Native Interfaces for R

Seth Falcon

Fred Hutchinson Cancer Research Center

20-21 May, 2010

Outline

The .Call Native Interface

R types at the C-level

Memory management in R

Self-Study Exercises

Odds and Ends

Resources

`which` as implemented in R prior to R-2.11

```
which <- function(x) {
    seq_along(x)[x & !is.na(x)]
}
```

`which` as implemented in R prior to R-2.11

```
which <- function(x) {
    seq_along(x)[x & !is.na(x)]
}
```

1. `is.na`
2. `!`
3. `&`
4. `seq_along`
5. `[`

`which` in C

```
 1 SEXP nid_which(SEXP v) {
 2     SEXP ans;
 3     int i, j = 0, len = Rf_length(v), *buf, *tf;
 4     buf = (int *) R_alloc(len, sizeof(int));
 5     tf = LOGICAL(v);
 6     for (i = 0; i < len; i++) {
 7         if (tf[i] == TRUE) buf[j] = i + 1; j++;
 8     }
 9     ans = Rf_allocVector(INTSXP, j);
10     memcpy(INTEGER(ans), buf, sizeof(int) * j);
11     return ans;
}
```

`which` is now 10x faster

```
system.time(which(v))
```

R Version	System time	Elapsed time
R-2.10	0.018	0.485
R-2.11	0.001	0.052

v is a logical vector with 10 million elements

Making use of existing code

Access algorithms

- RBGL
- RCurl
- Rsamtools

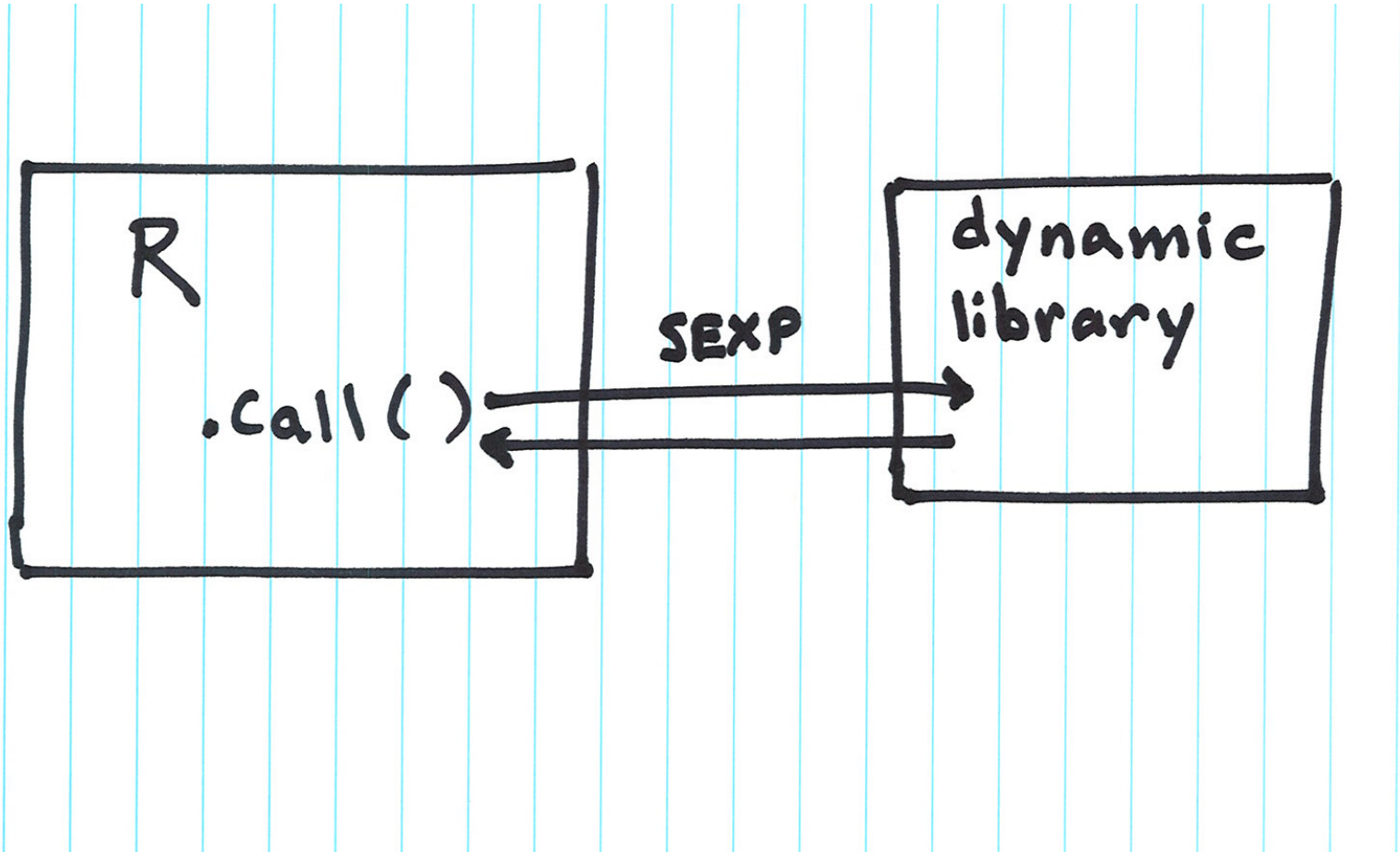
Interface to other systems

- RSQLite
- netcdf
- SJava

Review of C

```
#include <stdio.h>
double _nid_avg(int *data, int len) {
    int i;
    double ans = 0.0;
    for (i = 0; i < len; i++) {
      ans += data[i];
    }
    return ans / len;
}
main() {
    int ex_data[5] = {1, 2, 3, 4, 5};
    double m = _nid_avg(ex_data, 5);
    printf("%f\n", m);
}
```

The .Call interface



Symbolic Expressions (SEXPs)

- The SEXP is R's fundamental data type at the C-level
- SEXP is short for *symbolic expression* and is borrowed from Lisp
- History at `http://en.wikipedia.org/wiki/S-expression`

.Call Start to Finish Demo

Demo

.Call Disection

R

```
# R/somefile.R
avg <- function(x) .Call(.nid_avg, x)
# NAMESPACE
useDynLib("yourPackage", .nid_avg = nid_avg)
```

C

```
/* src/some.c */
#include <Rinternals.h>
SEXP nid_avg(SEXP data) { ... INTEGER(data) ... }
```

.Call C function

```c
#include <Rinternals.h>

SEXP nid_avg(SEXP data)
{
    SEXP ans;
    double v = _nid_avg(INTEGER(data), Rf_length(data));
    PROTECT(ans = Rf_allocVector(REALSXP, 1));
    REAL(ans)[0] = v;
    UNPROTECT(1);
    return ans;
    /* shortcut: return Rf_ScalarReal(v); */
}
```

.Call C function

```
#include <Rinternals.h>

SEXP nid_avg(SEXP data)
{
    SEXP ans;
    double v = _nid_avg(INTEGER(data), Rf_length(data));
    PROTECT(ans = Rf_allocVector(REALSXP, 1));
    REAL(ans)[0] = v;
    UNPROTECT(1);
    return ans;
    /* shortcut: return Rf_ScalarReal(v); */
}
```

.Call C function

```
#include <Rinternals.h>

SEXP nid_avg(SEXP data)
{
    SEXP ans;
    double v = _nid_avg(INTEGER(data), Rf_length(data));
    PROTECT(ans = Rf_allocVector(REALSXP, 1));
    REAL(ans)[0] = v;
    UNPROTECT(1);
    return ans;
    /* shortcut: return Rf_ScalarReal(v); */
}
```

.Call C function

```
#include <Rinternals.h>

SEXP nid_avg(SEXP data)
{
    SEXP ans;
    double v = _nid_avg(INTEGER(data), Rf_length(data));
    PROTECT(ans = Rf_allocVector(REALSXP, 1));
    REAL(ans)[0] = v;
    UNPROTECT(1);
    return ans;
    /* shortcut: return Rf_ScalarReal(v); */
}
```

.Call C function

```
#include <Rinternals.h>

SEXP nid_avg(SEXP data)
{
    SEXP ans;
    double v = _nid_avg(INTEGER(data), Rf_length(data));
    PROTECT(ans = Rf_allocVector(REALSXP, 1));
    REAL(ans)[0] = v;
    UNPROTECT(1);
    return ans;
    /* shortcut: return Rf_ScalarReal(v); */
}
```

.Call C function

```
#include <Rinternals.h>

SEXP nid_avg(SEXP data)
{
    SEXP ans;
    double v = _nid_avg(INTEGER(data), Rf_length(data));
    PROTECT(ans = Rf_allocVector(REALSXP, 1));
    REAL(ans)[0] = v;
    UNPROTECT(1);
    return ans;
    /* shortcut: return Rf_ScalarReal(v); */
}
```

.Call C function

```
#include <Rinternals.h>

SEXP nid_avg(SEXP data)
{
    SEXP ans;
    double v = _nid_avg(INTEGER(data), Rf_length(data));
    PROTECT(ans = Rf_allocVector(REALSXP, 1));
    REAL(ans)[0] = v;
    UNPROTECT(1);
    return ans;
    /* shortcut: return Rf_ScalarReal(v); */
}
```

Function Registration

- In NAMESPACE
- In C via `R_CallMethodDef` (See `ShortRead/src/R_init_ShortRead.c` for a nice example.

Outline

The .Call Native Interface

R types at the C-level

Memory management in R

Self-Study Exercises

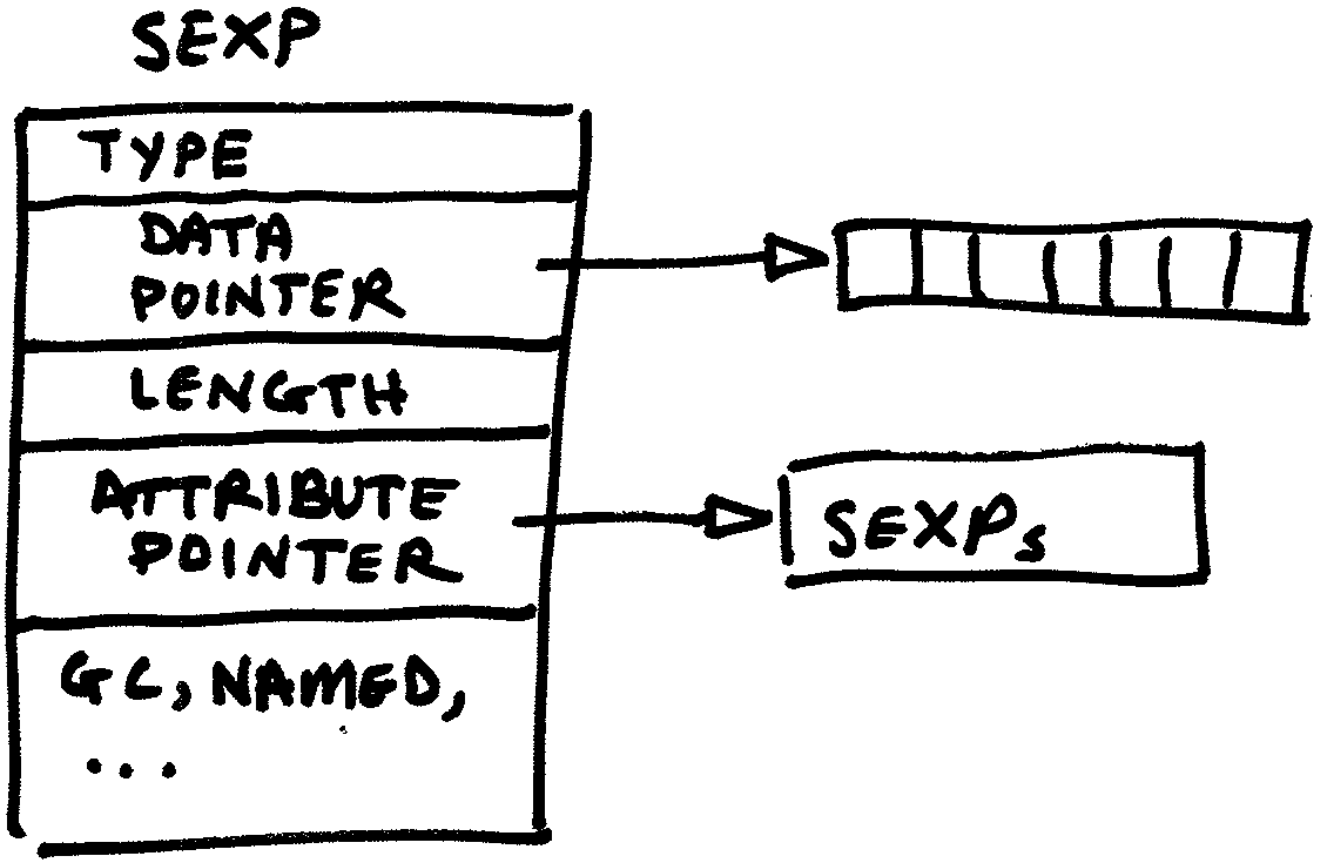
Odds and Ends

Resources

Just one thing to learn

- Everything is a SEXP
- But there are many different types of SEXPs
- INTSXP, REALSXP, STRSXP, LGLSXP, VECSXP, and more.

What's a SEXP?


SEXP
TYPE
DATA POINTER
LENGTH
ATTRIBUTE POINTER
SEXPs
GC, NAMED,
...

Common SEXP subtypes

R function	SEXP subtype	Data accessor
integer()	INTSXP	int *INTEGER(x)
numeric()	REALSXP	double *REAL(x)
logical()	LGLSXP	int *LOGICAL(x)
character()	STRSXP	CHARSXP STRING_ELT(x, i)
list()	VECSXP	SEXP VECTOR_ELT(x, i)
NULL	NILSXP	R_NilValue
externalptr	EXTPTRSXP	SEXP (accessor funcs)

STRSXP/CHARSXP Exercise

Try this:

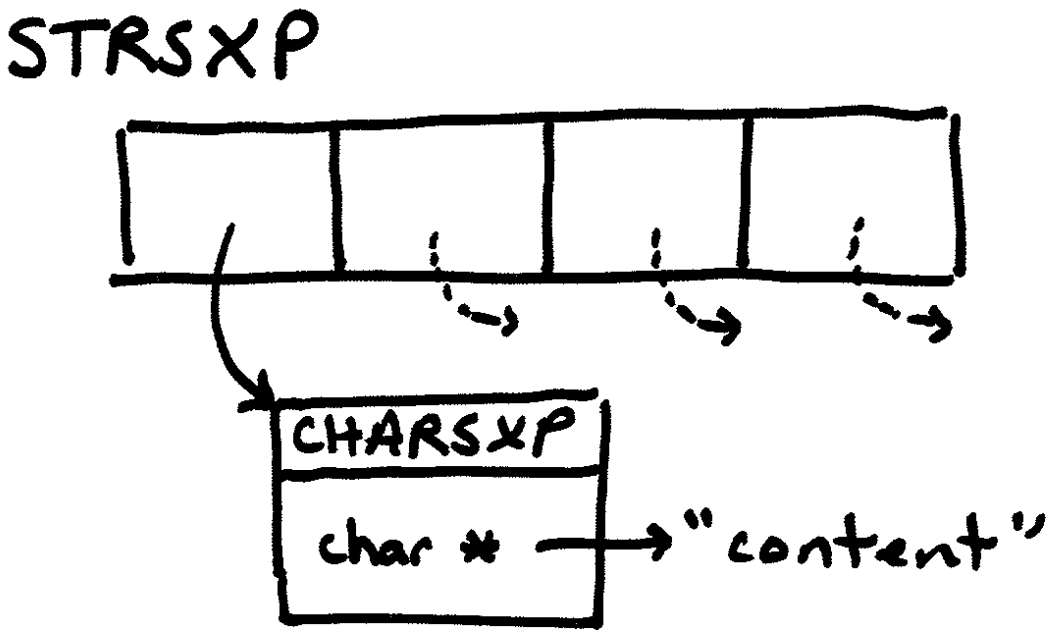
```
.Internal(inspect(c("abc", "abc", "xyz")))
```

STRSXP/CHARSXP Exercise

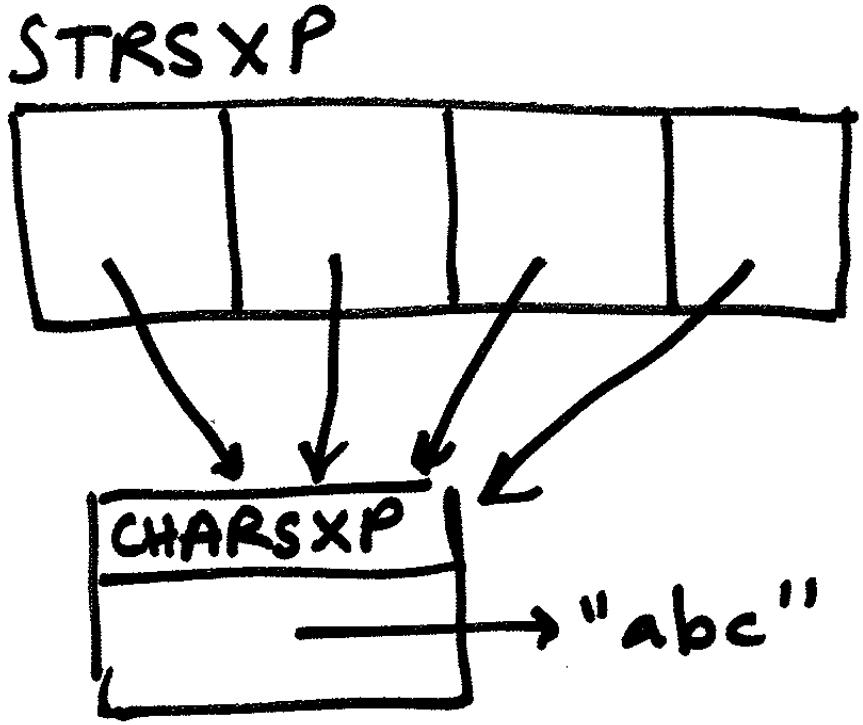
```
> .Internal(inspect(c("abc", "abc", "xyz")))

 @101d3f280 16 STRSXP g0c3 [] (len=3, tl=1)
   @101cc6278 09 CHARSXP g0c1 [gp=0x20] "abc"
   @101cc6278 09 CHARSXP g0c1 [gp=0x20] "abc"
   @101cc6308 09 CHARSXP g0c1 [gp=0x20] "xyz"
```

character vectors == STRSXPs + CHARSXPs



CHARSXPs are special



STRSXP/CHARSXP API Example

```
SEXP s = Rf_allocVector(STRSXP, 5);
PROTECT(s);
SET_STRING_ELT(s, 0, mkChar("hello"));
SET_STRING_ELT(s, 1, mkChar("goodbye"));
SEXP c = STRING_ELT(s, 0);
const char *v = CHAR(c);
UNPROTECT(1);
```

STRSXP/CHARSXP API Example

```
SEXP s = Rf_allocVector(STRSXP, 5);
PROTECT(s);
SET_STRING_ELT(s, 0, mkChar("hello"));
SET_STRING_ELT(s, 1, mkChar("goodbye"));
SEXP c = STRING_ELT(s, 0);
const char *v = CHAR(c);
UNPROTECT(1);
```

STRSXP/CHARSXP API Example

```
SEXP s = Rf_allocVector(STRSXP, 5);
PROTECT(s);
SET_STRING_ELT(s, 0, mkChar("hello"));
SET_STRING_ELT(s, 1, mkChar("goodbye"));
SEXP c = STRING_ELT(s, 0);
const char *v = CHAR(c);
UNPROTECT(1);
```

STRSXP/CHARSXP API Example

```
SEXP s = Rf_allocVector(STRSXP, 5);
PROTECT(s);
SET_STRING_ELT(s, 0, mkChar("hello"));
SET_STRING_ELT(s, 1, mkChar("goodbye"));
SEXP c = STRING_ELT(s, 0);
const char *v = CHAR(c);
UNPROTECT(1);
```

STRSXP/CHARSXP API Example

```
SEXP s = Rf_allocVector(STRSXP, 5);
PROTECT(s);
SET_STRING_ELT(s, 0, mkChar("hello"));
SET_STRING_ELT(s, 1, mkChar("goodbye"));
SEXP c = STRING_ELT(s, 0);
const char *v = CHAR(c);
UNPROTECT(1);
```

Outline

The .Call Native Interface

R types at the C-level

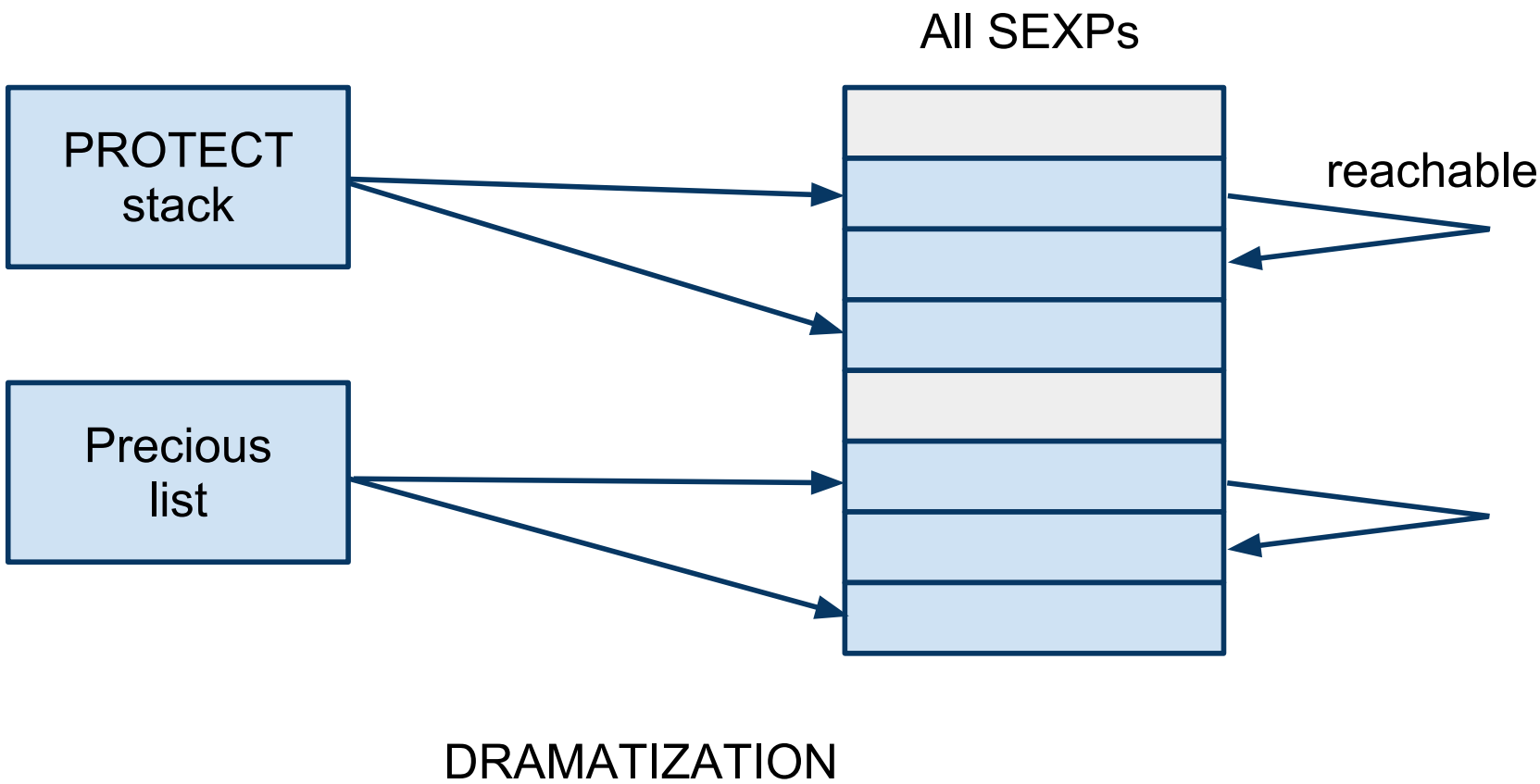
Memory management in R

Self-Study Exercises

Odds and Ends

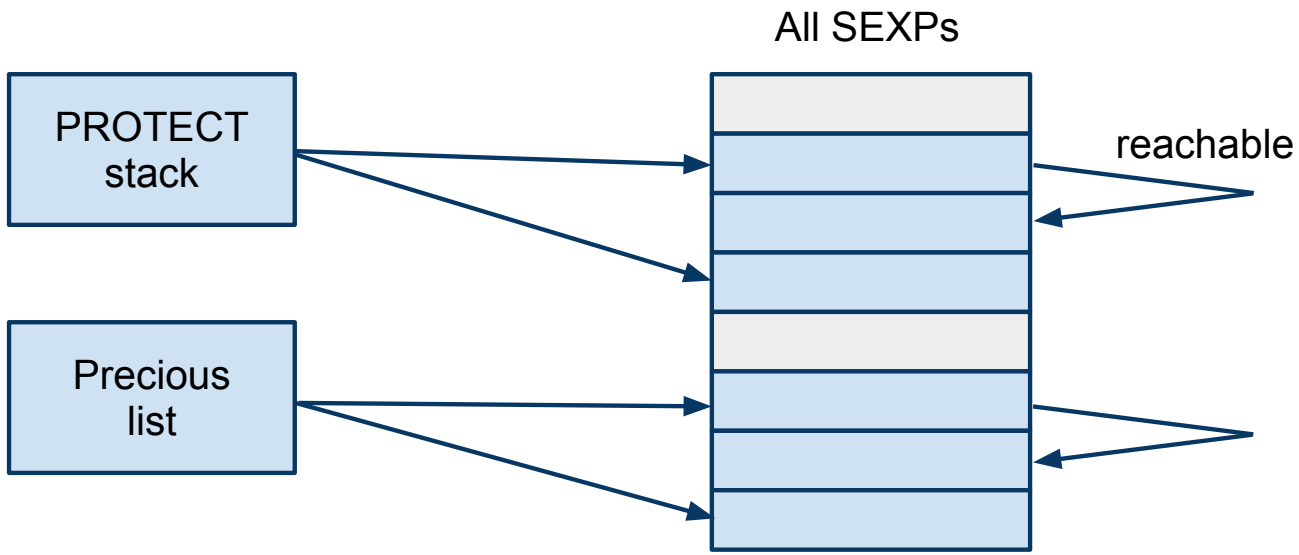
Resources

R's memory model (dramatization)



R's memory model

- R allocates, tracks, and garbage collects SEXPs
- gc is triggered by R functions
- SEXPs that are not *in-use* are recycled.
- A SEXP is *in-use* if:
  - it is on the protection stack (PROTECT/UNPROTECT)
  - it is in the precious list (R_PreserveObject/R_ReleaseObject)
  - it is *reachable* from a SEXP in the in-use list.



DRAMATIZATION

Preventing gc of SEXPs with the protection stack

`PROTECT(s)`	Push `s` onto the protection stack
`UNPROTECT(n)`	Pop top `n` items off of the protection stack

Generic memory: When everything *isn't* a SEXP

Allocation Function	Lifecycle
`R_alloc`	Memory is freed when returning from .Call
`Calloc`/`Free`	Memory persists until Free is called

Case Study: C implementation of `which`

Code review of `nidemo/src/which.c`

Outline

The .Call Native Interface

R types at the C-level

Memory management in R

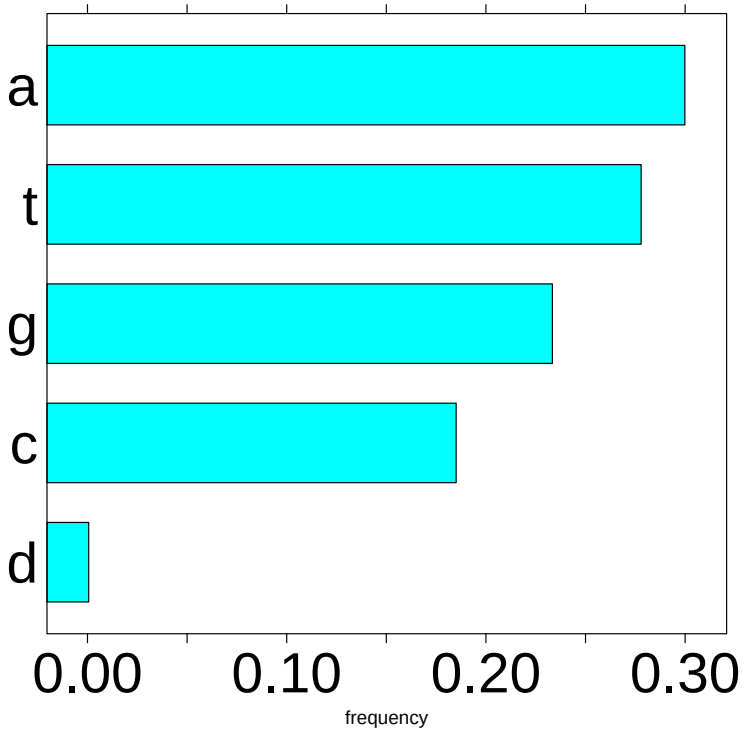
Self-Study Exercises

Odds and Ends

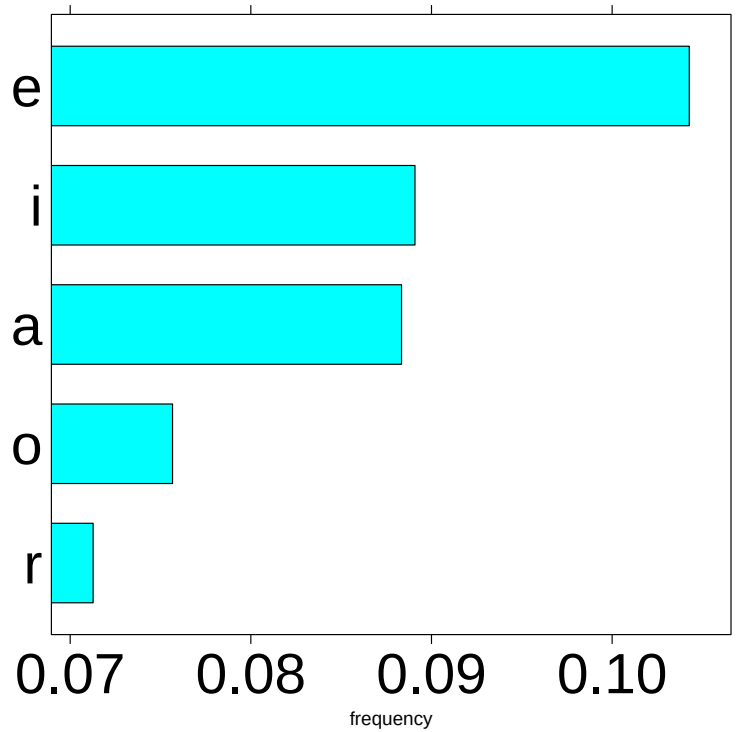
Resources

Alphabet frequency of a text file

Yeast Gene YDL143W



English Dictionary



Self-Study in nidemo package

1. Alphabet Frequency
   - Explore R and C implementations
   - Make C implementation robust
   - Attach names attribute in C
   - multiple file, matrix return enhancement
2. External pointer example
3. Calling R from C example and enhancement
4. Debugging: a video how-to

Outline

The .Call Native Interface

R types at the C-level

Memory management in R

Self-Study Exercises

Odds and Ends

Resources

Odds and Ends

1. Debugging
2. Public API
3. Calling R from C
4. Making sense of the remapped function names
5. Finding C implementations of R functions
6. Using a TAGS file

Debugging Native Code

```
Rprintf("DEBUG: v => %d, x => %s\n", v, x);
```

```
$ R -d gdb
(gdb) run
> # now R is running
```

> *There are two ways to write error-free programs; only the third one works.*
> *– Alan Perlis, Epigrams on Programming*

R's Public API

```
mkdir R-devel-build; cd R-devel-build
~/src/R-devel-src/configure && make
## hint: configure --help

ls include

  R.h      Rdefines.h    Rinterface.h  S.h
  R_ext/   Rembedded.h   Rinternals.h
```

Details in WRE

Calling R functions from C

1. Build a pairlist
2. call `Rf_eval(s, rho)`

Remapped functions

- Source is unadorned
- Preprocessor adds `Rf_`
- Object code contains `Rf_`

Finding C implementations

```
cd R-devel/src/main

grep '"any"' names.c
{"any", do_logic3, 2, ...}

grep -l do_logic3 *.c
logic.c
```

R CMD rtags

```
cd R-devel/src
R CMD rtags --no-Rd
# creates index file TAGS
```

Outline

The .Call Native Interface

R types at the C-level

Memory management in R

Self-Study Exercises

Odds and Ends

Resources

Writing R Extensions

```
RShowDoc("R-exts")
```

Book: The C Programming Language (K & R)

Java in a Nutshell	1264 pages
The C++ Programming Language	1030 pages
The C Programming Language	274 pages

Resources

- **Read WRE and "K & R"**
- Use the sources for R and packages
- R-devel, bioc-devel mailing lists
- Patience