R et C++ via Rcpp

D’après des supports de Timothée Tabouy & Félix Cheysson & Mathieu Carmassi, Agathe Guilloux
(modifié par Christophe Ambroise)

10 septembre 2018

Introduction

Objectif de la journée

1. Rafraîchir vos connaissances de R
2. Apprendre à produire des expériences scientifiques reproductible avec `Rmarkdown`
3. Apprendre à utiliser du `C++` pour accélérer vos codes R.

Évaluation

1. Vous comparerez plusieurs implémentations de l'algorithme des k plus proches voisins:

- votre implémentation de l'algorithme en R
- votre implémentation de l'algorithme en R et Cpp
- l'implémentation du package `caret`

1. Envoyez-moi un zip de vos répertoire de travail avec un seul fichier maitre Rmd. (Les fichiers .Rmd et .html doivent être inclus dans le dossier compressé). Tous les fichiers et dossiers compressés doivent porter le nom de votre nom. Par exemple, mon fichier correspondant à la première affectation sera appelé ambroise_knn.zip)

R et Markdown avec le package `knitr`

Installer `knitr`

Dans la console

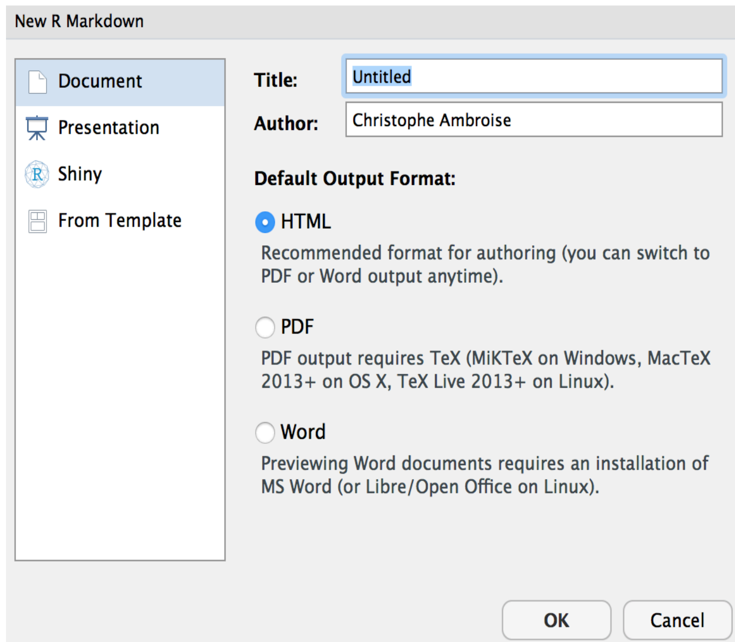
```
install.packages('knitr')
```

Par le menu

Tools -> Install Packages

Créer un de type fichier Rmarkdown (.Rmd)

Dans R Studio, cliquez sur Fichier → Nouveau fichier → R Markdown et vous obtiendrez une boîte de dialogue du type



YAML

Vous créez un fichier avec une entête dite yaml du type

```
---
title: "An Example Using the Tufte Style"
author: "John Smith"
output:
  tufte::tufte_handout: default
  tufte::tufte_html: default
---
```

qui précise comment peut être transformer le fichier

Code chunks

Les lignes dans le fichier .Rmd (Rmarkdown) suivantes

Ceci est un test d'addition:

````
```{r, eval=TRUE}
1 + 1
```
````

produisent ce type de sortie

Ceci est un test d'addition

```r
1 + 1
```

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

C'est simple mais …

Cheat sheet

Motivations: reproductibilité de la recherche

Principe de Claerbout (Géophysicien, Stanford)

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

Démarche

- Proposer une méthode, analyser des données,
- Écrire le code et le texte explicatif dans un seul fichier
- Eventuellement écrire et déposer un package sur CRAN,
- Publier un article dont les résultats pourront être reproduits par tous.

Les packages

Les packages sont des codes qui peuvent être inclus pour étendre les fonctionnalités de R

Simplicité de la création d'un package

Définir un objectif

```
Par exemple,  `SIMoNe` :  construire un graphe  des interactions
significatives entre gènes à partir de données du transcriptome.
```

Organisation type

- Fichier DESCRIPTION
- Répertoire R : fonctions R (fonction `inferGraph(data)`)
- Répertoire man : documentation des fonctions
- Répertoire data : données
- (Répertoire src : pour les fichiers à compiler, header etc.)

En bref

`knitr`

pour générer des rapports qui combinent le texte, le code et les résultats.

Markdown

pour mettre en forme le texte

Markdown est un langage de balisage léger créé par John Gruber en 2004. Son but est d'offrir une syntaxe facile à lire et à écrire. Un document balisé par Markdown peut être lu en l'état sans donner l'impression d'avoir été balisé ou formaté par des instructions particulières. — Wikipedia

[Lien Wikipedia]

Code Chunks

code dans des blocs délimités par des guillemets triples suivis de `{r}`.

Référence

Rmardown est un univers extensible, si vous voulez continuez, lisez

https://rmarkdown.rstudio.com/

Rappel sur l'apprentissage statistique

La théorie

Machine Learning Basics

La pratique

Machine Learning in R

Quelles sont les différences architecturales entre R et C ++?

Du code source au binaire

- Langage de haut niveau: le statisticien comprend et écrit "anglais".
- Langage bas niveau: le processeur lit et exécute les codes binaires (ou machine) uniquement.

!

Du code source au binaire

Nous avons besoin d'un traducteur (compilateur ou interprète) pour passer du code source au code binaire.

!

!

Compilateur versus interprète

	Compiler	Interpreter
L'entrée pr da	end le programme ns son ensemble d	**entire** et se traduit par **one déclaration à la fois**. ans le code machine.
La charge d po	e travail s'exécu ur ré-traduction.	te une seule fois et n'a besoin d'être appelée à nouveau qu'à chaque exécution du code.
Les erreurs an di	génèrent le mess alyser le program fficile. le débog	age d'erreur uniquement après la poursuite de la traduction du programme jusqu'à ce que me entier: le débogage est la première erreur est remplie, auquel cas il s'arrête: age est facile.
Partage plu OS re	s difficile à par différent, le co compilé.	tager: le binaire peut ne pas être lu par facile à partager: il suffit de transmettre le code source. de source doit être

Langage compilé: allocation de mémoire

Parce qu'au "moment de la compilation", le compilateur doit savoir exactement quels objets sont, l'utilisation des variables est moins flexible:

```
int x;        // Construction: allocate
              // memory to the object.

x = 15;       // Initialisation of
              // the object.

x = x + 1;    // Do stuff.

~x;           // Destruction: clear memory.
              // Usually implicit.
```

Unallocated memory

0100 1011	1101 1100	0010 0100	0110 0100	1000 1100	0011 0110	0101 0011	1000 1001

Construction: memory declaration

0100 1011	1101 1100	0010 0100	0110 0100	1000 1100	0011 0110	0101 0011	1000 1001

Variable initialisation

0100 1011	0000 0000	0000 0000	0000 0000	0000 1111	0011 0110	0101 0011	1000 1001

Variable manipulation

0100 1011	0000 0000	0000 0000	0000 0000	0001 0000	0011 0110	0101 0011	1000 1001

Destruction: memory unallocation

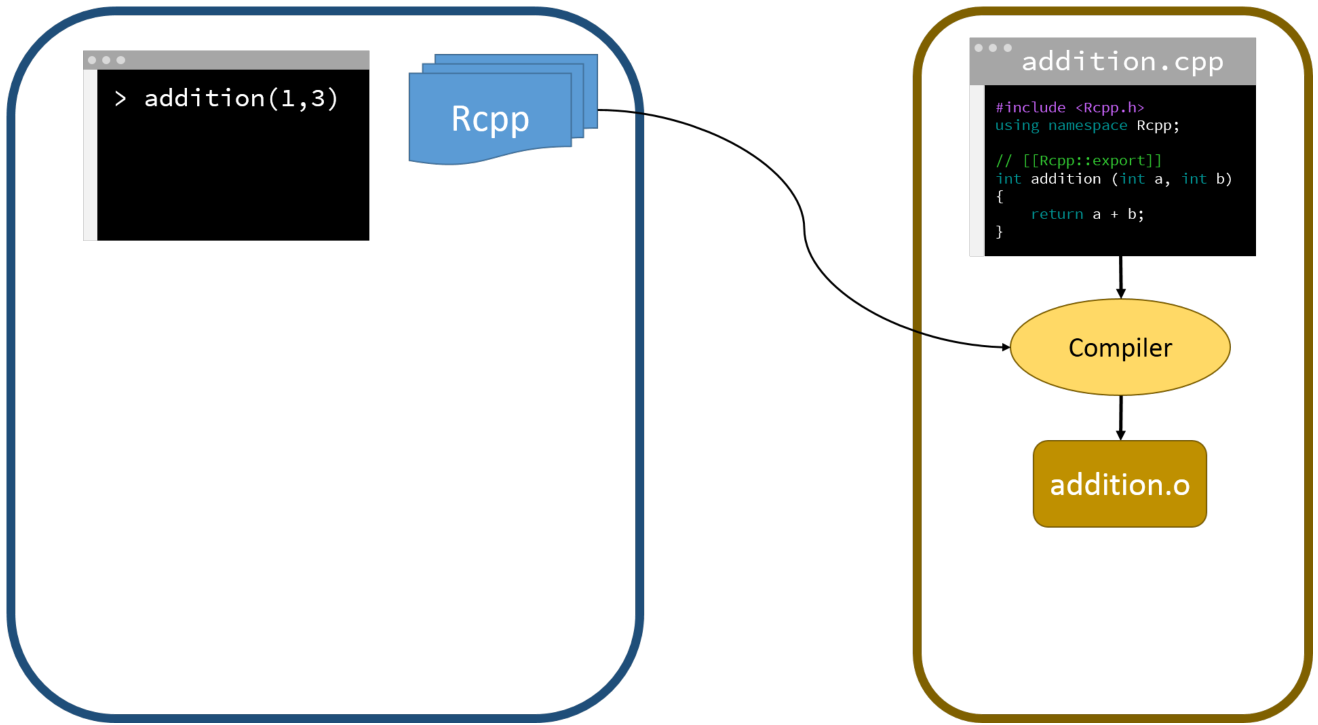
0100 1011	0000 0000	0000 0000	0000 0000	0001 0000	0011 0110	0101 0011	1000 1001

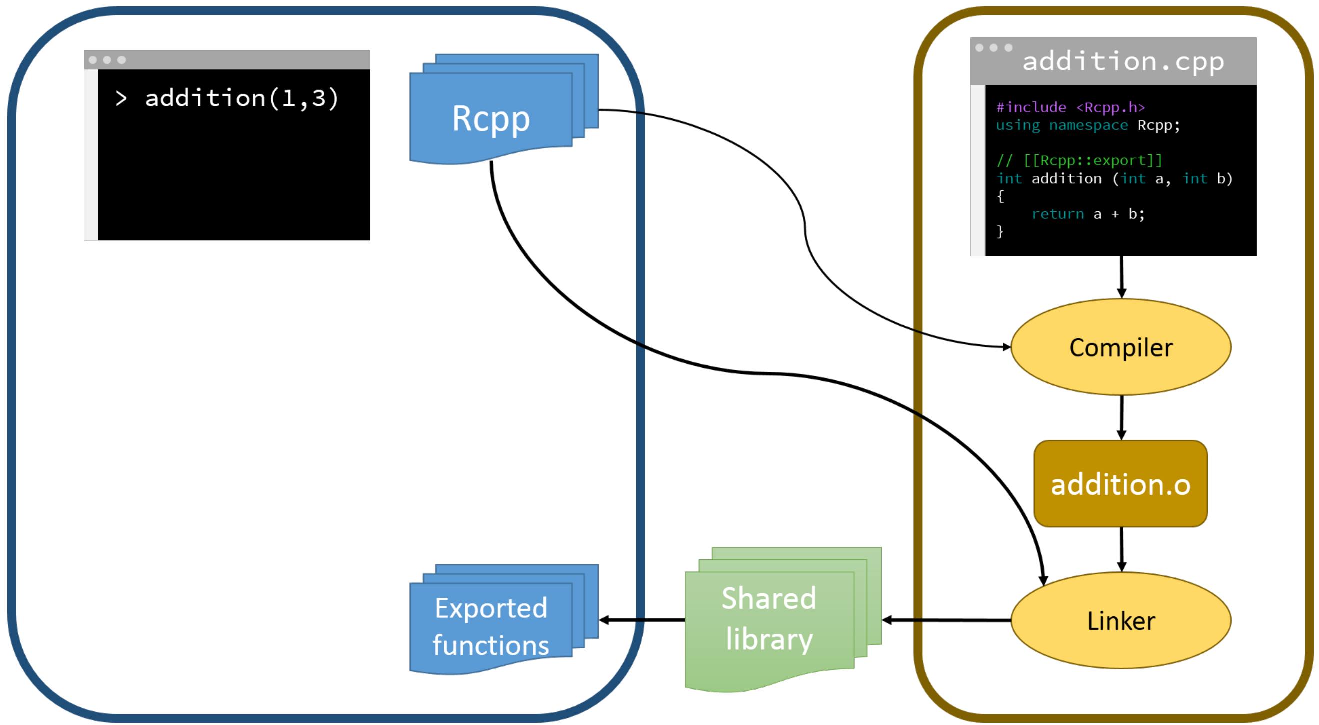
Comment Rcpp passe de C ++ à R ?

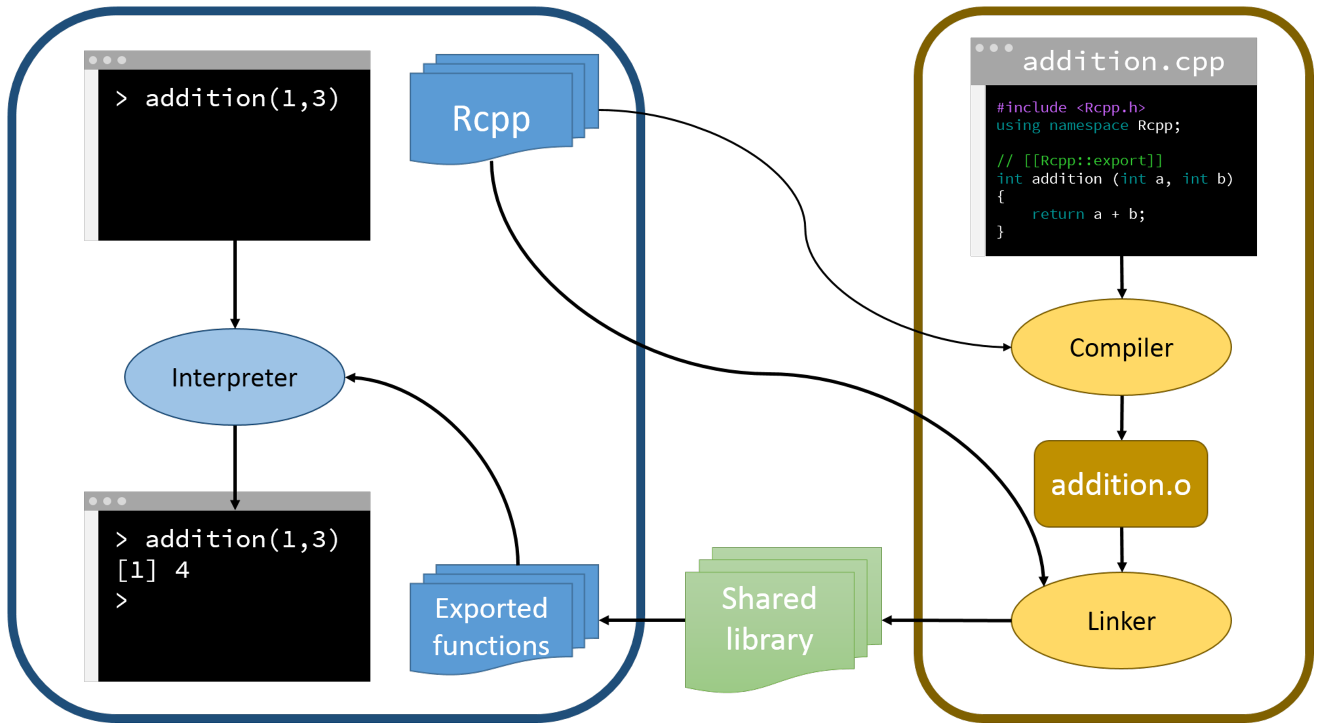
```
> addition(1,3)
```

addition.cpp

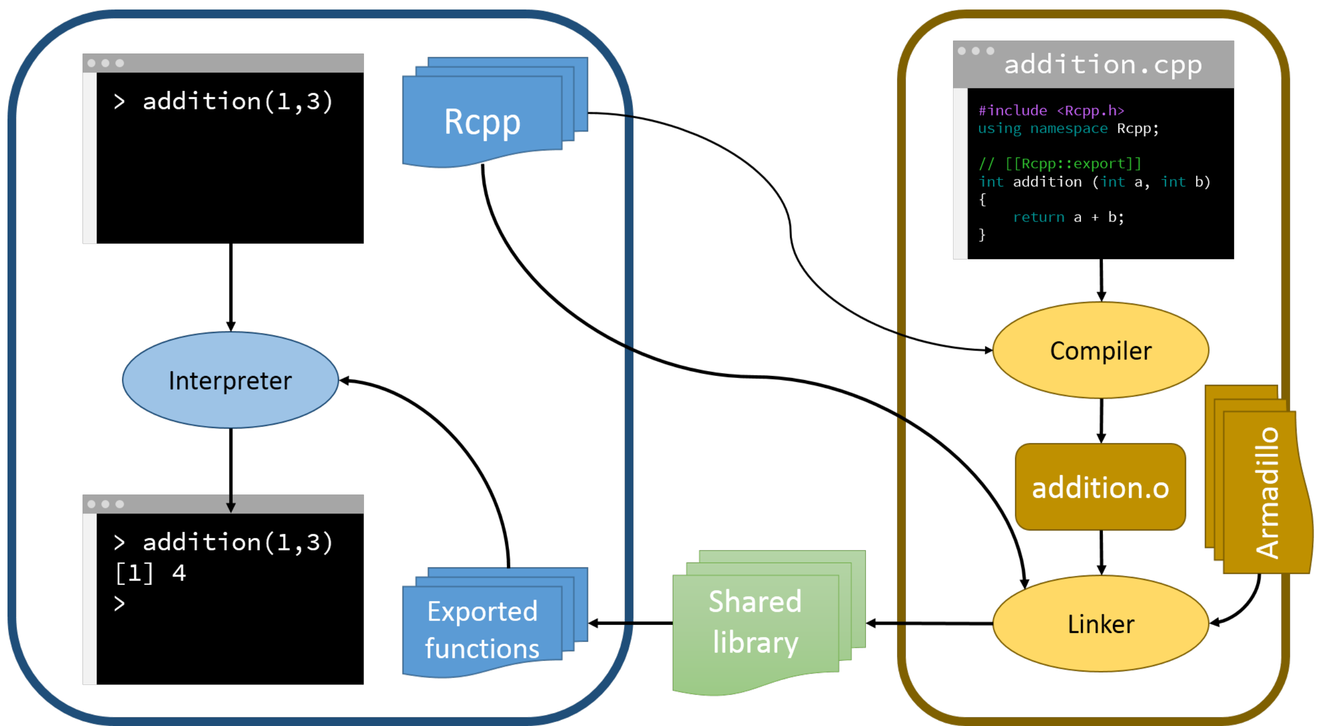
```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int addition (int a, int b)
{
    return a + b;
}
```


> addition(1,3)
Rcpp
addition.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int addition (int a, int b)
{
return a + b;
}
Compiler
addition.o

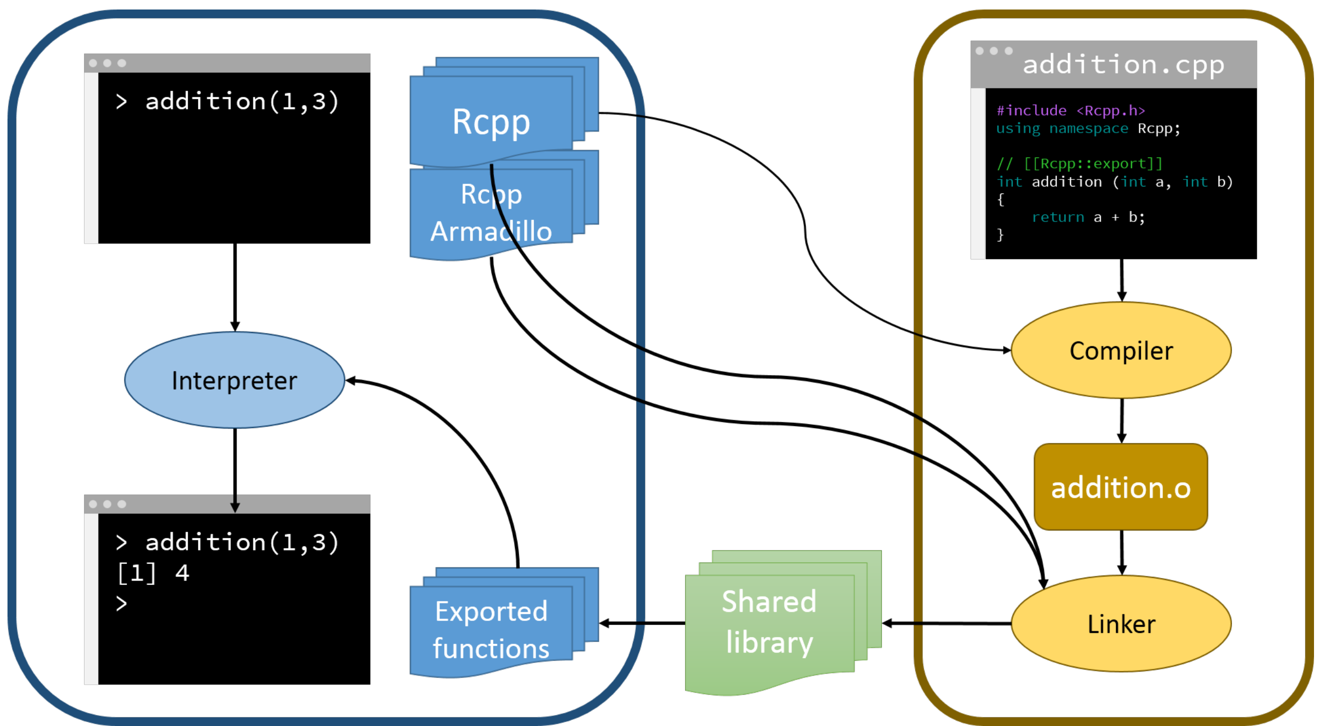

> addition(1,3)
Rcpp
addition.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int addition (int a, int b)
{
return a + b;
}
Compiler
addition.o
Linker
Shared library
Exported functions



> addition(1,3)
Rcpp
addition.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int addition (int a, int b)
{
return a + b;
}
Interpreter
Compiler
addition.o
> addition(1,3)
[1] 4
>
Exported functions
Shared library
Linker



> addition(1,3)
Rcpp
addition.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int addition (int a, int b)
{
return a + b;
}
Interpreter
Compiler
addition.o
Armadillo
> addition(1,3)
[1] 4
>
Exported functions
Shared library
Linker


> addition(1,3)
Rcpp
Rcpp Armadillo
addition.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int addition (int a, int b)
{
return a + b;
}
Interpreter
Compiler
addition.o
> addition(1,3)
[1] 4
>
Exported functions
Shared library
Linker

Principes de syntaxe

Principes de syntaxe

Exemple de code

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello, World!" << endl;
  // commentaire sur une ligne
  /*
   commentaire sur plusieurs lignes
  */
  return 0;
}
```

Principes de syntaxe

Explications

```
#include <iostream>

using namespace std;
```

- `#include` est une commande préprocesseur qui permet de charger des bibliothèques ou library (fichiers en .h) et d'utiliser les fonctions qui y sont codées. `<iostream>` pour "Standard Input / Output Streams Library".
- Un namespace, ou espace de nom (parfois aussi espace de nommage, voire référentiel lexical) est une zone de déclaration d'identificateurs permettant au compilateur de résoudre les conflits de noms.

Principes de syntaxe

Explications

```
int main()
{
  cout << "Hello, World!" << endl;
  // commentaire sur une ligne
  /*
   commentaire sur plusieurs lignes
  */
  return 0;
}
```

- `int` indique le type de la variable en sortie de la function `main`.
- `cout` et `endl` sont deux fonctions de la librarie `iostream` et permet d'écrire dans le flux de sortie (dans le terminal) la phrase "Hello, World!"

Principes de syntaxe

Explications

```
// commentaire sur une ligne
/*
 commentaire sur plusieurs lignes
*/
```

- Pour commenter une ligne il suffit d'ajouter `//` au début
- Pour commenter plusieurs lignes, il faut les entourer de `/*` et `*/`

Principes de syntaxe

Incrément / Décrément et attribution en C++

Code initial	racourci
`x = x+1;`	`x++;`
`x = x-1;`	`x--;`
`x = x + y;`	`x += y;`
`x = x - y;`	`x -= y;`
`x = x * y;`	`x *= y;`
`x = x / y;`	`x /= y;`

Les fonctions en C++

Comment créer une fonction en C++?

```
double sum(double a, double b)
{
    return a+b;
}
```

- `double` devant `sum` correspond au type de la sortie
- `sum` est le nom de la fonction
- `double a`, `double b` sont les paramètres de la fonctions (déclarés comme `double`)
- `;` à la fin des instructions (source d'erreur de compilation)

Les objets

Quels sont les différents type d'objets en C++?

Type de variables	Signification
int	entiers
float	réel simple précision
double	réel double précision
long double	réel précision étendu
char	caractère
bool	booléen
void	vide

Quels sont les différents type d'objets en C++?

Type de variables	Signification
int	entiers
float	réel simple précision
double	réel double précision
long double	réel précision étendu
char	caractère
bool	booléen
void	vide

Les entiers (1/2)

Type d'entiers	Taille	Signe	Autre nom
int	taille normale (de -32768 à 32767)	**signed** : *positif ou négatif*	**signed**
short int	taille normale (de -32768 à 32767)	**signed** : *positif ou négatif*	**signed short, short**
long int	grande taille (32 bit) (de $-2,147,483,648$ à $2,147,483,647$)	**signed** : *positif ou négatif*	**long, signed long**
long long int	grande taille (64 bit) (de $-9,223,372,036,854,775,808$ à $9,223,372,036,854,775,807$)	**signed** : *positif ou négatif*	**long long, signed long long**

Les entiers (2/2)

Type d'entiers	Taille	Signe	Autre nom
unsigned int	taille normale (de 0 à 65535)	**unsigned** : *positif ou nul*	**unsigned**
unsigned short int	taille normale (de 0 à 65535)	**unsigned** : *positif ou nul*	**unsigned short**
unsigned long int	grande taille (32 bit) (de 0 à 4, 294, 967, 295)	**signed** : *positif ou négatif*	**unsigned long**
unsigned long long int	grande taille (64 bit) (de 0 à 18, 446, 744, 073, 709, 551, 615)	**unsigned** : *positif ou nul*	**unsigned long long**

Quels sont les différents type d'objets en C++?

Type de variables	Signification
int	entiers
float	réel simple précision
double	réel double précision
long double	réel précision étendu
char	caractère
bool	booléen
void	vide

Les réels

Type de réels	Taille	Signe
float	taille normale (de $-3.4.10^{38}$ à $3.4.10^{38}$)	**signed** : *positif ou négatif*
double	grande taille (de $-1.7.10^{308}$ à $-1.7.10^{308}$)	**signed** : *positif ou négatif*
long double	grande taille (de $-1.7.10^{308}$ à $-1.7.10^{308}$)	**signed** : *positif ou négatif*

Quels sont les différents type d'objets en C++?

Type de variables	Signification
int	entiers
float	réel simple précision
double	réel double précision
long double	réel précision étendu
char	caractère
bool	booléen

Les caractères

Type de caractères	Taille	Signe
signed char	taille normale (de -128 à 127)	**signed** : *positif ou négatif*
unsigned char	grande taille (de 0 à 255)	**unsigned** : *positif ou null*

unsigned char est utile lorsque l'on travaille avec des caratères non ASCII. Si ni **signed** ou **unsigned** n'est renseigné le choix est pris par le compileur.

Quels sont les différents type d'objets en C++?

Type de variables	Signification
int	entiers
float	réel simple précision
double	réel double précision
long double	réel précision étendu
char	caractère
bool	booléen
void	vide

Problèmes de typage

Typage des variables en mathématiques

Attention :

Lors des opérations mathématiques, ne pas mélanger les types d'objet!

Par exemple si on définit `int Q` comme étant le quotient de `int A=1` et de `double B=1.5`.

De quelle forme sera le résultat? 0, 1, 0.66?

Il faudra faire attention au typage des objets à manipuler…

Les boucles

Les boucles

Trois types de boucles peuvent être écrites en C++ :

- while
- do … while
- for

Les boucles

Trois types de boucles peuvent être écrites en C++ :

- while
- do … while
- for

While

```
int compteur = 0; // Définition de la variable compteur comme un entier int

while (compteur < 10)
{
    compteur++;
}
```

La condition du `while` se situe entre parenthèse. Les instructions à répéter sont entre accolades et terminées d'un point virgule

Les boucles

Trois types de boucles peuvent être écrites en C++ :

- while
- do … while
- for

do … While

```
int compteur = 0;

do
{
    compteur++;
} while (compteur < 10);
```

Autre vision du `while`

Les boucles

Trois types de boucles peuvent être écrites en C++ :

- while
- do … while
- for

For

```
for (int compteur = 0 ; compteur < 10 ; compteur++)
{
    //intructions;
}
```

Dans les conditions du `for`, il y a 3 arguments séparés d'un point virgule :

- Initialisation (et/ou déclaration) de la variable
- La condition de fin de la boucle for
- La condition d'incrémentation de la variable

Les structures de contrôle

Les structures de contrôle

Trois structures :

- if
- if … else
- switch

Les structures de contrôle

Trois structures :

- if
- if … else
- switch

Le `if`

```
if (conditions)
    execution;
```

exemple :

```
#include<iostream>
using namespace std;

int test(int a)
{
    if (a > 10)
        cout << "a est plus grand que 10" << endl;
    return 0;
}
```

Les structures de contrôle

Trois structures :

- if
- if … else
- switch

Le `if ... else ...`

```
if (conditions) instruction1;
else instruction2;
```

exemple :

```
#include <iostream>
using namespace std;

int test(int a)
{
    if (a > 10)
        cout << "a est plus grand que 10" << endl;
    else
        cout << "a est inférieur ou égal à 10" << endl;
    return 0;
}
```

Les structures de contrôle

Trois structures :

- if
- if … else
- switch

Le `switch`

```
switch(expression)
  {
  case constante1:
      instructions1;
  case constante2:
      instructions2
  ...
  default:
      instructions_default
  }
```

exemple :

```cpp
#include <iostream>
using namespace std;

int test(int a)
{
  switch(a)
  {
  case 1 :
    cout << "a vaut 1" << endl;
    break;
  case 2 :
    cout << "a vaut 2" << endl;
    break;
  default :
    cout << "a ne vaut ni 1, ni 2"<< endl;
  break;
  }
  return 0;
}
```

La fonction somme

Exemple de fonction somme

Faire la somme d'une matrice et d'un scalaire est compliqué en C++ sans charger de librairie.

Il faudrait définir l'objet matrice et ensuite faire une somme terme à terme.

RcppArmadillo : lien entre C++ et R

Dans un fichier Cpp :

```
#include "RcppArmadillo.h"
using namespace arma;  // Evite d'écrire arma::fonctionArmadillo
// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]] // Importe la fonction qui suit dans l'environnement global de R
mat addition(int a, mat M) {
  return a + M;
}
```

RcppArmadillo : lien entre C++ et R

Dans un fichier R :

```
library(Rcpp)
```

```
## Warning: package 'Rcpp' was built under R version 3.4.4
```

```
sourceCpp("codes/addition.cpp")
a <- 1
M <- matrix(1:4,2)
sumIntMat <- addition(a,M)
```

RcppArmadillo : débugger du code C++

Ici on omet volontairement la commande namespace, il faut donc préciser à quel package appartiennent les fonctions.

```
#include "RcppArmadillo.h"
// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]
void printMatrix(arma::mat X) {
  X.print();
}
// [[Rcpp::export]]
void showMatrix(arma::mat X) {
  Rcpp::Rcout << "Armadillo matrix is" << std::endl << X << std::endl;
}
```

RcppArmadillo : débugger du code C++

On peut ajouter une balise R directement dans le C++, ceci compilera le R en "sourçant" le code C++. On peut ainsi tester directement dans le fichier C++, en utilisant R, les fonctons écrites en C++.

```
/*** R
M <- matrix(1:4,2)
printMatrix(M)
showMatrix(M)
*/
```

Documentation Armadillo

http://arma.sourceforge.net/"

TP RcppArmadillo : factorielle

Exercice :

1. Coder la fonction factorielle (récursivement et avec des boucles) en R et en C++.
2. Mesurer le temps d'éxécution des 4 fonctions pour n = 10,100,1000.

TP RcppArmadillo : correction code R

```r
factorielleR_Rec <- function(n) {
  if(n==1){
    return(1)
  } else {
    return(n*factorielleR_Rec(n-1))
  }
}
factorielleR_Loop <- function(n) {
  prod <- 1
  for (i in 1:n) {
    prod <- prod*i
  }
  prod  # Pas besoin de return ici en R
}
```

TP RcppArmadillo : correction code C++

```
double factorielleCpp_Rec(double n) {  // Le type de retour est crucial (source d'erreur)
  if(n==1) {
    return 1;  // Ne pas oublier le ";" !
  } else {
    return n*factorielleCpp_Rec(n-1); // Pas de parenthèses !
  }
}
double factorielleCpp_Loop(double n) {
  double prod = 1;  // Important : il faut initialiser les variables
  for(double i=1;i<=n;i++){
    prod *= i;
  }
  return prod;
}
```

TP RcppArmadillo : correction benchmark

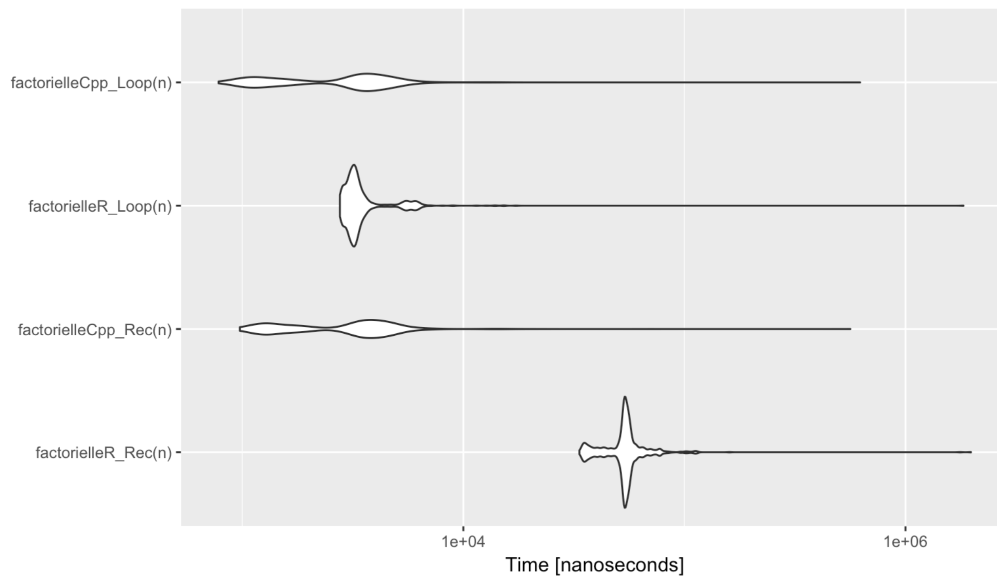
```
library(microbenchmark)
```

```
## Warning: package 'microbenchmark' was built under R version 3.4.3
```

```
library(Rcpp)
library(ggplot2)
source("codes/factorielle.R")
sourceCpp("codes/factorielle.cpp")

n <- 100
tm <- microbenchmark(factorielleR_Rec(n),
                     factorielleCpp_Rec(n),
                     factorielleR_Loop(n),
                     factorielleCpp_Loop(n), times=1000L)
```

TP RcppArmadillo : correction benchmark (plot)

```
autoplot(tm)
```



TP RcppArmadillo : vectoriser vs boucle

Exercice :

1. Écrire la quantité $\sum_{q,\ell=1}^{Q} \alpha_q \alpha_\ell \pi_{q\ell}$ sous forme vectorielle.
2. Comparer en R et en C++ le temps d'éxécution des codes (boucles et produit matriciel).

TP RcppArmadillo : correction code R

```
formeBilinR_Loop <- function(alpha, pi) {
  Q <- length(alpha)
  sum <- 0
  for(q in 1:Q){
    for (l in 1:Q) {
      sum <- sum + alpha[q]*alpha[l]*pi[q,l]
    }
  }
  sum
}
formeBilinR_Prod <- function(alpha, pi) {
  sum <- t(alpha) %*% pi %*% alpha
  sum
}
```

TP RcppArmadillo : correction code C++

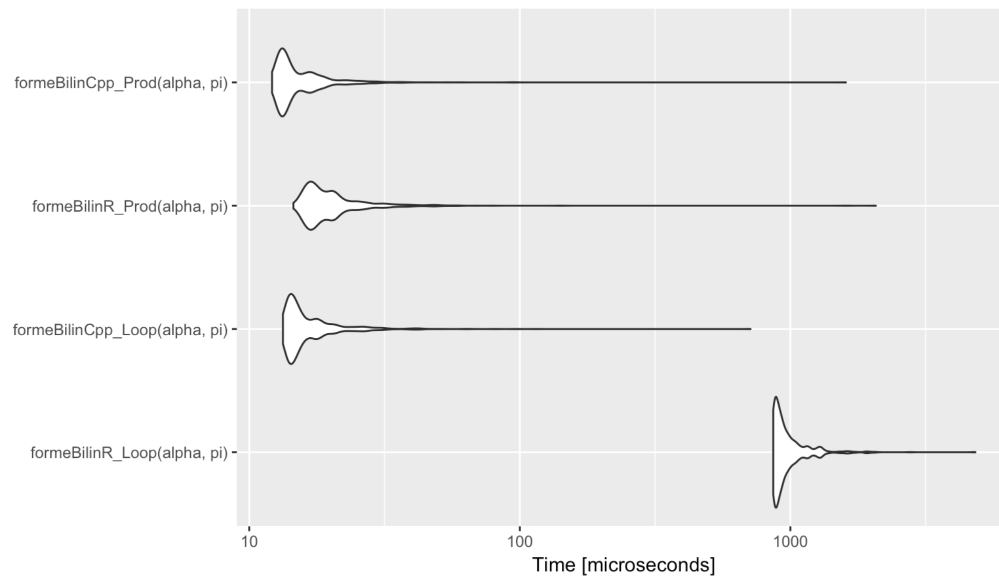
```
double formeBilinCpp_Loop(vec alpha, mat pi) {
  int Q = pi.n_cols;
  double sum = 0;
  for (int q=0; q<Q; q++) {
    for(int l=0; l<Q; l++) {
      sum += alpha[q]*alpha[l]*pi(q,l); // Attention aux parenthèses et crochets !
    }
  }
  return sum;
}
double formeBilinCpp_Prod(vec alpha, mat pi) {
  double sum = as_scalar(alpha.t() * pi * alpha);  // Attention aux types !
  return sum;
}
```

TP RcppArmadillo : correction benchmark

```
library(microbenchmark)
library(ggplot2)
source("codes/formeBilin.R")
sourceCpp("codes/formeBilin.cpp")

alpha <- 1:100
pi    <- matrix(1,100,100)

tm <- microbenchmark(formeBilinR_Loop(alpha,pi),
                     formeBilinCpp_Loop(alpha,pi),
                     formeBilinR_Prod(alpha,pi),
                     formeBilinCpp_Prod(alpha,pi), times=1000L)
```

TP RcppArmadillo : correction benchmark (plot)

```
autoplot(tm)
```



Rcpp dans un Package R

Utiliser Rcpp dans un package

Dans un package R utiliser la fonction de `devtools`

```
devtools::use_rcpp()
```

Le lancement de cette fonction :

- génère un dossier `src`
- fait les modifications nécessaires dans le fichier DESCRIPTION
- créer un fichier `.gitignore` (si le gestionnaire de version `git` est utilisé)

Dans le dossier `src`

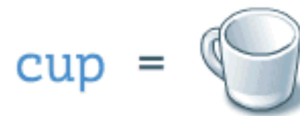
Le code C++ doit se situer dans le dossier `src`. (Fichier > Nouveau > Fichier C++)

Les parties chronophages du package doivent être implémentées dans un fichier dans lequel les fonctions utiles sont exportées par `[[Rcpp::export]]`.

La documentation de fonction Rcpp est aussi gérée par `Roxygen`

Astuces de codage: bonnes pratiques

Pass argument by reference



Pass argument by reference

```
void byValue
  (vec x, double newValue) {
  x.fill(newValue);
}

void byRef
  (vec& x, double newValue) {
  x.fill(newValue);
}

void byRefSafe
  (const vec& x, double newValue) {
  x.fill(newValue);
}
```

```
x <
x %
x

##

x %
x

##

x <
x %
```

Pass argument by reference

```
mat prodByValue (mat X, mat Y) {
  return X*Y;
}

mat prodByRef (mat& X, mat& Y) {
  return X*Y;
}
```

##

```
library(microbenchmark)
X <- matrix(rnorm(10000), 100, 100)
Y <- matrix(rnorm(10000), 100, 100)

tm <- microbenchmark(prodByValue(X, Y),
                     prodByRef(X, Y))
```

Inversion de matrices

##

```
vec invProdCpp (mat A, vec b) {
  return A.i() * b;
}

vec solveCpp (mat A, vec b) {
  return solve(A, b);
}
```

```
A <- matrix(rnorm(10000), 100, 100)
b <- rnorm(100)

tm <- microbenchmark(invProdCpp(A, b),
                     solveCpp(A, b),
                     solve(A) %*% b,
                     solve(A, b))
```

Declaration and scope

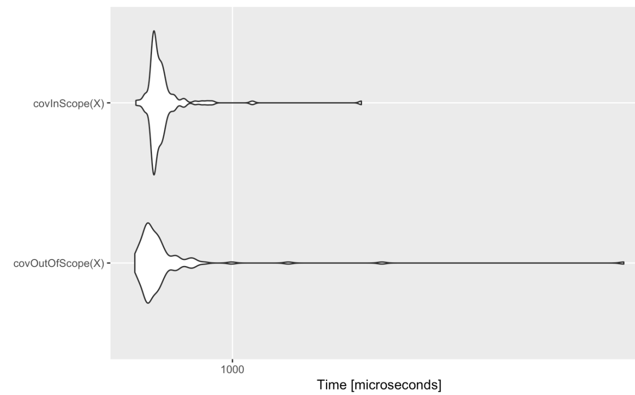
```
mat covOutOfScope (mat X)
{
  mat y = zeros(X.n_cols, X.n_cols);
  rowvec z = rowvec(X.n_cols);

  for (uword i = 0; i < X.n_rows; i++) {
    z = X.row(i);
    y += z.t() * z;
  }

  return y / X.n_rows;
}
```

```
mat
{
  m

  f



  }

  r
}
```

Declaration and scope

```
X <- matrix(rnorm(10000), 100, 100)

tm <- microbenchmark(covOutOfScope(X),
                     covInScope(X))
autoplot(tm)
```

```
## Coordinate system already present. Adding new coordinate system, which will replace the exis
```



Matrix storage of elements: column ordering

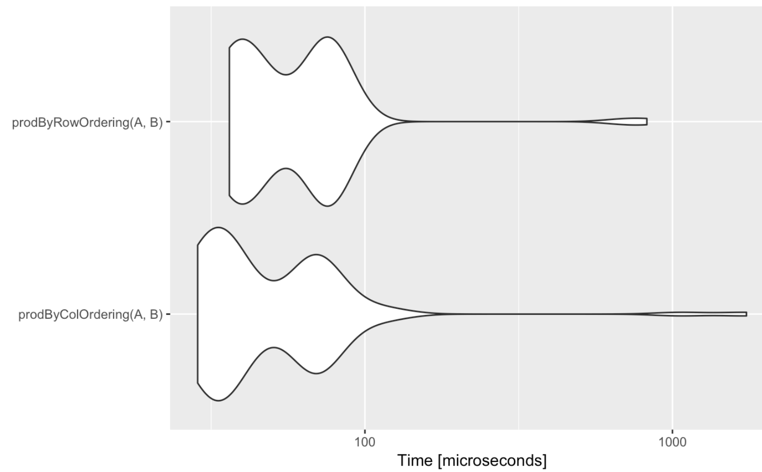
```
mat prodByRowOrdering (mat A, mat B)
{
  mat C = mat(size(A));

  for (uword i = 0; i < A.n_rows; i++)
    for (uword j = 0; j < A.n_cols; j++)
      C(i,j) = A(i,j) * B(i,j);

  return C;
}
```

```
mat
{
  m

  f



  r
}
```

Matrix storage of elements: column ordering

```
A <- matrix(rnorm(10000), 100, 100)
B <- matrix(rnorm(10000), 100, 100)

tm <- microbenchmark(prodByColOrdering(A, B), prodByRowOrdering(A, B))
autoplot(tm)
```

```
## Coordinate system already present. Adding new coordinate system, which will replace the exis
```



Code Profiling

In Rstudio it is easy to identify which part of code is worth coding in C++. This operation is called profiling and documentation are available here:

To quicky begin profiling one can select a part of code and then click on Profile > Profile Selected Lines (shortcut Ctrl+Alt+Shift+P).

The package loaded and used is profvis and similarly as microbenchmark it can be run to establish a comparison of function. It will get the time spent line by line of the code.

Code Profiling - Example

```
library(profvis)
```

```
## Warning: package 'profvis' was built under R version 3.4.3
```

```
profvis({
  data1 <- data
  # Four different ways of getting column means
  means <- apply(data1[, names(data1) != "id"], 2, mean)
  means <- colMeans(data1[, names(data1) != "id"])
  means <- lapply(data1[, names(data1) != "id"], mean)
  means <- vapply(data1[, names(data1) != "id"], mean, numeric(1))
})
```

Flame Graph | Data | Options

	<expr>	Memory		Time
1	`library(profvis)`			
2	`profvis({`			
3	`  data1 <- data`			
4	`  # Four different ways of getting column means`			
5	`  means <- apply(data1[, names(data1) != "id"], 2, mean)`	-793.5	1565.6	1800
6	`  means <- colMeans(data1[, names(data1) != "id"])`	-264.0	412.0	410
7	`  means <- lapply(data1[, names(data1) != "id"], mean)`			80
8	`  means <- vapply(data1[, names(data1) != "id"], mean, numeric(1))`			100
9	`})`			

Sample Interval: 10ms — 2390ms