An Introduction to The Interface Between C and R

Arpit Chaudhary*

23rd July 2007

Prof. in Charge: Prof. Anthony Davison[1]
Supervisor: Vahid Partovi Nia[2]

*Indian Institute of Technology-Delhi, Department of Mathematics, New Delhi, India
arpit0642004@gmail.com

[1]IMA Chair of Statistics, Ecole Polytechnique Federal De Lausanne, Switzerland, anthony.davison@epfl.ch

[2]Project Supervisor, Department of Statistic, EPFL, Switzerland. vahid.partovinia@epfl.ch.

Contents

1 Introduction

This manual will be useful for people looking to extend R with the C compiled code. The motivation for using the C interface with R may be bidirectional. For statisticians programming mainly in R, an interface to C provides fast execution of code and better memory management. The C interface also provides the user to access the existing C libraries. Similarly, people programming in C with the help of the R interface to get an access to huge set of mathematical functions implemented in R. It saves the labour of implementing such functions from scratch in C which may be quite cumbersome.

In this note, the document illustrates one way to link the C compiled code with R using R function called `.C`. The use of the `.C` interface is quite simple and has been explained with the help of examples. The code for examples have been developed in C and various examples shows its execution from R. It is assumed that you have some prior knowledge of C programming and acquaintance with the basics of R. The code discussed in this document should work with any unix installation of R.

There is also a PDF manual called *Writing R Extensions* available from the CRAN website. The manual describes other interfaces like .Call and .External which are more complicated to use but certainly more powerful. The following document explains briefly the `.C` interface.

2 Tools

The tools required for extending R with C compiled code are `cc` or `gcc` GNU compiler to compile the C code to create object files. Unix has a preinstallation of a C compiler. You will require a C compiler for compiling C code for windows if you are using windows. If you are using unix as an operating system then you may use *SciTE* editor for writing and compiling your C code. You can also install *Kate* software which is available in unix and it supports R programming. So, you may write your R code for running the C file in Kate. The use of *SciTE* editor and *Kate* provides a handy tool and prevents typing the same commands to execute the C code at the command prompt time and again. The cpp properties of *SciTE* editor may need to be changed by appropriate commands for building and compiling C functions. Also, R Statistical software is required to finally link and execute the compiled C code. The current version of R is 2.5.1.

3 Writing the C code :

The first step to link a compiled C code to R obviously requires a R compatible C file. That is, the C file used for linking to R has to have certain properties and the C functions to be called from R by `.C` should be written in a specific way. A C code which can

be easily compiled and executed using the `cc` or `gcc` compiler may not execute or may produce erroneous results when linked to R. In other words a proper execution of C code does not ensure proper execution from R if proper care is not taken. The following are few important points which should be followed for proper execution of C code using R.

1. You may include any C library that you need for execution of your function in the beginning of your code in the same way as it is done in C by using the include command. Example :

   `#include<stdio.h> or #include<string.h>`

2. There is no function called *main* as it is there in usual C codes while calling the C functions from R.

3. The C function must all be of type `void` that is the compiled code should not return anything except through its argument. We make calls to the C functions from R. The function should store the result of its execution in some parameter passed into the function as a pointer.

4. The code can have as many functions like usual C codes. Though there is no main function as stated above but the function being called by R acts as a main function. We may call it as a pseudo main function for our reference. All memory allocation and initialization for the global variables in C should be done in this function. This is an important point to consider. While writing the C program we may use global variables. If we initialize the global variables outside the main function that is in the beginning of the file it may lead to error in our code. Since R make functional calls to C so each of the C functions must be encapsulated and all initialization and memory allocation for the global and local variables must be done inside their respective functions.

5. Only the pseudo main function is required to be of the type `void` and must return results through its arguments. Other functions being used in this pseudo main function may return values as in usual C code.

6. You may develop a complex C code using the above rules. But for executing this code you need another function of type `void` which makes a functional call to your pseudo main function. All arguments passed to this C function are passed by reference, which means that we pass a pointer to an integer or an array. The call to the pseudo main function is done by simply listing the names of the variables if its prototype is an array or a pointer else by referencing it as a pointer if its a integer or double. We will clarify it in some examples later.

4 Compiling the code:

When compiling your C code, you use R to do the compilation rather than call the C compiler directly. This makes life much easier since R already knows where all of the necessary header files and libraries are located. If you have written C code in a file called ABC.c, then you can compile that code on the command line of your Terminal window with the command

```
R CMD SHLIB ABC.c
```

This command produces a file called ABC.so, which can be dynamically loaded into R(which we will explain later). If you do not want to name your library ABC.so, then you can do the following:

```
R CMD SHLIB -o newname.so ABC.c
```

The file extension .so is not necessary but it is something of a Unix tradition. Actually it is an object file.

Once you have compiled your C code you need to launch R and load the library. In R, loading external C code is done with the `dyn.load` function. If you have compiled C code from a file ABC.c into a file ABC.so then you can load the code using

```
> dyn.load("ABC.so")
```

Now all of the functions that you wrote in the file ABC.c are available to be called from R.

5 First Example : A Very Simple C Function

Consider a very simple example of adding two numbers. The C code for this in a file named `example1.c` is as follows :

```
void sum( int a, int b, int *result)
{
*result = a+b;
}

void Rsum( int *a, int *b, int *result)
{
sum(*a, *b, result);
}
```

Though the example may look simple but please take time to notice the the following key points in the above example :

1. The file has no main function.
2. In this file the pseudo main function as described earlier is the function `sum` which is of the type `void` and it returns the value of the execution of the code through the argument `*result` which is passed as a parameter in the function.
3. The purpose of the function `Rsum` will be clear later but note that all the arguments are passed as a reference that is through pointers in the function. This function is necessary to call the desired C function(pseudo main function) from the `.C` interface of R(as discussed later). Since `.C` interface of R has restrictions of use that is in the C function which `.C` calls, every argument should be a pointer. For this reason, we need to define the other function called `Rsum` in addition to `sum` function.

To execute the code from R first we need to compile this code which can be done by using the function as stated in previous section.

```
arpit@arpit-desktop:~/Desktop$ R CMD SHLIB example1.c
gcc -std=gnu99 -I/usr/share/R/include -I/usr/share/R/include -fpic  -g -O2
-c example1.c -o example1.o
gcc -std=gnu99 -shared  -o example1.so example1.o -L/usr/lib/R/lib -lR
arpit@arpit-desktop:~/Desktop$
```

The above command at the command prompt terminal generates a file called as `example1.so`. After compiling the file, begin the R software by simply typing R at the command prompt. Now you need to load the file with the help of `dyn.load` command as illustrated below.

```
> dyn.load("example1.so")
```

After loading the file you may pass the values to your arguments in the functions and call the function using the `.C` interface as illustrated below.

```
> dyn.load("example1.so")
> a <- 5
> b <- 6
> out <- .C("Rsum", a = as.integer(a), b = as.integer(b), result = as.integer(0))
```

Note that we can not simply pass the argument as $a = 5$ and $b = 6$ instead of `a = as.integer{a}` and `b = as.integer{b}` respectively. It is because 5,6 may be passed as an integer or double, etc which is not clear and we may get wrong answers. So, it is

necessary to properly coerce the prototype of the variables in C and the objects being passed to C from R. Now if you want to view the list of objects as an output simply type `out` which here contains the list of objects returned by R.

```
> out
$a
[1] 5

$b
[1] 6

$result
[1] 11
```

To view any particular argument as for example result you may use the following command.

```
> out$result
[1] 11
```

For understanding the procedure just illustrated above, please go through the following points carefully.

1. You have to allocate memory to the vectors passed to `.C` in R by creating vectors of the right length. R treats an integer as an array of integers with a unit length. So assigning the value of 5 to the variable a in R means creating an array with single element whose value is 5.

   ```
   > a
   [1] 5
   ```

   Similarly, result is declared as integer with an array size of 1 and initial value of 0. The final value after the execution of the code is passed into the variable called `result`.

2. The first argument to `.C` is a character string of the C function name which calls the pseudo main function. Both the function are of type `void`.

3. The rest of the arguments are R objects to be passed to the C function. Here, we create the R objects `a` and `b` to be passed in to the C function with correct type. Here as integers.

4. All arguments should be coerced to the correct R storage mode to prevent mismatching of types that can lead to errors. The type to which you coerce the variables passed to the C function and the types in the prototype of your C function should match(as well as the order of the variables passed). Notice that we coerce `a` to integer type using `as.integer` and in the C function we have set `a` to be of type `integer`.

5. `.C` returns a list objects. In the above example `.C` returns the value of `a,b` and `result` in a list.

6. The second `.C` argument is given the name `a`. This name is used as a dummy for the respective component in the returned list object(but not passed to the compiled code). Any letter instead of a would have worked fine. But the values passed `as.integer` must be passed in the same sequence as that is required in your pseudo main function.

7. With `.C`, the R objects are copied before being passed to the C code, and copied again to an R list object when the compiled code returns.

8. To pass the value of any outcome of your function in the next function you can obtain a particular value of any variable by using the `out$` followed by the name of variable. Example : `out$result` returns the value of the variable `result`.

9. There is nothing so specific about `int`. You may use `double` in place of `int` at all places above as per your requirement of the code.

10. Needless to say but the value of the variable `result` is `11` as `a =5` and `b=6`. You can pass on different values to get the result after loading the file once.

11. With this example its not possible to add decimal numbers. Use `double` in place of `integer` if you wish to add two decimal numbers.

This example illustrates the basic things required for writing the C code, passing and getting the values from the C code using the `.C` interface. Next, we consider an example to pass an array of integers as an input and try to get an array of integers as output.

6 2nd Example : To pass an array of integers

Here, we extend our previous simple example of adding two elements. Now, we wish to add all the elements in the given two array of equal size at their respective indexes and output the final array. The following C programme will provide an insight into passing the array from R and getting the value back in R. Arrays are the most common data structure supported by C, so understanding how to pass an array is quite important. The C code for the above stated problem is as follows

```
void sum_array( double A[], double B[], int L, double *result)
{
int i;

for ( i = 0; i < L; i++)
{
result[i] = A[i] + B[i] ;
}
}

void Rsum_array( double *A, double *B, int *L, double *result)
{
sum_array( A, B, *L, result);
}
```

Here, `A` and `B` are an array of type `double`. `L` denotes the length of the array of `A` and `B` which are equal. The argument `result` is an array of doubles which contains the results of the function `sum_array`. The code adds the individual member of the two array located at the same index.

The procedure for executing the code is as follow :

```
arpit@arpit-desktop:~/Desktop$ R CMD SHLIB example2.c
gcc -std=gnu99 -I/usr/share/R/include -I/usr/share/R/include -fpic  -g -O2
-c example2.c -o example2.o
gcc -std=gnu99 -shared  -o example2.so example2.o   -L/usr/lib/R/lib -lR
arpit@arpit-desktop:~/Desktop$

arpit@arpit-desktop:~/Desktop$ R
```

The above code compiles the C file `example.c` and then starts the R software. Now, we need to load the file and pass the values to the different C arguments. This is as shown below.

```
> dyn.load("example2.so")
> A <- c(1.5,2.5,3.5,4.5,5.5,6.5,7.5,8.5,9.5,10.5)
> A
 [1]  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5
> B <- c(1.5,2.8,3.5,4.5,5.5,6.5,7.5,8.8,9.5,10)
> B
 [1]  1.5  2.8  3.5  4.5  5.5  6.5  7.5  8.8  9.5 10.0
> L <- length(A)
```

```
> L
[1] 10
> L <- length(B)
> L
[1] 10
> out <- .C("Rsum_array", a = as.double(A), b = as.double(B), l = as.integer(L), res
> out
$a
 [1]  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5

$b
 [1]  1.5  2.8  3.5  4.5  5.5  6.5  7.5  8.8  9.5 10.0

$l
[1] 10

$result
 [1]  3.0  5.3  7.0  9.0 11.0 13.0 15.0 17.3 19.0 20.5

> out$result
 [1]  3.0  5.3  7.0  9.0 11.0 13.0 15.0 17.3 19.0 20.5

> out$a
 [1]  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5

> out$b
 [1]  1.5  2.8   3.5  4.5  5.5  6.5  7.5  8.8  9.5 10.0
```

Note the following points in the above execution of another simple C function from R.

1. The way an array is declared in R. It must be quite trivial for you if you have some experience working in R.
2. We need to allocate sufficient memory to the objects in R so that results from C after execution could be passed on to R. Here, results is initialized as an array of length `L`(same as length of array `A` and `B`). Initializing it with more than required space is not a problem but insufficient memory allocation may be source of error. Especially it happens in programs where the length of the output array may vary, that is the length may be dependent on the data you are operating on, i.e. it is decided dynamically. Sometime, you may face the problem of stack overflow indicating lack of physical memory allocated to your program. You may set the limit manually by using the `ulimit -a` command. This problem is also referred again in last section of the report in some common errors and methods to debug them.

3. The variables passed to the C function and the variable in the C function must be in the same order and of the same prototype. Here `A` and `B` are of the type `double` and `L` is an `integer` type which is being coerced as `as.double` and `as.integer` respectively. Any mismatch in the variable prototype may lead to an error or incorrect result.

4. The rest of the program is quite similar to the first example. Try to see the similarities in the first and the second example like no main function, return type set to `void`, etc.

5. Decimal numbers have been taken as an argument purposefully to illustrate the use of `double` array as an input instead of `integer` values.

The above two example expilicitly explains the procedure of passing an integer, an array of integers or doubles from R to C and how to get an array of integers or double as an output of the C functions. A point to consider is that if you input an array object of type `double` in R and use the data with C function which works with integers then even if all arguments were integers, there is no function like `as.integers()` of R in C to change property of an array. The only solution is to replace the values of double array in an integer array which is time consuming and decreases the efficiency of the code. So, we should input arguments as we need in C; try not to change the properties of arrays which may damage speed of execution of the code.

The two dimensional arrays or more popularly known as matrices in R can also be passed from R to C. The environment C does not support matrices datatypes but stores them in large single dimensional array. Proper handling of indices in R(which starts from 1 to n) and C(starts from 0 to n-1) may enable you to use matrices as a datatype in your programs.

By now it must be clear how to extend R with your C code with ease. Next, we look at some more kind of vector types such as `logical` and then we conclude this manual with some common errors and method of debugging in C and R.

7 Calling C with different vector types using .C

In this example, we present the way of calling different vector types using the `.C` interface to R. By now, we have seen how to pass an integer or an array of integers and a double value. This example illustrates how to pass a logical argument and characters to C. The C code is as shown :

```
void example3(int *i, double *d, char **c, int *l) {
    i[0] = 1;
    d[0] = 2.333;
    c[1] = "q";
```

```
    l[0] = 0;
}
```

To execute the code from R, compile the code as shown above and then try the following code.

```
> dyn.load("example3.so")
> i <- 1:10                       # integer vector
> d <- seq(length=3,from=1,to=2) # real number vector
> c <- c("a", "b", "c")           # string vector
> l <- c("TRUE", "FALSE")         # logical vector
> i
 [1]  1  2  3  4  5  6  7  8  9 10
> d
 [1] 1.0 1.5 2.0
> c
 [1] "a" "b" "c"
> l
 [1] "TRUE"  "FALSE"
> out <- .C("useC", i1 = as.integer(a), d1 = as.numeric(d), c1 = as.character(c), l1
> out
$i1
  [1] 1  2  3  4  5  6  7  8  9 10
$d1
 [1] 2.333 1.500 2.000
$c1
 [1] "a" "q" "c
$l1
 [1] FALSE FALSE
```

The above code clearly shows how to pass the logical arguments and characters to the C functions. Note, that the characters are passed as double reference that is by using pointer to pointers. The values of the variables before executing the program and after execution has been shown. The program is pretty simple and just illustrates the way of passing arguments and has been left for readers to take a close look and understand the mechanism behind it.

8 Some common errors and methods of debugging the code

In this section we discuss some common errors that you may encounter as a beginner to the R-C interface and the method to debug them easily.

1. While executing the code in R you may have errors like *function ABC does not exist in the load table*. You must have forgotten to load the function in R while running the program time and again. Though it is not necessary to load the same function again and again. It may happens that you are able to run the program once correctly but it produces garbage the next time. This may be due to improper coding in C. You must have initialized some variable outside the function which is not being initialised to its initial value when the function is called the next time leading to erroneous results.

2. You may face the problem of *stack overflow* when you run the C code on a huge data in R. Since running code in statistics may some time require huge data set which leads to lack of storage in the main memory and stack is full leading to stack overflow. The only way to solve this problem is to manually allocate more physical memory to your current C program. This can be done by using unix command `ulimit -a` or `Cstack_info()` in R. This command displays the default allocation of stack. You can modify it as per your expected or make it unlimited by using the unix command `ulimit -s unlimited`. The problem of stack overflow is also prominent if you are using recursion in your program. Since recursion first calculates all the values of its variables at each stage it may require huge space to store the values of each variable. Sometime removing recursion from the code may solve the problem.

3. Though trivial, but you may have errors like *object A not found* which may be due to misspelled name of the objects in R. R is also a case sensitive language.

4. If you encounter *segfault* then you are in trouble. This may be due to improper memory allocation, improper handling of pointers, etc. The best way to avoid this is to take proper care while programming in C. Else you may use the `printf` command in C at various stages of your code to see what went wrong and where. You will have to include the libraries in C like

   ```
   #include<stdio.h>, #include<string.h>, #include<stdlib.h>, etc.
   ```

   to use the `printf` and other commands of C.

5. The error message *memory not mapped* or *memory corrupted* is again a very common error. This may be due to improper management of memory, trying to write at the same place again and again, or improper array allocation, sloppy handling of pointers, etc. The way to debug is the same as in *segfault*. So you should keep a good track of allocation of memory while programming. Sometimes, *memory not mapped* may happen if the stack size is not enough. It can be handled by `ulimit` as described earlier.

6. The loops in script files usually runs slowly. It usually happens in complex functions or Monte Carlo simulations. In these cases, programming in C and execution from R benefits both high speed execution of C and high level commands of R.