High Performance Computing and Rcpp

James Balamuta

Department of Informatics, Statistics
University of Illinois at Urbana-Champaign

July 24, 2017

On the Agenda

- High Performance Computing (HPC)
  - Processors
  - Performant code
- Compiled Code with Rcpp
  - C++ and Rcpp

On the Agenda

1. High Performance Computing
   - Motivation
   - Terminology
   - Moving to HPC
2. Rcpp
   - Motivation
   - Setup
   - Compilation
   - File Naming
   - Libraries
   - C++ Code
   - Addition Example
   - Typing Issues
   - Mean Function
   - Proxy Model

Computing Quotation

"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"
— Seymore Cray

Computing Quotation - Explained

"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"
— Seymore Cray

- Lets assume for a moment that 2 oxen and 1024 chickens may provide the same amount of power.
- The question being asked is would you rather *manage* 2 operations or 1024 operations.
- In essence, 1024 would cause mass confusion as you may not be able to focus the each operation into usable power.

Power and Processors

When talking about computing, part of the **power** a computer has is given by the amount of information it can *process* within the Central Processing Unit.

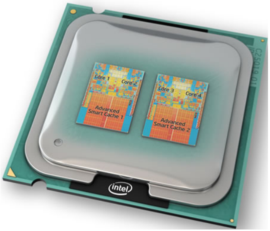
Processing Lingo

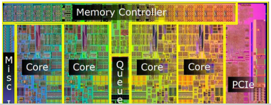
Central Processing Unit (CPU):

A CPU is the brains of the computer that takes care of a majority of the calculations.

Processing Lingo Cont.

- **Core:**
  - A core represents a single CPU.
- **Multi-core processor**
  - A multi-core processor is a single component with two or more independent CPUs (cores) on the same die or block.





Processing Lingo Cont.

Thread:

A thread is a single line of commands that are getting processed by a CPU.

Mmmm… High Performance Computing

Definition:

High Performance Computing (HPC) most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.

Source

Myths, damn myths, and. . .

There are many myths that exist around HPC. . .

Here are some of them:

- Supercomputer or clusters only need apply
- Way too expensive ($$$) to implement
- It's for Tech Firms or Academics
- Only useful for simulations
- There's no need for HPC in my field
- **This isn't available for R**

The Humor of the Situation

None of the myths are true!

In fact, we're barreling toward a future where it will be abnormal for code to run longer than 5 seconds.

Rear Admiral Grace Hopper and Nanoseconds

Figure 1: Grace Hopper on David Letterman in 1986 Video

NCSA on why #HPCMatters!

Figure 2: NCSA Video on why #HPCMatters!

Why HPC? Why now?

- Part of the reason for **HPC** is the elephant in the room: **Big Data**.
- Computers have come a long way from requiring an entire room to do simple calculations to rendering movies and videos on a skinny jean pocket size iDevice.
- Though, to fully understand the *why* part, we need to talk about **Moore's law** and how it relates to computing.

Moore's Law

> *"The complexity... has increased roughly a factor of two per year. [It] can be expected to continue... for at least 10 years"*
> *— Gordon Moore "Cramming More Components onto Integrated Circuits," Electronics, pp. 114–117, April 19, 1965*

Commonly ***mis***stated as:

> *"Computer performance doubles every 18 months"*

Video

Moore's Law - Transistors and Moores

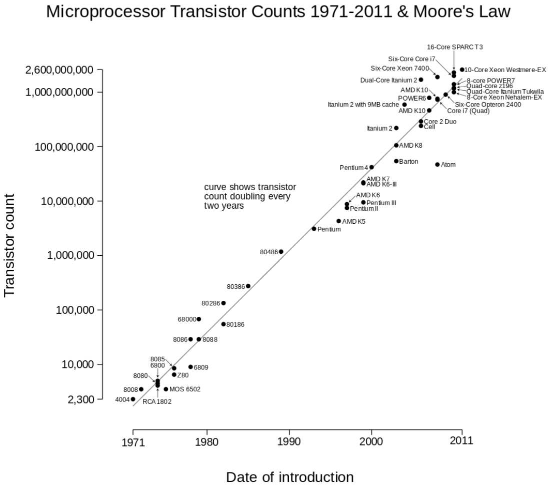


Figure 2: Source Wikipedia.org

Moore's Law - Clockspeeds of Processors

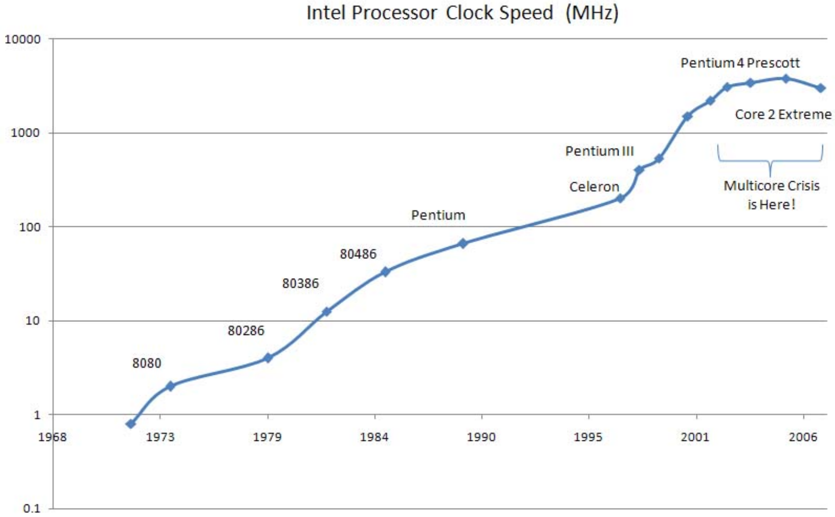


Figure 4: Source Rob Warfield

Moore's Law - Processors

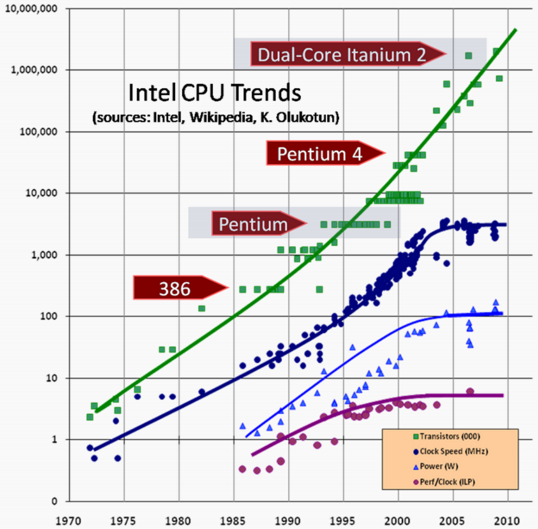


Figure 5: Source Herb Sutter

Moore's Law and Reality

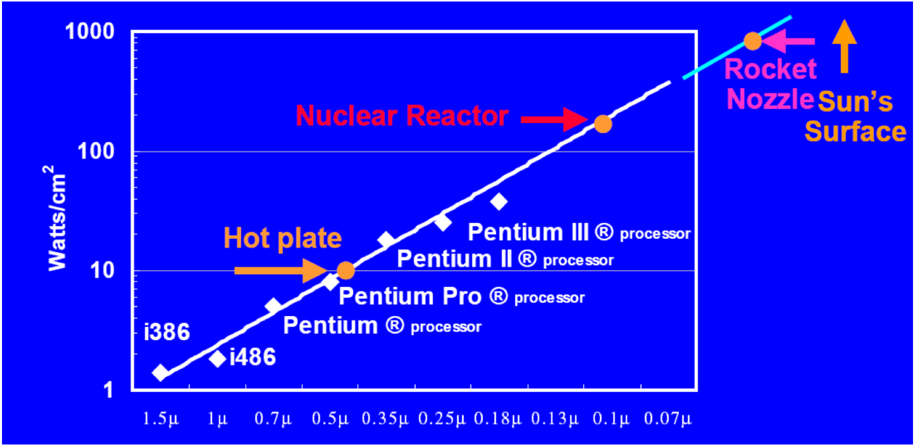


Figure 6: Mama Mia, that's a spicy meat-a-balla!

Full on Reality

- We are reaching / have reached a threshold with CPU performance.
- That is to say that we have become accustom to PCs becoming faster every year but the future holds a different kind of **change**.
- The next saga is writing high performing code, improving compilers, and using parallelization.

On the Agenda

1. High Performance Computing
   - Motivation
   - Terminology
   - Moving to HPC
2. Rcpp
   - Motivation
   - Setup
   - Compilation
   - File Naming
   - Libraries
   - C++ Code
   - Addition Example
   - Typing Issues
   - Mean Function
   - Proxy Model

R is dead, long live *R*?

- *R* is a wonderful language that allows for just about **anything**
- Each time *R* dies, it comes back stronger courtesy of community involvement.
- Every weakness gets addressed and extensions are made.

The Problem with *R*

"I like to think of R as one of the best programming languages with one of the worst 'standard libraries'"
— Kevin Ushey in Needless Inefficiencies in R – Head and Tail

- The downsides of a language for *statisticians* by *statisticians*.

R is dead, long live *R*?

The main staying power of *R* is:

- Interactively work with code
- Rapid prototyping (thanks to the interpreter)
- Syntax for statisticians by statisticians to explore data
- Top notch statistical methods
- Easy install and distribution (thanks CRAN!)
- Gateway or interface to other applications.

At the same time, R is incredibly weak in:

- Speed (downside of prototyping)
- Loops (*shudders*)
- Effective memory management (a bit aggressive on allocations)
- Multicore support (default compile of *R*'s BLAS is single core!)

Why compile code for R?

The why for compiling code is able to be reduced to:

- Speed
  - Making things go fast is fun and beneficial in this age of instanteous response.
- More libaries and tools
  - Libraries outside of the scope of R (boost, GSL, Eigen, Armadillo) are now able to be used!
- Great Support
  - There are countless answers to C++ questions and people willing to answer them! e.g. stackoverflow.com
- C++ is rock solid
  - C++ has been battle tested in industry and academia alike.

Why *Rcpp*?

Rcpp is:

- **Well Documented**
  - Rcpp Gallery has 100+ posts and over 1100 unit tests
- **Straightforward to Use (with RStudio)**
  - Press a button and code compiles! No worries about using terminal.
- **Seamless access to all R objects**
  - Move back and forth between *R* and *C++* with easy.

Rise of Rcpp

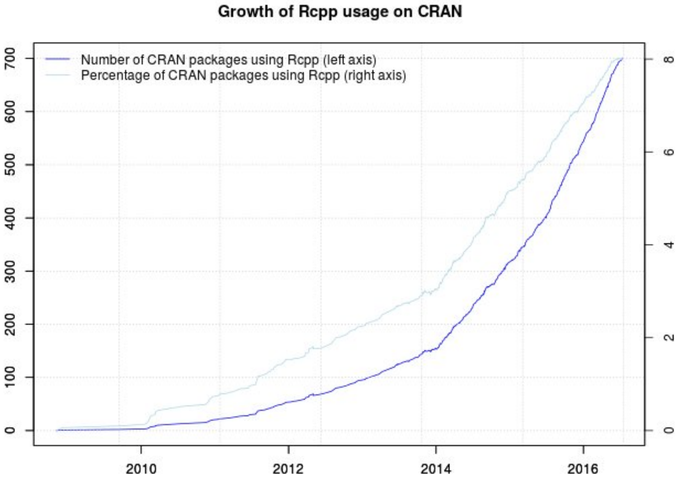


Figure 7: Dirk Eddelbuettel via Twitter

Rise of Rcpp

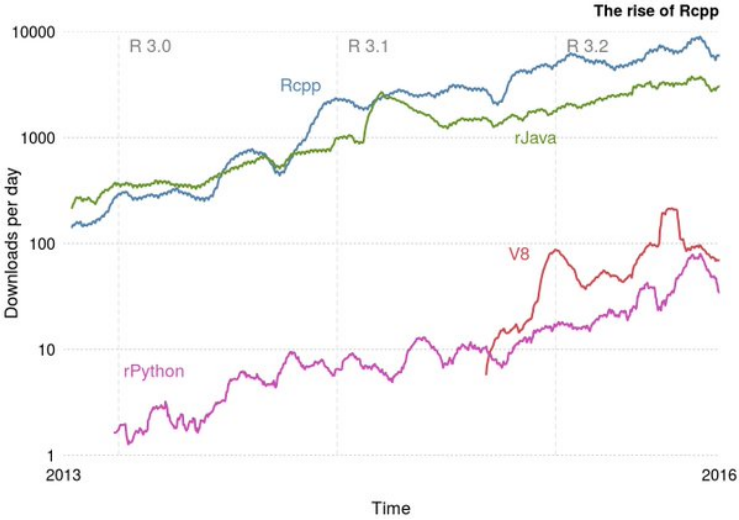


Figure 8: Colin Gillespie via Twitter

C++ Disclaimer

- C++ is a very **powerful** language.
- We're only going to briefly cover parts that relate to *non*-complex operations.
    - See CS 225 for full treatement
- E.g things that *R* is inherently bad at like loops...

Setting Up the Environment

Depending on the *type* of Operating System, there are different types of installations required.

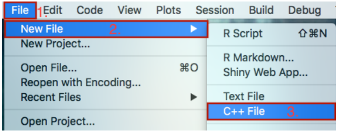
Please see the guide for each operating system:

- Windows (>= Windows 7)
- macOS (basic) **or** parallel on macOS (advanced)

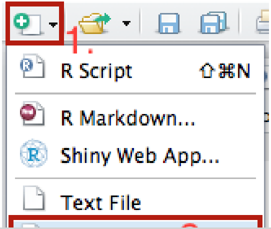
Creating C++ Files in RStudio

There are two ways to create a Cpp file within RStudio:

- **File ⇒ New File ⇒ C++ File**

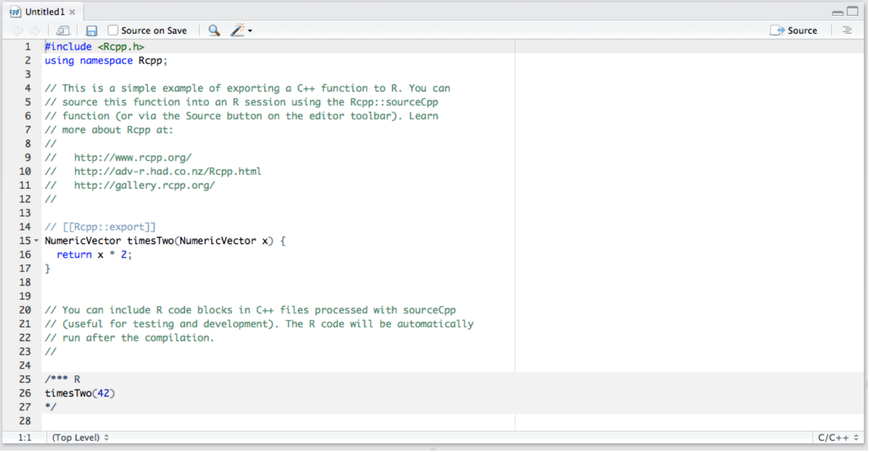


- **New Document Symbol ⇒ C++ File**



Results In. . .

- Standard Rcpp templated file with an *example*



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

// This is a simple example of exporting a C++ function to R. You can
// source this function into an R session using the Rcpp::sourceCpp
// function (or via the Source button on the editor toolbar). Learn
// more about Rcpp at:
//
//   http://www.rcpp.org/
//   http://adv-r.had.co.nz/Rcpp.html
//   http://gallery.rcpp.org/
//

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}


// You can include R code blocks in C++ files processed with sourceCpp
// (useful for testing and development). The R code will be automatically
// run after the compilation.
//

/*** R
timesTwo(42)
*/

```

- Save the file as: rcpp_twotimes.cpp

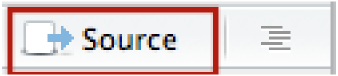
Compiling. . .

Three different ways to trigger a compilation of C++ code in a .cpp file:

- Use one of RStudio's keyboard shortcuts!
  - All OSes: `Ctrl` + `Shift` + `Enter`
- Type into console:

```
Rcpp::sourceCpp("rcpp_twotimes.cpp")
```

- Press the Source button at the top right of the editor window.



Output . . .

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

Note: This output was **automatic** after compiling the code due to the comment at the end.

Kinds of Compiling Techniques with Rcpp

There are three ways to compile code without embedding it within an R package.

1. `evalCpp()`
   - To quickly check different C++ expressions.
   - Limited scope
2. `cppFunction()`
   - For defining inline function code
   - Limit 1 function per call.
3. `sourceCpp()`
   - For code kept in an alternative file
   - Multiple files with interfunction dependence.
   - This is the **preferred** way to work with Rcpp code.

Main compilation technique

- For all intents and purposes, we will use `sourceCpp()` to compile C++ code.
- `sourceCpp()` is better as it allows for C++ syntax highlighting whereas `cppFunction()` loses the C++ syntax highlighting due to the string context.

C++ File Names

- When working with C++, avoid avoid using spaces or special symbols in either the file path or file name.
  - File Name: `example.cpp`
  - File Path: `/home/netid/example.cpp`
- If you need to use a space, use the underscore: _

Examples:

- Good: `rcpp_example.cpp`, `hello2rcpp.cpp`
- Bad: `C++ Example.cpp`, `rcpp is the bees knees.cpp`

C++ vs. R - Libraries

- To include different libraries using header (`.h`) files akin to *R*'s `library()` function, write in the C++ file:

```
#include <Rcpp.h>     // Includes the Rcpp C++ header
                      // Akin to calling library(Rcpp)
                      // in C++
```

- Unlike *R*, we also have to explicitly add the namespace of a header that we wish to use.

```
using namespace Rcpp; // C++ search scope
```

C++ vs. R - Libraries

All C++ files must therefore have at the top of them:

```
#include <Rcpp.h>      // Includes the Rcpp C++ header
using namespace Rcpp; // C++ search scope
```

C++ vs. R - Note on Namespaces of Libraries

- Specifying the namespace avoids having to prefix function calls with the `Rcpp::` namespace.
- **Note:** This is *not* a good style but it makes beginning in `C++` a bit easier.
- In particular, ambiguity is introduced into the code when two libraries provide functions with the same name (e.g. `std::sqrt` and `arma::sqrt`).
- *R* would warn when this overload happens on package load, *C++* will **not**.

C++ vs. R - Commenting in Code

Comments in C++ come in two flavors:

```
/* Group comment
 Across Multiple Lines
 */

// Single line comment
```

The use of *R*'s traditional comment:

```
# pound/hash commment
```

is used to declare preprocessor macros and, thus, should **not** be used within a `C++` file.

Steps toward a C++ Function

- Before writing a C++, try to write the function in *R* first.
- Consider an *R* function called `hello` whose goal is to print "Hello R/C++ World!"

```
hello = function(){
  cat("Hello R/C++ World!\n")
}
```

C++ vs. R - First Function in C++

We can mimic this by creating the following *C++* function.

```
// Akin to cat("Hello R/C++ World!\n")
void hello() {                               // Declaration
  Rcout << "Hello R/C++ World!\n";  // cat() in C++
}
```

Differences between *R* and *C++*:

- Return type of the function `void` is specified before function name.
- `void` indicates that no information is *returned*.
- `cat()` is done using `Rcout` with strings being delimiters of by <<.

First Function in C++ within R

In order to write a C++ function that works with *R*, you must specify the intent to export into *R* by writing directly above the function the **Rcpp Export Attribute**

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

e.g.

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

// [[Rcpp::export]]
void hello() {
  Rcout << "Hello R/C++ World!\n";
}
```

Compile it!

Pick one of the ways listed previously to compile:

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

// [[Rcpp::export]]
void hello() {
  Rcout << "Hello R/C++ World!\n";
}
```

Calling the C++ Function within *R*

C++ functions are automatically surfaced into the *R* environment by *Rcpp* under their defined name. So, in the previous example, we would have `hello()` in the global environment that we can now call.

```
# Call C++ Code like a normal R function
hello()
```

```
## Hello R/C++ World!
```

Automatically Run *R* Code on C++ Compile

To automatically test code after compile, you can embed the R code in special C++ group comment blocks like so:

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

// [[Rcpp::export]]
void hello() {
  Rcout << "Hello R/C++ World!\n";
}
/*** R
# This is R code in the C++ code file!
hello()
*/
```

C++ Function within *R*

The insides of the C++ function are slightly different

```
hello
```

```
## function ()
## invisible(.Primitive(".Call")(<pointer: 0x1079f3930>))
```

This is only problematic when you are trying to do cluster computing (more later).

Parameter Functions in *R*

- The beauty of a function is being able to slightly change variables and obtain a new input.
- To add only *two* numbers, an `add()` can be created as:

```r
add = function(a,b){
  return(a + b)
}

add(0L, 2L)    # Remember L means integer!
```

```
## [1] 2
```

```r
add(2.5, 1.1) # Double/numeric
```

```
## [1] 3.6
```

C++ vs. R - Parameter Functions

Difference between an *R* function and that of a *C++* function are as follows:

1. Return data type
2. Data type of input parameters

```
#include <Rcpp.h>
using namespace Rcpp;             // Import Statements

// [[Rcpp::export]]
double addRcpp(double a, double b) { // Declaration
  double out = a + b; // Add `a` to `b`
  return out;         // Return output
}
```

Note: Everything must be **pre-typed**.

Calling the C++ `addRcpp()` function

- You may have noticed that there should be a *typing* issue that arises.
- *Rcpp* is kind and allows for the seamless conversion of `integers` in *R* to `doubles` in C++.

```
add(0L, 2L)   # Integers into double
```

```
## [1] 2
```

```
add(2.5, 1.1) # Double into double
```

```
## [1] 3.6
```

Calling another C++ function with different types

```
#include <Rcpp.h>
using namespace Rcpp;           // Import Statements

// [[Rcpp::export]]
double addRcpp(double a, double b) { // Declaration
  double out = a + b;           // Add `a` to `b`
  return out;                   // Return output
}

// [[Rcpp::export]]
int addRcppInt(int a, int b) { // Declaration
  return addRcpp(a, b);        // Call previous function
}
```

Calling the C++ addRcpp() function with clashing types

Note, C++ will even *try* to handle the conversion between `int` and `double`.

```
addRcppInt(2.5, 1.1)  # Call in *R*
```

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

Follows *bias* rounding procedure of:

- If $x <= 0.5$, then round down: $\lfloor y \rfloor$ (floor)
- If $x > 0.5$, then round up: $\lceil y \rceil$ (ceiling)

where x is the fractional component of z and y is the integer component of z, e.g. $x = 0.3, y = 2 \Rightarrow z = 2.3$

- Why is the output *3* instead of *4*?

C++ vs. R - Functions

Data Types to choose from:

- double: 1 numeric
- int: 1 integer
- std::string: 1 character
- void: nothing
- *Vector: vector of either Integer, Numeric, or Character
- *Matrix: matrix of either Integer, Numeric, or Character

Pay Attention to Your Data Types!

R - Mean Function

Recall that

$$\bar{x}_n = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Given by:

```
muR = function(x) {
  sum_r = 0
  for (i in seq_along(x)) {
    sum_r = sum_r + x[i]
  }

  sum_r / length(x)
}
```

C++ - Mean Function

Goal: Obtain the Mean of a vector.

```
#include <Rcpp.h>
using namespace Rcpp;           // Import Statements

// [[Rcpp::export]]
double muRcpp(NumericVector x) { // Declaration

  int n = x.size();              // Find the vector length
  double sum_x = 0;              // Set up storage

  for(int i = 0; i < n; ++i) {   // For Loop in C++
    // Shorthand for sum_x = sum_x + x[i];
    sum_x += x[i];
  }
  return sum_x / n;             // Return division
}
```

Checking C++ and R Function Equality

Check the equality of both functions using `all.equal()`

```
# Done in *R*

set.seed(112)                       # Set seed

x = rnorm(10)                       # Generate data

all.equal(muRcpp(x), muR(x)) # Test Functions
```

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

Rcpp Proxy Model

- All objects using the `*Vector` or `*Matrix` tags are **proxy** objects.
- That means, they are acting as pointers to the actual memory.
- So, the call to function has *Rcpp* objects being passed by **reference**.

Rcpp Proxy Model - Example

```
#include <Rcpp.h>
using namespace Rcpp;             // Import Statements

// [[Rcpp::export]]
void ref_ex(IntegerVector x) { // Declaration
  x = x + 1; // Add 1 and save it to x via Rcpp Sugar
}
```

```
(x = 1:10)  # Span from 1 to 10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
ref_ex(x)    # No output due to no return
x            # Different Span!
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

Rcpp Proxy Model - Trouble Ho!

The proxy model works reasonably well to a degree...

```
(x = seq(0.5,5.5))  # Span from 0.5 to 5.5
```

```
## [1] 0.5 1.5 2.5 3.5 4.5 5.5
```

```
ref_ex(x)            # No output due to no return
x                    # Same span????
```

```
## [1] 0.5 1.5 2.5 3.5 4.5 5.5
```

Rcpp Proxy Model - The failures

- Note that the type of `x` is `numeric` in this case and not `integer` like before.
    - Check yourself via:

```r
typeof(x)
```

- Thus, the Rcpp proxy model **failed** to use existing memory and created a new object since it expected type `IntegerVector` but had to cast from `NumericVector`.

Lesson of the Day. . .

Pay Attention to Your Data Types!

Misc Note on Compiling Techniques with Rcpp

- When calling both the `cppFunction()` and `evalCpp()` there is a call to the `sourceCpp()` to generate the object. This is a really effective code design.
- For details see: **Rcpp Attributes Vignette** and **Code Source**