# High Performance R Programming

## Serial Performance (Rcpp)

# Introduction to Rcpp

1. What is Rcpp?
2. Essential Rcpp examples
3. Under the hood
4. Practical Rcpp advice
5. Performance considerations when programming in R and Rcpp
6. Dimension reduction example
7. Some profiling tools

# Motivation

- Many of us prefer R for the computing portion of our work.
- R programming is convenient, but resulting code can be slow. This slowness can be quite painful in large simulations or with very large datasets, for example.
- Rcpp allows us to write the slow parts of your code in C++ (fast, efficient memory use) and call it through plain R (friendly & familiar).

# C and C++

- C is a low-level, imperative procedural language.
  1. It was designed to be compiled.
  2. Provides low-level access to memory.
  3. Provides language constructs that map efficiently to machine instructions.

- C was originally developed by Dennis Ritchie between 1969 and 1973 at Bell Labs, and used to re-implement the UNIX operating system.

- C++ was developed by Bjarne Stroustrup in 1979 at Bell Labs, as an extension of C.

- C++ builds on standard C with features such as function overloading and support for object-oriented programming.

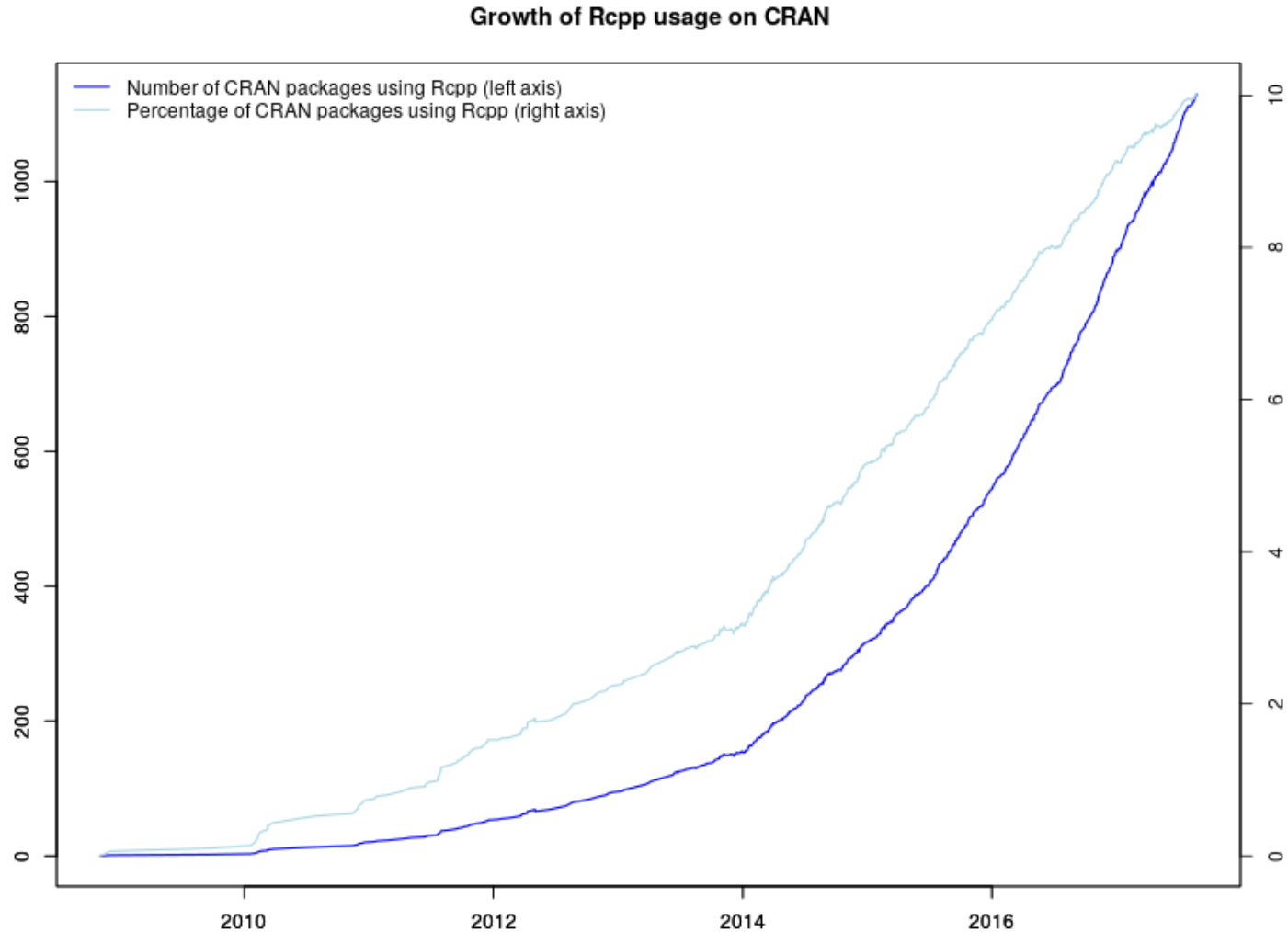
- Libraries such as the Standard Template Library (STL) that provide a variety of data structures and algorithms for C++ programmers.

# What is Rcpp?

- Rcpp is a suite of packages that facilitates interoperability between R and C++ written mainly by Dirk Eddelbuettel and Romain Francois.

- The Rcpp ecosystem includes the packages Rcpp, RcppArmadillo, RcppEigen, and RcppGSL.

# Rcpp Usage on CRAN



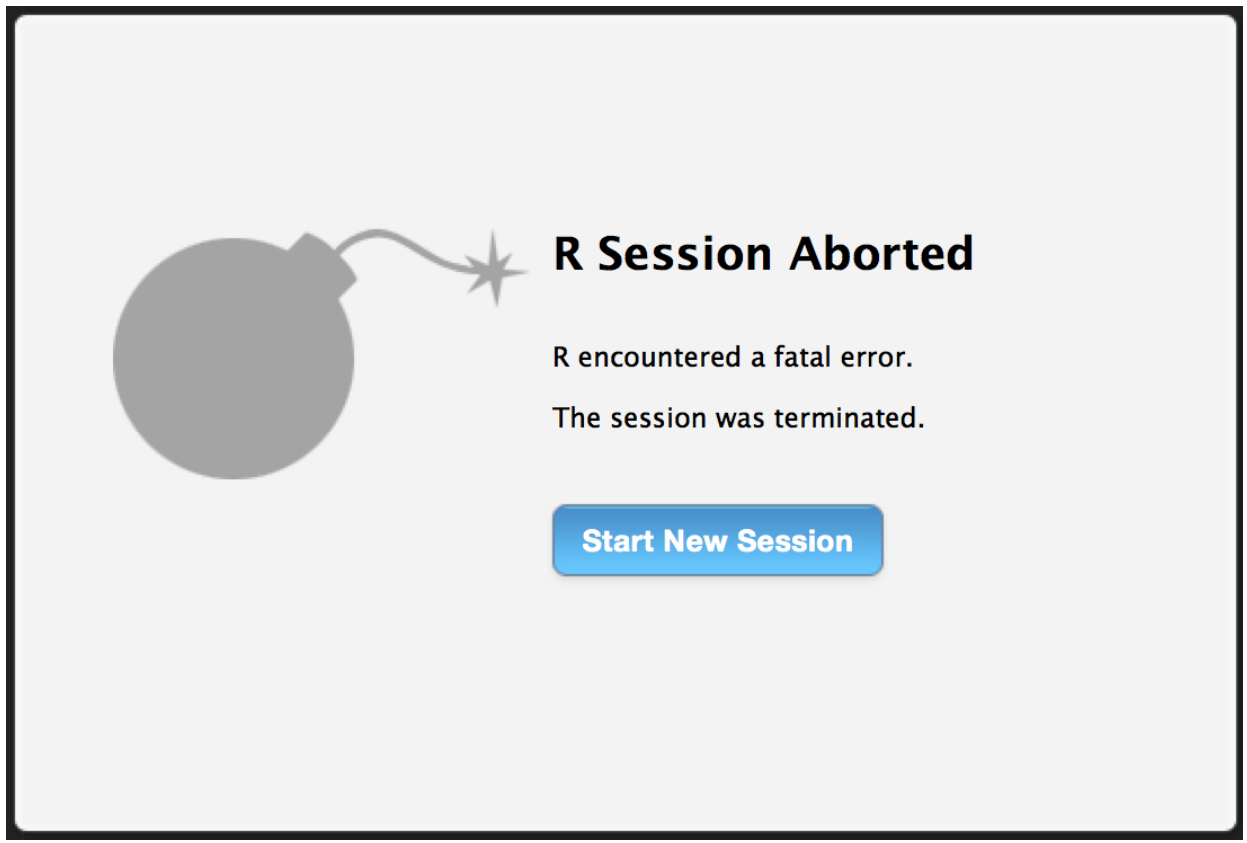

"Rcpp now used by 10 percent of CRAN packages"
Source: `http://dirk.eddelbuettel.com/blog/2017/08/23/`

# Why use Rcpp?

- Compiled code is fast. More efficient use of processors can reduce the need for parallelization.
- It is well-documented (sometimes). For example:
  1. Rcpp Gallery: articles and code examples for the Rcpp package http://gallery.rcpp.org/
  2. Rcpp vignettes: https://cran.r-project.org/web/packages/Rcpp/vignettes/
- Authors have tried to make it easy to install, learn, and use.
- Rcpp extensions are available to access to external C++ libraries.

# Potential Difficulties with Rcpp

- Many R users don't know C++. Programmers who know both will find C++ more time consuming.
- Rcpp code can be difficult to debug and profile.



# Essential Rcpp Examples

# Essential Rcpp Examples

## . . . Demonstration . . .

(See `essential.Rmd`)

# Under the Hood

# R Application Programming Interface

- R API is based on a set of functions and macros operating on S Expressions (SEXPs).
- An SEXP is a pointer to a C struct which is an internal representation of an R object.
- In C, **all R objects are SEXPs**.
- R provides several calling conventions:
  1. .C() : most basic interface intended for passing standard vectors (not including lists)
  2. .Call(): allows access to R data structures inside of C++
- Every .Call() access uses this interface pattern:

```
SEXP my_function(SEXP x, SEXP y) {
  ...
}
```

# .C Interface

## Compiling C code into a shared object library usable by R

- Write a function in C that returns void. All of the arguments for the C function should be pointers.

```
void foo(int* nin, double* x)
{
    int n = nin[0];
    for (int i = 0; i < n; i++) {
        x[i] = x[i] * x[i];
    }
}
```

- Compile that function in a UNIX shell, creating a .so file (shared object library). The compiled function will be an object in the .so file that can be accessed from R.

```
R CMD SHLIB foo.c
```

# .C Interface

## Compiling C code into a shared object library usable by R

- Load the .so in R.

```
dyn.load("foo.so")
```

- Create a wrapper function that allows you to call the C code.

```
foo <- function(x) {
    ret <- .C("foo", length(x), as.double(x))
    ret[[2]]
}
```

```
> foo(1:10)
[1]   1   4   9  16  25  36  49  64  81 100
```

# .Call Interface

## Compiling C++ code into a shared object library usable by R

```
#include <R.h>
#include <Rinternals.h>
#include <Rmath.h>

extern "C" {
    SEXP foo(SEXP sexp_x)
    {
        int n = length(sexp_x);
        double* x = REAL(sexp_x);

        SEXP sexp_y = PROTECT(allocVector(REALSXP, n));
        double* y = REAL(sexp_y);

        for (int i = 0; i < n; i++) {
            y[i] = x[i] * x[i];
        }

        UNPROTECT(1);
        return sexp_y;
    }
}
```

- Compile the function in a UNIX shell.

```
$ R CMD SHLIB foo.cpp
```

- Load the .so in R.

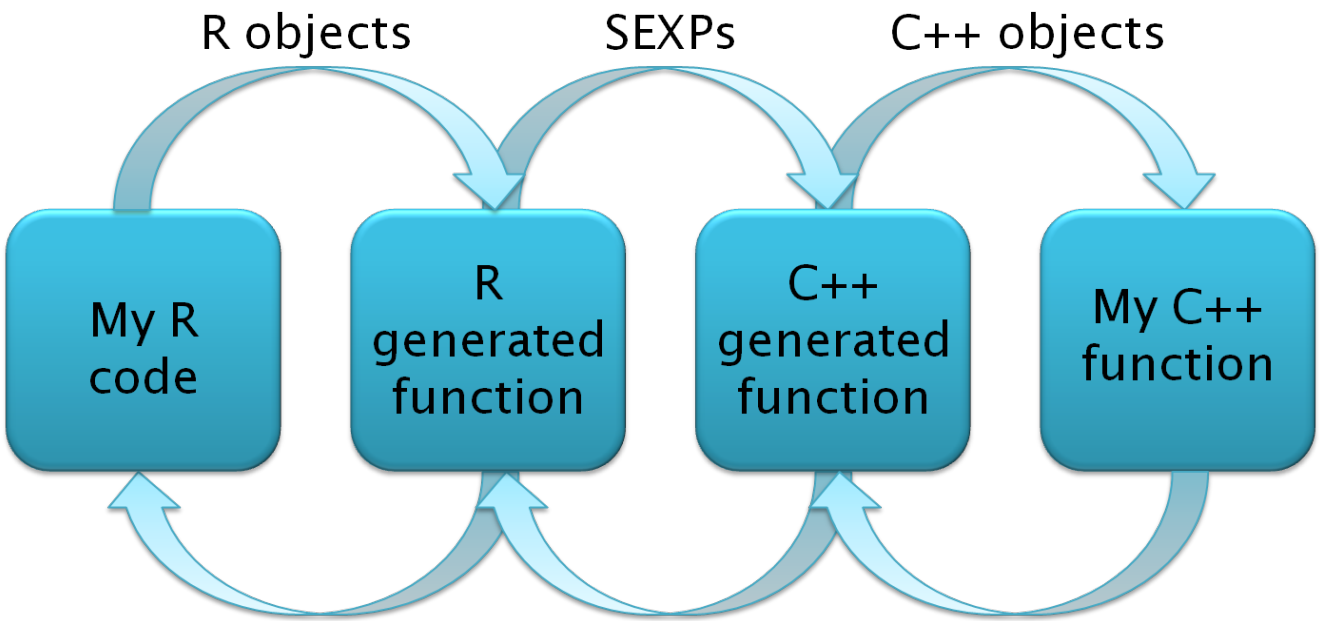
```
> dyn.load("foo.so")
```

- Create a wrapper function that allows you to call the C code.

```
foo <- function(x) {
    ret <- .Call("foo", as.numeric(x))
    return(ret)
}
```

```
> foo(1:10)
[1]   1   4   9  16  25  36  49  64  81 100
```

# A Function Call in Rcpp



# A Function Call in Rcpp

## . . . Demonstration . . .

(See `underthehood.Rmd`)

# Practical Rcpp

# Practical Rcpp

## . . . Demonstration . . .

(See `practical.Rmd`)

# Performance Considerations

# Performance Considerations

- To be an efficient R/Rcpp programmer, it helps to understand tradeoffs between:
  1. loops,
  2. the apply family (apply, lapply, sapply, mapply, vapply, etc), and
  3. matrix algebra.
- There is an art to writing good code. It should have good performance, while being as simple and readable as possible (which can be subjective).
- Loops and apply statements are very slow when computations inside are small. Either option can be more readable, depending on the situation.
- Matrix algebra in R directly uses compiled libraries (BLAS, LAPACK, etc), so it is generally very fast.
- Sometimes, loops and apply statements can be recast as matrix algebra to improve performance of R code. This technique is referred to as **vectorization**. If overused, it can lead to cryptic code, and sometimes requires much more memory than a loop/apply.
- Loops in Rcpp are fast and do not need to be avoided.
- Functions in established libraries are likely to outperform your implementation.

# Toeplitz Matrix Example

- A Toeplitz covariance matrix has the form

$$\boldsymbol{\Sigma} = \begin{pmatrix} \sigma_0 & \sigma_1 & \sigma_2 & \cdots & \sigma_{p-1} \\ \sigma_1 & \sigma_0 & \sigma_1 & \cdots & \sigma_{p-2} \\ \sigma_2 & \sigma_1 & \sigma_0 & \cdots & \sigma_{p-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sigma_{p-1} & \sigma_{p-2} & \sigma_{p-3} & \cdots & \sigma_0 \end{pmatrix}.$$

- In other words, the $(i, j)$th element of $\boldsymbol{\Sigma}$ is $\sigma_{|i-j|}$ for $i, j \in \{1, \ldots, p\}$, so that the covariance depends only on the lag.
- A special case is the autoregressive covariance AR(1)

$$\text{Cov}(Y_s, Y_t) = \sigma^2 \rho^{|t-s|}, \quad \text{Var}(Y_t) = \sigma^2$$

  for $s, t \in \{1, 2, \ldots\}$.
- Let us consider some R and Rcpp functions to form a symmetric Toeplitz matrix given its first row

$$\boldsymbol{\sigma} = (\sigma_0, \ldots, \sigma_{p-1}).$$

# Toeplitz Matrix Example

## . . . Demonstration . . .

(See `toeplitz.Rmd`)

# Bytecode Compiler

- R comes with a package for bytecode compilation called `compiler`.
- Bytecode is a low-level machine readable code. It is often faster for R to precompile human-readable R code into bytecode than to interpret human-readable code directly.
- A brief introduction is given in **?**, § 3.7.
- For details, see **?** and Tomas Kalibera's useR! 2017 talk (`https://channel9.msdn.com/Events/useR-international-R-User-conferences/`).
- R has a just-in-time (JIT) compiler which automatically pre-compiles some user code by default. We can change this behavior with the `compiler::enableJIT` function.

# Toeplitz Example with Bytecode Compiler

## . . . Demonstration . . .

(See `toeplitz-bytecode.Rmd`)

# Dimension Reduction Application

# Dimension Reduction Application

- Recall the OPG algorithm.

## Outer Product of Gradients (OPG) Algorithm

Inputs: dimension $d$ and bandwidth $h$.

**for** $j = 1$ to $n$ **do**

Compute weights $w_{ij} = \frac{K_h(\boldsymbol{x}_i - \boldsymbol{x}_0)}{\sum_{\ell=1}^{n} K_h(\boldsymbol{x}_\ell - \boldsymbol{x}_j)}$ for $i = 1, \ldots, n$.

Compute $(\hat{\alpha}_j, \hat{\gamma}_j)$ by minimizing $Q_{\boldsymbol{B}}(\alpha_j, \gamma_j)$ via WLS.

**end for**

**return** $\hat{\boldsymbol{B}}$ as matrix of the first $d$ eigenvectors of $\hat{\boldsymbol{\Sigma}} = \frac{1}{n} \sum_{j=1}^{n} \hat{\gamma}_j \hat{\gamma}_j^\top$.

- **Question**: Can we drastically improve the performance with Rcpp over a plain R implementation?

- Let us implement the computation of weight and local regression coefficients in Rcpp.

# Dimension Reduction Application

## . . . Demonstration . . .

(See `opg.Rmd`)

# Some Profiling Tools

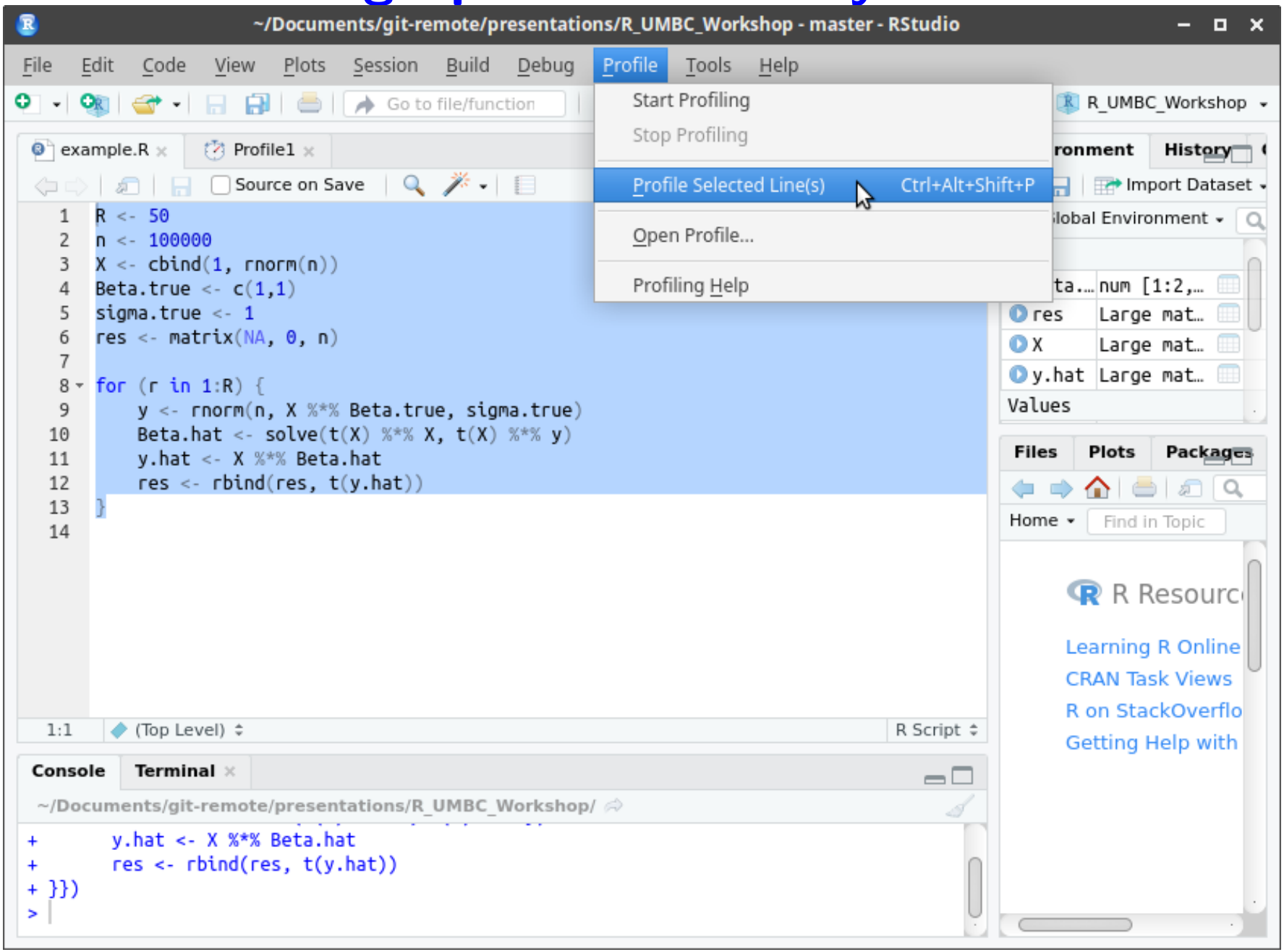
# Profiling Speed & Memory in Rstudio

- Rstudio has a built-in profiler which is based on the `profvis` package.
- Consider the following code, which is provided in `src/profiler/example.R`.

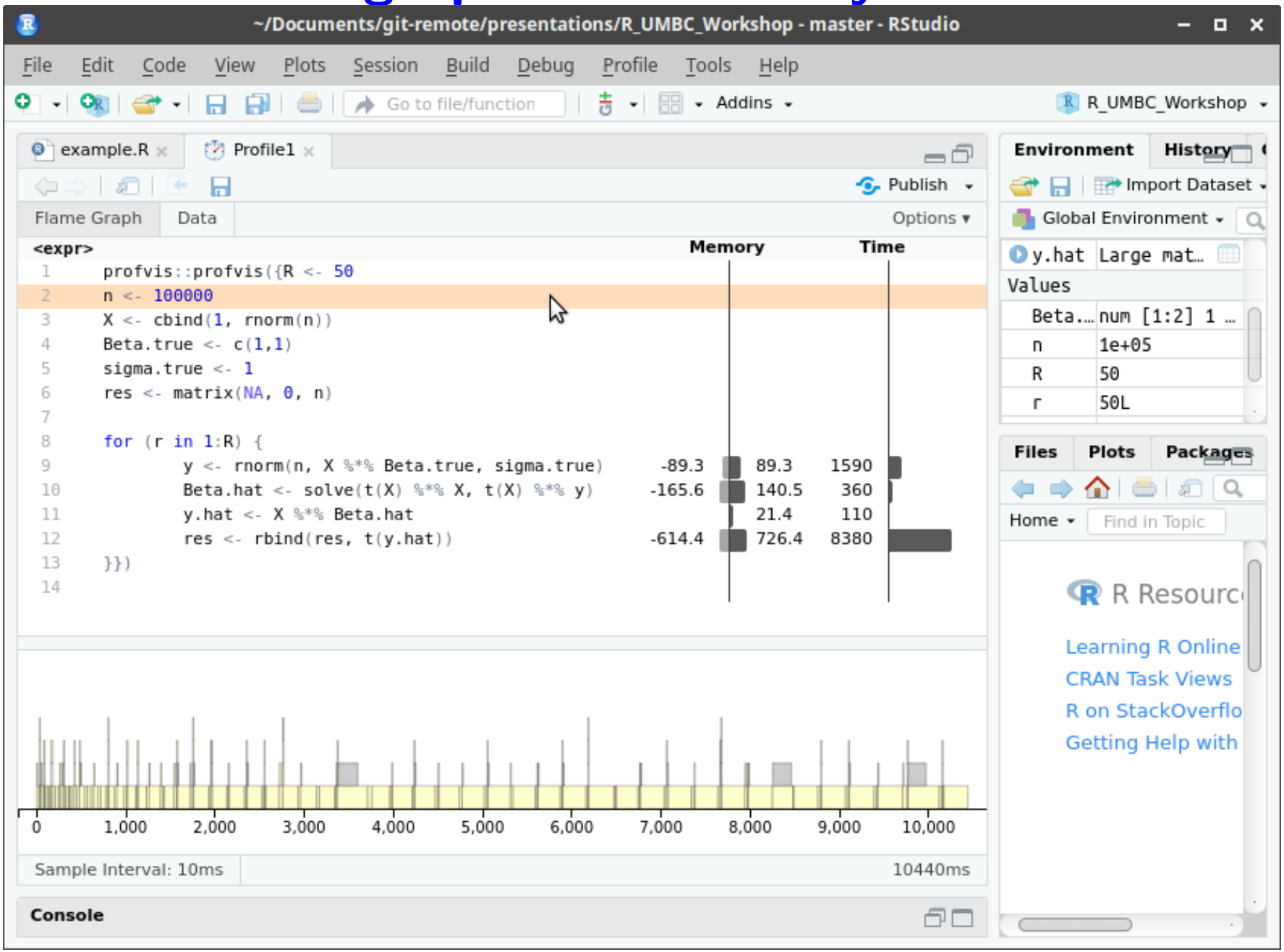
```
R <- 50
n <- 100000
X <- cbind(1, rnorm(n))
Beta.true <- c(1,1)
sigma.true <- 1
res <- matrix(NA, 0, n)

for (r in 1:R) {
    y <- rnorm(n, X %*% Beta.true, sigma.true)
    Beta.hat <- solve(t(X) %*% X, t(X) %*% y)
    y.hat <- X %*% Beta.hat
    res <- rbind(res, t(y.hat))
}
```

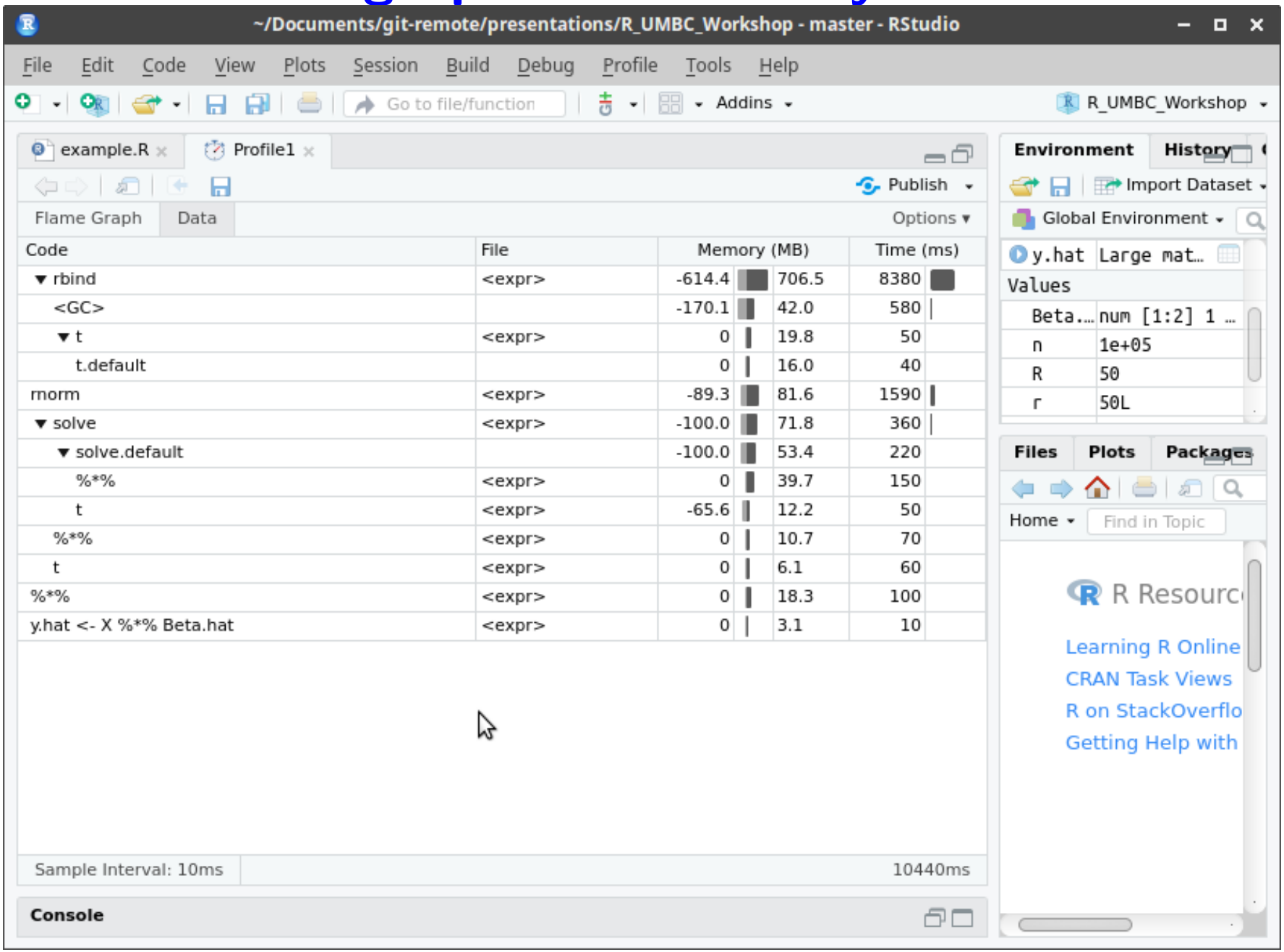
# Profiling Speed & Memory in Rstudio



# Profiling Speed & Memory in Rstudio



# Profiling Speed & Memory in Rstudio



# Profiling Speed & Memory in Rstudio

- **Exercise:** Try running the profiler on an improved version of the code, provided as `src/profiler/improved.R`.

```
R <- 50
n <- 100000
X <- cbind(1, rnorm(n))
Beta.true <- c(1,1)
sigma.true <- 1
res <- matrix(NA, R, n)

xbeta.true <- X %*% Beta.true
XtX.inv <- solve(t(X) %*% X)

for (r in 1:R) {
    y <- rnorm(n, xbeta.true, sigma.true)
    Beta.hat <- XtX.inv %*% (t(X) %*% y)
    y.hat <- X %*% Beta.hat
    res[r,] <- y.hat
}
```

# Debugging with Valgrind

- Valgrind (for Linux & Mac) provides a number of debugging and profiling tools that help you prevent memory leaks and illegal access. These can be hard to track down by yourself.
- In C / C++, whenever you dynamically request memory, you also have to explicitly free it when you are finished using it. If you do not give it back, then you have a memory leak.
- An illegal access occurs when your program tries to use memory which has not been allocated for it. The operating system will halt your program if it encroaches into other programs' space.

# Debugging with Valgrind

The ouch function tries to set a value in memory outside of allocated array. In C++, this is a bug.

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

// [[Rcpp::export]]
NumericVector ouch() {
    NumericVector x(10);
    x[1000000] = 1;
    return x;
}
```

If we try similar bad behavior in R, it covers for us by automatically extending the array.

```
> x <- numeric(10)
> x[1000000] <- 1
> head(x, 20)
 [1]  0  0  0  0  0  0  0  0  0  0 NA NA NA NA NA NA NA NA NA NA
```

http://kevinushey.github.io/blog/2015/04/05/debugging-with-valgrind/

# Debugging with Valgrind

Suppose the file `segfault.cpp` contains the `ouch` function. The following launches R with Valgrind attached.

```
$ R -d "valgrind --dsymutil=yes" -e "Rcpp::sourceCpp('segfault.cpp')"
```

Valgrind detected our illegal write.

```
> ouch()
==93684== Invalid write of size 8
==93684==    at 0x11A4C12E3: ouch() (segfault.cpp:7)
==93684==    by 0x11A4C13B7: sourceCpp_83425_ouch (segfault.cpp:23)
==93684==    by 0x10008A5E6: do_dotcall (dotcode.c:1251)
==93684==    by 0x1000B5F2B: Rf_eval (eval.c:655)
==93684==    by 0x10010C8E7: Rf_applyClosure (eval.c:1039)
==93684==    by 0x1000B6066: Rf_eval (eval.c:674)
==93684==    by 0x10011150F: do_eval (eval.c:2471)
==93684==    by 0x1000C88C8: bcEval (eval.c:5482)
==93684==    by 0x1000B5CD5: Rf_eval (eval.c:558)
==93684==    by 0x10010C8E7: Rf_applyClosure (eval.c:1039)
==93684==    by 0x1000C7509: bcEval (eval.c:5454)
==93684==    by 0x1000B5CD5: Rf_eval (eval.c:558)
==93684==  Address 0x11a9ee678 is not stack'd, malloc'd or (recently)
    free'd
==93684==
```

# Conclusions

# Other Rcpp Topics

- Rcpp can be used within R packages. Support for package development is built into Rstudio.
- Rcpp Sugar provides "syntactic sugar" to make C++ programming feel more like R programming.
- Rcpp Attributes provide a way to "tag" C++ code to inject it with certain features. For example, we have seen Rcpp::export and Rcpp::depends.
- Rcpp Modules allow the programmer to expose C++ classes to R. This provides another level of interoperability in object-oriented programs.
- RInside allows R code to be embedded in C++ programs.

# Where to go from here?

Rcpp books

- **?**, *Seamless R and C++ Integration with Rcpp*
- **?**, *Advanced R*

Rcpp online

- Rcpp website: `http://www.rcpp.org`
- Armadillo website: `http://arma.sourceforge.net`
- Ask `http://www.google.com`. E.g. "Rcpp how to make a 3d array?"
- StackOverflow: `http://stackoverflow.com`

C++ books

- **?**, *The C++ Programming Language*
- **?**, *Statistical Computing in C++ and R*

C++ online

- `http://www.cplusplus.com`
- Standard Template Library: `http://www.sgi.com/tech/stl`
- Boost: `http://www.boost.org`

# References I