Using C and C++ with R

Laurent Gatto lg390@cam.ac.uk

University of Cambridge

May 16, 2013

Plan

Calling foreign languages

Build-in `C` interface

The `Rcpp` package

Plan

Calling foreign languages

Build-in C interface

The Rcpp package

Foreign languages

- `C` , `C++`
- `Fortran`
- `Java`[1].

Other scripting languages

- R/Perl[2] and R/Python[3] bidirectional interfaces.
- There is also the `system()` function for direct access to OS functions.

[1] http://www.rforge.net/rJava/
[2] http://www.omegahat.org/RSPerl/
[3] http://www.omegahat.org/RSPython/

- **Why?** R is getting slow or is not doing well in terms of memory management.
- **When?** R can't do better **and** the slow code has been identified $\rightarrow$ code profiling.

- **Why?** R is getting slow or is not doing well in terms of memory management.
- **When?** R can't do better **and** the slow code has been identified $\rightarrow$ code profiling.

- **Why?** Re-using existing infrastructure

Requirement for C/C++

Working compilers. On Windows, Rtools[1,2]. On Mac, Xcode[3,4].

1. http://cran.r-project.org/bin/windows/Rtools/
2. http://cran.r-project.org/doc/manuals/R-admin.html#The-Windows-toolset
3. http://cran.r-project.org/doc/manuals/R-admin.html#Installing-R-under-_0028Mac_0029-OS-X
4. http://cran.r-project.org/doc/manuals/R-admin.html#Mac-OS-X

Plan

Calling foreign languages

Build-in C interface

The Rcpp package

The R C API

- Very frequent in R but with its quirks.
- Better know how to program in C.
- Documentation is not always easy to follow: R-Ext, R Internals as well as R and other package's code.

We need a C implementation and an R function that calls the R code.

`.C`

- Not recommended.
- Arguments and return values must be *primitives* (vectors of doubles or integers).

`.Call`

- Accepts any `R` data structures as arguments and return values (no type checking is done though).
- Manual memory management: allocate memory, protect objects to avoid them being garbage collected and subsequently unprotect them.

S-expression

SEXP is a super-type that matches all R data structures. Each data type has its own SEXP sub-type.

- REALSXP and INTSXP for double and integer vectors
- LGLSXP and STRSXP for logical and character vectors
- VECSXP for a list (NB: R list are called vectors at the C level)
- ...

Function input and outputs are always SEXP and will have to be coerced to the appropriate SXP sub-type.
Rinternals.h defines all C functions, data types and macros.

```
file.path(R.home(), "include", "Rinternals.h")
```

```
## [1] "/usr/local/lib/R/include/Rinternals.h"
```

```
library("inline")

## From Hadley Wickham, devtools wiki, adapted from inspect.c
## https://github.com/hadley/devtools/wiki/C-interface
sexp_type <- cfunction(c(x = "ANY"), '
  switch (TYPEOF(x)) {
    case NILSXP:      return mkString("NILSXP");
    case SYMSXP:      return mkString("SYMSXP");
    case LISTSXP:     return mkString("LISTSXP");
    case CLOSXP:      return mkString("CLOSXP");
    case ENVSXP:      return mkString("ENVSXP");
    case PROMSXP:     return mkString("PROMSXP");
    case LANGSXP:     return mkString("LANGSXP");
    case SPECIALSXP:  return mkString("SPECIALSXP");
    case BUILTINSXP:  return mkString("BUILTINSXP");
    case CHARSXP:     return mkString("CHARSXP");
    case LGLSXP:      return mkString("LGLSXP");
    case INTSXP:      return mkString("INTSXP");
    case REALSXP:     return mkString("REALSXP");
    case CPLXSXP:     return mkString("CPLXSXP");
    case STRSXP:      return mkString("STRSXP");
    case DOTSXP:      return mkString("DOTSXP");
    case ANYSXP:      return mkString("ANYSXP");
    case VECSXP:      return mkString("VECSXP");
    case EXPRSXP:     return mkString("EXPRSXP");
    case BCODESXP:    return mkString("BCODESXP");
    case EXTPTRSXP:   return mkString("EXTPTRSXP");
    case WEAKREFSXP:  return mkString("WEAKREFSXP");
    case S4SXP:       return mkString("S4SXP");
    case RAWSXP:      return mkString("RAWSXP");
    default:          return mkString("<unknown>");
}')
```

```
source("src/sexp.R")
sexp_type(1:3)

## [1] "INTSXP"

sexp_type(10L)

## [1] "INTSXP"

sexp_type(TRUE)

## [1] "LGLSXP"

sexp_type(letters)

## [1] "STRSXP"

sexp_type(list(a = 1, b = letters))

## [1] "VECSXP"

sexp_type(ls)

## [1] "CLOSXP"
```

Garbage collection

Every R object that is created at the C level (not function arguments, that R is already aware of) must be PROTECTed to avoid being garbage collected. Before the return statement, these must be explicitely UNPROTECTed.

```
SEXP x;
PROTECT(x = ... )
## do stuff
UNPROTECT(1)
return(y)
```

Object creation

1. Allocate memory: `allocVector`, `allocMatrix`, `alloc3DArray`
2. Initialise objects: `memset`

```
SEXP x;
PROTECT(x = allocVector(INTSXP, 10) )
memset(INTEGER(x), 0, 10 * sizeof(int))
## do stuff
UNPROTECT(1)
return(y)
```

Accessing/setting SXP elements

- `REAL(x)[i]` if `x` is a `REALSXP`
- `INTEGER(x)[i]` if `x` is a `INTSXP`
- `LOGICAL(x)[i]` if `x` is a `LGLSXP`
- ...

- `STRING_ELT(x, i)` to access individual `CHARSXP` elements of a `STRSXP`
- `VECTOR_ELT(x, i)` to access individual `SXP` elements of a `VECSXP`

- `SET_STRING_ELT(out, 0, x))` to set an element in a string.
- `SET_VECTOR_ELT(out, 0, x))` to set an element in a list.

Example

We have a DNA sequence, represented by a string of A, C, G and T and we want to compute the GC content.

```
x <- "AGCATCGCACTACAT"
table(strsplit(x, "")[[1]])

##
## A C G T
## 5 5 2 3
```

1. `ingccount`: embedding the `C` directly in `R` using the *inline* package.
2. `gccount`: writing the `C` into its own code file and using `.Call`.

./src/ingccount.R

```
library("inline")

ingccount <- cfunction(
  sig = c(inseq = "character"),
  body = "
  int i, l;
  char p;
  SEXP ans, dnaseq;
  PROTECT(dnaseq = STRING_ELT(inseq, 0)); // a CHARSXP
  l = length(dnaseq);
  PROTECT(ans = allocVector(INTSXP, 4));
  memset(INTEGER(ans), 0, 4 * sizeof(int));
  for (i = 0; i < l; i++) {
    p = CHAR(dnaseq)[i];
    if (p == \'A\')
      INTEGER(ans)[0]++;
    else if (p == \'C\')
      INTEGER(ans)[1]++;
    else if (p == \'G\')
      INTEGER(ans)[2]++;
    else if (p == \'T\')
      INTEGER(ans)[3]++;
    else
      error(\"Wrong alphabet\");
  }
  UNPROTECT(2);
  return(ans);
")
```

```
source("./src/ingccount.R")
ingccount(x)

## [1] 5 5 2 3
```

text/gccount.c

```
#include <R.h>
#include <Rdefines.h>

SEXP gccount(SEXP inseq) {
  int i, l;
  char p;
  SEXP ans, dnaseq;

  PROTECT(dnaseq = STRING_ELT(inseq, 0)); // a CHARSXP
  l = length(dnaseq);

  PROTECT(ans = allocVector(INTSXP, 4));
  memset(INTEGER(ans), 0, 4 * sizeof(int));

  for (i = 0; i < l; i++) {
    p = CHAR(dnaseq)[i];
    if (p == 'A')
      INTEGER(ans)[0]++;
    else if (p == 'C')
      INTEGER(ans)[1]++;
    else if (p == 'G')
      INTEGER(ans)[2]++;
    else if (p == 'T')
      INTEGER(ans)[3]++;
    else
      error("Wrong alphabet");
  }
  UNPROTECT(2);
  return(ans);
}
```

Use directly

1. Create a shared library: `R CMD SHLIB gccount.c`
2. Load the shared object: `dyn.load("gccount.so")`
3. Create an `R` function that uses it:

```
gccount <-
  function(inseq) .Call("gccount", inseq)
```

4. Use you `C` code: `gccount("GACAGCATCA")`

In a package

- The `C` code comes in the `src` directory.
- The `R` wrapper will be

```
gccount <- function(inseq)
  .Call("gccount", inseq, PACKAGE = "mypackage")
```

- Document the `R` function
- Export the `R` function and `useDynLib(mypackge)` in the `NAMESPACE`

Using the *sequences* package

```
library(sequences)
s <- "GACTACGA"
gccount

## function (inseq)
## {
##     .Call("gccount", inseq, PACKAGE = "sequences")
## }
## <environment: namespace:sequences>

gccount(s)

## [1] 3 2 2 1

table(strsplit(s, ""))

##
## A C G T
## 3 2 2 1
```

We could check that

```
if (TYPEOF(inseq) != STRSXP)
  error("Need a character vector!");
```

although

```
.Call("gccount", 123)

## Error:  STRING_ELT() can only be applied to a
'character vector', not a 'double'
```

and type checking could easily be done at the R level. There is also `isRead(x)`, `isInteger(x)`, ... for atomics vectors.

There is of course much more to this . . . see references at the end.

Plan

Calling foreign languages

Build-in C interface

The Rcpp package

The *Rcpp* package

- Dirk Eddelbuettel and Romain Francois, with contributions by Douglas Bates, John Chambers and JJ Allaire
- `R` functions as well as a `C++` library which facilitate the integration of `R` and `C++`
- Also very well suited for `C`
- `http://www.rcpp.org/`

The *Rcpp* package

- Dirk Eddelbuettel and Romain Francois, with contributions by Douglas Bates, John Chambers and JJ Allaire
- `R` functions as well as a `C++` library which facilitate the integration of `R` and `C++`
- Also very well suited for `C`
- `http://www.rcpp.org/`

Associated packages

- *RcppArmadillo* – Armadillo templated `C++` library for linear algebra.
- *RcppEigen* – high-performance Eigen linear algebra library.
- *RInside* – use `R` from inside another `C++` by wrapping the existing `R` embedding API in an easy-to-use `C++` class.

Rcpp is a great package for writing both `C` and `C++` code:

- It comes with **loads** of documentation and examples.
- All basic `R` types are implemented as `C++` classes.
- No need to worry about garbage collection.

./src/ingccount2.R

```
library("Rcpp")

cppFunction(code = "
NumericVector gccount2(CharacterVector inseq) {
  Rcpp::CharacterVector dnaseq(inseq);
  Rcpp::NumericVector ans(4);
  std::string s = Rcpp::as<std::string>(dnaseq);

  for (int i = 0; i < s.size(); i++) {
    char p = s[i];
    if (p == \'A\')
      ans[0]++;
    else if (p == \'C\')
      ans[1]++;
    else if (p == \'G\')
      ans[2]++;
    else if (p == \'T\')
      ans[3]++;
    else
      Rf_error(\"Wrong alphabet\");
  }
  return(ans);
}
")
```

Using in a package

1. You will need a `Makevars` file in the `src` directory
2. Modify `DESCRIPTION` file:
   ```
   Depends:  Rcpp
   LinkingTo:  Rcpp
   ```
3. Create an `R` function that uses it

```
gccount2 <- function(inseq)
  .Call("gccount2", inseq, PACKAGE = "mypackage")
```

4. Document the `R` function
5. Export the `R` function and `useDynLib(mypackge)` in the `NAMESPACE`

See package `sequences` for a working example.

Further reading

- Writing R Extensions, R Core team.
- *Rcpp* documentation.
- Dirk Eddelbuettel, Seamless `R` and `C++` Integration with *Rcpp*, Springer, 2013.
- Dirk Eddelbuettel and Romain Francois, *Rcpp: Seamless R and C++ Integration*, Journal of Statistical Software, Vol. 40, Issue 8, Apr 2011, `http://www.jstatsoft.org/v40/i08/`.
- Relevant devtools sections: *C interface* and *Rcpp*.

- This work is licensed under a CC BY-SA 3.0 License.
- Course (and more) web page: https://github.com/lgatto/TeachingMaterial

Thank you for you attention