An Alternative Mechanism for Resolving C Symbols

Duncan Temple Lang

July 19, 2001

Abstract

This is a note on the way the C level symbols are accessed in the S language (R and S-Plus). Luke and I propose that we introduce a more controlled and informative way to explicitly *register* C routines that can be called from S via the *.Call()*, *.C()* and *.Fortran()* functions. For a package that uses this registration mechanism, rather than dynamically resolving arbitrary C routines by name, we limit the available routines to those explicitly exported by a module. The export information include the external name and the address of the C routine itself. The *.Call()*, etc. functions are called with the alias name used to register the C routine, much like the way the internal/primitive functions are mapped to C routines in **names.c**.

This registration approach is used by Python. The benefits include

- restricted access to non-exported symbols, reducing the potential for fatal errors and increasing the potential for secure invocation;
- avoids the underscore issue;
- link-time resolution rather than load-time;
- allows aliasing/renaming of routines;
- allows static routines to be exported;
- reduces the differences between the different OSes.

Generally, there is more structure and the potential for exploiting this is greater. Also, we are relying less on linking and loading, and more on the more predictable C language!

In the near future, we can extend this information to optionally include details about the number and types of arguments. This great flexibility for handling external/foreign data types such as objects from other languages (e.g. Perl, Python, Java, etc.), and result sets from database queries, hdf5 objects, etc.

At present the author of the package would need to manually create the tables (arrays) of routines that are to be exported. While this is reasonably trivial, we can also simplify the use of this mechanism for library developers by creating a tool that can automatically generate the tables of available and called routine symbols. This tool can potentially even generate the S-level wrapper functions to these routines.

1 The Basic Idea

The basic setup is that the author of a package will explicitly register C routines that can be called via the *.Call()*, *.C()* and *.Fortran()* functions. This is done by putting entries for each of the symbols that the developer makes accessible to the S user into the appropriate table. We use three tables: one for the functions accessed via the *.C()* function, another for those accessed via the *.Call()*, and finally one for the *.Fortran()* function. These routines are registered with the S engine by calling the C routine `R_registerRoutines()`. This should be done within an initialization routine for the library. When the library is loaded, we look for such a routine by the name R˙init˙*library name* and invoke it if it is available. In this way, the library need only explicitly export one symbol for access via `dlsym()` or its equivalents on other platforms.

The `R_registerRoutines()` routine processes the symbol entries in each of the three tables and stores them away in internal structures. The internal code for *.Call(), .C()* and *.Fortran()* call `R_FindSymbol()` to resolve the native symbol. It is here that we see if the library/libraries in which we search for the symbol have this explicitly exported information. If so, we only look in those tables. We search for an entry in the tables whose public name matches the specified symbol being sought. If found, we return the associated routine pointer. If no such entry is found in this library, but there it has registered routines, we do not look further in this library. However, if there are no registered routines for this library, we use the current dynamic lookup mechanism using `dlsym()`.

This approach is backward compatible, using the existing dynamic lookup mechanism, and the new mechanism is merely an extension. The overhead should be minimal if it is not used. The current implementation of the lookup uses a linear search. As it is accepted and used more, we can move to a hash-table.

2 Example

The following is an example of how one uses this registration facility. One needs to include **R˙ext/Rdynload.h** to obtain the definition of the `R_CMethodDef` and `R_CallMethodDef` structures. Then, we specify the tables of {name, routine, number of argument} tuples. (Currently the number of arguments for the routine is not used. It will be used in another extension Robert, John and I are discussing.)

Finally, we register the entries in the tables. We do this by calling `R_registerRoutines()`. It is easiest to do this by creating an initialization routine for the library that R will call automatically when it is loaded. In our case, the library is named **foo.so**. Therefore, the name of this initialization routine for which R will search is `R_init_foo()`. This takes a single argument of type `DllInfo *`. This is passed back to `R_registerRoutines()` to identify for which DLL the symbols are being registered. If one chooses not to use an initialization routine, one can get the `DllInfo` reference to pass to `R_registerRoutines()` via a call to `R_getDllInfo()` with the fully qualified path of the library file (i.e. including the full directory and name). The other arguments to the `R_registerRoutines()` routine are the three different tables of routine/function definitions for C, Call and Fortran. Each of these arrays should be terminated by NULL. To ignore a type, just pass the value **NULL**.

```
#include "Rinternals.h"
#include "R_ext/Rforeign.h"

static void R_localMean(double *vals, long *len);
static void foo(double *vals, long *len, double *val);
static SEXP myCall(SEXP arg);

static R_CMethodDef R_CDef[] = {
   {"foo", &foo, 3},
   {"myMean", &R_localMean, 2},
   {NULL}
};

static R_CallMethodDef R_CallDef[] = {
  {"myCall", myCall, 1},
  {NULL}
};
```

```
void
R_init_foo(DllInfo *info)
{
 R_registerRoutines(info,
                    R_CDef,
                    R_CallDef,
                    NULL, 0);
}

static void
R_localMean(double *vals, long *len)
{
 double mean = 0;
 foo(vals, len, &mean);
 vals[0] = mean;

 return;
}

 /* Naive way to compute the mean. */
static void
foo(double *vals, long *len, double *val)
{
 double mean = 0;
 int i;
  for(i = 0; i < *len;i++) {
    mean += vals[i];
  }
  *val = mean/(*len);

  return;
}

static SEXP
myCall(SEXP arg)
{
 SEXP ans;
 PROTECT(ans = allocVector(INTSXP, 1));
 INTEGER(ans)[0] = Rf_length(arg);
 UNPROTECT(1);

 return(ans);
}
```

The following illustrates how one can modify an existing package, `ctest`, to use this mechanism. We put the code in the file **init.c** and add **init.o** to the list of object files to compile and link in this package. The code simply creates a 1-1 mapping between the C routine name and the string by which R will access it. Additionally, we include the number of parameters for each routine so that R can check the routine is being

called with the correct number of arguments. Finally, we define the routine `R_init_ctest()` which registers the C routines in the table. There are no calls to *.Fortran()* or *.Call()* in this package.

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

R_CMethodDef cmethods[] = {
  {"chisqsim", (DL_FUNC) &chisqsim, 11},
  {"d2x2xk", (DL_FUNC) &d2x2xk, 5},
  {"fexact",   (DL_FUNC) &fexact, 10},
  {"kendall_tau", (DL_FUNC) &kendall_tau, 4},
  {"pansari",  (DL_FUNC)&pansari, 4},
  {"pkendall", (DL_FUNC)  &pkendall, 3},
  {"pkstwo", (DL_FUNC) &pkstwo, 3},
  {"prho", (DL_FUNC) &prho, 5},
  {"psmirnov2x", (DL_FUNC) &psmirnov2x, 3},
  {"qansari",  (DL_FUNC) &qansari, 4},
  {"swilk", (DL_FUNC) &swilk, 9},
  {NULL, NULL, 0}
};

void R_init_ctest(DllInfo *dll)
{
  R_registerRoutines(dll, cmethods, NULL, NULL);
}
```

3 Comments

Python uses **NULL** to identify the end of the array so that the user doesn't have to specify the number of elements being registered.

We choose to have three separate tables for the routines that are to be accessed via the *.C()*, *.Call()* and *.Fortran()* functions. At present, the information is exactly the same for all of them. However, in the near future we will support additional fields which specify information about the argument types (convert routines, type constraints, etc.).

4 Necessary Code Changes

I have made the changes for UNIX. Nothing is conceptually different for Windows or the Macintosh versions. If anything, this mechanism reduces the differences as the symbol resolution does not use system level tools. Rather than adding support for each of the three **dynload.c** files, I believe it is best to consolidate these three different versions that currently contain large blocks of duplicated code into a single file that is in **main/** of the R base. Then I will abstract the platform specific differences using function pointers (or different OS-specific versions of the same symbols that are bound at link time). This consolidation of code is necessary if we are to go further with the extensions to the *.C()* and *.Call()* functions that Robert, John, Luke and I have been discussing and of which I will provide a description shortly.

The additions to support this facility in a backward compatible manner involve the following.

`AddDLL()` In `AddDLL()`, we look for a routine by the name `R_init_<library name>()`. If we find it, we invoke it. This is expected to register the exported routines via a call to `R_registerRoutines()`.

`DllInfo` We make an explicit structure to hold the `DllInfo` elements. This structure is extended to have fields for the arrays and lengths of the arrays of the different types of symbols.

`R_dlsym()` Rather than passing the handle returned from `dlopen()`, we pass the containing `DllInfo()` reference to the library in which the symbol is being searched. This routine now checks whether there are any symbols in the the S-specific export tables (created by the call to `R_registerRoutines()`), and looks through those. If this search fails to find the symbol in the tables, we signal an error. If there are no explicitly exported entries, we use the regular `dlsym()` mechanism to find the symbol in the shared library.

Rforeign.h We add a file named **Rforeign.h** in the **include/R˙ext/** directory. This contains declarations for the different types the user needs to create the different tables and call `R_registerRoutines()`.

We can make this faster by populating a hash-table when we register the routines for a different type (*.Fortran(), .C(), .Call()*).

5 Questions

Here are some questions that I would like some response

Shall I commit this for the Unix version? Should I wait to have a Windows and Mac version also? I suggest no and that I consolidate the code for the three versions into a single common/shared version that is parameterized with function pointers.

Under what circumstances is `CACHE_DLL_SYM()` activated?

The dynload code for the Mac looks very similar to that for Unix but does not share structure definitions, etc. Similarly, the Windows dynload is not quite as similar but does share the same structure. Should we move to function pointers for implementing the differences and centralize the common facilities? This will become more important if we decide to do more with these symbols such as checking arguments, calling user-level converters, etc.

Should *is.loaded(), symbol.C()* and *symbol.For()* take a *PACKAGE* argument. Similarly, *.Call()* is not documented as having one.

Should *symbol.C()* and *symbol.For()* also return the package they were found in, if they were found.

6 Tests

We compile the two shared libraries *foo* and *bar* from the tar file dynload test.

```
> dyn.load("bar.so")
> .Call("foo")
[1] TRUE
> .Call("foo", PACKAGE="bar")
[1] TRUE
> dyn.load("foo.so")
```

```
> .Call("foo", PACKAGE="bar")
[1] TRUE
> .C("foo", as.numeric(1:10), as.integer(10), x=numeric(1))$x
[1] 5.5
> .C("foo", as.numeric(1:10), as.integer(10), x=numeric(1), PACKAGE="foo")$x
[1] 5.5
```