# Topic 10: Introduction to C/C++ with R

Irina Gaynanova

# Prologue

- See PUBLIC Github repository with some examples
- For Mac users:
  - You may need to install **gfortran** in addition to already installed **Xcode Command Line Tools**
  - Troubleshooting for Mac users
- For Windows users:
  - If you installed **Rtools**, your should be set
- Install R packages **Rcpp** and **RcppArmadillo**

```
install.packages(c("Rcpp", "RcppArmadillo"))
```

# Some useful Rcpp references

- **Official vignettes** for Rcpp package
- **Official vignette** for RcppArmadillo package
- **H.Wickham's Advanced R**
- **H.Wickham's R packages**
- **Rcpp for everyone by M. Tsuda**

# I know R, why learn C++?

- **R** (high-level language) can be very slow
- **C** (mid-level language) is fast, powerful and widely-used, semi-friendly
- **Assembler** (low-level language) is very fast, not friendly

**C++** (inherits most of **C** syntax) has easy and powerful interfacing with **R** with the help of various R packages (Rcpp, RcppArmadillo, RcppEigen)

# When is it worth the effort to move my code to C++?

*C++* can help significantly if

- you have loops that cannot be vectorized due to dependence on previous iterations (e.g. kmeans, steepest descent/Newton's method, coordinate descent, MCMC chain updates)
- "Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than in R." (H. Wickham, Advanced R)

Our focus will be on moving relatively small chunks of code into C++ focusing on specific bottlenecks

# R example - whether number is odd or even

Let's write a function that determines whether the given number is even or odd

```
isOddR <- function(num){
  result <- num %% 2 == 1
  return(result)
}
isOddR(10)
```

```
## [1] FALSE
```

```
isOddR(13)
```

```
## [1] TRUE
```

# Same function in C++

```
bool isOddCpp(int num){
  bool result = (num%2 == 1);
  return result;
}
```

Compared to R

- we have to define the type of each variable, including the type of input variables (**int** - integer; **bool** - logical)
- each statement has to finish with semi-colon **;**
- we have to make explicit return statement
- only = can be used for assignments, no <-
- Some commands may have different syntax (%% versus %)

# Basic variable types of Rcpp

| Rcpp | type |
|---|---|
| int | integer |
| float | scalar, single precision (~7 digits) |
| double | scalar, double precision (~15 digits) |
| bool | logical |
| char | character |

# Simplest C++

We can directly use functions written in C++ in R via Rcpp

```
library(Rcpp)
cppFunction("
bool isOddCpp(int num){
  bool result = (num%2 == 1);
  return result;
}")
isOddCpp(10)
```

```
## [1] FALSE
```

- **cppFunction()** compiles, links and imports corresponding code into R

# Moderate C++

What if we have tons of C++ code? Should we wrap it all manually?

- No. Save as C++ file and then source from within R. In Rstudio, File $->$ New File $->$ C++ file
- Rstudio is smart and tries to make your job easier. What do you see in the .cpp file?

# .cpp files for Rcpp

This includes Rcpp header and states we are using Rcpp namespace so we don't need to write Rcpp::NumericVector

```
#include <Rcpp.h>
using namespace Rcpp;
```

This is a comment within C++, starts with //

```
// This is a simple example
```

This is exporting the C++ function defined right after for use in R

```
// [[Rcpp::export]]
```

**NumericVector** - Rcpp type for a vector with numeric elements

```
NumericVector x
```

*Note:* while NumericVector type already exists in Rcpp, we will eventually work with vectors in Armadillo C++ library, so we will use NumericVector types from standard Rcpp only temporary

# .cpp files for Rcpp

Save the file as Test.cpp. Use either **Source** at the top or

```
library(Rcpp)
sourceCpp("Test.cpp")
```

```
##
## > timesTwo(42)
## [1] 84
```

```
x <- c(1,2,3)
timesTwo(x)
```

```
## [1] 2 4 6
```

Note that two things happened: R code within cpp file run, and extra R code run

# Cumulative sum in R

Let's write a function that returns cumulative sums of the vector elements, i.e. for $x = (1, 3, 5)$ it will return $s = (1, 4, 9)$

```
cumul_sumR <- function(x){
  p <- length(x)
  s <- x
  for (i in 2:p){
    s[i] <- s[i-1] + x[i]
  }
  return(s)
}
cumul_sumR(x = c(1, 3, 5))
```

```
## [1] 1 4 9
```

# Cumulative sum in C++

```
NumericVector cumul_sumCpp(NumericVector x){
  int p = x.size(); // length of x
  NumericVector s = x;
  // indexing starts with 0 rather than 1
  for(int i = 1; i < p; i++){
    s[i] = s[i-1] + x[i];
  }
  return(s);
}
```

- **NumericVector** - vector of numeric values
- **.size()** - analog of **length** command in R that works with **NumericVector** type
- indexing of vectors (and matrices) **starts from 0** rather than 1
- **i++** is equivalent to **i = i + 1**, but slightly faster

# Cumulative sum in C++ from R

Using the C++ function in R via Rcpp

```
library(Rcpp)
cppFunction("NumericVector cumul_sumCpp(NumericVector x){
  int p = x.size();
  NumericVector s = x;
  for(int i = 1; i < p; i++){
    s[i] = s[i-1] + x[i];
  }
  return(s);
}")
cumul_sumR(x = c(1, 3, 5))
```

```
## [1] 1 4 9
```

```
cumul_sumCpp(x = c(1, 3, 5))
```

```
## [1] 1 4 9
```

# Cumulative sum in C++ from R

```
cppFunction("NumericVector cumul_sumCpp(NumericVector x){
            int p = x.size();
            NumericVector s = x;
            for(int i = 1; i < p; i++){
            s[i] = s[i-1] + x[i];
            }
            return(s);}")
x = c(1, 3, 5)
s = cumul_sumCpp(x)
print(s)
```

```
## [1] 1 4 9
```

```
print(x)
```

```
## [1] 1 4 9
```

What happened to $x$?

# What happened to $x$?

# Variable types: vectors

- NumericVector always passes vectors as pointers (even without explicit &, more on this later)
- Exact vector copying is a dangerous zone!!!
  - Copy a vector by clone

# Cumulative sum, C++ from R

Variation using the cloning

```
cppFunction("NumericVector cumul_sumCpp(NumericVector x){
            int p = x.size();
            NumericVector s = clone(x); // cloning
            for(int i = 1; i < p; i++){
            s[i] = s[i-1] + x[i];
            }
            return(s);}")
x = c(1, 3, 5)
s = cumul_sumCpp(x)
print(s)
```

```
## [1] 1 4 9
```

```
print(x)
```

```
## [1] 1 3 5
```

# Cumulative sum, C++ from R

Variation using a new vector of the same size

```
cppFunction("NumericVector cumul_sumCpp(NumericVector x){
             int p = x.size();
             NumericVector s(p); // new vector of the same s
             s[0] = x[0]; // extra 1st element initializatio
             for(int i = 1; i < p; i++){
             s[i] = s[i-1] + x[i];
             }
             return(s);}")
x = c(1, 3, 5)
s = cumul_sumCpp(x); print(s)
```

```
## [1] 1 4 9
```

```
print(x)
```

```
## [1] 1 3 5
```

# Cumulative sum, R vs C++

```
library(microbenchmark)
p = 1000
x = rnorm(p)
identical(cumul_sumR(x), cumul_sumCpp(x))
```

```
## [1] TRUE
```

```
microbenchmark(
  cumul_sumR(x),
  cumul_sumCpp(x), times = 50
)
```

```
## Warning in microbenchmark(cumul_sumR(x), cumul_sumCpp(x)
## accurate nanosecond times to avoid potential integer ove
```

```
## Unit: microseconds
##              expr    min     lq     mean median     uq
##    cumul_sumR(x) 40.344 41.902 42.40302 42.025 43.132 45
##  cumul_sumCpp(x)  1.927  2.255  2.69042  2.337  2.501  9
```

# Success

# Other variable types

| R | Rcpp |
|---|---|
| matrix | NumericMatrix |
| strings | CharacterVector |
| list | List |

# Bootstrap example

We would like to calculate confidence interval around the sample mean and sample standard deviation using **bootstrap**

Want to take $B$ samples **with replacement** from given data $x \in \mathbb{R}^n$, calculate mean/st.dev on each sample

# Bootstrap for confidence intervals

- **Given:** sample $x_1, \ldots, x_n$, sample mean $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, sample sd $s = \mathrm{sd}(x_1, \ldots, x_n)$
- **Goal:** construct confidence interval for true $\mu$
- **Bootstrap approach**: repeat $B$ times ($B$ is very large, e.g. 1000)
  - sample $n$ points $\widetilde{x}_1, \ldots, \widetilde{x}_n$ out of $x_1, \ldots, x_n$ *with replacement*
  - construct $\widetilde{x}_b$ as the sample mean of $\widetilde{x}_1, \ldots, \widetilde{x}_n$
  - construct $\widetilde{s}_b$ as the sample sd of $\widetilde{x}_1, \ldots, \widetilde{x}_n$
- The empirical distribution of resulting $\widetilde{x}_b$ and $\widetilde{s}_b$, $b = 1, \ldots, B$, can be used for construction of confidence intervals

# Bootstrap example in R

```r
# ds - vector of observations
# B - number of bootstrap samples
bootstrap_r <- function(ds, B = 1000){
  boot_stat <- matrix(NA, nrow = B, ncol = 2)
  n <- length(ds)
  # Perform bootstrap
  for(i in 1:B) {
    # Create a sample of size n with replacement
    gen_data <- ds[sample(n, n, replace=TRUE)]
    # Calculate sample data mean and SD
    boot_stat[i,] <- c(mean(gen_data),sd(gen_data))
  }
  return(boot_stat)
}
```

# Bootstrap example

Let's check how this works in practice

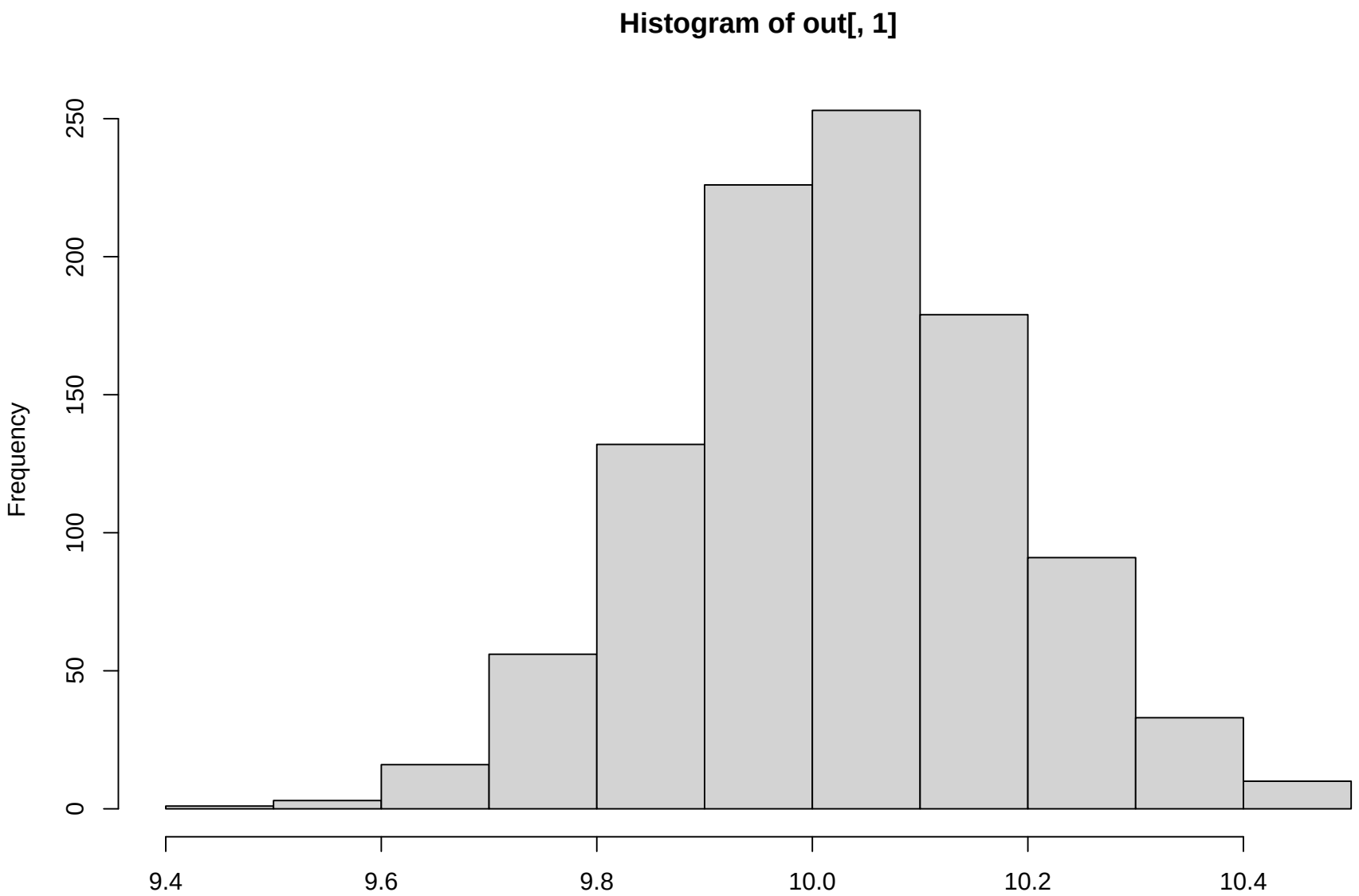
```
ds <- rnorm(1000, mean = 10, sd = 5)
out <- bootstrap_r(ds)
library(microbenchmark)
microbenchmark(bootstrap_r(ds), times = 10)
```

```
## Unit: milliseconds
##              expr      min       lq     mean   median
##  bootstrap_r(ds) 22.82978 23.16299 23.95334 23.45811 25.
```

# Bootstrap example

Distribution of mean over samples (truth 10)

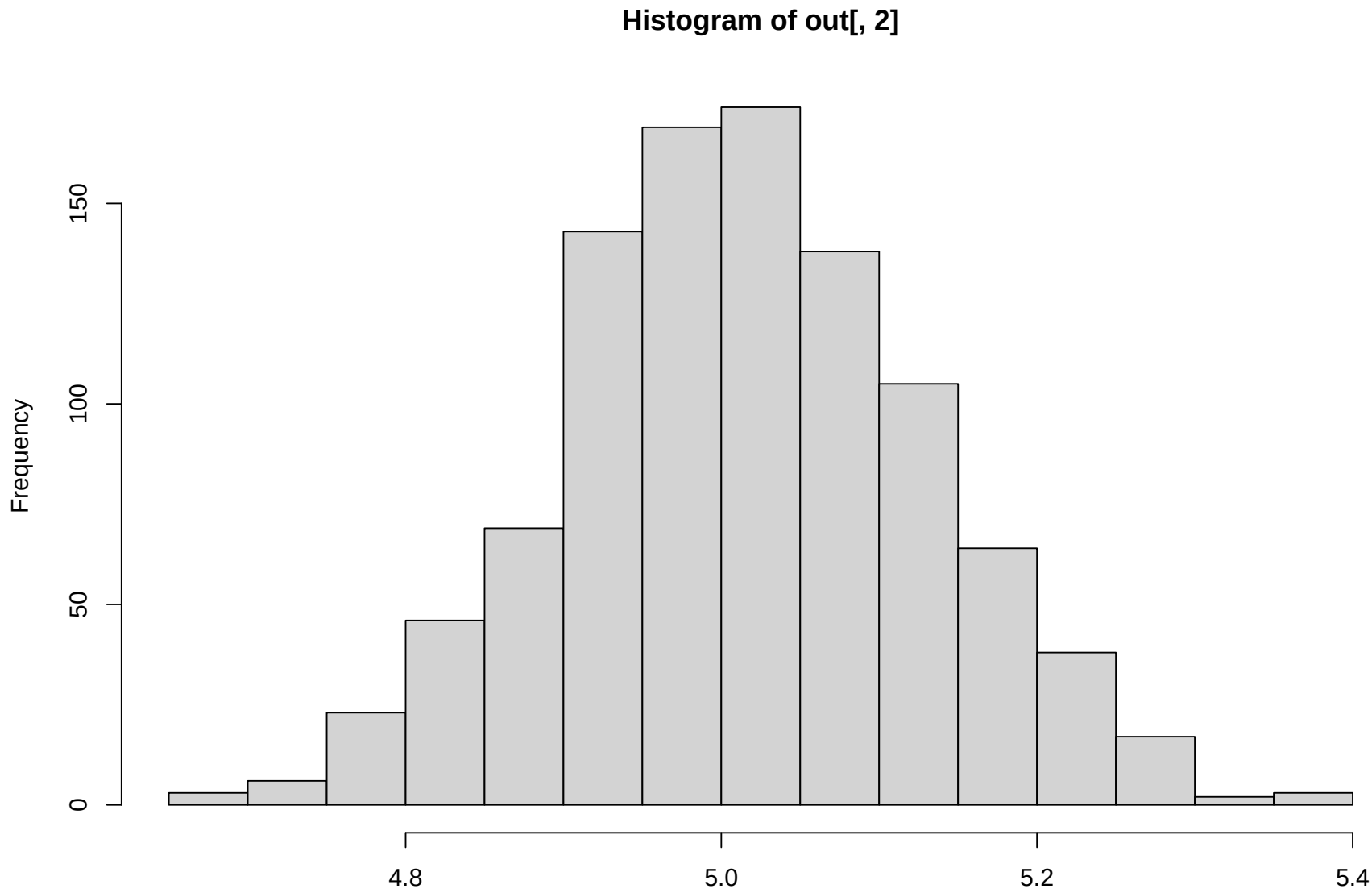
```
hist(out[, 1])
```



# Bootstrap example

Distribution of sd over samples (truth 5)

```
hist(out[, 2])
```



# Transfering the code to C++

# Transfering the code to C++

R wrapper

```r
# ds - vector of observations
# B - number of bootstrap samples
bootstrap_r <- function(ds, B = 1000){
  # Function code, returns B by 2 matrix
}
```

Rcpp wrapper

```cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix bootstrap_cpp(NumericVector ds,
int B = 1000) {
// Function goes here
}
```

# Transfering the code to C++

Rcpp wrapper

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix bootstrap_cpp(NumericVector ds,
int B = 1000) {
// Function goes here
}
```

- **NumericMatrix** - Rcpp matrix type, **NumericVector** - Rcpp vector type, **int** - integer

# Transfering the code to C++

Pre-allocation of storage in R

```
boot_stat <- matrix(NA, nrow = B, ncol = 2)
n <- length(ds)
```

Pre-allocation of storage in Rcpp

```
// Preallocate storage for statistics
NumericMatrix boot_stat(B, 2);
// Number of observations
int n = ds.size();
```

- ds.**size()** is used as length command (on input ds)

# Transfering the code to C++

For loop skeleton in R

```
for(i in 1:B) {
  # Assignment to the ith vector element
  x[i] = result
  }
```

For loop skeleton in Rcpp

```
for(int i = 0; i < B; i++) {
  // Assignment to the ith vector element
  x[i] = result;
}
```

- Indexing starts from 0 (rather than from 1)
- i++ means i gets increase by 1 at each loop (equivalent to $i = i + 1$ but slightly faster)

# Transfering the code to C++

Sampling with replacement in R

```
gen_data <- ds[sample(n, n, replace=TRUE)]
```

Sampling with replacement in Rcpp

```
NumericVector gen_data = ds[floor(runif(n, 0, n))];
```

- floor(runif(n,0,n)) - only return from 0 to n-1, consistent with indexing
- gen_data was not created in advance, so need to specify the type on first creation

# Transfering the code to C++

Calculate sample data mean and SD within each replication i in R

```
boot_stat[i,] <- c(mean(gen_data),sd(gen_data))
```

In Rcpp

```
boot_stat(i, 0) = mean(gen_data);
boot_stat(i, 1) = sd(gen_data);
```

- matrix indexing using (,) rather than [,]
- indexing still starts from 0 rather than 1

# Transfering the code to C++

Total function

```
NumericMatrix bootstrap_cpp(NumericVector ds,
int B = 1000) {
// Preallocate storage
NumericMatrix boot_stat(B, 2);
  int n = ds.size();
  // Perform bootstrap
for(int i = 0; i < B; i++) {
 // Sample initial data
  NumericVector gen_data = ds[ floor(runif(n, 0, n)) ];
  // Calculate sample mean and std dev
  boot_stat(i, 0) = mean(gen_data);
  boot_stat(i, 1) = sd(gen_data);
}
// Return bootstrap results
return boot_stat;
}
```

# Bootstrap code in C++ vs R

```
library(Rcpp)
sourceCpp("Bootstrap.cpp")
set.seed(2308)
ds <- rnorm(1000, mean = 10, sd = 5)
set.seed(34)
outR <- bootstrap_r(ds)
set.seed(34)
outCpp <- bootstrap_cpp(ds)
```

# Dreaded for loop

```
library(microbenchmark)
microbenchmark(bootstrap_cpp(ds), bootstrap_r(ds), times =
```

```
## Unit: milliseconds
##                 expr      min       lq     mean   median
##  bootstrap_cpp(ds) 12.40722 13.01926 14.50101 15.05565 1
##    bootstrap_r(ds) 22.55713 22.97111 24.21550 23.90202 2
##  cld
##   a
##    b
```