High Performance Computing and Rcpp

James Joseph Balamuta

Departments of Informatics and Statistics
University of Illinois at Urbana-Champaign

December 03, 2018

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
3. Deconstructed Hello Rcpp World
   - Compilation
   - File Naming
   - Libraries
   - C++ Code
4. Case Studies
   - Detecting Odd Numbers
   - Data Types in C++
   - Clashing Data Types
   - Implementing the Mean
5. Behind the Scenes
   - Proxy Model (Advanced)

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.

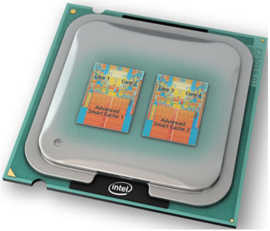
Processing Lingo

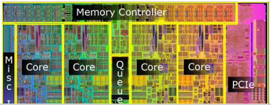
Central Processing Unit (CPU):

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

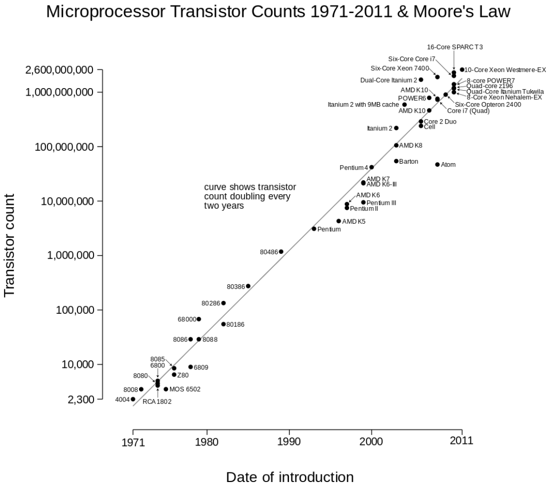


Figure 3: Source Wikipedia.org

Moore's Law - Clockspeeds of Processors

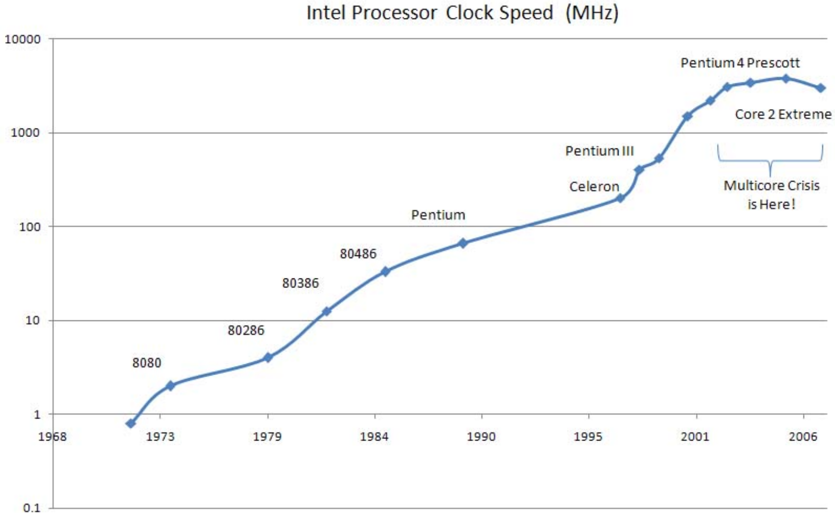


Figure 4: Source Bob Warfield

Moore’s Law - Processors

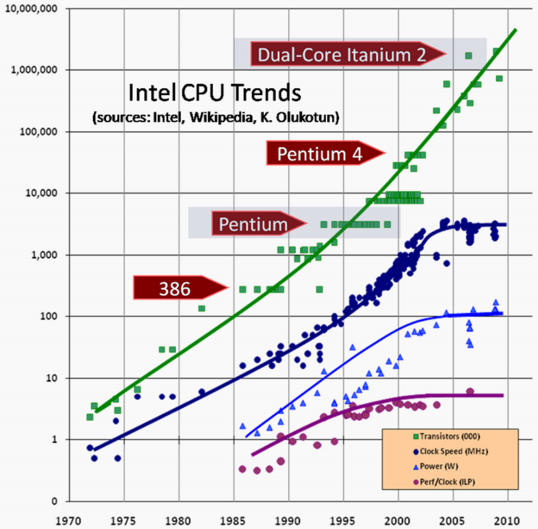


Figure 5: Source Herb Sutter

Moore’s Law and Reality

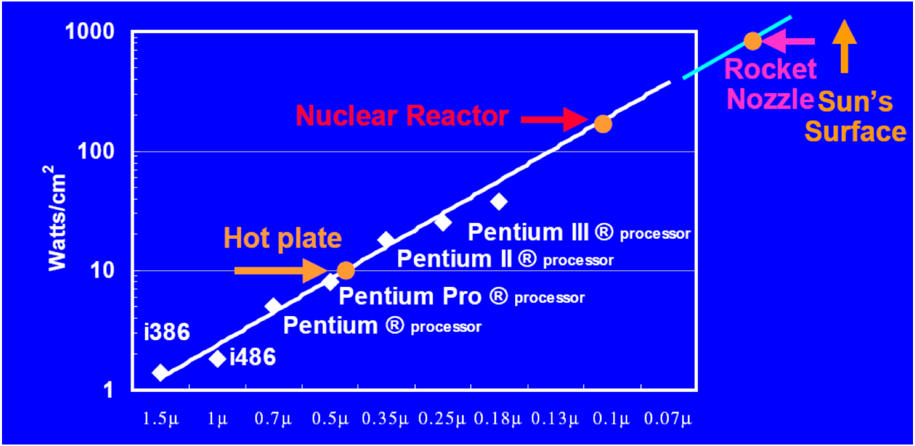


Figure 6: Mama Mia, that’s a spicy meat-a-balla!

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
3. Deconstructed Hello Rcpp World
   - Compilation
   - File Naming
   - Libraries
   - C++ Code
4. Case Studies
   - Detecting Odd Numbers
   - Data Types in C++
   - Clashing Data Types
   - Implementing the Mean
5. Behind the Scenes
   - Proxy Model (Advanced)

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, over 1100 unit tests, and an unofficial API.
- **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.
- **Supported**
  - Over 1500+ questions on StackOverflow, Rcpp-devel mailing list, and a book by one of the co-creators (**free** for students).

Rise of Rcpp

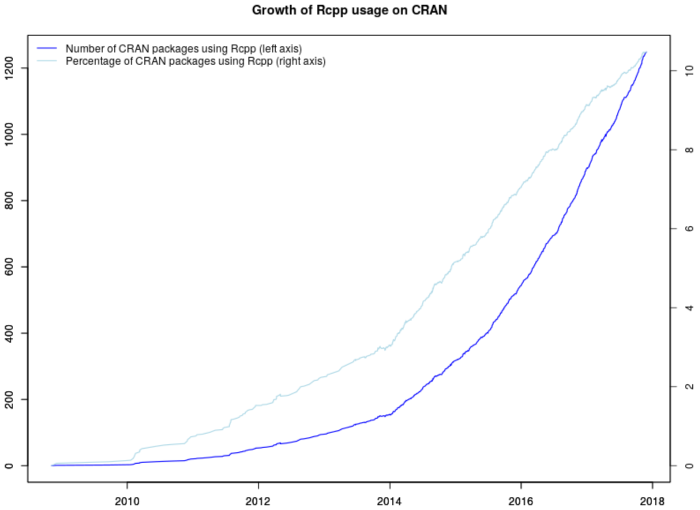


Figure 7: Dirk Eddelbuettel via Twitter

Rise of Rcpp

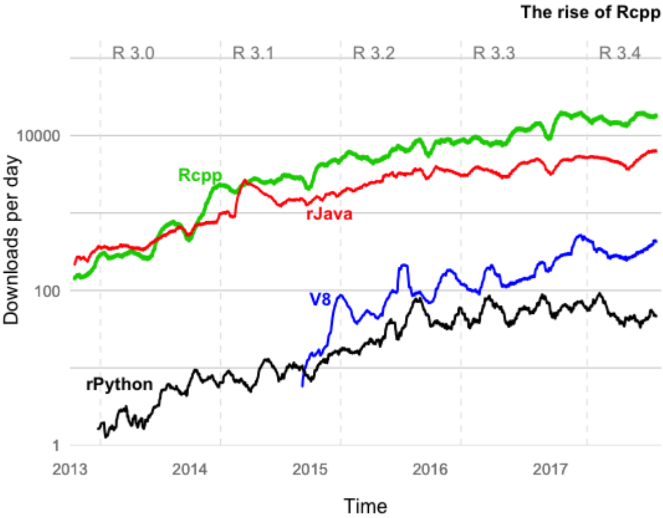


Figure 8: Colin Gillespie via Twitter

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.

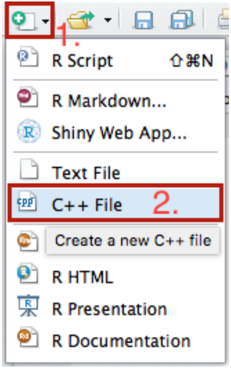
Please see the guide for each operating system:

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

RStudio Cloud's environment is pre-setup with a compiled code toolchain.

Creating C++ Files in RStudio

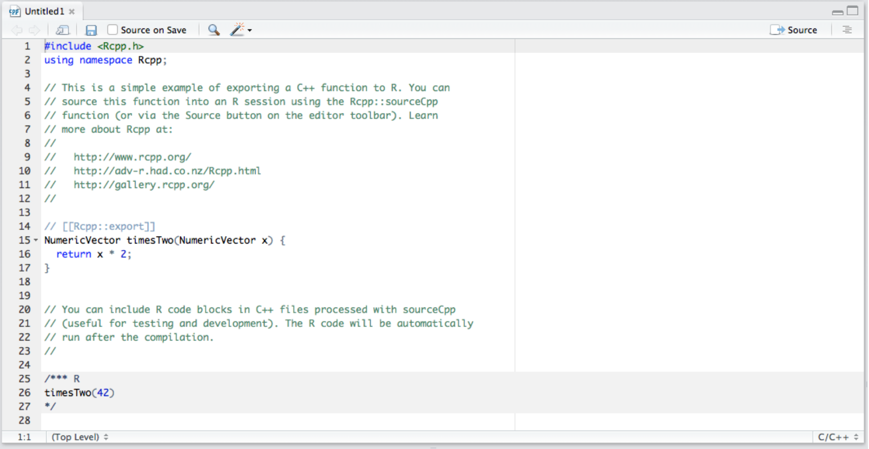
- **New Document Symbol ⇒ C++ File**



Results In. . .

Standard Rcpp templated file with an *example*.

Save the file as: `rcpp_twotimes.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)
*/

```

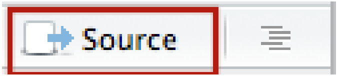
Compiling. . .

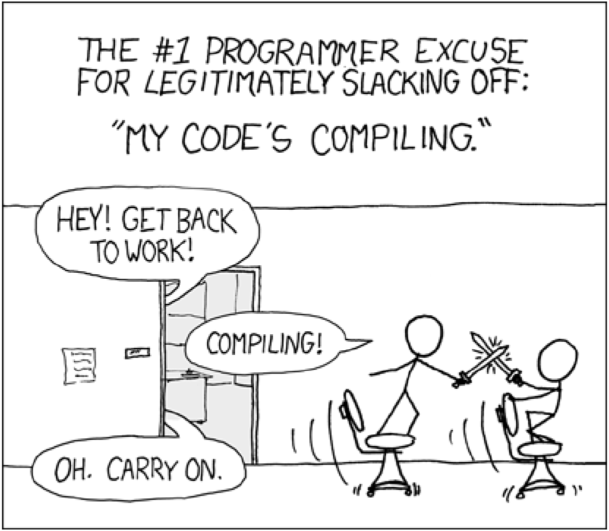
Three different ways to trigger a compilation of C++ code in a .cpp file:

- 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.





Source: XKCD 303

Output . . .

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

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

On the Agenda

1. High Performance Computing
   - Motivation
   - Terminology
   - Moving to HPC
2. Rcpp
   - Motivation
   - Setup
3. Deconstructed Hello Rcpp World
   - Compilation
   - File Naming
   - Libraries
   - C++ Code
4. Case Studies
   - Detecting Odd Numbers
   - Data Types in C++
   - Clashing Data Types
   - Implementing the Mean
5. Behind the Scenes
   - Proxy Model (Advanced)

Kinds of Compiling Techniques with Rcpp

There are three ways to compile code without embedding it within an R package.[1]

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.

[1] When calling both the `cppFunction()` and `evalCpp()` there is a call to the `sourceCpp()` to generate the object. c.f. **Rcpp Attributes Vignette** and **Code Source**

C++ File Names

- When working with C++, 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, then 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
```

Specifying the namespace avoids having to prefix function calls with the `Rcpp::` namespace.[2]

[2] 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. `dplyr::lag` vs. `stats::lag`).*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 - `hello_world()`

Let's create a function that outputs "Hello R/C++ World!"

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

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

```cpp
void hello_world_cpp() {                // 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 `<<` being the delimiter between strings.
- All statements end must end with a semicolon, e.g. `;`

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_world_cpp() {
  Rcout << "Hello R/C++ World!\n";
}
```

Compile it!

Pick one of the ways listed previously to compile:

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

// [[Rcpp::export]]
void hello_world_cpp() {
  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_world_cpp()` in the global environment that we can now call.

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

```
## 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_world_cpp() {
  Rcout << "Hello R/C++ World!\n";
}
/*** R
# This is R code in the C++ code file!
hello_world_cpp()
*/
```

C++ Function within *R*

The insides of the C++ function are slightly different

```
hello_world_r
```

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

```
hello_world_cpp
```

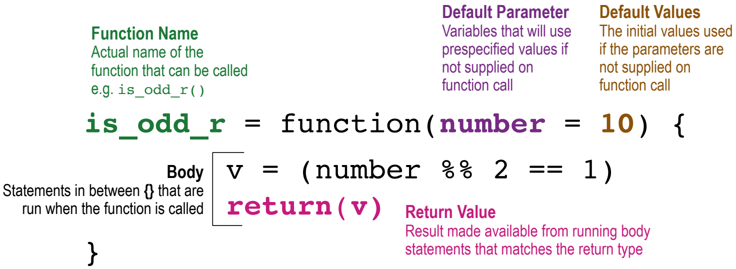
```
## function ()
## invisible(.Call(<pointer: 0x10f767320>))
```

This is only problematic when you are trying to do cluster computing.

On the Agenda

1. High Performance Computing
   - Motivation
   - Terminology
   - Moving to HPC
2. Rcpp
   - Motivation
   - Setup
3. Deconstructed Hello Rcpp World
   - Compilation
   - File Naming
   - Libraries
   - C++ Code
4. Case Studies
   - Detecting Odd Numbers
   - Data Types in C++
   - Clashing Data Types
   - Implementing the Mean
5. Behind the Scenes
   - Proxy Model (Advanced)

Steps toward a C++ Function - is_odd_r()



Function Name
Actual name of the function that can be called e.g. `is_odd_r()`

Default Parameter
Variables that will use prespecified values if not supplied on function call

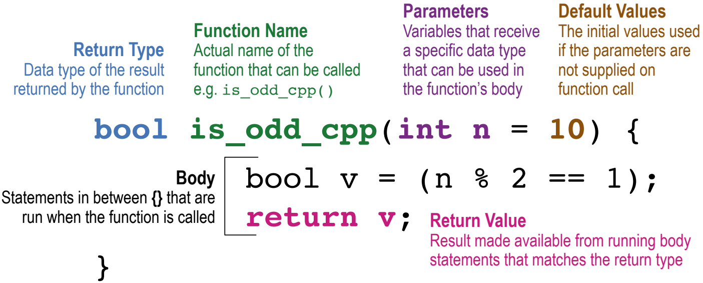
Default Values
The initial values used if the parameters are not supplied on function call

```
is_odd_r = function(number = 10) {
    v = (number %% 2 == 1)
    return(v)
}
```

Body
Statements in between **{}** that are run when the function is called

Return Value
Result made available from running body statements that matches the return type

Steps toward a C++ Function - is_odd_cpp()



Return Type
Data type of the result returned by the function

Function Name
Actual name of the function that can be called e.g. `is_odd_cpp()`

Parameters
Variables that receive a specific data type that can be used in the function's body

Default Values
The initial values used if the parameters are not supplied on function call

```
bool is_odd_cpp(int n = 10) {
    bool v = (n % 2 == 1);
    return v;
}
```

Body
Statements in between {} that are run when the function is called

Return Value
Result made available from running body statements that matches the return type

Embedding is_odd_cpp() in *R*

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

// [[Rcpp::export]]
bool is_odd_cpp(int n = 10) {// function definition
  bool v = (n % 2  == 1);    // modulus check
  return v;                  // return result
}
```

Calling is_odd_cpp() in *R*

```r
is_odd_cpp()
```

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

```r
is_odd_cpp(10)
```

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

```r
is_odd_cpp(3)
```

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

Vectorizing

int can only hold one value. What if we wanted to process a vector?

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

// [[Rcpp::export]]
LogicalVector is_odd_vec_cpp(IntegerVector x) {
  int n = x.size();            // number of elements
  LogicalVector v(n);          // logical vector
  for(int i = 0; i < n; i++) { // process each value
    v[i] = (x[i] % 2  == 1);   // modulus check
  }
  return v;
}
```

Differences

```r
is_odd_cpp(1:5)
# Error in is_odd_cpp(1:5) : Expecting a single value: [extent
```

```r
is_odd_vec_cpp(1:5)
```

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

R vs. C++ Data Types: Scalars

Scalars contain only *one* value. These are given as:

R Example	Rcpp Example	Description
`x = 3.14`	`double x = 3.14;`	A single *numeric* value with a decimal place
`x = 42L`	`int x = 42;`	A single *integer* value
`x = TRUE`	`bool x = true;`	A single *logical* value of `true`/`false`
`x = "toad"`	`std::string x = "toad";`	A single *character* value

C++ Data Types vs. *R*: Vectors/Matrices

Vector contains more than one value.

R Example	Rcpp Example	Description
`c(3.14, 4.5)`	`NumericVector x = NumericVector::create(3.14, 4.5);`	A *vector* of *numeric* values
`c(42L, -2L)`	`IntegerVector x = IntegerVector::create(42, -2);`	A *vector* of *integer* values
`c(TRUE, FALSE)`	`LogicalVector x = LogicalVector::create(true, false);`	A *vector* of *logical* values
`c("toad", "hi")`	`CharacterVector x = CharacterVector::create("toad", "hi");`	A *vector* of *character* values

To pre-allocate space use:

```
<TYPE>Vector x = <TYPE>Vector(num_observations);
NumericVector x = NumericVector(20); // Length 20 vector
```

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:

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

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

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

```
add_r(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 add_cpp(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++ `add_cpp()` 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++.

```r
add_cpp(0L, 2L)   # Integers into double
```

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

```r
add_cpp(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 add_cpp(double a, double b) { // Declaration
  double out = a + b;                // Add `a` to `b`
  return out;                        // Return output
}

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

Calling the C++ `add_cpp()` function with clashing types

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

```r
add_cpp_int(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*?

Loops

An *R* `for` loop is defined as:

```r
for (element in vector) {
  # expressions
}
```

In *C++*, `for` loops are given as:

```cpp
for (int i = 0; i < vector.size(); i++) {
  <TYPE> element = vector[i];
  // expressions
}
```

Assignment Shortcuts

R Statement	C++ Shorthand	Description
`x = x + y;`	`x += y;`	Add x to the value of y before assigning to x
`x = x - y;`	`x -= y;`	Subtract y from the value of x before assigning to x
`x = x * y;`	`x *= y;`	Multiple x by the value of y before assigning to x
`x = x / y;`	`x /= y;`	Divide x by y before assigning to x

R - Mean Function

Recall that

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

Given by:

```r
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 for a Vector

```cpp
#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()`

```r
# Done in *R*

set.seed(112)                # Set seed

x = rnorm(10)                # Generate data

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

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

Squaring Numbers

The squaring operation switches from `base^exp` to `pow(base, exp);`

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

// [[Rcpp::export]]
NumericVector square_numbers(NumericVector x) {

  int n = x.size();
  NumericVector res(n);

  for(int i = 0; i < n; i++) {
    res[i] = pow(x[i], 2.0); // x^2
  }

  return res;
}
```

On the Agenda

1. High Performance Computing
   - Motivation
   - Terminology
   - Moving to HPC
2. Rcpp
   - Motivation
   - Setup
3. Deconstructed Hello Rcpp World
   - Compilation
   - File Naming
   - Libraries
   - C++ Code
4. Case Studies
   - Detecting Odd Numbers
   - Data Types in C++
   - Clashing Data Types
   - Implementing the Mean
5. Behind the Scenes
   - Proxy Model (Advanced)

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, calling *Rcpp* functions from *R* has *R* objects being passed into *C++* by **reference**.

Rcpp Proxy Model - Example

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

// [[Rcpp::export]]
void ref_update_num(NumericVector x) { // Declaration
  x = x + 1;
}
```

```
(x = c(-8.0, 2.0))   # Numerics
```

```
## [1] -8  2
```

```
ref_update_num(x)    # No output due to no return
x                    # Different values!??!
```

```
## [1] -7  3
```

Rcpp Proxy Model - Trouble Ho!

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

// [[Rcpp::export]]
void ref_update_int(IntegerVector x) { // Declaration
  x = x + 1;
}
```

```
(x = c(-8L, 2L))
```

```
## [1] -8  2
```

```
ref_update_num(x) # No output due to no return
x                 # Same values!!!
```

```
## [1] -8  2
```

```
ref_update_int(x) # No output due to no return
x                 # Different values!!!
```

```
## [1] -7  3
```

Rcpp Proxy Model - Understanding Failure

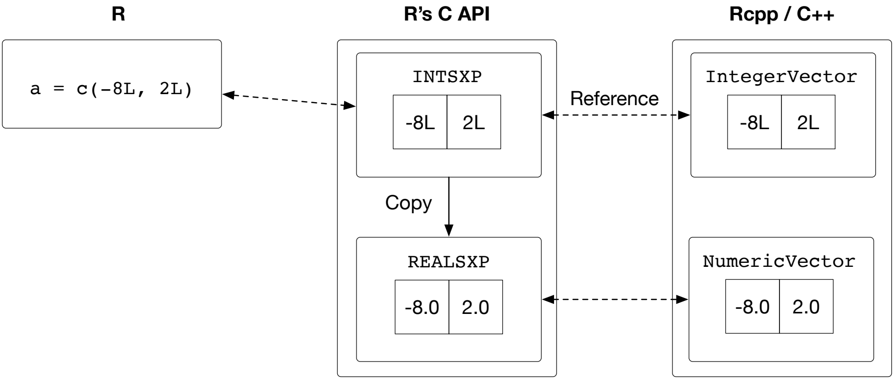
The failure occurred due to a type mismatch.

In this case, `x` is `integer` and not `numeric` like before, verify with:

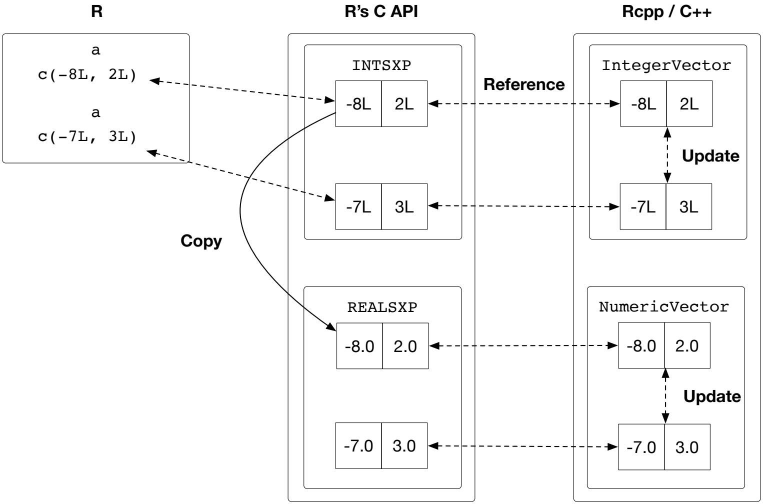
```r
typeof(x)
```

Thus, the Rcpp proxy model **failed** to use existing memory and created a new object since it expected type `NumericVector` but had to cast from `IntegerVector`. Only when we used a function that *matched* the input type did it “update” the prior object.

Proxy Memory Diagram



Proxy Domino Update



Lesson of the Day. . .

Pay Attention to Your Data Types!