which)` [Function]

This function invokes actions (such as change of cursor) when R embarks on an extended computation (*which*=1) and when such a state terminates (*which*=0).

`int R_ReadConsole (const char *prompt, unsigned char *buf, int buflen, int hist)` [Function]

`void R_WriteConsole (const char *buf, int buflen)` [Function]

`void R_WriteConsoleEx (const char *buf, int buflen, int otype)` [Function]

`void R_ResetConsole ()` [Function]

`void R_FlushConsole ()` [Function]

`void R_ClearerrConsole ()` [Function]

These functions interact with a console.

`R_ReadConsole` prints the given prompt at the console and then does a `fgets(3)`–like operation, transferring up to *buflen* characters into the buffer *buf*. The last two bytes should be set to '`"\n\0"`' to preserve sanity. If *hist* is non-zero, then the line should be added to any command history which is being maintained. The return value is 0 is no input is available and >0 otherwise.

`R_WriteConsoleEx` writes the given buffer to the console, *otype* specifies the output type (regular output or warning/error). Call to `R_WriteConsole(buf, buflen)` is equivalent to `R_WriteConsoleEx(buf, buflen, 0)`. To ensure backward compatibility of the callbacks, `ptr_R_WriteConsoleEx` is used only if `ptr_R_WriteConsole` is set to `NULL`. To ensure that `stdout()` and `stderr()` connections point to the console, set the corresponding files to `NULL` *via*

```
R_Outputfile = NULL;
```

```
R_Consolefile = NULL;
```

`R_ResetConsole` is called when the system is reset after an error. `R_FlushConsole` is called to flush any pending output to the system console. `R_ClearerrConsole` clears any errors associated with reading from the console.

`int R_ShowFiles (int *nfile*, const char **\*\*file*, const char **\*\*headers*, const char *\*wtitle*, Rboolean *del*, const char *\*pager*)` [Function]

This function is used to display the contents of files.

`int R_ChooseFile (int *new*, char *\*buf*, int *len*)` [Function]

Choose a file and return its name in *buf* of length *len*. Return value is 0 for success, > 0 otherwise.

`int R_EditFile (const char *\*buf*)` [Function]

Send a file to an editor window.

`int R_EditFiles (int *nfile*, const char **\*\*file*, const char **\*\*title*, const char *\*editor*)` [Function]

Send *nfile* files to an editor, with titles possibly to be used for the editor window(s).

`SEXP R_loadhistory (SEXP, SEXP, SEXP, SEXP);` [Function]
`SEXP R_savehistory (SEXP, SEXP, SEXP, SEXP);` [Function]
`SEXP R_addhistory (SEXP, SEXP, SEXP, SEXP);` [Function]

`.Internal` functions for `loadhistory`, `savehistory` and `timestamp`.

If the console has no history mechanism these can be as simple as

```
SEXP R_loadhistory (SEXP call, SEXP op, SEXP args, SEXP env)
{
    errorcall(call, "loadhistory is not implemented");
    return R_NilValue;
}
SEXP R_savehistory (SEXP call, SEXP op , SEXP args, SEXP env)
{
    errorcall(call, "savehistory is not implemented");
    return R_NilValue;
}
SEXP R_addhistory (SEXP call, SEXP op , SEXP args, SEXP env)
{
    return R_NilValue;
}
```

The `R_addhistory` function should return silently if no history mechanism is present, as a user may be calling `timestamp` purely to write the time stamp to the console.

`void R_Suicide (const char *\*message*)` [Function]

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

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

`void R_CleanUp (SA_TYPE saveact, int status, int RunLast)` [Function]
This function invokes any actions which occur at system termination. It needs to be quite complex:

```
#include <Rinterface.h>
#include <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);
}
```

These callbacks should never be changed in a running R session (and hence cannot be called from an extension package).

`SEXP R_dataentry (SEXP, SEXP, SEXP, SEXP);` [Function]
`SEXP R_dataviewer (SEXP, SEXP, SEXP, SEXP);` [Function]
`SEXP R_selectlist (SEXP, SEXP, SEXP, SEXP);` [Function]
`.External` functions for `dataentry` (and `edit` on matrices and data frames), `View` and `select.list`. These can be changed if they are not currently in use.

8.1.3 Registering symbols

An application embedding R needs a different way of registering symbols because it is not a dynamic library loaded by R as would be the case with a package. Therefore R reserves a special `DllInfo` entry for the embedding application such that it can register symbols to be

used with `.C`, `.Call` etc. This entry can be obtained by calling `getEmbeddingDllInfo`, so a typical use is

```
DllInfo *info = R_getEmbeddingDllInfo();
R_registerRoutines(info, cMethods, callMethods, NULL, NULL);
```

The native routines defined by `cMethods` and `callMethods` should be present in the embedding application. See Section 5.4 [Registering native routines], page 127, 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 for direct socket access in file `src/modules/internet/Rsock.c` and for the interface to `libcurl`.
- 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.

It is not intended that these facilities are used by packages, but if they are needed exceptionally, the package should ensure that it cleans up and removes its handlers when its namespace is unloaded. Note that the header `sys/select.h` is needed[3]: users should check this is available and define `HAVE_SYS_SELECT_H` before including `R_ext/eventloop.h`. (It is often the case that another header will include `sys/select.h` before `eventloop.h` is processed, but this should not be relied on.)

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. The ability to add a polled handler as `R_timeout_handler` is used by the **tcltk** package.

8.1.5 Threading issues

Embedded R is designed to be run in the main thread, and all the testing is done in that context. There is a potential issue with the stack-checking mechanism where threads are involved. This uses two variables declared in `Rinterface.h` (if `CSTACK_DEFNS` is defined) as

```
extern uintptr_t R_CStackLimit; /* C stack limit */
extern uintptr_t R_CStackStart; /* Initial stack address */
```

[3] At least according to POSIX 2004 and later. Earlier standards prescribed `sys/time.h`: `R_ext/eventloop.h` will include it if `HAVE_SYS_TIME_H` is defined.

Note that `uintptr_t` is an optional C99 type for which a substitute is defined in R, so your code needs to define `HAVE_UINTPTR_T` appropriately. To do so, test if the type is defined in C header `stdint.h` or C++ header `cstdint` and if so include the header and define `HAVE_UINTPTR_T` before including `Rinterface.h`. (For C code one can simply include `Rconfig.h`, possibly *via* `R.h`, and for C++11 code `Rinterface.h` will include the header `cstdint`.)

These will be set[4] 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` immediately after `Rf_initialize_R` is called, 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 10Mb is recommended.)

You may also want to consider how signals are handled: R sets signal handlers for several signals, including `SIGINT`, `SIGSEGV`, `SIGPIPE`, `SIGUSR1` and `SIGUSR2`, but these can all be suppressed by setting the variable `R_SignalHandlers` (declared in `Rinterface.h`) to `0`.

Note that these variables must not be changed by an R **package**: a package should not call R internals which makes use of the stack-checking mechanism on a secondary thread.

8.2 Embedding R under Windows

This section is only about '`x86_64`' 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* `https://www.autstat.com/`, which works with R packages to support transfer of data to and from R and remote execution of R commands, as well as embedding of an R graphics window.

  Recent versions have usage restrictions.

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

```
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.

[4] at least on platforms where the values are available, that is having `getrlimit` and on Linux or having `sysctl` supporting `KERN_USRSTACK`, including FreeBSD and macOS.

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

```
#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, unsigned char *buf, int len, int addtohistory)
{
    fputs(prompt, stdout);
    fflush(stdout);
    if(fgets((char *)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, *RUser;

    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_DefParamsEx(Rp, RSTART_VERSION);
    if((RHome = get_R_HOME()) == NULL) {
        fprintf(stderr, "R_HOME must be set in the environment or Registry\n");
        exit(1);
    }
    Rp->rhome = RHome;
```

```
    RUser = getRUser();
    Rp->home = RUser;
    Rp->CharacterMode = LinkDLL;
    Rp->EmitEmbeddedUTF8 = FALSE;
    Rp->ReadConsole = myReadConsole;
    Rp->WriteConsole = myWriteConsole;
    Rp->CallBack = myCallBack;
    Rp->ShowMessage = askok;
    Rp->YesNoCancel = askyesnocancel;
    Rp->Busy = myBusy;

    Rp->R_Quiet = TRUE;        /* Default is FALSE */
    Rp->R_Interactive = FALSE; /* Default is TRUE */
    Rp->RestoreAction = SA_RESTORE;
    Rp->SaveAction = SA_NOSAVE;
    R_SetParams(Rp);
    freeRUser(RUser);
    free_R_HOME(RHome);
    R_set_command_line_arguments(argc, argv);

    FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));

    signal(SIGBREAK, my_onintr);
    GA_initapp(0, 0);
    readconsolecfg();
    setup_Rmainloop();
#ifdef SIMPLE_CASE
    run_Rmainloop();
#else
    R_ReplDLLinit();
    while(R_ReplDLLdo1() > 0) {
/* add user actions here if desired */
    }
/* only get here on EOF (not q()) */
#endif
    Rf_endEmbeddedR(0);
    return 0;
}
```

The ideas are

- Check that the front-end and the linked `R.dll` match – other front-ends may allow a looser match.
- Find and set the R home directory and the user's home directory. The former may be available from the Windows Registry: it will 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. `R_DefParamsEx` takes an extra argument, the version number of the `Rstart` structure provided (`RSTART_VERSION` refers to the current version) and returns a non-zero status when that version is not supported by R.
- Record the command-line arguments used by `R_set_command_line_arguments` for use by the R function `commandArgs()`.
- Set up the signal handler and the basic user interface.
- Run the main R loop, possibly with our actions intermeshed.
- Arrange to clean up.

An underlying theme is the need to keep the GUI 'alive', and this has not been done in this example. The R callback `R_ProcessEvents` needs to be called frequently to ensure that

Windows events in R windows are handled expeditiously. Conversely, R needs to allow the GUI code (which is running in the same process) to update itself as needed – two ways are provided to allow this:

- `R_ProcessEvents` calls the callback registered by `Rp->callback`. A version of this is used to run package Tcl/Tk for **tcltk** under Windows, for the code is

```
void R_ProcessEvents(void)
{
    while (peekevent()) doevent(); /* Windows events for GraphApp */
    if (UserBreak) { UserBreak = FALSE; onintr(); }
    R_CallBackHook();
    if(R_tcldo) R_tcldo();
}
```

- The mainloop can be split up to allow the calling application to take some action after each line of input has been dealt with: see the alternative code below `#ifdef SIMPLE_CASE`.

It may be that no R GraphApp windows need to be considered, although these include pagers, the `windows()` graphics device, the R data and script editors and various popups such as `choose.file()` and `select.list()`. It would be possible to replace all of these, but it seems easier to allow GraphApp to handle most of them.

It is possible to run R in a GUI in a single thread (as `RGui.exe` shows) but it will normally be easier[5] 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.

Applications embedding R 4.2.0 and newer should use UCRT as the C runtime and opt in for UTF-8 as the active code page in their manifest, as all frontends shipped with R do. This will allow the embedded R to use UTF-8 as its native encoding on recent Windows systems.

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* 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. Only 64-bit versions are supported since R 4.2.

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, with Intel builds 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

[5] 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.

uninstalled. To allow information about other installed versions to be retained, there is also a key named something like `3.0.0` or `3.0.0 patched` or `3.1.0 Pre-release` with a value for `InstallPath`.

So a comprehensive algorithm to search for `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`.

Function and variable index

.

.C 124
.Call 141, 152
.External 141, 152
.Fortran 124
.Last.lib 51
.onAttach 51
.onDetach 51
.onLoad 51
.onUnload 51
.Random.seed 168

\

\abbr 91
\acronym 90
\alias 82
\arguments 84
\author 85
\bold 89
\cite 90
\code 89
\command 90
\concept 94
\cr 88
\CRANpkg{*pkg*} 96
\deqn 92
\describe 91
\description 83
\details 84
\dfn 90
\doi{*numbers*} 96
\dontdiff 86
\dontrun 85
\dontshow 85
\donttest 86
\dots 93
\dQuote 89
\email 90
\emph 89
\enc 93
\enumerate 91
\env 90
\eqn 92
\examples 85
\figure 93
\file 90
\format 87
\href 90
\if 94
\ifelse 94
\itemize 91
\kbd 89
\keyword 86
\ldots 93
\link 91
\linkS4class 91
\method 83
\name 82
\newcommand 96
\note 85
\option 90
\out 94
\packageAuthor 96
\packageDescription 96
\packageDESCRIPTION 96
\packageIndices 96
\packageMaintainer 96
\packageTitle 96
\pkg 90
\preformatted 89
\RdOpts 95
\references 85
\renewcommand 96
\R 93
\S3method 84
\samp 89
\section 88
\seealso 85
\Sexpr 95
\source 87
\sQuote 89
\sspace 96
\strong 89
\tabular 91
\title 83
\url 90
\usage 83
\value 84
\var 90
\verb 89

A

allocList 156
allocMatrix 159
allocVector 144
any_duplicated 150
any_duplicated3 150
asChar 149
asComplex 149
asInteger 149
asLogical 149
asReal 149
AUTHORS 17

B

bessel_i 177
bessel_j 177
bessel_k 177
bessel_y 177
beta 177
BLAS_LIBS 25
browser 106

C

CAAR ... 154
CAD4R ... 153
CAD5R ... 153
CADDDR ... 153
CADDR ... 153
CADR ... 153
Calloc ... 167
CallocCharBuf ... 168
CAR ... 153
CDAR ... 154
CDDDR ... 154
CDDR ... 154
CDR ... 153
cgmin ... 181
choose ... 177
CITATION ... 17, 78
classgets ... 147
CleanEd ... 197
coerceVector ... 145
COMPLEX ... 164
cons ... 150
CONS ... 150
COPYRIGHTS ... 6, 17
cospi ... 178
cPsort ... 182

D

dblepr ... 170
dblepr1 ... 170
debug ... 109
debugger ... 108
defineVar ... 149
digamma ... 177
dimgets ... 146
dimnamesgets ... 146
dpsifn ... 177
dump.frames ... 108
duplicate ... 150
dyn.load ... 126
dyn.unload ... 126

E

elt ... 150
error ... 168
errorcall ... 168
eval ... 155
exp_rand ... 168
expm1 ... 178
export ... 49
exportClasses ... 54
exportClassPattern ... 54
exportMethods ... 54
exportPattern ... 49, 54

F

FALSE ... 180
findFun ... 155
findInterval ... 183
findInterval2(double* ... 183
findVar ... 149
FLIBS ... 25
fmax2 ... 179
fmin2 ... 179
fprec ... 179
fpu_setup ... 197
Free ... 167
fround ... 179
fsign ... 179
ftrunc ... 179

G

gammafn ... 177
gctorture ... 111
GetArrayDimnames ... 146
getAttrib ... 146
getCharCE ... 164
GetColNames ... 146
GetMatrixDimnames ... 146
GetRNGstate ... 168
GetRowNames ... 146

I

imax2 ... 179
imin2 ... 179
import ... 49
importClassesFrom ... 55
importFrom ... 49
importMethodsFrom ... 55
install ... 147
installChar ... 149
INTEGER ... 164
integr_fn ... 181
intpr ... 170
intpr1 ... 170
iPsort ... 182
isArray ... 150
isComplex ... 145
isEnvironment ... 145
isExpression ... 145
isFactor ... 150
isFunction ... 150
isInteger ... 145
isLanguage ... 150
isList ... 150
isLogical ... 145
isMatrix ... 150
ISNA ... 154, 170
ISNAN ... 154, 170
isNewList ... 150
isNull ... 145
isNumeric ... 150
isObject ... 150
isOrdered ... 150
isPairList ... 150
isPrimitive ... 150
isReal ... 145

isString ... 145
isSymbol ... 145
isTs ... 150
isUnordered ... 150
isVector ... 150
isVectorAtomic ... 150
isVectorList ... 150

L

labelpr ... 170
lang1 ... 150
lang2 ... 150
lang3 ... 150
lang4 ... 150
lang5 ... 150
lang6 ... 150
LAPACK_LIBS ... 25
lastElt ... 150
lbeta ... 177
lbfgsb ... 181
lchoose ... 177
lcons ... 150
LCONS ... 150
lgamma1p ... 178
lgammafn ... 177
library.dynam ... 14, 126
list1 ... 150
list2 ... 150
list3 ... 150
list4 ... 150
list5 ... 150
list6 ... 150
log1mexp ... 178
log1p ... 178
log1pexp ... 178
log1pmx ... 178
LOGICAL ... 164
logspace_add ... 179
logspace_sub ... 179
logspace_sum ... 179

M

M_E ... 179
M_PI ... 179
MARK_NOT_MUTABLE ... 151
MAYBE_REFERENCED ... 151
MAYBE_SHARED ... 151
Memcpy ... 168
Memzero ... 168
mkChar ... 148
mkCharCE ... 164
mkCharLen ... 148
mkCharLenCE ... 165
mkString ... 149

N

NA_REAL ... 170
namesgets ... 146
NEWS.Rd ... 17
nmmin ... 180
norm_rand ... 168
nrows ... 166
nthcdr ... 150

O

OBJECTS ... 26, 135
onintr ... 202
optimfn ... 180
optimgr ... 180

P

pentagamma ... 177
PKG_CFLAGS ... 135
PKG_CPPFLAGS ... 135
PKG_CXXFLAGS ... 135
PKG_FCFLAGS ... 33
PKG_FFLAGS ... 135
PKG_LIBS ... 135
PKG_OBJCFLAGS ... 135
PKG_OBJCXXFLAGS ... 135
pow1p ... 177
PRINTNAME ... 154
PrintValue ... 119
prompt ... 82
protect ... 142
PROTECT ... 142
PROTECT_WITH_INDEX ... 143
psigamma ... 177
PutRNGstate ... 168

Q

qsort3 ... 183
qsort4 ... 183

R

R CMD build ... 43
R CMD check ... 39
R CMD config ... 22
R CMD Rd2pdf ... 97
R CMD Rdconv ... 97
R CMD SHLIB ... 135
R CMD Stangle ... 97
R CMD Sweave ... 97
R_addhistory ... 196
R_alloc ... 166
R_allocLD ... 166
R_Busy ... 195
R_Calloc ... 167
R_CheckUserInterrupt ... 185
R_ChooseFile ... 196
R_CleanTempDir ... 197
R_CleanUp ... 197
R_ClearerrConsole ... 195
R_ClearExternalPtr ... 162
R_compute_identical ... 150

R_ContinueUnwind 185
R_csort 182
R_dataentry 197
R_dataviewer 197
R_DefParams 201
R_DefParamsEx 201
R_dot_Last 197
R_EditFile 196
R_EditFiles 196
R_ExpandFileName 184
R_ExternalPtrAddr 162
R_ExternalPtrAddrFn 162
R_ExternalPtrProtected 162
R_ExternalPtrTag 162
R_FINITE 170
R_FlushConsole 195
R_forceSymbols 130
R_Free 167
R_free_tmpnam 183
R_GetCCallable 134
R_GetCurrentEnv 156
R_GetCurrentSrcref 161
R_getEmbeddingDllInfo 198
R_GetSrcFilename 161
R_INLINE 186
R_IsNaN 170
R_isort 182
R_LIBRARY_DIR 23
R_loadhistory 196
R_MakeExternalPtr 162
R_MakeExternalPtrFn 162
R_MakeUnwindCont 185
R_MakeWeakRef 163
R_MakeWeakRefC 163
R_max_col 183
R_NegInf 170
R_NewEnv 149
R_NewPreciousMSet 143
R_orderVector 182
R_orderVector1 182
R_PACKAGE_DIR 21, 23
R_PACKAGE_NAME 21, 23
R_ParseVector 161
R_PolledEvents 198
R_PosInf 170
R_pow 177
R_pow_di 177
R_PreserveInMSet 143
R_PreserveObject 143
R_ProcessEvents 202
R_ProtectWithIndex 143
R_PV 119
R_qsort 183
R_qsort_int 183
R_qsort_int_I 183
R_qsort_I 183
R_ReadConsole 195
R_Realloc 167
R_RegisterCCallable 134
R_RegisterCFinalizer 162
R_RegisterCFinalizerEx 162
R_RegisterFinalizer 162
R_RegisterFinalizerEx 162
R_registerRoutines 128
R_ReleaseFromMSet 143
R_ReleaseObject 143
R_ReplDLLdo1 193
R_ReplDLLinit 193
R_Reprotect 143
R_ResetConsole 195
R_rsort 182
R_RunExitFinalizers 197
R_RunWeakRefFinalizer 163
R_SaveGlobalEnv 197
R_savehistory 196
R_selectlist 197
R_set_command_line_arguments 200
R_SetExternalPtrAddr 162
R_SetExternalPtrProtected 162
R_SetExternalPtrTag 162
R_SetParams 201
R_setStartTime 200
R_ShowFiles 196
R_ShowMessage 195
R_Srcref 161
R_Suicide 196
R_tmpnam 183
R_tmpnam2 183
R_tryCatch 185
R_tryCatchError 185
R_unif_index 168
R_UnwindProtect 185
R_useDynamicSymbols 130
R_Version 186
R_wait_usec 198
R_WeakRefKey 163
R_WeakRefValue 163
R_withCallingErrorHandler 185
R_WriteConsole 195
R_WriteConsoleEx 195
RAW 164
rchkusr 185
Rdqagi 181
Rdqags 181
Realloc 167
realpr 170
realpr1 170
REAL 164
recover 109
reEnc 164
REprintf 170
REPROTECT 143
REvprintf 170
revsort 182
Rf_endEmbeddedR 193
Rf_initEmbeddedR 193
Rf_initialize_R 192
Rf_KillAllDevices 197
Rf_mainloop 192
Riconv 184
Riconv_close 184
Riconv_open 184
Rprintf 170
Rprof 99, 101
Rprofmem 102
rPsort 182
rsort_with_index 182
Rtanpi 178
run_Rmainloop 193

Rvprintf 170

S

S_alloc 166
S_realloc 166
S3method 50
SAFE_FFLAGS 26
samin 181
ScalarComplex 149
ScalarInteger 149
ScalarLogical 149
ScalarRaw 149
ScalarReal 149
ScalarString 149
SET_STRING_ELT 148
SET_TAG 156
SET_VECTOR_ELT 164
setAttrib 146
SETCAD4R 154
SETCADDDR 154
SETCADDR 154
SETCADR 154
SETCAR 154
SETCDR 154
setup_Rmainloop 200
setVar 149
sign 179
signrank_free 176
sinpi 178
str2type 150
STRING_ELT 148
summaryRprof 101
system 124
system.time 124
system2 124

T

TAG 154
tanpi 179
tetragamma 177
trace 110
traceback 107
tracemem 102
translateChar 164
translateCharUTF8 164
trigamma 177
TRUE 180
type2char 150
type2str 150
TYPEOF 154

U

undebug 110
unif_rand 168
unprotect 142
unprotect_ptr 143
UNPROTECT 142
UNPROTECT_PTR 143
untracemem 102
useDynLib 51

V

VECTOR_ELT 164
vmaxget 166
vmaxset 166
vmmin 180

W

warning 168
warningcall 168
warningcall_immediate 168
wilcox_free 176

Concept index

.

.install_extras file ... 47
.Rbuildignore file ... 43
.Rinstignore file ... 16

A

Allocating storage ... 144
Attributes ... 145

B

Bessel functions ... 177
Beta function ... 177
Building binary packages ... 44
Building source packages ... 43

C

C++ code, interfacing ... 136
Calling C from Fortran and vice versa ... 171
Checking packages ... 39
citation ... 17, 78
Classes ... 147
Cleanup code ... 184
cleanup file ... 3
Condition handling ... 184
conditionals ... 94
configure file ... 3
Copying objects ... 150
CRAN ... 3
Creating packages ... 2
Creating shared objects ... 135
Cross-references in documentation ... 91
cumulative hazard ... 175

D

Debugging ... 117
DESCRIPTION file ... 4
Details of R types ... 144
Distribution functions from C ... 175
Documentation, writing ... 81
Dynamic loading ... 126
dynamic pages ... 95

E

Editing Rd files ... 98
encoding ... 97
Error handling ... 184
Error signaling from C ... 168
Error signaling from Fortran ... 168
Evaluating R expressions from C ... 155
external pointer ... 162

F

Figures in documentation ... 93
finalizer ... 162
Finding variables ... 148

G

Gamma function ... 177
Garbage collection ... 142
Generic functions ... 190

H

handling character data ... 148
Handling lists ... 147
Handling R objects in C ... 141

I

IEEE special values ... 154, 170
INDEX file ... 13
Indices ... 94
Inspecting R objects when debugging ... 119
integration ... 181
Interfaces to compiled code ... 124, 152
Interfacing C++ code ... 136
Interrupts ... 185

L

LICENCE file ... 9
LICENSE file ... 9
Lists and tables in documentation ... 91

M

Marking text in documentation ... 89
Mathematics in documentation ... 92
Memory allocation from C ... 166
Memory use ... 101
Method functions ... 190
Missing values ... 154, 170

N

namespaces ... 48
news ... 17
Numerical analysis subroutines from C ... 175
Numerical derivatives ... 157

O

OpenMP ... 28, 186
Operating system access ... 124
optimization ... 180

P

Package builder 43
Package structure 3
Package subdirectories 13
Packages 2
Parsing R code from C 160
Platform-specific documentation 94
Polygamma functions 177
Printing from C 170
Printing from Fortran 170
Processing Rd format 97
Profiling 99, 101, 102

R

Random numbers in C 168, 175
Random numbers in Fortran 171
Registering native routines 127

S

Setting variables 148
Sort functions from C 182
Sweave 45

T

tarballs 43
Tidying R code 99

U

user-defined macros 96

V

Version information from C 186
vignettes 45
Visibility 187

W

weak reference 163

Z

Zero-finding 156