Lecture 14 – Using Memory Efficiently in R

Learning Objectives:

3. Learn the basic principles of software design.

    3.4. Learn about optimization and profiling code.

Install packages: pryr

Optimization continued (memory)

- Using memory efficiently makes code run faster and use fewer resources.

- Understanding memory management in R will help design good code.

Things to Know:

- How R allocates and clears memory

- How to check and profile memory usage

- How to avoid memory leaks and making copies

Memory in R

- Memory allocation occurs in RAM, unless using a big data package (which will write objects it to static memory).

- R does request memory in chunks and will allocate small amounts (128 bytes or less) for small vectors itself. Anything larger, it will farm task to OS.

- *Substantial differences between Mac and Windows.*

  - Windows imposes a physical memory limit on each instance of R.

  - Trying to find contiguous chunk of memory within this limit will lead to out-of memory errors quickly with large data sets.

  - Mac/Unix has no such limit

For Windows only:

- Check memory allocated to R:

```
> memory.limit()
```

- Check memory used in R:

```
> memory.size()
```

Checking Memory Usage

The utilities package (pre-loaded):

- Check object size.

```
> object.size(x)
```

- Check memory usage.

```
> gc()
```

- Check memory change with object.

```
> tracemem(x)
```

The pryr package

```
> library(pryr)
```

- Check object size.

```
> object_size(x)
```

- Check object location.

```
> address(x)
```

- Check memory usage.

```
> mem_used()
```

- Check memory change with action.

```
> mem_change()
```

Memory in R (continued)

- Each R object has some standard data:
  - *Object metadata*: base data type, info for debugging, memory management
  - *Two pointers*: one points to next object in memory, one points to previous object so R can loop through memory.
  - *One pointer*: to attributes.

- Vectors have some additional elements:
  - *Length of vector*: Total length of vector (up to 2^{52})
  - *Data:* data associated with vector.

Binding and Memory Usage

- Binding a value to a name: creates an address in physical memory

```
> x <- runif(1e6)
```

same

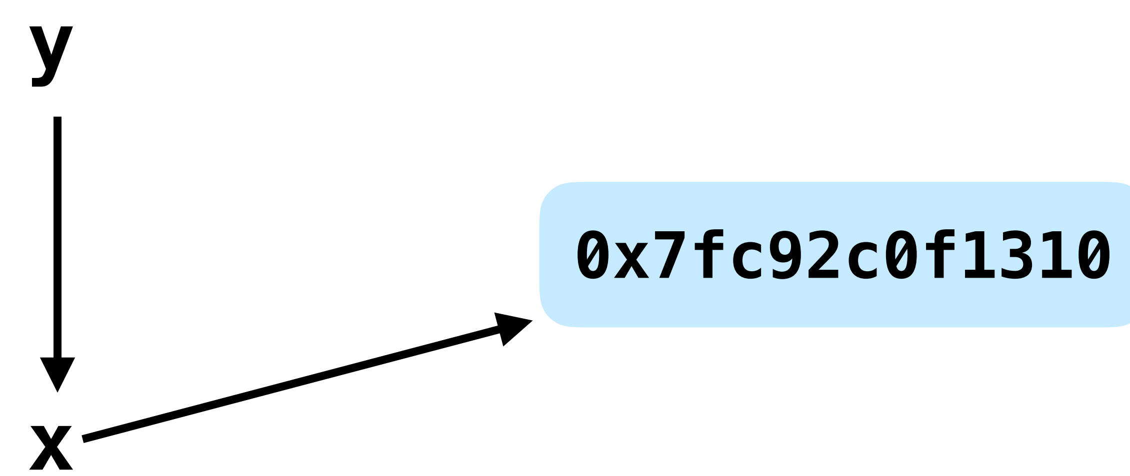
```
> object.size(x)
> object_size(x)
```

```
> y <- list(x,x,x)
```

different!

```
> object.size(y)
> object_size(y)
```

```
mem_used()
```



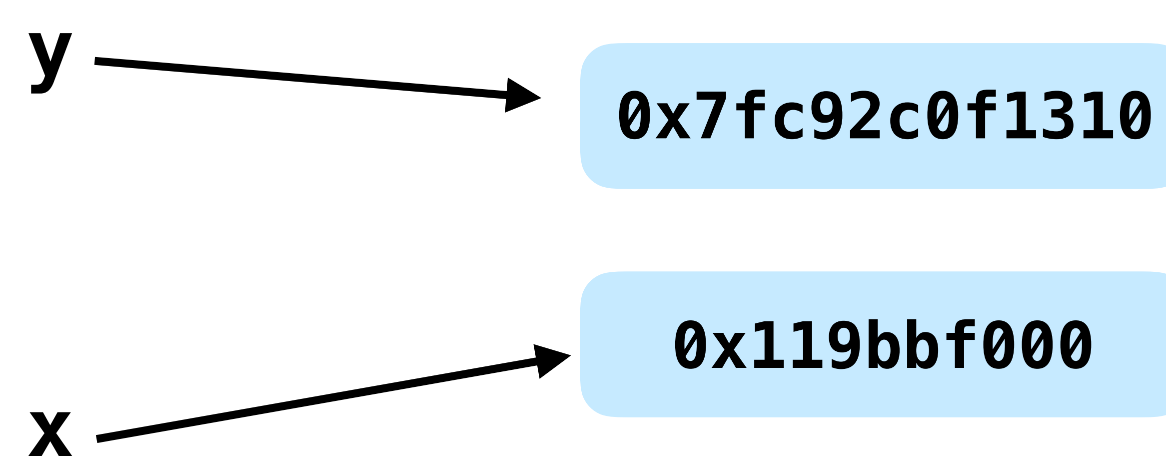

```
> x[20] <- 140
```

same `> object_size(x)`

different! `> address(x)`

```
> object_size(y)
```

`mem_used()` **a lot more!**



Garbage Collection in R

- Garbage collection: reclaims memory from removed objects or objects no longer used.

    - `gc()` initiates garbage collection and reports memory usage.

    - `gcinfo()` sets `gc(verbose=TRUE)` and will report when garbage collection happens (will normally happen in background).

- Removes anything in memory without pointers associated with it.

- Happens automatically when R needs more space, no need to manually call `gc()`.

- Garbage collection is relatively slow, it can recover a lot of memory, but will also slow code down. Trade off!

Memory Leaks: Example 1

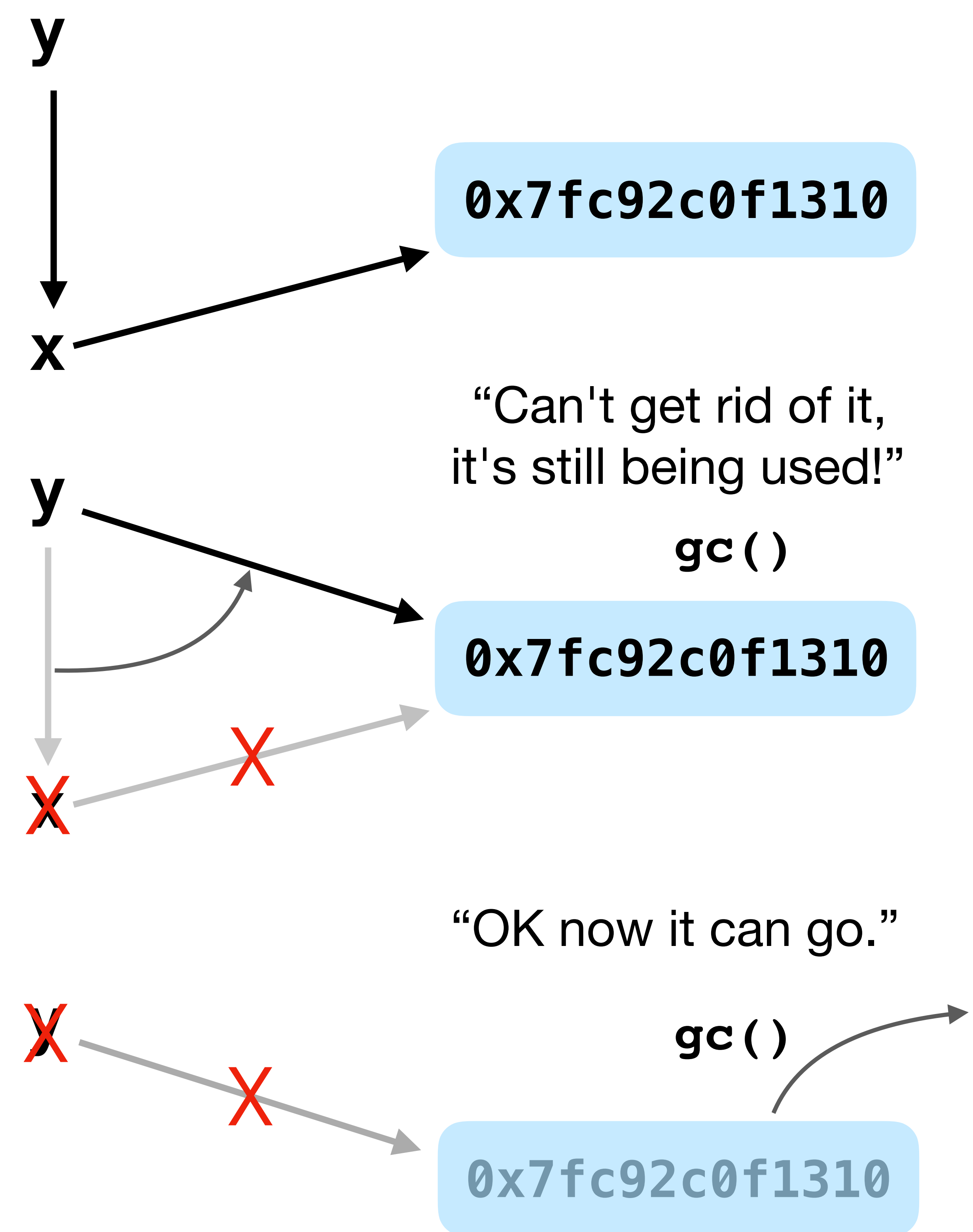
```
> mem_change(x <- runif(1e8))

> y <- list(x,x,x)

> mem_used()
```

```
> mem_change(rm(x))

> mem_used()
```

no change!

```
> mem_change(rm(y))

> mem_used()
```

back to baseline

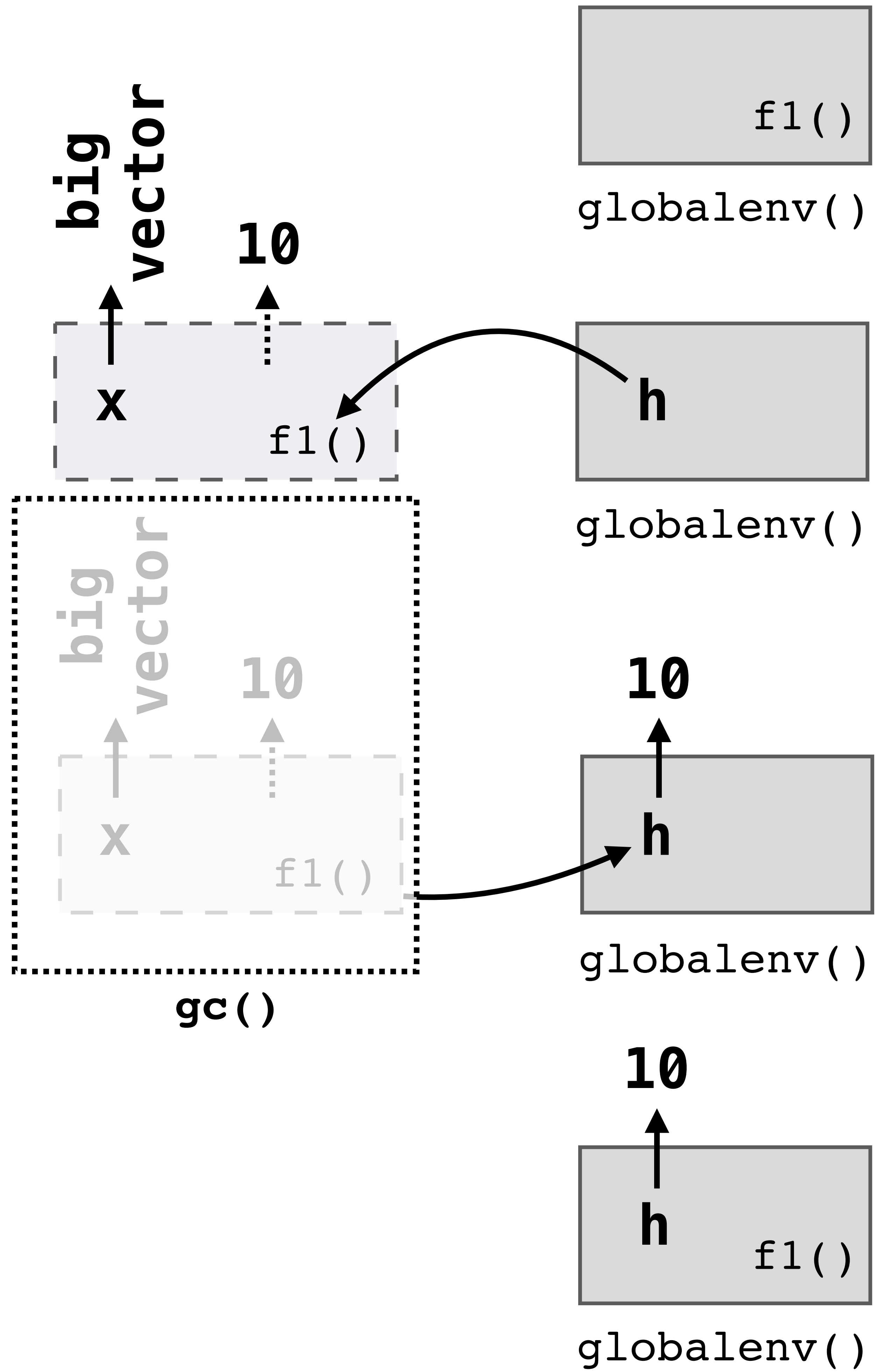


Make sure you know where your pointers are pointing!

Memory Leaks: Example 2

```
f1 <- function(){
  x <- 1:1e6
  10
}
```

```
> mem_change(h <- f1())
```

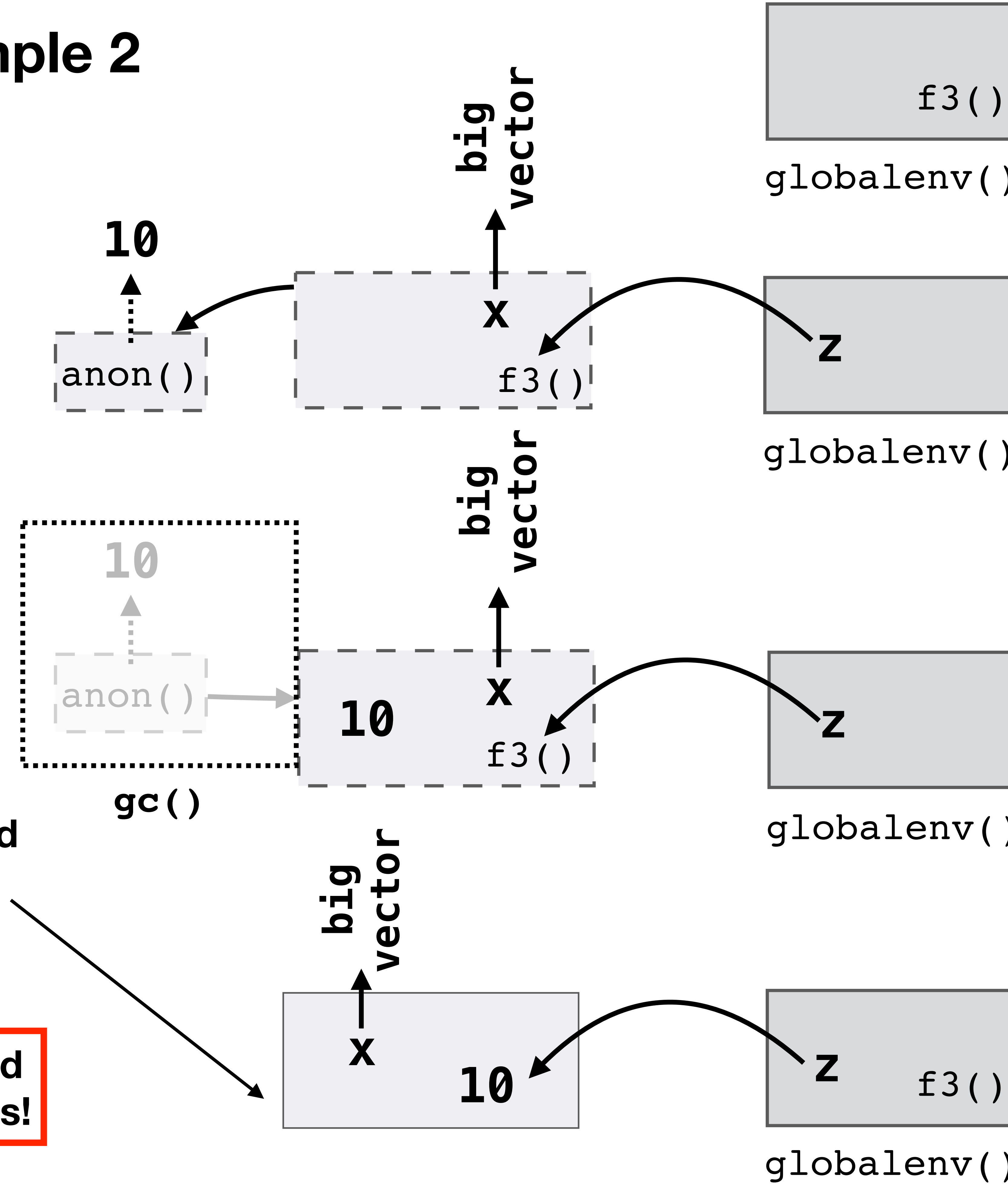



```
> object_size(h)
```

Memory Leaks: Example 2

```
f3 <- function(){
  x <- 1:1e6
  function () 10
}
```

```
> mem_change(z <- f3())

> object_size(z)
```



Object size includes associated environment, including x even if x isn't in `globalenv()`

BE CAREFUL with nested and anonymous functions!

Memory Leaks: Example 3

- Loops producing copies, growing object size in loop

```
g <- letters[1:26]
alphabet <- g[1]
for (i in 2:length(g)){
   alphabet <- c(alphabet, g[i])
   print(object_size(alphabet))
}
```

grows in size

`c()` makes copies!!

- These functions have to find new space, then copy over the old info to the new space.

Other offenders:

- `rbind()`
- `cbind()`
- `append()`
- `paste()`
- `as.data.frame()`

Consider modifying in place!

```
g <- letters[1:26]
alphabet <- seq(1,26)
for (i in 1:length(g)){
   alphabet[i] <- g[i]
   print(object_size(alphabet))
}
```

Primitive functions don't make copies.

Profiling code

- the `profvis` package, same as our last lecture!

- Rerun the pre and post refactoring code from last time.

Group work: In the post-refactored code, `inpolygon` is the most memory-intensive line.

1. Can you identify the lines that use the most memory on `inpolygon`?
2. Is garbage collection acting efficiently to reclaim space? If not, how could it be changed?
3. Is there anything else that could be changed to improve the speed and memory usage of this code?

More Information

http://adv-r.had.co.nz/memory.html – Memory Usage (*Advanced R*)

https://adv-r.hadley.nz/perf-improve.html#avoid-copies
– Avoiding copies (*Advanced R*)