Introduction à C++ et Rcpp

Hervé Perdry

2020-03-10

Table des matières

Chapitre 1

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 2

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.

2.1 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) # chaines 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] 1 1 1 1 0 1 4 3 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
```

2.2 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 différentie 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,]    1    1    0    4    1    1
[2,]    1    1    1    3    0    1
```

2.3 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"'.

2.3.1 Cas de la matrice

```
> dput(d)
structure(c(1L, 1L, 1L, 1L, 0L, 1L, 4L, 3L, 1L, 0L, 1L, 1L), .Dim = c(2L,
6L))
```

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

2.3.2 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 »).

2.3.3 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
```

2.4 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 ))
@55b1261eed50 13 INTSXP g0c0 [NAM(7)]  1 : 10 (compact)
```

```
> .Internal(inspect( c(0.1, 0.2) ))
@55b129690608 14 REALSXP g0c2 [] (len=2, tl=0) 0.1,0.2
```

```
> a <- c(0.1, 0.2)
> .Internal(inspect( a ))
@55b129664f08 14 REALSXP g0c2 [NAM(7)] (len=2, tl=0) 0.1,0.2
```

```
> names(a) <- c("A", "B")
> .Internal(inspect( a ))
@55b129661a88 14 REALSXP g0c2 [NAM(1),ATT] (len=2, tl=0) 0.1,0.2
ATTRIB:
  @55b1257bef58 02 LISTSXP g0c0 []
    TAG: @55b121b5df70 01 SYMSXP g1c0 [MARK,NAM(7),LCK,gp=0x4000] "names" (has value)
    @55b129661a48 16 STRSXP g0c2 [NAM(7)] (len=2, tl=0)
      @55b123bc0d28 09 CHARSXP g1c1 [MARK,gp=0x60] [ASCII] [cached] "A"
      @55b1246a3378 09 CHARSXP g1c1 [MARK,gp=0x60] [ASCII] [cached] "B"
```

```
> .Internal(inspect( h ))
@55b1269f3438 13 INTSXP g0c2 [OBJ,NAM(7),ATT] (len=4, tl=0) 2,1,1,2
ATTRIB:
  @55b127fd5788 02 LISTSXP g0c0 []
    TAG: @55b121b5e0c0 01 SYMSXP g1c0 [MARK,NAM(7),LCK,gp=0x6000] "levels" (has value)
    @55b1269f3338 16 STRSXP g0c2 [NAM(7)] (len=2, tl=0)
      @55b123da0178 09 CHARSXP g1c1 [MARK,gp=0x60] [ASCII] [cached] "L1"
      @55b123d9ed20 09 CHARSXP g1c1 [MARK,gp=0x60] [ASCII] [cached] "L2"
    TAG: @55b121b5e440 01 SYMSXP g1c0 [MARK,NAM(7),LCK,gp=0x6000] "class" (has value)
    @55b127130200 16 STRSXP g0c1 [NAM(7)] (len=1, tl=0)
      @55b121c05bb0 09 CHARSXP g1c1 [MARK,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 3

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

3.1 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 chaines de caractères) se fait bien sûr en manipulant une suite de tels nombres.

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

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

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

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

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

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

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

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

3.2 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;
  Rcout << "sizeof(a) = " << sizeof(a) << "\n";
  Rcout << "sizeof(b) = " << sizeof(b) << "\n";
  Rcout << "sizeof(c) = " << sizeof(c) << "\n";
  Rcout << "sizeof(d) = " << sizeof(d) << "\n";
}
```

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

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

Notez le `using namespace Rcpp` qui permet de taper simplement `Rcout` au lieu de `Rcpp::Rcout`.

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.

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`.

3.2.0.1 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;
}
```

3.2.1 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;`

3.3 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";
}
```

3.3.1 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 $a2^b$, ou a et b sont bien sûr codés en binaire (sur 53 bits – dont un bit 1 implicite – pour a, 11 pour b, et un bit de signe). Cette précision finie implique des erreurs d'arrondi. Pour plus de détails, voir Wikipedia sur la norme IEEE 754 : https://fr.wikipedia.org/wiki/IEEE_754

Quelle est la différence entre les nombres ci-dessus ?

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

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

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

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

3.3.2 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 divise(double a, double b) {
  double r = a/b;
  Rcpp::Rcout << a << " / " << b << " = " << r << std::endl;
  return r;
}
```

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

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

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

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

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

```
#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
```

3.4 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.

3.5 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'incrémentation `++` 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
```

3.6 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);
}
```

3.7 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 débuggage « manuel ».

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

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

3.8 Contrôle du flux d'exécution : les boucles

3.8.1 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`.

3.8.2 `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 = 0; i < n; i++) {
    Rcpp::Rcout << i ;
    if(i%3 == 0)
      continue;
    Rcpp::Rcout << ".";
  }
  Rcpp::Rcout << std::endl;
}
```

```
> trois(12)
01.2.34.5.67.8.910.11.
```

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)
AAAAAAAAAAAAAA
```

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

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

3.9 Contrôle du flux d'exécution : les alternatives

3.9.1 `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
```

3.9.2 `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";
  }
}
```

3.10 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.

3.10.1 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;
}
```

3.10.2 Exemple : compter les zéros

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeros(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++) {
    re += (x[i] == 0)?1:0;
  }
  return re;
}
```

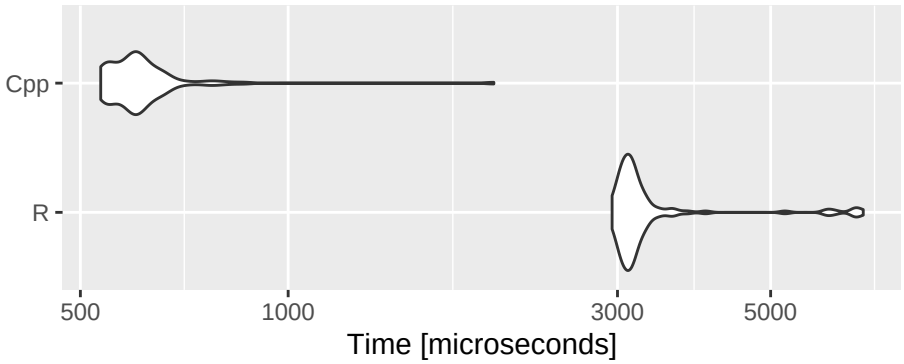
On a utilisé ici l'*opérateur ternaire* `test?a:b`, qui vaut `a` si `test` est `true`, `b` sinon. Comment les performances de cette fonction se comparent-elles avec le code R `sum(a == 0)` ?

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

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

```
> mbm <- microbenchmark::microbenchmark( R = sum(a == 0), Cpp = count_zeros(a))
> mbm
Unit: microseconds
 expr      min        lq     mean   median       uq      max neval
    R 2946.561 3048.3900 3410.511 3131.506 3256.573 6810.086   100
  Cpp  535.123  562.5595  620.619  600.019  628.106 1986.124   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.

La solution suivante a été proposée en cours.

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeros2(Rcpp::IntegerVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    (x[i] == 0)?re++:0;
  }
  return re;
}
```

Mais en fait, comme en R, ajouter un booléen à un int produit une conversion `false` : 0 et `true` : 1.

```
#include <Rcpp.h>
//[[Rcpp::export]]
int count_zeros3(Rcpp::IntegerVector x) {
  int re = 0;
  int n = x.size();
  for(int i = 0; i < n; i++) {
    re += (x[i] == 0);
  }
  return re;
}
```

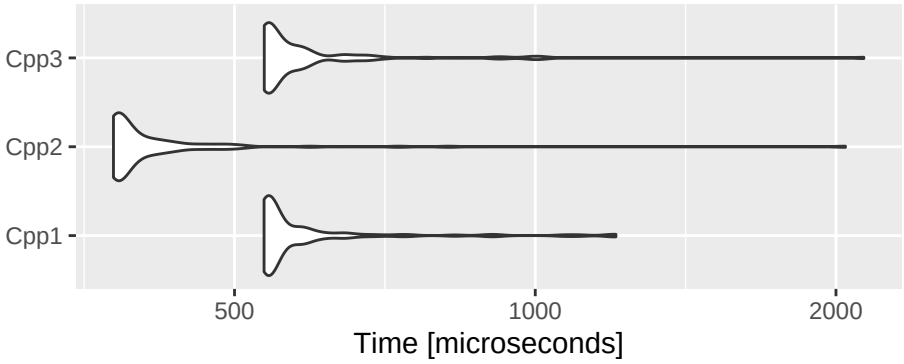
Une comparaison s'impose.

```
> count_zeros2(a)
[1] 10017
```

```
> count_zeros3(a)
[1] 10017
```

```
> mbm <- microbenchmark::microbenchmark( Cpp1 = count_zeros(a),
+             Cpp2 = count_zeros2(a),
+             Cpp3 = count_zeros3(a) )
> mbm
Unit: microseconds
 expr     min       lq     mean   median       uq      max neval
 Cpp1 535.193 537.2225 595.0243 546.2605 583.3950 1204.392   100
 Cpp2 378.243 380.1680 427.3225 387.6830 416.4425 2044.154   100
 Cpp3 535.448 537.2905 602.5900 549.3960 583.5315 2132.751   100
```

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



Selon les options de compilation ou les architectures, il peut y avoir ou non une différence entre ces solutions.