Interfacing R and C

(in Windows)
a basic introduction
from a basic user

Brock Stewart, PhD
jnn6@cdc.gov

CDC R User Group
August 27, 2015

Outline

Introduction

Why C in R? Interpreted languages (such as R) are slower than compiled code. Almost always unnecessary to create user-defined source code: many R functions already use compiled code; extra time involved is only justified in extreme situations. However, when we feel it is worth the effort, we can load compiled programs into R. Furthermore, learning C gives you a greater understanding of R, since much of R is written in C.

Why pointers? Memory management is fundamental to C programming; is a powerful feature of C. Even if using packages, wrappers, or other simplifying R tools, is useful to understand this aspect of C. R interfaces with C programs via pointers. Also, you'll likely need dynamic memory allocation, for which pointers are necessary.

C: A minimal intro just big enough for use with R

About C

- Dennis Ritchie, AT&T Bell Labs, released 1972
- *The C Programming Language*: Kernighan & Ritchie, 1978
- Still one of the most – if not the most – widely used programming language (esp. including it's variants)
- Was used to write OS's UNIX, Windows, Linux
- Was used to write R, MySQL, ...
- Can be executed by nearly any microprocessor you can find; portable to (almost?) every digital computing architecture that has ever existed: computers, phones, calculators, appliances, robots, machinery ... jet airplanes
- Allows programs to directly access and manipulate memory

C Basics

C

```c
/* my C program */
#include <stdio.h>

int main()
{
  printf("Hi RUG \n");
  return 0;
}
```

`#` Preprocessor directives

- header files
- constants
- macros

C program $\approx$ collection of functions

- Preprocessor directives
- Functions
- Variables

Statements end in semicolons
C is case-sensitive
Base C is considered "small" – header files are similar to R libraries
Most header files you need will be included with your compiler
Can even include R header files to use R functions, e.g. `rnorm()`

C – Preprocessor Directives

C

```c
#include <stdio.h>
#include myHeader.h
#define PI 3.141592654f
```

The statement `#include <stdio.h>` merges the contents of "stdio.h" into the program before compilation.

You can include your own header files by omitting the angle brackets.

Other things can be done with preprocessor directives; don't worry too much about them when first learning about C.

C – Variables

- Variable = a name given to an accessible storage area
- Define: `type variable_list;`
- Data types: `char`, `int`, `float`, `double`, `void`
- `variable_list` is one or more names separated by commas
- Initialize (Assign an initial value):

C

```c
int ack, foo=4, bar;
ack=3;
```

- Uninitialized variables have values either NULL or undefined, depending on type.

C – Variables & Memory

- `int a;`

  Assigns the name “a” to a block of memory with exactly the size to store an int

- `a = 3;`

  Assigns the value “3” to the storage block named “a”

- `int a=3;` does both

- Each memory block has:
  - a size, which depends on the data type and system
  - an address, which is assigned by the system

C – Functions

C

```c
type name( param list )
{
        body of the function
}
```

- `type` If the function returns a value, then the data type of the value. If the function does not return a value, then `void`
- `name` The function's name
- `param list` optional; comma-separated

Self-contained C programs must contain a function named "main"

C programs written for R should not contain a "main" function

C – Pointers

- Pointer = a variable whose value is the address of another variable
- Define: `type *name;` Same as other variables, with "*"
- The address of any variable can be obtained using "&"
- We also use "*" to access the value of the variable that a pointer is pointing to

C

```c
int ack = 20;
int *ptr;
ptr = &ack;
printf("%x",&ack); /* 12fee2 */
printf("%x", ptr); /* 12fee2 */
printf("%d",*ptr); /* 20 */
```

C – Arrays

- A data structure storing a fixed-size collection of elements (variables) of the same type
- Consists of contiguous memory locations
- Declare: `type name[size];` where size is an int>0
- Initialize:

C

```c
int arr[6] = { 2, 20, 40, 12, 16, 17 };
/* or */
arr[0] = 2;
arr[1] = 20; /* etc. */
printf("%i",arr[3]); /* 12 */
```

- Indexing begins at 0
- Access elements with square brackets

C – Arrays & Pointers

- The name of an array is a pointer to the first element in the array.
- Pointers can be indexed like arrays.

C – Dynamic Memory Allocation

Memory Management Functions (from cplusplus.com)

- These functions are found in the header file `<stdlib.h>`.

`malloc(size)` Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block. The content is not initialized; indeterminate values.

`calloc(num,size)` Allocates a block of memory for an array of *num* elements, each of *size* bytes long, initialized to zero. I.e. allocates a zero-initialized memory block of (*num***size*) bytes.

`realloc(ptr,size)` Changes the size of the memory block pointed to by *ptr*. Existing content is preserved. Value of newly allocated portion is indeterminate.

`free(ptr)` Deallocates a block of memory previously allocated by a call to one of the above functions – the one pointed to by *ptr* – making it available again for further allocations.

C – Dynamic Memory Allocation

C

```c
int conv=10, sum=3, i=2, j;
int *arr = calloc(i,sizeof(int));
arr[0] = 1;
arr[1] = 2;
while( sum < conv )
{
  i++;
  arr = realloc(arr,i*sizeof(int));
  arr[i-1] = i;
  sum += arr[i-1];
}
```

Can't do `int arr[i];` since we want to use realloc

Can replace calloc line with `int *arr = malloc(i*sizeof(int));`

C in R: Basic layout using .C()

Step 0: Install Rtools

- Why? In Linux, all the tools needed to build packages – including compiling source code – are included in the R installation. In Windows, these must be added seperately.
- Get Rtools here:
  https://cran.r-project.org/bin/windows/Rtools/
- Will install to `C:\Rtools` by default.
- Add Rtools to the Windows system `PATH` variable
- Authoritative info:
  https://cran.r-project.org/doc/manuals/R-admin.html#The-Windows-toolset

Step 1: Write C code

- Should be composed of functions
- Do not write a “main” function
- All function arguments should be pointers
- Functions should not return any value

Step 2: Compile C code

Command Prompt

```
> cd filepath\to\your\code
> R CMD SHLIB My_C_code.c
```

Can use `pushd` instead of `cd` for network drives

Step 3: Use Compiled code in R

R

```R
dyn.load()
.C("Fun",...)
```

list of parameters should be “typed”

Step 3: Use Compiled code in R

R

```
1 dyn.load()
2 .C("Fun",...)
```

list of parameters should be "typed"
What happens if you don't type the parameters?
Nobody knows for sure, but don't provoke the gremlins

Tiny Example

C: foo01.c

```c
void foo(int *n, double *x)
{
        int i;
        for (i=0; i<*n; i++)
                x[i] = x[i]*x[i];
}
```

Command Prompt

```
> pushd \\cdc.gov\private\L132\JNN6\fooProj
> R CMD SHLIB foo01.c
```

R

```r
dyn.load("foo01.dll")
.C("foo", n=as.integer(5), x=as.double(rnorm(5)))
```

Tiny Example

R

```
.C("foo", n=as.integer(3), x=as.double(rnorm(3)))
$n
[1] 3

$x
[1] 1.1859348 0.0226387 1.6485473
```

Output is a list, a named list when inputs are given names

Tiny Example

R

```
1  foo = function(y){
2          .C("foo",
3                  n=as.integer(length(y)),
4                  x=as.double(y)
5          )
6  }
7  foo(rnorm(3))
8  $n
9  [1] 3
10
11 $x
12 [1] 1.692649 2.530549 5.003614
```

Can wrap in an R function ...

Tiny Example

R

```
1  foo = function(y){
2          .C("foo",
3                  n=as.integer(length(y)),
4                  x=as.double(y)
5          )$x
6  }
7  foo(rnorm(3))
8  [1] 0.1241827 0.0109811 0.3376740
```

... and return only part of the outputted list ...

Tiny Example

R

```r
foo = function(y){
        if(!is.numeric(y))
                stop("Hey! y must be numeric!")
        val = .C("foo",
                n=as.integer(length(y)),
                x=as.double(y)
        )
        return(val$x)
}
foo(rnorm(3))
[1] 0.01478424 0.17330274 0.03974468
foo("bar")
Error in foo("bar") : Hey! y must be numeric!
```

... and include error-checking

Running-time Example

C

```c
void convolve(double *a, int *na, double *b, int
    *nb, double *ab)
{
        int nab = *na + *nb - 1;

        for(int i = 0; i < nab; i++)
                ab[i] = 0.0;

        for(int i = 0; i < *na; i++)
          for(int j = 0; j < *nb; j++)
                ab[i + j] += a[i] * b[j];
}
```

From the Writing R Extensions manual on CRAN

Running-time Example

R

```
dyn.load("convolve.dll")
conv = function(a,b)
    .C( "convolve",
        as.double(a),
        as.integer(length(a)),
        as.double(b),
        as.integer(length(b)),
        ab = double(length(a) + length(b) - 1)
    )$ab
```

Running-time Example

R

```
convR = function(a,b){
   an=length(a)
   bn=length(b)
   nab=an+bn
   ab=rep(0,nab)
   for(i in 1:an)
   for(j in 1:bn)
      ab[i + j] = ab[i+j] + a[i] * b[j]
   ab[-1]
}
```

Running-time Example

R

```
library(rbenchmark)
A=rnorm(1000)
B=rnorm(5000)
benchmark(
        conv(A,B),
        convR(A,B),
        replications=10,order="relative"
)
       test replications elapsed relative
 conv(A, B)           10    0.05      1.0
convR(A, B)           10  112.09   2241.8
```

Further info on `R CMD SHLIB`

```
R CMD SHLIB --help
```

- Build a shared object for dynamic loading from the specified source or object files (which are automatically made from their sources) or linker options. If not given via '–output', the name for the shared object is determined from the first source or object file.
- -h: print short help message and exit
- -o=LIB: use LIB as (full) name for the built library
- -c: remove files created during compilation

Also see:

- `?SHLIB`
- Writing R Extensions Section 5.5

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++
- 1st arg is char string specifying the function in a loaded DLL

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++
- 1st arg is char string specifying the function in a loaded DLL
- Can test if a function is loaded using `is.loaded()`

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++
- 1st arg is char string specifying the function in a loaded DLL
- Can test if a function is loaded using `is.loaded()`
- Up to 65 further args giving R objects to be passed to compiled code

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++
- 1st arg is char string specifying the function in a loaded DLL
- Can test if a function is loaded using `is.loaded()`
- Up to 65 further args giving R objects to be passed to compiled code
- These are **copied** before being passed in, and again to an R list when the compiled code returns – that is *copied twice*

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++
- 1st arg is char string specifying the function in a loaded DLL
- Can test if a function is loaded using `is.loaded()`
- Up to 65 further args giving R objects to be passed to compiled code
- These are **copied** before being passed in, and again to an R list when the compiled code returns – that is *copied twice*
- Look-up speed consideration: see PACKAGE argument

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++
- 1st arg is char string specifying the function in a loaded DLL
- Can test if a function is loaded using `is.loaded()`
- Up to 65 further args giving R objects to be passed to compiled code
- These are **copied** before being passed in, and again to an R list when the compiled code returns – that is *copied twice*
- Look-up speed consideration: see PACKAGE argument
- The compiled code should not return anything except through its args; C funcs should be of type `void`

Further info on .C()

Writing R Extensions Section 5.2

- Provides interface to compiled code that's been linked into R either at build time or via `dyn.load()`
- Primarily intended for compiled C, but can be used with other languages which can generate C interfaces, e.g. C++
- 1st arg is char string specifying the function in a loaded DLL
- Can test if a function is loaded using `is.loaded()`
- Up to 65 further args giving R objects to be passed to compiled code
- These are **copied** before being passed in, and again to an R list when the compiled code returns – that is *copied twice*
- Look-up speed consideration: see PACKAGE argument
- The compiled code should not return anything except through its args; C funcs should be of type `void`
- Coerce all args to an R storage mode before calling .C

Further info on dyn.load

Writing R Extensions Section 5.3

- Compiled code to be used with R is loaded as a shared object, a DLL in Wiindows
- `dyn.unload()` to unload; not normally necessary, but is in Windows for, e.g., developing the C code
- See further info in this section regarding package building

C in R: Advanced interfaces (.Call, .External, Rcpp, inline)

.Call vs .C

From http://mazamascience.com/WorkingWithData/?p=1099

`.C()`

- allows for simple C code that doesn't know R
- only simple data types can be passed
- all argument type conversion and checking must be done in R
- all arguments are copied locally before being passed to the C function (memory bloat)

`.Call()`

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

.Call vs .External and Rcpp

- Primary difference between .Call and .External: args are specified in .Call as in .C, i.e. as comma-separated list. .External has only 2 args: function name, and one argument holding all args sent to the compiled code.
- .External allows a variable number of arguments to be supplied to its args argument
- Rcpp: *Very* extensive development in recent years. Uses the .Call interface, so requires more knowledge of R internals than the simple use of .C
- My favorite Rccp starting point: http://mazamascience.com/WorkingWithData/?p=1125
- Rcpp, especially some of its extentions (sugar), requires less code than .Call

Some CRAN manual info on .Call (and .External)

Writing R Extensions Section 5.9

- Traditionally, speed up using C code done with .C. Use .Call or .External to utilize internal R structures. The R-side is similar between .C and .Call; the C-side changes substantially.
- .Call – 1st arg is also name of loaded function
- .Call – also takes up to 65 args to be passed to the compiled code
- Ask yourself: Do I really need it? Consider standard R first. If only atomic vectors, then consider .C.
- .Call and .External allow much more control, but impose greater responsibility and knowledge
- **Important**: With .Call (and .External) we use C structures, data types, functions, and macros available in header files, esp. `Rinternals.h` or `Rdefines.h`

Example C code for .Call – convolve2.c

C: convolve2.c

```c
#include <R.h>
#include <Rinternals.h>
SEXP convolve2(SEXP a, SEXP b){
int na, nb, nab;
double *xa, *xb, *xab;
SEXP ab;
a = PROTECT(coerceVector(a, REALSXP));
b = PROTECT(coerceVector(b, REALSXP));
na = length(a); nb = length(b); nab = na + nb - 1;
ab = PROTECT(allocVector(REALSXP, nab));
xa = REAL(a); xb = REAL(b); xab = REAL(ab);
for(int i = 0; i < nab; i++) xab[i] = 0.0;
for(int i = 0; i < na; i++)
for(int j=0;j<nb;j++) xab[i+j] += xa[i]*xb[j];
UNPROTECT(3);
return ab;
}
```

Example C code for .Call – convolve2.c

Command Prompt

```
> pushd \\cdc.gov\private\L132\JNN6\convProj
> R CMD SHLIB convolve2.c
```

R

```r
dyn.load("convolve2.dll")
conv2 = function(a, b) .Call("convolve2", a, b)
```

New C-side code

- Notice the use of R.h and Rinternals.h
- SEXP: a pointer to a structure with typdef SEXPREC
- SEXP = *S*imple *EXP*ression (from LISP language)
- SEXPREC = *S*imple *EXP*ression *REC*ord
- C func and its args are always type SEXP
- SEXP is kind of a "catch all" data type used to handle R objects; converted to int, double, etc. in C code
- Requires handling garbage collection: PROTECT, UNPROTECT (must balance)
- Details on SEXP and all of its types (e.g. REALSXP) found in R Internals Section 1
- Other functions from Rinternals.h, e.g.: coerceVector, allocVector

Postscript

Summary

.C is easy. Normal C code. Used with atomic R objects. Can use R-analogs Calloc and Realloc for dynamic memory allocation (Writing R Extensions 6.1.2). Quite a bit of R built with .C, you can do quite a bit with it too, although .Call considered more appropriate for "modern" usage by enthusiasts.

.Call has a *much* higher learning curve, due to use of utilities in Rinternals.h and other header files. Unfortunately, these aren't necessarily well-documented – might have to RTFS. However, allows much more utility, e.g. easily handles all types of R objects, including multi-dimensional, functions, expressions, and object attributes (names, class, etc.).

Rcpp *et al.* (which I didn't really cover) improve on .Call, allowing you the same (or more) power with simpler code than .Call. I strongly suggest learning and going through examples in the order presented here: C, .C, .Call, then Rcpp.

Additional References & Resources

C

- http://www.tutorialspoint.com/cprogramming/index.htm
- https://en.wikibooks.org/wiki/A_Little_C_Primer
- https://en.wikibooks.org/wiki/C_Programming
- http://www.cprogramming.com/
- http://en.cppreference.com/w/
- http://www.cplusplus.com/
- http://forum.codecall.net/forum/161-cc-tutorials/

R

- http://adv-r.had.co.nz/C-interface.html
- http://users.stat.umn.edu/~geyer/rc/
- http://mcglinn.web.unc.edu/blog/linking-c-with-r-in-windows/
- http://wiki.math.yorku.ca/index.php/R:_C_code:_including_it_in_R
- https://nugaemeae.wordpress.com/2012/08/30/the-why-and-the-how-of-installing-rtools/

References

- *Writing R Extensions*, Chapters 5 & 6
  https://cran.r-project.org/doc/manuals/r-release/R-exts.html
- *R Internals*, Chapter 1
  https://cran.r-project.org/doc/manuals/R-ints.html
- `?SHLIB`
- `?.C`
- `?.Call`
- `?.External`