Finding C & Fortran Routines

Duncan Temple Lang
Bell Labs, Lucent Technologies

One of the powerful features of the S language (i.e. R and S-Plus) is the facility to allow users dynamically (i.e. in the middle of a session) add and call arbitrary C/C++ and Fortran routines. The `.C()`, `.Call()`[1] and `.Fortran()` allow us to call routines in third-party libraries suc as NAG, with data managed in S. Importantly, it also allows us to develop algorithms entirely in S and then gradually move the computationally intensive parts to more efficent C code. More recently, we have generalized these interfaces to "foreign" languages to provide access to, for example, Java, Python, Perl and JavaScript.

The `dyn.load()` function in R loads a C or Fortran shared library or dynamically linked library (*DLL*) into the session and makes all the routines in that library available to the R user. The low-level details of `dyn.load()` is usually provided by the user's operating system, and in the other cases can be implemented with some clever, non-portable code. While we get much of this for free, one of the problems is that while the basic functionality is the same across different operating systems, the precise behaviour differs in subtle details from Windows to Linux to Irix to Dec. Many uses of `dyn.load()` are quite straightforward, involving C code that doesn't make use of any other libraries (e.g. the *mva* and *eda* packages). Things become more complex when that C code uses other libraries (e.g. the NAG and Lapack libraries), and significantly more variable when those third party libraries in turn use other libraries. The main issue for users is how are these other libraries found on the system. Developers have to be careful that symbols in one library do not conflict with those in other libraries and the wrong one gets called. And of course, users also have to be careful to pass the correct number and type of arguments to the routines they are supposed to be calling. Getting this wrong typically terminates the S session in an inelegant manner (read crash!).

Generally, while DLLs have many benefits, they can also be quite complicated to manage and there exists a non-trivial chance of performing computations with the wrong code and getting subtly incorrect results. The intent of explaining the different things that can go wrong is to suggest that a simpler mechanism is more appropiate for most S users. R-1.3.0 allows developers of R packages and DLLs to use a more coherent and predictable mechanism for making routines available to the `.C()`, `.Call()` and `.Fortran()`. In the next page or two, we'll take a brief look at an example of using this new mechanism and explain how it works. We'll also discuss how we will be able to use it to make accessing native code more resistant to errors and also automate aspects of passing data to C routines from S and back. The new *Slcc* package has potential to programmatically generate S and C code that provides access to arbitrary C libraries. This work is in its infancy and the purpose of this article is to encourage people to experiment with it and provide feedback.

1 Registering Routines

A call to `.C()`, `.Call()` or `.Fortran()` supplies the name of the native routine to invoke, the arguments to be passed to that routine and a non-obligatory **PACKAGE** argument identifying the DLL in which to search for the routine. The standard mechanism uses the operating system facilities to look in the DLL corresponding to the **PACKAGE** argument (or through all DLLs if the caller did not specify a value for the **PACKAGE** argument.) This lookup means that we can ask for any available routine in the library, whether it was intended to be called by the S programmer or internally by other routines in the DLL. Also, we know nothing about that routine: the number or type of arguments it expects, what it returns. One can mistakenly invoke a routine designed for use with `.Call()` using the `.C()` mechanism and end up crashing R and losing the data in the session.

A better approach is to have the DLL tell S which routines are available for which purposes. When we load the DLL, R calls the R-specific initialization routine in that DLL (named `R_init_<dll name>()`). This can be used to register routines with R for the different interface types: `.C()`, `.Call()`, `.Fortran()` and `.External()`. R stores this information about the routines and uses it when the user calls a native routine.

An example will hopefully make things clear. We will create a DLL named **myRoutines.so**. This provides two routines (*fooC()* and *barC()*) to be called via the `.C()` function and one (*myCall()*) to be access via `.Call()`. We'll ignore the code in the routine since our purpose is to illustrate how to register routines.

```
static void fooC(void)
{
  ...
}

static void barC(double *x, Rint *len)
{
  ...
}
```

[1] The `.Call()` function allows one to pass regular S objects directly between S and C.

```
static SEXP myCall(SEXP obj)
{
 return(obj);
}
```

Note that we can make these routines **static** since they are only accessed directly from within the file. This helps to minimize the potential for conflicts and finding the wrong symbol.

Now that we have these routines, we can add the code to register them. We create two arrays, one for each of the `.C()` and `.Call()` routines. The types of the arrays are *R_CMethodDef* and *R_CallMethodDef*, respectively. Each routine to be registered has an entry in the appropriate array. These entries (currently) have the same form for each routine type and consist of three entries:

S name The name by S users crefer to the routine. This does not have to be the same as the name of the C routine.

C routine This is the address of the C routine. It should be cast to type *DL_FUNC*.

argument count The number of arguments the routine expects. This is used by R to check that the number of arguments passed in the call matches what is expect. In some circumstances one needs to avoid this check. Specifying a value of -1 in this field allows this.

The last entry in each array must be `NULL` so that the R internals can count the number of routines being registered.

Returning to our example, we can now define these arrays. We include so as to get the definitions of the array types. Then we list the two entries for the `.C()` routines and the single entry in the *R_CallMethoDef* array.

```
#include "R_ext/Rdynload.h"

static const R_CMethodDef cMethods[] = {
 {"foo",  (DL_FUNC) &fooC, 0},
 {"barC", (DL_FUNC) &barC, 2},
 NULL
};

static const R_CallMethodDef callMeth-
ods[] = {
 {"myCall", (DL_FUNC) &myCall, 1},
 NULL
};
```

The very final step is to define the initialization routine that is called when the DLL is loaded by R. This is *R_init_myRoutines()* (since the DLL is named **myRoutines.so**). R calls this with a single argument used to store information about the DLL being loaded. We use th

```
void R_init_myRoutines(DllInfo *info)
{
   /* Register the .C and .Call routines.
      No .Fortran() or .External() rou-
tines here.
    */
   R_registerRoutines(info, cMethods, call-
Methods NULL, NULL);
}
```

From this point on, the library developer proceeds in the usual manner. We compile using the usual command and load the library. The only differences are internal to R and specifically how it finds routines in this library. In my example, I have a single file named and create the DLL with the command.

```
R SHLIB myRoutines.c
```

Now we start R and experiment with DLL and its routines. We first load the DLL and then call the routine *foo()*. Next, we intentionally call this with errors and see how R catches these.

```
> dyn.load("myRoutines.so")
> .C("foo")
In fooC
list()
> .C("foo", 1)
Error: Incorrect number of arguments (1), ex-
pecting 0 for foo
> .Call("foo")  # Should be .C("foo")
Error in .Call("foo") : .Call function name not in load
```

Next, we move to the `.Call()` routine *myCall()*.

```
> .Call("myCall") # no argument
Error: Incorrect number of arguments (0), ex-
pecting 1 for myCall
> .Call("myCall", 1)
In myCall
[1] 1
> .C("myCall", 1) # Should be .Call("myCall")
Error in .C("myCall", 1) : C/Fortran func-
tion name not in load table
>
```

Fortran routines and those use via the `.External()` function are handled in the same way. In our example, we specified `NULL` in the call to *R_registerRoutines()* to indicate that we had no routines in either of these categories.

One can find additional examples of how to use the registration mechanism in the packages shipped with R itself (*ctest*, *mva*, ...).

2 Future Work

Now that we have this mechanism for storing information about the R-accessible native routines in a DLL, we can make more use of it. We have seen how we can ensure that routines are called via the correct interface. In other words, we check that `.C()` routines are not called via `.Call()`, and similarly for the other interfaces. Verifying the number of arguments is convenient, especially when the author of the DLL is activel developing the code and changing the number of arguments.

We can take this one step further by specifying the types of the expected arguments in `.C()` and `.Fortran()` routines.[2] For intance, in our example, we could give the types of the two parameters of *barC()*. This might be done by modifying the routine specifications in something like the following manner

```
static const R_CMethodDef cMethods[] = {
 {"foo",  (DL_FUNC) &fooC, 0},
 {"barC", (DL_FUNC) &barC, 2, { REALSXP, INTSXP} },
 NULL
};
```

When the internal mechanism associated with the `.C()` function handles a call to *barC()* it can then check that the S objects passed in the `.C()` call have correspond to these types. We can take this one step further and allow the `.C()` mechanism actually perform any necessary conversion from the S argument to the type expected by the routine. Indeed, R-1.3.0 has a feature that allows users to specify this conversion for certain types of objects.

3 Building the Table Automatically

This registration mechanism offers the advantages that we have mentioned above. However, it requires a little more work by the developer. Since the original lookup mechanism still works, many developers may not take the time to create the arrays of routine definitions and register them. It would be beneficial to have a mechanism for reading the native code and automating the construction of the table (at least partially). There are a variety of different ways to do this. The *Slcc* is an R interface to a C parser, http://www.cs.princeton.edu/software/lcc/lcc. It can be used to process one or more C sources files and gather information about the routines, variables and data structures used within those. This information is collected as S objects and can be manipulated with regular S functions.

The *Slcc* package is in the early stages of development. It runs on Linux, but there are some minor installation details to be worked out for other

[2] This isn't as useful for `.Call()` and `.External()` since these take S objects which all have the same type.