Lecture: Interfaces to C (and other languages)

Martin Morgan, Hervé Pagès
Fred Hutchinson Cancer Research Center
Seattle, WA, USA

14 February, 2008
Interfaces to C (and other languages)

When to interface with C?

- Access facilities available in other libraries, e.g., Rgraphviz and the `graphviz` library for graph manipulation and visualization.
- Perform algorithms not easily captured by the R programming model, e.g., efficiently updating ‘sliding window’ statistics (??)
- Represent data structures that do not fit easily into the R programming model, e.g., Biostrings maintains a single ‘read-only’ copy of a (e.g., nucleotide) sequence, with light-weight ‘views’.
- Move slow R computations to C, for speed.
Tools for assessing code I

**codetools** Identify likely programming errors, e.g., referencing global variables.

**system.time** Overall time evaluating code chunk.

**Rprof** Profile time spent in each R function (and optionally memory-allocations).

**tracemem** Record when an object is ‘duplicated’.

**gctorture** Flags internal (C-level) errors related to R memory management.

**valgrind** C-level user memory management errors.
Caveats:

- Memory information (from Rprofmem) requires R to be built with memory profiling enabled
  - Linux: `$R_SRC/configure --enable-memory-profiling`
  - Windows:
    
    ```
    cd $R_SRC/src/gnuwin32
    make R_MEMORY_PROFILING=T
    ```

- `valgrind` is a Linux-only tool.

Useful to remember:

- *Copy on change* semantics describe (approximately) how R manages memory.
> v <- list(5)
> tracemem(v)

[1] "<0x01cc1c80>"

> v[[1]] <- 1
> w <- v
> v[[1]] <- 1

tracemem[0x01cc1c80 -> 0x01a98350]: eval.with.vis doTryCatch

> v <- list(5)
> tracemem(v)

[1] "<0x01358e88>"
Understand R’s use of memory: copy-on-change II

```r
> f <- function(x) {
  + x[[1]] <- 1
  + x
  + }
> v <- f(v)

tracemem[0x01358e88 -> 0x01d7dd08]: f eval.with.vis doTryCatch tryCatchOne tryCatchList tryCatch try evalFunc <Anonymous> Sweave

> v <- list(5)
> tracemem(v)
> tracemem(v)
[1] "<0x01d81bc8>"

> g <- function(x) x[[1]]
> g(v)
[1] 5
```
Think in R

E.g., for loops imply memory copies.

```r
> reps <- 10000
> system.time({
+   r1 <- list()
+   for (i in seq(1, reps)) r1[[i]] <- i
+ }, gcFirst = TRUE)[["user.self"]]
[1] 3.11

> system.time({
+   r2 <- lapply(seq(1, reps), function(i) i)
+ }, gcFirst = TRUE)[["user.self"]]
[1] 0.05
```
Use existing methods

E.g., running median

```r
> v <- seq(1, reps)
> window <- 10
> system.time({
+     r1 <- sapply(seq(1, length(v) - window),
+                   function(i, v, w) median(v[seq(i, i + w)]), v = v, w = window)
+ }, gcFirst = TRUE)[["user.self"]]
[1] 4

> system.time({
+     r2 <- runmed(v, window + 1)
+ }, gcFirst = TRUE)[["user.self"]]
[1] 0
```
Understand Rprof I

```r
f <- function(reps) g(reps)
g <- function(reps) {
  lst <- list()
  for (i in seq(1, reps)) lst[[i]] = i
  lst
}
Rprof()
res <- f(10000)
Rprof(NULL)
```
Understand Rprof II

```r
> summaryRprof()$by.self[c("f", "g")], ]

self.time self.pct total.time total.pct
f 0.00 0 2.06 99
g 2.06 99 2.06 99

- **self.time**: time spent *in* the function.
- **total.time**: time spent in the function, and all contained functions.
- Most effective when profiling small pieces of code.
```
Interfaces to C

.C
- R types coerced to familiar C types
- Useful for quick algorithms or simple interfaces to existing libraries.

.Call
- C-level access to R data structures and language features.
- Much greater knowledge and responsibility.
- Useful for calling R functions from C code, manipulating R objects, developing extensive library interfaces, implementing novel data structures.

Also
- .Fortran: calling Fortran code
- .Internal, .External: primarily useful for understanding R.
- Main references: *Writing R Extensions, R internals*. 
Overall approach

R-level

- Write a wrapper function to perform error checking and invoke C code.
- Load shared library into R using `dyn.load`.
- Invoke from R using `.C` or `.Call`.

C- and system-level

- Write C functions, with argument and return types depending on whether function will use `.C` or `.Call`.
- Compile as a shared library, using shell script `R CMD SHLIB`.
- Makefile and Makevars often *not* necessary.

Incorporating C code into packages.

- Add code to package `src` directory.
- Write C code to ‘register’ native routines.
- Load with `useDynLib` in `NAMESPACE` file.
The .C interface

\[
\text{noquote(names(as.list(args(.C))))}
\]

[1] name ... NAOK DUP PACKAGE

[6] ENCODING

- **name**: Name of C routine.
- **...**: Arguments to *name*, in order (names are *not* matched to C argument names).
- **NAOK**: Allow NA and other special values to be passed to C.
- **DUP**: Duplicate the object to be passed to C? Almost always a good idea to do this.
- **PACKAGE**: Look for *name* in the dll of the specified package.
- **ENCODING**: Encoding used for character vectors.

Return value is a (possibly named) list corresponding to ....
For each element $x[i]$, find the minimum of $x[i]-y[j]$ over all $j$. E.g., in file R/distance.R

```r
> minimumDistance <- function(x, y) {
+   xlen <- length(x)
+   ylen <- length(y)
+   if (ylen < 2)
+     return(x - rep(y, xlen))
+   .C("min_dist", as.numeric(x), xlen, as.numeric(y),
+       ylen, dist = numeric(xlen))$dist
+ }
```
.C example II

E.g., in file src/dist.c

#include <R.h>

void min_dist(double *x, int *x_len,
              double *y, int *y_len, double *dist) {
    int i, j;
    double cur;
    for (i = 0; i < *x_len; ++i) {
        cur = abs(x[i] - y[0]);
        for (j = 1; j < *y_len; ++j) {
            if (abs(x[i] - y[j]) < cur)
                cur = abs(x[i] - y[j]);
        }
        dist[i] = cur;
    }
}
## Important details

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>C</th>
<th>R SEXP</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>int *</td>
<td></td>
<td>LGLSXP</td>
</tr>
<tr>
<td>integer</td>
<td>int *</td>
<td></td>
<td>INTSXP</td>
</tr>
<tr>
<td>numeric</td>
<td>double *</td>
<td></td>
<td>REALSXP</td>
</tr>
</tbody>
</table>

Vectors: R is 1-based, C is 0-based.

Matricies: row-major vectors: $x[i, j]$ in R is $x[(i-1) + (j-1) * \text{nrow}]$ in C.

Transient memory management

- Calloc, Free: under complete user control – a common source of memory leaks.
- R_alloc: automatically garbage-collected on return.

Character vectors: represented as char **.

Special values: NA, Inf, etc. not allowed by default (see ?.C for how to pass to C).
The `.Call` interface

```r
> noquote(names(as.list(args(.Call))))

[1] name ... PACKAGE
```

*name* Name of the C routine.

... Arguments to *name*, in order (names are *not* matched to C argument names).

*PACKAGE* Look for *name* in the dll of the specified package.
For each element $x[i]$, find the maximum of $x[i] - y[j]$ over all $j$. E.g., in file R/maximumDistance.R

```r
maximumDistance <- function(x, y) {
  if (!all(is.finite(x)) || !all(is.finite(y)))
    stop("'x', 'y' must not be NA, NaN, Inf, -Inf")
  .Call("max_dist", x, y)
}
```

- Information about $x$, $y$ (e.g., length) accessible at the C level.
- Return value of `.Call` is the return value of `max_dist`.
.Call example II

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

SEXP max_dist(SEXP x_sxp, SEXP y_sxp) {
    if (isReal(x_sxp) == FALSE)
        Rf_error("'x' must be 'double'");
    if (isReal(y_sxp) == FALSE)
        Rf_error("'y' must be 'double'");

    int x_len = LENGTH(x_sxp),
           y_len = LENGTH(y_sxp);

    SEXP dist_sxp;
    PROTECT(dist_sxp = allocVector(REALSXP, x_len));

    double *x = REAL(x_sxp), *y = REAL(y_sxp),
           *dist = REAL(dist_sxp);
```
int i, j;
double cur;
for (i = 0; i < x_len; ++i) {
    cur = 0;
    for (j = 0; j < y_len; ++j) {
        if (abs(x[i] - y[j]) > cur)
            cur = abs(x[i] - y[j]);
    }
    dist[i] = cur;
}

UNPROTECT(1);
return dist_sxp;
}
Details and use

- PROTECT all SEXP\text{s} created in \text{C}; a \text{LISTSXP} needs to be protected, but not its elements.
- Main interface defined in \text{Rinternals.h}; alternative in \text{Rdefines.h}.
- Additional interface (also relevant to \text{.C} programming) exposed in \text{R_ext/*.h}

Use

```
% R CMD SHLIB src/dist
% R --vanilla
> dyn.load("src/dist")
> source("R/distance.R")
> minimumDistance(1:10, 2:5)
> dyn.unload("src/dist")
```
Debugging C code

Configuring R

- **Linux**: `CFLAGS="-g -O0" $R_SRC/configure`
- **Windows**: Edit `$R_SRC/src/gnuwin32/Makefile` to read `OPTFLAGS=-O0 -Wall -pedantic`, then cd `$R_SRC/src/gnuwin32` make `DEBUG=T`

Debug

- Start R, load dynamic library, attach debugger, set break points (details are system- and debugger specific)
Interfaces to other languages

- .Fortran: like .C.
- Other languages possible (e.g., Java, Python) via user-contributed packages (e.g., rJava); usually offer R-level interface to call- or eval-like functions.