Table of Contents

Consider the getrusage example in the examples directory. Specifically, let's look at the gettimeofday routine.

R
```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```
When we look at this routine, we can "recognize" that the two arguments are out arguments. We might want to call this in several different ways from R.

- gettimeofday() no arguments and we just want two values back representing the contents of the timeval and timezone structures. In this case, the C code could use local variables and explicitly copy them back to R objects.

  R
  ```
  SEXP
  R_gettimeofday()
  {
    SEXP ans;

    struct timeval tv;
    struct timezone tz;

    gettimeofday(&tv, &tz);
    PROTECT(ans = NEW_LIST(2));
    SET_VECTOR_ELT(ans, 0, R_copyStruct_timeval(&tv));
    SET_VECTOR_ELT(ans, 1, R_copyStruct_timezone(&tz));
     /* Put names on the elements of the list ... */
    UNPROTECT(1);
    return(ans);
  }
  ```

- An alternative approach is that again the R users calls the function with no arguments, but the R function allocates C-level structures for these two arguments and then passes the references to the C routine that calls gettimeofday.

  R
  ```
  gettimeofday =
  function()
  {
     tv = alloc("struct timeval")
     tz = alloc("struct timezone")
     .Call(R_gettimeofday, tv, tz)
  }
  ```
  The C routine R_gettimeofday is then defined as

R

```
SEXP
R_gettimeofday(SEXP r_tv, SEXP r_tz)
{
  SEXP ans;

  struct timeval *tv;
  struct timezone *tz;

  tv = DEREF_PTR(r_tv, struct timeval);
  tz = DEREF_PTR(r_tz, struct timezone);

  gettimeofday(tv, tz);
  PROTECT(ans = NEW_LIST(2));
  SET_VECTOR_ELT(ans, 0, r_tv);
  SET_VECTOR_ELT(ans, 1, r_tz);
   /* Put names on the elements of the list ... */
  UNPROTECT(1);
  return(ans);
}
```

This just gets access to the allocated data structures and passes them to gettimeofday and then, with the structures filled in, passes the two back as a list. We could do the last step in R. Alternatively, we could explicitly copy the contents as before rather than returning references. But again, we can do this in R, either in the R function or leaving it to the user

R

```
as(gettimeofday()[[1]], "timeval")
```

So we have several combinations.

- Create the local variables in C and copy the results back to R objects.
- Create the local variables from R as references and pass them to C to be filled in and then either copy them back as R objects or return as references to the C data structures and fetch the invdividual elements as needed.

R

```
gettimeofday =
function(tv = NULL, tz = NULL, copy = TRUE)
{
  if(!copy) {
       # not copying results so need to have references we can hold onto.
     if(is.null(tv))
        tv = alloc(R_alloc_struct_timeval)
     if(is.null(tz))
        tz = alloc(R_alloc_struct_timezone)
  }

  if(!is.null(tv) && !is(tv, "timevalRef"))
   stop("need a NULL or reference to a timevalRef")
```

```
  if(!is.null(tz) && !is(tz, "timezoneRef"))
   stop("need a NULL or reference to a timezoneRef")

  .Call(R_gettimeofday, tv, tz, as.logical(copy))
}
```

And the C code would look like

```
                                                                        R
SEXP
R_gettimeofday(SEXP r_tz, SEXP r_tv, SEXP r_copy)
{
 bool copy = LOGICAL(r_copy)[0];

  struct timeval dummy_tv, *tv = &dummy_tv;
  struct timezone dummy_tz, *tz = &dummy_tz;

      /* Will actually be an object that has a slot containing the externalptr */
  if(TYPEOF(r_tv) == EXTPTRSXP)
     tv = R_ExternalPtrAddr(r_tv);

  if(TYPEOF(r_tz) == EXTPTRSXP)
     tz = R_ExternalPtrAddr(r_tz);

  gettimeofday(tv, tz);

  PROTECT(ans = NEW_LIST(2));

  if(copy) {
     SET_VECTOR_ELT(ans, 0, R_copyStruct_timeval(tv));
     SET_VECTOR_ELT(ans, 1, R_copyStruct_timezone(tz));
  } else {
     SET_VECTOR_ELT(ans, 0, TYPEOF(r_tv) == EXTPTRSXP ? r_tv : R_createNativeRefer
     SET_VECTOR_ELT(ans, 1, TYPEOF(r_tz) == EXTPTRSXP ? r_tz : R_createNativeRefer
  }
     /* Put names on the elements of the list ... */
  UNPROTECT(1);
}
```

A simple example

I have put together a simple, artificial example that allows us to test some of this in different ways. We start with C code

```
                                                                        R
#include "outargs.h"
void
myVoid(int x, A *a, B *b)
```

```
{
    a->x = x;
    a->y = 3.1415;
    b->str = "my string";
}



int
myInt(int x, A *a, B *b)
{
    myVoid(x, a, b);
    return(11);
}
```

which defines two routines which are very similar with one calling the other. The difference is that the second one returns an integer. The first returns nothing. Both routines take an integer and then two out arguments.

R

```
vv = parseTU.Perl("examples/outargs.c.t00.tu", "C")
r = getRoutines(vv)
types = DefinitionContainer()
rr = lapply(r, resolveType, vv, types)

rr$myInt$paramStyle = c("", "out", "out")
rr$myVoid$paramStyle = c("", "out", "out")

bindings = lapply(rr, createMethodBinding)

A = generateStructInterface(types$A, types)
B = generateStructInterface(types$B, types)

con = file("/tmp/Routargs.c", "w")
writeCode(A, "native", file = con,
              includes = c('"outargs.h"', "<Rdefines.h>", '"RConverters.h"'))
writeCode(B, "native", file = con )
writeCode(bindings, "native", file = con )
close(con)

con = file("/tmp/Routargs.R", "w")
writeCode(A, "r", file = con )
writeCode(B, "r", file = con )
writeCode(bindings, "r", file = con )
close(con)
```

R

```
cd tmp
R CMD SHLIB outargs.c Routargs.c  RConverters.c
```

R

```
dyn.load("/tmp/outargs.so")
source("/tmp/Routargs.R")


myInt(10, NULL, NULL)
myInt(10)

myInt(10, .copy = c('a' = NA, 'b' = NA))

myInt(10, NULL, NULL, .copy = c('a' = FALSE, 'b' = FALSE))


myVoid(10)
myVoid(10, .copy = c(a=TRUE, b= NA))
```

We now test the finalizers

R

```
a = new_A(.finalizer = TRUE)
a$x = 10
a$y = 20.4
as(a, "A")

aa = as(as(a, "A"), "ARef")

b = new_B(.finalizer = TRUE)
as(b, "B")

rm(a,b)
gc()
```

Now to check explicit freeing.

R

```
a = new_A()
Free(a)
```

Note

We need to work with more complex examples where we have pointers to other things and can exercise the recursive facility.

Copying Results back to R

We introduce the support for default values for out arguments. We also allow the user to specify which values to copy to R and which to leave as references and which to ignore entirely via the *.copy* parameter.

We can implement this in R or we can do it more succinctly in C. Let's look at the myInt example. We would end up with code like the following

R

```
SEXP
R_myInt(SEXP r_x, SEXP r_a, SEXP r_b, SEXP r__copy, SEXP r_resultLength)
{
  int r_ctr = 0;

  .....

 ans =  myInt ( x, _p_a, _p_b );

 PROTECT(r_ans = NEW_LIST( INTEGER(r_resultLength)[0] ));
 PROTECT(r_names = NEW_CHARACTER( INTEGER(r_resultLength)[0] ));

 SET_VECTOR_ELT(r_ans, r_ctr,  ScalarInteger ( ans ) );
 SET_STRING_ELT(r_names, r_ctr++, mkChar(".result"));

 if(LOGICAL(r__copy)[0] != NA_LOGICAL) {
     SET_VECTOR_ELT( r_ans, r_ctr , LOGICAL(r__copy)[0]  == FALSE && GET_LENGTH( r
     SET_STRING_ELT(r_names, r_ctr++, mkChar("a"));
 }

 if(LOGICAL(r__copy)[1] != NA_LOGICAL) {
     SET_VECTOR_ELT( r_ans, r_ctr , LOGICAL(r__copy)[1]  == FALSE && GET_LENGTH( r
     SET_STRING_ELT(r_names, r_ctr++, mkChar("b"));
 }
 SET_NAMES(r_ans, r_names);
 UNPROTECT( 2 );
```