Developping R packages with C++

A beginner guide

Hervé Perdry

2022-07-04

Table des matières

About this document

This document assumes that the reader is familiar with R; no previous knowledge of C++ is assumed.

The first chapter presents rapidly the main data structures used in R (vectors, matrix, factors, list, data frames), showing in particular how

The second chapter presents the very bases of C++ and RCpp. In this chapter you will use the function `Rcpp::sourceCpp` to compile C++ code from R. All the example code is available on github.

The third chapter shows how to create a R package. The resulting package can be installed from github.

After that, every chapter is associated to a package that you can install to test directly the code in it.

Les comparaisons de temps d'exécution qui apparaissent ici ont été obtenues avec une installation de R « standard » (pas de librairie comme openBlas ou autre), une compilation avec `clang++`, sur une machine linux disposant de 8 cœurs à 3.60 GHz, avec un cache de 8 MB. Des comparaisons avec d'autres compilateurs ou sur d'autres machines peuvent donner des résultats (très) différents, tant en valeur des temps d'exécution qu'en comparaison des performances.

L'idéal serait d'amener les lecteurs d'une part à une bonne connaissance des possibilités offertes par `Rcpp`, d'autre part au niveau nécessaire pour ouvrir `The C++ programming language` de Bjarne Stroustrup – on recommande, avant de se frotter à cet énorme et patibulaire ouvrage de référence (1300 pages), le plus court et plus amène `A tour of C++` du même auteur.

Chapitre 1

Objets de R

On supposera que les objets de R sont bien connus. Dans ce court chapitre nous allons simplement voir comment examiner leur structure.

1.1 R objects have types

L'instruction `typeof` permet de voir le type des objets. Considérons trois vecteurs, une matrice, une liste, un data frame, un facteur.

1.1.1 Numerical types

```
> typeof( c(1.234, 12.34, 123.4, 1234) )
[1] "double"
```

```
> typeof( runif(10) )
[1] "double"
```

```
> M <- matrix( rpois(12, 2), 4, 3)
> typeof(M)
[1] "integer"
```

```
> F <- factor( c("F", "M", "F", "F") )
> typeof(F)
[1] "integer"
```

Il y a deux types de variables numériques : `double` (nombres « à virgule », en format dit « flottant ») et `integer` (entiers). Les entiers s'obtiennent en tapant `0L`, `1L`, etc; certaines commandes renvoient des entiers:

```
> typeof(0)
[1] "double"
```

```
> typeof(0L)
[1] "integer"
```

```
> typeof(0:10)
[1] "integer"
```

```
> typeof( which(runif(5) > 0.5) )
[1] "integer"
```

```
> typeof( rpois(10, 1) )
[1] "integer"
```

On remarque que le facteur `F` a pour type `integer`. Ce petit mystère s'éclaircira bientôt.

1.1.2 Logical

We shall see later that, internally, the logical `TRUE` and `FALSE` are stored as integers `1` and `0`. They however have their proper type.

```
> typeof( c(TRUE, FALSE) )
[1] "logical"
```

1.1.3 Lists

Data frame are lists. This will be clarified soon.

```
> L <- list(a = runif(10), b = "dada")
> typeof(L)
[1] "list"
```

```
> D <- data.frame(x = 1:10, y = letters[1:10])
> typeof(D)
[1] "list"
```

1.1.4 A glimpse on the objects type

Pour examiner le contenu d'un objet avec une information sur son type, on peut utiliser `str`.

```
> str(M)
 int [1:4, 1:3] 1 2 1 2 4 4 1 1 4 4 ...
```

```
> str(F)
 Factor w/ 2 levels "F","M": 1 2 1 1
```

```
> str(L)
List of 2
 $ a: num [1:10] 0.235 0.515 0.59 0.831 0.649 ...
 $ b: chr "dada"
```

```
> str(D)
'data.frame':   10 obs. of  2 variables:
 $ x: int  1 2 3 4 5 6 7 8 9 10
 $ y: chr  "a" "b" "c" "d" ...
```

1.2 R objects have attributes.

Les objets de R ont des « attributs ». Ainsi donner des noms aux éléments d'un vecteur revient à lui donner un attribut `names`.

```
> c <- runif(4)
> names(c) <- c("elt1", "elt2", "elt3", "elt4")
> c
      elt1       elt2       elt3       elt4
0.64943651 0.28351013 0.09413375 0.92942244
```

```
> attributes(c)
$names
[1] "elt1" "elt2" "elt3" "elt4"
```

Ce qui différentie une matrice d'un vecteur, c'est l'attribut `dim`:

```
> attributes(M)
$dim
[1] 4 3
```

Les data frames et les facteurs ont également des attributs :

```
> attributes(D)
$names
[1] "x" "y"

$class
[1] "data.frame"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
> attributes(F)
$levels
[1] "F" "M"

$class
[1] "factor"
```

Les attributs peuvent être modifiés avec la syntaxe `attributes(x) <- ...` ou un individuellement avec `attr(x, which)` :

```
> attr(M, "dim")
[1] 4 3
```

```
> attr(M, "dim") <- c(2L, 6L)
> M
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    4    1    4    0
[2,]    2    2    4    1    4    2
```

1.3 How to look further at the objects' structure

La fonction `dput` permet d'obtenir une forme qui peut être copiée dans une autre session R ; ceci permet parfois d'obtenir des informations plus précises sur la représentation interne d'un objet. Nous allons l'utiliser ici pour mieux comprendre la construction des matrices, des data frames, et des facteurs.

Il est nécessaire de jeter au préalable un œil à l'aide de `structure` pour mieux comprendre le résultat. On y précise notamment :

```
For historical reasons (these names are used when deparsing),
attributes '".Dim"', '".Dimnames"', '".Names"', '".Tsp"' and
'".Label"' are renamed to '"dim"', '"dimnames"', '"names"',
'"tsp"' and '"levels"'.
```

1.3.1 Matrices are vectors

```
> dput(M)
structure(c(1L, 2L, 1L, 2L, 4L, 4L, 1L, 1L, 4L, 4L, 0L, 2L), .Dim = c(2L,
6L))
```

Une matrice est un vecteur muni d'un attribut `dim` (qui apparaît comme `.Dim` dans le résultat de `dput`).

1.3.2 Data Frame are lists

```
> dput(D)
structure(list(x = 1:10, y = c("a", "b", "c", "d", "e", "f",
"g", "h", "i", "j")), class = "data.frame", row.names = c(NA,
-10L))
```

Un data frame est une liste munie d'un attribut `class = "data.frame"` et d'un attribut `row.names` (ici, la valeur de cet attribut est la convention pour « 4 lignes non nommées »).

1.3.3 Factors are integer vectors

```
> dput(F)
structure(c(1L, 2L, 1L, 1L), .Label = c("F", "M"), class = "factor")
```

```
> levels(F)
[1] "F" "M"
```

Un facteur est qu'un vecteur d'entiers muni d'attributs `class = "factor"`. et `levels` (les niveaux du facteur), qui apparaît dans `structure` sous le nom `.Label` ; cet attribut est également accessible via la fonction `levels`.

On peut par exemple fabriquer un facteur à partir d'un vecteur d'entiers, ainsi :

```
> G <- c(2L, 1L, 1L, 2L)
> attributes(G) <- list(levels = c("L1", "L2"), class = "factor")
> G
[1] L2 L1 L1 L2
Levels: L1 L2
```

1.4 Pour les apprentis sorciers

La fonction interne `inspect` permet de voir l'adresse où se trouve l'objet, son type (d'abord codé numériquement, par exemple 13 pour `integer` puis le nom conventionnel de ce type, `INTSXP`), et quelques autres informations ; les objets complexes (leurs attributs) sont déroulés.

```
> .Internal(inspect( 1:10 ))
@55d4c67e3388 13 INTSXP g0c0 [REF(65535)]  1 : 10 (compact)
```

```
> .Internal(inspect( c(0.1, 0.2) ))
@55d4c76a2598 14 REALSXP g0c2 [] (len=2, tl=0) 0.1,0.2
```

```
> a <- c(0.1, 0.2)
> .Internal(inspect( a ))
@55d4c7708778 14 REALSXP g0c2 [REF(2)] (len=2, tl=0) 0.1,0.2
```

```
> names(a) <- c("A", "B")
> .Internal(inspect( M ))
@55d4c75d0428 13 INTSXP g0c4 [MARK,REF(5),ATT] (len=12, tl=0) 1,2,1,2,4,...
ATTRIB:
  @55d4c6595c18 02 LISTSXP g0c0 [MARK,REF(1)]
    TAG: @55d4c4214330 01 SYMSXP g0c0 [MARK,REF(2719),LCK,gp=0x4000] "dim" (has value)
    @55d4c78dd6b0 13 INTSXP g0c1 [MARK,REF(65535)] (len=2, tl=0) 2,6
```

```
> .Internal(inspect( L ))
@55d4c4c69b08 19 VECSXP g0c2 [MARK,REF(3),ATT] (len=2, tl=0)
  @55d4c5f85918 14 REALSXP g0c5 [MARK,REF(2)] (len=10, tl=0) 0.235022,0.514864,0.590083,0.830803,0.6492
  @55d4c42a31a0 16 STRSXP g0c1 [MARK,REF(4)] (len=1, tl=0)
    @55d4c42a3168 09 CHARSXP g0c1 [MARK,REF(1),gp=0x60] [ASCII] [cached] "dada"
ATTRIB:
  @55d4c434c860 02 LISTSXP g0c0 [MARK,REF(1)]
    TAG: @55d4c4213f40 01 SYMSXP g0c0 [MARK,REF(20948),LCK,gp=0x4000] "names" (has value)
    @55d4c4c69b88 16 STRSXP g0c2 [MARK,REF(65535)] (len=2, tl=0)
      @55d4c44f6ef8 09 CHARSXP g0c1 [MARK,REF(36),gp=0x61] [ASCII] [cached] "a"
      @55d4c480b740 09 CHARSXP g0c1 [MARK,REF(15),gp=0x61] [ASCII] [cached] "b"
```

Les plus braves pourront consulter le code de cette fonction, ainsi que tout le code de R, à cette adresse : [https://github.com/wch/r-source/tree/trunk/src] plus précisément pour `inspect`, dans `src/main/inspect.c`...

```
## Loading required package: readr
## Warning: replacing previous import 'lifecycle::last_warnings' by 'rlang::last_warnings' when
## loading 'hms'
## Warning: replacing previous import 'lifecycle::last_warnings' by 'rlang::last_warnings' when
## loading 'tibble'
## Warning: replacing previous import 'lifecycle::last_warnings' by 'rlang::last_warnings' when
## loading 'pillar'
```

Chapitre 2

Introducting C++

Types (integer, floats, bool), arrays, and flow control statements.

2.1 Using `Rcpp::sourceCpp`

If you are using linux, install R, Rstudio and the Rcpp package (use `install.packages("Rcpp")`), as well as a C++ compiler such as g++.

If you are using Windows or macOS, install R, Rstudio, the Rcpp package, and Rtools. The simplest way to make sure everything works is to follow the following instructions:

1. Installer Rcpp via la commande install.packages("Rcpp")
2. Cliquer dans le menu `file > new > c++ file` (ou l'équivalent en français) pour créer un nouveau fichier C++ ; un fichier contenant quelques lignes d'exemples va être créé. Sauvez ce fichier puis cliquez sur Source. Rstudio doit vous proposer d'installer "Rtools" : acceptez.
3. Cliquer à nouveau sur Source. Tout doit fonctionner…! Vous êtes prêt à apprendre le C++.

Les exemples de code proposés utilisent souvent le standard C++11. Pour pouvoir compiler avec ce standard, n'omettez pas la ligne `// [[Rcpp::plugins(cpp11)]]` in the source files. Altenatively, using `Sys.setenv("PKG_CXXFLAGS" = "-std=c++11")` inside a R session will enable C++11 compilation once for all.

2.2 Hello world

```
// file: vec.cpp
#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector vec0(int n) {
  Rcpp::NumericVector x(n);
  return x;
}

// accès aux éléments
```

```
// [[Rcpp::export]]
Rcpp::IntegerVector vec1(int n) {
  Rcpp::IntegerVector x(n);
  for(int i = 0; i < n; i++) {
    x[i] = i*i;
  }
  return x;
}
```

Il faut toujours commencer par saluer le monde. Créez un fichier `hello.cpp` contenant le code suivant :

```
// file: hello.cpp
#include <Rcpp.h>
#include <iostream>
//[[Rcpp::export]]
void hello() {
  Rcpp::Rcout << "Hello world!\n";
}
```

Compilez le depuis R (il faut avoir installé le package Rcpp) :

```
library(Rcpp)
sourceCpp("hello.cpp")
```

(ou, si vous utilisez R Studio, cliquez sur « source »…). Appelez ensuite la fonction en R :

```
> hello()
Hello world!
```

Dans le programme C++, les directives d'inclusion `#include` servent à inclure des librairies. La librairie `Rcpp.h` permet l'interaction avec les objets de R ; la définition de l'objet `Rcpp::Rcout`, un « flux de sortie » (output stream) qui permet l'écriture dans la console R y est incluse. La librairie `iostream` contient en particulier la définition de l'opérateur `<<`. Elle n'est en fait pas nécessaire ici car `Rcpp.h` contient une directive d'inclusion similaire.

2.3 Why types are necessary

C++ est un langage compilé et non interprété. Le compilateur est le programme qui lit le code C++ et produit un code assembleur puis du langage machine (possiblement en passant par un langage intermédiaire).

Les instructions de l'assembleur (et du langage machine qui est sa traduction numérique directe) manipulent directement les données sous formes de nombre codés en binaire, sur 8, 16, 32 ou 64 bits. La manipulation de données complexes (des vecteurs, des chaines de caractères) se fait bien sûr en manipulant une suite de tels nombres.

Pour que le compilateur puisse produire de l'assembleur, il faut qu'il sache la façon dont les données sont codées dans les variables. La conséquence est que toutes les variables doivent déclarées, et ne pourrons pas changer de type ; de même, le type des valeurs retournées par les fonctions doit être fixé, ainsi que celui de leurs paramètres.

Les fantaisies permises par R (voir ci-dessous) ne sont plus possibles (étaient-elles souhaitables ?).

```
fantaisies <- function(a) {
  if(a == 0) {
    return(a)
  } else {
    return("Non nul")
  }
}
```

```
> fantaisies(0)
[1] 0
```

```
> fantaisies(1)
[1] "Non nul"
```

```
> fantaisies("0")
[1] "0"
```

```
> fantaisies("00")
[1] "Non nul"
```

La librairie standard de C++ offre une collection de types de données très élaborés et de fonctions qui les manipulent. Nous commencerons par les types fondamentaux : entiers, flottants, booléens.

2.4 Integers

There are several types of integers in C++.

2.4.1 The four main types of integers.

Compilez ce programme qui affiche la taille (en octets) des quatre types d'entiers (signés) (le résultat peut théoriquement varier d'une architecture à l'autre, c'est-à-dire qu'il n'est pas fixé par la description officielle du C++).

```
// file: int_types.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void int_types() {
  char a;
  short b;
  int c;
  long int d;
  int64_t e;

  Rcout << "sizeof(a) = " << sizeof(a) << "\n";
  Rcout << "sizeof(b) = " << sizeof(b) << "\n";
  Rcout << "sizeof(c) = " << sizeof(c) << "\n";
  Rcout << "sizeof(d) = " << sizeof(d) << "\n";
  Rcout << "sizeof(e) = " << sizeof(e) << "\n";
}
```

Notez une nouveauté ci-dessus : la directive `using namespace Rcpp` qui permet de taper `Rcout` au lieu de `Rcpp::Rcout`. C'est commode mais à utiliser avec parsimonie (et à bon escient) : il n'est en effet pas rare que des fonctions appartenant à des namespace différents portent le même nom. La syntaxe `namespace::fonction` permet d'éviter toute ambiguïté.

```
> int_types()
sizeof(a) = 1
sizeof(b) = 2
sizeof(c) = 4
sizeof(d) = 8
sizeof(e) = 8
```

Une compilation sous Windows ne produit pas les mêmes résultats: les `long int` ne font que 32 bits. Pour une implémentation portable, la solution est d'utiliser des types où la taille est explicite, comme `int64_t`.

Les entiers de R correspondent au type `int`(sur 32 bits) mais cela ne vous empêche pas de manipuler dans vos fonctions C++ des entiers plus courts ou plus longs si vous en avez besoin.

2.4.2 Unsigned integers

Il existe aussi des types non signés, par exemple `unsigned int` ou `unsigned char` ; et des raccourcis variés, par exemple `size_t` pour `unsigned long int` ou `uint16_t` pour des entiers non signées de 8 bits.

```
// file: non_signes.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void non_signes() {
  int16_t x = 32766;
  uint16_t y = 32766;
  Rcout << "x = " << x << ", y = " << y << "\n";
  x = x+1; y = y+1;
  Rcout << "x = " << x << ", y = " << y << "\n";
  x = x+1; y = y+1;
  Rcout << "x = " << x << ", y = " << y << "\n";
  x = x+1; y = y+1;
  Rcout << "x = " << x << ", y = " << y << "\n";
}
```

Sur 16 bits, les entiers non signés vont de -32768 à 32767, et les entiers signés de 0 à 65535:

```
> non_signes()
x = 32766, y = 32766
x = 32767, y = 32767
x = -32768, y = 32768
x = -32767, y = 32769
```

2.4.3 Numerical overflow

```cpp
// file: overflow.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void overflow() {
  unsigned short x(65530);
  Rcout << "x = " << x << "\n" ;
  x = x+5;
  Rcout << "x = " << x << "\n" ;
  x = x+5;
  Rcout << "x = " << x << "\n" ;
}
```

```
> overflow()
x = 65530
x = 65535
x = 4
```

2.4.4 Notre première fonction « non void »

Écrivons notre première fonction qui renvoie une valeur. Son type doit être déclaré comme ceci :

```cpp
// file: somme_entiers.cpp
//[[Rcpp::export]]
int somme_entiers(int a, int b) {
  return a+b;
}
```

Et testons la :

```
> somme_entiers(1L, 101L)
[1] 102
```

```
> somme_entiers(1.9,3.6)
[1] 4
```

Que se passe-t-il ? Utilisez la fonction suivante pour comprendre.

```cpp
// file: cast_to_int.cpp
//[[Rcpp::export]]
int cast_to_int(int x) {
  return x;
}
```

2.4.5 Initialisation des variables

Il est nécessaire d'initialiser les variables.

```cpp
// file: uninit.cpp
//[[Rcpp::export]]
int uninit() {
  int a; // a peut contenir n'importe quoi
  return a;
}
```

Testons :

```
> uninit()
[1] 0
```

```
> uninit()
[1] 0
```

On aura parfois 0, mais pas systématiquement (cela dépend de l'état de la mémoire). On peut initialiser `a` lors de la déclaration : `int a = 0;`

2.5 Les flottants

Il y a trois types de nombres en format flottant. Le type utilisé par R est le `double` de C++.

2.5.1 The free types of floating point numbers

```cpp
// file: float_types.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
void float_types() {
  float a;
  double b;
  long double c;
  Rcpp::Rcout << "sizeof(a) = " << sizeof(a) << "\n";
  Rcpp::Rcout << "sizeof(b) = " << sizeof(b) << "\n";
  Rcpp::Rcout << "sizeof(c) = " << sizeof(c) << "\n";
}
```

2.5.2 Précision du calcul

Parenthèse de culture informatique générale. Voici ce que répond R au test `1.1 - 0.9 = 0.2`.

```
> 1.1 - 0.9 == 0.2
[1] FALSE
```

Pourquoi ? Est-ce que C++ fait mieux ? (Rappel : R utilise des `double`).

Sur les architectures courantes, les nombres au format `double` sont codés sur 64 bits (voir ci-dessus, taille 8 octets). C'est un format « à virgule flottante », c'est-à-dire qu'ils sont représentés sous la forme $a2^b$, ou a et b sont bien sûr codés en binaire (sur 53 bits – dont un bit 1 implicite – pour a, 11 pour b, et un bit de signe).

Cette précision finie implique des erreurs d'arrondi. Pour plus de détails, voir Wikipedia sur la norme IEEE 754 : https://fr.wikipedia.org/wiki/IEEE_754

Quelle est la différence entre les nombres ci-dessus ?

```
> (1.1 - 0.9) - 0.2
[1] 5.551115e-17
```

C'est-à-dire 2^{-54} (une erreur au 53e chiffre...). Affichons la représentation interne des nombres en question avec la fonction `bits` du package `pryr`.

```
> pryr::bits(1.1 - 0.9)
[1] "00111111 11001001 10011001 10011001 10011001 10011001 10011001 10011100"
```

```
> pryr::bits(0.2)
[1] "00111111 11001001 10011001 10011001 10011001 10011001 10011001 10011010"
```

2.5.3 Valeurs spéciales et extrêmes

Il y a des valeurs spéciales en C++ comme en R : une valeur infinie, et une valeur non-définie `NaN`, pour *not a number*.

```
// file: divise.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
double divise(double a, double b) {
  double r = a/b;
  Rcpp::Rcout << a << " / " << b << " = " << r << std::endl;
  return r;
}
```

```
> divise(1,2)
1 / 2 = 0.5
[1] 0.5
```

```
> divise(1,0)
1 / 0 = inf
[1] Inf
```

```
> divise(-1,0)
-1 / 0 = -inf
[1] -Inf
```

```
> divise(0,0)
0 / 0 = -nan
[1] NaN
```

En C++, la fonction `numeric_limits` permet d'obtenir les valeurs extrêmes que peuvent prendre les `double`.

```cpp
// file: numeric_limits.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
void numeric_limits() {
  Rcpp::Rcout
    << "plus petite valeur positive 'normale' = "
    << std::numeric_limits<double>::min()         << "\n"
    << "plus petite valeur positive           = "
    << std::numeric_limits<double>::denorm_min()  << "\n"
    << "plus grande valeur                    = "
    << std::numeric_limits<double>::max() << "\n"
    << "epsilon                               = "
    << std::numeric_limits<double>::epsilon()     << "\n";
}
```

```
> numeric_limits()
plus petite valeur positive 'normale' = 2.22507e-308
plus petite valeur positive           = 4.94066e-324
plus grande valeur                    = 1.79769e+308
epsilon                               = 2.22045e-16
```

2.5.4 Constantes numériques

Attention, si le R considère que `0` ou `1` est un `double` (il faut taper `0L` ou `1L` pour avoir un `integer`), pour C++ ces valeurs sont des entiers. Pour initialiser proprement un `double` il faudrait normale taper `0.` ou `0.0`, etc; cependant le compilateur fera la conversion de type si nécessaire.

2.6 Opérateurs arithmétiques

Les opérateurs arithmétiques sont bien entendu `+`, `-`, `*` et `/`. Pour les entiers, le modulo est `%`.

```cpp
// file: division_entiere.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void division_entiere(int a, int b) {
  int q = a / b;
  int r = a % b;
  Rcpp::Rcout << a << " = " << b << " * " << q << " + " << r << std::endl;
}
```

```
> division_entiere(128, 7)
128 = 7 * 18 + 2
```

À ces opérateurs, il faut ajouter des opérateurs d'assignation composée `+=`, `-=`, `*=` et `/=` qui fonctionnent ainsi : `x += 4;` est équivalent à `x = x + 4`, et ainsi de suite. Il y a aussi les opérateurs d'incrémentation `++` et de décrémentation `--`.

```cpp
// file: operateurs_exotiques.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
```

```cpp
void operateurs_exotiques(int a) {
  Rcpp::Rcout << "a = " << a << std::endl;

  Rcpp::Rcout << "a *= 2;" << std::endl;
  a *= 2;
  Rcpp::Rcout << "a = " << a << std::endl;

  Rcpp::Rcout << "int b = a++;" << std::endl;
  int b = a++; // post incrementation
  Rcpp::Rcout << "b = " << b << std::endl;
  Rcpp::Rcout << "a = " << a << std::endl;

  Rcpp::Rcout << "int c = ++a;" << std::endl;
  int c = ++a; // pre incrementation
  Rcpp::Rcout << "c = " << c << std::endl;
  Rcpp::Rcout << "a = " << a << std::endl;
}
```

```
> operateurs_exotiques(3)
a = 3
a *= 2;
a = 6
int b = a++;
b = 6
a = 7
int c = ++a;
c = 8
a = 8
```

2.7 Conversion de type: les “cast”

Le compilateur peut réaliser une conversion d’un type à l’autre: on parle de *cast*. Cette conversion peut être implicite, lors par exemple d’une copie d’un type `double` vers un type `int` ; elle peut être explicite, lors par exemple de la copie d’une valeur de type `double` vers un `int` ; elle peut être rendue explicite en mettant un nom de type entre parenthèses devant une variable : `(int) x` fera une conversion de `x` vers le type `int` (si le type de `x` rend ça possible).

```cpp
// file: cast.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
void cast(int x, int y) {
  double a = x; // cast implicite
  double b = (double) y; // cast explicite
  double q1 = x / y; // cast implicite (à quel moment a-t-il lieu ?)
  double q2 = (double) x / (double) y; // cast explicite
  Rcpp::Rcout << "q1 = " << q1 << "\n";
  Rcpp::Rcout << "q2 = " << q2 << "\n";
}
```

Cet exemle montre les écueils du cast implicite :

```
> cast(4,3)
q1 = 1
q2 = 1.33333
```

Lors du calcul de `q1`, le cast a été fait après la division entière… était-ce le comportement désiré ?

2.8 Booléens

Le type `bool` peut prendre les valeurs vrai/faux. Il correspond au type `logical` de R.

```
// file: test_positif.cpp
// [[Rcpp::export]]
bool test_positif(double x) {
  return (x > 0);
}
```

Les opérateurs de test sont comme en R, >, >=, <, <=, == et !=. Les opérateurs logiques sont && (et), || (ou) et ! (non). **Attention !** Les opérateurs & et | existent également, ce sont des opérateurs logiques bit à bit qui opèrent sur les entiers.

```
// file: test_interval.cpp
// [[Rcpp::export]]
bool test_interval(double x, double min, double max) {
  return (min <= x && x <= max);
}
```

2.9 Tableaux de taille fixe

On peut définir des tableaux de taille fixe fixe (connue à la compilation) ainsi:

```
// file: petit_tableau.cpp
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
//[[Rcpp::export]]
void petit_tableau() {
  int a[4] = {0,2,7,11};
  SHOW(a)    // wut ?
  SHOW(a[0])
  SHOW(a[1])
  SHOW(a[2])
  SHOW(a[3])
}
```

L'occasion est saisie pour montrer l'utilisation d'une macro. La ligne `SHOW(a[0])` est remplacée par `Rcpp::Rcout << "a[0]" << " = " << (a[0]) << std::endl;` **avant** la compilation. Les macros peuvent rendre de grand services pour la clarté du code ou pour faciliter le débuggage « manuel ».

L'utilisation de parenthèse autour de `(x)` dans la définition de la macro est très conseillée : si on utilisait par exemple `SHOW(a == b)` il n'y a aucun problème avec la syntaxe `Rcout << (a == b) << std::endl;` mais `Rcout << a == b << std::endl;` pourrait poser des problèmes de priorité des opérateurs == et <<…

Le résultat de `SHOW(a)` sera expliqué plus tard (pointeurs).

2.10 Contrôle du flux d'exécution : les boucles

2.10.1 Boucles `for`

Plus de 90% des boucles `for` s'écrivent ainsi :

```
// file: ze_loop.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void ze_loop(int n) {
  for(int i = 0; i < n; i++) {
    Rcpp::Rcout << "i = " << i << std::endl;
  }
}
```

```
> ze_loop(4)
i = 0
i = 1
i = 2
i = 3
```

Le premier élément dans la parenthèse (ici, `int i = 0`) est l'initialisation ; il sera exécuté une seule fois, et c'est généralement une déclaration de variable (avec une valeur initiale). Le deuxième élément (`i < n`) est la condition à laquelle la boucle sera exécutée une nouvelle fois, c'est généralement une condition sur la valeur de la variable ; et le dernier élément (`i++`) est exécuté à la fin de chaque tour de boucle, c'est généralement une mise à jour de la valeur de cette variable.

Il est facile par exemple d'aller de 2 en 2 :

```
// file: bouclette.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void bouclette(int n) {
  for(int i = 0; i < n; i += 2) {
    Rcpp::Rcout << "i = " << i << std::endl;
  }
}
```

```
> bouclette(6)
i = 0
i = 2
i = 4
```

Pour revenir sur les types d'entiers : gare au dépassement arithmétique.

```
// file: arithmetic_overflow.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void arithmetic_overflow() {
  int x = 1;
  for(int i = 0; i < 33; i++) {
    Rcpp::Rcout << "2^" << i << " = " << (x) << "\n";
    x = 2*x;
```

```
  }
}
```

Essayer avec `unsigned int`, `long int`.

2.10.2 `continue` et `break`

Une instruction `continue` en cours de boucle fait passer au tour suivant :

```
// file: trois.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void trois(int n) {
  for(int i = 1; i <= n; i++) {
    Rcpp::Rcout << i << " " ;
    if(i%3 != 0)
      continue;
    Rcpp::Rcout << "\n";
  }
  Rcpp::Rcout << "\n";
}
```

```
> trois(9)
1 2 3
4 5 6
7 8 9
```

Quant à `break`, si s'agit bien sûr d'une interruption de la boucle.

```
// file: zz.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void zz(int n, int z) {
  for(int i = 0; i < n; i++) {
    Rcpp::Rcout << "A" ;
    if(i > z)
      break;
  }
  Rcpp::Rcout << std::endl;
}
```

```
> zz(14, 100)
AAAAAAAAAAAAAA
```

```
> zz(14, 5)
AAAAAAA
```

2.10.3 Boucles `while` et `do while`

Ces boucles ressemblent fort à ce qui existe en R. Dans un cas, le test est fait avant la boucle, dans l'autre il est fait après.

```cpp
// file: a_rebours_1.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void a_rebours_1(int n) {
  while(n-- > 0) {
    Rcpp::Rcout << n << " ";
  }
  Rcpp::Rcout << std::endl;
}

// [[Rcpp::export]]
void a_rebours_2(int n) {
  do {
    Rcpp::Rcout << n << " ";
  } while(n-- > 0);
  Rcpp::Rcout << std::endl;
}
```

```
> a_rebours_1(3)
2 1 0
```

```
> a_rebours_2(3)
3 2 1 0
```

On peut aussi utiliser `continue` et `break` dans ces boucles.

Considérons un exemple un peu moins artificiel : le calcul d'une racine carrée par l'algorithme de Newton. L'avantage de la syntaxe `do while` est apparent ici.

```cpp
// file: racine_carree.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
double racine_carree(double x, double eps = 1e-5) {
  double s = x;
  do {
    s = 0.5*(s + x/s);
  } while( fabs(s*s - x) > eps);
  return s;
}
```

```
> racine_carree(2)
[1] 1.414216
```

```
> racine_carree(2, 1e-8)
[1] 1.414214
```

Cherchez sur le site `cpprefrence.com` la description des fonctions `abs` et `fabs`. Pourquoi ne pouvait-on pas utiliser `abs` ici ? Est-il raisonnable de proposer une valeur trop petite pour `eps` ? Proposer une modification de la fonction qui évite cet écueil.

2.11 Contrôle du flux d'exécution : les alternatives

2.11.1 `if` et `if else`

Cela fonctionne tout à fait comme en R ; la x

```
// file: mini.cpp
// [[Rcpp::export]]
double mini(double x, double y) {
  double re = 0;
  if(x > y) {
    re = y;
  } else {
    re = x;
  }
  return re;
}
```

```
> mini(22, 355)
[1] 22
```

2.11.2 `switch`

Un exemple simple devrait permettre de comprendre le fonctionnement de `switch`.

```
// file: combien.cpp
#include <Rcpp.h>
// [[Rcpp::export]]
void combien(int n) {
  switch(n) {
    case 0:
      Rcpp::Rcout << "aucun\n";
      break;
    case 1:
      Rcpp::Rcout << "un seul\n";
      break;
    case 2:
      Rcpp::Rcout << "deux\n";
      break;
    case 3:
    case 4:
    case 5:
      Rcpp::Rcout << "quelques uns\n";
      break;
    default:
      Rcpp::Rcout << "beaucoup\n";
  }
}
```

Chapitre 3

Manipuler les objets de R en C++

All the examples are in the R package... Install it with

```
> devtools::install_github("introRcpp/introRcppManipulation")
```

Load it with

```
library(introRcppManipulation)
```

3.1 Premiers objets Rcpp : les vecteurs

La librairie `Rcpp` définit des types `NumericVector`, `IntegerVector` et `LogicalVector` qui permettent de manipuler en C++ les vecteurs de R.

3.1.1 Créer des vecteurs, les manipuler

L'initialisation avec la syntaxe utilisée dans `vec0` remplit le vecteur de 0. Notez l'accès aux éléments d'un vecteur par l'opérateur `[]`; **contrairement à la convention utilisée par R, les vecteurs sont numérotés de 0 à n-1.**

```
// file: vec.cpp
#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector vec0(int n) {
  Rcpp::NumericVector x(n);
  return x;
}

// accès aux éléments
// [[Rcpp::export]]
Rcpp::IntegerVector vec1(int n) {
  Rcpp::IntegerVector x(n);
  for(int i = 0; i < n; i++) {
    x[i] = i*i;
  }
```

```
  return x;
}
```

3.1.2 Exemple : compter les zéros

```
// file: countZeroes.cpp
#include <Rcpp.h>
//[[Rcpp::export]]
int countZeroes(Rcpp::IntegerVector x) {
  int re = 0;
  // x.size() et x.length() renvoient la taille de x
  int n = x.size();
  for(int i = 0; i < n; i++) {
    if(x[i] == 0) ++re;
  }
  return re;
}
```

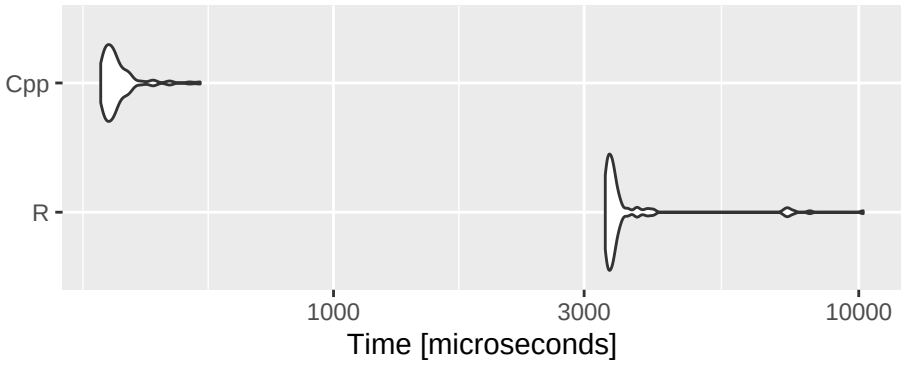
Comment les performances de cette fonction se comparent-elles avec le code R `sum(a == 0)` ?

```
> set.seed(1); a <- sample(0:99, 1e6, TRUE)
> countZeroes(a);
[1] 10017
```

```
> sum(a == 0)
[1] 10017
```

```
> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp = countZeroes(a))
> mbm
Unit: microseconds
 expr      min       lq      mean   median        uq       max neval
    R 3291.519 3335.910 3745.7610 3397.898 3482.5030 10199.323   100
  Cpp  360.382  370.028  388.7066  380.161  393.5625   557.493   100
```

```
> ggplot2::autoplot(mbm)
```



La différence de vitesse d'exécution s'explique en partie par le fait que le code R commence par créer un vecteur de type `logical` (le résultat de `a == 0`), puis le parcourt pour faire la somme. Ceci implique beaucoup de lectures écritures en mémoire, ce qui ralentit l'exécution.

3.2 Vecteurs

3.2.1 Creating vectors

On a vu l'initialisation avec la syntaxe `NumericVector R(n)` qui crée un vecteur de longueur n, rempli de 0. On peut utiliser `NumericVector R(n, 1.0)` pour un vecteur rempli de 1 ; **attention à bien taper `1.0` pour avoir un `double` et non un `int`; dans le cas contraire, on a un message d'erreur difficilement compréhensible à la compilation**.

On peut utiliser `NumericVector R = no_init(n);` pour un vecteur non initialisé (ce qui fait gagner du temps d'exécution).

```
// file: zeros.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector zeros(int n) {
  IntegerVector R(n);
  return R;
}
```

```
// file: whatever.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector whatever(int n, int a) {
  IntegerVector R(n, a);
  return R;
}
```

```
// file: uninitialized.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector uninitialized(int n) {
  IntegerVector R = no_init(n);
  return R;
}
```

```
// file: favourites.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector favourites() {
  IntegerVector R = IntegerVector::create(1, 4, 8);
  return R;
}
```

```
> zeros(5)
[1] 0 0 0 0 0
```

```
> whatever(5, 2L)
[1] 2 2 2 2 2
```

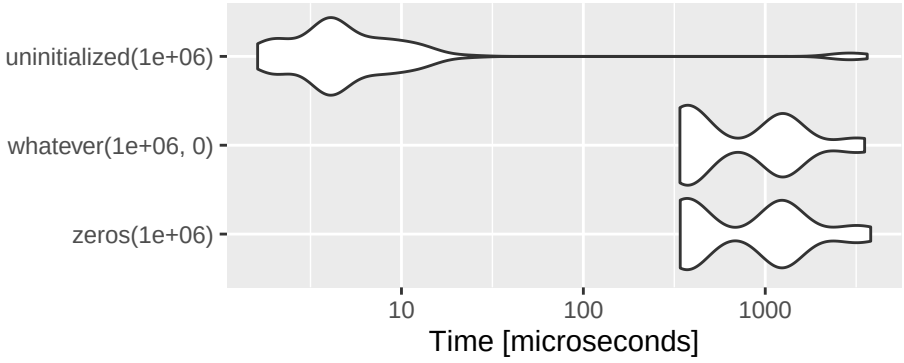
```
> uninitialized(5) # sometime 0s, not always
[1] -1004444712       21972  -999642040       21972  -999769224
```

```
> favourites()
[1] 1 4 8
```

Comparons les performances des trois premières fonctions (comme à chaque fois, les résultats peuvent varier d'une architecture à l'autre).

```
> mbm <- microbenchmark::microbenchmark(zeros(1e6), whatever(1e6, 0), uninitialized(1e6))
> mbm
Unit: microseconds
                 expr     min       lq      mean   median       uq      max neval
        zeros(1e+06) 340.899 367.2795 1062.0242 1195.743 1248.899 3801.419   100
   whatever(1e+06, 0) 339.599 362.8370  974.8824  495.964 1251.943 3518.681   100
 uninitialized(1e+06)   1.619   3.3465  125.0326    4.232    7.841 3637.273   100
```

```
> ggplot2::autoplot(mbm)
```



3.2.2 Accessing elements

Using `x.size()` or `x.length()`. Beware 0-based indices.

BLA BLA

3.2.3 Missing data

Cette fonction utilise `IntegerVector::is_na` qui est la bonne manière de tester si un membre d'un vecteur entier est `NA`.

```
// file: countNAs.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
```

```
int countNAs(NumericVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    re += NumericVector::is_na(x[i]);
  }
  return(re);
}
```

Une nouveauté : le fichier `countNAS.h` ...

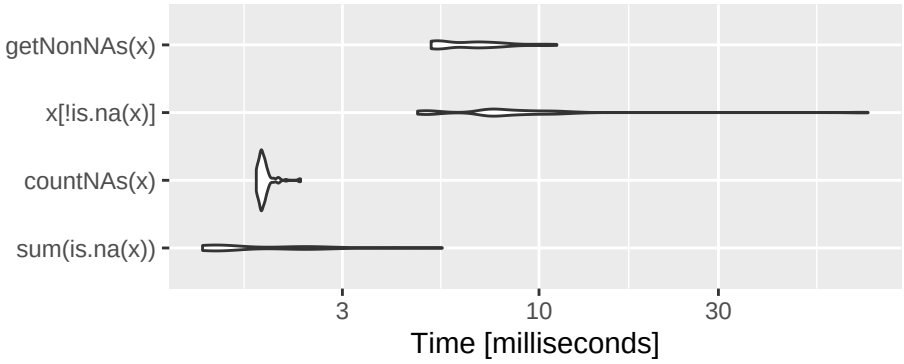
```
// file: countNAs.h
#include <Rcpp.h>
int countNAs(Rcpp::NumericVector x);
```

```
// file: getNonNAs.cpp
#include <Rcpp.h>
#include "countNAs.h"
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector getNonNAs(NumericVector x) {
  int nbNAs = countNAs(x);
  int n = x.size();
  NumericVector R(n - nbNAs);
  int j = 0;
  for(int i = 0; i < n; i++) {
    if(!NumericVector::is_na(x[i])) {
      R[j++] = x[i];
    }
  }
  return R;
}
```

Comparons ces deux fonctions avec leurs analogues R, `sum(is.na(x))` et `x[!is.na(x)]`.

```
> x <- sample( c(NA, rnorm(10)), 1e6, TRUE)
> mbm <- microbenchmark::microbenchmark( sum(is.na(x)), countNAs(x), x[!is.na(x)], getNonNAs(x) )
> mbm
Unit: milliseconds
          expr      min       lq     mean   median       uq       max neval
 sum(is.na(x)) 1.274110 1.376793 2.022847 1.456892 2.390356  5.532537   100
   countNAs(x) 1.770462 1.812295 1.858786 1.833413 1.874128  2.322655   100
  x[!is.na(x)] 4.752685 7.179820 9.917222 7.762717 9.426737 75.024881   100
  getNonNAs(x) 5.165764 5.335571 6.660444 6.625144 7.583220 11.155156   100
```

```
> ggplot2::autoplot(mbm)
```



3.3 Vecteurs nommés

Ça n'est pas passionnant en soi (on ne manipule pas si souvent des vecteurs nommés), mais ce qu'on voit là sera utile pour les listes et les data frames.

3.3.1 Créer des vecteurs nommés

Voici d'abord comment créer un vecteur nommé.

```
// file: createVec1.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector createVec1() {
  NumericVector x = NumericVector::create(Named("un") = 10, Named("deux") = 20);
  return x;
}
```

Application :

```
> a <- createVec1()
> a
  un deux
  10   20
```

Une syntaxe plus dense est possible :

```
// file: createVec2.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector createVec2() {
  NumericVector x = NumericVector::create(_["un"] = 10, _["deux"] = 20);
 return x;
}
```

Cela produit le même résultat.

```
> createVec2()
  un deux
  10   20
```

3.3.2 Accéder aux éléments par leurs noms

On utilise toujours la syntaxe `x[]` :

```
// file: getOne.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
double getOne(NumericVector x) {
  if(x.containsElementNamed("one"))
    return x["one"];
  else
    stop("No element 'one'");
}
```

Notez la fonction `Rcpp::stop` qui correspond à la fonction R du même nom.

```
> getOne(a)
Error in getOne(a): No element 'one'
```

```
> getOne(b)
Error in getOne(b): object 'b' not found
```

3.3.3 Obtenir les noms d'un vecteur

Et voici comment obtenir les noms d'un vecteur.

```
// file: names1.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector names1(NumericVector x) {
  CharacterVector R = x.names(); // ou R = x.attr("names");
  return R;
}
```

Utilisons cette fonction:

```
> names1(a)
[1] "un"   "deux"
```

Cette fonction semble se comporter correctement, elle a cependant un gros défaut. Nous y reviendrons dans la section suivante.

3.4 Objets génériques : SEXP

Les objets R les plus génériques sont les `SEXP`, « S expression ». Les principaux types de `SEXP` sont illustrés par la fonction suivante.

```
// file: RType.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
std::string RType(SEXP x) {
  switch( TYPEOF(x) ) {
    case INTSXP:
      return "integer";
    case REALSXP:
      return "double";
    case LGLSXP:
      return "logical";
    case STRSXP:
      return "character";
    case VECSXP:
      return "list";
    case NILSXP:
      return "NULL";
    default:
      return "autre";
  }
}
```

Utiliser les types définis par Rcpp est généralement plus facile et plus sûr. Cependant à l'intérieur des fonctions Rcpp ils peuvent être utiles, par exemple dans le cas où une fonction peut renvoyer des objets de types différents, par exemple soit un `NILSXP`, soit un objet d'un autre type.

3.4.1 Exemple : vecteurs nommés (ou pas)

Testons à nouveau la fonction `names1`, en lui passant un vecteur non nommé.

```
> b <- seq(0,1,length=6)
> names1(b)
Error in names1(b): Not compatible with STRSXP: [type=NULL].
```

Bien sûr, le vecteur `b` n'a pas de noms ; la fonction `x.names()` a renvoyé l'objet `NULL`, de type `NILSXP`, qui ne peut être utilisé pour initialiser le vecteur `R` de type `STRSXP`. Une solution est d'attraper le résultat de `x.names()` dans un `SEXP`, et de tester son type avec `TYPEOF`.

```
// file: names2.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector names2(NumericVector x) {
  SEXP R = x.names();
  if( TYPEOF(R) == STRSXP )
    return R;
```

```
  else
    return CharacterVector(0);
}
```

```
> names2(a)
[1] "un"   "deux"
```

```
> names2(b)
character(0)
```

3.4.2 Exemple : énumerer les noms et le contenu

On va utiliser l'opérateur `CHAR` qui, appliqué à un élément d'un `CharacterVector`, renvoie une valeur de type `const char *` c'est-à-dire un pointeur vers une chaîne de caractère (constante, ie non modifiable) « à la C » (voir chapitre dédié).

```
// file: enumerate.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void enumerate(NumericVector x) {
  SEXP r0 = x.names();
  if(TYPEOF(r0) != STRSXP) {
    Rcout << "No names\n";
    return;
  }
  CharacterVector R(r0);
  for(int i = 0; i < R.size(); i++) {
    double a = x[ CHAR(R[i]) ];
    Rcout << CHAR(R[i]) << " : " << a << "\n";
  }
}
```

```
> enumerate(a)
un : 10
deux : 20
```

3.5 Facteurs

```
// file: getLevels.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector getLevels(IntegerVector x) {
  SEXP R = x.attr("levels");
  switch(TYPEOF(R)) {
  case STRSXP:
    return R; // Rcpp prend soin que ce SEXP soit converti en CharacterVector
  case NILSXP:
```

```
    stop("No 'levels' attribute");
  default:
    stop("'levels' attribute of unexpected type");
  }
}
```

```
> x <- factor( sample(c("M","F"), 10, TRUE) )
> getLevels(x)
[1] "F" "M"
```

```
> x <- sample(1:2, 10, TRUE)
> # getLevels(x)
> attr(x, "levels") <- c(0.1, 0.4)
> # getLevels(x)
```

```
// file: someFactor.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector someFactor() {
  IntegerVector x = IntegerVector::create(1,1,2,1);
  x.attr("levels") = CharacterVector::create("F", "M");
  x.attr("class") = CharacterVector::create("factor");
  return x;
}
```

```
> someFactor()
[1] F F M F
Levels: F M
```

3.6 Listes et Data frames

Nous avons déjà vu les fonctions utiles dans le cas des vecteurs nommés, en particulier `containsElementNamed`.

La fonction suivante prend une liste `L` qui a un élément `L$alpha` de type `NumericVector` et renvoie celui-ci à l'utilisateur. En cas de problème un message d'erreur informatif est émis.

```
// file: getAlpha.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector getAlpha(List x) {
  if( x.containsElementNamed("alpha") ) {
    SEXP R = x["alpha"];
    if( TYPEOF(R) != REALSXP )
      stop("alpha is not of type 'NumericVector'");
    return R;
  } else
    stop("No element named alpha");
}
```

Pour renvoyer des valeurs hétéroclites dans une liste c'est très facile:

```cpp
// file: createList.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
List createList() {
  List L;
  L["a"] = NumericVector::create(1.0, 2.0, 4.0);
  L["b"] = 12;
  L["c"] = rnorm(4, 0.0, 1.0);
  return L;
}
```

```
> createList()
$a
[1] 1 2 4

$b
[1] 12

$c
[1] -1.2275839 -0.7487483  1.4054582 -0.1656251
```

Les data frames, ont l'a vu, sont des listes avec quelques attributs supplémentaires. En Rcpp cela fonctionne de la même façon, avec la classe `DataFrame`. Ils ont une certaine tendance à se transformer en liste quand on leur ajoute des éléments.

Here is a useful trick.

```cpp
// file: createDF.cpp
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
DataFrame createDF() {
  List L;
  L["a"] = NumericVector::create(1.0, 2.0, 4.0, 8.0);
  L["b"] = CharacterVector::create("alpha", "beta", "gamma", "delta");
  L["c"] = rnorm(4, 0.0, 1.0);

  L.attr("class") = "data.frame";
  L.attr("row.names") = IntegerVector::create(NA_INTEGER, -4); // 4 unnamed rows
  return L;
}
```

```
> createDF()
  a     b          c
1 1 alpha -0.2627742
2 2  beta  0.3077454
3 4 gamma  1.1554909
4 8 delta  1.7582734
```