Using `.Call` in R

Brian Caffo

R's `.Call` Interface to C

- `.Call` is a suped up version of `.C`

  → Pass R objects to C

  → Create R objects in C

  → Manipulate R objects in C

  → Return R objects from C

  → Call R functions from C

- The "Writing R Extensions" manual is the definative source of information about `.Call`

- The manual suggests trying to write native R code first, then use `.C` then try `.Call`

Learning `.Call`

- You will only learn to use `.Call` if you start and keep using it (as with all other topics in this class.)
- Read the “Writing R Extensions” manual over and over
- `.External` is another interface which does not seem as popular within the department
- Today we’ll talk about using `.Call` in generic R code, using it while creating a package introduces minor changes
- Using `.Call` in Microsoft Windows is easy, but requires some tinkering. See Duncan Murdoch’s web page about compiling R on Windows for more information.

Running `.Call`

- `.Call` requires
  - → A C function, say `myCfunc.c`
  - → The C function to be compiled via `R CMD SHLIB`, which creates the object code `myCfunc.o` and the dll `myCfunc.so`
  - → The dll to be loaded into R, say with `dyn.load("myCfunc.so")`
  - → A `.Call` statement `.Call("myfunc", arguments)`
- I almost always use the naming convention: one C function per file, the filename is the function name plus `.c`
- I get tired of typing `R CMD SHLIB`

```
alias Rcs="R CMD SHLIB"
```

Header Files

R has several utility header files that you should include

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

An Example, Summing the Elements of a Vector

In `vecSum.c` we have the header files plus

```
SEXP vecSum(SEXP Rvec){
  int i, n;
  double *vec, value = 0;
  vec = REAL(Rvec);
  n = length(Rvec);
  for (i = 0; i < n; i++) value += vec[i];
  printf("The value is: %4.6f \n", value);
  return R_NilValue;
}
```

Executing `vecSum`

At the command line

```
R CMD SHLIB vecSum.c
```

which creates `vecSum.o` and `vecSum.so`

In an R session

```
> dyn.load("vecSum.so")
> .Call("vecSum", rnorm(10))
The value is: 3.230545
NULL
>
```

Some details

- `SEXP` is a structure defined by the R gurus. It stands for *S expression*
- Functions to be used with `.Call` should accept and return `SEXP`
- If you don't want your function to return anything use

  ```
  return R_NilValue
  ```

- The statement `vec = REAL(Rvec);` defines a pointer to the real part of `Rvec`
- This is useful so we can type `vec[0]` instead of `REAL(Rvec)[0]`
- (Remember since `vec` is a pointer, changes to it change `Rvec`)

Error checking and type coercion

- You should do error checking and type coercion
- You can do this in your C function or in an R function wrapper
- (I find it easier to do it in R)
- Example

```
vecSum <- function(vec){
  if (!is.vector(vec))
     stop("vec must be a vector")
  if (!is.real(vec)) vec <- as.real(vec)
  .Call("vecSum", vec)
}
```

Defining and returning a new SEXP

- Write a C program, `ab.c` that returns a vector of the numbers from `a` to `b`
- Coerce possibly real arguments `a` and `b` into integers in the C code
- Create and return an S expression
- Use `PROTECT` and `UNPROTECT`

The C code

In `ab.c` we have the header files plus

```
SEXP ab(SEXP Ra, SEXP Rb){
  int i, a, b;
  SEXP Rval;
  Ra = coerceVector(Ra, INTSXP);
  Rb = coerceVector(Rb, INTSXP);
  a = INTEGER(Ra)[0];
  b = INTEGER(Rb)[0];
  PROTECT(Rval = allocVector(INTSXP, b - a + 1));
  for (i = a; i <= b; i++)
     INTEGER(Rval)[i - a] = i;
  UNPROTECT(1);
  return Rval;
}
```

In an R session

```
> dyn.load("ab.so")
> .Call("ab", 1, 5)
[1] 1 2 3 4 5
>
```

Another example

- Create a function that returns upper triangular matrix

```
SEXP upTri(SEXP RinMatrix)
```

- Get the dimensions of the input matrix

```
  Rdim = getAttrib(RinMatrix, R_DimSymbol);
  I = INTEGER(Rdim)[0];
  J = INTEGER(Rdim)[1];
```

- Do some error checking and coerce to real

```
  if (I != J)
     error("Input must be a square matrix");
  RinMatrix = coerceVector(RinMatrix, REALSXP);
```

More code for `upTri`

- Allocate the memory for the returned matrix

```
PROTECT(Rval = allocMatrix(REALSXP, I, J));
```

- Set it’s values

```
for (i = 0; i < I; i++)
  for (j = 0; j < I; j++)
    if (i <= j)
      REAL(Rval)[i + I * j] =
          REAL(RinMatrix)[i + I * j];
    else
      REAL(Rval)[i + I * j] = 0;
```

- Return it

```
UNPROTECT(1);
return Rval;
```

Here’s what you get

```
> dyn.load("upTri.so")
> tmp <- matrix(1 : 4, 2, 2)
> tmp
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> .Call("upTri", tmp)
     [,1] [,2]
[1,]    1    3
[2,]    0    4
>
```

Ahhhhhhhhh, now we never have to deal with those pesky lower diagonal elements again. (Of course, R already has a function to do this.)

Final Thoughts

- You can read in, create and return lists using `.Call`
- You can get and set attributes such as rownames, dimnames etcetera
- You can call R functions in your C code
- We used vector allocation methods from `Rinternals.h`, alternative methods from `Rdefines.h` can also be used
- Look over *path to R*`/src/include/Rinternals.h` when you need to know how/if something is defined
- It's almost always better to write a "slow version" in native R first before trying any C code