Contents

How to contribute (/contribute.html)

Edit this page (https://github.com/hadley/adv-r/edit/master/C-interface.rmd)

R's C interface

Reading R's source code is an extremely powerful technique for improving your programming skills. However, many base R functions, and many functions in older packages, are written in C. It's useful to be able to figure out how those functions work, so this chapter will introduce you to R's C API. You'll need some basic C knowledge, which you can get from a standard C text (e.g., *The C Programming Language* (http://amzn.com/0131101633?tag=devtools-20) by Kernigan and Ritchie), or from Rcpp (Rcpp.html#rcpp). You'll need a little patience, but it is possible to read R's C source code, and you will learn a lot doing it.

The contents of this chapter draw heavily from Section 5 ("System and foreign language interfaces") of Writing R extensions (http://cran.r-project.org/doc/manuals/R-exts.html), but focus on best practices and modern tools. This means it does not cover the old `.C` interface, the old API defined in `Rdefines.h`, or rarely used language features. To see R's complete C API, look at the header file `Rinternals.h`. It's easiest to find and display this file from within R:

```
rinternals <- file.path(R.home("include"), "Rinternals.h")
file.show(rinternals)
```

All functions are defined with either the prefix `Rf_` or `R_` but are exported without it (unless `#define R_NO_REMAP` has been used).

I do not recommend using C for writing new high-performance code. Instead write C++ with Rcpp. The Rcpp API protects you from many of the historical idiosyncracies of the R API, takes care of memory management for you, and provides many useful helper methods.

Outline

- Calling C (C-interface.html#calling-c) shows the basics of creating and calling C functions with the inline package.
- C data structures (C-interface.html#c-data-structures) shows how to translate data structure names from R to C.
- Creating and modifying vectors (C-interface.html#c-vectors) teaches you how to create, modify, and coerce vectors in C.
- Pairlists (C-interface.html#c-pairlists) shows you how to work with pairlists. You need to know this because the distinction between pairlists and list is more important in C than R.
- Input validation (C-interface.html#c-input-validation) talks about the importance of input validation so that your C function doesn't crash R.
- Finding the C source for a function (C-interface.html#c-find-source) concludes the chapter by showing you how to find the C source code for internal and primitive R functions.

Prerequisites

To understand existing C code, it's useful to generate simple examples of your own that you can experiment with. To that end, all examples in this chapter use the `inline` package, which makes it extremely easy to compile and link C code to your current R session. Get it by running `install.packages("inline")`. To easily find the C code associated with internal and primitive functions, you'll need a function from pryr. Get the package with `install.packages("pryr")`.

You'll also need a C compiler. Windows users can use Rtools (http://cran.r-project.org/bin/windows/Rtools/). Mac users will need the Xcode command line tools (http://developer.apple.com/). Most Linux distributions will come with the necessary compilers.

In Windows, it's necessary that the Rtools executables directory (typically `C:\Rtools\bin`) and the C compiler executables directory (typically `C:\Rtools\gcc-4.6.3\bin`) are included in the Windows `PATH` environment variable. You may need to reboot Windows before R can recognise these values.

Calling C functions from R

Generally, calling a C function from R requires two pieces: a C function and an R wrapper function that uses `.Call()`. The simple function below adds two numbers together and illustrates some of the complexities of coding in C:

```
// In C --------------------------------------
#include <R.h>
#include <Rinternals.h>

SEXP add(SEXP a, SEXP b) {
  SEXP result = PROTECT(allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return result;
}
```

```
# In R ----------------------------------------
add <- function(a, b) {
  .Call("add", a, b)
}
```

(An alternative to using `.Call` is to use `.External`. It is used almost identically, except that the C function will receive a single argument containing a `LISTSXP`, a pairlist from which the arguments can be extracted. This makes it possible to write functions that take a variable number of arguments. However, it's not commonly used in base R and `inline` does not currently support `.External` functions so I don't discuss it further in this chapter.)

In this chapter we'll produce the two pieces in one step by using the `inline` package. This allows us to write:

```
add <- cfunction(c(a = "integer", b = "integer"), "
  SEXP result = PROTECT(allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return result;
")
add(1, 5)
```

```
## [1] 6
```

Before we begin reading and writing C code, we need to know a little about the basic data structures.

C data structures

At the C-level, all R objects are stored in a common datatype, the `SEXP`, or S-expression. All R objects are S-expressions so every C function that you create must return a `SEXP` as output and take `SEXP`s as inputs. (Technically, this is a pointer to a structure with typedef `SEXPREC`.) A `SEXP` is a variant type, with subtypes for all R's data structures. The most important types are:

- `REALSXP`: numeric vector
- `INTSXP`: integer vector
- `LGLSXP`: logical vector
- `STRSXP`: character vector
- `VECSXP`: list
- `CLOSXP`: function (closure)
- `ENVSXP`: environment

Beware: In C, lists are called `VECSXP`s not `LISTSXP`s. This is because early implementations of lists were Lisp-like linked lists, which are now known as "pairlists".

Character vectors are a little more complicated than the other atomic vectors. A `STRSXP`s contains a vector of `CHARSXP`s, where each `CHARSXP` points to C-style string stored in a global pool. This design allows individual `CHARSXP`'s to be shared between multiple character vectors, reducing memory usage. See object size (memory.html#object-size) for more details.

There are also `SEXP`s for less common object types:

- `CPLXSXP`: complex vectors
- `LISTSXP`: "pair" lists. At the R level, you only need to care about the distinction lists and pairlists for function arguments, but internally they are used in many more places
- `DOTSXP`: '...'
- `SYMSXP`: names/symbols
- `NILSXP`: `NULL`

And `SEXP`s for internal objects, objects that are usually only created and used by C functions, not R functions:

- `LANGSXP`: language constructs
- `CHARSXP`: "scalar" strings
- `PROMSXP`: promises, lazily evaluated function arguments
- `EXPRSXP`: expressions

There's no built-in R function to easily access these names, but pryr provides `sexp_type()`:

```
library(pryr)

sexp_type(10L)
```

```
## [1] "INTSXP"
```

```
sexp_type("a")
```

```
## [1] "STRSXP"
```

```
sexp_type(T)
```

```
## [1] "LGLSXP"
```

```
sexp_type(list(a = 1))
```

```
## [1] "VECSXP"
```

```
sexp_type(pairlist(a = 1))
```

```
## [1] "LISTSXP"
```

Creating and modifying vectors

At the heart of every C function are conversions between R data structures and C data structures. Inputs and output will always be R data structures (`SEXP`s) and you will need to convert them to C data structures in order to do any work. This section focusses on vectors because they're the type of object you're most likely to work with.

An additional complication is the garbage collector: if you don't protect every R object you create, the garbage collector will think they are unused and delete them.

Creating vectors and garbage collection

The simplest way to create a new R-level object is to use `allocVector()`. It takes two arguments, the type of `SEXP` (or `SEXPTYPE`) to create, and the length of the vector. The following code creates a three element list containing a logical vector, a numeric vector, and an integer vector, all of length four:

```
dummy <- cfunction(body = '
  SEXP dbls = PROTECT(allocVector(REALSXP, 4));
  SEXP lgls = PROTECT(allocVector(LGLSXP, 4));
  SEXP ints = PROTECT(allocVector(INTSXP, 4));

  SEXP vec = PROTECT(allocVector(VECSXP, 3));
  SET_VECTOR_ELT(vec, 0, dbls);
  SET_VECTOR_ELT(vec, 1, lgls);
  SET_VECTOR_ELT(vec, 2, ints);

  UNPROTECT(4);
  return vec;
')
dummy()
```

```
## [[1]]
## [1] 7.76e-317 7.80e-317 7.76e-317 7.75e-317
##
## [[2]]
## [1] TRUE TRUE TRUE TRUE
##
## [[3]]
## [1] 1 2 3 1
```

You might wonder what all the `PROTECT()` calls do. They tell R that the object is in use and shouldn't be deleted if the garbage collector is activated. (We don't need to protect objects that R already knows we're using, like function arguments.)

You also need to make sure that every protected object is unprotected. `UNPROTECT()` takes a single integer argument, `n`, and unprotects the last n objects that were protected. The number of protects and unprotects must match. If not, R will warn about a "stack imbalance in .Call". Other specialised forms of protection are needed in some circumstances:

- `UNPROTECT_PTR()` unprotects the object pointed to by the `SEXP`s.
- `PROTECT_WITH_INDEX()` saves an index of the protection location that can be used to replace the protected value using `REPROTECT()`.

Consult the R externals section on garbage collection (http://cran.r-project.org/doc/manuals/R-exts.html#Garbage-Collection) for more details.

Properly protecting the R objects you allocate is extremely important! Improper protection leads to difficulty diagnosing errors, typically segfaults, but other corruption is possible as well. In general, if you allocate a new R object, you must `PROTECT` it.

If you run `dummy()` a few times, you'll notice the output varies. This is because `allocVector()` assigns memory to each output, but it doesn't clean it out first. For real functions, you may want to loop through each element in the vector and set it to a constant. The most efficient way to do that is to use `memset()`:

```
zeroes <- cfunction(c(n_ = "integer"), '
  int n = asInteger(n_);

  SEXP out = PROTECT(allocVector(INTSXP, n));
  memset(INTEGER(out), 0, n * sizeof(int));
  UNPROTECT(1);

  return out;
')
zeroes(10);
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Missing and non-finite values

Each atomic vector has a special constant for getting or setting missing values:

- `INTSXP`: `NA_INTEGER`
- `LGLSXP`: `NA_LOGICAL`
- `STRSXP`: `NA_STRING`

Missing values are somewhat more complicated for `REALSXP` because there is an existing protocol for missing values defined by the floating point standard (IEEE 754 (http://en.wikipedia.org/wiki/IEEE_floating_point)). In doubles, an `NA` is `NaN` with a special bit pattern (the lowest word is 1954, the year Ross Ihaka was born), and there are other special values for positive and negative infinity. Use `ISNA()`, `ISNAN()`, and `!R_FINITE()` macros to check for missing, NaN, or non-finite values. Use the constants `NA_REAL`, `R_NaN`, `R_PosInf`, and `R_NegInf` to set those values.

We can use this knowledge to make a simple version of `is.NA()`:

```
is_na <- cfunction(c(x = "ANY"), '
  int n = length(x);

  SEXP out = PROTECT(allocVector(LGLSXP, n));

  for (int i = 0; i < n; i++) {
    switch(TYPEOF(x)) {
      case LGLSXP:
        LOGICAL(out)[i] = (LOGICAL(x)[i] == NA_LOGICAL);
        break;
      case INTSXP:
        LOGICAL(out)[i] = (INTEGER(x)[i] == NA_INTEGER);
        break;
      case REALSXP:
        LOGICAL(out)[i] = ISNA(REAL(x)[i]);
        break;
      case STRSXP:
        LOGICAL(out)[i] = (STRING_ELT(x, i) == NA_STRING);
        break;
      default:
        LOGICAL(out)[i] = NA_LOGICAL;
    }
  }
  UNPROTECT(1);

  return out;
')
is_na(c(NA, 1L))
```

```
## [1]  TRUE FALSE
```

```
is_na(c(NA, 1))
```

```
## [1]  TRUE FALSE
```

```
is_na(c(NA, "a"))
```

```
## [1]  TRUE FALSE
```

```
is_na(c(NA, TRUE))
```

```
## [1]  TRUE FALSE
```

Note that `base::is.na()` returns `TRUE` for both `NA` and `NaN`s in a numeric vector, as opposed to the C `ISNA()` macro, which returns `TRUE` only for `NA_REAL`s.

Accessing vector data

There is a helper function for each atomic vector that allows you to access the C array which stores the data in a vector. Use `REAL()`, `INTEGER()`, `LOGICAL()`, `COMPLEX()`, and `RAW()` to access the C array inside numeric, integer, logical, complex, and raw vectors. The following example shows how to use `REAL()` to inspect and modify a numeric vector:

```
add_one <- cfunction(c(x = "numeric"), "
  int n = length(x);
  SEXP out = PROTECT(allocVector(REALSXP, n));

  for (int i = 0; i < n; i++) {
    REAL(out)[i] = REAL(x)[i] + 1;
  }
  UNPROTECT(1);

  return out;
")
add_one(as.numeric(1:10))
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

When working with longer vectors, there's a performance advantage to using the helper function once and saving the result in a pointer:

```
add_two <- cfunction(c(x = "numeric"), "
  int n = length(x);
  double *px, *pout;

  SEXP out = PROTECT(allocVector(REALSXP, n));

  px = REAL(x);
  pout = REAL(out);
  for (int i = 0; i < n; i++) {
    pout[i] = px[i] + 2;
  }
  UNPROTECT(1);

  return out;
")
add_two(as.numeric(1:10))
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12
```

```
x <- as.numeric(1:1e6)
microbenchmark(
  add_one(x),
  add_two(x)
)
```

```
## Unit: milliseconds
##        expr  min   lq mean median  uq  max neval
##  add_one(x) 5.46 8.57 9.08   8.89 9.9 42.3   100
##  add_two(x) 1.33 4.29 5.03   4.59 5.6 38.6   100
```

On my computer, `add_two()` is about twice as fast as `add_one()` for a million element vector. This is a common idiom in base R.

Character vectors and lists

Strings and lists are more complicated because the individual elements of a vector are `SEXP`s, not basic C data structures. Each element of a `STRSXP` is a `CHARSXP`s, an immutable object that contains a pointer to C string stored in a global pool. Use `STRING_ELT(x, i)` to extract the `CHARSXP`, and `CHAR(STRING_ELT(x, i))` to get the actual `const char*` string. Set values with `SET_STRING_ELT(x, i, value)`. Use `mkChar()` to turn a C string into a `CHARSXP` and `mkString()` to turn a C string into a `STRSXP`. Use `mkChar()` to create strings to insert in an existing vector, use `mkString()` to create a new (length 1) vector.

The following function shows how to make a character vector containing known strings:

```
abc <- cfunction(NULL, '
  SEXP out = PROTECT(allocVector(STRSXP, 3));

  SET_STRING_ELT(out, 0, mkChar("a"));
  SET_STRING_ELT(out, 1, mkChar("b"));
  SET_STRING_ELT(out, 2, mkChar("c"));

  UNPROTECT(1);

  return out;
')
abc()
```

```
## [1] "a" "b" "c"
```

Things are a little harder if you want to modify the strings in the vector because you need to know a lot about string manipulation in C (which is hard, and harder to do right). For any problem that involves any kind of string modification, you're better off using Rcpp.

The elements of a list can be any other `SEXP`, which generally makes them hard to work with in C (you'll need lots of `switch` statements to deal with the possibilities). The accessor functions for lists are `VECTOR_ELT(x, i)` and `SET_VECTOR_ELT(x, i, value)`.

Modifying inputs

You must be very careful when modifying function inputs. The following function has some rather unexpected behaviour:

```
add_three <- cfunction(c(x = "numeric"), '
  REAL(x)[0] = REAL(x)[0] + 3;
  return x;
')
x <- 1
y <- x
add_three(x)
```

```
## [1] 4
```

```
x
```

```
## [1] 4
```

```
y
```

```
## [1] 4
```

Not only has it modified the value of `x`, it has also modified `y`! This happens because of R's lazy copy-on-modify semantics. To avoid problems like this, always `duplicate()` inputs before modifying them:

```
add_four <- cfunction(c(x = "numeric"), '
  SEXP x_copy = PROTECT(duplicate(x));
  REAL(x_copy)[0] = REAL(x_copy)[0] + 4;
  UNPROTECT(1);
  return x_copy;
')
x <- 1
y <- x
add_four(x)
```

```
## [1] 5
```

```
x
```

```
## [1] 1
```

```
y
```

```
## [1] 1
```

If you're working with lists, use `shallow_duplicate()` to make a shallow copy; `duplicate()` will also copy every element in the list.

Coercing scalars

There are a few helper functions that turn length one R vectors into C scalars:

- `asLogical(x): LGLSXP -> int`
- `asInteger(x): INTSXP -> int`
- `asReal(x): REALSXP -> double`
- `CHAR(asChar(x)): STRSXP -> const char*`

And helpers to go in the opposite direction:

- `ScalarLogical(x): int -> LGLSXP`
- `ScalarInteger(x): int -> INTSXP`
- `ScalarReal(x): double -> REALSXP`
- `mkString(x): const char* -> STRSXP`

These all create R-level objects, so they need to be `PROTECT()`ed.

Long vectors

As of R 3.0.0, R vectors can have length greater than $2^{31} - 1$. This means that vector lengths can no longer be reliably stored in an `int` and if you want your code to work with long vectors, you can't write code like `int n = length(x)`. Instead use the `R_xlen_t` type and the `xlength()` function, and write `R_xlen_t n = xlength(x)`.

Pairlists

In R code, there are only a few instances when you need to care about the difference between a pairlist and a list (as described in Pairlists (Expressions.html#pairlists)). In C, pairlists play much more important role because they are used for calls, unevaluated arguments, attributes, and in `...`. In C, lists and pairlists differ primarily in how you access and name elements.

Unlike lists (`VECSXP`s), pairlists (`LISTSXP`s) have no way to index into an arbitrary location. Instead, R provides a set of helper functions that navigate along a linked list. The basic helpers are `CAR()`, which extracts the first element of the list, and `CDR()`, which extracts the rest of the list. These can be composed to get `CAAR()`, `CDAR()`, `CADDR()`, `CADDDR()`, and so on. Corresponding to the getters, R provides setters `SETCAR()`, `SETCDR()`, etc.

The following example shows how `CAR()` and `CDR()` can pull out pieces of a quoted function call:

```
car <- cfunction(c(x = "ANY"), 'return CAR(x);')
cdr <- cfunction(c(x = "ANY"), 'return CDR(x);')
cadr <- cfunction(c(x = "ANY"), 'return CADR(x);')

x <- quote(f(a = 1, b = 2))
# The first element
car(x)
```

```
## f
```

```
# Second and third elements
cdr(x)
```

```
## $a
## [1] 1
##
## $b
## [1] 2
```

```
# Second element
car(cdr(x))
```

```
## [1] 1
```

```
cadr(x)
```

```
## [1] 1
```

Pairlists are always terminated with `R_NilValue`. To loop over all elements of a pairlist, use this template:

```
count <- cfunction(c(x = "ANY"), '
  SEXP el, nxt;
  int i = 0;

  for(nxt = x; nxt != R_NilValue; el = CAR(nxt), nxt = CDR(nxt)) {
    i++;
  }
  return ScalarInteger(i);
')
count(quote(f(a, b, c)))
```

```
## [1] 4
```

```
count(quote(f()))
```

```
## [1] 1
```

You can make new pairlists with `CONS()` and new calls with `LCONS()`. Remember to set the last value to `R_NilValue`. Since these are R objects as well, they are eligible for garbage collection and must be `PROTECT`ed. In fact, it is unsafe to write code like the following:

```
new_call <- cfunction(NULL, '
  return LCONS(install("+"), LCONS(
    ScalarReal(10), LCONS(
      ScalarReal(5), R_NilValue
    )
  ));
')
gctorture(TRUE)
new_call()
```

```
## 5 + 5
```

```
gctorture(FALSE)
```

On my machine, I get the result `5 + 5` — highly unexpected! In fact, to be safe, we must `PROTECT` each `ScalarReal` that is generated, as every R object allocation can trigger the garbage collector.

```
new_call <- cfunction(NULL, '
  SEXP REALSXP_10 = PROTECT(ScalarReal(10));
  SEXP REALSXP_5 = PROTECT(ScalarReal(5));
  SEXP out = PROTECT(LCONS(install("+"), LCONS(
    REALSXP_10, LCONS(
      REALSXP_5, R_NilValue
    )
  )));
  UNPROTECT(3);
  return out;
')
gctorture(TRUE)
new_call()
```

```
## 10 + 5
```

```
gctorture(FALSE)
```

`TAG()` and `SET_TAG()` allow you to get and set the tag (aka name) associated with an element of a pairlist. The tag should be a symbol. To create a symbol (the equivalent of `as.symbol()` in R), use `install()`.

Attributes are also pairlists, but come with the helper functions `setAttrib()` and `getAttrib()`:

```
set_attr <- cfunction(c(obj = "SEXP", attr = "SEXP", value = "SEXP"), '
  const char* attr_s = CHAR(asChar(attr));

  duplicate(obj);
  setAttrib(obj, install(attr_s), value);
  return obj;
')
x <- 1:10
set_attr(x, "a", 1)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
## attr(,"a")
## [1] 1
```

(Note that `setAttrib()` and `getAttrib()` must do a linear search over the attributes pairlist.)

There are some (confusingly named) shortcuts for common setting operations: `classgets()`, `namesgets()`, `dimgets()`, and `dimnamesgets()` are the internal versions of the default methods of `class<-`, `names<-`, `dim<-`, and `dimnames<-`.

Input validation

If the user provides unexpected input to your function (e.g., a list instead of a numeric vector), it's very easy to crash R. For this reason, it's a good idea to write a wrapper function that checks arguments are of the correct type. It's usually easier to do this at the R level. For example, going back to our first example of C code, we might rename the C function to `add_` and write a wrapper around it to check that the inputs are ok:

```
add_ <- cfunction(signature(a = "integer", b = "integer"), "
  SEXP result = PROTECT(allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return result;
")
add <- function(a, b) {
  stopifnot(is.numeric(a), is.numeric(b))
  stopifnot(length(a) == 1, length(b) == 1)
  add_(a, b)
}
```

Alternatively, if we wanted to be more accepting of diverse inputs we could do the following:

```
add <- function(a, b) {
  a <- as.numeric(a)
  b <- as.numeric(b)

  if (length(a) > 1) warning("Only first element of a used")
  if (length(b) > 1) warning("Only first element of b used")

  add_(a, b)
}
```

To coerce objects at the C level, use `PROTECT(new = coerceVector(old, SEXPTYPE))`. This will return an error if the `SEXP` can not be converted to the desired type.

To check if an object is of a specified type, you can use `TYPEOF`, which returns a `SEXPTYPE`:

```
is_numeric <- cfunction(c("x" = "ANY"), "
  return ScalarLogical(TYPEOF(x) == REALSXP);
")
is_numeric(7)
```

```
## [1] TRUE
```

```
is_numeric("a")
```

```
## [1] FALSE
```

There are also a number of helper functions which return 0 for FALSE and 1 for TRUE:

- For atomic vectors: `isInteger()`, `isReal()`, `isComplex()`, `isLogical()`, `isString()`.
- For combinations of atomic vectors: `isNumeric()` (integer, logical, real), `isNumber()` (integer, logical, real, complex), `isVectorAtomic()` (logical, integer, numeric, complex, string, raw).
- For matrices (`isMatrix()`) and arrays (`isArray()`).
- For more esoteric objects: `isEnvironment()`, `isExpression()`, `isList()` (a pair list), `isNewList()` (a list), `isSymbol()`, `isNull()`, `isObject()` (S4 objects), `isVector()` (atomic vectors, lists, expressions).

Note that some of these functions behave differently to similarly named R functions with similar names. For example `isVector()` is true for atomic vectors, lists, and expressions, where `is.vector()` returns `TRUE` only if its input has no attributes apart from names.

Finding the C source code for a function

In the base package, R doesn't use `.Call()`. Instead, it uses two special functions: `.Internal()` and `.Primitive()`. Finding the source code for these functions is an arduous task: you first need to look for their C function name in `src/main/names.c` and then search the R source code. `pryr::show_c_source()` automates this task using GitHub code search:

```
tabulate
```

```
## function (bin, nbins = max(1L, bin, na.rm = TRUE))
## {
##     if (!is.numeric(bin) && !is.factor(bin))
##         stop("'bin' must be numeric or a factor")
##     if (typeof(bin) != "integer")
##         bin <- as.integer(bin)
##     if (nbins > .Machine$integer.max)
##         stop("attempt to make a table with >= 2^31 elements")
##     nbins <- as.integer(nbins)
##     if (is.na(nbins))
##         stop("invalid value of 'nbins'")
##     .Internal(tabulate(bin, nbins))
## }
## <bytecode: 0x50dc4b8>
## <environment: namespace:base>
```

```
pryr::show_c_source(.Internal(tabulate(bin, nbins)))
#> tabulate is implemented by do_tabulate with op = 0
```

This reveals the following C source code (slightly edited for clarity):

```
SEXP attribute_hidden do_tabulate(SEXP call, SEXP op, SEXP args,
                                  SEXP rho) {
  checkArity(op, args);
  SEXP in = CAR(args), nbin = CADR(args);
  if (TYPEOF(in) != INTSXP)  error("invalid input");

  R_xlen_t n = XLENGTH(in);
  /* FIXME: could in principle be a long vector */
  int nb = asInteger(nbin);
  if (nb == NA_INTEGER || nb < 0)
    error(_("invalid '%s' argument"), "nbin");

  SEXP ans = allocVector(INTSXP, nb);
  int *x = INTEGER(in), *y = INTEGER(ans);
  memset(y, 0, nb * sizeof(int));
  for(R_xlen_t i = 0 ; i < n ; i++) {
    if (x[i] != NA_INTEGER && x[i] > 0 && x[i] <= nb) {
      y[x[i] - 1]++;
    }
  }

  return ans;
}
```

Internal and primitive functions have a somewhat different interface than `.Call()` functions. They always have four arguments:

- `SEXP call`: the complete call to the function. `CAR(call)` gives the name of the function (as a symbol); `CDR(call)` gives the arguments.
- `SEXP op`: an "offset pointer". This is used when multiple R functions use the same C function. For example `do_logic()` implements `&`, `|`, and `!`. `show_c_source()` prints this out for you.
- `SEXP args`: a pairlist containing the unevaluated arguments to the function.
- `SEXP rho`: the environment in which the call was executed.

This gives internal functions an incredible amount of flexibility as to how and when the arguments are evaluated. For example, internal S3 generics call `DispatchOrEval()` which either calls the appropriate S3 method or evaluates all the arguments in place. This flexibility come at a price, because it makes the code harder to understand. However, evaluating the arguments is usually the first step and the rest of the function is straightforward.

The following code shows `do_tabulate()` converted into standard a `.Call()` interface:

```
tabulate2 <- cfunction(c(bin = "SEXP", nbins = "SEXP"), '
  if (TYPEOF(bin) != INTSXP)  error("invalid input");

  R_xlen_t n = XLENGTH(bin);
  /* FIXME: could in principle be a long vector */
  int nb = asInteger(nbins);
  if (nb == NA_INTEGER || nb < 0)
    error("invalid \'%s\' argument", "nbin");

  SEXP ans = allocVector(INTSXP, nb);
  int *x = INTEGER(bin), *y = INTEGER(ans);
  memset(y, 0, nb * sizeof(int));
  for(R_xlen_t i = 0 ; i < n ; i++) {
    if (x[i] != NA_INTEGER && x[i] > 0 && x[i] <= nb) {
      y[x[i] - 1]++;
    }
  }

  return ans;
')
tabulate2(c(1L, 1L, 1L, 2L, 2L), 3)
```

```
## [1] 3 2 0
```

To get this to compile, I also removed the call to `_()` which is an internal R function used to translate error messages between different languages.

The final version below moves more of the coercion logic into an accompanying R function, and does some minor restructuring to make the code a little easier to understand. I also added a `PROTECT()`; this is probably missing in the original because the author knew that it would be safe.

```
tabulate_ <- cfunction(c(bin = "SEXP", nbins = "SEXP"), '
  int nb = asInteger(nbins);

  // Allocate vector for output - assumes that there are
  // less than 2^32 bins, and that each bin has less than
  // 2^32 elements in it.
  SEXP out = PROTECT(allocVector(INTSXP, nb));
  int *pbin = INTEGER(bin), *pout = INTEGER(out);
  memset(pout, 0, nb * sizeof(int));

  R_xlen_t n = xlength(bin);
  for(R_xlen_t i = 0; i < n; i++) {
    int val = pbin[i];
    if (val != NA_INTEGER && val > 0 && val <= nb) {
      pout[val - 1]++; // C is zero-indexed
    }
  }
  UNPROTECT(1);

  return out;
')

tabulate3 <- function(bin, nbins) {
  bin <- as.integer(bin)
  if (length(nbins) != 1 || nbins <= 0 || is.na(nbins)) {
    stop("nbins must be a positive integer", call. = FALSE)
  }
  tabulate_(bin, nbins)
}
tabulate3(c(1, 1, 1, 2, 2), 3)
```

```
## [1] 3 2 0
```