Peter Dalgaard
First UseR! Conference
Vienna, May 2004

Language interfaces
`.Call` and `.External`

Introduction

- The `.C` and `.Fortran` functions are commonly used for interfacing to numerical routines
- However, they have shortcomings for advanced use: Only certain data types can be passed, and quite a bit of storage allocation and data conversion happens in interpreted code
- `.Call` and `.External` allow R objects to be passed to and returned from compiled C code

This is an elementary introduction, but I shall assume that you have a fairly good working knowledge of the C language.

Plan

- Differences between `.C`, `.Call`, and `.External`
- Basic usage
- Things to do in C code
  - R object internals
  - Accessing R vectors and creating new ones
  - Dealing with internal list structures, expressions, etc.
  - The garbage collector and how to keep things out of its way
  - The write barrier
  - Parsing and evaluating R code

Synopsis of the interfaces

From "Writing R Extensions":

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

.Call("convolve2", a, b)
.External("convolveE", a, b)
```

Notice that `.C` requires quite a lot of "red tape", whereas the others tend to be simpler (but of course they need to do the same things, only on the C side).

`.Call` vs. `.External`

- Very similar. Identical on the R side; the C side of `.Call` gets a fixed number of arguments, whereas `.External` passes an argument list (of any length).
- `.External` is based on `.Internal` which is used for R internals, but `.Call` the same access to R internals
- `.Call` has origins in S version 4. "Translation macros" (in `Rdefines.h`) allow same code to work with both R and S-PLUS
- The R source code (excl. recommended packages) has many more calls to `.Call` than to `.External` but very little use of the macros in `Rdefines.h`

An example of `.External`

From the `tcltk` package

```
SEXP RTcl_StringFromObj(SEXP args)
{   char *str;
    str = Tcl_GetStringFromObj(
            (Tcl_Obj *) R_ExternalPtrAddr(CADR(args)),
            NULL);
    return mkString(str);
}
```

Notice: `CADR` to get argument, `mkString` to make result an R object.

The R interface is

```
tclvalue.tclObj <- function(x)
   .External("RTcl_StringFromObj", x, PACKAGE="tcltk")
```


R object structures

- The SEXPREC and SEXP types (Symbolic EXPression RECord/Pointer)
- (You'll need to know about these, at least when debugging)
- 22 subtypes, some esoteric. Mostly you need:
  - vectors (LGLSXP, INTSXP, REALSXP, CPLXSXP, STRSXP, VECSXP, EXPRSXP)
  - list-alikes (LISTSXP, LANGSXP)
  - symbols and strings (SYMSXP, CHARSXP)

Inside SEXPs

- Basically a SEXP is a header struct + a union construct
- A major special case is made of the VECTOR_SEXPREC which uses a slightly shorter structure immediately followed by data
- Other subtypes are generally a header plus a 3-pointer structure (CAR/CDR/TAG for lists, formals/body/env for functions, etc.)

Accessing and creating vector types

Excerpt from `RTcl_ObjAsDoubleVector`:

```
ans = allocVector(REALSXP, count);
for (i = 0 ; i < count ; i++){
    ret = Tcl_GetDoubleFromObj(RTcl_interp, elem[i], &x);
    if (ret != TCL_OK) x = NA_REAL;
    REAL(ans)[i] = x;
}
```

Things to notice:

- `REAL(ans)` gives a pointer to the base of an array, which can be indexed as usual
- `NA_REAL` to encode missing values
- Allocation with `allocVector`

Character vectors

Similar code from `RTcl_ObjAsCharVector`:

```
PROTECT(ans = allocVector(STRSXP, count));
for (i = 0 ; i < count ; i++)
    SET_STRING_ELT(ans, i,
        mkChar(Tcl_GetStringFromObj(elem[i], NULL)));
UNPROTECT(1);
```

Things to notice:

- Need to use `mkChar()` to generate CHARSXP object
- Need to use `SET_STRING_ELT` to change element of vector (write barrier)
- Need to `PROTECT`

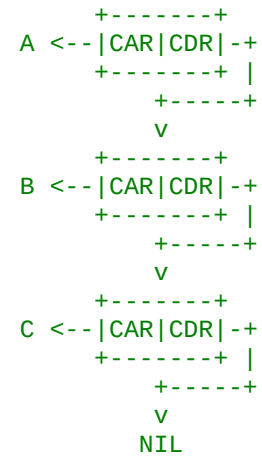
List-like structures

This requires a bit of explanation...

- R is internally based on Scheme, a variant of LISP
- "Lists" in R are really VECSXP objects (generic vector)
- Internally, we have LISTSXP objects, which are similar to LISP lists
- These are (almost) invisible at the R level
- LANGSXP objects are structurally similar to LISTSXP; EXPRSXP objects are like VECSXPs with (mostly) LANGSXP elements

CAR and CDR

Lists, traditionally written `(A B C)`, are constructed from paired pointers (apologies for the graphics...)



But what *is* CAR and CDR?

- LISP folklore
- Holdover from early IBM 704 series computers (vacuum-tube!)
- Content of Address Register
- Content of Decrement Register
- Terms sort of stuck, partly because of "cute" abbreviations like `CADDR(x)` for `CAR(CDR(CDR(x)))`

Pairlists in R

- Argument lists (formal and actual)
- Calls (unevaluated)
- Actually, contains *three* pointers, `carval`, `cdrval`, `tagval`
- The latter is used for named arguments, as in `f(a=1,b=2,3)`

Handling argument lists in `.External`

For up to four fixed arguments, use `CADR(lst)`, `CADDR(lst)`, `CADDDR(lst)`, `CAD4R(lst)`

(`CAR(lst)` is the function name, so skipped)

for more than 4 arguments you might use a loop

```
for ( p = CDR(lst); p != R_NilValue ; p = CDR(p)){
    ...
    handle CAR(p)
    ...
}
```

Notice that for a fixed number of arguments with a fixed meaning, you might as well use `.Call`.

Unevaluated code

- The kind returned from `quote()` or `substitute()`
- Can be a SYMSXP
- . . . or an atomic constant . . .
- . . . or a LANGSXP . . .
- . . . which is essentially a (pair-)list of the above element types
- So, e.g. `f(a,2+2)` is internally represented as a list `(f a (+ 2 2))`

Constructing lists

Use `lst=CONS(CAR,CDR)` or `LCONS` for LANGSXPs. Excerpt from `R_call` in package `tcltk`

```
alist = R_NilValue;
for (i = argc - 1 ; i > 1 ; i--){
    PROTECT(alist);
    alist = LCONS(mkString(argv[i]), alist);
    UNPROTECT(1);
}

fun = (SEXP) strtoul(argv[1], NULL, 16);

expr = LCONS(fun, alist);
expr = LCONS(install("try"), LCONS(expr, R_NilValue));

ans = eval(expr, R_GlobalEnv);
```


PROTECTing yourself

- R constantly creates and discards objects. So as not to run out of memory, objects must be reclaimed periodically.
- When this happens, you had better hold on to objects that you want to keep!
- A *protection stack* is maintained:

```
for (i = argc - 1 ; i > 1 ; i--){
    PROTECT(alist);
    alist = LCONS(mkString(argv[i]), alist);
    UNPROTECT(1);
}
```

- `PROTECT(obj)` pushes the object onto the protection stack. `UNPROTECT(n)` pops the top n objects off the stack.

What *not* to PROTECT

- In general it is better to PROTECT too often. If you miss a PROTECT, you will have code that *almost* always runs
- On the other hand, superfluous protection may clutter the code and make it hard to maintain
- You do not need to PROTECT
  1. when you really don't need the object any more
  2. when the object is part of an object that is already protected
  3. across calls where no allocation is involved

The write barrier

- Did you wonder about the following difference?

```
REAL(ans)[i] = x;
SET_STRING_ELT(ans, i, x);
```

- Why not `STRING(ans)[i] = x;` ?
- *Generational garbage collector*: New objects more likely to be reclaimed.
- Need to keep track of age and what happens when two objects are combined.
- `SET_STRING_ELT` et al. constitute a *write barrier*.
- For efficiency, there is normally no verification that the write barrier is not bypassed (configuration option).

Parsing and evaluating from C

From the `R_eval` command in the R-Tcl/Tk interface:

```
text = PROTECT(allocVector(STRSXP, argc - 1));
for (i = 1 ; i < argc ; i++)
    SET_STRING_ELT(text, i-1, mkChar(argv[i]));

expr = PROTECT(R_ParseVector(text, -1, &status));
if (status != PARSE_OK) {....}

n = length(expr);
for(i = 0 ; i < n ; i++)
    ans = eval(VECTOR_ELT(expr, i), R_GlobalEnv);
```


Demo

```
#include<Rdefines.h>
SEXP printargs(SEXP alist)
{
        SEXP p, ans; int n;
        for (p = alist, n = 0; p != R_NilValue ; p = CDR(p), n++)
                PrintValue(CAR(p));
        ans = allocVector(INTSXP, 1);
        INTEGER(ans)[0] = n;
        return ans;
}
---
R CMD SHLIB demo.c
---
dyn.load("demo.so")
.External("printargs",1,2,3:5,"hello")
```


Things that got skipped

— and how to move on

- Coercion
- S4 methods at C level
- Dealing with the context stack and environments
- Defining and accessing variables
- Check out “Writing R Extension”
- Look in the include files (beware of things that are sitting in `#ifndef USE_WRITE_BARRIER` though!)
- Use the R-devel list