

Introduction à C++ et Rcpp

Hervé Perdry

2022-06-24

Table des matières

À propos de ce document	5
1 Objets de R	7
1.1 Types	7
1.2 Attributs	9
1.3 Mieux examiner les objets	10
1.4 Pour les apprentis sorciers	11
2 Bases de C++	13
2.1 Nécessité des types	13
2.2 Les entiers	14
2.3 Les flottants	16
2.4 Constantes numériques	18
2.5 Opérateurs arithmétiques	18
2.6 Conversion de type: les “cast”	19
2.7 Booléens	20
2.8 Tableaux de taille fixe	20
2.9 Contrôle du flux d’exécution : les boucles	21
2.10 Contrôle du flux d’exécution : les alternatives	24
3 Manipuler les objets de R en C++	25
3.1 Premiers objets Rcpp : les vecteurs	25
3.2 Vecteurs	26
3.3 Vecteurs nommés	29
3.4 Objets génériques : SEXP	31
3.5 Facteurs	33
3.6 Listes et Data frames	33
4 Sucre syntaxique	35
4.1 Efficacité	35
4.2 Un exemple de la Rcpp Gallery	37

5 Exemple : Metropolis-Hastings	39
5.1 L'algorithme	39
5.2 Une version R	42
5.3 Première version en C++	42
5.4 Une version améliorée	42
6 Un peu plus de C++	45
6.1 Pointeurs	45
6.2 Références	47
6.3 Les objets de Rcpp sont passés par référence	49
6.4 Surcharge	51
6.5 Templates	52
6.6 C++11	55
7 Introduction à la librairie standard	57
7.1 Vecteurs	57
7.2 Pairs & tuples	61
7.3 Quelques algorithmes	61
7.4 Maps	64
7.5 Heap	64
8 Création d'un package R	65
8.1 Créer l'arborescence de fichiers	65
8.2 Inclure une fonction C++	66
8.3 Ce que Rstudio a fait avant la compilation	67
8.4 Inclure une fonction R	67
8.5 Contrôler quelles sont les fonctions exportées	68
8.6 Documenter les fonctions avec roxygen2	68
8.7 Générer le fichier NAMESPACE avec roxygen2	69
8.8 Compilation multifichier	69
9 Chaînes de caractères	75
9.1 Chaînes de la vieille école : C strings	76
9.2 Chaînes C++	78
9.3 Chaînes Rcpp::String	80
9.4 Exemple : lecture d'un fichier	81
9.5 Addendum sur les listes	81
9.6 Exemple : chaînes de caractère et spécialisation de template	82

À propos de ce document

Ce document est issu d'un cours d'école doctorale donné en février/mars 2019. Un effort a été fait pour le rendre lisible sans les commentaires fait en cours, mais il y a certainement toujours des lacunes.

Les exemples de code proposés utilisent souvent le standard C++11. Pour pouvoir compiler avec ce standard, utilisez `Sys.setenv("PKG_CXXFLAGS" = "-std=c++11")`.

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 Types

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

```
> a <- c("bonjour", "au revoir")
> b <- c(TRUE, FALSE, TRUE, TRUE)
> c <- c(1.234, 12.34, 123.4, 1234)
> d <- matrix( rpois(12, 2), 4, 3)
> e <- list(un = a, deux = b)
> f <- data.frame(b, c)
> g <- factor( c("F", "M", "F", "F") )
> typeof(a) # chaînes de caractères
[1] "character"
```

```
> typeof(b) # valeurs vrai / faux
[1] "logical"
```

```
> typeof(c) # nombres "à virgule"
[1] "double"
```

```
> typeof(d) # nombres entiers
[1] "integer"
```

```
> typeof(e) # liste
[1] "list"
```

```
> typeof(f) # data frame = liste
[1] "list"
```

```
> typeof(g) # facteur = entiers (!)
[1] "integer"
```

Il y a deux types de variables numériques : `double` (nombres « à virgule », en formant 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(c > 100))
[1] "integer"
```

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

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

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

```
> str(a) # chaines de caractères
chr [1:2] "bonjour" "au revoir"
```

```
> str(b) # vrai / faux
logi [1:4] TRUE FALSE TRUE TRUE
```

```
> str(c) # "à virgule"
num [1:4] 1.23 12.34 123.4 1234
```

```
> str(d) # entiers (matrice 4x3)
int [1:4, 1:3] 2 0 5 3 1 1 2 2 1 0 ...
```

```
> str(e) # liste
List of 2
 $ un  : chr [1:2] "bonjour" "au revoir"
 $ deux: logi [1:4] TRUE FALSE TRUE TRUE
```

```
> str(f) # data frame
'data.frame': 4 obs. of 2 variables:
 $ b: logi TRUE FALSE TRUE TRUE
 $ c: num 1.23 12.34 123.4 1234
```

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


1.2 Attributs

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

```
> names(c) <- c("elt1", "elt2", "elt3", "elt4")
> c
      elt1      elt2      elt3      elt4
  1.234   12.340   123.400  1234.000
```

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

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

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

Les data frames et les facteurs ont également des attributs :

```
> attributes(f)
$names
[1] "b" "c"

$class
[1] "data.frame"

$row.names
[1] 1 2 3 4
```

```
> attributes(g)
$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(d, "dim")
[1] 4 3
```

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

1.3 Mieux examiner les objets

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 Cas de la matrice

```
> dput(d)
structure(c(2L, 0L, 5L, 3L, 1L, 1L, 2L, 2L, 1L, 0L, 2L, 3L), .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 Cas du data frame :

```
> dput(f)
structure(list(b = c(TRUE, FALSE, TRUE, TRUE), c = c(1.234, 12.34,
123.4, 1234)), class = "data.frame", row.names = c(NA, -4L))
```

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 Cas du facteur :

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

```
> levels(g)
[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 :

```
> h <- c(2L, 1L, 1L, 2L)
> attributes(h) <- list(levels = c("L1", "L2"), class = "factor")
> h
[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 ))
@55ab147dcce0 13 INTSXP g0c0 [REF(65535)] 1 : 10 (compact)

> .Internal(inspect( c(0.1, 0.2) ))
@55ab14f75688 14 REALSXP g0c2 [] (len=2, tl=0) 0.1,0.2

> a <- c(0.1, 0.2)
> .Internal(inspect( a ))
@55ab14f3e908 14 REALSXP g0c2 [REF(2)] (len=2, tl=0) 0.1,0.2

> names(a) <- c("A", "B")
> .Internal(inspect( a ))
@55ab14a6acd8 14 REALSXP g0c2 [REF(1),ATT] (len=2, tl=0) 0.1,0.2
ATTRIB:
  @55ab1453d280 02 LISTSXP g0c0 [REF(1)]
    TAG: @55ab13ce2570 01 SYMSXP g0c0 [MARK,REF(19194),LCK,gp=0x4000] "names" (has value)
    @55ab14a6aad8 16 STRSXP g0c2 [REF(2)] (len=2, tl=0)
      @55ab1411eec8 09 CHARSEX g0c1 [REF(3),gp=0x60] [ASCII] [cached] "A"
      @55ab1411f600 09 CHARSEX g0c1 [REF(3),gp=0x60] [ASCII] [cached] "B"

> .Internal(inspect( h ))
@55ab16ef19a8 13 INTSXP g0c2 [OBJ,MARK,REF(4),ATT] (len=4, tl=0) 2,1,1,2
ATTRIB:
  @55ab16834f88 02 LISTSXP g0c0 [MARK,REF(1)]
    TAG: @55ab13ce26c0 01 SYMSXP g0c0 [MARK,REF(65535),LCK,gp=0x6000] "levels" (has value)
    @55ab16ef1aa8 16 STRSXP g0c2 [MARK,REF(65535)] (len=2, tl=0)
      @55ab1749f930 09 CHARSEX g0c1 [MARK,REF(8),gp=0x60] [ASCII] [cached] "L1"
      @55ab1749f888 09 CHARSEX g0c1 [MARK,REF(8),gp=0x60] [ASCII] [cached] "L2"
    TAG: @55ab13ce2ab0 01 SYMSXP g0c0 [MARK,REF(23323),LCK,gp=0x6000] "class" (has value)
    @55ab1749f818 16 STRSXP g0c1 [MARK,REF(65535)] (len=1, tl=0)
      @55ab13d85e08 09 CHARSEX g0c1 [MARK,REF(280),gp=0x61] [ASCII] [cached] "factor"
```

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...`

Chapitre 2

Bases de C++

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

```
#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 bibliothèques. La bibliothèque `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 bibliothèque `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.

2.1 Nécessité des types

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 chaînes 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 être 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.

2.2 Les entiers

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++).

```
#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 parcimonie (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.

2.2.1 Entiers non signés.

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 signés de 16 bits.

```
#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
```

2.2.2 Notre première fonction « non void »

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

```

//[[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.

```

//[[Rcpp::export]]
int cast_to_int(int x) {
    return x;
}

```

2.2.3 Initialisation des variables

Il est nécessaire d'initialiser les variables.

```

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

```

Testons :

```

> 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;`

2.3 Les flottants

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

```

#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";
}

```


2.3.1 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 $a2^b$, où 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"
```

2.3.2 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*.

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

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

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

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

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

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

```
#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
```

2.4 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 faut taper 0. ou 0.0, etc. La plupart du temps le compilateur corrige ces petites erreurs.

2.5 Opérateurs arithmétiques

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

```
#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'incrémentement ++ et de décrémentation --.

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

2.6 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).

```
#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 exemple 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 ?

2.7 Booléens

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

```
// [[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.

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

2.8 Tableaux de taille fixe

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

```
#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 débogage « 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).

2.9 Contrôle du flux d'exécution : les boucles

2.9.1 Boucles for

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

```
#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 :

```
#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.

```
#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.

2.9.2 continue et break

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

```
#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.

```
#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)
AAAAAAAAAAAAAAAA
```

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

2.9.3 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.

```
#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.

```
#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 cppreference.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.

2.10 Contrôle du flux d'exécution : les alternatives

2.10.1 if et if else

Cela fonctionne tout à fait comme en R ; la x

```
// [[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
```

2.10.2 switch

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

```
#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++

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.**

```
#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

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeroes(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)
> count_zeroes(a);
[1] 10017

```

```

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

```

```

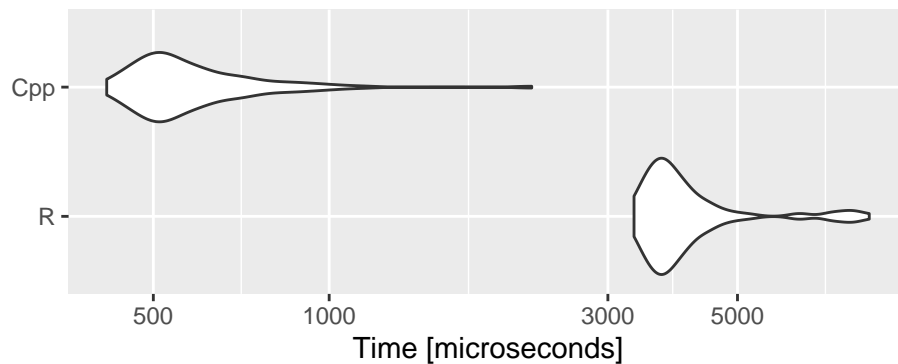
> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp = count_zeroes(a))
> mbm
Unit: microseconds
expr      min       lq      mean    median      uq     max  neval
  R 3325.529 3602.4975 4188.0514 3821.9055 4171.792 8405.202   100
 Cpp 416.106 492.6505 601.7615 537.4245 631.496 2222.019   100

```

```

> 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

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).

```

#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
IntegerVector zeros(int n) {
  IntegerVector R(n);
  return R;
}
//[[Rcpp::export]]
IntegerVector whatever(int n, int a) {
  IntegerVector R(n, a);
  return R;
}
//[[Rcpp::export]]
IntegerVector uninitialized(int n) {
  IntegerVector R = no_init(n);
  return R;
}
//[[Rcpp::export]]
IntegerVector nombres_fetiches() {
  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] 389376816      21931 389376760      21931 389376704

```

```

> nombres_fetiches()
[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).

```

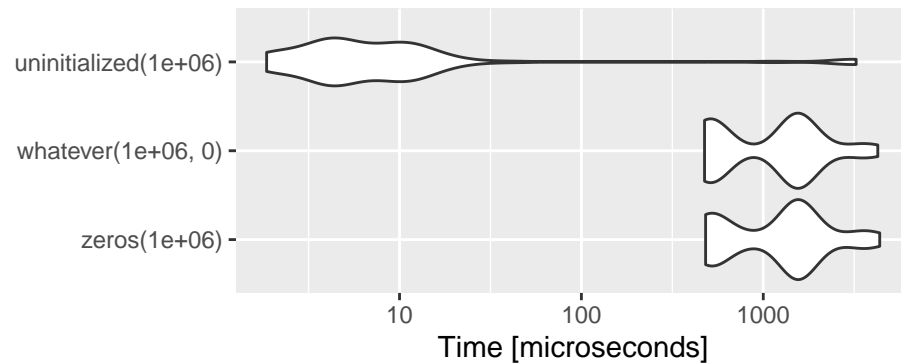
> mbm <- microbenchmark::microbenchmark(zeros(1e6), whatever(1e6, 0), uninitialized(1e6))
> mbm
Unit: microseconds
      expr      min       lq      mean   median       uq      max neval
zeros(1e+06) 481.904 539.4305 1430.3807 1488.565 1585.9675 4368.844   100
whatever(1e+06, 0) 475.458 511.9295 1323.9939 1468.812 1546.6785 4263.731   100
uninitialized(1e+06)  1.859   3.8595  144.3751    5.420  11.5245 3241.166   100

```

```

> ggplot2::autoplot(mbm)

```



3.2.1 Gestions des valeurs manquantes

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

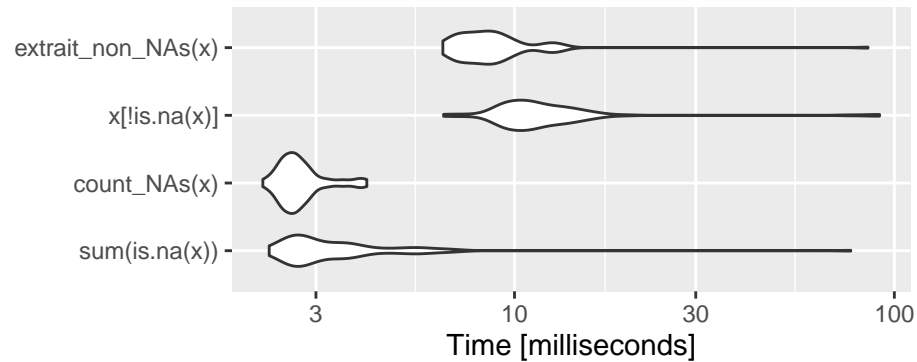
```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int count_NAs(NumericVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    re += NumericVector::is_na(x[i]);
  }
  return(re);
}
// [[Rcpp::export]]
NumericVector extrait_non_NAs(NumericVector x) {
  int nb_nas = count_NAs(x);
  int n = x.size();
  NumericVector R(n - nb_nas);
  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)), count_NAs(x), x[!is.na(x)], extrait_non_NAs(x) )
> mbm
Unit: milliseconds
      expr      min       lq      mean     median       uq      max neval
sum(is.na(x)) 2.259593 2.659893 4.150552 2.937179 3.708989 76.813362   100
count_NAs(x)  2.174587 2.480646 2.752600 2.640607 2.854555  4.084991   100
```

```
x[!is.na(x)] 6.503074 9.782958 13.645441 10.919586 13.062088 91.414625 100
extrait_non_NAs(x) 6.475742 7.443319 9.483934 8.451729 9.436607 85.223911 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.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector creer1() {
  NumericVector x = NumericVector::create(Named("un") = 10, Named("deux") = 20);
  return x;
}
```

Application :

```
> a <- creer1()
> a
  un deux
  10   20
```

Une syntaxe plus dense est possible :

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector creer2() {
  NumericVector x = NumericVector::create(_["un"] = 10, _["deux"] = 20);
  return x;
}
```

Cela produit le même résultat.

```
> creer2()
un deux
10 20
```

3.3.2 Obtenir les noms d'un vecteur

Et voici comment obtenir les noms d'un vecteur.

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

Utilisons cette fonction:

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

```
> b <- seq(0,1,length=6)
> noms(b)
Error in noms(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`. La solution est d'attraper le résultat de `x.names()` dans un `SEXP`, et de tester son type avec `TYPEOF`.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector noms2(NumericVector x) {
  SEXP R = x.names();
  if( TYPEOF(R) == STRSXP )
    return R;
  else
    return CharacterVector(0);
}
```

```
> noms2(a)
[1] "un" "deux"
```

```
> noms2(b)
character(0)
```

3.3.3 Accéder aux éléments par leurs noms

On utilise toujours la syntaxe `x[]` :

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
double get_un(NumericVector x) {
  if(x.containsElementNamed("un"))
    return x["un"];
  else
    stop("Pas d'élément 'un'");
}
```

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

```
> get_un(a)
[1] 10
```

```
> get_un(b)
Error in get_un(b): Pas d'élément 'un'
```

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.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
std::string le_type(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";
  }
}
```

Ça n'est généralement pas une bonne idée d'écrire des fonctions génériques comme celle-ci. 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 `noms`, en lui passant un vecteur non nommé.

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

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`. La solution est d'attraper le résultat de `x.names()` dans un `SEXP`, et de tester son type avec `TYPEOF`.

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector noms2(NumericVector x) {
  SEXP R = x.names();
  if( TYPEOF(R) == STRSXP )
    return R;
  else
    return CharacterVector(0);
}
```

```
> noms2(a)
[1] "un" "deux"
```

```
> noms2(b)
character(0)
```

3.4.2 Exemple : énumérer 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).

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void enumere(NumericVector x) {
  SEXP r0 = x.names();
  if(TYPEOF(r0) != STRSXP) {
    Rcout << "Pas de noms\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";
  }
}
```



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

3.5 Facteurs

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
CharacterVector niveaux(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("Pas d'attribut 'levels'");
    default:
      stop("Attribut 'levels' de type inconvenant");
  }
}
```

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

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

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

```
> un_facteur()
[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.

```

#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
NumericVector get_alpha_list(List x) {
  if( x.containsElementNamed("alpha") ) {
    SEXP R = x["alpha"];
    if( TYPEOF(R) != REALSXP )
      stop("elt alpha n'est pas un 'NumericVector'");
    return R;
  } else
    stop("Pas d'elt alpha");
}

```

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

```

#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
List cree_liste() {
  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;
}

```

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 : il suffit la plupart de transformer le résultat en liste (avec `as.data.frame`) une fois qu'on l'a obtenu.

Chapitre 4

Sucre syntaxique

La fonction R suivante fait un usage abondant de la vectorisation.

```
bonne.vectorisation <- function(x, y) {  
  z <- 3*x + y  
  if(any(x > 1))  
    z <- z*2;  
  sum( ifelse(z > 0, z, y) )  
}
```

```
> set.seed(1); x <- rnorm(10); y <- rnorm(10)  
> bonne.vectorisation(x,y)  
[1] 26.99719
```

La transcription en C++ devrait faire intervenir trois boucles ; c'est un peu fastidieux. Le sucre syntaxique ajouté par les fonctions dites *Rcpp sugar* permet d'éviter de les écrire.

```
#include <Rcpp.h>  
using namespace Rcpp;  
// [[Rcpp::export]]  
double fonction_sucree(NumericVector x, NumericVector y) {  
  NumericVector z = 3*x + y;  
  if( is_true(any(x > 1)) )  
    z = z*2;  
  return sum(ifelse(z > 0, z, y));  
}
```

```
> fonction_sucree(x,y)  
[1] 26.99719
```

4.1 Efficacité

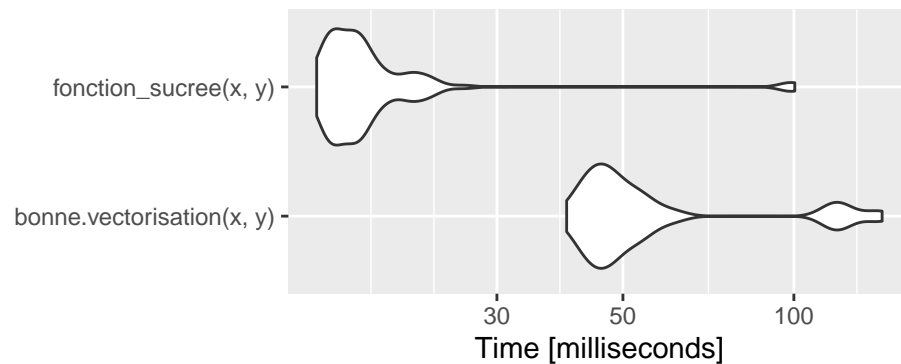
Grâce à une implémentation soignée, les fonctions Rcpp sugar sont redoutablement efficaces.

```

> x <- rnorm(1e6)
> y <- rnorm(1e6)
> mbm <- microbenchmark::microbenchmark( bonne.vectorisation(x,y), fonction_sucree(x,y) )
> mbm
Unit: milliseconds
      expr      min       lq      mean    median      uq      max  neval
bonne.vectorisation(x, y) 39.80109 44.38909 60.23114 48.22533 54.59131 143.0204   100
fonction_sucree(x, y) 14.44535 15.45062 19.72121 16.78379 18.25459 100.3774   100

> ggplot2::autoplot(mbm)

```



Revenons cependant à notre exemple de comptage de 0 dans un vecteur.

```

#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
int count_zeroes_sugar(IntegerVector x) {
  return sum(x == 0);
}

```

Comparons cette solution à celle proposée au chapitre précédent.

```

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

```

```

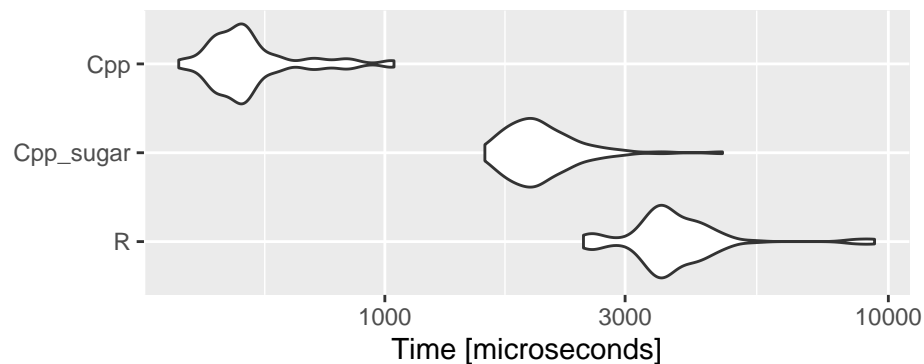
> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp_sugar = count_zeroes_sugar(a), Cpp = count_zeroes(a) )
> mbm
Unit: microseconds
      expr      min       lq      mean    median      uq      max  neval
R      2479.845 3394.1375 3877.0203 3639.3450 4115.249 9390.263   100
Cpp_sugar 1580.004 1806.6085 2074.0855 1979.1575 2217.605 4684.446   100
Cpp      389.618 469.4635 553.0272 516.1225 554.676 1042.642   100

```

```

> ggplot2::autoplot(mbm)

```



On le voit, la fonction qui utilise le sucre syntaxique, bien que toujours très efficace, n'atteint pas toujours la performance d'une fonction plus rustique.

4.2 Un exemple de la Rcpp Gallery

Un exemple tiré de la Rcpp Gallery <http://gallery.rcpp.org/articles/simulating-pi/> (estimation de π par la méthode de Monte-Carlo) et la variante avec une boucle.

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double pi_sugar(const int N) {
  NumericVector x = runif(N);
  NumericVector y = runif(N);
  // NumericVector d = sqrt(x*x + y*y);
  NumericVector d = x*x + y*y;
  return 4.0 * sum(d < 1.0) / N;
}

// [[Rcpp::export]]
double pi_boucle(const int N) {
  int S = 0;
  for(int i = 0; i < N; i++) {
    double x = R::runif(0, 1);
    double y = R::runif(0, 1);
    S += (x*x + y*y) < 1.0; // cast implicite bool vers int
  }
  return (4.0 * (double) S / (double) N);
}
```

```
> pi_sugar(1e6)
[1] 3.14262
```

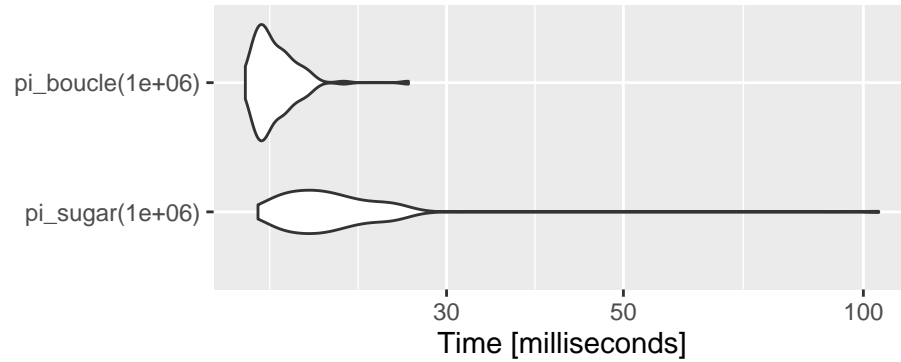
```
> pi_boucle(1e6)
[1] 3.141956
```

```
> mbm <- microbenchmark::microbenchmark(pi_sugar(1e6), pi_boucle(1e6))
> mbm
```

Unit: milliseconds

expr	min	lq	mean	median	uq	max	neval
pi_sugar(1e+06)	17.40162	19.23264	22.08418	20.85463	23.06672	104.53050	100
pi_boucle(1e+06)	16.77738	17.46614	18.30804	17.94709	18.77209	26.87273	100

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



Chapitre 5

Exemple : Metropolis-Hastings

5.1 L'algorithme

L'algorithme de Metropolis-Hastings permet de faire des tirages aléatoires dans une loi de densité proportionnelle à une fonction $\pi(x)$ positive – il n'y a pas besoin que $\int \pi(x)dx = 1$, autrement dit, on n'a pas besoin de connaître la constante de normalisation.

Nous présentons tout d'abord la notion de marche aléatoire, puis l'algorithme de Metropolis-Hastings.

5.1.1 Marche aléatoire

Une suite de valeurs aléatoires $x_1, x_2, \dots \in \mathbb{R}^d$ est une marche aléatoire¹ si chaque point x_{t+1} est tiré dans une loi dont la densité ne dépend que x_t . On pourra noter $q(x|x_t)$ cette densité.

Un exemple simple est la marche aléatoire gaussienne : la densité $q(x|x_t)$ est la densité d'une loi normale de variance $\sigma^2 I_d$ et d'espérance x_t . Cela revient à dire que

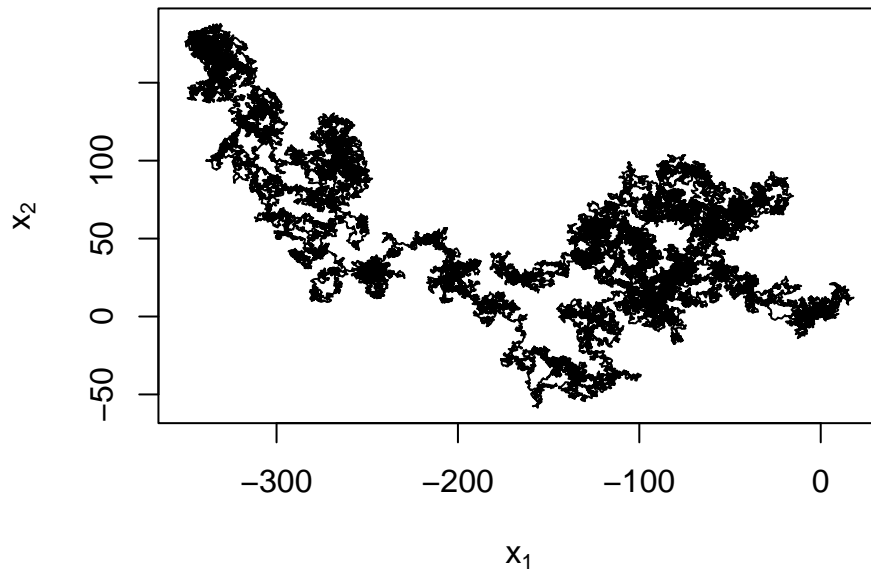
$$x_{t+1} = x_t + z$$

avec z tiré dans une loi normale centrée de variance $\sigma^2 I$.

La fonction suivante permet d'illustrer ceci avec $d = 2$. Elle réalise B étapes d'une marche aléatoire dont le point de départ est $x_1 = (0, 0)$. Le résultat est présenté sous la forme d'une matrice à B lignes et deux colonnes. Le paramètre sd permet de spécifier la valeur de σ .

```
Marche <- function(B, sd) {  
  R <- matrix(0.0, nrow = B, ncol = 2)  
  x <- R[1,]  
  for(b in 2:B) {  
    x <- x + rnorm(2, 0, sd = sd)  
    R[b, ] <- x  
  }  
  return(R);  
}  
X <- Marche(5e4, sd = 0.8)  
plot(X[,1], X[,2], xlab = expression(x[1]), ylab = expression(x[2]), type = "l")
```

1. On pourra trouver d'autres définitions pour « marche aléatoire ». La définition donnée ici est celle d'une chaîne de Markov dite « homogène » ou parfois « stationnaire ».



5.1.2 L'algorithme

Voici l'algorithme pour faire des tirages dans une loi de densité proportionnelle à une fonction positive $\pi(x)$, définie sur \mathbb{R}^d . On part d'un point x_1 arbitraire, ou bien tiré au hasard dans une loi bien choisie. Supposons qu'on a $x_t \in \mathbb{R}^d$. On va tirer x_{t+1} en s'aidant d'une marche aléatoire de la façon suivante :

1. On génère une valeur y en faisant "un pas de marche aléatoire depuis x_t ", autrement dit en tirant y dans la loi de densité $q(x|x_t)$.

2. On calcule

$$\rho = \frac{\pi(y)q(x_t|y)}{\pi(x_t)q(y|x_t)}.$$

3. Si $\rho \geq 1$, on pose $x_{t+1} = y$; sinon, $x_{t+1} = y$ avec probabilité ρ et $x_{t+1} = x_t$ avec probabilité $1 - \rho$.

La valeur y s'appelle « valeur proposée »; l'étape 3 consiste à décider si on accepte ou non la proposition. En pratique, on peut la réaliser ainsi

- On tire u dans la loi uniforme $U(0, 1)$
- Si $u < \rho$ on pose $x_{t+1} = y$ (on accepte y), et sinon $x_{t+1} = x_t$.

5.1.2.1 Cas particulier d'une marche symétrique

si pour tous x et y on a $q(x|y) = q(y|x)$ (la probabilité de faire un pas de y à x est la même que celle de faire un pas de x à y ; c'est le cas de la marche gaussienne donnée en exemple), alors on a simplement

$$\rho = \frac{\pi(y)}{\pi(x_t)}.$$

Dans ce cas, la valeur proposée y est toujours acceptée quand $\pi(y) > \pi(x_t)$, c'est-à-dire quand la marche aléatoire propose un point où la densité est plus grande qu'au point actuel.

5.1.2.2 Les propriétés du résultat

Si t est assez grand, alors x_t est approximativement de loi de densité $\pi(x)$ (ou proportionnelle à $\pi(x)$).

On pourrait donc utiliser cette méthode avec $t = 4000$ (par exemple) pour générer une valeur dans la loi voulue, puis recommencer, etc. C'est très coûteux en temps de calcul ; en fait pour la plupart des applications on peut garder toutes les valeurs au-delà d'une certaine valeur de t . Elles ne sont pas indépendantes mais cela n'est pas très gênant.

L'opération, souvent nécessaire, qui consiste à supprimer les premières valeurs (par exemple les 4000 premières) s'appelle le *burn-in*. Si il est important que les valeurs échantillonnées soient indépendantes, on peut s'en approcher en ne gardant, par exemple, qu'une valeur toutes les 100 itérations. Cette opération s'appelle le *thinning*.

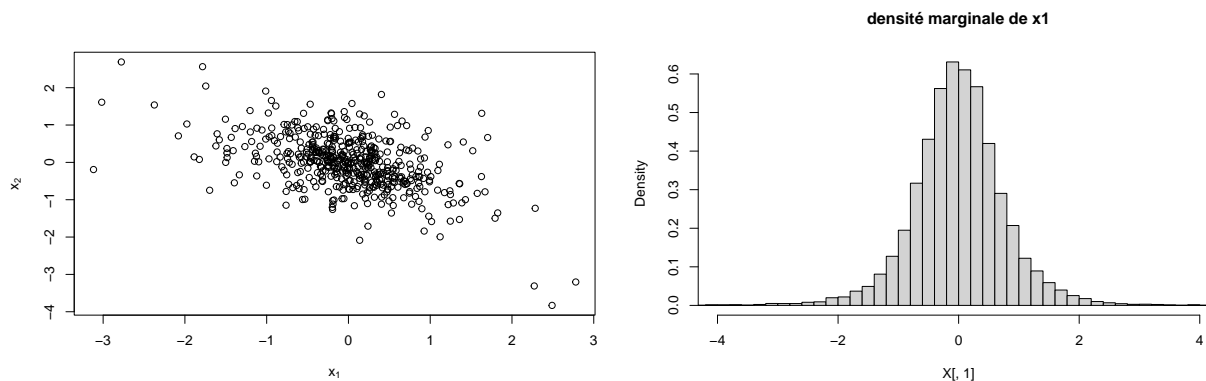
5.1.3 Application

On prend pour $x = (x_1, x_2)$, $\pi(x) = (1 + x_1^2 + x_1 x_2 + x_2^2)^{-3}$, et on utilise une marche aléatoire gaussienne comme celle présentée plus haut, qui est une marche symétrique : la formule simplifiée ci-dessus peut être utilisée.

L'implémentation en R n'est pas difficile:

```
PI <- function(x) (1 + x[1]**2 + x[1]*x[2] + x[2]**2)^(-3)
MH <- function(B, sd) {
  R <- matrix(0.0, nrow = B, ncol = 2)
  x <- R[1,]
  for(b in 2:B) {
    y <- x + rnorm(2, 0, sd = sd)
    rho <- PI(y) / PI(x)
    u <- runif(1)
    if(u < rho)
      x <- y
    R[b, ] <- x
  }
  return(R);
}
```

Voici un exemple de mise en œuvre :



5.2 Une version R

```

PI <- function(x) (1 + x[1]**2 + x[1]*x[2] + x[2]**2)^(-3)
MH <- function(B, sd) {
  R <- matrix(0.0, nrow = B, ncol = 2)
  x <- R[1,]
  for(b in 2:B) {
    y <- x + rnorm(2, 0, sd = sd)
    rho <- PI(y) / PI(x)
    u <- runif(1)
    if(u < rho)
      x <- y
    R[b, ] <- x
  }
  return(R);
}

```

5.3 Première version en C++

Une version obtenue en « traduisant » l'implémentation en R.

```

#include <Rcpp.h>
using namespace Rcpp;

double Pi(double x1, double x2) {
  return pow((1 + x1*x1 + x1*x2 + x2*x2), -3.0);
}

//[[Rcpp::export]]
NumericMatrix MH_cpp(int B, double sd) {
  NumericMatrix R(B, 2);
  double x1 = 0.0, x2 = 0.0;
  for(int b = 1; b < B; b++) {
    double y1 = x1 + R::rnorm(0, sd);
    double y2 = x2 + R::rnorm(0, sd);
    double rho = Pi(y1, y2) / Pi(x1, x2);
    double u = R::unif_rand();
    if(u < rho) {
      x1 = y1;
      x2 = y2;
    }
    R(b, 0) = x1;
    R(b, 1) = x2;
  }
  return R;
}

```

5.4 Une version améliorée

Il est souhaitable d'éviter de recalculer $Pi(x_1, x_2)$ à chaque tour de boucle, alors que cette valeur est déjà connue.

```

#include <Rcpp.h>
using namespace Rcpp;

double Pi(double x1, double x2) {
  return pow((1 + x1*x1 + x1*x2 + x2*x2),-3.0);
}
//[[Rcpp::export]]
NumericMatrix MH_cpp_1(int B, double sd, int burn = 0, int thin = 1) {
  NumericMatrix R(B, 2);
  thin = (thin < 1)?1:thin;
  double x1 = 0.0, x2 = 0.0;
  int b = 1;
  double pi_x = Pi(x1, x2);
  for(int k = 0; b < B ; k++) { // !! boucle exotique !!
    double y1 = x1 + R::rnorm(0,sd);
    double y2 = x2 + R::rnorm(0,sd);
    double pi_y = Pi(y1, y2) / Pi(x1, x2);
    double u = R::unif_rand();
    if(u * pi_x < pi_y) {
      x1 = y1;
      x2 = y2;
      pi_x = pi_y;
    }
    if(k > burn && (k % thin) == 0) {
      R(b,0) = x1;
      R(b,1) = x2;
      b++;
    }
    if((k % 1000) == 0) // toutes les mille itérations
      checkUserInterrupt();
  }
  return R;
}

```

Le paramètre `burn` permet de ne pas retenir les premières itérations ; le paramètre `thin` permet de ne retenir qu'une itération sur `thin` (pour réduire la dépendance entre les tirages successifs).

À noter : la fonction `checkUserInterrupt()` qui permet d'interrompre le programme en cas d'appui sur `ctrl + C`, ou sur le petit panneau `STOP` de R studio. On n'appelle pas cette fonction à chaque tour de boucle car elle est longue à exécuter !

À noter également : une boucle exotique, puisque la condition d'arrêt n'est pas sur le compteur de boucle `k`, mais sur `b`, qui est régulièrement incrémenté dans la boucle (mais, en général, pas à chaque tour).

Chapitre 6

Un peu plus de C++

6.1 Pointeurs

Les pointeurs sont un héritage de C. Il s'agit de « pointer » vers l'adresse d'un objet ou une variable. Un pointeur vers un `int` est un `int *`. Si `p` est un pointeur, `*p` est l'objet à l'adresse pointée par `p`.

Pour obtenir le pointeur vers `x`, on utilise `&x`.

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
// [[Rcpp::export]]
void pointeurs() {
    int x = 12;    // entier
    int * p;       // pointeur vers un entier (non initialisé !!)
    p = &x;        // p pointe vers x
    SHOW(x);
    SHOW(p);
    SHOW(*p);
    Rcpp::Rcout << "On ajoute 1 à x\n";
    x += 1;
    SHOW(x);
    SHOW(p);
    Rcpp::Rcout << "On ajoute 1 à *p\n";
    *p += 1;
    SHOW(x);
    SHOW(p);
}
```

```
> pointeurs()
x = 12
p = 0x7ffe099ceae4
*p = 12
On ajoute 1 à x
x = 13
p = 0x7ffe099ceae4
On ajoute 1 à *p
x = 14
p = 0x7ffe099ceae4
```

Nous avons déjà rencontré des pointeurs sans le savoir : les tableaux sont des pointeurs ! Plus précisément, si on a déclaré `int a[4]`, `a` pointe vers le premier élément du tableau, `a+1` vers le second, etc.

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
// [[Rcpp::export]]
void tableaux() {
  int a[4] = {10,20,30,40};
  SHOW(a);
  SHOW(a[0]);
  SHOW(*a);
  SHOW(a[2]);
  SHOW(a+2);
  SHOW(*a+2);
  SHOW(*(a+2));
}
```

```
> tableaux()
a = 0x7ffe099cead0
a[0] = 10
*a = 10
a[2] = 30
a+2 = 0x7ffe099cead8
*a+2 = 12
*(a+2) = 30
```

Notez la différence entre `a` et `a+2` : la taille des `int` est prise en compte dans le calcul.

En passant à une fonction un pointeur vers une variable, d'une part on permet la modification de cette variable, d'autre part on évite la copie de cette variable (intéressant quand on passe des objets de grande taille).

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;

// les arguments sont des pointeurs vers des entiers
inline void swap(int * x, int * y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

// [[Rcpp::export]]
void demonstrate_swap(int a, int b) {
  if(a > b) swap(&a,&b); // &a = pointeur vers a ...
  SHOW(a);
  SHOW(b);
}
```

Attention, la fonction `swap` ne peut pas être exportée vers R : les objets de R ne peuvent être transformés en `int *`, en pointeurs vers des entiers. Elle est destinée à n'être utilisée que dans notre code C++.

Le mot clef `inline` ci-dessus indique au compilateur qu'on désire que la fonction soit insérée dans le code aux endroits où elle est utilisée. Cela économise un appel de fonction...

```
> demonstrate_swap(12, 14)
a = 12
b = 14
```

```
> demonstrate_swap(14, 12)
a = 12
b = 14
```

6.2 Références

Le mécanisme derrière les références est similaire à celui des pointeurs ; la syntaxe est plus simple et permet d'éviter de promener des étoiles dans tout le code. Une référence à un entier est de type `int &`, et peut être manipulée directement comme un `int`. On l'initialise à l'aide d'une autre variable de type `int`.

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
// [[Rcpp::export]]
void references() {
    int x = 12;    // entier
    int &y = x;    // référence à x
    SHOW(x);
    SHOW(y);
    SHOW(&x);
    SHOW(&y);
    Rcpp::Rcout << "On ajoute 1 à x\n";
    x += 1;
    SHOW(x);
    SHOW(y);
    Rcpp::Rcout << "On ajoute 1 à y\n";
    y += 1;
    SHOW(x);
    SHOW(y);
}
```

```
> references()
x = 12
y = 12
&x = 0x7ffe099ceae4
&y = 0x7ffe099ceae4
On ajoute 1 à x
x = 13
y = 13
On ajoute 1 à y
x = 14
y = 14
```

C'est surtout utile pour spécifier les arguments d'une fonction. Voici ce que devient notre fonction `swap`:

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
```

```
// les arguments sont des références à des entiers
inline void swap2(int & x, int & y) {
    int tmp = x;
    x = y;
    y = tmp;
}

// [[Rcpp::export]]
void demonstrate_swap2(int a, int b) {
    if(a > b) swap2(a,b); // on passe directement x et y
    SHOW(a);
    SHOW(b);
}
```

Cette fonction swap2 ne peut pas plus être exportée que la fonction swap définie plus haut.

```
> demonstrate_swap(12, 14)
a = 12
b = 14
```

```
> demonstrate_swap(14, 12)
a = 12
b = 14
```

6.2.1 Autres exemples

Un appel `divise_par_deux(x)` est équivalent à `x /= 2`. L'utilisation de cette fonction ne simplifie pas vraiment le code, mais ce mécanisme pourrait être utile pour des opérations plus complexes:

```
void divise_par_deux(int & x) {
    x /= 2;
}

// [[Rcpp::export]]
int intlog2(int a) {
    int k = 0;
    while(a != 0) {
        divise_par_deux(a);
        k++;
    }
    return k-1;
}
```

```
> intlog2(17)
[1] 4
```

Une autre utilité est de renvoyer plusieurs valeurs. Bien sûr quand on utilise Rcpp on peut manipuler des vecteurs ou des listes, mais utiliser des variables passées par référence pour récupérer les résultats d'un calcul est souvent très commode.


```

#include <Rcpp.h>
#include <cmath>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;

bool eq_quadratique(double a, double b, double c, double & x1, double & x2) {
    double delta = b*b - 4*a*c;
    if(delta < 0)
        return false;
    double sqrt_delta = std::sqrt(delta);
    x1 = (-b - sqrt_delta)/(2*a);
    x2 = (-b + sqrt_delta)/(2*a);
    return true;
}
//[[Rcpp::export]]
void deuxieme_degre(double a, double b, double c) {
    double x1, x2;
    bool solvable = eq_quadratique(a, b, c, x1, x2);
    if(solvable) {
        SHOW(x1);
        SHOW(x2);
    } else {
        Rcpp::Rcout << "Pas de solution\n";
    }
}

```

```

> deuxieme_degre(1,-5,6)
x1 = 2
x2 = 3

```

```

> deuxieme_degre(1,0,1)
Pas de solution

```

6.3 Les objets de Rcpp sont passés par référence

La fonction suivante le démontre : les vecteurs de Rcpp sont passés par référence (ou sont des pointeurs, comme vous préférez). Ceci permet un grand gain de temps, en évitant la copie des données, mais a aussi des effets potentiellement indésirables.

```

#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector doubler(Rcpp::NumericVector x) {
    int n = x.size();
    for(int i = 0; i < n; i++) {
        x[i] *= 2;
    }
    return x;
}

```

```

> x <- seq(0,1,0.1);
> x
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

```

```
> doubler(x)
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> x # oops
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

De plus, quand on copie un objet en R, comme ci-dessous avec `y <- x`, la recopie des données n'a pas lieu dans un premier temps. Au début, `y` pointe vers le même objet que `x` ; la recopie n'aura lieu que si on modifie `y`. Quand on utilise Rcpp ce mécanisme n'est plus actif:

```
> x <- seq(0,1,0.1);
> y <- x
> doubler(y)
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> x
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> y
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

La prudence est de mise! Notez qu'on a aussi le comportement suivant:

```
> x <- 1:5;
> x
[1] 1 2 3 4 5
```

```
> doubler(x)
[1] 2 4 6 8 10
```

```
> x
[1] 1 2 3 4 5
```

Utiliser `typeof(x)` pour résoudre cette énigme.

À supposer que ce comportement soit indésirable, comment y remédier ?

```
#include <Rcpp.h>
// Création d'un vecteur (initialisé à 0)
// [[Rcpp::export]]
Rcpp::NumericVector doubler2(Rcpp::NumericVector x) {
  int n = x.size();
  // contrairement à ce qu'on pense, ceci ne copie pas x
  Rcpp::NumericVector y = x;
  for(int i = 0; i < n; i++) y[i] *= 2;
  return y;
}

// [[Rcpp::export]]
Rcpp::NumericVector doubler3(Rcpp::NumericVector x) {
  int n = x.size();
  // il faut utiliser clone
```

```
Rcpp::NumericVector y = Rcpp::clone(x);
for(int i = 0; i < n; i++) y[i] *= 2;
return y;
}
```

```
> x <- seq(0,1,0.1); invisible(doubler2(x)); x
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> x <- seq(0,1,0.1); invisible(doubler3(x)); x
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

On peut également modifier « en place » les éléments d'une liste:

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void increment_alpha(List x) {
  if( x.containsElementNamed("alpha") ) {
    SEXP R = x["alpha"];
    if( TYPEOF(R) != REALSXP )
      stop("elt alpha n'est pas un 'NumericVector'");
    NumericVector Alpha(R);
    Alpha = Alpha+1; // sugar
  } else
    stop("Pas d'elt alpha");
}
```

Exemple:

```
> x <- list( alpha = c(0.1,7), beta = 1:4)
> x
$alpha
[1] 0.1 7.0

$beta
[1] 1 2 3 4
```

```
> increment_alpha(x)
> x
$alpha
[1] 1.1 8.0

$beta
[1] 1 2 3 4
```

6.4 Surcharge

En C++ deux fonctions peuvent avoir le même nom, à condition que le type des arguments soit différent, ce qui permet au compilateur de les différencier. À noter : le type retourné par la fonction n'est pas pris en compte dans ce mécanisme.

Ceci s'appelle la *surcharge* des fonctions. On a également une surcharge des opérateurs (comme +, <<, etc).

Attention ! On ne peut pas exporter des fonctions surchargées avec Rcpp. Il faut donc maintenir ce mécanisme dans la partie strictement C++ de votre code.

Voici un exemple de fonction surchargée.

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

double d2(double a, double b) {
    return (a*a + b*b);
}

int d2(int a, int b) {
    return (a*a + b*b);
}

//[[Rcpp::export]]
void exemple_d2() {
    double x = 1.0;
    double y = 2.4;
    double z = d2(x, y);
    Rcout << "d2(x,y) = " << z << std::endl;
    int u = 1;
    int v = 4;
    int w = d2(u, v);
    Rcout << "d2(u,v) = " << w << std::endl;
}

> exemple_d2()
d2(x,y) = 6.76
d2(u,v) = 17
```

On peut également avoir des fonctions qui ont le même nom, et un nombre différent d'arguments.

6.5 Templates

Les templates facilitent la réutilisation de code avec des types différents. Prendre le temps d'utiliser des méthodes propres pour éviter les copier-coller dans le code est de toute première importance quand il s'agit de « maintenir » le code.

6.5.1 Fonctions polymorphes

L'exemple ci-dessous définit un template d2 ; lors de la compilation de la fonctions addition_int et addition_double ce template sera instancié avec TYPE = int puis avec TYPE = double.

```
#include <Rcpp.h>
template<typename TYPE>
TYPE d2(TYPE a, TYPE b) {
    return a*a + b*b;
}
```

```

//[[Rcpp::export]]
void exemple_template_d2() {
    double x = 1.0;
    double y = 2.4;
    double z = d2(x, y);
    Rcpp::Rcout << "d2(x,y) = " << z << std::endl;
    int u = 1;
    int v = 4;
    int w = d2(u, v);
    Rcpp::Rcout << "d2(u,v) = " << w << std::endl;
}

```

Exemple:

```

> exemple_template_d2()
d2(x,y) = 6.76
d2(u,v) = 17

```

Si le compilateur peinait à trouver par quoi il faut remplacer TYPE, il serait possible de préciser en tapant par exemple `d2<float>`. Ce template peut fonctionner dès que les opérateurs `+` et `*` sont définis.

```

#include <Rcpp.h>
using namespace Rcpp;

template<typename TYPE>
TYPE d2(TYPE a, TYPE b) {
    return a*a + b*b;
}
//[[Rcpp::export]]
NumericVector exemple_template_d2bis(NumericVector x, NumericVector y) {
    return d2(x,y);
}

```

```

> exemple_template_d2bis( c(1,2), c(3,1) )
[1] 10 5

```

On peut donner un autre exemple en faisant un template pour la fonction `swap` que nous avons écrite il y a quelques chapitres.

```

template<typename TYPE>
void swap(TYPE & a, TYPE & b) {
    TYPE tmp = a;
    a = b;
    b = tmp;
}

```

Note : n'utilisez pas ce template dans votre code, C++ a déjà un template `std::swap`, qui est mieux fait que celui-ci ! (Il évite la copie). Voir <https://en.cppreference.com/w/cpp/algorithm/swap>

6.5.2 Laisser le compilateur s'occuper des types

Une des utilités des templates est de permettre de ne pas trop se soucier de types « compliqués » comme ceux des fonctions. Ci-dessous on définit une fonction `num_derive` qui donne une approximation de la dérivée de

son premier argument.

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
#define epsilon 0.001

template<typename TYPE, typename FTYPE>
TYPE num_derive(FTYPE f, TYPE a) {
    return (f(a+epsilon) - f(a-epsilon))/(2.0*epsilon);
}

double square(double a) {
    return a*a;
}

//[[Rcpp::export]]
void test1(double x) {
    double fx = square(x);
    double dfx = num_derive(square, x);
    SHOW(fx);
    SHOW(dfx);
}

> test1(3)
fx = 9
dfx = 6
```

On voit que num_derive prend deux arguments, une fonction f et un point a, et donne une approximation de $f'(a)$ en calculant

$$\frac{f(a+\epsilon) - f(a-\epsilon)}{2\epsilon}.$$

Quel est le type d'une fonction ?! Nous laissons au compilateur le soin de remplacer FTYPE par la bonne valeur. Dans notre exemple, lors d'instanciation du template, TYPE est remplacé par double et FTYPE par double (*) (double) (pointeur vers une fonction qui renvoie un double et dont l'argument est un double). Un code équivalent serait

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
#define epsilon 0.001
double num_derive(double (*f)(double), double a) {
    return (f(a+epsilon) - f(a-epsilon))/(2*epsilon);
}

double square(double a) {
    return a*a;
}

//[[Rcpp::export]]
void test_derive(double x) {
    double fx = square(x);
    double dfx = num_derive(square, x);
    SHOW(fx);
    SHOW(dfx);
}
```

```
> test_derive(3)
fx = 9
dfx = 6
```

6.6 C++11

Le standard C++11 est maintenant accepté dans les packages R. Pour permettre la compilation avec les extensions offertes par C++11, vous devez faire :

```
> Sys.setenv("PKG_CXXFLAGS" = "-std=c++11")
```

6.6.1 Les boucles sur un vecteur et auto

Parmi les extensions offertes, les plus séduisantes sont sans doute : le mot-clef `auto`, qui laisse le compilateur deviner le type, quand le contexte le permet, de façon similaire à ce qui est fait dans les templates, et les boucles `for` qui parcourent un vecteur... comme en R.

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeroes3(Rcpp::IntegerVector x) {
  int re = 0;
  for(auto a : x) {
    if(a == 0) ++re;
  }
  return re;
}
```

```
> set.seed(1); a <- sample(0:99, 1e6, TRUE)
> count_zeroes3(a);
[1] 10017
```

Si on veut pouvoir modifier la valeur des éléments du vecteur, il suffit d'utiliser une référence :

```
#include <Rcpp.h>
//[[Rcpp::export]]
void add_one(Rcpp::IntegerVector x) {
  for(auto & a : x) {
    a++;
  }
}
```

```
> a <- 1:10
> add_one(a)
> a
[1] 2 3 4 5 6 7 8 9 10 11
```

NOTE En utilisant une référence, on évite la copie. Même si on ne désire pas modifier les éléments du vecteur, cela peut modifier énormément les performances de la boucle en question !

TODO ajouter exemple !!!

6.6.2 Nombres spéciaux

C++11 propose aussi des fonctions pour tester si des nombres flottants sont finis ou non, sont NaN. Il est aussi plus facile de renvoyer un NaN ou une valeur infinie (c'était possible avec `std::numeric_limits`).

```
#include <Rcpp.h>
#include <cmath>          // !!! CE HEADER EST NÉCESSAIRE
//[[Rcpp::export]]
void tests(double x) {
  if(std::isnan(x)) Rcpp::Rcout << "NaN\n";
  if(std::isinf(x)) Rcpp::Rcout << "infini\n";
  if(std::isfinite(x)) Rcpp::Rcout << "fini\n";
}

//[[Rcpp::export]]
Rcpp::NumericVector specials() {
  Rcpp::NumericVector x(2);
  x[0] = NAN;
  x[1] = INFINITY;
  return x;
}
```

```
> tests( NA )
NaN
```

```
> tests( -Inf )
infini
```

```
> tests( pi )
fini
```

```
> specials();
[1] NaN Inf
```


Chapitre 7

Introduction à la librairie standard

Nous allons dans ce chapitre très parcellaire présenter rapidement quelques *containers* proposés par la librairie standard de C++.

7.1 Vecteurs

Les vecteurs `std::vector` de la librairie standard sont beaucoup plus performants que les vecteur de R. Il s'agit d'une classe « templatée »; on peut avoir des vecteurs de n'importe quels objets. Par exemple, un `std::vector<int>` est un vecteur d'entiers.

La fonction wrap de Rcpp permet de créer des vecteurs de R à partir de vecteurs standards: les `std::vector<int>` seront transformés en `IntegerVector`, les `std::vector<double>` seront transformés en `NumericVector`, et les `std::vector<std::string>` en `CharacterVector`.

7.1.1 `push_back`, `pop_back` et `resize`

Le premier intérêt des vecteurs standard est leur capacité à changer de longueur sans perte grâce à `push_back`, et rétrécir avec `pop_back`. Leur taille peut aussi être modifiée avec `resize`. L'exemple suivant déclare un vecteur de longueur 4, y ajoute un élément égal à 5.25, l'enlève, et redimensionne le vecteur à la taille de 8 éléments.

```
#include <Rcpp.h>
#include <vector>
using namespace Rcpp;
//[[Rcpp::export]]
void demo_vector_1() {
    std::vector<double> x(4);
    for(auto a : x) Rcout << a << " "; Rcout << "\n";
    x.push_back(5.25);
    for(auto a : x) Rcout << a << " "; Rcout << "\n";
    x.pop_back();
    for(auto a : x) Rcout << a << " "; Rcout << "\n";
    x.resize(8);
    for(auto a : x) Rcout << a << " "; Rcout << "\n";
}
```

Note : j'ai mis `#include <vector>` pour le principe. En fait l'inclusion de `Rcpp.h` suffit, car elle implique l'inclusion de nombreuses composantes de la librairie standard, dont celle-ci.

```
> demo_vector_1()
## 0 0 0 0
## 0 0 0 0 5.25
## 0 0 0 0
## 0 0 0 0 0 0 0
```

7.1.2 Contrôler la capacité avec `resize`

En pratique, les vecteurs standard allouent plus de mémoire que nécessaire ; on parle de « réserve » de mémoire. Quand la réserve est épuisée, un nouveau vecteur est créé avec une réserve plus importante, et l'ancien vecteur y est recopié. Cela représente une perte de temps, mais la réserve doublant de taille à chaque fois que cela se produit, cela ne peut pas se produire trop souvent. On peut anticiper sur les possibilités de changements de taille avec `reserve`.

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

template<typename T>
inline void affiche(const std::vector<T> & x) {
    Rcout << "adresse = " << &x[0] << ", ";
    Rcout << "size = " << x.size() << ", ";
    Rcout << "capacity = " << x.capacity() << "\n";
}

//[[Rcpp::export]]
void demo_vector_2(int n, bool res) {
    std::vector<int> x;
    std::cout << "vecteur vide :\n";
    affiche(x);

    if(res) {
        x.reserve(n);
        std::cout << "vecteur avec une réserve :\n";
        affiche(x);
    }

    for(int i = 0; i < n; i++) {
        x.push_back(3);
    }
    std::cout << "vecteur dans lequel on a 'poussé' " << n << " éléments :\n";
    affiche(x);
}
```

```
> demo_vector_2(4, TRUE)
## adresse = 0, size = 0, capacity = 0
## adresse = 0x55ab188782d0, size = 0, capacity = 4
## adresse = 0x55ab188782d0, size = 4, capacity = 4
```

```
> demo_vector_2(4, FALSE)
## adresse = 0, size = 0, capacity = 0
## adresse = 0x55ab17c71870, size = 4, capacity = 4
```

```
> demo_vector_2(9, FALSE)
## adresse = 0, size = 0, capacity = 0
## adresse = 0x55ab19b5fe40, size = 9, capacity = 16
```

Ceci permet par exemple d'extraire les éléments non NA d'un vecteur sans avoir à les compter avant:

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

//[[Rcpp::export]]
NumericVector extrait_non_NA(NumericVector x) {
    std::vector<double> R; // vecteur de doubles, vide par défaut
    R.reserve( x.size() ); // on prévoit de pouvoir étendre la taille de R jusqu'à x.size
    for(auto a : x) {
        if(!NumericVector::is_na(a))
            R.push_back(a);
    }
    return wrap(R);
}
```

7.1.3 move et emplace_back

En C++11 est apparu l'opérateur *move* qui permet d'éviter la copie. Nous allons l'illustrer ci-dessous sur des `std::string`.

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

//[[Rcpp::export]]
void moves() {
    std::string s1("Bonjour cher ami !");
    std::string s2 = s1; // copie
    Rcout << "s1 = " << s1 << " adresse = " << (void *) &s1[0] << "\n";
    Rcout << "s2 = " << s2 << " adresse = " << (void *) &s2[0] << "\n";
    std::string s3( std::move(s1) );
    Rcout << "s1 = " << s1 << "\n";
    Rcout << "s3 = " << s3 << " adresse = " << (void *) &s3[0] << "\n";
}
```

```
> moves()
## s1 = Bonjour cher ami ! adresse = 0x55ab18243960
## s2 = Bonjour cher ami ! adresse = 0x55ab1734b490
## s1 =
## s3 = Bonjour cher ami ! adresse = 0x55ab18243960
```

Note : Cet exemple pourrait échouer ; si la chaîne `s1` fait moins de 16 caractères, elle est copiée dans `s3` malgré l'opérateur `move`!

Il est donc plus efficace pour remplir un vecteur `V` d'utiliser `V.push_back(std::move(x))` que `V.push_back(x)` – si on n'a pas besoin de `x` après. La solution idéale est (normalement) l'utilisation d'`emplace_back` qui permet de construire directement l'objet en dernière position du vecteur.

```

#include <Rcpp.h>
#include <ctime>

//[[Rcpp::export]]
void temps_de_remplissage() {
    clock_t t = clock();
    for(int k = 0; k < 100; k++) {
        std::vector<std::string> V;
        V.reserve(1e5);
        for(int i = 0; i < 1e5; i++) {
            std::string s(50, 'a');
            V.push_back(s);
        }
    }
    t = clock() - t;
    Rcpp::Rcout << "V rempli en " << (double) t / CLOCKS_PER_SEC / 100. << " s\n";

    t = clock();
    for(int k = 0; k < 100; k++) {
        std::vector<std::string> V;
        V.reserve(1e5);
        for(int i = 0; i < 1e5; i++) {
            std::string s(50, 'a');
            V.push_back( std::move(s) );
        }
    }
    t = clock() - t;
    Rcpp::Rcout << "V rempli en " << (double) t / CLOCKS_PER_SEC / 100. << " s\n";

    t = clock();
    for(int k = 0; k < 100; k++) {
        std::vector<std::string> V;
        V.reserve(1e5);
        for(int i = 0; i < 1e5; i++) {
            V.emplace_back(50, 'a');
        }
    }
    t = clock() - t;
    Rcpp::Rcout << "V rempli en " << (double) t / CLOCKS_PER_SEC / 100. << " s\n";
}

```

Voilà le résultat. Le gain de temps peut être beaucoup plus important si on manipule de grosses structures (et non comme ici des chaînes de caractères de longueur 50, qui se copient très vite).

```

temps_de_remplissage()
## V rempli en 0.00811657 s
## V rempli en 0.00614014 s
## V rempli en 0.00574644 s

```

Note : Les Nous reviendrons sur ce sujet dans le chapitre sur la programmation objet.

7.2 Pairs & tuples

Définir une paire :

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

//[[Rcpp::export]]
void exemple_de_paire() {
    std::pair<int, double> x;
    x.first = 10;
    x.second = 11.25;
    Rcpp::Rcout << "x.first = " << x.first << "\n";
    Rcpp::Rcout << "x.second = " << x.second << "\n";
}
```

On peut bien sûr faire des vecteurs de paires. Pour ne pas alourdir la syntaxe utilisons typedef pour définir pts comme abréviation de std::pair<double, double>:

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

typedef std::pair<double, double> pts;

//[[Rcpp::export]]
void pts_vector() {
    std::vector<pts> x;
    for(int i = 0; i < 10; i++) {
        x.push_back( std::make_pair(R::rnorm(0,1), R::rnorm(0,1)) );
    }
    std::sort(x.begin(), x.end());
    // afficher contenu
    for(auto & a : x) {
        Rcout << "(" << a.first << "," << a.second << ")\n";
    }
}
```

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

//[[Rcpp::export]]
void exemple_de_tuple() {
    std::tuple<double, double, int> x;
    x = std::make_tuple(0.25, 0.8, 1);
    Rcout << "(" << std::get<0>(x) << "," << std::get<1>(x)
        << "," << std::get<2>(x) << ")\n";
}
```

7.3 Quelques algorithmes

std::sort trie « en place » ...

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

//[[Rcpp::export]]
NumericVector trie1(std::vector<double> x) {
    std::sort(x.begin(), x.end());
    return wrap(x);
}

//[[Rcpp::export]]
void trie2(NumericVector x) {
    std::sort(x.begin(), x.end());
}

//[[Rcpp::export]]
NumericVector trie3(std::vector<double> x, int n) {
    std::sort(x.begin(), x.begin() + n);
    return wrap(x);
}
```

Dans la première fonction on a fait une copie de `x` vers un `std::vector<double>`, `x` n'est donc pas modifié.

```
> x <- runif(6)
> x
## [1] 0.10224394 0.75911866 0.54284178 0.60787772 0.09529427 0.23922794
```

```
> trie1(x)
## [1] 0.09529427 0.10224394 0.23922794 0.54284178 0.60787772 0.75911866
```

```
> x
## [1] 0.10224394 0.75911866 0.54284178 0.60787772 0.09529427 0.23922794
```

```
> trie2(x)
> x
## [1] 0.09529427 0.10224394 0.23922794 0.54284178 0.60787772 0.75911866
```

Les *itérateurs* sont une généralisation des pointeurs ; en première approximation vous pouvez voir `x.begin()` et `x.end()` comme des pointeurs sur le début et la fin du vecteur. La fonction `trie3` ne trie que les `n` premiers éléments de `x`:

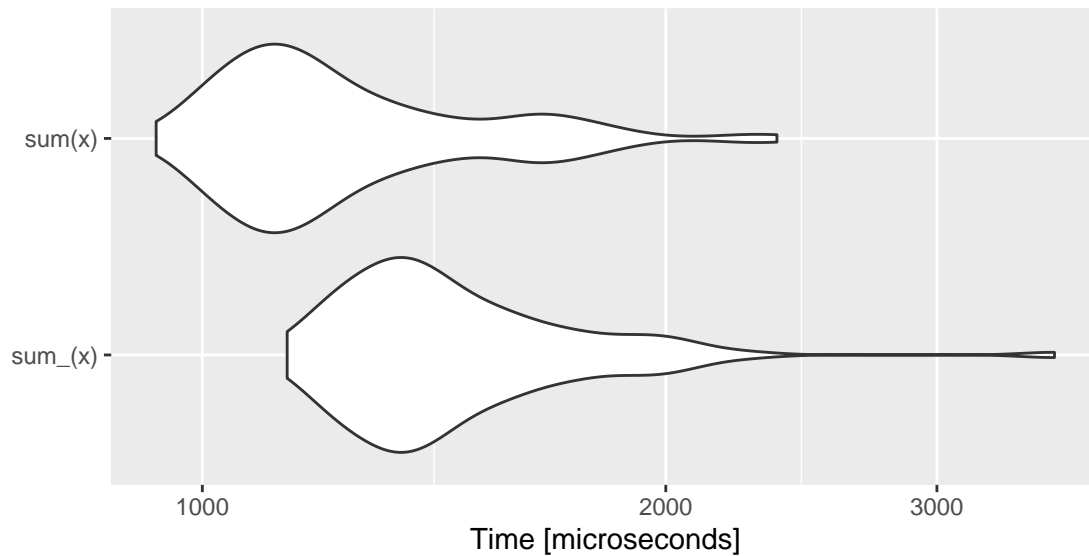
```
> trie3(x, 4)
## [1] 0.09529427 0.10224394 0.23922794 0.54284178 0.60787772 0.75911866
```

TODO `std::fill` et `std::accumulate`

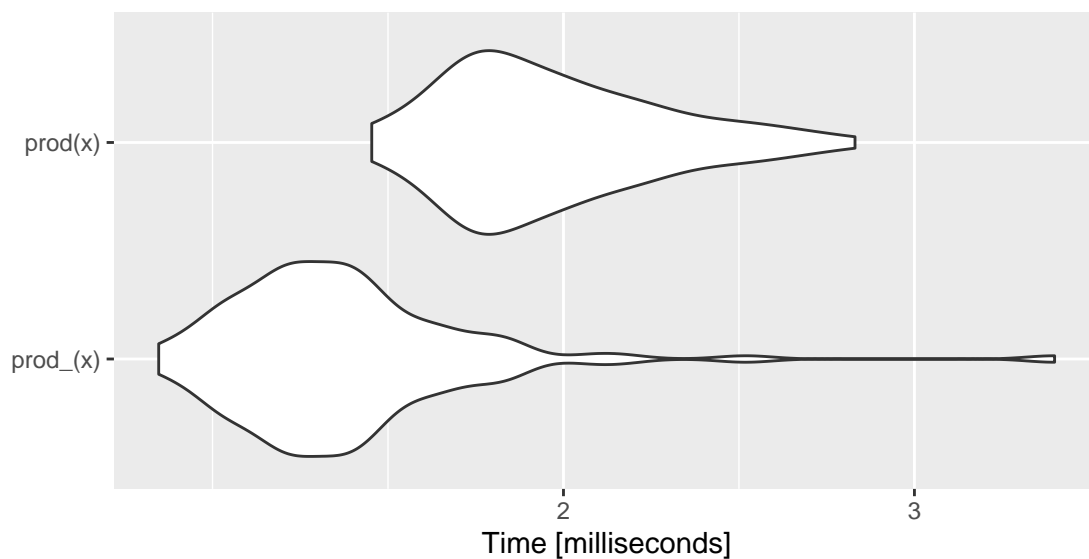
```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
double sum_(NumericVector x) {
    return std::accumulate(x.begin(), x.end(), 0.0);
}
//[[Rcpp::export]]
```

```
double prod_(NumericVector x) {
  return std::accumulate(x.begin(), x.end(), 1.0, std::multiplies<double>());
}
```

```
> x <- runif(1e6);
> mbm <- microbenchmark::microbenchmark(sum_(x), sum(x))
> ggplot2::autoplot(mbm)
## Coordinate system already present. Adding new coordinate system, which will replace the existing one
```



```
> mbm <- microbenchmark::microbenchmark(prod_(x), prod(x))
> ggplot2::autoplot(mbm)
## Coordinate system already present. Adding new coordinate system, which will replace the existing one
```



7.4 Maps

TODO

7.5 Heap

TODO

Chapitre 8

Création d'un package R

Pour diffuser votre travail, ou pour ne pas avoir besoin de recompiler vos fonctions à chaque fois, il faut créer un package R. Rstudio est d'une grande aide pour cela. Nous allons dans ce chapitre faire les premiers pas dans la création d'un package ; le tome du manuel de R intitulé *Writing R extensions* reste une référence indispensable.

Le package que nous allons créer s'appelle milou. Vous pouvez le retrouver à l'adresse <https://github.com/introRcpp/milou>.

8.1 Créer l'arborescence de fichiers

Pour commencer sélectionner le menu File l'option New Project, puis New Directory, puis R package using Rcpp. Vous allez pouvoir choisir à quel endroit le nouveau répertoire qui va contenir votre package (et portera son nom) sera installé. Nous choisissons d'appeler notre package milou.

Le répertoire milou/ contient:

- les fichiers DESCRIPTION et NAMESPACE
- les répertoires R/, src/ et man/ dont nous allons parler plus bas
- des fichiers qui ne font pas partie du package :
 - un fichier Read-and-delete-me (obtempérez)
 - un fichier milou.Rproj et un dossier (caché) .Rproj.user/ qui sont utilisés par Rstudio
 - un fichier .Rbuildignore qui contient des expressions régulières destinées à informer R de la présence de fichiers qui ne font pas partie du package...

Le contenu de DESCRIPTION est assez clair – vous pouvez et devez le modifier:

```
Package: milou
Type: Package
Title: What the Package Does in One 'Title Case' Line
Version: 1.0
Date: 2020-03-24
Author: Your Name
Maintainer: Your Name <your@email.com>
Description: One paragraph description of what the package does as one or more full
             sentences.
License: GPL (>= 2)
Imports: Rcpp (>= 1.0.3)
LinkingTo: Rcpp
```

Il est possible d'inclure des informations supplémentaires, par exemple un champ `Encoding` pour spécifier la façon dont les éventuelles lettres accentuées sont encodées (`latin1` et `UTF-8` sont les solutions les plus fréquentes). J'insère pour ma part la ligne

```
Encoding: UTF-8
```

qui correspond à l'encodage par défaut sous Linux et Mac OS et me permet d'accentuer correctement mon prénom dans le champ `Author`. Les utilisateurs de Windows choisiront peut-être plus commodément l'encodage `latin1`, mais Rstudio peut gérer l'encodage de votre choix et vous demande de choisir lors de la première sauvegarde d'un fichier.

Le fichier `NAMESPACE` contient deux lignes importantes pour l'utilisation de fonctions écrites avec Rcpp:

```
useDynLib(milou, .registration=TRUE)
importFrom(Rcpp, evalCpp)
```

La ligne

```
exportPattern("^[:alpha:]]+")
```

dit à R que toutes les fonctions dont le nom commence par un caractère alphanumérique sont exportées du package. C'est très bien quand on ne développe que pour soi, pour un package destiné à la diffusion il est souvent nécessaire de modifier cela. Nous le ferons plus tard.

8.2 Inclure une fonction C++

Les fonctions C++ sont dans le répertoire `src/`. Il contient deux fichiers, `rcpp_hello_world.cpp` qui contient quelques exemples basiques ; et `RcppExports.cpp`, qui est généré par la fonction `Rcpp::compileAttributes()`. Dans un premier temps vous pouvez ignorer son contenu.

Créons un fichier `count_zeros.cpp`, contenant

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

//[[Rcpp::export]]
int count_zeroes(IntegerVector x) {
  int re = 0;
  for(auto a : x)
    if(a == 0) ++re;
  return re;
}
```

Le nom du fichier n'a pas besoin de coïncider avec celui de la fonction, c'est juste plus commode pour s'y retrouver. Vous avez remarqué que nous utilisons du C++11. Pour que la compilation du package soit possible, il faut inclure dans `src/` un fichier `Makevars` contenant

```
PKG_CXXFLAGS = -std=c++11
```

Vous pouvez maintenant cliquer sur `Install and Restart` (sous l'onglet `Build`). Rstudio compile le package et relance la session R, puis charge le package. Vous pouvez tester la fonction `count_zeroes` !

```
library(milou)
x <- sample(0:10,1000,TRUE)
count_zeroes(x)
## [1] 113
```

8.3 Ce que Rstudio a fait avant la compilation

Rstudio a appelé la fonction `Rcpp::compileAttributes` qui a modifié le fichier `RcppExports.cpp`. Elle a créé cette fonction

```
RcppExport SEXP _milou_count_zeroes(SEXP xSEXP) {
BEGIN_RCPP
    Rcpp::RObject rcpp_result_gen;
    Rcpp::RNGScope rcpp_rngScope_gen;
    Rcpp::traits::input_parameter< IntegerVector >::type x(xSEXP);
    rcpp_result_gen = Rcpp::wrap(count_zeroes(x));
    return rcpp_result_gen;
END_RCPP
}
```

qui est en fait celle qui est appelée par R. Comment cela ? La fonction R qui correspond est dans le fichier `R/RcppExports.R`:

```
count_zeroes <- function(x) {
  .Call('_milou_count_zeroes', x)
}
```

8.4 Inclure une fonction R

La fonction `count_zeroes` n'est pas totalement satisfaisante. Une bonne idée serait de vérifier – dans le code R – que l'utilisateur a bien passé un vecteur d'entiers. On peut créer dans le répertoire R/ un fichier qu'on appellera par exemple `n_zero.r` et qui contient

```
n.zero <- function(x) {
  if( typeof(x) != "integer" )
    stop("Cette fonction compte les zéros dans les vecteurs d'entiers")
  count_zeroes(x)
}
```

Une définition alternative peut éviter l'appel à `count_zeroes` qui n'est qu'un « wrapper » pour un appel à `.Call`:

```
n.zero <- function(x) {
  if( typeof(x) != "integer" )
    stop("Cette fonction compte les zéros dans les vecteurs d'entiers")
  .Call('_milou_count_zeroes', x)
}
```

8.5 Contrôler quelles sont les fonctions exportées

Puisqu'on a créé `n.zero`, on ne veut pas que l'utilisateur puisse utiliser `count_zeroes`. On va donc modifier notre `NAMESPACE` pour qu'il contienne

```
useDynLib(milou, .registration=TRUE)
importFrom(Rcpp,evalCpp)
export(n.zero)
```

Ainsi la seule fonction exportée par notre package est `n.zero`. On peut toujours, à nos risques et périls, utiliser la fonction non exportée avec la syntaxe `milou:::count_zeroes` (notez le triple deux-points).

8.6 Documenter les fonctions avec roxygen2

Les fichiers de documentation sont inclus dans le répertoire `man/`. Ce sont des fichiers en `.Rd` qui peuvent être écrits à la main ; une solution qui s'avère à l'usage très commode (facilité d'écriture d'une part, de maintenance du package d'autre part) est de les faire générer par `roxygen2`. Pour cela il faut tout d'abord installer ce package : `install.packages("roxygen2")`. Il est possible que l'installation soit pénible car ce package dépend de `xml2` qui nécessite d'installer d'autres composantes logicielles sur le système. Soyez attentifs aux messages d'erreur, ils sont informatifs. Une solution simple sous un linux de type Ubuntu est d'utiliser le gestionnaire de paquets pour installer `r-cran-xml2` ou `r-cran-roxygen2` !

Pour documenter la fonction `n.zero`, placez le curseur dans cette fonction puis cliquez sur la baguette magique et choisissez `Insert Roxygen Skeleton`. Votre fichier ressemble maintenant à ceci:

```
#' Title
#'
#' @param x
#'
#' @return
#' @export
#'
#' @examples
n.zero <- function(x) {
  if( typeof(x) != "integer" )
    stop("Cette fonction compte les zéros dans les vecteurs d'entiers")
  .Call('_milou_count_zeroes', x)
}
```

On va compléter cette ébauche ainsi:

```
#' Compter les zéros
#'
#' @param x un vecteur de type 'integer'
#'
#' @details Cette fonction démontre l'utilisation d'une boucle C++11
#' pour compter les zéros dans un vecteur.
#'
#' @return le nombre de 0 dans x
#' @export
#'
#' @examples
```

```
#' a <- sample(0:99, 1e6, TRUE )
#' n.zero(a)
#'

n.zero <- function(x) {
  if( typeof(x) != "integer" )
    stop("Cette fonction compte les zéros dans les vecteurs d'entiers")
  .Call('_milou_count_zeroes', x)
}
```

Maintenant, on va faire générer à Rstudio le fichier `man/n.zero.Rd` qui correspond. Pour cela, il faut d'abord aller dans l'onglet Build, puis cliquer sur More, Configure Build Tools, cocher la case Generate documentation with Roxygen (cocher au minimum la case Rd files).

Ensuite, cliquez sur Build > More > Document pour faire générer ce fichier. Comme notre page d'aide contient des lettres accentuées, cela ne fonctionnera que si vous avez inséré dans DESCRIPTION la ligne Encoding: UTF-8. Le message d'erreur en l'absence de ce champ n'est pas très instructif, l'information se trouve dans un des warnings qui suit. En pratique, la documentation est généralement écrite en anglais, et les lettres accentuées y sont presque toujours absentes.

Une fois la documentation générée (regardez le contenu de `man/n.zero.Rd`) vous pouvez réinstaller le package et admirer la page de documentation en tapant `?n.zero` et `example(n.zero)`.

8.7 Générer le fichier NAMESPACE avec roxygen2

Le tag `@export` de roxygen2 signale que cette fonction est exportée. Cela permet à roxygen2 de générer, en plus de l'aide, le fichier NAMESPACE et nous évite d'insérer à la main des lignes d'exportation comme `export(n.zero)`. Cependant roxygen2 refuse (sagement) d'effacer un NAMESPACE qu'il n'a pas généré lui-même.

Pour y remédier je n'ai pas trouvé de meilleure solution que d'insérer au début du fichier NAMESPACE la ligne suivante, qui permet à roxygen2 de considérer que le NAMESPACE peut être effacé:

```
# Generated by roxygen2: do not edit by hand
```

Il existe sûrement une solution plus propre. Un problème subsiste cependant: roxygen2 ne génère pas les deux lignes indispensables au fonctionnement d'un package avec Rcpp, que nous avons mentionnées plus haut. La solution est d'insérer dans le répertoire R/ un fichier à cette fin. Nous l'appellerons par exemple `zzz.r` et il contiendra les lignes suivantes:

```
#' @useDynLib milou, .registration=TRUE
#' @importFrom Rcpp evalCpp
NULL
## NULL
```

Le NULL final peut être remplacé par un 0 ou ce que vous voulez (des appels aux fonctions `.onLoad` et `.onAttach` par exemple), mais il faut qu'il y ait un objet R à évaluer sinon le fichier n'est pas pris en compte par roxygen2.

8.8 Compilation multifichier

Avoir un fichier par fonction est une bonne pratique. Si une fonction a besoin d'appeler une fonction définie dans un autre fichier, la bonne solution est d'utiliser des fichiers de header `.h` appelés par une directive `#include`.

`n.zero {milou}`

R Documentation

Compter les zéros

Description

Compter les zéros

Usage

```
n.zero(x)
```

Arguments

`x` un vecteur de type 'integer'

Details

Cette fonction démontre l'utilisation d'une boucle C++11 pour compter les zéros dans un vecteur.

Value

le nombre de 0 dans `x`

Examples

```
a <- sample(0:99, 1e6, TRUE )  
n.zero(a)
```

[Package *milou* version 1.0 [Index](#)]

FIGURE 8.1 – Documentation de la fonction `n.zero`

Les templates vont également dans des fichiers .h. Nous allons ajouter au package milou quelques fichiers pour illustrer ceci.

Dans le dossier src/ ajoutons un fichier de template split.h:

```
#ifndef _milou_split_
#define _milou_split_

#include <vector>
#include <utility>

template<typename T1, typename T2>
std::pair< std::vector<T2>, std::vector<T2> > split(T1 x, T2 a0) {
    std::vector<T2> lo;
    std::vector<T2> hi;
    for(auto & a : x)
        if(a < a0)
            lo.push_back(a);
        else
            hi.push_back(a);
    return std::make_pair(lo,hi);
}

#endif
```

Notez le mécanisme à base de #define, qui permet d'éviter de potentiels problèmes d'inclusions multiples. Définissons deux fonctions qui utilisent ce template dans splitR.cpp:

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

//[[Rcpp::export]]
List splitR_double(NumericVector x, double pivot) {
    auto re = split(x, pivot);
    List L;
    L["lo"] = wrap(std::get<0>(re));
    L["hi"] = wrap(std::get<1>(re));
    return L;
}

//[[Rcpp::export]]
List splitR_int(IntegerVector x, int pivot) {
    auto re = split(x, pivot);
    List L;
    L["lo"] = wrap(std::get<0>(re));
    L["hi"] = wrap(std::get<1>(re));
    return L;
}
```

Le fichier splitR.h va contenir uniquement la déclaration de ces fonctions:

```
#include <Rcpp.h>

Rcpp::List splitR_double(Rcpp::NumericVector x, double pivot);
Rcpp::List splitR_int(Rcpp::IntegerVector x, int pivot);
```

Enfin, le fichier `random_split.cpp` contient deux définitions de fonctions à exporter:

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

// [[Rcpp::export]]
List random_split_double(NumericVector x) {
  return splitR_double(x, sample(x, 1)[0] );
}

// [[Rcpp::export]]
List random_split_int(IntegerVector x) {
  return splitR_int(x, sample(x, 1)[0] );
}
```

Sa compilation (qui crée le fichier `random_split.o`) est rendue possible par l'inclusion de `splitR.h` qui informe le compilateur de l'existence de `splitR_double` et `splitR_int`, et du type de leurs arguments. C'est lors de la phase finale de la compilation (création du fichier `milou.so`) que le lien avec les fonctions compilées dans `splitR.o` est réalisé.

Pour utiliser ces fonctions ajoutons dans R/ le fichier `split.r`:

```
#' Fonction split
#'
#' @param x un vecteur d'entiers ou de doubles
#' @param pivot (facultatif) un pivot
#'
#' @details si 'pivot' est absent,
#' un élément de 'x' pris au hasard sera utilisé
#'
#' @return une liste avec composantes 'lo' et 'hi'
#' @export
#'
#' @examples
#' x <- runif(20)
#' split(x, 0.5)
#'
#' x <- sample.int(100, 20)
#' split(x, 50)
#' split(x)
split <- function(x, pivot) {
  if(is.integer(x))
    if(missing(pivot))
      .Call('_milou_random_split_int', x)
    else
      .Call('_milou_splitR_int', x, pivot)
  else if(is.double(x))
    if(missing(pivot))
```



```

    .Call('_milou_random_split_double', x)
  else
    .Call('_milou_splitR_double', x, pivot)
  else
    stop("Mauvais type de x")
}

```

N'oublions pas de lancer la documentation avec roxygen2 pour que le NAMESPACE soit mis à jour.

```

require(milou)
example(split)
##
## split> x <- runif(20)
##
## split> split(x, 0.5)
## $lo
## [1] 0.3404985 0.2278287 0.4980164 0.2053555 0.3890900 0.2078072 0.3469400 0.3912514
## [9] 0.1085389
##
## $hi
## [1] 0.7081108 0.6530628 0.8058490 0.8926156 0.7855901 0.5876352 0.6634487 0.9953385
## [9] 0.5541547 0.7165223 0.5925015
##
##
## split> x <- sample.int(100, 20)
##
## split> split(x, 50)
## $lo
## [1] 8 2 16 39 5 42 15 25 27
##
## $hi
## [1] 67 82 84 99 83 95 91 87 97 94 68
##
##
## split> split(x)
## $lo
## [1] 8 2 16 39 5 15 25 27
##
## $hi
## [1] 67 82 42 84 99 83 95 91 87 97 94 68

```


Chapitre 9

Chaînes de caractères

Notions de code Ascii. Iso-8859-1 et variantes. UTF-8.

Les codes 0 à 31 correspondent à des codes de contrôles (par exemple 8 pour ‘backspace’) et ne sont pas imprimables ; c’est aussi le cas du caractère 127 qui correspond à ‘del’.

Affichons le code Ascii de 32 à 127 sur 8 colonnes.

```
#include <Rcpp.h>
#define COLS 8
#define OFF (96/COLS)
//[[Rcpp::export]]
void ascii() {
  for(int i = 32;;i++) {
    for(int k = 0; k < COLS; k++) {
      Rprintf("%03d : ", i + k*OFF);
      if(i + k*OFF < 127)
        Rcpp::Rcout << (char) (i + k*OFF) << "    ";
      else {
        Rcpp::Rcout << "DEL" << std::endl;
        return;
      }
    }
    Rcpp::Rcout << std::endl;
  }
}
```

```
> ascii()
## 032 :      044 : ,      056 : 8      068 : D      080 : P      092 : \      104 : h      116 : t
## 033 : !      045 : -      057 : 9      069 : E      081 : Q      093 : ]      105 : i      117 : u
## 034 : "      046 : .      058 : :      070 : F      082 : R      094 : ^      106 : j      118 : v
## 035 : #      047 : /      059 : ;      071 : G      083 : S      095 : _      107 : k      119 : w
## 036 : $      048 : 0      060 : <      072 : H      084 : T      096 : ‘      108 : l      120 : x
## 037 : %      049 : 1      061 : =      073 : I      085 : U      097 : a      109 : m      121 : y
## 038 : €      050 : 2      062 : >      074 : J      086 : V      098 : b      110 : n      122 : z
## 039 : ’      051 : 3      063 : ?      075 : K      087 : W      099 : c      111 : o      123 : {
## 040 : (      052 : 4      064 : @      076 : L      088 : X      100 : d      112 : p      124 : |
## 041 : )      053 : 5      065 : A      077 : M      089 : Y      101 : e      113 : q      125 : }
## 042 : *      054 : 6      066 : B      078 : N      090 : Z      102 : f      114 : r      126 : ~
## 043 : +      055 : 7      067 : C      079 : O      091 : [      103 : g      115 : s      127 : DEL
```

9.1 Chaînes de la vieille école : C strings

Ce sont des pointeurs vers un entier de type `char`, c'est-à-dire codé sur 8 bits – à moins de travailler sur une architecture très exotique – et pouvant prendre des valeurs de 0 à 255. La chaîne de caractère est un tableau dont la fin est signalée par la présence d'une valeur nulle.

EN C et C++, la syntaxe `'a'` (guillemet simple) implique le remplacement de la lettre par son code ascii (pour `a` c'est 97). Dans l'exemple ci-dessous, la chaîne `'b'` est explicitement présentée comme le tableau contenant 116, 111, 105, 0. Les lettres de code ascii qui correspondent sont `t`, `o` et `i` ; si on avait écrit `char b[] = "toi"` ; le résultat aurait été identique.

```
#include <Rcpp.h>
//[[Rcpp::export]]
void quelques_chaines() {
  char a[] = "Bonjour";
  char b[4] = {116, 111, 105, 0}; // 3 lettres et un 0 final
  char c[5] = {'v', 'o', 'u', 's', 0}; // 4 lettres et un 0 final
  Rcpp::Rcout << a << " " << b << std::endl;
  Rcpp::Rcout << a << " " << c << std::endl;
  Rcpp::Rcout << "Taille de a = " << sizeof(a) << std::endl;
  Rcpp::Rcout << "Longueur de la chaîne = " << std::strlen(a) << std::endl;
  Rcpp::Rcout << "a[0] = " << a[0] << std::endl;
  Rcpp::Rcout << "a[1] = " << a[1] << std::endl;
}
```

```
> quelques_chaines()
Bonjour toi
Bonjour vous
Taille de a = 8
Longueur de la chaîne = 7
a[0] = B
a[1] = o
```

Notez `std::strlen(a)` qui renvoie la longueur de la chaîne `a`.

Comme ce sont des pointeurs, on ne va pas bêtement tester leur égalité... Il faut utiliser `std::strcmp(a,b)` qui renvoie 0 si les chaînes sont égales, et dans le cas contraire un nombre dont le signe est positif si $a < b$ dans l'ordre lexicographique, négatif sinon.

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
//[[Rcpp::export]]
void compare() {
  char a[] = "Gaston";
  char b[] = "Fantasio";
  char c[] = "Gaston";
  SHOW(a == b);
  SHOW(a == c);
  SHOW( std::strcmp(a,b) );
  SHOW( std::strcmp(b,a) );
  SHOW( std::strcmp(a,c) );
}
```

```
> compare()
a == b = 0
a == c = 0
std::strcmp(a,b) = 1
std::strcmp(b,a) = -1
std::strcmp(a,c) = 0
```

9.1.1 Manipuler les chaînes de R comme des C strings

La fonction `CHAR`, appliquée à un élément d'un `CharacterVector`, renvoie un `const char *`. Le mot clef `const` devant un type indique au compilateur que le contenu ne peut être modifié : un `const char` est char « constant » ; le pointeur renvoyé par `CHAR` pointe vers des `const char`, c'est-à-dire vers une chaîne de caractères que l'utilisateur n'a pas le droit de modifier. On peut tricher en utilisant un cast vers un `char *` mais c'est à nos risques et périls...

```
#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void c_strings(CharacterVector x) {
    int n = x.size();
    for(int i = 0; i < n; i++) {
        const char * a = CHAR(x[i]);
        Rcout << i << " : " << a << std::endl;
        Rcout << " adresse : " << (void *) a << std::endl;
    }
}
```

```
> a <- c("gaston", "jeanne", "gaston")
> c_strings(a)
0 : gaston
  adresse : 0x55ab1c525ea0
1 : jeanne
  adresse : 0x55ab1a252878
2 : gaston
  adresse : 0x55ab1c525ea0
```

On s'aperçoit ici que les deux "gaston" sont stockés par R à la même adresse (les apprentis sorciers qui auraient joué avec `.Internal(inspect(a))` l'auront constaté). Ceci incite à être très prudent, toute modification *in place* du premier modifierait l'autre ! Ça n'est pas pour rien que `CHAR` renvoie un `const char *` et non un `char *`.

9.1.2 Exemple d'utilisation des C strings

C'est rustique (et risqué) mais ça peut être redoutablement efficace. Si on sait que les codes ascii des nombres de 0 à 9 se suivent...

```
#include <Rcpp.h>
inline int fast_atoi(const char * a)
{
    int re = 0;
    while( *a ) {
```

```

        re = re*10 + (*(a++) - '0');
    }
    return re;
}
//[[Rcpp::export]]
Rcpp::IntegerVector as_int(Rcpp::CharacterVector x) {
    int n = x.size();
    Rcpp::IntegerVector R(n);
    for(int i = 0; i < n; i++) {
        const char * a = CHAR(x[i]);
        R[i] = fast_atoi(a);
    }
    return R;
}

```

Cette fonction file comme le vent mais ne fait rien pour vérifier que la chaîne code correctement un entier !

```

> as_int( c("456", "123", "789", "Dave Brubeck") )
[1]      456      123      789 1069870157

```

Autres possibilités offertes par les C-strings : modifier une chaîne *in place* en remplaçant les espaces par des 0 (code de fin de chaîne) pour en isoler les mots, etc. Cela peut aller beaucoup plus vite que les opérations équivalentes avec les chaînes C++.

9.2 Chaînes C++

La librairie standard contient la classe string qui facilite grandement le travail.

```

#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
//[[Rcpp::export]]
void autres_chaines() {
    std::string x = "ça va ?";
    std::string y = "Bonjour";
    std::string z = y + " " + x; // Concaténation
    SHOW(x);
    SHOW(y);
    SHOW(y[0]);
    SHOW(y[1]);
    SHOW(z);
}

```

```

> autres_chaines()
x = ça va ?
y = Bonjour
y[0] = B
y[1] = o
z = Bonjour ça va ?

```

La comparaison est grandement facilitée :

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
//[[Rcpp::export]]
void compare2() {
    std::string a = "Gaston";
    std::string b = "Fantasio";
    std::string c = "Gaston";
    SHOW(a == b);
    SHOW(a == c);
    SHOW(a < b);
    SHOW(b < c);
}
```

```
> compare2()
a == b = 0
a == c = 1
a < b = 0
b < c = 1
```

9.2.1 Passer d'un style à l'autre

Cette fonction démontre comment passer facilement d'un style de chaîne à l'autre :

```
#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
//[[Rcpp::export]]
void conversions() {
    char a[] = "Gaston";
    std::string b(a);
    SHOW(a);
    SHOW(b);

    std::string x = "Longtarin";
    const char * y = x.c_str();
    SHOW(x);
    SHOW(y);
}
```

```
> conversions()
a = Gaston
b = Gaston
x = Longtarin
y = Longtarin
```

9.2.2 Manipuler les chaînes de R comme des std::string

On peut pour commencer les passer simplement en argument d'une fonction :

```
#include <Rcpp.h>
//[[Rcpp::export]]
void perroquet(std::string x) {
```

```

Rcpp::Rcout << "Vous dites : " << x << std::endl;
Rcpp::Rcout << "longueur = " << x.size() << std::endl;
Rcpp::Rcout << "adresse &x[0] = " << (void *) &x[0] << std::endl;
Rcpp::Rcout << "adresse x.c_str() = " << (void *) x.c_str() << std::endl;
}

```

Le (void *) est nécessaire pour faire afficher l'adresse du pointeur : si on fournit un objet de type char * à l'opérateur << il considère qu'on veut afficher la chaîne de caractère. Essayez Rcpp::Rcout << y si nécessaire.

```

> perroquet("bavard")
Vous dites : bavard
longueur = 6
adresse &x[0] = 0x7ffe099ceb10
adresse x.c_str() = 0x7ffe099ceb10

```

La fonction `as<std::string>` permet la conversion vers une `std::string` (notez la présence des chevrons <> : `as` est une fonction templétée).

```

#include <Rcpp.h>
using namespace Rcpp;
//[[Rcpp::export]]
void cpp_strings(CharacterVector x) {
    int n = x.size();
    for(int i = 0; i < n; i++) {
        std::string a = as<std::string>(x[i]);
        Rcout << i << " : " << a << std::endl;
    }
}

```

Application :

```

> a <- c("gaston", "jeanne", "gaston")
> cpp_strings(a)
0 : gaston
1 : jeanne
2 : gaston

```

9.3 Chaînes Rcpp::String

Les `CharacterVector` sont des vecteurs de chaînes de caractères. On a rarement besoin de manipuler leurs éléments séparément autrement qu'en les convertissant en `char *` ou en `std::string`, comme on l'a fait plus haut ; cependant la classe `Rcpp::String` permet de le faire. Voici un bref exemple.

```

#include <Rcpp.h>
#define SHOW(x) Rcpp::Rcout << #x << " = " << (x) << std::endl;
using namespace Rcpp;
//[[Rcpp::export]]
void atomic_strings(CharacterVector x) {
    int n = x.size();
    for(int i = 0; i < n; i++) {
        String a = x[i];
    }
}

```



```

    const char * b = a.get_cstring();      // équivaut à b = CHAR(x[i]);
    Rcout << i << " : " << b << std::endl;
}
}

```

```

> atomic_strings(a)
0 : gaston
1 : jeanne
2 : gaston

```

9.4 Exemple : lecture d'un fichier

Quand on aura vu les vecteurs standards !

(TODO)

9.5 Addendum sur les listes

Dans le cas des vecteurs nommés ou des listes (ou des data frame), on peut utiliser indifféremment

Les fonctions ci-dessous ne gèrent pas les erreurs, voir dans le chapitre sur les listes la fonction `get_alpha`.

```

#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector get_from_list_1(List L, std::string k) {
    NumericVector R = L[k];
    return R;
}

```

Pour en finir avec le sujet un dernier exemple de traitement du contenu d'une liste : énumérer les éléments, leur nom, et leur type.

```

#include <Rcpp.h>
using namespace Rcpp;

std::string le_type(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";
    }
}

//[[Rcpp::export]]
void contenu_liste(List L) {
    int n = L.size();
    if(n == 0) {
        Rcout << "liste vide\n";
        return;
    }
    SEXP R = L.names();
    bool has_names = (TYPEOF(R) == STRSXP);
    for(int i = 0; i < n; i++) {
        Rcout << i ;
        if(has_names)
            if( strcmp( CHAR(String_ELT(R, i)), "" ) )
                Rcout << " \" << CHAR(String_ELT(R, i)) << "\"";
        Rcout << " : " << le_type(L[i]) << std::endl;
    }
}

```

Notez l'usage de `String_ELT(R, i)` : si `R` était un `CharacterVector` on pourrait écrire `R[i]`, mais on a ici utilisé un `SEXP`, ce qui oblige à utiliser cette fonction un peu inélégante.

9.6 Exemple : chaînes de caractère et spécialisation de template

Saisissons l'occasion de la manipulation de chaînes de caractères pour illustrer la possibilité de donner des implémentations différentes d'un même template.

La conversion d'une `std::string` vers un `double` se fait avec `std::stod`, vers un `int` avec `std::stoi`, vers un `float` avec `std::stof`. Le template ci-dessous permet d'utiliser `sto<double>`, `sto<int>`, `sto<float>`.

Note l'argument `const std::string & x` est une référence à une chaîne `std::string` *constante*; ceci indique au compilateur que la fonction ne modifiera jamais `x`. Cela peut permettre des optimisations à la compilation.

```

#ifndef STO_
#define STO_
// Déclaration du template
template<typename T>
T sto(const std::string & x);

// Implémentations pour les types 'double', '
template<>
double sto<double>(const std::string & x) {
    return std::stod(x);
}

template<>
float sto<float>(const std::string & x) {
    return std::stof(x);
}

```

```
}  
  
template<>  
int sto<int>(const std::string & x) {  
    return std::stoi(x);  
}  
#endif
```

L'intérêt d'une telle définition n'est pas forcément apparent. Un des avantages est qu'un tel template peut être appelé dans un autre template plus complexe. Par exemple :

```
template<typename T>  
T somme(std::string a, std::string b) {  
    return sto<T>(a) + sto<T>(b);  
}
```