Working With Data

Manning the helm during the data deluge.

Using R — Calling C Code ‘Hello World!’

Posted on November 6, 2012 by Jonathan Callahan

This entry is part 7 of 21 in the series Using R

One of the reasons that R has so much functionality is that people have incorporated a lot of academic code written in C, C++, Fortran and Java into various packages. Libraries written in these languages are often both robust and fast. If you are using R to support people in a particular field, you may be called upon to incorporate some outside code into your R environment. Unfortunately, much of the documentation on how to do this is written at a very high level. In this post we will distil some of the available information on calling C code from R into three “Hello World” examples.

Most programmers have some experience with the C language. C is low level, quite fast and is often used to write other programming languages including about 50% of R and most of the Python Standard Library. Many libraries of C code exist that can greatly enhance the speed and functionality of R for a particular use case. Most of the time, C code will be used to speed up algorithms involving loops which can be very slow in R, but there are other reasons to link R with existing C libraries.

I my case, I am creating an R package to work with seismic data available through IRIS DMC webservices. After discussions with the client we decided it would be desirable to access binary versions of the data that can be processed with an available libmseed C library. Now my C is a little rusty, not having written anything in C in over a decade, but I should be fine once I find the “Hello World” example of how to call C code from R. Ay, there’s the rub! The official R documentation has little in the way of easy examples and the highly acclaimed Rcpp package is targeted at experienced C++ programmers, not scripting-language science mashup hacks. After a little googling, I found the following sites to contain useful information for an R-calling-C beginner:

- Calling C and Fortran from R (Charlie Geyer)
- An Introduction to the .C Interface to R (Peng and de Leeuw, 2002) **
- Calling other languages from R (R.M. Ripley, 2009) *
- Calling C from R (Darren Wilkinson, 2010)
- Calling C Functions from R (Acadia Centre for Mathematical Modelling and Computation, 2010)
- System and foreign language interfaces (Writing R Extensions manual)

The short list presented above may be all you need to get started but, to be thoroughly pedantic, we’ll go through a couple of iterations of a “Hello World!” example below that explain some of the important details. These ‘baby steps’ may be helpful for those who, like myself, could use a C refresher.

Birth of C code

Just to start at the very beginning here is the classic introduction to C — `helloA.c`:

```c
#include <stdio.h>
main() {
  printf("Hello World!\n");
}
```

On Unix/Linux systems you can compile this exciting bit of C code with `make` which will take care of starting up the compiler with the right compiler flags and arguments. Then just run our new executable at the command line;

```
$ make helloA
cc     helloA.c   -o helloA
$ ./helloA
Hello World!
```

To make this executable easier to run from our R session, we'll create a file named `wrappers.R` where we will put a wrapper function that will invoke this C code:

```
# Wrapper function to invoke "helloA" at the shell.
helloA <- function() {
  system(paste(getwd(),"helloA",sep="/"))
}
```

And now we can start up R and try out our new function.

```
> source('wrappers.R')
> helloA()
Hello World!
```

OK, that example was included primarily for those out there who haven't seen C code in a while. But it walked us through all the necessary steps that we will repeat as things get more complicated: write C code; compile; write R wrapper function; invoke from R. In the example above, we didn't accept any arguments or return anything to R. We also started up two separate processes (R and 'helloA'), each with their own memory space. One of the main reasons to call C code directly from R is to avoid system calls and keep our data in a single memory space. The bigger your datasets and the more often you call your C code, the more important this becomes.

Crawling with C

The most basic method for calling C code from R is to use the `.C()` function described in the System and foreign language interfaces section of the Writing R Extensions manual. Other methods exist including the `.Call()` and `.External()` functions but we'll stick with the simplest version for this post. Here are the important points to understand about the `.C()` function in R:

- Arguments to the user compiled C code must be pointers to simple R/C data types.
- The C code must have a return type of `void`.
- With no return value, results are communicated from the C code to R by modifying one of the arguments.
- R sees the values returned by the `.C()` function as a list.

From the R Extensions manual we get a nice table of R/C data type equivalences:

R storage mode	**C type**
`logical`	`int *`
`integer`	`int *`
`double`	`double *`
`complex`	`Rcomplex *`
`character`	`char **`
`raw`	`unsigned char *`

Note that to get the type `Rcomplex` you will have to add `#include <R.h>` in your code. Also note that a "character vector" in R is a vector of strings, each of which may contain multiple characters. That's why the corresponding C type is `char **`.

So let’s change our standalone C code to make it callable via the `.C()` function. Instead of printing our greeting directly to `stdout()` we’ll return it to R and let the R session deal with it. Here is our new version of the code:

```c
void helloB(char **greeting) {
  *greeting = "Hello World!";
}
```

This time we compile it into a `.so` shared object file using `R CMD SHLIB`. This will take care of a lot of compiler flags that we really don’t want to have to remember.

```
$ R CMD SHLIB helloB.c
gcc -m32 -std=gnu99 -I/usr/include/R  -I/usr/local/include    -fpic  -O2 -g -pipe -Wall
-Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-buffer-size=4 -m32
-march=i686 -mtune=atom -fasynchronous-unwind-tables -c helloB.c -o helloB.o
gcc -m32 -std=gnu99 -shared -L/usr/local/lib -o helloB.so helloB.o -L/usr/lib/R/lib -lR
```

We’ll create an additional wrapper function in `wrappers.R` to invoke this new function. But before we can invoke our new function we will have to load the shared object file we just created with `dyn.load()`. Also note that we must pass a character string argument to our function so that memory for the string is allocated in R. Finally, we will name the arguments we pass to `.C()` to make it easy to extract our greeting from the list returned by `.C()`.

```r
# Wrapper function to invoke helloB with a named argument
dyn.load("helloB.so")
helloB <- function() {
  result <- .C("helloB",
               greeting="")
  return(result$greeting)
}
```

Running our two examples side by side we see that `helloA()` prints out immediately and returns an integer — the error code returned by the `system()` call. The function `helloB()` on the other hand, returns the character string we want.

```
> source('wrappers.R')
> greeting <- helloA()
Hello World!
> class(greeting)
[1] "integer"
> greeting <- helloB()
> class(greeting)
[1] "character"
> greeting
[1] "Hello World!"
```

Baby Steps

Now that we have successfully brought C code into R, let’s try to do some useful work without straying too far from our “Hello World!” example. We will follow the exact same recipe and create a function that accepts some input, does some computation and then returns the result. To keep things simple we’ll just count the characters in the greeting we pass to our function. Here is the C code for `helloC.c`:

```c
#include <string.h>
void helloC(char **greeting, int *count) {
 *count = strlen(*greeting);
}
```

You may now compile the code with `R CMD SHLIB helloC.c`. The R wrapper function should be a little smarter this time and should ensure that `greeting` is of type `character`. It is a very good idea to use `as.~type~()` functions to ensure that your arguments have the correct type before passing them to `.C()`.

```r
# Wrapper function to invoke helloC with two arguments
dyn.load("helloC.so")
helloC <- function(greeting) {
  if (!is.character(greeting)) {
    stop("Argument 'greeting' must be of type 'character'.")
  }
  result <- .C("helloC",
               greeting=greeting,
               count=as.integer(1))
  return(result$count)
}
```

So lets try it out to see what we get for different greetings:

```
> helloC("Hello World!")
[1] 12
> helloC("Bonjour tout le monde!")
[1] 22
> helloC("Привет мир!")
[1] 20
```

As we see, the C `strlen()` function counts the number of bytes rather than the number of characters and may return surprising results for non-ASCII strings. For our Russian greeting we have one byte each for the space and exclamation point but two bytes for each Cyrillic character. (Note to self — Be careful when working with international character sets.)

That aside aside, I hope this very low level introduction to R's .C() interface takes some of the fear out of compiling C code for use with R. Like so many things in life, it's not really that hard once you know how to do it. If you intend to move forward with the .C() interface your next stop should probably be the excellent 2002 tutorial: An Introduction to the .C Interface to R. This pdf file covers the use of the .C() interface and walks users through basic "Hello World!" examples and on to statistically useful examples including a kernel density estimator and C code for convolutions.

`.C()` *vs* `.Call()`

There has been some recent (March 2012) discussion on r-devel on the pros and cons of .C() *vs* .Call(). This discussion lays out some of the the issues concerning the .C() interface and includes links to Peter Dallgard's excellent 2004 presentation explaining R interfaces to C code. This is a *must read* if you want to delve into the gory details of R internal data types. This discussion also daylighted Simon Urbanek's 2012 Quick R API cheat sheet which colllects a lot of the details you need when using the .Call() interface.

The next post in this series, .Call("hello"), works through the same three "Hello World!" examples using the .Call() interface. The important differences between the two interfaces are summarized here:

`.C()`

- allows you to write simple C code that knows nothing about R
- only simple data types can be passed
- all argument type conversion and checking must be done in R
- all memory allocation must be done in R
- all arguments are copied locally before being passed to the C function (memory bloat)

`.Call()`

- allows you to write simple R code
- allows for complex data types
- allows for a C function return value
- allows C function to allocate memory
- does not require wasteful argument copying
- requires **much** more knowledge of R internals
- is the recommended, modern approach for serious C programmers

Series Navigation

Using R — Easier Error Handling with try()   Using R — Standalone Scripts & Error Messages

This entry was posted in R and tagged C, R, seismology. Bookmark the permalink.

9 Responses to *Using R — Calling C Code ‘Hello World!’*

Barry Rowlingson *says:*
November 7, 2012 at 10:46 am

You should stick your C code in a src folder, your R code in an R folder, use devtools, and just go load_all(). It makes R and C programming so much easier.

I wrote an Rpub document about it.

http://rpubs.com/geospacedman/lazydevtools

Jonathan Callahan *says:*
November 7, 2012 at 4:23 pm

Thanks for the tip Barry! I’m looking at your post right now and may base another “baby steps” post on your recommendations.

Igor *says:*
November 7, 2012 at 11:11 am

Thanks for such a detailed tutorial, was very useful to read. I need to deal with C++ for a last few months at my work, due to project connected with morpher.com so I’m pretty inrerested in C.

Pingback: *Using R — Calling C Code ‘Hello World’ | Working With Data | EEDSP | Scoop.it*

Steve Lianoglou *says:*
November 7, 2012 at 5:19 pm

Also, if you really want to get serious about interfacing between R C(++), I think you’d be doing yourself a disservice if you don’t have a look at the Rcpp and inline packages.

Dirk Eddelbuettel *says:*
November 7, 2012 at 6:04 pm

No need for extra packages and devtools, just follow the manual. If you drop your C, C++, … files into a directory src/, R does the right thing on all OS. You only need extra steps if you have external libraries. But __please__ do not recommend .C() in 2012 — everybody should use .Call(). See various threads on r-devel and in other places this year. You are unlikely to find an expert recommending .C() for a new project.

Jonathan Callahan *says:*
November 7, 2012 at 8:04 pm

Thanks for commenting, Dirk. I’ll look into .Call(), create a new post and update this one once I learn enough.

Tal Galili *says:*
November 7, 2012 at 10:02 pm

Hello dear Jonathan,
Great post!

Can I ask you to please open up your full feed so we could read your post on r-bloggers?

With regards,
Tal

Jonathan Callahan *says:*
November 8, 2012 at 1:20 am

I opened it up. Let me know if it's not working.
