Package: debug (via r-universe)

September 13, 2024

Version 1.4.18

Author Mark V. Bravington <markb2@summerinsouth.net>

Maintainer Mark V. Bravington <markb2@summerinsouth.net>

Depends R (>= 4.0), tcltk

Imports utils, mvbutils (>= 2.7)

NeedsCompilation no

ByteCompile false

Title MVB's Debugger for R

Description Proper debugger for R functions and scripts, with code display, graceful error recovery, line-numbered conditional breakpoints, access to exit code, flow control, and full keyboard input.

License GPL (>=2)

Repository https://markbravington.r-universe.dev

RemoteUrl https://github.com/markbravington/debug

RemoteRef HEAD

RemoteSha 1ae9442fd112fc515bcad3d9d5cab575b60fdc32

Contents

`debug-package` *How to use the debug package*

Description

`debug` is an alternative to `trace` and `browser`, offering:

- a visible code window with line-numbered code and highlighted execution point;
- the ability to set (conditional) breakpoints in advance, at any line number;
- the opportunity to keep going after errors;
- multiple debugging windows open at once (when one debuggee calls another, or itself);
- full debugging of `on.exit` code;
- the ability to move the execution point around without executing intervening statements;
- direct interpretation of typed-in statements, as if they were in the function itself;
- debugging of scripts– i.e. text files or character vectors (version >= 1.3.11).
- debugging of vignettes (version >= 1.3.66)
- `browser()`-like functionality (a minor thing)

Even if you don't write functions, or even if you don't write buggy functions, you may find it helpful to run the debugger on functions in `package:base` or other packages. Watching what actually happens while a function is executing, can be much more informative than staring at a piece of code or terse documentation.

Debugging your function `f` is a two-stage process. First, call `mtrace(f)` to store debugging information on `f`, and to overwrite `f` with a debug-ready version that will call the debugger itself. Second, do whatever you normally do at the command prompt to invoke `f`. This is often a direct call to `f`, but can be any command that eventually results in `f` being invoked. [The third, fourth, etc. stages, in which you actually fix the problem, are not covered here!]

When `f` is invoked, a window will appear at the bottom of the screen, showing the code of `f` with a highlight on the first numbered line. (There is also an asterisk in the far left hand column of the same row, showing that there's a breakpoint.) The command prompt in R will change to "D(...)> ", showing that you are "inside" a function that is being debugged. The debugger is now in "step mode". Anything you type will be evaluated in the frame of the function– this includes assignments and creation of new variables. If you just type <ENTER>, the highlighted statement in `f` will be executed. The result of the statement will be printed in the R command window, and the highlight will (probably) move in the `f` code window.

To progress through the code of f, you can keep pressing <ENTER>, but you can also type go() to put the debugger into "go mode", whereupon it will keep executing code statements without manual intervention. In "go mode", nothing will be printed in the command window (except if there are cat or print calls in the function code) until either:

- the function completes normally, or
- an error occurs in the function, or
- there's a user interrupt (e.g. ESCAPE is pressed), or
- a breakpoint is triggered.

In the first case, control is returned to the normal R command prompt, just as if the debugger had not been used. In the other cases, the D(...)> prompt will return and the line associated with the error / interrupt / breakpoint will be highlighted in the code window. You are then back in step mode. If there was an error, you can type statement(s) that will cause the error not to happen when the highlighted line executes again, or you can move the highlighted execution point to another line number by calling skip. Execution carries on quite normally after errors, just as if the offending statement had been wrapped in a try call. If your function eventually exits normally (i.e. not via qqq(), as described next), it will be as if the error never happened (though the error message(s) will be displayed when the R command prompt returns).

When in step mode, you can finish debugging and return to the normal R command prompt by typing qqq(). If you type <ESC> while in go mode, you should be returned to step mode, but sometimes you may be dumped back at the R command prompt (this is on to-be-fixed list), and sometimes there will be no immediate effect (e.g. if C code is running).

Breakpoints, including conditional breakpoints, are set and cleared by bp. Calling go(n) puts the debugger into go mode, but also sets a temporary breakpoint at line n, which will be triggered the first time execution reaches line n but not subsequently.

When the main function code has completed, the debugger moves into any on.exit code, which is also displayed and line-numbered in the code window. (Even if there are no calls to on.exit, a numbered NULL statement is placed in the exit code section, so that you can always set a "run-until-return" breakpoint.) If you exit via qqq(), the exit code will not be executed first; this can lead to subsequent trouble with open connections, screwed-up par values in graphics, etc.. To make sure the exit code does get executed:

- use skip to move to the start of the exit code;
- then use go(n) to run to the final NULL in the exit code;
- then use qqq() to finish debugging.

When you want to restore f to its normal non-debugging state (and you are back at the real R command prompt), type mtrace(f,FALSE). To restore all debuggees, type mtrace.off(). It is advisable not to save functions in an mtraced state; to avoid manual untracing and retracing, look up Save in package **mvbutils**.

You can debug several functions "at once" (e.g. if f calls g, you can mtrace both f and g, with mtrace(g) called either inside or outside the debugger), causing several code windows to be open simultaneously. If f is called again inside f (either via some statement in f, or from something you type in step mode), another f code window will open. The number in the window title is the frame number, and the currently-active frame number is shown in the D(...)> prompt.

For statements typed in at the D(...)> prompt, only the first syntactically-complete R expression will be executed; thus, typing `a <- 1; a <- 2` will set `a` to 1, but typing `{ a <- 1; a <- 2}` will set `a` to 2.

See section **Special functions** for handling of `try`, `with`, etc.

See section **Methods** for how to handle S3 methods (easy), reference class methods (pretty easy), and S4 methods (not easy).

See section **Scripts and examples** for precisely that, and VIGNETTES.AND.NON.INTERACTIVE.USE for precisely that (though the latter has been temporarily disabled in v1.4 because something changed in R).

See section **Browser-like functionality** for ditto.

For further information, see R-news 3/3.

Methods

S3 methods work fine with `mtrace`; just do e.g. `mtrace( print.classofthing)`. Reference class methods aren't too bad either— see `?mtrace` for details. Unsurprisingly, S4 methods are much more painful to work with. I've only done it once; the following approach worked in R 2.12, but probably isn't optimal. Suppose you have figured out that the offending call is something like `scrunge( x, y, z)`, where `scrunge` is the name of an S4 generic; e.g. you may have tried `mtrace( scrunge)`, and found yourself with a debug window containing a 1-line function `standardGeneric("scrunge")`. First, use `findFunction( "scrunge")` to work out which package contains the definition of `scrunge`– say it's in package **scrungepack**. Next, you need to work out which specific `scrunge` method will be dispatched by `scrunge( x, y, z)`. Try this:

```
selectMethod( "scrunge", signature=character())
# Look for the 'attr(,"target")' line; it might be e.g.
# attr(,"target")
#   x     y
#  "ANY" "ANY"
```

Now you know that it's the `x` and `y` args that will direct traffic (it could have been just `x`, or just `z`, or...). So do `class(x)` and `class(y)` to figure out what the classes are; say they are `matrix` and `character`. Now do

```
selectMethod( "scrunge", sig=c( "matrix", "character"))
```

Hopefully you'll see another `attr(,"target")` line, which will tell you which method to `mtrace`. Suppose it's the same as before (ANY and ANY); then you need to mtrace the `ANY#ANY` method. (If only one argument was used for dispatching, there wouldn't be a hash symbol.) The following magic formula will do it:

```
mtrace( 'ANY#ANY', from=environment( scrungepack:::scrunge)$.AllMTable)
```

Then you can proceed as usual. Note that the method that is first dispatched may end up dispatching other methods– you will have to work out which yourself, and `mtrace` them accordingly. You can `mtrace` various functions in the **methods** package, e.g. `callGeneric`, which might help you track down what's going on. In short: good luck!

Special functions

Certain system functions with "code" arguments are handled specially in step-mode: currently `try`, `suppressWarnings`, `local`, `eval`, `evalq`, `with`, and `within`, plus `mvbutils::FOR` and `mvbutils::do.on`. In step-mode only, your code argument in these is stepped-into by default if it is more than a simple statement, using a new code window. In go-mode, and in step-mode if the default behaviour has been changed, these functions are handled entirely by R. Hence, if you are in go-mode and an error occurs in one of these statements, the debugger will stop at the `with` etc. statement, not inside it; but you can then step inside by pressing <ENTER>. The step-into-in-step-mode behaviour can be controlled globally using `step.into.sysfuns`. To avoid step-in at a specific line, you can also just use `go` to proceed to the following statement; this can be much faster than first stepping-in and then calling `go`, because R itself handles the evaluation.

To mimic the true behaviour of `try`, the debugger should really just return a `"try-error"` object if there is an error. However, that is probably not what you want when debugging. Instead, the debugger just behaves as usual with errors, i.e. it breaks into step-mode. If you do then want `try` to return the `try-error`, just as it would if you weren't debugging, type `return( last.try.error())`. NB: `tryCatch` is a bit too scarily complicated (I've always avoided it for that reason) so I haven't had a go at that one yet.

Note that the code window for `with`, `eval`, etc. contains an on-exit block, just like normal debugger code windows. Its main use is probably to let you set a breakpoint-on-return. However, it seems that you can successfully put `on.exit` statements inside your `eval` etc. call, whether debugging or not.

`with` and `within` are S3 generics, and the **debug** package only knows how to deal with the standard methods: currently `with.default`, `within.data.frame`, and `within.list`. You can debug specific methods manually, e.g. via `mtrace( with.myS3class)`.

`within` is more complicated than the others, and two extra code windows are currently used: one for the code of `within` itself, and one for your statement. The operation of `within` itself is not usually very interesting, so the debugger does not pause except at the end of it, unless there is an error. Errors can occur during the updating of your object, which happens after your expression has been evaluated, e.g. from

```
within( data.frame(), bad <- quote( symbols.not.allowed))
```

Scripts and examples

For debugging "scripts" (i.e. sequences of "live" R commands, either in a textfile or in an R character vector), see `mrun` and `msource`. You might ask why this is useful, since of course you *could* paste each statement into a console one-by-one and do it live. But, with `mrun/msource` you can set breakpoints in advance, and errors inside loops or other compound statements will stop at the actual statement not break back to the console, and so on. Note that `mrun` is useful in its own right without debugging, for people who (like me) hate having 100s of different files lying around, and prefer to keep everything as R objects in a single ".rda" file per workspace.

You can run the examples from a helpfile via eg `mdrunex("topic","package")`, instead of `utils::example`. This gives you much better debug/flow-control facilities. (On the downside, you can't see any comments in the example.)

Vignettes and non interactive use

[v1.4: I have temporarily *disabled* this facility by default, because the current consolette doesn't work with R4.4 and maybe earlier. Thus, the consolette no longer starts automatically at `library(debug)` in a non-interactive R session. To turn it on again, set the system envar "debug_noninteractive_consolette" to any non-empty value before `library(debug)` happens. Eventually I will try to fix the consolette. You *might* still be able to use the consolete within a live R session via `use_consolette` (qv), but I'm not sure there's any point.]

You can debug code in vignettes, too— i.e. stuff that is executed in a non-interactive R session. The ideas can probably be made to work for general non-interactive use (and at one point, they did), not just for vignettes— but I've mostly tested it just with `knitr` and under Lyx. Interaction is via a little "consolette", because there's no *interactive* R session available. True, the consolette interface does have all the grace of a flying tortoise. But, it's a debugger— just be grateful it exists.

To debug a vignette, include this in the first chunk of code:

```
debug::debug_knitr(mtrace=<arg>)
```

where `<arg>` sets the default debugging level for each subsequent chunk (TRUE, NA, or FALSE, explained below). Don't use `library` to load debug— `debug::debug_knitr()` does that automatically, and sets a few other options that need to happen straightaway. In that first setup chunk, you can also set display options such as `options(debug.post.window.launch.hook=tcltk.window.shaker)`.

The debug level for each chunk is set either according to the default used in `debug_knitr` call, or via the chunk option `mtrace=TRUE/NA/FALSE`. TRUE means that the consolette will appear as soon as line 1 of the code is reached; NA means the debugger will be running, but won't pause for input unless it hits an error (akin to setting `bp(1,F)` when debugging a function normally); FALSE means no debugging for that chunk.

Debugging should work either on a directly-included code chunk, or for a chunk whose code is stored in a separate file which is read on-demand using the `code=readLines(...)` chunk hook. The latter relies on an ingenious hack which only works if `readLines` is used; if the external file (or whatever) is read another way, then it won't be debugged (something could probably be figured out for special cases, but is it really worth it?). Note that `knitr` offers some quite exotic possibilities for where a chunk gets its code, which debug won't understand.

Printing/display is a bit stuffed at present; quite what should go to the consolette, and what should go to the knitr output, and so on, I haven't figured out. It's quite possible that final output won't look the same if you have interacted with the debugger— but if you set `mtrace=FALSE` throughout, all should be well. Also, there's no graphics yet.

Inline statements using `\Sexpr` aren't handled (but will be executed by `knitr` as normal).

Consolette: The consolette has command-recall via the down-arrow. You can control some options such as `debug.width` and `debug.screen.pos`, plus `debug.consolette.maxlines` (that get stored of output). Overall appearance should be self-explanatory, but just in case: the debugging codeframes appear as tabs above an output window (the equivalent of the usual "R console"), and at the bottom of the output window there's an input line you type your commands after the "D(xxx)>" prompt. The consolette is created by the call to `debug_knitr` and exists for the duration of the `knitr` run; however, it will only pop to the top of the "display stack" if it's time for user input. Unfortunately, even though it pops up fine in Windows, I don't know how to give it focus; this is something to do with separate processes and focus-grabbing being bad and so on. Might be fixable; dunno how. No idea in Linux; ditto.

Browser-like functionality

If you don't want to fully `mtrace` a function (eg because it runs too slowly) you can insert a call to `debug_BROWSE` (qv) somewhere in the code, and control whether it gets triggered via `mtrace(debug_BROWSE,<TRUE/FALSE>)`. It's like `browser` but keeps things in the **debug** package idiom.

Options

As of version 1.2.0, output is sent *by default* to `stderr()`; this means that you get to see the step-mode output even if the debuggee is redirecting "real" output (e.g. via `sink`). If you don't like this (and I believe Tinn-R doesn't), set `options( debug.catfile="stdout")`.

Command recall is ON by default, but this means that anything typed while debugging will also be seen in `history()` after leaving the debugger. If this is a problem, set `options( debug.command.recall=FALSE)`.

There are two adjustable limits on what gets printed out in step mode (otherwise, your screen will often fill with junk, and displaying results may take several minutes). First, printing will be aborted if it takes longer than `getOption( "debug.print.time.limit")` seconds, which by default is 0.5 seconds. You might need to increase that, e.g. for displaying big help files in the browser. Also, setting a finite time limit cutoff overrides any other time limits that have been set with `setTimeLimit`; this can be prevented by setting `options( debug.print.time.limit=Inf)`. Second, by default only objects with `object.size` < 8192 bytes will be printed in full; for larger objects, a summary is given instead. You can force printing of any individual object via `print`, but you can also increase (or decrease) the threshold to `X` bytes, by setting `options( threshold.debug.autoprint.size=X)`. The `object.size` test isn't foolproof, as some minuscule objects occupy hectares of screen real estate and take ages to print, whereas some big objects print compactly and fast. In my own work, I set the "threshold.debug.autoprint.size" option to `Inf` and the time limit usually to 0.5 seconds.

Various TCL/TK-related aspects of the code window can be altered:

- `debug.screen.pos` defaults to "+5-5" for BL corner; try "-5-5" for BR, "-5+5" for TR, "+5+5" for TL.
- `debug.fg` is foreground text colour, defaulting to "Black"
- `debug.execline` is background colour of next-line-to-execute, defaulting to "LightGreen"
- `debug.font` defaults to "Courier"; try e.g. ="Courier 24 italic"
- `debug.height` (in lines) defaults to 10
- `debug.width` (in characters) defaults to 120. This is the *window* size; *contents* size is set by...
- `debug.max.line.length` defaults to 256; longer lines are truncated (ie you can't pan to them); not TCL/TK
- `tab.width` defaults to 4, for indenting code lines; not TCL/TK

If option `debug.wordstar.keys` is TRUE, various somewhat Wordstar-like key-bindings are provided: CTRL-E and CTRL-X to move blue selection up and down, CTRL-D and CTRL-S to scroll right/left, CTRL-W and CTRL-Z to scroll up/down, CTRL-R and CTRL-C to scroll up/down one page, and CTRL-K C to copy the current blue line to the clipboard (since CTRL-C has acquired a different meaning). Now that I've figured out how to add key bindings (sort of), more might appear in future.

Hooks: There are three hooks associated with creation/destruction of the debug windows. The first, `debug.first.window.hook`, is triggered whenever a new debug window is created (even if other debug windows are already open and the new window is "nested"). The second, `debug.last.window.hook`, is triggered only when the *last* debug window closes. To set them, define them as zero-argument functions via `options( debug.first.window.hook=function() {...})` etc. For example, I use the hooks to automatically resize the RGui console (on Windows). On the stupidly wide-screen displays that modern laptops seem to be cursed with, I want RGui full height most of the time except when debug windows are open. To automatically shrink the main screen when the first debug window opens, and restore it when the last one closes, I use this in my `.First`:

```
options( debug.first.window.hook= function() do.in.envir( envir=debug:::find.debug.HQ( FALSE), {
     if( !any( .frames.$has.window.yet)) { # ie only for 1st one
       set.window.pos( r.window.handle, state=SW_SHOWNORMAL,
           activate=T, pos=c( 85, 0, 1920, 700))
       set.window.state( r.window.handle, SW_SHOWMAXIMIZED)
    set.window.state( r.window.handle, SW_RESTORE) # this seems to work, not sure why
       set.window.state( 'R Console', SW_SHOWMAXIMIZED)
     }
  }),
  debug.last.window.hook= function() {
     set.window.pos( r.window.handle, state=SW_SHOWNORMAL,
         activate=T, pos=c( 85, 0, 1920, 1078))
     set.window.state( r.window.handle, SW_SHOWMAXIMIZED)
   set.window.state( r.window.handle, SW_RESTORE) # this seems to work, not sure why
     set.window.state( 'R Console', SW_SHOWMAXIMIZED)
  }
 )
```

The incomprehensible magic incantations at the start of that `debug.first.window.hook` example are so the function can work out what's going on with the debug windows, and thus shrink the console when the first one is opened; I shan't attempt to explain. If you just want the hook to execute for *every* new debug window (which would in fact be OK for this example), you don't need them— it could look more like the code for `debug.last.window.hook` example. As you'll also see, these examples uses special code for window resizing, which is not part of debug but lives in another package **winjiggle**, written by me and not on CRAN. There may be other ways to do such resizing (eg by writing a partial "RConsole" file and then loading via `loadRconsole`; you only need to write a few plain-text lines).

The third hook is for workarounds to TCL/TK bugs— see next section. Hopefully you won't ever need it.

Display bugs

Over the years, there have been sporadic and unreproduceable display problems with the TCL/TK window, under both Linux and Windows. Sometimes the window frame will appear, but with garbled contents or no contents at all. With RTERM in MS-Windows, a couple of ALT-TABs and mouse clicks to change focus are sometimes necessary. In extremis, the window will eventually sort itself out if you manually minimize, maximize, and restore it– admittedly an irritation. The problem is connected with the timing of signals between R and tcltk, and it seems to disappear and reappear sporadically with new R versions.

As a workaround, you can try setting the third hook, `debug.post.window.launch.hook`, which should be a two-parameter function (see next para). The hook function will be called just after the creation of each new TCL/TK window. This gives you the user a chance to work round TCL/TK bugs that lead to windows coming up empty (just a frame), etc. If set, it should be a function that expects two parameters (or one dot-dot-dot parameter). The first parameter will be a window title (character), which can potentially be used to find the window in the "desktop manager" and fiddle with it, as above. The second parameter is the tcltk window itself.

Currently (as of R3.6.2) I am needing to use this hook myself, and I set it as follows:

```
options( debug.post.window.launch.hook=debug::tcltk_window_shaker)
```

See `?tcltk_window_shaker` for basically this documentation. I **think** it might work on all platforms, but I only use Windows so who knows?

Display bugs part 2: In case you have display problems that `tcltk_window_shaker` doesn't fix, and you're on Windows, this may help. In the past, I used to have a window-shaking routine which in effect did this:

```
options( debug.post.window.lauch.hook = function( window_name, tclwin) {
    win.num <- windows.running( window_name)
    for( ij in c( SW_MAXIMIZED, SW_MINIMIZED, SW_RESTORE))
      set.window.state( win.num, ij)
    set.focus.win( r.window.handle)
  }
)
```

where the various functions are defined in that `winjiggle` package I mentioned, and do more-or-less what they say. You may want to append something like `grDevices::bringToTop(-1)`. Since `tcltk_window_shaker` has been working OK for me for a while, I haven't tested this recently.

Apparently the **RGtk2** package doesn't play nicely with debug– the debugging window displays a blank. I haven't checked this out, but would be grateful for advice.

Emacs

Update: in 2019, debug apparently works fine with ESS (sample size of 1)...

For ESS users: I'm not an Emacs user and so haven't tried ESS with the **debug** package myself. However, a read-through of the ESS documentation suggests that at least one ESS variable may need changing to get the two working optimally, as shown below. Please check the ESS documentation for further details on these points (and see also `?mvbutils`). I will update this helpfile when I have more info on what works.

- The prompt will change when the debugger is running, so you may want to change "inferior-ess-prompt". Prompts will always be of the form D(XXX)> where XXX is a positive integer.
- Command recall probably won't work inside the debugger (or if it does, it's thanks to ESS rather than to R's own command recall mechanism). It should be disabled by default; if you do get error messages about "history not available", make sure to set `options( debug.command.recall=FALSE)` before debugging.

Author(s)

Mark Bravington

See Also

mtrace, go, skip, qqq, bp, get.retval, mrun, debug_BROWSE, step.into.sysfuns, last.try.error

---

bp *Breakpoints for debugging*

Description

Sets/clears breakpoints (including conditional) breakpoints in functions that have been mtraced for debugging.

Usage

```
bp( line.no, expr=TRUE, fname) # fname rarely needed
```

Arguments

line.no	line number
expr	unquoted expression to be tested when execution reaches line.no
fname	name of function to twiddle breakpoints in

Details

Breakpoints can only be set after mtraceing a function, and are normally set while the function is actually being debugged. The simplest way is to look at the code window to identify which lines to stop at, call bp(n) for each of those lines, and then call go() to enter go mode. Your function code will then be executed without pausing for input until a breakpoint is triggered (or an error occurs, or the function finishes normally). To clear a breakpoint for line n, type bp(n,FALSE).

All line-numbered statements actually have an associated breakpoint expression. When the debugger reaches a line-numbered statement, it evaluates the corresponding breakpoint expression in the function's frame. If the result is not identical to FALSE, the breakpoint is triggered. By default, all statements have their breakpoint expressions set to FALSE (by mtrace), except for line 1 where the expression is set to TRUE.

After setting a breakpoint for line n, you will see an asterisk (*) in line n of the left-hand column of the code window. The asterisk is shown whenever the unevaluated breakpoint expression is not identical to FALSE.

Conditional breakpoints are just expressions other than TRUE or FALSE. To get the debugger to stop at line 5 whenever a is greater than b, type bp(5, a>b)– don't quote() the breakpoint expression. Any statement, including a braced statement, can be used, and the debugger will only pause if the result is not FALSE. You can therefore use "non-breaking breakpoints" to patch expressions into the code. For instance, if you realize that you should have inserted the statement a <- a+1 just before

line 7 of your code, you can type `bp( 7, { a <- a+1; FALSE})`; when the debugger reaches line 7, it will increment `a` but will not switch to step mode, because the overall result was `FALSE`.

Sometimes it is useful to clear the line 1 breakpoint before invoking a function, especially if the function is being called repeatedly. The debugger actually starts in go mode, and does not display a code window or pause for input until a breakpoint is triggered; so if the line 1 breakpoint is cleared, execution can continue at full speed until an error occurs (or another breakpoint is triggered). To adjust breakpoints before a function is invoked, you will need to use the `fname` argument. To set/clear breakpoints in function `f` at lines other than 1, first type `tracees$f$line.list` to see which line numbers correspond to which statements.

Breakpoints in body code apply "globally" to all incarnations of a function, and will be retained when the debugger finishes and the R prompt returns. Breakpoint expressions for `f` will be saved in `tracees$f$breakpoints`.

Breakpoints can be set in `on.exit` code as well (but are specific to the incarnation they are set in). It is often useful to set a breakpoint at the first exit code statement (which will be `NULL` if `on.exit` has not yet been called); this has the effect of a "run-until-finished-then-pause" breakpoint. Whenever `on.exit` is called, any existing exit code breakpoints are lost; but if any were present, a new unconditional breakpoint is set at the start of the exit code.

Breakpoints are evaluated in step mode too, but the debugger remains in step mode whatever the result.

At present, all breakpoints are destroyed when functions are edited; if you use `fixr`, `mtrace` will be re-applied automatically, but breakpoints will be lost. However, my S+ versions of debug and `mvbutils` used to make an effort to preserve breakpoints across edits, and I plan to introduce something similar in R. (The documentation has said "I plan" for about 15 years now... so I guess I don't really plan to, after all.)

Author(s)

Mark Bravington

See Also

`mtrace`, `go`

Examples

```
## Not run:
mtrace( glm)
glm( 35)
# Once the debugger starts:
bp(7) # unconditional breakpoint at line 7
bp(7,F) # to clear it.
bp(7,x>1) # conditional; will trigger if "x>1" (or if "x>1" causes error)
bp(1,F,"glm") # can be called BEFORE debugging glm (but after mtrace);
# ... prevents debugger from halting at start of function
qqq() # exit debugger
mtrace.off()

## End(Not run)
```

debug.C *For debugging C-calls with many arguments.*

Description

Sometimes you call `.C` with huge numbers of arguments (e.g. when initializing data structures), and you get an error message complaining about one of the arguments. Trouble is, the error message doesn't tell you which one, and it can be hard to track down. A convenient way to find out is to `mtrace` the caller and run as far as the `.C` call, then do this at the prompt:

```
D(n)> .C <- debug.C
```

and then hit <ENTER>. All *named* arguments will be evaluated and summary information will be printed. So you do need to make sure your "important" `.C` arguments all have names– which is good practice anyway, for matching up the R and C code. Return value is `NULL`; `debug.C` doesn't actually try to run any C code. (You wouldn't be using this if your `.C` worked!)

Usage

```
# This is mandatory, but not useful here. See *Description*
# You would never call 'debug.C' directly
debug.C(...)
```

Arguments

`...`	a la `.C`

debug.eval *Evaluate expression in debugging window*

Description

You shouldn't need to call this directly— calls to `eval()` while debugging (either in code, or from the debug console) should divert automatically to this function. This documentation pre-dates the automated version, and you probably don't need to read it. But FWIW, the old documentation continues from here...

Like `eval` but summons up a debug window, as when debugging a function call, so that you can step thru `expr`. For use at the command-line while debugging where, if you just typed `eval(expr)` it would not step into `expr`. Step-in only happens if `expr` has multiple elements.

```
debug.eval( quote({ a <- 1})) #  won't step
debug.eval( quote({ a <- 1; b <- 2})) #  will step
```

If you want to step thru a sequence of text commands (e.g. a script), see msource and mrun.

As explained under "SPECIAL FUNCTIONS" in package?debug, there are analogous "steppy" functions for evalq (i.e. debug.evalq), with, within, local, suppressWarnings, and try. However, these are currently not exported, so you'd need e.g. debug:::debug.with(dataset, commands). I might export them eventually, but I suspect they're less likely to be used during "live" debugging.

Usage

```
debug.eval( expr, envir = parent.frame(),
   enclos = if (is.list(envir) || is.pairlist(envir)) parent.frame() else baseenv())
```

Arguments

expr, envir, enclos
: see eval

See Also

fixr, package?debug under SPECIAL FUNCTIONS, msource, mrun

---

debug_BROWSE *Debugging support*

Description

Suppose you are trying to debug a function, but you don't want to mtrace it (eg because the mtraced version is too slow). If you know roughly where the problem manifests itself (eg via traceback() after an error), you can insert a call to debug_BROWSE()— which is like *provisionally* inserting browser(), but with the functionality of the **debug** package. The call will do nothing until you "awaken" the sleeper with mtrace(debug_BROWSE). If triggered subsequently, it will pop up a debug window where you can inspect/change the variables in your main function; as usual in package **debug**, any expression you type is evaluated in the running environment of your main function. However, you can't actually step back into the main function, of course, because you haven't mtraced it.

While I prefer mtrace() 99% of the time, occasionally I'll use debug_BROWSE() if I have error-trap code that's getting triggered when I don't expect it. See **Examples**.

Call debug_BROWSE() has no effect unless mtrace(debug_BROWSE) has been set, so you can safely leave it in. If your function is inside a package, you might need debug::debug_BROWSE(); remember to comment it out before CRANtorting your package.

Usage

```
# Never ever call it like this... the next line exists only because of Rd rules
debug_BROWSE( nlocal = sys.parent())
# just do this...
# debug_BROWSE()
```

Arguments

nlocal	See mlocal if you must know. I wouldn't bother, though.

Examples

```
## Not run:
surprise <- function( weeble) {
  result <- sqrt( weeble)
  if( !all( is.finite( result))) {
    browse_DEBUG() # now you can look at weeble
stop( "WTF?")
  }
return( result)
}
mtrace( debug_BROWSE)
surprise( 1:10 - 3)

## End(Not run)
```

---

debug_knitr *Debugging knitr documents*

Description

NOT WORKING in R4.4, and should be disabled by default in package **debug** >= v1.4. To be fixed...

To debug a knitr document (during compilation, i.e. while it's on its way to become PDF or HTML), put one line into the first chunk:

```
debug::debug_knitr( mtrace=TRUE)
```

Then, every time it gets to a code chunk, a "consolette" should pop up to allow you to interact with the evaluation, just as if you had mtraced a function (or, more accurately, as if you were mrunning the code in the chunk with argument debug=TRUE).

What that line actually does, is automatically load the package *and* set some options needed for it to work easily. It also turns on mtrace for all chunks (because of the TRUE), but you can override that per-chunk with eg mtrace=FALSE in the chunk options. mtrace=NA is also useful; see below.

Default use of debug_knitr does change a few knitr options and hooks, explained below. If you don't want that, you can set some parameters as explained below, but then you're in charge of making the debugging actually happen. You might also need to set cache=FALSE to make the consolette appear.

When everything is working (or at least running), I just disable that line by putting # at the start. Setting mtrace=FALSE will stop the consolette appearing, but you may find that the output isn't right (e.g. currently it's only showing results not commands— it's on the To-Do list, but it's got a lot of company).

Before the above line, you can also set some options to control the placement of the consolette (the window that lets you interact with the debugging process). Obvious ones are debug.height and debug.screen.pos. They are safe to leave, even if you disable the debug_knitr line. There is a little more info about the consolette in package?debug.

Usage

```
debug_knitr(auto = TRUE, readLines_hack = TRUE, mtrace = TRUE)
```

Arguments

auto	?globally modify the code chunk-option so that all "normal" chunks will be mtraced, or not according to the level of opts_chunk$get("mtrace").
readLines_hack	?globally modify readLines so that it will auto-debug a chunk read from external file, with code=readLines("<myfile>")? Operation should be unchanged otherwise.
mtrace	default level of mtrace option. TRUE, FALSE, or NA which is like TRUE except that the debugger won't stop unless there's an error (because the default breakpoint on line 1 gets turned off).

See Also

package?debug, non-existent vignette

Examples

```
## Not run:
# Put this in the first chunk of your knitr doco for an au
debug::debug_knitr()
# If you want "only-on-crash" debugging:
debug::debug_knitr( mtrace=NA)
# or chunk-by-chunk, EG :
# <<mychunk, mtrace=FALSE, ...>>= # run this chunk normally, ie without mtrace

## End(Not run)
```

---

duhook *Call user hook for debugging window*

Description

Package **debug** lets you set a "user hook" to be called when a new TCL/TK debugging window gets opened. This can be useful e.g. to do some manual wiggling to get TCL/TK to synch properly. Sometimes you might want to call your own user hook deliberately, e.g. if the hook hasn't managed to synch automatically. duhook lets you do that. See DISPLAY stuff in package?debug.

Usage

```
duhook(D = 0)
```

Arguments

D	Frame number (displayed in the window's title) to call the hook on. By default, it's the most-recently displayed window.

See Also

package?debug, tcltk_window_shaker

---

fun.locator *Get environment(s) where an object exists.*

Description

Checks the frame stack, the search list, any namespaces, and any S3 method tables for copies of an object (normally a function). Used by mtrace. If the search starts from a reference-class object, only the object itself is searched, and legit methods are forced into existence.

Usage

```
fun.locator( fname, from=.GlobalEnv, mode='function')
```

Arguments

fname	character string (the object's name)
from	Where to start looking:either an environment, another function (in which case, that function's defining environment is used), or a frame number. 0 is treated as .GlobalEnv. from *can* also be a list object, which will be returned if it contains fname; however, the result may not be of much use (e.g. mtrace can't use it directly).
mode	What type of object to look for– by default, a function. Set to any if the object's mode is not relevant.

Details

When fname is defined in a namespaced package **foo**, several copies will exist at once. If foo exports fname, then fname will be found both in the search path (in package:foo) and in the "hidden" environment asNamespace("foo"). The first version will be found by "normal" calls to fname, but the hidden version will be found whenever fname is invoked by a function in package:foo. If the S3 method somegeneric.fname is defined in package:foo, then it will exist both in asNamespace("foo") and in get(".__S3MethodsTable__.", asNamespace("foo")), although not in the main search path. If fname is both exported and an S3 method, then there will be three copies. Other packages that import from package **foo** may also have fname in the parent environments of their namespaces. All such duplicates should probably be identical, and if you want to change any one copy, you probably want to change all of them.

Normally, the search path environment(s) where fname is found will be given before (i.e. with lower index than) the hidden namespace environment. However, if from is used to start the search in a

hidden namespace environment where `fname` exists, then the hidden namespace will be returned first. Duplicated environments are removed from the return list.

For reference-class and S4 methods, you need to set `from` explicitly; see `?mtrace` and `package?debug` respectively.

For functions within `lists` (as opposed to existing directly within an `environment`), `mtrace` needs you to mess about manually; see its `FINDING FUNCTIONS` subsection.

Value

A list of environments where `fname` was found, of length 0 if no copies were found. First, any environments between `from` and `topenv( from)` are searched (normally, this means temporary frames wherein `fname` could have been defined; but see **Details**). Next, the search list is checked. Next, all loaded namespaces are checked. Finally, all S3 method tables are checked.

Author(s)

Mark Bravington

See Also

`getAnywhere`

---

get.retval *Show current return value when debugging*

Description

When debugging a function, `get.retval` gives the current return value, i.e. the result of the most recent valid statement executed in the function body, whether typed or in the original code, and excluding calls to `go` etc.. Presumably only relevant when in exit code.

Usage

```
get.retval()
```

Details

To **change** the return value while in exit code, use `skip` to move back into the function code, then `return(whatever)`.

Author(s)

Mark Bravington

See Also

`skip`, `go`, `last.try.error`

`go-skip-qqq` *Flow control for debugger*

Description

`go`, `skip` and `qqq` ONLY work inside the debugger, i.e. while you are paused at a D(...)> prompt during the execution of a function that has been `mtrace`d. `go` makes the debugger begin executing code continuously, without user intervention; `skip(n)` shifts the execution point; `qqq()` quits the debugger.

Usage

```
go(line.no) # line.no can be missing
skip(line.no)
qqq()
```

Arguments

`line.no`	a line number, as shown in the code window (see also **Details**)

Details

`go()` without any argument puts the debugger into "go mode", whereby it starts executing function code without pausing for input (see `package?debug`). `go(n)` basically means "run continuously until you reach line `n`". It sets a temporary breakpoint at line `n`, which is triggered the first time line `n` is reached and then immediately cleared.

`skip(n)` moves the execution point (highlighted in the code window) to line `n`, without executing any intervening statements. You can `skip` forwards and backwards, and between the main function code and the exit code. You can skip in and out of loops and conditionals, except that you can't skip into a `for` loop (the execution point will move to the start of the loop instead). Note that skipping backwards does not undo any statements already executed. `skip` is useful for circumventing errors, and for ensuring that exit code gets run before calling `qqq()`; for that, it's easier to just type `return` which will "skip" directly to the start of the exit code if any, or otherwise to a NULL dummy exit statement.

`qqq()` quits the debugger, closing all code windows, and returns to the command prompt. No further code statements will be executed, which means no exit code either; take care with open files and connections.

Author(s)

Mark Bravington

See Also

package **debug**, `mtrace`, `bp`

is.mtraced *Check if a function has been 'mtrace'd.*

Description

Check if a function has been mtraced.

Usage

```
is.mtraced(f)
```

Arguments

f	name of a function (quoted or unquoted)

Value

TRUE or FALSE.

See Also

mtrace

Examples

```
## Not run:
fff <- function() 99
is.mtraced( fff)
mtrace( fff)
is.mtraced( fff)

## End(Not run)
```

last.try.error *Get last try-error*

Description

Suppose you are debugging, and have stepped-into a `try` statement, and the debugger has stopped at an error. You might actually want to return that error as an object of class `"try-error"`, just like `try` would if you weren't debugging. To do that, type `return( last.try.error())` at the debug prompt.

Usage

```
# Normally you would type return( last.try.error()) at the debug prompt
last.try.error()
```

Value

An object of class `try-error`, just like `try` itself returns in case of error.

`mrun` *Run script with optional debugging*

Description

Run an "R script" either from a file (`msource`) or directly from a character vector (`mrun`, `mdrun`, `mpdrun`) or from a package example (`mdrunex`). The main point is to allow debugging with `mtrace`, controlled via the `debug` argument. This defaults to TRUE for `msource` (where you could just use `source` instead if you didn't want to debug) and `mdrun` and `mpdrun` and `mdrunex`, but to FALSE for `mrun` (which is useful in its own right without debugging, in which case it amounts just to `eval( parse( text=...))`). Evaluation by default takes place in `.GlobalEnv`, so objects created are permanent.

I use `mrun` because, although scripts (as opposed to "pure functions") can be quite useful, I don't like separate script files— too many Things to keep track of. Also, debugging "scripts" is normally painful compared to debugging functions, for which `mtrace` makes life easy. My "scripts" are created with `fixtext` as opposed to `fixr`, and are character vectors of S3 class `cat`, which means they display nicely; I give them names ending ".r", so that they are obvious. `mdrun` just saves me having to type `mrun( <greatbiglongname>, debug=TRUE)` which I was finding myself doing a lot of. `mpdrun` is convenient when a scriptlet doesn't parse completely (which might be deliberate); it lets you debug the first parsable portion, and saves me from typing `mdrun(<greatbiglongname>,partial_parse_OK=TRUE)`.

For package examples, you can use `utils::example` (qv), but I often want to be able to pause inside that code, etc. So `mdrunex` first uses `utils::example` to extract the example code, then `mdrun` to actually run it in the debugger (but still putting results into `.GlobalEnv`, FBOW). The arguments `run.dontrun` and `run.donttest` might be useful.

The debugging trick behind all this is to make the "script" into the body of an `mlocal` function— which is thereby debuggable— then forcing it to execute directly in the `local` environment. The "frame number" when debugging— ie "xxx" inside the "D(xxx)>" prompt— will be 0 for the default `local=.GlobalEnv`, or some unpredictable number for a bespoke environment.

Usage

```
mrun( script, local=FALSE, debug=FALSE, echo=TRUE, print.eval=FALSE, partial_parse_OK=FALSE, ...)
mdrun( script, ...) # passed to mrun() with debug=TRUE
mdrunex( topic, package, ...)
msource( file, local=FALSE, debug=TRUE, ...)
```

Arguments

`script`	Character vector of R commands
`local`	FALSE for `.GlobalEnv`, TRUE for frame where `msource` / `mrun` is called, or an environment (or something that can be coerced to an environment)
`debug`	set to TRUE if you want to use the **debug** package

topic, package — strings (ie must be quoted)

file — filename of R "script"

echo, print.eval, ... — as per source, to which they are passed iff debug=FALSE. For mdrun, elements of ... that match named args to mrun will be handled by mrun (so the sentence before this one is false in strict legalistic terms, but strict legalistic people don't deserve any better).

partial_parse_OK — if TRUE, don't just fail if the scriptlet fails to parse completely; rather, just start debugging on the bits at the front that do parse.

Value

Whatever the last expression gave, invisibly.

Examples

```
## Not run:
# To use 'mdrunex', add argument 'run.dontrun=TRUE'
scriptio.r <- c( 'a <- 99', 'b <- a+1')
mrun( scriptio.r)
a # 99; created in .GlobalEnv
b # 100
leftovers <- 555555
mdrunex( 'mdrunex', 'debug') # NULL code, coz it's in a don't-run block

## End(Not run)
```

---

mtrace *Interactive debugging*

Description

mtrace sets or clears debugging mode for a function; mtrace.off clears debugging mode for all functions; check.for.tracees shows which functions are in debugging mode.

Usage

```
# Usual: mtrace( fname) or mtrace( fname, F) or mtrace( apackage:::afunction)
mtrace( fname, tracing=TRUE, char.fname,
    from=mvb.sys.parent(), update.tracees=TRUE, return.envs=FALSE)
mtrace.off()
check.for.tracees( where=1)
```

Arguments

fname	quoted or unquoted function name, or unquoted reference to function in package (via :: or :::) or list/environment (via $)
tracing	TRUE to turn tracing on, FALSE to turn it off
char.fname	(rarely used) if your function name is stored in a character object x, use char.fname=x. If you want to *turn off* tracing while doing so, mtrace(char=x, F) won't work because of argument matching rules; you need mtrace(char.fname=x, tracing=F).
from	where to start looking for fname (not usually needed)
where	(character or integer) position in search path
update.tracees	don't set this parameter! It's only for use by other functions
return.envs	if TRUE, this will return a list of the environments where the function has been replaced by the mtraced version

Details

mtrace(myfun) modifies the body code of myfun, and also stores debugging information about myfun in tracees$myfun. Next time the function is invoked, the modified debug-ready version will be called instead of the orginal. mtrace does not modify source code (or other) attributes, so myfun will "look" exactly the same afterwards. mtrace(myfun,FALSE) restores myfun to normal. mtrace.off unmtraces all mtraced functions (see below for exceptions).

Because mtrace modifies function bodies (possibly in several places, if namespaced packages are involved), calling save.image or save while functions are still mtraced is probably not a good idea– if the saved image is reloaded in a new R session, the debugger won't know how to handle the previously mtraced functions, and an error message will be given if they are invoked. The Save and Save.pos functions in package **mvbutils** will get round this without your having to manually untrace and retrace functions.

If you do see a "maybe saved before being un-mtraced?" error message when myfun is invoked, all is not lost; you can restore myfun to health via mtrace(myfun,F), or put it properly into debugging mode via mtrace(myfun). mtrace.off won't work in such cases, because myfun isn't included in tracees.

check.for.tracees checks for functions which have been mtraced, but only in one directory. By contrast, names(tracees) will return all functions that are currently known to be mtraced. However, unlike check.for.tracees, names(tracees) won't show functions that were saved during a previous R session in an mtraced state.

mtrace.off will try to untrace all functions. Specifically, it deals with those returned by names(tracees) and/or check.for.tracees(1). It doesn't currently deal with methods of reference-class and S4-class objects, for which you'll need to call mtrace(..., tracing=FALSE) manually.

mtrace puts a breakpoint (see bp) at line 1, but clears all other breakpoints.

mtrace can handle mlocal functions, but not (yet) do.in.envir functions– the latter appear as monolithic statements in the code window. See package **mvbutils** for more details.

If you use fixr to edit functions, mtrace will automatically be re-applied when an updated function file is sourced back in. Otherwise, you'll need to call mtrace manually after updating a function.

Finding functions: `mtrace` by default looks for a function in the following *environments*: first in the frame stack, then in the search path, then in all namespaces, then in all S3 methods tables. If several copies of a function are found, all will get modified (mtraced) to the **same** code; ditto when unmtracing.

For functions that live somewhere unusual, you'll need to set the `from` argument. One case is for functions that live inside a list, such as family-functions like `poisson` for GLMs. In that case, because the list is not an environment, you will have to do something like :

```
funtotrace <- mylist$funtotrace
mtrace( funtotrace)
mylist$funtotrace <<- copyfun
```

and a very similar sequence to un-mtrace later. I have thought about ways to automate this, but (i) they are hard to code, and (ii) they might go wrong— so, it's Do-It-Yourself, please.

Another non-obvious case is as follows. Suppose there is a function `f` which first defines functions `g` and `h`, then calls `g`. Now suppose you have `mtraced` `f` and then `g` from inside `f`, and that `g` is currently running. If you now want to `mtrace(h)`, the problem is that `h` is not visible from the frame of `g`. To tell `mtrace` where to find `g`, call `mtrace( h, from=sys.parent())`. [You can also replace `sys.parent()` with the absolute frame number of `f`, if `f` has been `mtraced` and its code window is visible.] `mtrace` will then look through the enclosing environments of `from` until it finds a definition of `h`.

If `myfun` has been defined in a namespaced package, then there may be several copies of `myfun` in the system, different ones being used at different times. `mtrace` will change them all; see `fun.locator` if you really want to know more.

If `mtrace(bar)` is called while function `foo` is being debugged (`mtrace(foo)` having previously been called), and `bar` has been redefined within `foo` or a parent environment of `foo`, then only the redefined copy of `bar` will be `mtraced`.

S4 and reference class methods: S4 methods can be `mtraced`, but like much about S4 it's clunky; see `package?debug`. Reference class methods can be `mtraced` easily after an object has been instantiated. You might call this "object-level" mtracing, because it only works for one object of each class at a time. To `mtrace` e.g. the `edit` method in the example for "?ReferenceClasses", just do:

```
mtrace( edit, from=xx) # NB will force a method into existence even if it's not been invoked yet
 mtrace( edit, from=xx, FALSE) # to clear it; mtrace.off() won't work properly
```

You can also do "class-level" mtracing, so that all subsequently-created objects of that class will use the `mtraced` version. Just do this:

```
 mtrace( edit, from=mEditor$def@refMethods)
 xx <- mEditor$new( ...)
mtrace( edit, from=mEditor$def@refMethods, FALSE) # to clear it; mtrace.off() won't work properly
```

In the "class-level" case, `xx` will still have an `mtraced` version of `edit` even after the `mtrace( from=mEditor..., FALSE)`. You'll need to use the "object-level" technique to clear it.

As of April 2011, methods are only set up inside a ref-class object when they are first *accessed*, not when the object is created. `mtrace` (actually `fun.locator`) works round this.

Limitations: Here's a few; I doubt this is all of them!

A few basic functions (used by the **debug** package itself) can't be `mtrace`d directly— things go horribly wrong, in a usually obvious way. You have to make a copy with a different name, and `mtrace` that instead. I'm not going to try to list them all here!

You can't have `mtrace` on simultaneously for two functions that have the same name but that have different bodies and live in different places. In theory, the solution is for me to incorporate "location" into the function-level debug info in `mtracees`, but I've not been able to figure out a good general-purpose way to do so. If this describes your particular debugging hell, you certainly have my sympathy...

Functions inside `list` objects need to be manually `mtrace`d and un-'mtrace'd; see **Finding functions** above.

Value

`mtrace` by default returns an invisible copy of the modified function body. If you set `return.envs=TRUE`, it will instead return a list of the environments in which the function has been modified. This is only intended for "internal use". `check.for.tracees` returns a character vector of function names.

Examples

```
## Not run:
mtrace(glm) # turns tracing on
names( tracees) # "glm"
check.for.tracees( "package:base") # "glm"
glm(stupid.args) # voila le debugger
qqq() # back to command prompt
mtrace( glm, FALSE)
mtrace.off() # turns it off for all functions
mtrace( debug:::setup.debug.admin) # woe betide ye

## End(Not run)
```

---

step.into.sysfuns *Control which "special" functions get stepped into*

Description

When the **debug** package is in step-mode, there are a few special system functions that it can either step into, or leave the system to handle *en bloc*. These functions all have expression-type arguments. Currently, they are: `try`, `suppressWarnings`, `eval`, `evalq`, and (for built-in methods only) `with` and `within`. The step-into behaviour is controlled by calling `step.into.sysfuns`, which operates like `par`. You can also circumvent step-into at particular lines, by using `go(n)` to zoom through to the next statement. Additional methods for `with` and `within` can be handled via e.g. `mtrace( with.myS3class)`, so are not considered "special" here.

Usage

```
# USAGE is not useful here-- see *Arguments*
step.into.sysfuns( ...)
```

Arguments

...	tag-value pairs of logicals, e.g. `with=TRUE, evalq=FALSE`. Legal tags are shown in **Description**. If empty, return all tags-value pairs, as a logical vector.

Value

Either the previous value(s) of tags that are set, or the entire logical vector of tags.

Examples

```
step.into.sysfuns() # all of them-- shows which are legal
step.into.sysfuns()['with'] # extract one of them
owith <- step.into.sysfuns( with=FALSE) # don't step into with-statements
step.into.sysfuns( with=owith) # revert to previous
```

---

tcltk_window_shaker *Display fix for tcltk window in debug package*

Description

In some versions of R on some platforms, there is a display problem with the tcltk window which should show your source code when your `mtrace`d source code is activated: the window comes up blank or incomplete or whatever. The issue has come and gone for me erratically over the years with new R versions; basically, the linke from R `tcltk` is a bit flaky. If it happens to you, try setting `options( debug.post.window.lauch.hook=debug::tcltk_window_shaker)`. The alternative is manual shrinking/maximizing/resizing/noodling around with every new tcltk window that the **debug** package opens... which works eventually, but is very tedious. `tcltk_window_shaker` tries to automate that.

If `tcltk_window_shaker` doesn't work, there are other tricks that might help; see `package?debug`, section "Display bugs", for (slightly) more info and hints on how to write your own.

There might also be a focus problem, whereby focus is left with the tcltk window rather than back in the R console/main input panel. On Windows only, `tcltk_window_shaker` will attempt to call `grDevices::bringToTop` iff package **grDevices** is already loaded (not necessarily attached). Hopefully that doesn't cause problems from Rstudio etc.

Usage

```
# You shouldn't be calling this function yourself;
# ... use it only via options(), as per the DESCRIPTION.
tcltk_window_shaker(tcltk_window_name, tcl.win)
```

Arguments

You probably don't need to know this, but any function that is passed as option `debug.post.window.lauch.hook` should expect two parameters. The first parameter will be a window title (character), which can potentially be used to find the window in the "desktop manager" and fiddle with it, as above. The second parameter is the tcltk window itself, which might be easier to fiddle with— at your own risk.

`tcltk_window_name`, `tcl.win`
: things you shouldn't tinker with.

Examples

```
## Not run:
options( debug.post.window.launch.hook=debug::tcltk_window_shaker)

## End(Not run)
```

`use_consolette` *Consolette*

Description

You should haaaardly ever need to call these; the consolette is meant to be used iff R is running non-interactively, and should always be present (albeit not necessarily visible) if so. However, you *might* prefer the consolette even during interactive use, so "manual" control might be useful (at least for me when I am debugging package **debug**). Thus:

- `use_consolette(TRUE)` will turn the consolette on (and turn off normal debug windows), which is OK if you prefer the consolette anyway in an interactive session)
- `use_consolette(FALSE)` should revert to no

You shouldn't close the consolette while actually debugging, i.e. if there are codeframes visible in it. In fact, `use_consolette(F)` won't let you, unless you override it with `force=TRUE`. Only do that if you're desperate; god knows what state the debugger will be in afterwards.

Usage

```
use_consolette(useit = TRUE, force = FALSE)
```

Arguments

`useit`	obvious
`force`	only set to `TRUE` if you absolutely must close the consolette while it has e.g. an "orphaned" codeframe still visible. Don't use unless desperate.

See Also

`package?debug`, `mrun`

Index