**Preamble: what do we mean by mutable?**

**What is mutable**

Often, when you think you’re modifying an object,

x <- c(1, 7)

x[2] <- 2

a new object with a new value has been created and bound to the initial name. “Once you call that function with an object, you’ll get the object’s current address”.

x <- c(1, 7)

tracemem(x)

## [1] "<0x564b0de79f38>"

x[2] <- 2

## tracemem[0x564b0de79f38 -> 0x564b0dedfc18]: eval eval withVisible withCallingHandlers handle timing\_fn evaluate\_call evaluate in\_dir block\_exec call\_block process\_group.block process\_group withCallingHandlers process\_file eval eval eval eval eval.parent local

We see that the address of “x” changes, a new object has been bound to that name, it’s not the original object with the original address that has been modified.  
Environments are one of the R object types that *are* mutable

**What feels mutable**

In this post, we’re actually interested in any method or tool that makes something *feel* mutable.

We’re used to code such as

# initial value

x <- 1

# change the value

x <- 2

But the code below also changes x without explicitely assigning a value to it!

x <- 1

length(x) <- 2

x

## [1] 1 NA

One can say the code above is a bit odd. This post is a collection of patterns that might feel odd. 

**What’s not mutable and doesn’t feel mutable either**

data.frames are not mutable and one doesn’t feel they are, even with dplyr::mutate(): you don’t write

dplyr::mutate(df, newcol = 1)

to modify df, you need to write

df <- dplyr::mutate(df, newcol = 1)



**A replacement function in the urltools package**

In the urltools package there are a few functions for getting or setting parts of an URL such as the fragment.

url <- "<https://docs.r-hub.io/#package-builder>"

urltools::fragment(url)

## [1] "package-builder"

urltools::fragment(url) <- "intro"

url

## [1] "<https://docs.r-hub.io#intro>"

urltools::fragment(url) <- NULL

url

## [1] "<https://docs.r-hub.io>"

The original url value is not modified. Below we use the [tracemem() function](https://adv-r.hadley.nz/names-values.html#tracemem).

url <- "<https://docs.r-hub.io/#package-builder>"

tracemem(url)

## [1] "<0x564b0e61ac88>"

urltools::fragment(url) <- "intro"

## tracemem[0x564b0e61ac88 -> 0x564b0d5cc928]: eval eval withVisible withCallingHandlers handle timing\_fn evaluate\_call evaluate in\_dir block\_exec call\_block process\_group.block process\_group withCallingHandlers process\_file eval eval eval eval eval.parent local

url

## [1] "<https://docs.r-hub.io#intro>"

So how does the above work, exactly? What’s that fragment method?

getMethod(urltools::"fragment<-")

## Method Definition (Class "derivedDefaultMethod"):

##

## function (x, value)

## {

## if (length(value) == 0 && is.null(value)) {

## return(rm\_component\_(x, 5))

## }

## return(set\_component\_f(x, 5, value, "#"))

## }

##

##

##

## Signatures:

## x

## target "ANY"

## defined "ANY"

Actually, reading the source above doesn’t help us. Sure it creates a new string, but how on Earth is it the new string bound to the initial name? Well, it’s because the function is called fragment<- with an arrow at the end and has a last argument called value, both criteria together make it a **replacement function**.

Let’s create our own replacement function to make sure we got it right!

x <- 1:5

x

## [1] 1 2 3 4 5

# function that will replace all values of x

# with the new value

`replace\_all<-` <- function(x, value) {

x[seq\_along(x)] <- value

x

}

# the argument called value is passed at the right of the arrow

replace\_all(x) <- 42

x

## [1] 42 42 42 42 42

So we’ve modified x, but not in place, see below the same code with tracemem()

x <- 1:5

tracemem(x)

## [1] "<0x564b0b93b3f8>"

`replace\_all<-` <- function(x, value) {

x[seq\_along(x)] <- value

x

}

replace\_all(x) <- 42

## tracemem[0x564b0b93b3f8 -> 0x564b0b7a07b8]: eval eval withVisible withCallingHandlers handle timing\_fn evaluate\_call evaluate in\_dir block\_exec call\_block process\_group.block process\_group withCallingHandlers process\_file eval eval eval eval eval.parent local

## tracemem[0x564b0b7a07b8 -> 0x564b0b7a0808]: replace\_all<- eval eval withVisible withCallingHandlers handle timing\_fn evaluate\_call evaluate in\_dir block\_exec call\_block process\_group.block process\_group withCallingHandlers process\_file eval eval eval eval eval.parent local

## tracemem[0x564b0b7a0808 -> 0x564b0b7ac168]: replace\_all<- eval eval withVisible withCallingHandlers handle timing\_fn evaluate\_call evaluate in\_dir block\_exec call\_block process\_group.block process\_group withCallingHandlers process\_file eval eval eval eval eval.parent local

x

## [1] 42 42 42 42 42

So replacement functions are a standard way to give a mutable flavour to R code. Let’s move on to another mutable feel.

**Exposing the C API in xml2**

With xml2 you can modify and remove XML nodes from a tree which makes you feel the tree is mutable.

xml2::xml\_replace(xml2::xml\_find\_all(xml, "//softbreak"),

xml2::read\_xml("\n"))

That code changes nodes in the xml object without our assigning it back to it.

**Interfacing an external process that’s actually mutable in ps::ps\_handle()**

Now, speaking of objects that are actually mutable, the ps package offers an interesting example: the ps\_handle() function creates an object that’s essentially a pointer to a system process. System processes are of course mutable, they run, then die, can be suspended, etc.

In the example below we launch a process using processx, create a ps\_handle object corresponding to it i.e. just an external pointer with an S3 class, and we query its status using ps. ps can’t *create* processes but it can query, list and manipulate them.

p <- processx::process$new("sleep", "0.5")

With such a definition, after half a second the process will no longer exist.

p$get\_pid()

## [1] 10148

phandle <- p$as\_ps\_handle()

phandle

## PID=10148, NAME=sleep, AT=2020-01-22 08:48:38

ps::ps\_status(phandle)

## [1] "sleeping"

Sys.sleep(0.5)

ps::ps\_status(phandle)

## Error: No such process, pid 10148, ???

This example corresponded to an object in R referring to something mutable *outside* of R. What about an object corresponding to something mutable that can also be *inside* of R and mutable? An answer is: R6 objects!

**Actually mutable objects with R6**

An example of a package using R6 is desc. Let’s create an object corresponding to the DESCRIPTION of the rhub package.

rhub\_desc <- desc::desc(text = readLines("<https://raw.githubusercontent.com/r-hub/rhub/master/DESCRIPTION>"))

rhub\_desc$get\_authors()

## [1] "Gábor Csárdi [aut, cre]"

## [2] "Maëlle Salmon [aut] ()"

## [3] "R Consortium [fnd]"

rhub\_desc$add\_author\_gh("testingjerry")

rhub\_desc$get\_authors()

## [1] "Gábor Csárdi [aut, cre]"

## [2] "Maëlle Salmon [aut] ()"

## [3] "R Consortium [fnd]"

## [4] "Testing Jerry [ctb]"

” Firstly, if you use R6 it’s very easy to create a non-idiomatic API that will feel very odd to native R users, and will have surprising pain points because of the reference semantics.”

Yep, the mutable aspect can feel odd, otherwise we wouldn’t write a whole post about it. (Other aspects of R6 are odd, e.g. the use of $ to call *methods* on *objects*.)

In the case of desc all methods exist both as methods and as functions. The functions operate on the DESCRIPTION of the current folder which is handy when working on a package. E.g. say you’re working on a package inside its folder and want to add a contributor to DESCRIPTION, you can do

desc::desc\_add\_author\_gh("")

And the local DESCRIPTION file will be updated. So what’s become mutable is the DESCRIPTION file itself via an object that’s written to disk each time it’s changed!

To mention another R6 example and to come back to system processes: processx uses R6 and an external pointer, because some of the mutable state is in R. ps uses an external pointer and then all the mutable state is in C or is external.

**Conclusion**

In this post we have shown different reasons and ways to provide a mutable *API*/interface to R users. As a summary, in many cases, when you want a mutable API, setter methods that are in fact **replacement functions** are the way to go, like urltools. If you need to represent an external object, that is mutable itself (e.g system process like processx or database connection, etc.), then external pointers. If you want to avoid copying for performance or other reasons, then R6.