Using C and C++ with R

Laurent Gatto lg390@cam.ac.uk

University of Cambridge

May 18, 2013

Plan

Plan

Calling foreign languages

Built-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/

Robert Gentleman, in *R Programming for Bioinformatics*, 2008, about `R`'s built-in `C` interfaces

> *Since R is not compiled, in some situations its performance can be substantially improved by writing code in a compiled language. There are also reasons not to write code in other languages, and in particular we caution against premature optimization, prototyping in R is often cost effective. And in our experience very few routines need to be implemented in other languages for efficiency reasons. Another substantial reason not to use an implementation in some other language is increased complexity. The use of another language almost always results in higher maintenance costs and less stability. In addition, any extensions or enhancements of the code will require someone that is proficient in both R and the other language.*

`Rcpp` does make some of the above caution statements slightly less critical.

- **Why?** `R` is getting slow or is not doing well in terms of memory management: for example `for` loops that can't be vectorised, recursion, ...
- **When?** `R` can't do better **and** the slow code has been identified $\rightarrow$ `Rprof`

- **Why?** `R` is getting slow or is not doing well in terms of memory management: for example `for` loops that can't be vectorised, recursion, ...
- **When?** `R` can't do better **and** the slow code has been identified $\rightarrow$ `Rprof`

- **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

We will be using the following packages:

- `inline` and the `cfunction` to write inline `C` code that is compiled on the fly. (There is also a `cxxfunction` for `C++` code).
- `Rcpp`, illustrating some of its functionality as well as the `cppFunction` for inline/on the fly compilation of `C++` code.

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 <- "ACCGGGTTTT"
gccountr <- function(x) table(strsplit(x, "")[1])
gccountr(x)

##
## A C G T
## 1 2 3 4
```

Plan

Calling foreign languages

Built-in `C` interface

The `Rcpp` package

The R C API

- Very frequent in R but has 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.

`.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 as `SEXP`.
- Manual memory management: allocate memory, protect objects to avoid them being garbage collected and subsequently unprotect them.

S (or symbolic) 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"
```

```r
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>");
}')
```

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

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

```r
sexp_type(10L)
```

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

```r
sexp_type(TRUE)
```

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

```r
sexp_type(letters)
```

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

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

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

```r
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(x)
```

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(x)
```

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 elements of a `VECSXP`

- `SET_STRING_ELT(str, i, x))` to set an element in a string.
- `SET_VECTOR_ELT(vec, i, x))` to set an element in a list.

```c
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);
}
```

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);
")
```

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

## [1] 1 2 3 4
```

src/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:

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

4. Use you `C` code:

```
gccountC(x)

## [1] 1 2 3 4
```

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

```r
library(sequences)
gccount

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

gccount(x)

## [1] 1 2 3 4
```

We could check that

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

although

```
gccountC(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 `isReal(x)`, `isInteger(x)`, ... for atomics vectors.

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

Benchmarking

```
library(microbenchmark)
microbenchmark(gccountr(x),
               ingccount(x),
               gccountC(x),
               times = 1e4)
```

```
## Unit: microseconds
##          expr     min      lq  median      uq     max neval
##   gccountr(x) 228.032 234.912 237.391 242.071 6688.42 10000
##  ingccount(x)   1.048   1.747   2.376   3.353   65.93 10000
##   gccountC(x)   3.074   3.772   5.798   7.264  108.11 10000
```

Could we do better in `R` ? (should be asked first, really)

Could we do better in R ? (should be asked first, really)

```r
gccountr2 <-
  function(x) tabulate(factor(strsplit(x, "")[[1]]))
```

Could we do better in R ? (should be asked first, really)

```
gccountr2 <-
  function(x) tabulate(factor(strsplit(x, "")[[1]]))
```

```
microbenchmark(gccountr(x),
               gccountr2(x),
               ingccount(x),
               gccountC(x),
               times = 1e4)
```

```
## Unit: microseconds
##          expr     min      lq  median      uq     max neval
##   gccountr(x) 228.102 233.898 236.902 242.978 4958.52 10000
##  gccountr2(x) 118.101 122.920 124.737 127.810 1978.74 10000
##  ingccount(x)   1.187   1.956   3.073   3.422 1621.44 10000
##   gccountC(x)   3.143   4.121   6.705   7.473   68.38 10000
```

Plan

Calling foreign languages

Built-in `C` interface

The `Rcpp` package

The Rcpp package

- Dirk Eddelbuettel and Romain Francois, with contributions by Douglas Bates, John Chambers and JJ Allaire.
- A flexible framework that facilitates the integration of R and C/C++ .
- http://www.rcpp.org/

- It comes with **loads** of documentation and examples: `vignette(package = "Rcpp")`.
- All basic R types are implemented as C++ classes.
- No need to worry about garbage collection.

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.

C++ classes

Scalar	Vector	Matrix
`double`	`NumericVector`	`NumericMatrix`
`int`	`IntegerVector`	`IntegerMatrix`
`string`	`CharacterVector`	`CharacterMatrix`
`bool`	`LogicalVector`	`LogicalMatrix`

And `Function`, `List`, `DataFrame` ...
Automatic conversions from `R` (`C`) to `C++` (`R`) using `as` (`wrap`).

1. `ingccount2`: embedding the `C++` directly in `R` using the `Rcpp::cppFunction` package.
2. `gccount2`: in a package, writing the `C++` into its own code file and using `.Call`.
3. `gccountX`: using `sourceCpp` to source the `C++` file and export the function to `R` .

./src/ingccount2.R

```
IntegerVector ingccount2(CharacterVector inseq) {
  IntegerVector ans(4);
  std::string s = Rcpp::as<std::string>(inseq[0]);
  int n = inseq(0).size();
  for (int i = 0; i < n; i++) {
    if (s[i] == 'A')
      ans[0]++;
    else if (s[i] == 'C')
      ans[1]++;
    else if (s[i] == 'G')
      ans[2]++;
    else if (s[i] == 'T')
      ans[3]++;
    else
      Rf_error(\"Wrong alphabet\");
  }
  return(ans);
}
```

./src/ingccount2.R

```
library("Rcpp")

cppFunction("
IntegerVector ingccount2(CharacterVector inseq) {
  IntegerVector ans(4);
  std::string s = Rcpp::as<std::string>(inseq[0]);
  int n = inseq(0).size();
  for (int i = 0; i < n; i++) {
    if (s[i] == 'A')
      ans[0]++;
    else if (s[i] == 'C')
      ans[1]++;
    else if (s[i] == 'G')
      ans[2]++;
    else if (s[i] == 'T')
      ans[3]++;
    else
      Rf_error(\"Wrong alphabet\");
  }
  return(ans);
}
")
```

```
x <- "ACCGGGTTTT"
source("src/ingccount2.R")
ingccount2(x)

## [1] 1 2 3 4
```

./src/gccount2.cpp

```cpp
#include <Rcpp.h>

using namespace Rcpp;

RcppExport SEXP gccount2(SEXP inseq)
{
  Rcpp::IntegerVector ans(4);
  Rcpp::CharacterVector dnaseq(inseq);
  std::string s = Rcpp::as<std::string>(dnaseq[0]);

  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);
}
```

In a package

1. You will need a `Makevars` file in the `src` directory.
2. Modify `DESCRIPTION` file:
   ```
   Depends:  Rcpp (>= 0.10.1)
   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`

Check the `sequences` package. But see `Rcpp.package.skeleton` below.

Using sourceCpp

- Write the C++ code into a cpp file, including headers and dedicated export statement (see next slide).
- Source it and use the R function.

```
sourceCpp("src/gccountX.cpp")
gccountX(x)
```

```
## [1] 1 2 3 4
```

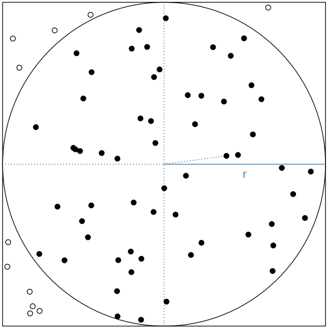
./src/gccountX.R

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector gccountX(CharacterVector inseq) {
  IntegerVector ans(4);
  std::string s = Rcpp::as<std::string>(inseq[0]);

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

sugar

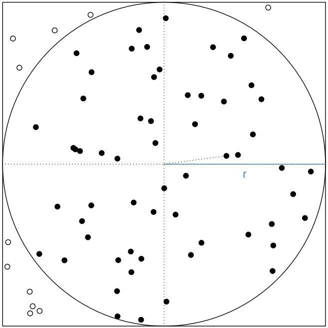
sugar (for *syntactic sugar*) is a set of `C++` functions that (mostly) work and look like their `R` couterparts. Allows for example compact vectorised expression. Looks like `R` with the `C++` efficiency. (Rcpp-sugar vignette/paper)

- Vectorised arithmetic and logical operators: `+`, `>`, `!`, ...
- Functions: `seq_len`, `all`, `any`, `sapply`, `rnorm`, `abs`, ...



$$\frac{d}{n} = \frac{\pi r^2}{4r^2} = \pi/4$$

$$\pi \approx \frac{4d}{n}$$



$$\frac{d}{n} = \frac{\pi r^2}{4r^2} = \pi/4$$
$$\pi \approx \frac{4d}{n}$$

```
piR <- function(N) {
    x <- runif(N)
    y <- runif(N)
    d <- sqrt(x^2 + y^2)
    4 * sum(d < 1.0) / N
}
```

```r
piR <- function(N) {
    x <- runif(N)
    y <- runif(N)
    d <- sqrt(x^2 + y^2)
    return(4 * sum(d < 1.0) / N)
}
```

```r
piR <- function(N) {
    x <- runif(N)
    y <- runif(N)
    d <- sqrt(x^2 + y^2)
    return(4 * sum(d < 1.0) / N)
}
```

```cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double piSugar(const int N) {
    RNGScope scope; // ensure RNG gets set/reset
    NumericVector x = runif(N);
    NumericVector y = runif(N);
    NumericVector d = sqrt(x*x + y*y);
    return 4.0 * sum(d < 1.0) / N;
}
```

src/sugar.R

```r
source("src/pi.R")

library("Rcpp")
sourceCpp("src/pi.cpp")

N <- 1e+06
set.seed(42)
resR <- piR(N)

set.seed(42)
resCpp <- piSugar(N)
stopifnot(identical(resR, resCpp))
```

```r
library(rbenchmark)
res <- benchmark(piR(N), piSugar(N),
                 order="relative")
print(res[,1:4])

##          test replications elapsed relative
## 2 piSugar(N)          100    5.41    1.000
## 1     piR(N)          100   19.50    3.604
```

Exercise 1: Translate these R functions into C or C++

```
pdistR <- function(x, ys) sqrt((x - ys)^2)
```

Rcpp modules

Using S4 Reference Classes to reflect `C++` classes and methods (see the Rcpp-modules vignette).

Interface to the Standard template library

`C++` library with data structure and algorithms: vectors, arrays, stacks, iterators, accumulators, . . .
add `link`.

Using Rcpp in packages

```r
Rcpp.package.skeleton("mypackage")
```

```
mypackage
|-- DESCRIPTION
|-- man
|   |-- mypackage-package.Rd
|    -- rcpp_hello_world.Rd
|-- NAMESPACE
|-- R
|    -- rcpp_hello_world.R
|-- Read-and-delete-me
 -- src
    |-- Makevars
    |-- Makevars.win
    |-- rcpp_hello_world.cpp
     -- rcpp_hello_world.h
```

```r
Rcpp.package.skeleton("mypackage", attribute = TRUE)
Rcpp.package.skeleton("mypackage", module = TRUE)
Rcpp.package.skeleton("mypackage",
                      cpp_files = c("convolve.cpp"))
```

Further reading

- Writing R Extensions, R Core team.
- Robert Gentleman, `R` Programming for Bioinformatics, 2008.
- `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*.

- Course web page and more material: `https://github.com/lgatto/TeachingMaterial`

Thank you for you attention