

# General principle: Memory allocation

WRITING EFFICIENT R CODE



Colin Gillespie

Jumping Rivers & Newcastle University



# If we programmed in C...

## WE'RE IN CHARGE OF MEMORY ALLOCATION

```
// C code: request memory for a number  
x = (double *) malloc(sizeof(double));  
  
// Free the memory  
free(x);
```

- In R, memory allocation happens automatically
- R allocates memory in RAM to store variables
- Minimize variable assignment for speed

# Example: Sequence of integers

1, 2, ..., n

The obvious and best way

```
## Method 1  
x <- 1:n
```

Not so bad

```
## Method 2  
x <- vector("numeric", n) # length n  
for(i in 1:n)  
+     x[i] <- i
```

Don't ever do this!

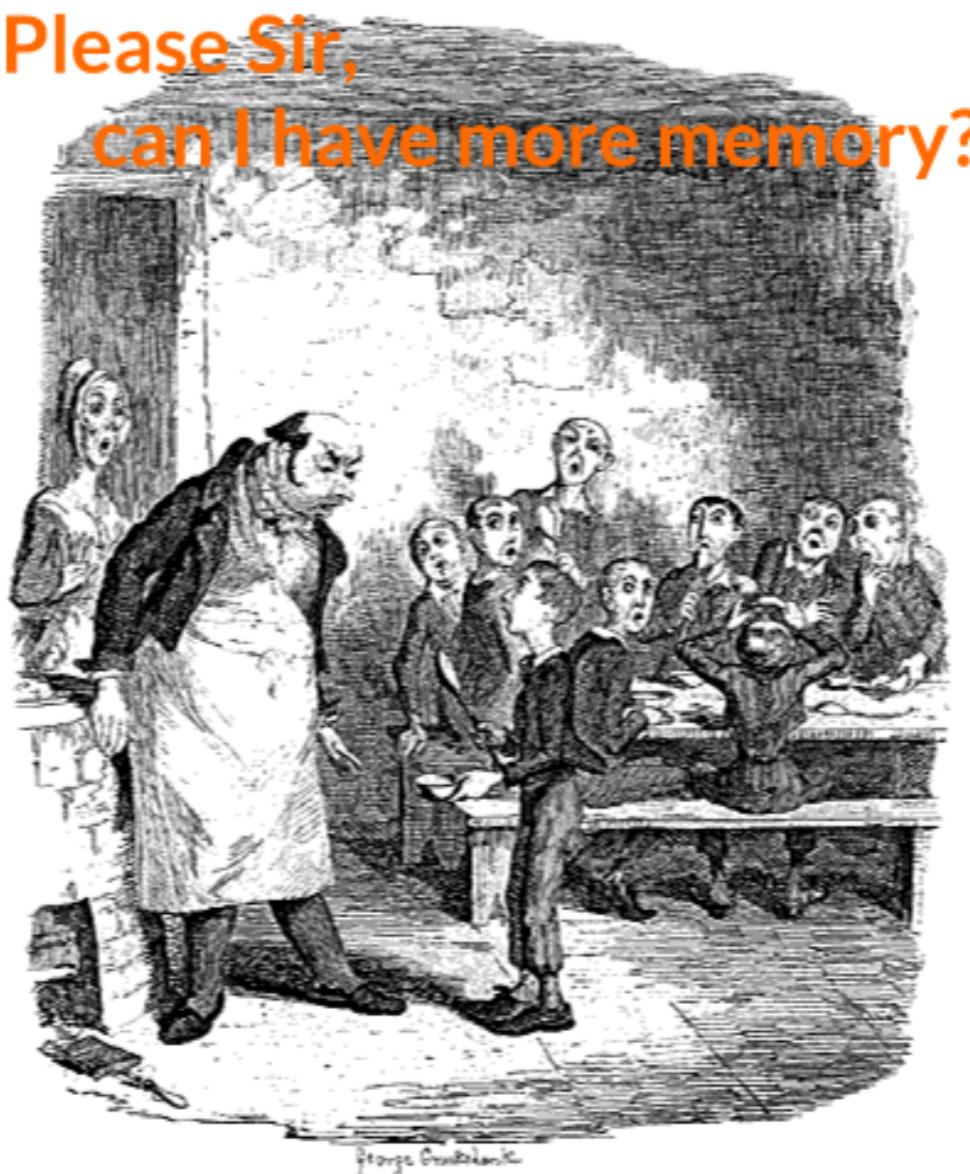
```
## Method 3  
x <- NULL # Length zero  
for(i in 1:n)  
+     x <- c(x, i)
```

# Benchmarking

- Method 1: `1:n`
- Method 2: Preallocate
- Method 3: Growing

TIME IN SECONDS

n	1	2	3
$10^5$	0.00	0.02	0.2
$10^6$	0.00	0.2	30
$10^7$	0.00	2	3800



# Welcome to R club!

The first rule of R club: never, ever grow a vector.

# **Let's practice!**

**WRITING EFFICIENT R CODE**

# The importance of vectorizing your code

WRITING EFFICIENT R CODE



Colin Gillespie

Jumping Rivers & Newcastle University

# General rule

- Calling an R function eventually leads to C or FORTRAN code
  - This code is *very* heavily optimized

## Goal

- Access the underlying C or FORTRAN code as quickly as possible; the fewer functions call the better.

# Vectorized functions

- Many R functions are vectorized
  - Single number but return a vector

```
rnorm(4)
```

```
-0.7247  0.2502  0.3510  0.6919
```

- Vector as input

```
mean(c(36, 48))
```

```
42
```

# Generating random numbers

```
library(microbenchmark)
n <- 1e6
x <- vector("numeric", n)
microbenchmark(
+   x <- rnorm(n),
+   {
+     for(i in seq_along(x))
+       x[i] <- rnorm(1)
+   },
+   times = 10
+ )
```

```
# Unit: milliseconds
# expr      lq  mean    uq  cld
# rnorm(n)  60   70   80  a
# Looping  2600 2700 2800  b

## Output trimmed for presentation
```

## Compare

```
x <- vector("numeric", n)
for(i in seq_along(x))
  x[i] <- rnorm(1)
```

to

```
x <- rnorm(n)
```

# Why is the loop slow?

## LOOPING

```
x <- vector("numeric", n)
for(i in seq_along(x))
+   x[i] <- rnorm(1)
```

## ALLOCATION

```
x <- vector("numeric", n)
```

- *Loop*: One-off cost
- *Vectorized*: Comparable

## GENERATION

- *Loop*: one million calls to `rnorm()`
- *Vectorized*: a single call to `rnorm()`

## ASSIGNMENT

- *Loop*: One million calls to the assignment method
- *Vectorized*: a single assignment

# R club

The second rule of R club: use a vectorized solution wherever possible.

# **Let's practice!**

**WRITING EFFICIENT R CODE**

# Data frames and matrices

WRITING EFFICIENT R CODE



Colin Gillespie

Jumping Rivers & Newcastle University

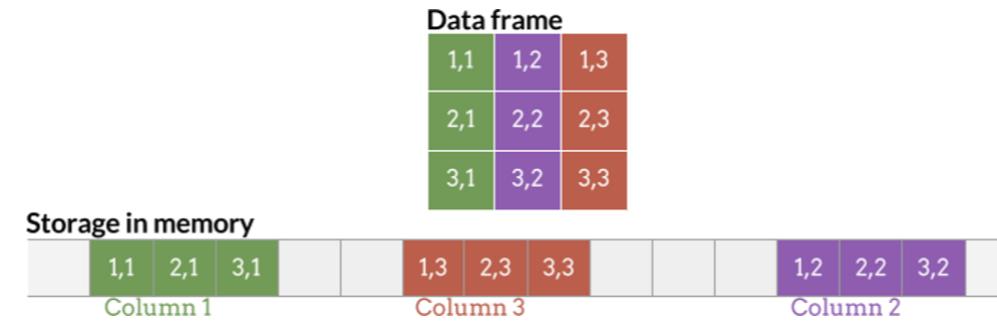
# The data frame

- Key data structure in R
  - Copied in other languages
    - Python: pandas data frame
    - If you can't beat them, join them!

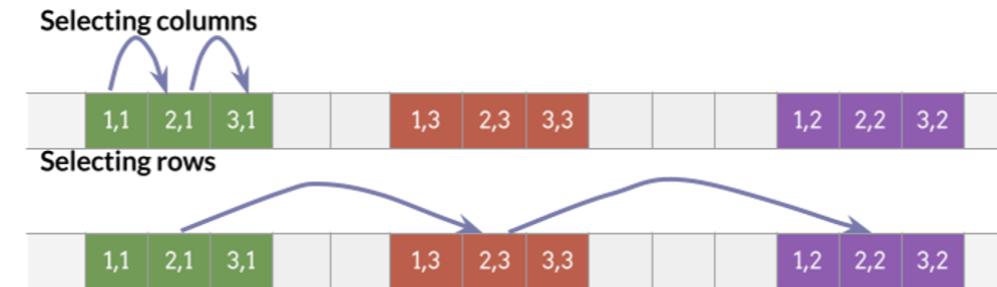
# Data frames

- Tabular structure: rows and columns
  - `read.csv()` and friends returns a data frame
    - Columns
    - Data must be the same type
      - Rows
    - Different type

## STORAGE



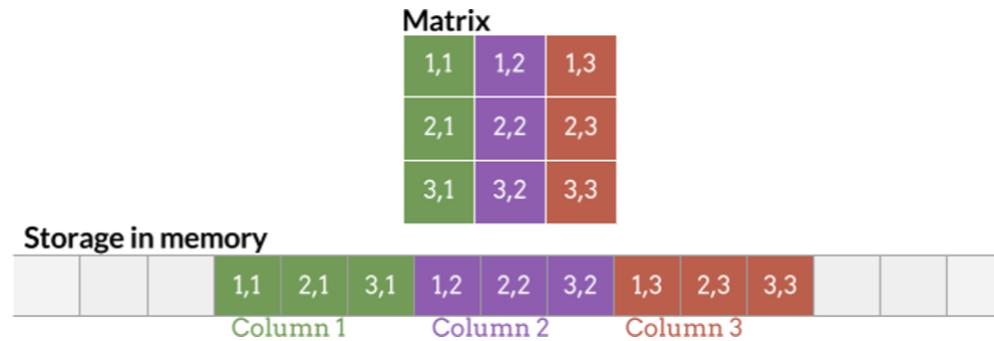
## SELECTION



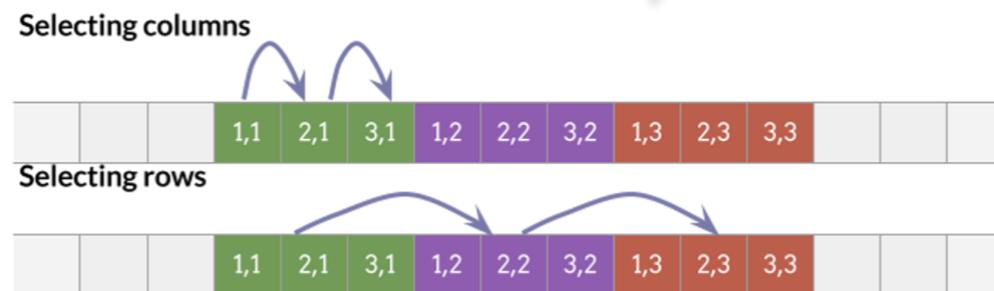
# Matrices

- It's a rectangular data structure
  - You can perform usual subsetting and extracting operations
  - BUT - every element must be the same data type

## STORAGE



## SELECTION



# R club

The third rule of R club: Use a matrix whenever appropriate.

# **Let's practice!**

**WRITING EFFICIENT R CODE**