

# Acelerando o R com C++



# Curso-R



**Athos Damiani**  
*Curso-R*  
*Bacharel em*  
*Estatística*



**William Amorim**  
*Curso-R*  
*Doutor em*  
*Estatística*



**Fernando Corrêa**  
*Curso-R*  
*Mestrando em*  
*Estatística*



**Julio Trecenti**  
*Curso-R, Terranova,*  
*ABJ, Conre*  
*Doutorando em*  
*Estatística*

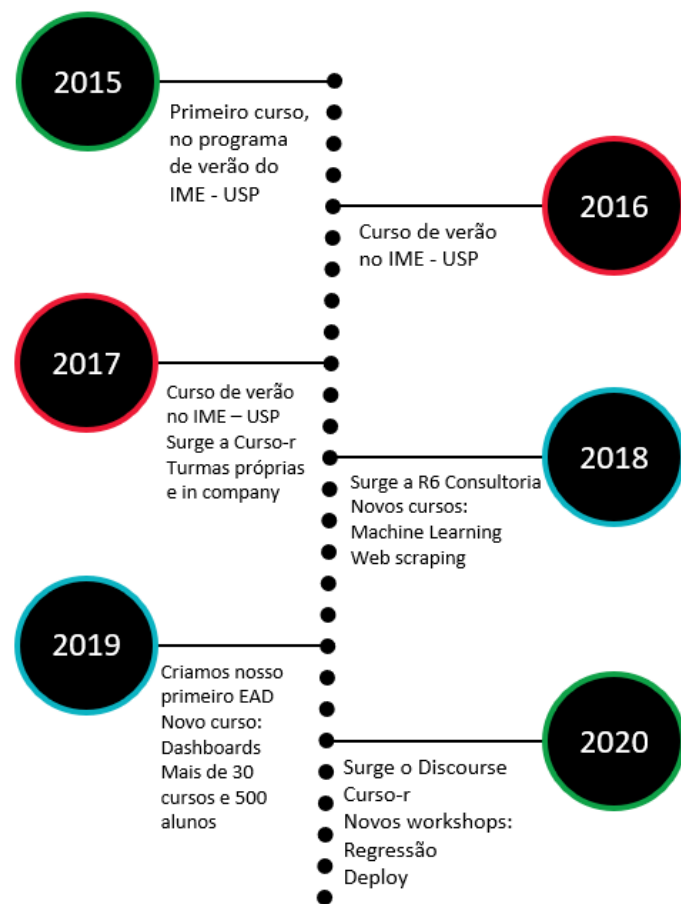


**Daniel Falbel**  
*Curso-R e RStudio*  
*Bacharel em*  
*Estatística*



**Caio Lente**  
*Curso-R, Terranova,*  
*ABJ, Mestrando em*  
*ciências da*  
*computação*

# Linha do tempo



# Informações gerais

- As aulas vão das 9h às 13, com uma pausa de 10 min em torno das 11:00
- As aulas serão gravadas e disponibilizadas no Google Classroom
- Podem mandar dúvidas no chat do Zoom ou abrir o microfone para perguntar
- Teremos bastante exercícios para resolver durante o workshop, então se prepare!

# Informações de vocês

- Nós gostaríamos de saber sobre vocês:
  - Nome
  - Com o que trabalha?
  - Como imagina usar {Rcpp} no futuro?

# Nesse curso vamos falar de

## Introdução

- Diferenças entre R e C++
- O que é {Rcpp}?
- Quando usar {Rcpp}?
- Introdução ao {Rcpp}

## Miscelânea

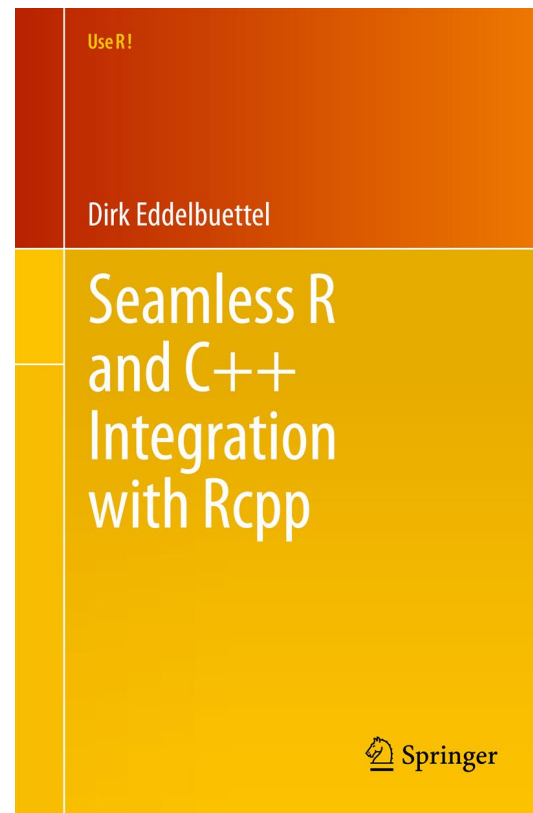
- Pacotes com código C++
- Um pouco sobre a API do R em C
- Introdução ao {cpp11}
- Paralelismo com {RcppParallel}

## Intermediário

- Usando matrizes e arrays
- Como interromper loops pelo R
- Casos de uso
- Ponteiros externos (XPtrs)

# Outros materiais

- Rcpp gallery
- Paper do Rcpp no JSS
- FAQ do Rcpp
- Referência rápida
- Capítulo do Advanced R
- Rcpp for everyone
- Curso na UseR 2020



# Diferenças entre R e C++

- As duas linguagens tem propósitos bastante diferentes. O R é uma linguagem focada em análise de dados e tem bastante desenvolvimento pensado na interatividade.
- C++ é uma linguagem de mais baixo nível e tem foco em performance e proximidade com a linguagem de máquina. É uma linguagem de propósito muito mais geral.
- Apesar de as duas linguagens possuírem muitos paradigmas diferentes. R tende a ser uma linguagem funcional: em geral escrevemos o que queremos fazer e não o 'como fazer'. C++ é imperativa, o que implica em escrever exatamente o 'como fazer'.

Dito isso, as principais diferenças que precisam ser compreendidas por um programador R aprendendo C++ são:

# Diferenças entre R e C++

## R

- **interpretada:** existe um interpretador que *parseia* o código e o executa. Esse processo ocorre toda vez que uma linha de código é executada.
- **tipada dinamicamente:** os tipos dos objetos só são verificados na hora da execução do código.

## C++

- **(pré) compilada:** o código é compilado, isto é, transformado em linguagem de máquina e depois pode ser executado. Não é necessário nenhum interpretador para executar o código.
- **estaticamente tipada:** durante o processo de compilação verifica-se se os tipos estão corretos. Por exemplo: uma função que retorna um número inteiro só pode retornar um número inteiro.



# Diferenças entre R e C++

Além de todas as diferenças conceituais, é claro, as duas linguagens também diferem bastante com relação à sintaxe.

```
hello ← function(name) {  
  print(paste("hello", name))  
}  
hello("world")
```

```
#> [1] "hello world"
```

```
#include <Rcpp.h>  
// [[Rcpp::export]]  
void hello (std::string name)  
{  
  Rcpp::Rcout <<  
    "hello " + name <<  
    std::endl;  
}
```

```
hello("world")
```

```
#> hello world
```

# Rcpp

- O R é escrito principalmente em C (não é C++) e então o R possui uma API em C que permite que você crie extensões.
- A API do R é difícil e exige que você conheça bastante detalhes da linguagem. Além disso, você precisa entender bastante como funciona o *garbage collector* para poder usar corretamente a API em C.
- Rcpp não apenas implementa uma forma de chamar funções do C++ a partir da API do R que é escrita em C, como fornece um conjunto grande de *açúcar sintático* para você não precisar entender tantos detalhes da API C do R.
- O pacote Rcpp é um dos mais utilizados no CRAN e é atualmente a principal ferramenta para criar extensões do R que utilizam C/C++

# Quando usar Rcpp?

Existem dois principais motivos para usar Rcpp:

- Você um código lento em R (geralmente envolvendo loops) que não é trivial de vetorizar. Seu objetivo então, é escrever esse código em C++ para se beneficiar da velocidade, sem necessariamente precisar mudar o algoritmo.

Essa é talvez a forma mais comum de se usar Rcpp. Você escreve seu código em R, e otimiza as partes que são *funis* de performance em C++. Pacotes como {text2vec}, {ranger}, {tm} e versões anteriores do {dplyr} usam Rcpp desta forma.

- Você deseja usar, pelo R, uma biblioteca já consolidada escrita em C++. Por exemplo, a **libmagick** é uma biblioteca escrita em C++ que possui diversas funções para manipulação de imagens - ao invés de re-escrever a sua funcionalidade em R, usamos Rcpp para *conectá-la* ao R.

Pacotes como {magick}, {hunspell}, {haven}, {opencv} e etc. usam Rcpp desta forma.

# Ambiente de desenvolvimento

## Linux

- Instalar o r-base-dev: rodar `sudo apt-get install r-base-dev`.

## Mac

- Instalar o Xcode Command Line Tools. Rodar: `xcode-select --install` no terminal.

## Windows

- Instalar o **RTools**: o RTools junta um compilador (MinGW) de código C++, um compilador de Latex e outras ferramentas úteis.

# Arquivos do tipo .cpp

O RStudio possui suporte para arquivos do tipo .cpp e vamos usá-lo como IDE.

- Você pode usar a função `Rcpp::sourceCpp()` para compilar um arquivo .cpp a partir do R.
- É possível usar a função `Rcpp::cppFunction()` para escrever os códigos em scripts .R
- O RMarkdown permite programar nas duas linguagens, além de Markdown tradicional
- Vamos usar arquivos .cpp puros com comentários que permitem programar em R

```
---  
title: "Rcpp no Rmarkdown"  
output: html_document  
---  
  
```{Rcpp}  
// [[Rcpp::export]]  
double soma (double x, double y)  
{  
  return x + y;  
}  
```  
  
```{r}  
soma(1.2, 1.8)  
```  
  
[1] 3
```

# Introdução ao Rcpp

O esquema abaixo apresenta o esqueleto de uma função C++. Preste bastante atenção na declaração dos tipos dos objetos (tanto argumentos, quanto variáveis), na especificação do tipo da saída, no ponto-e-vírgula depois de absolutamente todas as linhas e no fato de que `return` é obrigatório (apesar de nem ser uma função como no R).

| Return Type                                      | Function Name  | Parameters  | Default Values  |
|--|--|---|---|
| Data type of the result returned by the function | Actual name of the function that can be called<br>e.g. <code>is_odd_cpp()</code> | Variables that receive a specific data type that can be used in the function's body | 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<br>Statements in between {} that are run when the function is called | Return Value<br>Result made available from running body statements that matches the return type |
|---|---|

Fonte: *Extending R with C++: A Brief Introduction to Rcpp*

**Exemplo 00**

## **Exemplo 01**



## **Exercício 01**

# Vetores

Assim como o resto dos objetos do C++, vetores precisam ter seus tipos especificados no momento de criação. Felizmente, há tipos específicos que podemos usar sem ter muito trabalho. Aqui falaremos sobre `NumericVector`, `StringVector` e `LogicalVector`.

```
NumericVector v(n);  
NumericVector v(n, k);  
NumericVector v = NumericVector::create(Named("x") = 1, _["y"] = 2);
```

```
StringVector v(n);  
StringVector v(n, c);  
StringVector v = StringVector::create(Named("x") = 'a', _["y"] = 'b');
```

Redimensionar vetores não é uma tarefa fácil! Muitas vezes o jeito mais simples é copiar o vetor para um "esqueleto" vazio maior.

# Acessar elementos

Acessar elementos de vetores é tão simples quanto no R. Podemos usar tanto `[]` quanto `()`: o primeiro ignora acessos fora dos limites e o segundo retorna um erro. Ambos aceitam números (índices), strings (nomes) ou booleanos.

Atribuições funcionam exatamente da mesma forma que o R. **Note que os índices sempre começam em 0!**

```
double x1          = v[n];  
double x2          = v[c];  
NumericVector res1 = v[numeric];  
NumericVector res2 = v[integer];  
NumericVector res3 = v[character];  
NumericVector res4 = v[logical];
```

```
v[n]      = x1;  
v[c]      = x2;  
v[numeric] = v2;  
v[integer] = v2;  
v[character] = v2;  
v[logical] = v2;
```

# Atributos

Funções membras (também conhecidas como "métodos") são funções que pertencem a um objeto, ou seja, para qualquer objeto de um dado tipo, você pode chamar as suas funções membras com a sintaxe `v.f()`. Isso é comum no Python, mas apenas pacotes que usam {R6} no R têm métodos.

| Função                                  | Descrição                                 |
|---|---|
| <code>length()/size()</code>            | Comprimento do vetor                      |
| <code>names()</code>                    | Nomes do vetor                            |
| <code>fill(x)</code>                    | Preencher o vetor com valor x             |
| <code>sort()</code>                     | Ordenar o vetor                           |
| <code>begin()/end()</code>              | Iteradores para o começo e o fim do vetor |
| <code>push_back(x)/push_front(x)</code> | Colocar o valor x no fim/início do vetor  |

## Exemplo 02

## **Exercício 02**

## **Exercício 03**

# NA, NaN, Inf & NULL

Dependendo do tipo do vetor que você está usando você vai precisar usar diferentes símbolos para NA. Isso é consequência direta do fato do C++ ser forte e estaticamente tipada: não podemos criar um vetor sem declarar seu tipo e nem trocar o tipo de um vetor sem declarar a conversão.

| Vetor           | Símbolo do NA |
|-----------------|---------------|
| NumericVector   | NA_REAL       |
| IntegerVector   | NA_INTEGER    |
| LogicalVector   | NA_LOGICAL    |
| CharacterVector | NA_STRING     |

Para Inf, -Inf e NaN usamos os símbolos: R\_PosInf, R\_NegInf e R\_NaN respectivamente.

Para NULL usamos R\_NilValue.



## Exemplo 03

## **Exercício 04**

## **Exercício 05**

# Fluxo de controle

O fluxo de controle do C++ é praticamente idêntico ao do R, com algumas pequenas diferenças no for.

```
if (x > 0)
  Rcout << "positivo" << std::endl;
else if (x < 0)
  Rcout << "negativo" << std::endl;
else
  Rcout << "zero" << std::endl;
```

Chaves são necessárias se o corpo de uma condição tiver mais de uma linha, assim como no R também!

```
int n = 10;
while (n > 0) {
  Rcout << n << ", ";
  n--;
}
Rcout << "liftoff!" << std::endl;
```

```
for (int n = 10; n > 0; n--) {
  Rcout << n << ", ";
}
Rcout << "liftoff!" << std::endl;
```

## Exemplo 04

## **Exercício 06**

# Data frames

Para criar um data frame, usamos a função `DataFrame::create()`, que recebe vetores que servirão como as colunas. A sintaxe é muito parecida com a que vimos até agora para vetores, com uma diferença importante: as colunas serão **referências** aos vetores originais.

```
NumericVector v = {1,2};  
  
// Criando um data frame  
DataFrame df = DataFrame::create(Named("V1") = v,  
                                  Named("V2") = clone(v));  
  
// Alterando o vetor v  
v = v * 2;
```

No código acima, a coluna V1 será multiplicada por 2, enquanto a coluna V2 permanecerá intocada.

# Atributos

As funções membras de data frames são muito parecidas com as dos vetores, mas têm algumas distinções importantes. No geral, os métodos operam nas colunas e não nos elementos como acontecia antes com os vetores.

| Função                                  | Descrição                                     |
|---|---|
| <code>length()/size()</code>            | Número de colunas                             |
| <code>nrows()</code>                    | Número de linhas                              |
| <code>names()</code>                    | Nomes das colunas                             |
| <code>fill(x)</code>                    | Preencher todas as colunas com valor x        |
| <code>begin()/end()</code>              | Iteradores para a primeira/última coluna      |
| <code>push_back(x)/push_front(x)</code> | Colocar o vetor x no fim/início da data frame |



## Exemplo 05

## **Exercício 07**

# RcppArmadillo

{RcppArmadillo} é um pacote de álgebra linear que consegue acelerar *ainda mais* operações com vetores e matrizes. Aqui vamos dar apenas um pequeno exemplo de como ela funciona, mas existem infinitas possibilidades de como usar essa biblioteca.

```
#include <RcppArmadillo.h>
using namespace arma;

vec v;
v.subvec(from, to); // Acesso contíguo

mat M;
M.t() // Matriz transposta
M.reshape() // Redimensionar matrix
M(i, j) // Acessar elemento
inv(M) // Matriz inversa
M.submat(row_from, col_from, row_to, col_to); // Acesso contíguo
```

## Exemplo 06

## **Exercício 08**

# Chamando funções do R

É possível chamar funções do R pelo Rcpp e, para isso, usamos o tipo `Function`. Note que é necessário saber exatamente o tipo de saída retornado pela função!

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

// [[Rcpp::export]]
NumericVector f1(Function f) {
  return f(1);
}
```

```
f1(function(x) x + 100)
```

```
#> [1] 101
```

Programar em C++ é uma tarefa árdua no começo, mas extremamente recompensadora. Depois de alguns dias batalhando contra erros, sua programação vai ficar consideravelmente mais "defensiva".

```
f1(as.character)
```

```
#> Error in f1(as.character): Not compatible
```

## Exemplo 07

# Rápido mas perigoso

Já ficou provado que o C++ pode tornar um programa em R exponencialmente mais rápido, mas isso também pode trazer alguns problemas. Um deles é a dificuldade de interromper a execução de um programa! Um loop rodando em {Rcpp} pode ser impossível de parar da mesma forma que faríamos com a tecla ESC em um loop R comum.

```
void forever()  
{  
  for (int i = 0; true; i++)  
  {  
    Rcout << "Iteration: " << i << std::endl;  
    ::sleep(1);  
    Rcpp::checkUserInterrupt();  
  }  
}
```



## **Exercício 09**