Native Interfaces for R: Self-Study Exercises

Seth Falcon

20-21 May, 2010

1 Exercises

The AdvancedR package contains the sources of the nidemo package that demonstrates how to use R's native interfaces. The exercises below will take the nidemo package code as a starting point, so you will want to have the source for this package at hand. If you have installed the AdvancedR package, you can find the sources for nidemo as follows:

```
> system.file("packages", package = "AdvancedR")

[1] "/tmp/Rinst2381276910/AdvancedR/packages"
```

The nidemo package contains two versions of a function that computes the occurrence counts of the letters a–z in a text file. The `alpha_freq_R` function is written entirely in R and the `alpha_freq` function performs the letter counting in a C function accessed via `.Call`.

Exercise 1
Install the nidemo package and test out both `alpha_freq_R` and `alpha_freq`. On unix-like systems, a good input file to use for testing is the system word dictionary (possibly in `/usr/share/dict/words`).

Use `system.time` to compare the run time for the two versions. You may not see much of a difference if your input file is small.

The C function `nid_alpha_freq` in `nidemo/src/charfreq.c` is not very robust. In particular, the function does not verify that the argument it receives is of the type that it expects. When a bad argument is provides, it will crash R.

Exercise 2
Try the following inputs to see the ways in which the function breaks (NOTE: some or all of these may crash your R session).

- `alpha_freq(TRUE)`
- `alpha_freq("")`
- `alpha_freq(as.character(NA))`

- *alpha_freq(character(0))*

Exercise 3
Modify the `nid_alpha_freq` function to display a standard R error message when the argument provided is invalid. Do this in C, not in the R wrapper function using `Rf_error`. After your changes, the function should provide an error message for all of the invalid inputs from Exercise 2. Hint: you can use `Rf_isString` to test for a character vector.

Exercise 4
Force the input to be `character(1)` by making it an error if the input is not of the right length. Do this in C.

Exercise 5
Use `TYPEOF()` and `Rf_type2char()` to provide a more informative error message when the input is not a character vector.

Exercise 6
The labels for the result table are set in the R wrapper for `alpha_freq`. Modify the code to set the names of the result table in C. Outline:

- *Create a character vector using `Rf_allocVector(STRSXP, 26)`*
- *Use `SET_STRING_ELT` and `mkChar` to fill the vector*
- *Set this as the names attribute of the return SEXP using `Rf_setAttrib`. The attribute name is `R_NamesSymbol`.*

Exercise 7
At the top of `nid_alpha_freq`, an INTSXP is allocated using `Rf_allocVector`. This vector is protected a few lines later. Is this safe? Discuss.

Exercise 8
Enhance `alpha_freq` so that it can be given a length n vector of file names and return an $n \times 26$ matrix with the alphabet frequency count for each file. Set the row names of the matrix to be the corresponding file names. There is an example of returning a matrix from a C function in WRE.

2 R Package Development Setup

2.1 Makevars customization

You can ensure that debugging symbols and compiler warnings are enabled for all packages that you build from source by creating a `Makevars-PLATFORM` file in your home directory (where PLATFORM matches the output of `R.version$platform`.

```
> dir.create("~/.R", showWarnings = FALSE)
> lines <- "CFLAGS=-g -Wall -pedantic"
> fname <- paste("~/.R/Makevars",
+                R.version[["platform"]],
+                sep = "-")
> writeLines(lines, con = fname)
```

2.2 Useful items for $HOME/.Rprofile

- Install packages based on the R_LIBS_USER environment variable. This keeps installed packages separate from the packages that come with R and also has the advantage of avoiding mixing packages across different versions of R.
- Set a default CRAN repository
- Source biocLite script for interactive sessions.
- A helper function to reload a package, useful during development.

```
dir.create(Sys.getenv("R_LIBS_USER"),
           recursive = TRUE,
           showWarnings = FALSE)

options(repos="http://cran.fhcrc.org")

if (TRUE && interactive()) {
tryCatch({
    source("http://bioconductor.org/biocLite.R")
}, error=function(e) invisible(NULL),
         warning=function(w) message("Not connected to the net"))
}

reload_pkg <- function(p)
{
    detach(paste("package", p, sep = ":"),
           unload = TRUE, character.only = TRUE)
    library(p, character.only = TRUE)
}
```

2.3 Windows Setup

For reference, read the *Installing R under Windows* in the *R Installation and Administration* manual.

Download and install the standard R package for Windows.

Download the appropriate version of the Windows tools from http://www.murdoch-sutherland.com/Rtools/ and install.

Install a package from source as:

```
c:\"Program Files\R\R-2.11.0"\bin\R CMD INSTALL nidemo
```

3 Exploration of External Pointers

R provides the EXTPTRSXP type for managing memory that is external to R. You can register a *finalizer* that will run when an external pointers is garbage collected. See Figure 1.

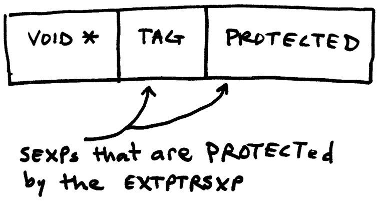

Figure 1: Conceptual diagram of R's external pointer type (EXTPTRSXP)

The nidemo package contains an example of using R's external pointers. The code is somewhat contrived, but extends the alphabet frequency counter to be a persistent data structure such that you can create a counter and update it with the contents of different files and then retrieve the cumulative alphabet frequency counts.

A user session looks like this:

```
> x <- make_freq("alice")
> update_freq(x, "some/file.txt")
> update_freq(x, "some/another.txt")
> report_freq(x)
> rm(x)
> gc()
```

Exercise 9
Read through the implementation of the external pointer based alphabet frequency counter and try out a sample session.

In addition to registering finalizers at the C-level, it is also possible to create R-based finalizers using `reg.finalizer`. One subtlety of finalizers is that any functions called within a finalizer must satisfy a weak-form of re-entrancy. Consider the pseudo-code in Figure 2.

4 Calling R from C

You can evaluate R code from C using `Rf_eval`. This allows package code to make use of R functions that do not have a C entry point as well as provide a

```
f() {
  /* amazing computations */
  g() /* this allocates R objects,
         can trigger gc */
  /* more computations here */
}

finalizer() {
  /* clean up */
  f() /* danger! */
}
```

Figure 2: Even though R is single-threaded, a second call to `f` will occur before the first call to `f` is finished. This can cause serious problems if `f` manipulates global variables or modifies its arguments in-place.

mechanism for users to specify callback functions in R that will be executed in the context of package C code.

Exercise 10
Take a few minutes to review the section Evaluating R expressions from C in the WRE manual.

The most elegant way of calling R functions from C is to build a function call using a LANGSXP and then evaluate it using `Rf_eval`. R function calls are represented in C using the LANGSXP type that provides a linked list data structure. The first element of the list must be a symbol naming the function you want to call (SYMSXP). Additional elements in the list are the arguments to the function.

There are helper functions to construct pairlists of different sizes: `lang1`, `lang2`, `lang3`, `lang4`. Below, `Rf_lang2` is used to call `path.expand` at the C level.

```
PROTECT(v = Rf_mkString("~/src/somefile.txt"));
PROTECT(fun = Rf_lang2(Rf_install("path.expand"), v));
ans = eval(fun, R_BaseEnv);
```

Exercise 11
Enhance the `nid_alpha_freq` function in `nidemo/src/charfreq.c` so that file names are expanded. Do this by constructing a call to `path.expand` at the C level.