Advanced R for Econometricians (Summer 2022)

Advanced R Concepts — Notes and Examples

Martin Arnold, Jens Klenke

Make sure the `lobstr` package is attached!

```r
install.packages('lobstr')
library(lobstr)
```

Slide 5: Assignment Operators

Solution to the task

- `->` is called ‘right-assignment operator’
- The precedence relation is `->` » `<-` » `=` so `x <- 1 -> b` is another working alternative to the last line on the Slide 4.

Slide 12: Copy-on-modify — Function Calls

Solution to the task

- **`f(x)`**: `f()` binds `a` to the same memory location `x` points to *during execution*. In contrast to the example on the previous slide, a copy *is* made since `f()` modifies `a`.
- **`z <- f(x)`**: as above. `z` is a binding the same location as `a` (no additional copy made).
- **Q**: What would happen if we’d super-assign to `x` *inside* of `f()`?
  A: No additional reference to `x`, so calling `f()` would *not* trigger a copy!

Slide 13: Copy-on-modify — Lists

Solution to the task

```r
l1 <- list(1, 2, 3)     # left
l2 <- l1                # right
l2[[3]] <- 4            # bottom
```

Note that the copy-on-modification (triggered by `l2[[3]] <- 4`) is a so-called *shallow copy*: the bindings are copied, *not* the values. This is non-expensive ⇒ performance considerations!

Slide 15: Copy-on-modify — Data frames

Solution to the task

Printing `df` to the console gives the impression that we are working with an array-type object but we are not—remember that `typeof(df)` is `list()`!

Modifying a single row implicates modifying *all* columns, i.e. all `list` elements. This cannot be tracked by `tracemem()` but can be seen using `lobstr::ref()`:

```
# overwrite first entries
df[1, ] <- c(42, 42)
# check memory addresses
ref(d1)

# now modify only one variable
df$x <- c(7, 7, 7)
ref(d1)
```

Slide 16: Exercises

Solution to the tasks

1. `1:10` is no binding so there's no point in tracing a value.
2. In this example `x` is an integer vector (note that `x` as defined on Slide 11 has `double` type). Replacing `3L` with `4`, i.e. integer with double, triggers coercion of `x` to `double`. Coercion *always* results in a copy!
3. `:` is special in the sense that it may generate integer sequences that can be stored using only the first and the last elements. The length of the generated sequenced doesn't affect the memory required to store the value.

Slide 18: Case Study: Copy-On-Modify Inferno

- Note that `vapply()`'s `FUN.VALUE` requires a template for the return value (which is numeric 1×1 here)
- We will come back to consequences of this behavior in the Chapter *Improving Performance* and benchmark against alternatives that require less copies.

Slide 19: Case Study: Copy-On-Modify Inferno

Note that Using the `$` operator would make no difference here: `$<-` also has a `$<-.data.frame` method. The output of `tracemem()` then looks similar to this:

```
tracemem[0x7fe515db8d88 -> 0x7fe513bcd788]:
tracemem[0x7fe513bcd788 -> 0x7fe513b79a88]: $<-.data.frame $<-
tracemem[0x7fe513b79a88 -> 0x7fe513b79c88]: $<-.data.frame $<-
```

(Note that the addresses will not match the ones you will see on your device. These are location in your random-access memory (RAM) so they are different for different devices and also *randomly assigned*.)

Slide 20: Case Study: Copy-On-Modify Inferno

- The two copies made by `[[<-.data.frame` are **shallow copies** (only column references are copied)
- **Q**: What kind of function is `[[<-.data.frame`?
  A: A regular function. It is a method of `[[<-` which is a primitive (a fast C function)
- **Q**: How can you view the source?
  A: Use backticks around `[[<-.data.frame`
- More on primitives on the next slides. More on methods, dispatch etc. in the 'OOP' Chapter.
- More on how to write efficient code (and especially efficient `for()` loops) in Chapter 'Improving Performance'

Slide 21: Case Study: Copy-On-Modify Inferno

- A single copy is made from internal C code the first time we use `[[<-`
- This a good example where tweaking the code reduces the amount of copies made
- If such a solution is not readily at hand we may resort to C++ code. More on this in the `Rcpp` chapter.

Slide 28: First-Class Functions

Solution to the task

- `(` is a primitive:

  ```
  `(`
  ```

- The R-help hints that `(` is semantically equivalent to `function(x) x`, so `(` evaluates its argument.

  Therefore, `(x<-5)` is valid (and useful) R code: we bind to and then evaluate `x`

  `(function(x) x^2)(5)` defines an anonymous function and with the argument `x = 5`.

Slide 31: Lexical Scoping

Solution to the task

```
rm(x) # remove `x` from GE
```

```
## Warning in rm(x): object 'x' not found
```

```
z <- function(x) x^2

# 3. everything in f() happens in an _ephemeral_ environment
f <- function(g) {
    if(!exists("x")) {
        x <- 1
    } else {
        x <- x + 1
    }
    z <- 2                 # 1. name masking (`z` is defined in GE)

    z(x + y)               # 2. functions before variables (`z()` is defined in GE)
}

y <- 20
f(x)                       # 4. dynamic lookup: R will search for y when f() actually
                           # needs it for computing z(x + y)---R does not care
                           # for the value when we create f().
```

```
## [1] 441
```

Slide 35: Lazy Evaluation

Solution to the task

- Due to lazy evaluation, the evaluation environment for *default arguments* is the ephemeral function environment
- User supplied arguments are evaluated in the parent environment (GE here)

Slide 36: Lazy Evaluation

Solution to the task

- `NULL` represents the null object in R: `NULL` is used mainly to represent a list with zero length, and is often returned by expressions and functions whose value is *undefined*.
- Without lazy evaluation this statement would throw an error because `x > 0` evaluates to a logical value of length zero (you cannot compare `NULL` to `double`)
- Control flow stops after evaluating the first part of the condition in `if()`: the second statement would be evaluated only if the first is `TRUE` (here it is `FALSE`).

Slide 37: Exercises

Solution to the tasks

1. Lazy evaluation of default arguments happens at function definition, *not invocation*: R looks for `z` in the global environment because `x` is not `z^2` by default (like in the fixed version below).

   ```
   f <- function(x = z^2, z) {
     z + x^2
   }
   f(2, 2)
   ```

   A workaround using `with`:

   ```
   with(list(z = 2), f(x = z^2, z))
   ```

2. `...` enables us to pass arguments the body of `f()` (to other functions!) which do not have to be specified when defining `f()`.

   Here, `f()` is a simple *wrapper* that returns names of list elements as passed to the `...` argument.