Calling other languages from R

R.M. Ripley

Department of Statistics
University of Oxford

2008/9

Why or why not?

Pro:
Speed  R is not a compiled language, so some functions will be faster if written in a compiled language like C or Fortran. Check by profiling that it is likely to be worth the effort.

Convenience  You have C code already which does what you need.

Con:
Speed  Writing a compiled language usually takes longer and is much more difficult to debug and test.

Convenience  R is usually easier to understand and adapt.

How?

We will only discuss calling C (or C++): other interfaces are possible (e.g. to Fortran and Java), but the mechanisms are similar. Three different functions:

.C  designed to call code that does not know about R. Straightforward, but limited types of arguments and all checking of arguments must be done in the R. No return value, but may alter its arguments.

.Call  designed for calling code that understands about R. Allows multiple arguments and a return value (which can be a list).

.External  designed for calling code that understands about R. Passes a single argument, which the calling code must interpret. Allows a return value. Not discussed further in this lecture.

The details of .C

.C(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE, ENCODING) where the "..." is for the data you wish to pass to C.

name  The name of the C function you wish to call, as it appears in the C.

NAOK  If FALSE, R will stop with an error if there is any NA, NAN, Inf in the input data. Usually a good idea, as code that does not know about R is unlikely to know how to deal with R's values for special data.

DUP  If TRUE, the arguments are copied locally before the address is passed to the C function. Only change if you really, really know what you are doing.

PACKAGE  Can be used to specify where R should look for the name: recommended when writing a package.

ENCODING  Can be used to specify an encoding for character data.
Example of `.C`:

An example, adapted from Venables and Ripley, *S Programming*:

```c
ourdist <- function(x)
{
  n <- nrow(x)
  ans <- .C("ourCdist", as.double(x),
            as.integer(n),
            as.integer(ncol(x)),
            res=double(n*(n-1)/2))$res
  ans
}

void ourCdist(double *x, int *nin, int *pin,
               double *res)
```

Example of `.C`, continued:

- `as.double` or similar is used to coerce existing vectors into the
correct type for the C
  (See table of possible coercions/conversions)
- `double(m)` or similar is used to create an empty vector of length
  `m` for the return values
- Matrices and arrays will be passed as vectors.
- All dimensions must be passed explicitly.
- `.C` returns a list with named or unnamed elements corresponding
to the "..." of the call: here one element is named and that will
be extracted by the `$res`.

Correspondence between R and C types:

<table>
<thead>
<tr>
<th>R storage mode</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>int *</td>
</tr>
<tr>
<td>integer</td>
<td>int *</td>
</tr>
<tr>
<td>double</td>
<td>double *</td>
</tr>
<tr>
<td>complex</td>
<td>Rcomplex *</td>
</tr>
<tr>
<td>character</td>
<td>char **</td>
</tr>
<tr>
<td>raw</td>
<td>unsigned char *</td>
</tr>
</tbody>
</table>

Table: Mapping between R storage modes and C types

Notice that sometimes a vector or matrix of integer values is often
stored in R as reals. Coercion is always advisable.

Further details on `.C` interface:

- `debug` Can use `printf` within your C code. But output
  not visible in Rgui (the R console on Windows).
  Better to include the header file `<R.h>` and
  replace `printf` by `Rprintf`.
  Other debugging is platform dependent: consult
  the manual *Writing R Extensions*.

- `error` Use `error()` or `warning()`, with syntax as
  for `printf` (header file `<R.h>`)

- `random numbers` Can access R's random number generators
  Use `GetRNGstate()` at the start
  Then use `runif()` etc.
  Finally, call `PutRNGstate()`
  Header file `<Rmath.h>` will be required.

- `other functions` see the manual *Writing R Extensions* for further possibilities
Using C: compiling and linking

- Same procedure for .c and .Call
- Some tools are required: these will probably exist under Linux, but for Windows, install Rtools. You need Rtools early in your path: try not to install too many programs later that might put themselves earlier in the path. Mac OS X users will need to install the Xcode Tools.
- Compiled code is loaded as a shared object (Linux or MacOS X) or as a DLL (Windows).
- Load the object using `dyn.load()` and unload it using `dyn.unload()`. (Often via another function if within a package: see next lecture!) (On Windows must unload before recreating the DLL.)
- Create the object using `R CMD SHLIB` at a command prompt (For Windows, Start/All programs/Accessories/Command Prompt)

R CMD SHLIB

Making and loading shared libraries or dlls

For one single C source file:

```
R CMD SHLIB mysrc.c
```

For multiple .c, .o:

```
R CMD SHLIB mysrc.c myobj1.o myobj2.o
```

Then, within R,

```
dyn.load("mysrc.so") or dyn.load("mysrc.dll")
```

(.dll not needed under Windows)

If all goes well, this will create `mysrc.so` or `mysrc.dll`

If it fails, apart from compilation errors, look in the manual Writing R Extensions for advice on how to tailor the Make process.

Running our example

```
running ourdist
at command prompt
R CMD SHLIB ourCdist.c
```

```
in R
  x<- matrix(runif(9),nrow=3)
  dyn.load("ourCdist.dll")
  ourdist(x)
  [1] 0.6207019 0.7226275 0.5782042
  dyn.unload("ourCdist.dll")
## in Windows, unload if you need to alter it
```

.Call

- The call itself is very simple:

Syntax for .Call

```
.Call(name, ..., PACKAGE)
```

- But the C code is more complicated.
- The arguments are passed as a sequence of R objects
- Arguments are accessed using macros.
- Must return an R object
- Use header file `<Rinternals.h>` to get the macro definitions
An example

From Venables and Ripley, S programming: a function to convolve finite sequences:

\[ c_i = \sum_{j,k \geq 0 : j+k = 1} a_j b_k, \quad i = 0, \ldots, n_a + n_b \]

First the R, using .C

"%+%" <- function(a, b)
  .C("convolve1",
      as.double(a), as.integer(length(a)),
      as.double(b), as.integer(length(b)),
      ab=double(length(a) + length(b) -1))$ab

Now the C, using .C

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

SEXP convolve(SEXP a, SEXP b){
  int i, j, na, nb, nab;
  SEXP ab;
  na = length(a); nb = length(b); nab = na + nb -1;
  PROTECT(ab = allocVector(REALSXP, nab));
  for (i = 0; i < nab; i++) REAL(ab)[i] = 0.0;
  for (i = 0; i < na; i++)
    for (j = 0; j < nb; j++)
      REAL(ab)[i+j] += REAL(a)[i] + REAL(b)[j];
  UNPROTECT(1);
  return ab;
}
```

The lengths are no longer passed.
No space in the argument list is created for returning the result.
The arguments should be treated as read only.
The function returns a value.

An example, continued

Now the R, using .Call

"%+%" <- function(a, b)
  .Call("convolve2",
        as.double(a),
        as.double(b))

The lengths are no longer passed.
No space in the argument list is created for returning the result.
The arguments should be treated as read only.
The function returns a value.
An explanation!

- The basic object is a SEXP, a pointer to a SEXPREC
- You can find length of the object it points to using `length`
- Similarly, if the SEXP is really a matrix, you can use `nrows` to find out how many rows it has.
- Create an R object for the return value using `allocVector` or similar, with appropriate type.
- Access these R objects using function calls to `REAL` (INTEGER also available).
- Tell R not to garbage collect the vector you have allocated by using `PROTECT`.
- At the end, remove one item from the stack of protected items using `UNPROTECT(1)`
- Return the R object you have created. If no return object required, use `R_NilValue`.

A slightly faster way

More efficient version

```c
... double *xa, *xb, *xab;
    xa = REAL(a);
    xb = REAL(xb);
    xab = REAL(ab);
...
    for (i = 0; i < nab; i++) xab[i] = 0.0;
    for (i = 0; i < na; i++)
        for (j = 0; j < nb; j++)
            xab[i + j] += xa[i] + xb[j];
... }
```

REAL is a function: more efficient to call it once for each object and then use a pointer.

Running our second example

```r
running convolve

at command prompt
R CMD SHLIB convolve1.c
R CMD SHLIB convolve2.c

in R
a <- rnorm(10); b <- rnorm(10)
dyn.load("convolve1.dll")
a %+% b

## load second definition of %+

dyn.load("convolve2.dll")
a %+% b

## should get the same results!
```

One or two notes about C++

- Don’t tell the compiler that your source is C++ if it is really C: the compiler for C will give you more efficient code.
- To use C++ code, surround the functions you wish to call from R with

```c
extern "C" {

}
```
External pointers

You may find the `externalptr` useful to store pointers between calls to C or C++. A very simple example:

To store a pointer `p` to a `myobj`:

```r
SEXP Rptr;
Rptr = R_MakeExternalPtr((void *)p, R_NilValue, R_NilValue);
return Rptr;
```

and to get it back, if `Rptr`, a `SEXP`, is an argument of the call:

```r
p = (myobj *) R_ExternalPtrAddr(Rptr);
```

Exercises 9

See the project for the course.
One function can be written to use compiled code.