In Search of C/C++ & FORTRAN Routines

by Duncan Temple Lang

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

In this article we discuss some of the pitfalls of the current mechanism that R uses to locate these native routines. Then we present a new mechanism which is more portable, and offers several beneficial side effects which can make using native routines more robust and less error-prone.

The current system

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. This allows S users to call any symbol in that library, including variables! The low-level details of `dyn.load()` are usually provided by the user's operating system, and in other cases can be implemented with some clever, non-portable code. While we get much of this for free, there are many subtle but important differences across the different operating systems on which R runs, Windows, Linux, Solaris, Irix, Tru64, Darwin, to name a few. And worse still, the behavior depends on both the machines and the user's own configurations. Therefore, porting working code to other platforms may be non-trivial.

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 depend on 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 that the wrong symbols do not get called, directly or indirectly. There exists a non-trivial chance of performing computations with the wrong code and getting subtly incorrect results. If one is lucky, such errors lead to catastrophic consequences and not hard to identify errors in the results.

And, of course, regardless of finding the correct routine, users also have to be careful to pass the correct number and type of arguments to the routines they are intending to call. 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 for the user to manage precisely. Why I am telling you about the potential pitfalls of the dynamic loading facility, especially when for most users things have worked quite well in the past? One reason is that as we use R in more complex settings (e.g., embedded in browsers, communicating with databases) these problems will become more common. Also, the main point, however, is that we only use a small part of the dynamic loading capabilities of the operating system but have to deal with all of the issues. A simpler mechanism is more appropriate 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()`, `.Fortran()` and `.External()` functions.

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.

The default mechanism

When one calls a native routine using one of the `.C()`, `.Call()` or `.Fortran()` interface functions, one 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.

It is common to mistakenly invoke a routine designed for use with `.Call()`, but using the `.C()` func-

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

tion. On some machines, this this can crash R and one can lose the data in the R session. For example, on Solaris this will usually cause a crash but not on Linux. Or is it on Solaris only if one uses `gcc`? or Sun's own compilers? That's really the point: we are depending on highly system-specific features that are not entirely reproducible and can be very, very frustrating to diagnose. Ideally, we want S to help out and tell us we are calling native routines with the wrong function, signal that we have the wrong number of arguments, and perhaps even convert those arguments to the appropriate types.

Registering routines

Well, there is a better approach which allows S to do exactly these things. The idea is to have the DLL explicitly tell S which routines are available to S, and for which interface mechanisms (`.C()`, `.Call()`, ...). R stores the information about these routines and consults it when the user calls a native routine. When does the DLL get to tell R about the routines? When we load the DLL, R calls the R-specific initialization routine in that DLL (named `R_init_`*dllname*`()`), if it exists. This routine can register the routines as well as performing any other initialization it wants.

An example will hopefully make things clear. We will create a shared library named 'myRoutines.so'[2]. This provides two routines (*fooC()* and *barC()*) to be called via the `.C()` function and one (*myCall()*) to be accessed via `.Call()`. We'll ignore the code in the routines here since our purpose is only to illustrate how to register the routines.

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

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

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

Now that we have defined these routines, we can add the code to register them (see figure 2). 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 type of routine and have 3 required elements:

S name The name by which S users refer 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 routine, given simply by its name in the code. 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 expected. In some circumstances one needs to avoid this check. Specifying a value of `-1` in this field allows this.

The last entry in each top-level array must be `NULL`. R uses this to count the number of routines being registered.

For our example, these arrays are defined in figure 1. The code includes the file 'R_ext/Rdynload.h' 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_CallMethodDef* 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 callMethods[] = {
   {"myCall", (DL_FUNC) &myCall, 1},
   NULL
};
```

Figure 1: Defining the registration information

The very final step is to define the initialization routine that is called when the DLL is loaded by R. Since the DLL is called 'myRoutines.so', the name of the initialization routine is *R_init_myRoutines()*. When the DLL is loaded, R calls this with a single argument (`info`) which is used to store information about the DLL being loaded. So we define the routine as follows:

```
void R_init_myRoutines(DllInfo *info)
{
 /* Register the .C and .Call routines.
    No .Fortran() or .External() routines,
    so pass those arrays as NULL.
  */
 R_registerRoutines(info,
                   cMethods, callMethods,
                   NULL, NULL);
}
```

Figure 2: Registering the `.C()` and `.Call()` routines

From this point on, the library developer can proceed in the usual manner, and does not need to do

[2]The extension is platform-specific, and will '.dll' on Windows.

anything else for the registration mechanism. She compiles the library using the usual command and loads it using `dyn.load()` or `library.dynam()`. In my example, I have a single file named 'myRoutines.c' and, in Unix, create the DLL with the command

```
R CMD SHLIB myRoutines.c
```

The internal R code will determine whether the registration mechanism is being used and take the appropriate action.

Now we can test our example and see what the registration mechanism gives us. First, we start R and load the DLL. Then we 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),
   expecting 0 for foo
> .Call("foo")  # Should be .C("foo")
Error in .Call("foo") :
   .Call function name not in load table
```

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

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

The very observant reader may have noticed that the three routines have been declared to be **static**. Ordinarily this would mean that they are not visible to R. Since we explicitly register the routines with R by their addresses (and not during compilation), this works as intended. The routines are only accessed directly from within the file. And now we have reduced the potential for conflicts between symbols in different libraries and of finding the wrong symbol.

Our example dealt with routines to be called via the `.C()` and `.Call()` functions. FORTRAN routines and those called via the `.External()` function are handled in exactly the same way, defining arrays for those routines. In our example, we specified `NULL` for the 3rd and 4th arguments in the call to *R_registerRoutines()* to indicate that we had no routines in either of these categories.

Rarely are libraries completely cast in stone. We occasionally add routines and want to be able to call them from R. To do this, one should register them and this merely involves adding them to the appropriate array which is passed in the *R_registerRoutines()* call. When one is developing the library, it can be inconvenient to have to remember to register routines each time we add them. Instead, it would be useful to be able to use the registration mechanism *and*, if the routine was not found there, to default to the dynamic lookup mechanism. This is easy to do from within the initialization routine for the DLL. In that routine, add the call

```
  R_usedDynamicSymbols(info, TRUE);
```

where `info` is the `DllInfo` object passed as argument to the initialization routine.

One can find additional examples of how to use the registration mechanism in the packages shipped with R itself (**ctest**, **mva**, ...). Also more technical overview of the mechanism with some annotated examples and more motivation is available at http://developer.r-project.org/DynamicCSymbols.pdf.

Extended applications

The motivation for developing the registration mechanism was to avoid the problems discussed at the beginning of this article. However, now that we have this mechanism in place, it turns out that 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 actively 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.[3] For instance, in our example, we could give the types of the two parameters of *barC()*. We haven't yet finalized the details of this interface and so it is not part of R quite yet. However, it might look something like the following:

```
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 correspond to these types. R can raise an error if it discovers an argument of the wrong type, or alternatively can convert it to the type the routine is expecting. This is a powerful facility that not only reduces errors, but also proves to be very useful for handling large, external datasets. Indeed, R 1.3.0 has

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

a feature that allows users to specify conversion routines for certain types of objects that are handled via the `.Call()` (see http://cm.bell-labs.com/stat/duncan/SCConverters).

A potentially important use of the registration mechanism relates to security, and specifically prohibiting some users calling certain native routines that have access to sensitive data. We have been developing packages that embed R within spreadsheets such as Gnumeric and Excel; Web browsers such as Netscape; relational databases such as Postgres; and so on. One benefit of this approach is that one can run R code that is dynamically downloaded from the Web. However, as we all know, this is a common way to download viruses and generally make ones machine vulnerable. Using the registration mechanism, developers can mark their routines as being vulnerable and to be used only in "secure" sessions. What this means exactly remains to be defined!

Building the table automatically

This registration mechanism offers all 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 convenient to be able to generate the registration code easily and without a lot of manual effort by the developer.

The **Slcc** (http://www.omegahat.org/Slcc/) package from the Omegahat project provides a general mechanism for processing C source code and returning information about the data structures, variables and routines it contains. This information is given as S objects and can be used to generate C code. The package provides a function to read both the S and C code of a library and generate the C code to register (only) the routines that are referenced in the S code.

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 platforms.

Summary

The new registration mechanism is being used in the R packages within the core R distribution itself and seems to be working well. We hope some of the benefits are obvious. We expect that others will appear over time when we no longer have to deal with subtle differences in the behavior of various operating systems and how to handle dynamically loaded code. The only extra work that developers have to do is to explicitly create the table of routines that are to be registered with R. The availability of the **Slcc** package will hopefully help to automate the creation of the registration code and make it a trivial step. We are very interested in peoples' opinions and suggestions.

Duncan Temple Lang
Bell Labs, Murray Hill, New Jersey, U.S.A
duncan@research.bell-labs.com

Support Vector Machines

The Interface to `libsvm` in package e1071

by David Meyer

"Hype or Hallelujah?" is the provocative title used by Bennett & Campbell (2000) in an overview of Support Vector Machines (SVM). SVMs are currently a hot topic in the machine learning community, creating a similar enthusiasm at the moment as Artificial Neural Networks used to do before. Far from being a panacea, SVMs yet represent a powerful technique for general (nonlinear) classification, regression and outlier detection with an intuitive model representation.

Package **e1071** offers an interface to the award-winning[1] C++ SVM implementation by Chih-Chung Chang and Chih-Jen Lin, `libsvm` (current version: 2.31), featuring:

- C- and ν-classification
- one-class-classification (novelty detection)
- ϵ- and ν-regression

and includes:

- linear, polynomial, radial basis function, and sigmoidal kernels
- formula interface
- k-fold cross validation

For further implementation details on `libsvm`, see Chang & Lin (2001).

[1]The library won the IJCNN 2001 Challenge by solving two of three problems: the Generalization Ability Challenge (GAC) and the Text Decoding Challenge (TDC). For more information, see: http://www.csie.ntu.edu.tw/~cjlin/papers/ijcnn.ps.gz.