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] 4 3 1 3 2 2 2 2 1 4 ...
```

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

```
> str(L)
List of 2
 $ a: num [1:10] 0.844 0.101 0.955 0.384 0.543 ...
 $ 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
0.1882559 0.1879496 0.8935739 0.1018279
```

```
> 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,]    4    1    2    2    1    1
[2,]    3    3    2    2    4    1
```

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(4L, 3L, 1L, 3L, 2L, 2L, 2L, 2L, 1L, 4L, 1L, 1L), .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 ))
@55671e94f148 13 INTSXP g0c0 [REF(65535)]  1 : 10 (compact)
```

```
> .Internal(inspect( c(0.1, 0.2) ))
@55671f81bdb8 14 REALSXP g0c2 [] (len=2, tl=0) 0.1,0.2
```

```
> a <- c(0.1, 0.2)
> .Internal(inspect( a ))
@55671f87fdb8 14 REALSXP g0c2 [REF(2)] (len=2, tl=0) 0.1,0.2
```

```
> names(a) <- c("A", "B")
> .Internal(inspect( M ))
@55671f7478f8 13 INTSXP g0c4 [MARK,REF(5),ATT] (len=12, tl=0) 4,3,1,3,2,...
ATTRIB:
  @55671e706e10 02 LISTSXP g0c0 [MARK,REF(1)]
    TAG: @55671c38b330 01 SYMSXP g0c0 [MARK,REF(2719),LCK,gp=0x4000] "dim" (has value)
    @55671fa55978 13 INTSXP g0c1 [MARK,REF(65535)] (len=2, tl=0) 2,6
```

```
> .Internal(inspect( L ))
@55671cef6248 19 VECSXP g0c2 [MARK,REF(3),ATT] (len=2, tl=0)
  @55671c7972c8 14 REALSXP g0c5 [MARK,REF(2)] (len=10, tl=0) 0.843966,0.101238,0.955249,0.384423,0.5427
  @55671c41a910 16 STRSXP g0c1 [MARK,REF(4)] (len=1, tl=0)
    @55671c41a8d8 09 CHARSXP g0c1 [MARK,REF(1),gp=0x60] [ASCII] [cached] "dada"
ATTRIB:
  @55671c4d3100 02 LISTSXP g0c0 [MARK,REF(1)]
    TAG: @55671c38af40 01 SYMSXP g0c0 [MARK,REF(20950),LCK,gp=0x4000] "names" (has value)
    @55671cef62c8 16 STRSXP g0c2 [MARK,REF(65535)] (len=2, tl=0)
      @55671c66def8 09 CHARSXP g0c1 [MARK,REF(36),gp=0x61] [ASCII] [cached] "a"
      @55671c982740 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`...

```
# child="02-bases.Rmd"}
```

```
## 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

Manipuler les objets de R en C++

All the examples are in the R package…

```
library(manipulation)
```

2.1 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.

2.1.1 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;
}
```

2.1.2 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;
}
```

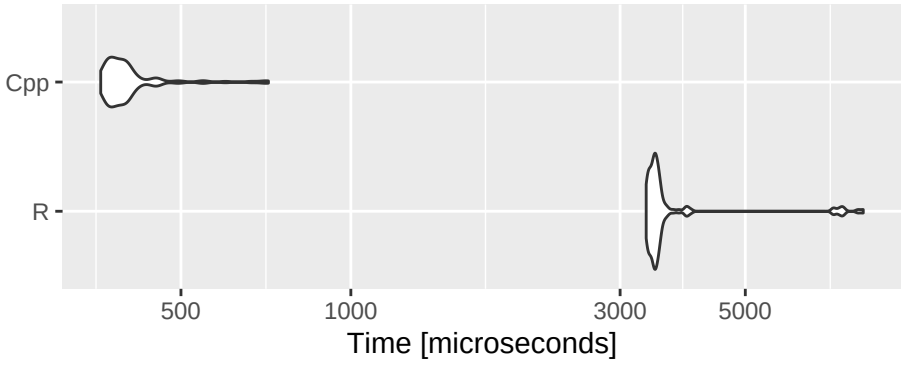
Comment les performances de cette fonction se comparent-elles avec le code R `sum(a == 0)` ?

```
> 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 3336.581 3400.197 3798.7695 3460.3445 3522.978 8090.451   100
  Cpp  360.402  373.093  407.5573  390.3395  405.691  714.441   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.

2.2 Vecteurs

2.2.1 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
```

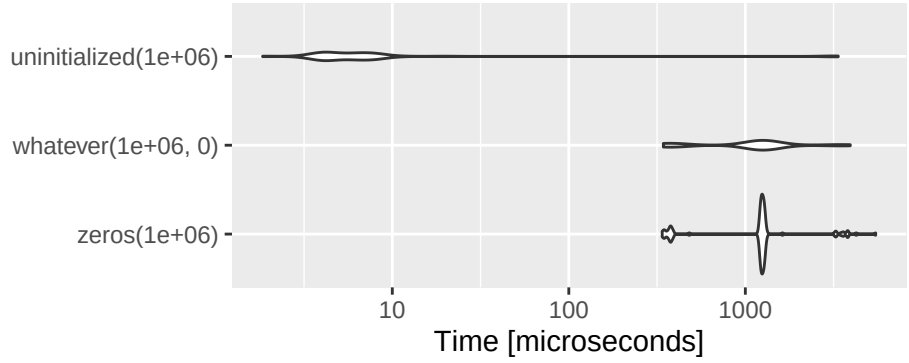
```
> uninitialized(5) # sometime 0s, not always
[1]    6    0    6    0 9999
```

```
> 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) 338.065 1211.9625 1365.6894 1248.508 1281.439 5479.493   100
   whatever(1e+06, 0) 343.198  413.7950 1230.7880 1247.390 1270.996 3931.159   100
 uninitialized(1e+06)   1.842    4.1455  131.7469    5.733    7.672 3360.254   100
```

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



2.2.2 Accessing elements

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

BLA BLA

2.2.3 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` ...

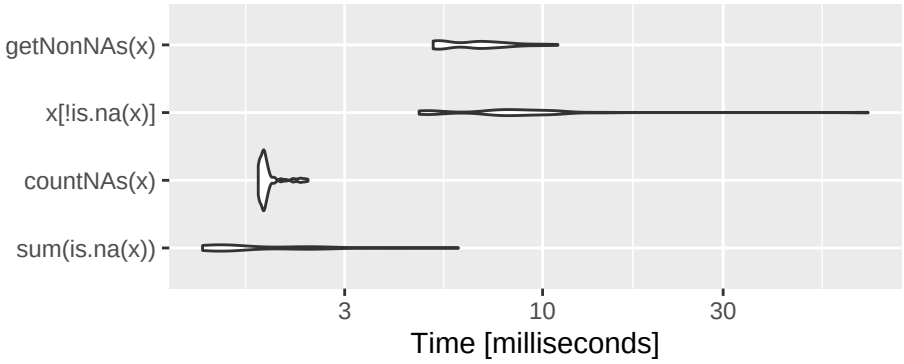
```
// file: countNAs.h
#include <Rcpp.h>
int countNAs(Rcpp::NumericVector x);
```

```
// 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)) 1.263474 1.370375 2.048044 1.494245 2.394342  5.994505   100
   countNAs(x) 1.773070 1.806960 1.881767 1.834421 1.876297  2.399723   100
  x[!is.na(x)] 4.717501 6.807713 9.990207 8.155187 9.659752 72.472244   100
  getNonNAs(x) 5.138561 5.286307 6.661075 6.690502 7.640535 10.976388   100
```

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



2.3 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.

2.3.1 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
```

2.3.2 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
```

2.3.3 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.

2.4 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.

2.4.1 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)
```

2.4.2 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
```

2.5 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
```

2.6 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
```