R: A Language and Environment for Statistical Computing

Reference Index

Debugging and Profiling

The R Development Core Team

Version 2.15.0 Under development (unstable) (2011-10-31)

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 Development Core Team.

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License. For more information about these matters, see http://www.gnu.org/copyleft/gpl.html.

ISBN 3-900051-07-0

Date: Oct. 2011
Revised: Nov. 2011 by G. Sawitzki
Typeset, with minor revisions: November 3, 2011 from CVS Revision : 1.5
URL: http://sintro.r-forge.r-project.org/
Description

How \textit{R} manages its workspace.

Details

\textit{R} has a variable-sized workspace. Prior to \textit{R} 2.15.0 there were (rarely-used) command-line options to control its size, but it is now sized automatically.

\textit{R} maintains separate areas for fixed and variable sized objects. The first of these is allocated as an array of \textit{cons cells} (Lisp programmers will know what they are, others may think of them as the building blocks of the language itself, parse trees, etc.), and the second are thrown on a \textit{heap} of ‘Vcells’ of 8 bytes each. Each cons cell occupies 28 bytes on a 32-bit build of \textit{R}, (usually) 56 bytes on a 64-bit build.

The default values are (currently) an initial setting of 350k cons cells, 6Mb of vector heap: note that the areas are not actually allocated initially: rather these values are the sizes for triggering garbage collection. Thereafter \textit{R} will grow or shrink the areas depending on usage, never decreasing below the initial values.

You can find out the current memory consumption (the heap and cons cells used as numbers and megabytes) by typing \texttt{gc()} at the \textit{R} prompt. Note that following \texttt{gcinfo(\text{TRUE})}, automatic garbage collection always prints memory use statistics.

The command-line option ‘--max-ppsize’ controls the maximum size of the pointer protection stack. This defaults to 50000, but can be increased to allow deep recursion or large and complicated calculations to be done. \textit{Note} that parts of the garbage collection process goes through the full reserved pointer protection stack and hence becomes slower when the size is increased. Currently the maximum value accepted is 500000.

See Also

\textit{An Introduction to R} for more command-line options.
\texttt{Memory-limits} for the design limitations.
\texttt{gc} for information on the garbage collector and total memory usage, \texttt{object.size(a)} for the (approximate) size of \textit{R} object \texttt{a}. \texttt{memory.profile} for profiling the usage of cons cells.
Memory-limits

Description

\( \mathbb{R} \) holds objects it is using in virtual memory. This help file documents the current design limitations on large objects: these differ between 32-bit and 64-bit builds of \( \mathbb{R} \).

Details

Currently \( \mathbb{R} \) runs on 32- and 64-bit operating systems, and most 64-bit OSes (including Linux, Solaris, Windows and Mac OS X) can run either 32- or 64-bit builds of \( \mathbb{R} \). The memory limits depends mainly on the build, but for a 32-bit build of \( \mathbb{R} \) on Windows they also depend on the underlying OS version.

\( \mathbb{R} \) holds all objects in memory, and there are limits based on the amount of memory that can be used by all objects:

- There may be limits on the size of the heap and the number of cons cells allowed – see Memory – but these are usually not imposed.
- There is a limit on the (user) address space of a single process such as the \( \mathbb{R} \) executable. This is system-specific, and can depend on the executable.
- The environment may impose limitations on the resources available to a single process: Windows’ versions of \( \mathbb{R} \) do so directly.

Error messages beginning cannot allocate vector of size indicate a failure to obtain memory, either because the size exceeded the address-space limit for a process or, more likely, because the system was unable to provide the memory. Note that on a 32-bit build there may well be enough free memory available, but not a large enough contiguous block of address space into which to map it.

There are also limits on individual objects. On all builds of \( \mathbb{R} \), the maximum length (number of elements) of a vector is \( 2^{31} - 1 \approx 2 \times 10^9 \), as lengths are stored as signed integers. In addition, the storage space cannot exceed the address limit, and if you try to exceed that limit, the error message begins cannot allocate vector of length. The number of characters in a character string is in theory only limited by the address space.

Unix

The address-space limit is system-specific: 32-bit OSes imposes a limit of no more than 4Gb: it is often 3Gb. Running 32-bit executables on a 64-bit OS will have similar limits: 64-bit executables will have an essentially infinite system-specific limit (e.g. 128Tb for Linux on x86_64 cpus).

See the OS/shell’s help on commands such as limit or ulimit for how to impose limitations on the resources available to a single process. For example a bash user could use

```
ulimit -t 600 -m 2000000
```

whereas a csh user might use

```
limit cputime 10m
limit memoryuse 2048m
```

to limit a process to 10 minutes of CPU time and (around) 2Gb of memory.
Windows

The address-space limit is 2Gb under 32-bit Windows unless the OS’s default has been changed to allow more (up to 3Gb). See http://www.microsoft.com/whdc/system/platform/server/PAE/PAEmem.mspx and http://msdn.microsoft.com/en-us/library/bb613473(VS.85).aspx. Under most 64-bit versions of Windows the limit for a 32-bit build of R is 4Gb: for the oldest ones it is 2Gb. The limit for a 64-bit build of R (imposed by the OS) is 8Tb.

It is not normally possible to allocate as much as 2Gb to a single vector in a 32-bit build of R even on 64-bit Windows because of preallocations by Windows in the middle of the address space.

Under Windows, R imposes limits on the total memory allocation available to a single session as the OS provides no way to do so: see memory.size and memory.limit.

See Also

object.size(a) for the (approximate) size of R object a.
**debug**

*Debug a Function*

**Description**

Set, unset or query the debugging flag on a function. The text and condition arguments are the same as those that can be supplied via a call to browser. They can be retrieved by the user once the browser has been entered, and provide a mechanism to allow users to identify which breakpoint has been activated.

**Usage**

```r
debug(fun, text="", condition=NULL)
debugonce(fun, text="", condition=NULL)
undebug(fun)
isdebugged(fun)
```

**Arguments**

- `fun`: any interpreted R function.
- `text`: a text string that can be retrieved when the browser is entered.
- `condition`: a condition that can be retrieved when the browser is entered.

**Details**

When a function flagged for debugging is entered, normal execution is suspended and the body of function is executed one statement at a time. A new browser context is initiated for each step (and the previous one destroyed).

At the debug prompt the user can enter commands or R expressions, followed by a newline. The commands are

- `n` (or just an empty line, by default). Advance to the next step.
- `c` continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.
- `cont` synonym for `c`.
- `where` print a stack trace of all active function calls.
- `Q` exit the browser and the current evaluation and return to the top-level prompt.

(Leading and trailing whitespace is ignored, except for an empty line).

Anything else entered at the debug prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and `ls()` lists the objects in the calling frame. (If you want to look at an object with a name such as `n`, print it explicitly.)

Setting option "browserNLdisabled" to TRUE disables the use of an empty line as a synonym for `n`. If this is done, the user will be re-prompted for input until a valid command or an expression is entered.

To debug a function is defined inside a function, single-step though to the end of its definition, and then call debug on its name.
If you want to debug a function not starting at the very beginning, use `trace(..., at = *)` or `setBreakpoint`.

Using `debug` is persistent, and unless debugging is turned off the debugger will be entered on every invocation (note that if the function is removed and replaced the debug state is not preserved). Use `debugonce` to enter the debugger only the next time the function is invoked.

In order to debug S4 methods (see Methods), you need to use `trace`, typically calling `browser`, e.g., as `trace("plot", browser, exit=browser, signature = c("track", "missing"))`.

The number of lines printed for the deparsed call when a function is entered for debugging can be limited by setting `options(deparse.max.lines)`.

**See Also**

`browser`, `trace`; `traceback` to see the stack after an `Error: ...` message; `recover` for another debugging approach.
**browser**

**Environment Browser**

**Description**

Interrupt the execution of an expression and allow the inspection of the environment where `browser` was called from.

**Usage**

```r
browser(text="", condition=NULL, expr=TRUE, skipCalls=0L)
```

**Arguments**

- **text**: a text string that can be retrieved once the browser is invoked.
- **condition**: a condition that can be retrieved once the browser is invoked.
- **expr**: An expression, which if it evaluates to `TRUE` the debugger will invoked, otherwise control is returned directly.
- **skipCalls**: how many previous calls to skip when reporting the calling context.

**Details**

A call to `browser` can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the R interpreter.

The purpose of the `text` and `condition` arguments are to allow helper programs (e.g. external debuggers) to insert specific values here, so that the specific call to `browser` (perhaps its location in a source file) can be identified and special processing can be achieved. The values can be retrieved by calling `browserText` and `browserCondition`.

The purpose of the `expr` argument is to allow for the illusion of conditional debugging. It is an illusion, because execution is always paused at the call to `browser`, but control is only passed to the evaluator described below if `expr` evaluates to `TRUE`. In most cases it is going to be more efficient to use an `if` statement in the calling program, but in some cases using this argument will be simpler.

The `skipCalls` argument should be used when the `browser()` call is nested within another debugging function: it will look further up the call stack to report its location.

At the browser prompt the user can enter commands or R expressions, followed by a newline. The commands are

- **c** (or just an empty line, by default) exit the browser and continue execution at the next statement.
- **cont** synonym for `c`.
- **n** enter the step-through debugger. This changes the meaning of `c`: see the documentation for `debug`.
- **where** print a stack trace of all active function calls.
- **Q** exit the browser and the current evaluation and return to the top-level prompt.

(Leading and trailing whitespace is ignored, except for an empty line).

Anything else entered at the browser prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and
`ls()` lists the objects in the calling frame. (If you want to look at an object with a name such as `n`, print it explicitly.)

The number of lines printed for the deparsed call can be limited by setting `options(deparse.max.lines)`.

Setting `option "browserNLdisabled"` to `TRUE` disables the use of an empty line as a synonym for `c`. If this is done, the user will be re-prompted for input until a valid command or an expression is entered.

This is a primitive function but does argument matching in the standard way.

**References**


**See Also**

`debug`, and `traceback` for the stack on error. `browserText` for how to retrieve the text and condition.
Description

By default traceback() prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message. It can also be used to print arbitrary lists of deparsed calls.

Usage

traceback(x = NULL, max.lines = getOption("deparse.max.lines"))

Arguments

x NULL (default, meaning .Traceback), or a list or pairlist of deparsed calls.
max.lines The maximum number of lines to be printed per call. The default is unlimited.

Details

The stack of the last uncaught error is stored as a list of deparsed calls in .Traceback, which traceback prints in a user-friendly format. The stack of deparsed calls always contains all function calls and all foreign function calls (such as .Call): if profiling is in progress it will include calls to some primitive functions. (Calls to builtins are included, but not to specials.) Errors which are caught via try or tryCatch do not generate a traceback, so what is printed is the call sequence for the last uncaught error, and not necessarily for the last error.

Value

traceback() returns nothing, but prints the deparsed call stack deepest call first. The calls may print on more than one line, and the first line for each call is labelled by the frame number. The number of lines printed per call can be limited via max.lines.

Warning

It is undocumented where .Traceback is stored nor that it is visible, and this is subject to change. Prior to R 2.4.0 it was stored in the workspace, but no longer.

References


Examples

foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a-variable-which.does.not.exist }

## Not run:
foo(2) # gives a strange error
traceback()
## End(Not run)
## 2: bar(2)
.traceback

    ## 1: foo(2)
    bar
    ## Ah, this is the culprit ...
Description

This function allows the user to browse directly on any of the currently active function calls, and is suitable as an error option. The expression `options(error=recover)` will make this the error option.

Usage

```
recover()
```

Details

When called, `recover` prints the list of current calls, and prompts the user to select one of them. The standard R `browser` is then invoked from the corresponding environment; the user can type ordinary R language expressions to be evaluated in that environment.

When finished browsing in this call, type c to return to `recover` from the browser. Type another frame number to browse some more, or type 0 to exit `recover`.

The use of `recover` largely supersedes `dump.frames` as an error option, unless you really want to wait to look at the error. If `recover` is called in non-interactive mode, it behaves like `dump.frames`. For computations involving large amounts of data, `recover` has the advantage that it does not need to copy out all the environments in order to browse in them. If you do decide to quit interactive debugging, call `dump.frames` directly while browsing in any frame (see the examples).

Value

Nothing useful is returned. However, you can invoke `recover` directly from a function, rather than through the error option shown in the examples. In this case, execution continues after you type 0 to exit `recover`.

Compatibility Note

The R `recover` function can be used in the same way as the S function of the same name; therefore, the error option shown is a compatible way to specify the error action. However, the actual functions are essentially unrelated and interact quite differently with the user. The navigating commands up and down do not exist in the R version; instead, exit the browser and select another frame.

References


See the compatibility note above, however.

See Also

`browser` for details about the interactive computations; `options` for setting the error option; `dump.frames` to save the current environments for later debugging.
Examples

## Not run:

options(error = recover) # setting the error option

### Example of interaction

> myFit <- lm(y ~ x, data = xy, weights = w)
Error in lm.wfit(x, y, w, offset = offset, ...) :
  missing or negative weights not allowed

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 2

Called from: eval(expr, envir, enclos)
Browse[1]> objects() # all the objects in this frame
[1] "method" "n" "ny" "offset" "tol" "w"
[7] "x" "y"
Browse[1]> w
[1] -/zero.noslash.5/zero.noslash13844 1.3112515 /zero.noslash.2939348 -/zero.noslash.89837/zero.noslash5 -/zero.noslash.1538642
[6] -/zero.noslash.9772989 /zero.noslash.788879/zero.noslash -/zero.noslash.1919154 -/zero.noslash.3/zero.noslash26882
Browse[1]> dump.frames() # save for offline debugging
Browse[1]> c # exit the browser

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 0 # exit recover
>

## End(Not run)
try

Try an Expression Allowing Error Recovery

Description

try is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery.

Usage

try(expr, silent = FALSE)

Arguments

expr        an R expression to try.
silent      logical: should the report of error messages be suppressed?

Details

try evaluates an expression and traps any errors that occur during the evaluation. If an error occurs then the error message is printed to the stderr connection unless options("show.error.messages") is false or the call includes silent = TRUE. The error message is also stored in a buffer where it can be retrieved by geterrmessage. (This should not be needed as the value returned in case of an error contains the error message.)

try is implemented using tryCatch; for programming, instead of try(expr, silent=TRUE), something like tryCatch(expr, error = function(e) e) (or other simple error handler functions) may be more efficient and flexible.

Value

The value of the expression if expr is evaluated without error, but an invisible object of class "try-error" containing the error message, and the error condition as the "condition" attribute, if it fails.

See Also

options for setting error handlers and suppressing the printing of error messages; geterrmessage for retrieving the last error message. tryCatch provides another means of catching and handling errors.

Examples

## this example will not work correctly in example(try), but
## it does work correctly if pasted in
options(show.error.messages = FALSE)
try(log("a"))
print(.Last.value)
options(show.error.messages = TRUE)

## alternatively,
print(try(log("a"), TRUE))
try

## run a simulation, keep only the results that worked.
set.seed(123)
x <- stats::rnorm(50)
doit <- function(x) {
    x <- sample(x, replace=TRUE)
    if(length(unique(x)) > 30) mean(x)
    else stop("too few unique points")
}
## alternative 1
res <- lapply(1:100, function(i) try(doit(x), TRUE))
## alternative 2
## Not run: res <- vector("list", 100)
for(i in 1:100) res[[i]] <- try(doit(x), TRUE)
## End(Not run)
unlist(res[sapply(res, function(x) !inherits(x, "try-error"))])
Interactive Tracing and Debugging of Calls to a Function or Method

Description

A call to `trace` allows you to insert debugging code (e.g., a call to `browser` or `recover`) at chosen places in any function. A call to `untrace` cancels the tracing. Specified methods can be traced the same way, without tracing all calls to the function. Trace code can be any R expression. Tracing can be temporarily turned on or off globally by calling `tracingState`.

Usage

```r
trace(what, tracer, exit, at, print, signature,
     where = topenv(parent.frame()), edit = FALSE)
untrace(what, signature = NULL, where = topenv(parent.frame()))

tracingState(on = NULL)
doTrace(expr, msg)
```

Arguments

- `what` The name (quoted or not) of a function to be traced or untraced. For `untrace` or for `trace` with more than one argument, more than one name can be given in the quoted form, and the same action will be applied to each one.
- `tracer` Either a function or an unevaluated expression. The function will be called or the expression will be evaluated either at the beginning of the call, or before those steps in the call specified by the argument `at`. See the details section.
- `exit` Either a function or an unevaluated expression. The function will be called or the expression will be evaluated on exiting the function. See the details section.
- `at` optional numeric vector or list. If supplied, `tracer` will be called just before the corresponding step in the body of the function. See the details section.
- `print` If `TRUE` (as per default), a descriptive line is printed before any trace expression is evaluated.
- `signature` If this argument is supplied, it should be a signature for a method for function `what`. In this case, the method, and not the function itself, is traced.
- `edit` For complicated tracing, such as tracing within a loop inside the function, you will need to insert the desired calls by editing the body of the function. If so, supply the `edit` argument either as `TRUE`, or as the name of the editor you want to use. Then `trace()` will call `edit` and use the version of the function after you edit it. See the details section for additional information.
- `where` where to look for the function to be traced; by default, the top-level environment of the call to `trace`.

An important use of this argument is to trace a function when it is called from a package with a namespace. The current namespace mechanism imports the functions to be called (with the exception of functions in the base package). The functions being called are not the same objects seen from the top-level (in general, the imported packages may not even be attached). Therefore, you must ensure that the correct versions are being traced. The way to do this is to set argument `where` to a function in the namespace. The tracing computations will
then start looking in the environment of that function (which will be the namespace of the corresponding package). (Yes, it’s subtle, but the semantics here are central to how namespaces work in R.)

on logical; a call to the support function tracingState returns TRUE if tracing is globally turned on, FALSE otherwise. An argument of one or the other of those values sets the state. If the tracing state is FALSE, none of the trace actions will actually occur (used, for example, by debugging functions to shut off tracing during debugging).

expr, msg arguments to the support function .doTrace, calls to which are inserted into the modified function or method: expr is the tracing action (such as a call to browser()), and msg is a string identifying the place where the trace action occurs.

Details

The trace function operates by constructing a revised version of the function (or of the method, if signature is supplied), and assigning the new object back where the original was found. If only the what argument is given, a line of trace printing is produced for each call to the function (back compatible with the earlier version of trace).

The object constructed by trace is from a class that extends "function" and which contains the original, untraced version. A call to untrace re-assigns this version.

If the argument tracer or exit is the name of a function, the tracing expression will be a call to that function, with no arguments. This is the easiest and most common case, with the functions browser and recover the likeliest candidates; the former browses in the frame of the function being traced, and the latter allows browsing in any of the currently active calls.

The tracer or exit argument can also be an unevaluated expression (such as returned by a call to quote or substitute). This expression itself is inserted in the traced function, so it will typically involve arguments or local objects in the traced function. An expression of this form is useful if you only want to interact when certain conditions apply (and in this case you probably want to supply print=FALSE in the call to trace also).

When the at argument is supplied, it can be a vector of integers referring to the substeps of the body of the function (this only works if the body of the function is enclosed in {...}). In this case tracer is not called on entry, but instead just before evaluating each of the steps listed in at. (Hint: you don’t want to try to count the steps in the printed version of a function; instead, look at as.list(body(f)) to get the numbers associated with the steps in function f.)

The at argument can also be a list of integer vectors. In this case, each vector refers to a step nested within another step of the function. For example, at = list(c(3,4)) will call the tracer just before the fourth step of the third step of the function. See the example below.

Using setBreakpoint (from package utils) may be an alternative, calling trace(..., at, ...).

An intrinsic limitation in the exit argument is that it won’t work if the function itself uses on.exit, since the existing calls will override the one supplied by trace.

Tracing does not nest. Any call to trace replaces previously traced versions of that function or method (except for edited versions as discussed below), and untrace always restores an untraced version. (Allowing nested tracing has too many potentials for confusion and for accidentally leaving traced versions behind.)

When the edit argument is used repeatedly with no call to untrace on the same function or method in between, the previously edited version is retained. If you want to throw away all the previous tracing and then edit, call untrace before the next call to trace. Editing may be combined with
automatic tracing; just supply the other arguments such as tracer, and the edit argument as well. The edit=TRUE argument uses the default editor (see edit).

Tracing primitive functions (builtns and specials) from the base package works, but only by a special mechanism and not very informatively. Tracing a primitive causes the primitive to be replaced by a function with argument ...(only). You can get a bit of information out, but not much. A warning message is issued when trace is used on a primitive.

The practice of saving the traced version of the function back where the function came from means that tracing carries over from one session to another, if the traced function is saved in the session image. (In the next session, untrace will remove the tracing.) On the other hand, functions that were in a package, not in the global environment, are not saved in the image, so tracing expires with the session for such functions.

Tracing a method is basically just like tracing a function, with the exception that the traced version is stored by a call to setMethod rather than by direct assignment, and so is the untraced version after a call to untrace.

The version of trace described here is largely compatible with the version in S-Plus, although the two work by entirely different mechanisms. The S-Plus trace uses the session frame, with the result that tracing never carries over from one session to another (R does not have a session frame). Another relevant distinction has nothing directly to do with trace: The browser in S-Plus allows changes to be made to the frame being browsed, and the changes will persist after exiting the browser. The R browser allows changes, but they disappear when the browser exits. This may be relevant in that the S-Plus version allows you to experiment with code changes interactively, but the R version does not. (A future revision may include a ‘destructive’ browser for R.)

Value

In the simple version (just the first argument), invisible NULL. Otherwise, the traced function(s) name(s). The relevant consequence is the assignment that takes place.

Note

The version of function tracing that includes any of the arguments except for the function name requires the methods package (because it uses special classes of objects to store and restore versions of the traced functions).

If methods dispatch is not currently on, trace will load the methods namespace, but will not put the methods package on the search list.

References


See Also

browser and recover, the likeliest tracing functions; also, quote and substitute for constructing general expressions.

Examples

require(graphics)

### Very simple use
trace(sum)
trace

hist(stats::rnorm(100)) # shows about 3-4 calls to sum()
untrace(sum)

if(.isMethodsDispatchOn()) { # non-simple use needs 'methods' package
  f <- function(x, y) {
    y <- pmax(y, 0.001)
    if (x > 0) x ^ y else stop("x must be positive")
  }

  ## arrange to call the browser on entering and exiting
  ## function f
  trace("f", quote(browser(skipCalls=4)),
    exit = quote(browser(skipCalls=4)))

  ## instead, conditionally assign some data, and then browse
  ## on exit, but only then. Don’t bother me otherwise
  trace("f", quote(if(any(y < 0)) yOrig <- y),
    exit = quote(if(exists("yOrig")) browser(skipCalls=4),
      print = FALSE)

  ## Enter the browser just before stop() is called. First, find
  ## the step numbers
  as.list(body(f))
  as.list(body(f)[[3]])

  ## Now call the browser there
  trace("f", quote(browser(skipCalls=4)), at=list(c(3,4)))

  ## trace a utility function, with recover so we
  ## can browse in the calling functions as well.
  trace("as.matrix", recover)

  ## turn off the tracing
  untrace(c("f", "as.matrix"))

  ## Not run:
  ## trace calls to the function lm() that come from
  ## the nlme package.
  ## (The function nlme is in that package, and the package
  ## has a namespace, so the where= argument must be used
  ## to get the right version of lm)
  trace(lm, exit = recover, where = nlme)

  ## End(Not run)
}
Description

These functions provide a mechanism for handling unusual conditions, including errors and warnings.

Usage

\[
\begin{align*}
\text{tryCatch}(\text{expr, ..., finally}) \\
\text{withCallingHandlers}(\text{expr, ...}) \\
\text{signalCondition}(\text{cond}) \\
\text{simpleCondition}(\text{message, call = NULL}) \\
\text{simpleError}(\text{message, call = NULL}) \\
\text{simpleWarning}(\text{message, call = NULL}) \\
\text{simpleMessage}(\text{message, call = NULL}) \\
\text{as.character}(\text{x, ...}) \\
\text{as.character}(\text{x, ...}) \\
\text{print}(\text{x, ...}) \\
\text{conditionCall}(\text{c}) \\
\text{conditionCall}(\text{c}) \\
\text{conditionMessage}(\text{c}) \\
\text{conditionMessage}(\text{c}) \\
\text{withRestarts}(\text{expr, ...}) \\
\text{computeRestarts}(\text{cond = NULL}) \\
\text{findRestart}(\text{name, cond = NULL}) \\
\text{invokeRestart}(\text{r, ...}) \\
\text{invokeRestartInteractively}(\text{r}) \\
\text{isRestart}(\text{x}) \\
\text{restartDescription}(\text{r}) \\
\text{restartFormals}(\text{r}) \\
\text{.signalSimpleWarning}(\text{msg, call}) \\
\text{.handleSimpleError}(\text{h, msg, call})
\end{align*}
\]
**Arguments**

- `c`: a condition object.
- `call`: call expression.
- `cond`: a condition object.
- `expr`: expression to be evaluated.
- `finally`: expression to be evaluated before returning or exiting.
- `h`: function.
- `message`: character string.
- `msg`: character string.
- `name`: character string naming a restart.
- `r`: restart object.
- `x`: object.
- `...`: additional arguments; see details below.

**Details**

The condition system provides a mechanism for signaling and handling unusual conditions, including errors and warnings. Conditions are represented as objects that contain information about the condition that occurred, such as a message and the call in which the condition occurred. Currently conditions are S3-style objects, though this may eventually change.

Conditions are objects inheriting from the abstract class condition. Errors and warnings are objects inheriting from the abstract subclasses `error` and `warning`. The class `simpleError` is the class used by `stop` and all internal error signals. Similarly, `simpleWarning` is used by `warning`, and `simpleMessage` is used by `message`. The constructors by the same names take a string describing the condition as argument and an optional call. The functions `conditionMessage` and `conditionCall` are generic functions that return the message and call of a condition.

Conditions are signaled by `signalCondition`. In addition, the `stop` and `warning` functions have been modified to also accept condition arguments.

The function `tryCatch` evaluates its expression argument in a context where the handlers provided in the `...` argument are available. The `finally` expression is then evaluated in the context in which `tryCatch` was called; that is, the handlers supplied to the current `tryCatch` call are not active when the `finally` expression is evaluated.

Handlers provided in the `...` argument to `tryCatch` are established for the duration of the evaluation of `expr`. If no condition is signaled when evaluating `expr` then `tryCatch` returns the value of the expression.

If a condition is signaled while evaluating `expr` then established handlers are checked, starting with the most recently established ones, for one matching the class of the condition. When several handlers are supplied in a single `tryCatch` then the first one is considered more recent than the second. If a handler is found then control is transferred to the `tryCatch` call that established the handler, the handler found and all more recent handlers are disestablished, the handler is called with the condition as its argument, and the result returned by the handler is returned as the value of the `tryCatch` call.

Calling handlers are established by `withCallingHandlers`. If a condition is signaled and the applicable handler is a calling handler, then the handler is called by `signalCondition` in the context where the condition was signaled but with the available handlers restricted to those below the handler called in the handler stack. If the handler returns, then the next handler is tried; once the last handler has been tried, `signalCondition` returns `NULL`. 
User interrupts signal a condition of class `interrupt` that inherits directly from class `condition` before executing the default interrupt action.

Restarts are used for establishing recovery protocols. They can be established using `withRestarts`. One pre-established restart is an `abort` restart that represents a jump to top level. `findRestart` and `computeRestarts` find the available restarts. `findRestart` returns the most recently established restart of the specified name. `computeRestarts` returns a list of all restarts. Both can be given a condition argument and will then ignore restarts that do not apply to the condition.

`invokeRestart` transfers control to the point where the specified restart was established and calls the restart's handler with the arguments, if any, given as additional arguments to `invokeRestart`. The restart argument to `invokeRestart` can be a character string, in which case `findRestart` is used to find the restart.

New restarts for `withRestarts` can be specified in several ways. The simplest is in name=function form where the function is the handler to call when the restart is invoked. Another simple variant is as name=string where the string is stored in the description field of the restart object returned by `findRestart`; in this case the handler ignores its arguments and returns NULL. The most flexible form of a restart specification is as a list that can include several fields, including handler, description, and test. The test field should contain a function of one argument, a condition, that returns TRUE if the restart applies to the condition and FALSE if it does not; the default function returns TRUE for all conditions.

One additional field that can be specified for a restart is interactive. This should be a function of no arguments that returns a list of arguments to pass to the restart handler. The list could be obtained by interacting with the user if necessary. The function `invokeRestartInteractively` calls this function to obtain the arguments to use when invoking the restart. The default interactive method queries the user for values for the formal arguments of the handler function.

`.signalSimpleWarning` and `.handleSimpleError` are used internally and should not be called directly.

References

The `tryCatch` mechanism is similar to Java error handling. Calling handlers are based on Common Lisp and Dylan. Restarts are based on the Common Lisp restart mechanism.

See Also

`stop` and `warning` signal conditions, and `try` is essentially a simplified version of `tryCatch`.

Examples

```r
tryCatch(1, finally=print("Hello"))
e <- simpleError("test error")
## Not run:
stop(e)
tryCatch(stop(e), finally=print("Hello"))
tryCatch(stop("fred"), finally=print("Hello"))
## End(Not run)
tryCatch(stop(e), error = function(e) e, finally=print("Hello"))
tryCatch(stop("fred"), error = function(e) e, finally=print("Hello"))
withCallingHandlers({
  warning("A"); 1+2
}, warning = function(w) { })
## Not run:
{ withRestarts(stop("A"), abort = function() { }); 1 }
```
```r
## End(Not run)
withRestarts(invokeRestart("foo", 1, 2), foo = function(x, y) {x + y})

##--> More examples are part of
##--> demo(error.catching)
```
Description
A call of gc causes a garbage collection to take place. gcinfo sets a flag so that automatic collection is either silent (verbose=FALSE) or prints memory usage statistics (verbose=TRUE).

Usage
```r
gc(verbose = getOption("verbose"), reset=FALSE)
gcinfo( verbose)
```

Arguments
- `verbose` logical; if TRUE, the garbage collection prints statistics about cons cells and the space allocated for vectors.
- `reset` logical; if TRUE the values for maximum space used are reset to the current values.

Details
A call of gc causes a garbage collection to take place. This will also take place automatically without user intervention, and the primary purpose of calling gc is for the report on memory usage. However, it can be useful to call gc after a large object has been removed, as this may prompt R to return memory to the operating system.

R allocates space for vectors in multiples of 8 bytes: hence the report of "Vcells", a relict of an earlier allocator (that used a vector heap).

When gcinfo(TRUE) is in force, messages are sent to the message connection at each garbage collection of the form

```
Garbage collection 12 = 10+0+2 (level 0) ...
6.4 Mbytes of cons cells used (58%)
2.0 Mbytes of vectors used (32%)
```

Here the last two lines give the current memory usage rounded up to the next 0.1Mb and as a percentage of the current trigger value. The first line gives a breakdown of the number of garbage collections at various levels (for an explanation see the ‘R Internals’ manual).

Value
gc returns a matrix with rows "Ncells" (cons cells), usually 28 bytes each on 32-bit systems and 56 bytes on 64-bit systems, and "Vcells" (vector cells, 8 bytes each), and columns "used" and "gc trigger", each also interpreted in megabytes (rounded up to the next 0.1Mb).

If maxima have been set for either "Ncells" or "Vcells", a fifth column is printed giving the current limits in Mb (with NA denoting no limit).

The final two columns show the maximum space used since the last call to gc(reset=TRUE) (or since R started).

gcinfo returns the previous value of the flag.
See Also

The ‘R Internals’ manual.

Memory on R’s memory management, and gctorture if you are an R developer.

reg.finalizer for actions to happen at garbage collection.

Examples

gc() #-- do it now
gcinfo(TRUE) #-- in the future, show when R does it
x <- integer(100000); for(i in 1:18) x <- c(x,i)
gcinfo(verbose = FALSE)#-- don't show it anymore

gc(TRUE)

gc(reset=TRUE)
**Description**

This function reports the time spent in garbage collection so far in the \( \mathbb{R} \) session while GC timing was enabled.

**Usage**

\[
gc.time(on = \text{TRUE})
\]

**Arguments**

- **on** logical; if TRUE, GC timing is enabled.

**Details**

The timings are rounded up by the sampling interval for timing processes, and so are likely to be over-estimates.

It is a primitive.

**Value**

A numerical vector of length 5 giving the user CPU time, the system CPU time, the elapsed time and children’s user and system CPU times (normally both zero), of time spent doing garbage collection whilst GC timing was enabled.

Times of child processes are not available on Windows and will always be given as NA.

**See Also**

- `gc.proc.time` for the timings for the session.

**Examples**

\[
gc.time()
\]


Description

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

Usage

gctorture(on = TRUE)
gctorture2(step, wait = step, inhibit_release = FALSE)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>logical; turning it on/off.</td>
</tr>
<tr>
<td>step</td>
<td>integer; run GC every step allocations; step = 0 turns the GC torture off.</td>
</tr>
<tr>
<td>wait</td>
<td>integer; number of allocations to wait before starting GC torture.</td>
</tr>
<tr>
<td>inhibit_release</td>
<td>logical; do not release free objects for re-use: use with caution.</td>
</tr>
</tbody>
</table>

Details

Calling gctorture(TRUE) instructs the memory manager to force a full GC on every allocation. gctorture2 provides a more refined interface that allows the start of the GC torture to be deferred and also gives the option of running a GC only every step allocations.

The third argument to gctorture2 is only used if R has been configured with a strict write barrier enabled. When this is the case all garbage collections are full collections, and the memory manager marks free nodes and enables checks in many situations that signal an error when a free node is used. This can greatly help in isolating unprotected values in C code. It does not detect the case where a node becomes free and is reallocated. The inhibit_release argument can be used to prevent such reallocation. This will cause memory to grow and should be used with caution and in conjunction with operating system facilities to monitor and limit process memory use.

Value

Previous value of first argument.

Author(s)

Peter Dalgaard and Luke Tierney
memory.profile  Profile the Usage of Cons Cells

Description

Lists the usage of the cons cells by SEXPREC type.

Usage

memory.profile()

Details

The current types and their uses are listed in the include file ‘Rinternals.h’.

Value

A vector of counts, named by the types. See typeof for an explanation of types.

See Also

gc for the overall usage of cons cells. Rprofmem and tracemem allow memory profiling of specific code or objects, but need to be enabled at compile time.

Examples

memory.profile()
system.time

Description

Return CPU (and other) times that expr used.

Usage

system.time(expr, gcFirst = TRUE)
unix.time(expr, gcFirst = TRUE)

Arguments

expr Valid \texttt{R} expression to be timed.
gcFirst Logical - should a garbage collection be performed immediately before the timing? Default is \texttt{TRUE}.

Details

system.time calls the function \texttt{proc.time}, evaluates expr, and then calls \texttt{proc.time} once more, returning the difference between the two \texttt{proc.time} calls.

\texttt{unix.time} is an alias of \texttt{system.time}, for compatibility with \texttt{S}.

Timings of evaluations of the same expression can vary considerably depending on whether the evaluation triggers a garbage collection. When \texttt{gcFirst} is \texttt{TRUE} a garbage collection (\texttt{gc}) will be performed immediately before the evaluation of \texttt{expr}. This will usually produce more consistent timings.

Value

A object of class "\texttt{proc.time}": see \texttt{proc.time} for details.

See Also

\texttt{proc.time}, \texttt{time} which is for time series.

Examples

```r
require(stats)
system.time(for(i in 1:100) mad(runif(1000)))
## Not run:
exT <- function(n = 1/zero.noslash/zero.noslash/zero.noslash/zero.noslash) {
  # Purpose: Test if system.time works ok; n: loop size
  system.time(for(i in 1:n) x <- mean(rt(1/zero.noslash/zero.noslash/zero.noslash/zero.noslash, df=4)))
}
#-- Try to interrupt one of the following (using Ctrl-C / Escape):
exT()  #-- about 4 secs on a 2.5GHz Xeon
system.time(exT())  #-- +/- same
## End(Not run)
```
**proc.time**

### Description

`proc.time` determines how much real and CPU time (in seconds) the currently running R process has already taken.

### Usage

```r
proc.time()
```

### Details

`proc.time` returns five elements for backwards compatibility, but its `print` method prints a named vector of length 3. The first two entries are the total user and system CPU times of the current R process and any child processes on which it has waited, and the third entry is the 'real' elapsed time since the process was started.

### Value

An object of class "proc_time" which is a numeric vector of length 5, containing the user, system, and total elapsed times for the currently running R process, and the cumulative sum of user and system times of any child processes spawned by it on which it has waited. (The `print` method combines the child times with those of the main process.)

The definition of 'user' and 'system' times is from your OS. Typically it is something like

*The 'user time' is the CPU time charged for the execution of user instructions of the calling process.*

*The 'system time' is the CPU time charged for execution by the system on behalf of the calling process.*

Times of child processes are not available on Windows and will always be given as NA.

The resolution of the times will be system-specific and on Unix-alikes times are rounded down to milliseconds. On modern systems they will be that accurate, but on older systems they might be accurate to 1/100 or 1/60 sec. They are typically available to 10ms on Windows.

This is a primitive function.

### References


### See Also

`system.time` for timing an R expression, `gc.time` for how much of the time was spent in garbage collection.
Examples

## Not run:
## a way to time an R expression: system.time is preferred
ptm <- proc.time()
for (i in 1:50) mad(stats::runif(500))
proc.time() - ptm

## End(Not run)
setTimeLimit  Set CPU and/or Elapsed Time Limits

Description

Functions to set CPU and/or elapsed time limits for top-level computations or the current session.

Usage

```r
setTimeLimit(cpu = Inf, elapsed = Inf, transient = FALSE)
setSessionTimeLimit(cpu = Inf, elapsed = Inf)
```

Arguments

- `cpu` double. Limit on total cpu time.
- `elapsed` double. Limit on elapsed time.
- `transient` logical. If TRUE, the limits apply only to the rest of the current computation.

Details

`setTimeLimit` sets limits which apply to each top-level computation, that is a command line (including any continuation lines) entered at the console or from a file. If it is called from within a computation the limits apply to the rest of the computation and (unless `transient = TRUE`) to subsequent top-level computations.

`setSessionTimeLimit` sets limits for the rest of the session. Once a session limit is reached it is reset to Inf.

Setting any limit has a small overhead – well under 1% on the systems measured.

Time limits are checked whenever a user interrupt could occur. This will happen frequently in R code and during `Sys.sleep`, but only at points in compiled C and Fortran code identified by the code author.

‘Total cpu time’ includes that used by child processes where the latter is reported.
Description

Enable or disable profiling of the execution of R expressions.

Usage

Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02,
memory.profiling=FALSE)

Arguments

filename The file to be used for recording the profiling results. Set to NULL or "" to disable profiling.
append logical: should the file be over-written or appended to?
interval real: time interval between samples.
memory.profiling logical: write memory use information to the file?

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling works by writing out the call stack every interval seconds, to the file specified. Either the summaryRprof function or the wrapper script R CMD Rprof can be used to process the output file to produce a summary of the usage; use R CMD Rprof --help for usage information.

How time is measured varies by platform: on a Unix-alike it is the CPU time of the R process, so for example excludes time when R is waiting for input or for processes run by system to return.

Note that the timing interval cannot usefully be too small: once the timer goes off, the information is not recorded until the next timing click (probably in the range 1–10msecs).

Functions will only be recorded in the profile log if they put a context on the call stack (see sys.calls). Some primitive functions do not do so: specifically those which are of type "special" (see the ‘R Internals’ manual for more details).

Note

Profiling is not available on all platforms. By default, support for profiling is compiled in if possible – configure R with ‘--disable-R-profiling’ to change this.

As R profiling uses the same mechanisms as C profiling, the two cannot be used together, so do not use Rprof in an executable built for C-level profiling.

See Also

The chapter on “Tidying and profiling R code” in “Writing R Extensions” (see the ‘doc/manual’ subdirectory of the R source tree).

summaryRprof
tracemem, Rprofmem for other ways to track memory use.
Examples

## Not run: Rprof()
## some code to be profiled
Rprof(NULL)
## some code NOT to be profiled
Rprof(append=TRUE)
## some code to be profiled
Rprof(NULL)
...
## Now post-process the output as described in Details

## End(Not run)
**summaryRprof**

*Summarise Output of R Sampling Profiler*

**Description**

Summarise the output of the `Rprof` function to show the amount of time used by different R functions.

**Usage**

```r
summaryRprof(filename = "Rprof.out", chunksize = 5000, 
memory=c("none","both","tseries","stats"), 
index=2, diff=TRUE, exclude=NULL)
```

**Arguments**

- `filename` Name of a file produced by `Rprof()`
- `chunksize` Number of lines to read at a time
- `memory` Summaries for memory information. See ‘Details’ below
- `index` How to summarize the stack trace for memory information. See ‘Details’ below.
- `diff` If TRUE memory summaries use change in memory rather than current memory
- `exclude` Functions to exclude when summarizing the stack trace for memory summaries

**Details**

This function provides the analysis code for `Rprof` files used by `R CMD Rprof`.

As the profiling output file could be larger than available memory, it is read in blocks of chunksize lines. Increasing chunksize will make the function run faster if sufficient memory is available.

When called with `memory.profiling = TRUE`, the profiler writes information on three aspects of memory use: vector memory in small blocks on the R heap, vector memory in large blocks (from `malloc`), memory in nodes on the R heap. It also records the number of calls to the internal function `duplicate` in the time interval. `duplicate` is called by C code when arguments need to be copied. Note that the profiler does not track which function actually allocated the memory.

With `memory = "both"` the change in total memory (truncated at zero) is reported in addition to timing data.

With `memory = "tseries"` or `memory = "stats"` the index argument specifies how to summarize the stack trace. A positive number specifies that many calls from the bottom of the stack; a negative number specifies the number of calls from the top of the stack. With `memory = "tseries"` the index is used to construct labels and may be a vector to give multiple sets of labels. With `memory = "stats"` the index must be a single number and specifies how to aggregate the data to the maximum and average of the memory statistics. With both `memory = "tseries"` and `memory = "stats"` the argument `diff = TRUE` asks for summaries of the increase in memory use over the sampling interval and `diff = FALSE` asks for the memory use at the end of the interval.
Value

If memory = "none", a list with components

by.self Timings sorted by ‘self’ time

by.total Timings sorted by ‘total’ time

sample.interval The sampling interval

sampling.time Total length of profiling run

If memory = “both” the same list but with memory consumption in Mb in addition to the timings.

If memory = "tseries" a data frame giving memory statistics over time

If memory = "stats" a by object giving memory statistics by function.

See Also

The chapter on “Tidying and profiling R code” in “Writing R Extensions” (see the ‘doc/manual’ subdirectory of the R source tree).

Rprof

tracemem traces copying of an object via the C function duplicate.

Rprofmem is a non-sampling memory-use profiler.

http://developer.r-project.org/memory-profiling.html

Examples

## Not run:
## Rprof() is not available on all platforms
Rprof(tmp <- tempfile())
example(glm)
Rprof()
summaryRprof(tmp)
unlink(tmp)

## End(Not run)
Description

This function marks an object so that a message is printed whenever the internal function `duplicate` is called. This happens when two objects share the same memory and one of them is modified. It is a major cause of hard-to-predict memory use in R.

Usage

```r
tracemem(x)
untracemem(x)
retracemem(x, previous = NULL)
```

Arguments

- `x` An R object, not a function or environment or `NULL`.
- `previous` A value as returned by `tracemem` or `retracemem`.

Details

This functionality is optional, determined at compilation, because it makes R run a little more slowly even when no objects are being traced. `tracemem` and `untracemem` give errors when R is not compiled with memory profiling; `retracemem` does not (so it can be left in code during development).

When an object is traced any copying of the object by the C function `duplicate` or by arithmetic or mathematical operations produces a message to standard output. The message consists of the string `tracemem`, the identifying strings for the object being copied and the new object being created, and a stack trace showing where the duplication occurred. `retracemem()` is used to indicate that a variable should be considered a copy of a previous variable (e.g. after subcripting).

The messages can be turned off with `tracingState`.

It is not possible to trace functions, as this would conflict with `trace` and it is not useful to trace `NULL`, environments, promises, weak references, or external pointer objects, as these are not duplicated.

These functions are primitive.

Value

A character string for identifying the object in the trace output (an address in hex enclosed in angle brackets), or `NULL` (invisibly).

See Also

- `trace`, `Rprofmem`
- http://developer.r-project.org/memory-profiling.html
Examples

```r
## Not run:
a <- 1:10
tracemem(a)
## b and a share memory
b <- a
b[1] <- 1
untracemem(a)

## copying in lm
d <- stats::rnorm(10)
tracemem(d)
lm(d ~ a+log(b))

## f is not a copy and is not traced
f <- d[-1]
f+1
## indicate that f should be traced as a copy of d
retracemem(f, retracemem(d))
f+1

## End(Not run)
```
Rprofmem

Enable Profiling of R’s Memory Use

Description

Enable or disable reporting of memory allocation in R.

Usage

Rprofmem(filename = "Rprofmem.out", append = FALSE, threshold = 0)

Arguments

filename  The file to be used for recording the memory allocations. Set to NULL or "" to disable reporting.
append    logical: should the file be over-written or appended to?
threshold numeric: allocations on R’s "large vector” heap larger than this number of bytes will be reported.

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling writes the call stack to the specified file every time malloc is called to allocate a large vector object or to allocate a page of memory for small objects. The size of a page of memory and the size above which malloc is used for vectors are compile-time constants, by default 2000 and 128 bytes respectively.

The profiler tracks allocations, some of which will be to previously used memory and will not increase the total memory use of R.

Value

None

Note

The memory profiler slows down R even when not in use, and so is a compile-time option. The memory profiler can be used at the same time as other R and C profilers.

See Also

The R sampling profiler, Rprof also collects memory information.

tracemem traces duplications of specific objects.

The "Writing R Extensions” manual section on “Tidying and profiling R code"
Examples

## Not run:
## not supported unless R is compiled to support it.
Rprofmem("Rprofmem.out", threshold=1000)
exmple(glm)
Rprofmem(NULL)
noquote(readLines("Rprofmem.out", n=5))

## End(Not run)
Index

*Topic debugging
  recover, 10
  trace, 14
*Topic environment
  browser, 6
gc, 22
gctorture, 25
Memory, 1
  Memory-limits, 2
*Topic error
  conditions, 18
*Topic programming
  browser, 6
conditions, 18
dbg, 4
  recover, 10
  trace, 14
  traceback, 8
try, 12
*Topic utilities
  gc.time, 24
  memory.profile, 26
  proc.time, 28
  Rprof, 31
  Rprofmem, 37
  setLimit, 30
  summaryRprof, 33
  system.time, 27
  traceback, 35
.Call, 8
.Traceback (traceback), 8
.doTrace (trace), 14
.handleSimpleError (conditions), 18
.signalSimpleWarning (conditions), 18
as.character.condition (conditions), 18
as.character.error (conditions), 18
computeRestarts (conditions), 18
condition (conditions), 18
  conditionCall (conditions), 18
  conditionMessage (conditions), 18
  conditions, 18
  debug, 4, 6, 7
dbgonce (dbg), 4
dump.frames, 10
edit, 14, 16
findRestart (conditions), 18
gc, 1, 22, 24, 26, 27
gc.time, 24, 28
gcinfo, 1
gcinfo (gc), 22
gctorture, 23, 25
gctorture2 (gctorture), 25
geterrmessage, 12
invokeRestart (conditions), 18
invokeRestartInteractively (conditions), 18
isdbugged (debug), 4
isRestart (conditions), 18
limitedLabels (recover), 10
Memory, 1, 2, 23
  Memory-limits, 1
  Memory-limits, 2
  memory.limit, 3
  memory.profile, 1, 26
  memory.size, 3
Methods, 5
.object.size, 1, 3
option, 4, 7
options, 5, 7, 10, 12
primitive, 24, 28, 31, 35
print.condition (conditions), 18
print.proc.time (proc.time), 28
print.restart (conditions), 18
proc.time, 24, 27, 28

quote, 15, 16

recover, 5, 10, 14–16
reg.finalizer, 23
restartDescription (conditions), 18
restartFormals (conditions), 18
retracemem (tracemem), 35
Rprof, 31, 33, 34, 37
Rproffmem, 26, 31, 34, 35, 37

setBreakpoint, 5, 15
setMethod, 16
setSessionTimeLimit (setTimeLimit), 30
setTimeLimit, 30
signalCondition (conditions), 18
simpleCondition (conditions), 18
simpleError (conditions), 18
simpleMessage (conditions), 18
simpleWarning (conditions), 18
stderr, 12
stop, 20
substitute, 15, 16
summaryRprof, 31, 33
sys.calls, 31
Sys.sleep, 30
system, 31
system.time, 27, 28
time, 27
trace, 5, 14, 35
traceback, 5, 7, 8
tracemem, 26, 31, 34, 35, 37
tracingState, 35
tracingState (trace), 14
try, 8, 12, 20
tryCatch, 8, 12
tryCatch (conditions), 18
type, 31
typeof, 26

undebug (debug), 4
unix.time (system.time), 27
untrace (trace), 14
untracemem (tracemem), 35

warning, 20
withCallingHandlers (conditions), 18
withRestarts (conditions), 18