Langage C

Interfaçage entre R et C

Thomas Guyet

AGROCAMPUS OUEST, Centre de Rennes

Version 1.0, Dernière modification : 18 avril 2012

CHAPITRE

1 Introduction

Dans cette partie du cours, nous introduisons les principes de l'interfaçage des langage R et C. L'interfaçage consiste à pouvoir utiliser les qualités d'un des langages dans l'autre. Le langage R propose une utilisation facile de fonction statistiques qui n'existent pas dans R. Il peut alors être intéressant d'aller chercher ces fonctionnalités existantes pour développer des outils en C qui nécessite des calculs statistiques. Dans l'autre sens, le langage C est un langage universel et puissant. En tant que langage universel, il donne accès à de nombreuses fonctionnalités de traitement de données développées dans des librairies dédiées. En tant que langage très proche de la machine, il offre des capacités d'exploitation des processeurs beaucoup plus pointues que le langage R et, en particulier, il peut aider les programmeur à traiter de grands volumes de données, et d'améliorer les performances de temps de calcul.

Certains de mes polycopiés sont peu utiles, parce qu'ils sont redondants avec bon nombre de cours librement accessible sur le net. Dans ce cas du sujet de ce document, les documentations claires sont peu nombreuses, la documentation officielle est difficile d'accès et des tutoriels didactiques inexistants. J'espère donc que le lecteur trouvera un polycopié comblant ce manque.
Il s'adresse à des lecteurs plutôt statisticiens, avec une bonne connaissance de R mais non-nécessairement informaticiens. Des connaissances préalables des langages C sont également nécessaires [1] pour suivre les exemples présentés. Une compréhension des pointeurs est nécessaire et un certain recul sur la gestion de la mémoire sera également utile.

1.1 Préparation de l'environnement de programmation

1.1.1 Installation sous Linux

1.1.2 Installation sous Windows

Notons que la démarche d'incorporation de C en R est la même peu importe la plateforme informatique avec laquelle on travaille si tous les logiciels nécessaires sont correctement installés.

1.2 Benchmark "Quicksort" et objectif

L'algorithme Quicksort est un algorithme très efficace de tri d'un tableau. Cet algorithme est particulièrement inadapté à R parce que :

1. il nécessite beaucoup de tests et que la complexité des structures de données de R rend la simple comparaison d'entier plutôt compliqué,

1. il s'agit de connaissance données dans le cadre d'un enseignement court d'introduction au langage C pour des non-informaticien.

2. la récursivité de l'algorithme nécessite beaucoup d'appel de fonctions, ces appels de fonctions sont peu efficaces en R,
3. il ne peut pas facilement être vectorisé et donc l'usage des méthodes de programmation vectorielle [2] ne sont pas très utiles.

On donne dans l'Annexe de la section Annexe A les implémentations de l'algorithme Quicksort en langage R et en langage C.
Voici les temps de calculs obtenus pour ces programmes en ordonnant un tableau d'entiers de taille 20000 à 200000. On constate alors que le temps de calcul est entre 260 et 600 fois inférieur pour un programme en C. Il n'est donc pas raisonnable d'utiliser l'algorithme Quicksort développé en R. **L'un des objectifs applicatif de l'interfaçage est d'utiliser la fonction Quicksort développée C directement dans l'environnement R** pour bénéficier de ses performances dans l'environnement de programmation R.

Taille tableau	**C (s)**	**R (s)**
20000	0.010	5.544
40000	0.030	18.217
60000	0.070	37.867
80000	0.120	54.831
100000	0.180	70.081
120000	0.260	82.801
140000	0.350	110.747
160000	0.460	124.584
180000	0.570	148.761
200000	0.700	183.423

2. Les fonctions `mapply`, `sapply`, etc. sont des fonctions qui permettent de faire efficacement des traitements sur un ensemble de données représentées respectivement dans des vecteurs ou dans des séquences. La modification des algorithmes pour qu'ils utilisent ces fonctions s'appelle la vectorisation.

CHAPITRE

2 Étendre les capacités de R grâce au langage C

L'hypothèse de base de cette partie est que le programmeur souhaite réaliser un programme dans l'environnement de programmation R. Il aura par exemple déjà à disposition une partie de son programme en C et souhaitera offrir la possibilité d'avoir les fonctionnalités de son programme dans l'environnement R.

2.1 Principe

Le principe de la création de code C utilisable dans R consiste à développer une fonction C dans un fichier `.c` qui sera compilé avec le compilateur de R (en fait des options de R de votre compilateur C). La compilation ne consistera pas à construire un programme exécutable, mais uniquement une librairie partagée (*shared object* avec l'extension `.so` sous Linux et *dynamic linked library* avec l'extension `.dll` sous Windows). Ces fichiers comprennent des versions de vos fonctions compilées pour votre ordinateur.

Pour utiliser ces fonctions compilées dans R, il faut charger ce fichier compilé.

Illustrons maintenant les trois étapes présentées dans l'introduction afin d'incorporer du code C en R en prenant l'exemple du programme "Hello, world !" tiré de [5].

Le fichier `hello.c` contenant le code C très simple est donné ci-dessous. Ce programme contient une fonction qui affiche à l'écran n fois la chaîne de caractères "Hello, world !".

```c
#include <R.h>

void hellofct(int *n)
{
  int i;
  for(i=0; i < *n; i++) {
    Rprintf("Hello, world!\n");
  }
}
```

Ce programme à quelques particularités par rapport à nos programmes habituellement développés :

- il n'y a pas de fonction **main**(). En effet, nous n'allons pas créer un programme exécutable, donc il n'y a pas de "point de départ" d'exécution, mais juste une collection de fonctions (ici, une seule fonction **hellofct** ()).
- le programme commence par la ligne `#include <R.h>`. En ajoutant le fichier d'en-tête `R.h`, nous rendons possible la compilation de ce fichier pour R.
- la fonction usuelle **printf** est remplacée par la fonction **Rprintf** qui permet de faire afficher du texte dans la console de R,

2.1.1 Compilation du code C

La première étape consiste à compiler le code C afin de créer une librairie partagée (fichier `.so` si on travaille sur Linux ou fichier `.dll` si on travaille sur Windows). On effectue cette étape dans une fenêtre terminal sur Linux ou dans une fenêtre de commandes sur Windows. On se place d'abord dans le répertoire où se trouve le fichier contenant le code C, puis on tape la commande :

```
R CMD SHLIB hello.c
```

L'objet partagé ou la librairie dynamique créé par cette commande se nommera `hello.so` ou `hello.dll`. Dans la suite, je ne considérerai plus le cas Windows, mais le lecteur sera adapté les instructions sans problème.

Remarque 2

Il est tout à fait possible d'intégrer plusieurs fichiers `.c` dans une même librairie partagée. C'est plutôt même une bonne démarche d'avoir décomposé votre code dans plusieurs fichiers différents. La commande ci-dessous permettra de créer une librairie `malib.so` à partir de trois fichiers source.

```
R CMD SHLIB -o malib.so fichier1.c fichier2.c fichier3.c
```

Je précise également que tous les fichiers `.h` qui sont inclus dans un fichier `.c` seront bien intégrés dans la librairie partagée sans que vous les listiez. Le préprocesseur se sera charger de faire l'inclusion. Pour plus de détails et de compréhension sur les étapes de compilation, je reporte le lecteur à la partie du cours qui traite explicitement de ces questions.

2.1.2 Chargement en R du code C compilé

La suite de la procédure se déroule dans R. Il faut donc commencer par lancer R, et par choisir le répertoire contenant votre librairie partagée comme répertoire de travail de R [1].
Il faut ensuite charger en R la librairie partagée avec la fonction **dyn.load** (le nom du fichier doit obligatoirement être entre guillemets) :

```
dyn.load("hello.so")
```

On peut ici faire un test pour s'assurer que les fonctions C ont été correctement chargées en R. Pour ce faire, on utilise la fonction **is.loaded** avec, en argument, le nom entre guillemets d'une fonction C contenue dans notre fichier. Si cette commande retourne **TRUE**, alors tout est correct.

2.1.3 Appel en R des fonctions programmées en C

On peut finalement appeler en R les fonctions créées dans le code C avec la fonction d'interface **.C**. On utilise cette fonction de la façon suivante :

```
.C("nomfonction",argument1,argument2,...)
```

Dans le cas de notre fonction servant d'exemple, nous pourrions ainsi faire appel à la commande suivante faire appel à la fonction **hellofct** avec en paramètre 3.

1. la choix du répertoire de travail n'est pas indispensable, mais utile pour ne pas se poser de question sur la localisation des fichiers.

```
.C("hellofct",3)
```

! Attention ! - Passage des paramètres

Le paramètre de la fonction C est bien un pointeur, mais lorsqu'on utilise l'interface R, alors l'argument qui est donné à la fonction est bien un pointeur sur un emplacement mémoire qui contiendra la valeur désirée (ici, 3).

! Attention ! - Vérification des types des paramètres

Le langage R est un langage de script, il n'y a pas de vérification a priori des type, en revanche, si les types ne sont pas compatibles, au mieux ça plante, au pire la fonction C est bien exécutée sans que personne ne sache bien ce qu'il va se passer!!
Il est préférable de s'assurer que chaque argument passé à la fonction C est du bon type en leur appliquant une fonction de conversion de type telle que **as.integer**, **as.double**, **as.character** ou **as.logical**. Souvenez vous qu'en R, les nombres sont des **double** par défaut, ceci peut poser des problèmes si vous pensiez que c'était des entiers.

Pour rendre les appels à la fonction C totalement transparents (ce sera plus facile à utiliser!), il est conseillé de créer une fonction R qui `enveloppe` l'appel à la fonction C :

```
Hello <- function(n){
  .C("hellofct",as.integer(n))
}
```

On peut ensuite utiliser cette fonction pour afficher 4^2 fois `Hello, world!` par la commande **Hello**(4).

2.2 Interfaces

Lorsqu'on parle d'interfaces, on parle ici de fonctions R qui vont permettre d'appeler les fonctions chargées depuis une librairie partagée. Elles réfèrent aux différents modes d'appels d'une fonction C dans un script R. Il en existe trois, du plus limité au plus général (et donc du plus simple au plus complexe pour le programmeur) :
– `.C`
– `.Call`
– `.External`
Chacune de ces interfaces impose au programmeur le profil de la fonction C qui peut être chargé. Il faut comprendre les contraintes liées à l'utilisation ces différentes interfaces pour choisir la plus appropriée et l'utiliser correctement.

2.2.1 Interface `.C` : extension simple(-iste)

L'interface `.C` est l'interface simple pour charger une fonction C. Cette interface est simple parce qu'elle ne va pas demander beaucoup de travail de modification du code C, que le programmeur

2. par exemple!

n'aura pas à entrer dans les détails de R pour la comprendre mais elle ne permettra pas de faire tout ce qu'on pourrait imaginer car tous les types de variable de R ne pourront être utilisés (en particulier les listes).

Definition 1 - Fonction pour l'interface `.C`

Une fonction C qui peut être chargée par l'interface `.C` est une fonction dont
- les arguments passés sont des pointeurs sur un type simple (*e.g.* `int`, `double`, `char` ou `char *`). Toutes les variables simples de R étant des vecteurs, il faut s'attendre à ce qu'une variable récupérée dans la fonction C soit un tableau.
- le type de retour de la fonction doit être `void`. Les résultats d'une fonction doivent être retournés par les pointeurs passés en paramètre.

Le résultat de la fonction `.C` est une *liste* R contenant les valeurs des arguments après l'exécution de la fonction.

Exemple 1 - Fonction max()

On donne comme exemple une fonction de calcul d'un maximum de deux nombres réels. Dans le fichier R, on retrouve le chargement de la librairie (nommée ici `max.so`) ainsi que la création d'une fonction enrobant l'interface. Ce fichier se termine en montrant l'utilisation de la fonction pour afficher le maximum de deux nombres.
Dans le fichier C, on propose une fonction calculant simplement le maximum de deux nombres **d1** *et* **d2** *passés en paramètre. Notez que comme il s'agit de pointeurs, il faut bien penser à les déréférencer.*
Une troisième argument **output** *sert de variable de sortie de la fonction. On ne peut pas utiliser de* `return` *alors on dépose le résultat dans l'emplacement mémoire désigné par* **output** *pour que ce résultat soit récupéré dans R.*
Dans l'appel de l'interface `.C`, on a récupéré une variable **tmp** *qui contient la valeurs des emplacements mémoires pointés par arguments de la fonction. En particulier, le troisième élément de la liste contient la valeur de* **output**. *Dans la fonction R, on peut donc récupérer le résultat de notre fonction C.*

```r
dyn.load("max.so")

max <- function (d)
{
  out=0
  tmp <- .C("max", as.double(d), as.integer(length(d)), as.double(out))
  return(tmp[[3]])
}

m<-max(4.45, 5.54)
print (m)
```

```c
#include <R.h>

void max(double *d1, double *d2, double *output)
{
  *output = (*d1>*d2?*d1:*d2);
}
```

Le tableau ci-dessous donne la correspondance entre les types R et les types du langage C [6]. Comme indiqué en section 2.1.3, le programmeur doit assurer la cohérence entre les données envoyées par l'interface et le profil de la fonction C.

Type R	**Type C**	**Type interne** (SEXPTYPE)
logical	int *	LGLSXP
integer	int *	INTSXP
double	double *	REALSXP
complex	Rcomplex * (type de R.h)	CPLXSXP
character	char **	STRSXP
raw unsigned	char *	RAWSXP
list		LISTSXP
vector		VECSXP

Remarque 3 - REAL et NUMERIC

La dénomination des types pour les nombres à virgule (double) n'est pas très stable dans la librairie de R. Parfois il est question de NUMERIC (principalement dans Rdefines.h, compatible avec S) et parfois de REAL comme ci-dessus. Cette inconstance est regrettable.

! Attention ! - Attention aux NULL

L'utilisation de notre fonction dans le cas ci-dessous va provoquer une grave erreur (*segmentation fault*). Le cas de passage d'une variable **NULL** n'a pas été prévu dans notre code.

```
a <- 34
b <- NULL
m<-max(a, b)
```

Trois solutions possibles pour éviter ce problème aux utilisateurs :
- Prévoir les cas de pointeurs nuls dans la fonction C (*cf.* exemple suivant),
- Prévoir les cas de variables R nulles dans la fonction d'enrobage définie en R,
- Faire les deux protections ci-dessus (!).

Exemple 2 - Maximum deux à deux

Les variables de R étant par défaut des vecteurs, il est préférable de proposer des fonctions qui vont traiter des vecteurs. On modifie donc le code de notre programme pour traiter des vecteurs : on cherche le maximum dans un tableau de nombres.

```
1 dyn.load("max.so")
2
3 max <- function (d)
4 {
5   out=0
6   tmp <- .C("max", as.double(d), as.integer(length(d)), as.double(out))
7   return(tmp[[3]])
```

```
}

v <- c(4.45, 5.54, 55.4, 45.5)
m<-max(v)
print(m)
```

```
#include <R.h>

void max(double *d, int *n, double *output)
{
  if (!output) return; // Verification de la nullite de output
  *output=-1;
  if (!d || !n) return; // Verification de la nullite de d et n
  int i=0;
  *output=d[0];
  for (i=0; i<*n; i++) {
    if (*output< d[i])
      *output=d[i];
  }
}
```

! Attention ! - Taille des tableaux

Vous savez que vous récupérez un pointeur sur un tableau, mais vous ne savez pas quel sera la taille du tableau ! Il peut être très utile d'ajouter des paramètres pour connaître la taille du tableau à traiter.

Remarque 4 - Utilisation sur différentes structures de données R

Avec l'interface `.C`, votre traitement sur des vecteurs fonctionnera aussi bien sur des variables de type `list` ou de type `array`.
Notez néanmoins qu'une liste dans R ne comporte pas nécessairement des objets de même type ... là, je ne sais pas comment ça se passera pour votre programme, mais il faut s'attendre à un beau bug !
L'utilisation des listes nécessite d'utiliser une interface plus complexe !

2.2.2 Interface `.Call` : extension complète

Definition 2 - Fonction pour l'interface `.Call`

Une fonction C qui peut être chargée par l'interface `.Call` est une fonction dont
- les arguments passés sont de type SEXP (*cf.* définition 2.2.2 ci-dessous). Le nombre de paramètre est limité à 65 ... je pense que ça devrait aller pour la plupart des usages.
- le type de retour de la fonction doit être SEXP.

L'utilisation du type SEXP nécessite a définition d'une fonction avec l
Le résultat de la fonction `.Call` est une *liste* contenant les valeurs des arguments à la fin de la fonction.

Definition 3 - Types SEXP et SEXPREC

SEXP correspond à un `pointeur` sur une variable de type **SEXPREC**. Ces types de données correspondent aux représentations internes des variables de R. Le terme SEXP signifie à "*Simple-Expression*".

Nous ne rentrerons pas dans les détails de la structure de données **SEXPREC**. Il faut imaginer que, comme pour une variable R, c'est une variable qui peut contenir différents type de données (c'est une union). Pour plus de détails, vous pouvez consulter [6] ou le fichier `Rinternals.h`.

Exemple 3 - Profil de la fonction max

Continuons à créer des fonctions de calcul de maximum et revenons au cas d'un max de valeurs. Le profil de la fonction que nous aurons à implémenter sera le suivant :

```
SEXP max(SEXP d1, SEXP d2)
```

On peut alors faire deux remarques :
- *Nous n'avons plus besoin de passer par un argument de fonction pour faire un retour, nous pourrons utiliser l'instruction return et renvoyer une variable de type SEXP*
- *Les deux arguments sont des variables utilisées par R ... mais qu'est ce qu'il y a dedans ? ? comment récupérer leurs valeurs ?*

La bibliothèque `Rinternals.h` propose des "fonctions" (plutôt des macros) pour extraire les données d'un SEXP. Si `x` est une variable de type SEXP,
- **CHAR**(`x`) : récupère un pointeur sur un caractère (chaîne de caractères),
- **INTEGER**(`x`) : récupère un pointeur sur un entier,
- **REAL**(`x`) : récupère un pointeur sur un double.

Bien évidemment, pour les utiliser, il faut connaître le type des variables que contient la variable `x`. Ceci est possible en imposant correctement la correspondance des types entre la fonction enrobant l'utilisation de l'interface `.Call` (peu générique et dangeureux), ou en utilisant des fonctions C pour tester les types des variables (*e.g.* **isString**, **isReal**, **isInteger** ... et des fonctions comme **length**). Nous ne rentrons pas dans les détails de cette seconde solution.

Exemple 4 - Fonction max avec `.Call`

Voici donc la fonction **max** *avec l'utilisation l'interface* `.Call`*. Le premier code illustre l'utilisation de la fonction* `.Call`*. On constate que la variable récupérée en retour de* `.Call` *contient uniquement le résultat de notre fonction.*
Pour la fonction en C, les lignes 7 et 8 servent à récupérer les valeurs des variables passées en paramètre. Aucune vérification du type réel des variables R n'est effectué. La variable **output** *sert comme variable de retour. Comme un SEXP est fondamentalement un pointeur, il faut donc penser à allouer de la mémoire pour la variable grâce à la fonction allocVector (*cf. *Section 2.3.2. De plus, on utilise les macros PROTECT et UNPROTECT pour indiquer à R que cette variable est en cours d'utilisation (*cf. *Section 2.3.1).*

```r
dyn.load("max.so")

max <- function (d1, d2)
{
  tmp <- .Call("max", as.numeric(d1), as.numeric(d2))
  return(tmp)
}

m<-max(4.45, 5.54)
print (m)
```

```c
#include <R.h>
#include <Rinternals.h>

SEXP max(SEXP d1, SEXP d2)
{
  SEXP output;
  double dd1 = REAL(d1)[0];
  double dd2 = REAL(d2)[0];
  PROTECT( output = allocVector(REALSXP, 1) );
  REAL(output)[0] = (dd1>dd2?dd1:dd2);
  UNPROTECT(1);
  return output;
}
```

Remarque 5 - Valeur de non-retour

En cas d'absence de retour, il est possible R_NilValue comme valeur de retour.

Remarque 6 - Copie des arguments dans l'interface `.C`

Avec l'interface `.C`, les objets de R sont copiés avant d'être transmis à la fonction C même si se sont des pointeurs. Ils sont également recopié dans la liste de variables R lors du retour de la fonction. Lorsque vous utilisez les variables de type SEXP vous manipuler directement les mêmes variables que R, celles-ci ne sont donc pas recopiées, c'est un gain d'efficacité notable.

! Attention !

Vous devez traiter les paramètres de la fonction uniquement en lecture et ne pas les modifier sans protection particulière. Pour les modifier, il est nécessaire de les "protéger" à l'aide de la fonction PROTECT que nous verrons plus en détail dans la section 2.3.1.

Exemple 5 - Récupération d'un vecteur d'entiers / macros de `Rdefines.h`

`Rdefines.h` propose des macros et des fonctions plus avancées que `Rinternals.h`. Dans l'exemple ci-dessous, on donne une fonction qui illustre comment récupérer un tableau d'entiers directement issu d'un vecteur R en utilisation l'interface `.Call`.
On trouve dans cet exemple l'utilisation de fonctions comme GET_LENGTH permettant de récupérer la longueur d'un vecteur dans un SEXP, AS_INTEGER pour convertir proprement un SEXP en INTSXP ou encore INTEGER_POINTER pour récupérer le pointeur sur le tableau des données.

```
#include <R.h>
#include <Rdefines.h>

SEXP getInt(SEXP myint) {
  int *Pmyint;  // Pmyint sera un pointeur sur notre vecteurs d'entiers

  PROTECT(myint = AS_INTEGER(myint)); //Recuperation du parametre comme un vecteur d'entiers !
  Pmyint = INTEGER_POINTER(myint); //Recuperation de l'adresse memoire du tableau d'entiers
      correspondant

  int n = GET_LENGTH(myint); //Recuperation de la taille du tableau
  int i=0;
  for (;i<n; i++) printf("[%d] %d\n",i, Pmyint[i]);
  UNPROTECT(1);

  return(R_NilValue);
}
```

Pour connaître les différentes macros très utiles de `Rdefines.h`, vous pouvez directement consulter ce fichier (il faut simplement le rechercher sur votre ordinateur ...). Il est très lisible.

Exemple 6 - Calcul du min et du max d'un vecteur de réels

L'exemple ci-dessous permet d'illustrer une fonction qui prend en argument un vecteur à traiter et qui retourne également un vecteur (de taille 2). Le vecteur retourné comprend la valeur minimal et maximale du vecteur en entrée.

```
#include <R.h>
#include <Rdefines.h>

SEXP maxmin(SEXP v)
{
  SEXP output;
  int i;
  double *p, *q;

  PROTECT( v=AS_NUMERIC(v) );

  int size = GET_LENGTH(v);
  p = NUMERIC_POINTER(v);

  PROTECT( output = allocVector(REALSXP, 2) );
  q = NUMERIC_POINTER(output);

  //Calcul du max
  q[0] = p[0];
  q[1] = p[0];
  for( i=0; i< size; i++) {
```

```
22      REAL(output)[0] = (p[i]<q[0]?q[0]:p[i]) ;
23      REAL(output)[1] = (p[i]>q[1]?q[1]:p[i]) ;
24    }
25
26    UNPROTECT(2); //liberation de output et v
27    return output;
28  }
```

! Attention ! - Pensez à retourner qqc ! !

Il est très important que votre fonction retourne un SEXP ! Si votre fonction ne doit rien retourner alors il faut retourner la valeur R_NilValue. Contrairement à beaucoup de librairie qui représente leur constante "valeur nulle" par 0, la constante R_NilValue correspond à autre chose (ben oui ! un SEXP est un pointeur !).
Si vous oubliez le return, **le compilateur ne vous blâmera pas nécessairement** et risque d'imaginer que vous souhaitiez faire un return 0;. S'il vous venait cette idée, ce serait un pur attentat contre R qui planterait lors de l'utilisation de votre valeur de retour !

2.2.3 Interface `.External` : extension la plus générale

Definition 4 - Fonction pour l'interface `.External`

Une fonction C qui peut être chargée par l'interface `.External` est une fonction dont
- tous les arguments sont rangés dans une unique liste R passés en paramètre de la fonction C. La fonction n'a donc qu'un seul argument de type SEXP. Il n'y a plus de limitation de nombre de paramètre.
- le type de retour de la fonction doit être SEXP, comme pour la fonction `.Call`.

L'interface `.External` est la plus générale. La seule différence avec l'interface `.Call` est dans le passage des paramètres. Sinon tout est identique.

Exemple 7 - Profil de la fonction max

Toujours sur la fonction de calcul de maximum, le profil de la fonction que nous aurons à implémenter sera le suivant :

```
SEXP max(SEXP args)
```

mais on peut remarquer que se profil de fonction sera toujours le même quelque soit le nom de la fonction : 1 seul argument de type SEXP et un type de retour SEXP.

Les macros ci-dessous offrent un accès facile aux quatre premiers arguments de la liste de paramètres (variable **args**).

```
SEXP first = CADR(args);
SEXP second = CADDR(args);
```

```
SEXP third = CADDDR(args);
SEXP fourth = CAD4R(args);
```

De manière beaucoup plus générique, il est possible d'utiliser les macros ci-dessous pour extraire tous les arguments de la liste :

```
args = CDR(args); a = CAR(args);
args = CDR(args); b = CAR(args);
```

La fonction **CDR** "dépile" un élément de la liste des arguments, *i.e.* enlever le élément premier élément de la liste. Tandis que la fonction **CAR** accède au dernier élément "dépilé" de la liste, pour récupérer sa valeur (variable de type SEXP).

Je ne crois pas qu'il soit bien nécessaire de rentrer plus dans les détails de cette fonction. Son utilisation ne se justifie que pour un nombre d'argument au delà de 65 (contrairement à `.Call`) ... je ne vois pas bien qui cela concerne??

2.3 Gestion de la mémoire

R gère lui même sa mémoire avec un *garbage collector*. Le *garbage collector* est un petit démon[3] qui cherche les emplacements non-protégés pour les libérer.
Si vous ne voulez pas de mauvaises surprises (désallocation de votre mémoire sans que vous ayez rien demandé) il faut protéger les espaces mémoire allouées! On verra ensuite comme allouer de la mémoire utilisable dans votre code C mais également après l'exécution de votre fonction, dans le R principal.

2.3.1 Protection des variables contre le *garbage collector* de R

Dès que vous utiliser des variables de type SEXP, il faut penser qu'il s'agit de variables gérées par R. Votre fonction ne sera qu'utilisatrice de cette variable même si c'est elle qui l'a créée.
En R, **toutes les variables doivent être protégées avant d'être utilisées**. C'est grâce à ce mécanisme de protection que le *garbage collector* peut fonctionner correctement.
Lorsque vous programmer une fonction avec l'interface `.Call` ou `.External`, vous disposez de deux macros pour gérer la protection de vos variables :
– PROTECT(SEXP x) : indique à R de protéger un emplacement mémoire,
– UNPROTECT(int n) : dé-protège les *n* derniers emplacements protégés.
Il faut noter que la macro PROTECT permet également de rapporter l'existante d'une variable nouvelle créée. Toute variable ayant pour but d'exister dans R (en dehors de la fonction C) doit protégée.

Exemple 8 - Protection de la mémoire

L'exemple ci-dessous illustre des protections de variables SEXP. La protection est réalisée en même temps que leur déclaration.

```
SEXP ab, i;
PROTECT(ab = NEW_NUMERIC(1)); //Allocation d'un nombre reel et protection
PROTECT(i = NEW_INTEGER(1)); //Allocation d'un nombre entier et protection

SEXP val;
PROTECT( val = allocVector(REALSXP, 3) ); // Alloue un tableau de 3 reels et protege tout le tableau
```

3. Un démon est un petit programme transparent qui fonctionne en permanence.

```
REAL(val)[0] = 128.4; //Attribution de la valeur au premier lment du tableau
REAL(val)[1] = 28.4;
REAL(val)[2] = -456.123;

REAL(ab)[0] = REAL(val)[1];

UNPROTECT(1); //libere val
//ICI, on ne doit pas utiliser val !!

REAL(i)[0]=REAL(ab)[0];//Copie du contenu de ab dans i

UNPROTECT(2); //libere ab et i
```

La protection des variables fonctionne sur le principe d'une pile[4]. C'est-à-dire qu'une variable est déposée sur une pile (imaginer une pile de papier avec le nom de la variable à protéger), et lorsque vous faites appel à un UNPROTECT, la première feuille de la pile est enlevée, donc la variable correspondante est libérée. **Il n'est pas possible de désigner une variable à libérer**.
La protection des variables est donc nécessairement imbriquée. Dans l'exemple 2.3.1, une fois que la variable **val** a été protégée, il n'est plus possible de libérer la variable **i** directement. Il faudra d'abord libérer **val**.

En cas d'erreur dans les protections, vous pouvez voir apparaître un message d'avertissement comme ci-dessous. Même si ce n'est qu'un *warning*, il est préférable de résoudre cette erreur qui a des conséquences sur la protection de toutes les autres variables (décalage de la pile).

```
Warning: stack imbalance in '.Call', 36 then 37
Warning: stack imbalance in '<-', 34 then 35
```

Remarque 7 - Inutile de protéger les variables passées en paramètres

Les variables passées en paramètre d'une fonction C appelée par une interface n'ont pas besoin d'être protégées dans la fonction. Si elles ont été passées en paramètre, c'est que leur usage aura été annoncé au *garbage collector* avant l'appel. Il n'y a donc pas de risque de leur suppression inopinée.

2.3.2 Allocation de mémoire pour des SEXP

L'allocation de la mémoire d'une variable destinée à être utilisées dans R doit se faire au moyen de la fonction d'allocation allocVector(**SEXPTYPE type**, **R_len_t taille**).
Les variables R étant toutes des vecteurs (car un SEXP est un pointeur, donc un tableau), la fonction permet d'allouer un tableau de valeurs d'un même type. Cette fonction prend ainsi deux paramètres :
- **type** : le type des variables R (**CHARSXP**, **INTSXP**, **REALSXP**, **VECSXP**, etc.)
- **taille** : la taille du tableau, c'est-à-dire le nombre d'éléments du tableau. Peu importe le type interne (**R_len_t**), vous pouvez simplement mettre un entier (**int**).

Tous les éléments sont bien de même type. Si vous souhaitez faire des listes R, c'est-à-dire des structures de données R qui contiennent des vairables de type différents, ceci se représente en interne comme un vecteur de SEXP, *i.e.* un **VECSXP**. Pour plus de détails sur les listes, consulter la Section 2.4.

4. Peu d'entre vous savent ce qu'est une pile, mais pour les lecteurs ayant pratiqué un peu d'algorithmique, cela les aidera ! Je les aiderai encore plus si je dis que c'est une pile FILO (soit *First In, Last Out*).

Remarque 8 - Libération de la mémoire allouée

L'avantage du *garbage collector*, c'est que vous n'avez pas à vous soucier de la libération de la mémoire.

Remarque 9 - Peut on continuer à utiliser des malloc/ free comme avant ?

Tout d'abord, je vous remercie de cette question intéressante ... à laquelle je vais donner une réponse assez peu claire, car je n'ai pas trouvé cette réponse dans les différentes documentations.

Disons que si vous avez besoin de variables locales à votre programme C, *i.e.* des variables qui ne seront jamais utilisées à l'extérieur de l'appel de la fonction C utilisée dans l'interface `.Call` ou `.External`. Alors, il semble possible d'utiliser les fonctions **malloc** et **free** pour gérer vous même votre mémoire (ça peut être utile si vous croyez être plus malin que le *garbage collector*). Ceci sera d'autant plus vrai si vous utiliser des variables statiques (le mot clé **static** n'ayant pas été abordé dans ce cours, je vous invite à explorer ailleurs pour comprendre cette remarque !).
Quelques remarques :
– le *garbage collector* ne traquera pas votre espace mémoire,
– éviter d'utiliser vos allocations personnalisées sur des variables de types **SEXP** (déjà que c'est pas très catholique, il faudrait pas n'énerver trop !),
– vous ne disposerez pas de plus de mémoire qu'avec les allocations de R.
Mais, il reste un doute ... donc mieux vaut éviter de les utiliser trop.

2.4 Utilisation des listes

La structure de données les plus fréquemment utilisées en R est certainement la liste. Cette structure de données est complexe puisqu'il est possible, entre autre :
– de nommer les éléments de cette liste,
– d'avoir des éléments de différents types (y compris d'autres listes)
Dans cette partie, on propose un exemple tiré de [4] pour créer une liste dans une fonction C qui pourra être appelée par `.Call` ou `.External`.

Definition 5 - Listes R

Une liste R est une structure de données qui contient plusieurs "variables" pouvant être de types différents. D'un point de vue C, on peut imaginer que c'est une liste de **SEXP**, c'est à dire des pointeurs, mais du fait de la nature pluriforme d'un **SEXP**, l'élément pointé peut être de différents types.
La particularité de la structure de liste de R est que chaque élément de la liste peut être identifiée soit par sa position dans la liste (un index) soit par un nom.

Intimement parlant, une liste R reste une variable R, c'est donc un **SEXP**, mais qui pointe vers un **VECSXP**, c'est à dire un vecteur de **SEXP**.

Pour traiter les listes, nous allons devoir expliquer comment les manipuler en détaillant : 1) comment allouer une liste en mémoire, 2), comment donner un nom aux éléments de notre liste, 3) comment remplir les éléments d'une liste, 4) comment accéder aux éléments d'une liste. À chacune de ces étapes, des fonctions ou macros de `Rdefines.h` seront introduites.

Allocation d'une liste R

L'exemple ci-dessous illustre comment allouer une nouvelle liste **list** comprenant 2 valeurs :

```
SEXP list;
PROTECT( list = allocVector(VECSXP, 2) );
```

Donner des noms aux éléments de la liste

Pour donner un nom aux éléments de la liste, il faut attribuer (fonction **setAttrib**) un vecteur de chaîne de caractère à notre liste. Pour cela, il faut créer ce vecteur de nom, **name** dans l'exemple ci-dessous. Le premier élément de la liste s'appellera `random_values` et le second `size`.

```
SEXP names;
PROTECT( names = allocVector(STRSXP, 2) );
SET_STRING_ELT(names, 0, mkChar('random_values') );
SET_STRING_ELT(names, 1, mkChar('size') );

setAttrib( list , R_NamesSymbol, names);
```

Donner des noms aux éléments de la liste

Notez que pour le moment, on a juste créé une liste avec deux éléments et leurs noms, mais on n'a pas spécifier ce qu'il va y avoir dedans ! En particulier, nous n'avons pas choisi le type des données à y mettre. Dans la suite de l'exemple, nous allons mettre un vecteurs de 10 réels dans `random_values` et la taille de ce vecteur, un entier, dans `size`.
Pour cela, il faut allouer la mémoire nécessaire pour toutes ces données. Comme se sont des objets R, on utilise naturellement l'allocation mémoire des **SEXP**.
Les deux dernières lignes de l'exemple ci-dessous permettent d'associer le premier élément de **list** (index 0) au **SEXP dim1** et le second élément de **list** (index 1) à **dim2**.

```
SEXP dim1, dim2;
PROTECT( dim1 = allocVector(REALSXP, 10) );
for(int i=0; i<10; i++)
  REAL(dim1)[0] = rnorm(0, 1.0);

PROTECT( dim2 = allocVector(INTSXP, 1) );
INTEGER(dim2)[0] = 10;

SET_VECTOR_ELT(list, 0, dim1);
SET_VECTOR_ELT(list, 1, dim2);
```

Accès aux éléments d'une liste par nom

Nous reprenons ici la fonction **getListElement** de [4] pour faciliter l'accès aux éléments d'une liste par nom. Cette fonction parcours la liste des noms de la liste et retourne l'élément de la liste correspondant. Si aucun élément ne correspond à ce nom, alors la fonction retourne **R_NilValue**.

```
SEXP getListElement(SEXP list, const char *str) {
  SEXP elmt = R_NilValue;
  const char *tempChar;
  int i;
  PROTECT(SEXP names=getAttrib (list, R_NamesSymbol));
```

```
  for(i=0; i<length(list); i++) {
    tempChar = CHAR(STRING_ELT(names, i));
    if( strcmp(tempChar,str) == 0) {
      elmt = VECTOR_ELT(list, i);
      break;
    }
  }
  UNPROTECT(1);
  return elmt;
}
```

Utilisation des paramètres d'une fonction

```
SEXP viewValue(SEXP arglist)
{
  int val=0;
  SEXP sval;

  if( !IS_LIST(arglist) )
    error("la fonction attend une liste comme argument");

  sval = getListElement(arglist, "value");
  if( sval==R_NilValue )
    error("Impossible de trouver l'lment 'value' dans la liste");

  PROTECT(val=INTEGER(sval)[0]);

  Rprintf("L' lment  est %d\n", val);
  UNPROTECT(1);
}
```

2.5 Débugger un programme C interfacé avec R

Et oui, même si vous bénéficier d'un super cours je ne peux pas m'engager contractuellement sur le fait que vous ne fassiez plus d'erreurs ... alors autant les anticiper.

2.5.1 Gestion des erreurs

Definition 6 - Fonction error

La fonction `void error(const char *)` est une fonction qui permet de générer proprement une erreur dans votre fonction C. L'appel de cette fonction met immédiatement fin à l'exécution de la fonction et affiche le message d'erreur dans la console R.

Remarque 10 - Comportement avec PROTECT

Je ne sais pas comment est géré la pile de protection dans le cas d'une erreur. Il ne me semble pas que les variables préalablement protégées soient libérées automatiquement. Il est donc préférable de libérer manuellement les variables qui auraient été protégées pour éviter les problèmes de déséquilibre de PROTECT.

2.5.2 Quelques erreurs

TODO : à compléter au gré de l'expérience ...

Débordement de pile

Vous pouvez rencontrer le problème de débordement de la pile (*stack overflow*) 1) si vous exécutez le code C sur un ensemble de données énormes dans R ou 2) si vous utilisez un programme récursif. Pour le second cas, chaque appel récursif utilise un peu de mémoire de la pile, mais si les appels récursifs sont nombreux, on peut facilement dépasser l'espace de stockage. La suppression de la récursivité peut parfois résoudre le problème.
La taille de la pile est une limite imposée par le système d'exploitation. La seule façon de résoudre ce problème consiste à manuellement allouer plus de mémoire physique pour la pile en modifiant les paramètres de votre système d'exploitation. Sous Linux, Vous pouvez connaître les limites intrinsèques à votre système en utilisant la commande `ulimit -a`.

2.5.3 Utiliser le débuggeur C dans R

CHAPITRE

3 Utiliser des fonctions R dans C

On part du principe que le programmeur cherche à réaliser un programme en langage C qui nécessite des fonctionnalités statistiques riches. Le programmeur fainéant (donc malin) propose d'utiliser les fonctions de R pour son besoin plutôt que de refaire (en moins bien) les fonctions statistiques utiles. On commence par présenter une partie facile ... puis on donne quelques idées sur les possibilités d'intégration de R dans des programmes écris en C sans rentrer dans les détails (pas seulement pour des raisons d'accessibilité du lecteur, mais parce que votre serviteur n'a pas pris le temps de rentrer dans les entrailles profondes de R ... lui aussi à ses limites !).

3.1 Utilisation de fonctions de base de R dans un programme C autonome

Si l'installation de R le permet, c'est-à-dire si cela a été prévu à l'étape de configuration, l'utilisation de la librairie `Rmath.h` de R est possible depuis un programme C autonome, dite *standalone*. Il est possible d'utiliser en C certaines fonctions mathématiques R en incluant le fichier d'en-tête `Rmath.h` dans votre code C par la ligne `#include <Rmath.h>` Ceci nécessitera néanmoins que vous ayez installé R en précisant que vous souhaitiez cette possibilité !

Les fonctions statistiques qui peuvent être directement utilisées dans un programme C comprennent :

- des fonction de génération aléatoire (normales, uniformes, exponentielles).
- des distributions (Uniform, Normal, Chi-2, Student, Poisson, Binomial, Multinomial, Hypergeometric, Geometric, F, logistic, Beta, Exponential, Cauchy, Gamma, Wilcoxon, Weibull),
- des fonctions liées à la fonction Gamma,
- les fonctions de Bessel.

```c
#define MATHLIB_STANDALONE
#include <Rmath.h> // inclusion du fichier de R qui declare les programmes
#include <stdio.h>
void ma_norm(int N, double *res) {
  int i;
  for (i=0; i< N; i++) {
    res[i] = norm_rand(); //Utilisation de la fonction norm_rand de Rmath.h

    printf("res[%d] = %f\n", i, res[i]);
  } // fin i

}

int main() {
  int N=5;
  double res[N];
  ma_norm(N, res);
  return(0);
}
```

En début de programme, avant l'inclusion de `Rmath.h`, nous avons placé une variable d'environnement (`#define MATHLIB_STANDALONE`) pour indiquer que nous utiliserons les fonctions de R en dehors de R.

Exemple 9 - Génération de nombres aléatoires

Voici un exemple permettant d'utiliser un générateur aléatoire de R.

```c
#include <stdio.h>
#include <time.h>
#define MATHLIB_STANDALONE
#include "Rmath.h"

void main(){
  int i,n;
  double sum;

  // Initialisation du generateur aleatoire
  set_seed(time(NULL),77911);

  printf("Nombre de tirage ?");
  scanf("%d",&n);

  sum = 0.;
  for(i=0;i<n;i++){
    sum += rnorm(0,1);
  }

  printf("La moyenne est %lf\n",sum/(double)n);
}
```

La commande de compilation doit spécifier l'emplacement de la librairie Rmath qui contient les implémentations compilées des fonctions R. Cet emplacement varie selon les installations. Par exemple (chez moi) :

```
gcc -o rd -I/usr/share/R/include -L/usr/lib/R/lib -lRmath -lm Rmath_genrand.c
```

3.2 Évaluer des expressions R dans un programme C

! Attention !

! Cette partie est à faire !

3.3 Intégration R dans d'autres applications

! Attention !

! Cette partie est très préliminaire !
Voir `http://developer.r-project.org/embedded.html` pour plus informations (illisibles).

Sous Linux, il est possible de compiler R comme une bibliothèque autonome qui pourra être chargée dynamiquement dans d'autres applications. Il est alors possible de lancer le *moteur* de R directement depuis ces autres applications. L'intérêt est alors de bénéficier de toute la puissance de R (plus étendue que si on utilise uniquement la version autonome) dans un logiciel sans que l'utilisateur n'est à lancer R ou même installer R. Une fois le *moteur* de R lancé, il est possible d'utiliser l'interface de programmation défini dans les extensions R pour évaluer les expressions de R, appeler des fonctions de R, accéder aux routines mathématiques, etc comme il a été vu dans la section précédente.

3.3.1 Initialisation de R embarqué dans une application

La fonction **Rf_initEmbeddedR** initialise le moteur R et **Rf_endEmbeddedR** l'arrête. Ces fonctions sont disponibles depuis le fichier `Rembedded.h`.

```
1  #include <Rinternals.h>
2  #include <Rembedded.h>
3
4  SEXP hello() {
5    return mkString("Hello, world!\n");
6  }
7
8  int main(int argc, char **argv) {
9    SEXP x;
10
11   Rf_initEmbeddedR(argc, argv);
12   x = hello();
13   Rprintf("%s\n",CHAR(STRING_ELT(x,0)));
14   Rf_endEmbeddedR(0);
15
16   return 0;
17 }
```

3.3.2 Compilation et exécution d'une application embarquant R

Pour compiler sous Linux, je vous conseille la commande suivante, en supposant que le fichier s'appelle `test_embed.c` :

```
gcc -o test_embed `pkg-config --cflags --libs libR` test_embed.c
```

Notez que la commande de compilation comprend des *backquotes*, guillemets à l'envers pour faire appel à la commande `pkg-config`.

Lorsque ce programme est appelée, les variables d'environnement telles que `R_HOME`, `R_PROFILE`, `R_LIBS` doivent être correctement configurées (en particulier `R_HOME`). Si ce n'est pas le cas, il peut y avoir une erreur. Pour exécuter la commande, vous pouvez alors utiliser la commande suivante (qui assure que la variable `R_HOME` est bien définie) :

```
R_HOME=/usr/lib/R ./test_embed
```

En exécutant le programme, vous pourrez observer que R a été lancé par l'information de version traditionnellement affiché lors de l'exécution d'une console R. Pour masquer cet affichage, ajouter l'option `-quiet` ainsi :

```
R_HOME=/usr/lib/R ./test_embed --quiet
```

Voilà pour le principe ... toute la suite est suffisamment complexe pour que je l'élude !

CHAPITRE

4 Conclusion

Les deux langages R et C s'opposent fortement dans leur philosophie. Il est donc très intéressant de les croiser. Pour les applications R, le gain d'efficacité et la meilleur gestion de la mémoire est très intéressant pour arriver à traiter de grand jeux de données. Pour les applications C, l'accès aux fonctionnalités de haut niveau en statistiques pallie un manque de développement de librairie dans ce domaine. Il faut noter également que la facilité d'insertion de code C dans un package R rend cette solution très attrayante.
Pour profiter de cette richesse, cela à un prix ! Il est nécessaire pour le programmeur d'avoir quelques bases solides de C.

Tout ce qui est dans ce document n'est pas nécessairement utile à la production de code C pour l'extension des fonctionnalités en R. Le lecteur aura certainement intelligemment fait la distinction entre les exemples didactiques amenant une compréhension des mécanismes sousjacents et les exemples pragmatiques desquels s'inspirer pour adapter ses proposes solutions.
Si je devais vous faire un résumer pragmatique, je vous inviterai à vous concentrer sur la compréhension de l'interface `.Call`, de bien comprendre l'utilisation du type **SEXP** à l'aide des macros de `Rdefines.h`.
Ensuite, il faudra penser à tous les pièges du C et en particulier à la gestion de la mémoire par R : penser à protéger les allocations !

Ce document est certainement incomplet. Sa compréhension permettra peut être aux plus acharnés de mieux aborder les (quelques) autres documents disponibles. Mais si le lecteur n'y trouve toujours pas son compte, il peut toujours se plaindre auprès de l'auteur et lui transmettre ses doléances. Qui sait, dans un bon jour, il en tiendra peut être compte.

Annexes

Annexe A Implémentation du Quicksort

```
partition <- function( a, l, r) {
  pivot <- a[l]
  i <- l
  j <- r+1

  while( 1 )
  {
    i<-i+1
    while( a[i] <= pivot && i <= r ) {
      i<-i+1
    }
    j<-j-1
    while( a[j] > pivot ) {
      j<-j-1
    }
    if( i >= j ) break

    t <- a[i]
    a[i] <- a[j]
    a[j] <- t
  }
  t <- a[l]
  a[l] <- a[j]
  a[j] <- t
  ret <- list( "pivot" = j, "tableau" = a)
  return(ret)
}

quicksort <- function( a, l, r)
{
  if( l < r ) {
    ret = partition( a, l, r)
    a <- ret$tableau
    j <- ret$pivot
    a <- quicksort( a, l, j-1)
    a <- quicksort( a, j+1, r)
  }
  return(a)
}


nbtest <- 10
nbsize <- 20000
for( i in 1:nbtest) {
  mytime <- proc.time()
```

```
  vect <- rnorm( nbsize*i )
  vect <- quicksort(vect, 1, length(vect))
  print(proc.time() - mytime)
}
```

Listing 1 – Implémentation de Quicksort en R

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#define MAX_VAL 100

/**
* @param a tableau de donnees
* @param l position gauche du sous-tableau a trier
* @param r position droite du sous-tableau a trier
* @return la position du pivot qui separe les deux zone du sous-tableau entre les valeurs inferieures au pivot et les valeurs superieures au pivot.
*
* La fonction range le tableau a (entre l et r) par rapport a la valeur a[l] : a gauche du tableau, toutes les valeurs inferieures ou egales a a[l], a droite toutes les valeurs superieures a a[l]. Au final, a[l] est place a la limite des deux zones, a la position retournee par la fonction.
*/
int partition( int *a, int l, int r) {
  int pivot, i, j, t;
  pivot = a[l];
  i = l;
  j = r+1;

  //Rangement du tableau
  while( 1)
  {
    do ++i; while( a[i] <= pivot && i <= r );
    //ICI, toutes les positions entre l et i sont inferieures ou egales au pivot (a[l])
    do --j; while( a[j] > pivot );
    //ICI, toutes les positions entre j et r sont superieures au pivot
    if( i >= j ) break;
    // ICI, il existe des valeurs de a qui ne sont pas bien classees par rapport au pivot en particulier a[i] > pivot et a[j] < pivot, alors que i<j

    //Donc, on inverse a[i] et a[j]
    t = a[i];
    a[i] = a[j];
    a[j] = t;
  }

  //On replace pivot au centre du tableau
  t = a[l];
  a[l] = a[j];
  a[j] = t;
  return j;
}

/**
* Algorithme de tri rapide ( recursif )
* @param a tableau de donnees global a trier
* @param l index de la limite gauche a trier
* @param r index de la limite droite a trier
*/
```

```c
void quicksort( int *a, int l, int r)
{
  int j;
  if( l < r ) {
    //On separe le tableau en deux selon la valeur de a[l]
    j = partition( a, l, r);
    // a[l] est replace en j, et travaille maintenant sur les sous-tableaux :
    quicksort( a, l, j-1);
    quicksort( a, j+1, r);
  }
}

/**
* Programme de test de la fonction quicksort
*/
int main()
{
  int i,it;
  clock_t start, finish;
  int nIter=10, size=20000;

  srand(time(NULL));

  for(it=0; it<nIter;it++) {
    //Allocation memoire pour le tableau de taille dynamique
    int *a = (int *)malloc(sizeof(int)* size*(it+1));

    // Generation d'un tableau aleatoire
    for(i=0; i<size*(it+1); i++) {
      a[i]=rand()%MAX_VAL;
    }
    //Application de l'algorithme QuickSort pour la classification
    start=clock();
    quicksort( a, 0, size*(it+1)-1);
    finish = clock();
    double duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf( "%2.3fs\n", duration );
    free(a);
  }

  return 0;
}
```

Listing 2 – Implémentation de Quicksort en langage C

Bibliographie

[1] An introduction to the interface between C and R. Technical report, EPFL, 2007. http://infoscience.epfl.ch/record/112204.

[2] Sigal Blay. http://www.sfu.ca/~sblay/R-C-interface.txt.

[3] Dirk Eddelbuettel and Romain Francois. Rcpp : Seamless R and C++ integration. *Journal of Statistical Software*, 40(8) :1–18, 4 2011. http://www.jstatsoft.org/v40/i08.

[4] Gopi Goswami. CCR05 courses (Harvard). http://www.stat.harvard.edu/ccr2005/.

[5] J. Peng, R.D et Leeuw. An introduction to the .C interface to R. Technical report, UCLA Department of Statistics, 2002. http://www.ats.ucla.edu/stat/r/library/interface.pdf.

[6] R Development Core Team. Writing R extensions. Technical report, CRAN, 2012. http://cran.r-project.org/doc/manuals/R-exts.html.