R Programming

October 2, 2013

References:

- Adler
- Chambers
- Murrell, Introduction to Data Technologies.
- R intro manual (R-intro) and R language manual (R-lang), both on CRAN.
- Venables and Ripley, Modern Applied Statistics with S

1 Efficiency

In general, make use of R's built-in functions, as these tend to be implemented internally (i.e., via compiled code in C or Fortran). In particular, if R is linked to optimized BLAS and Lapack code (e.g. Intel's *MKL*, *OpenBLAS* [on the SCF Linux servers], AMD's *ACML* [on the SCF Linux cluster, *vecLib* for Macs [on the SCF Macs]), you should have good performance (potentially comparable to Matlab and to coding in C). Often you can figure out a trick to take your problem and transform it to make use of the built-in functions.

Note that I run a lot of MCMCs so I pay attention to making sure my calculations are fast as they are done repeatedly. Similarly, one would want to pay attention to speed when doing large simulations and bootstrapping, and in some cases for optimization. And if you're distributing code, it's good to have it be efficient. But in other contexts, it may not be worth your time. Also, it's good practice to code it transparently first to reduce bugs and then to use tricks to speed it up and make sure the fast version works correctly.

Results can vary with with your system setup and version of R, so the best thing to do is figure out where the bottlenecks are in your code (e.g., with *Rprof()* or just some basic use of *system.time()* and *benchmark()*), and then play around with alternative specifications. And as you

gain more experience, you'll get some intuition for what approaches might improve speed, but even with experience I find myself often surprised by what matters and what doesn't. It's often worth trying out a bunch of different ideas; *system.time()* and *benchmark()* are your workhorse tools in this context.

First, let's see some tools for assessing the speed of your code.

1.1 Tools for assessing efficiency

1.1.1 Benchmarking

system.time() is very handy for comparing the speed of different implementations.

```
n <- 1000
x <- matrix(rnorm(n^2), n)
system.time({
    mns <- rep(NA, n)
    for (i in 1:n) mns[i] <- mean(x[i, ])
})

##    user  system elapsed
##   0.028   0.000   0.025

system.time(rowMeans(x))

##    user  system elapsed
##   0.000   0.000   0.003
```

The *rbenchmark* package provides a nice wrapper function, *benchmark()*, that automates speed assessments.

```
library(rbenchmark)
# speed of one calculation
n <- 1000
x <- matrix(rnorm(n^2), n)
benchmark(crossprod(x), replications = 10, columns = c("test", "elapsed", "

##           test elapsed replications
## 1 crossprod(x)   0.314           10
```

```
# comparing different approaches to a task
benchmark({
    mns <- rep(NA, n)
    for (i in 1:n) mns[i] <- mean(x[i, ])
}, rowMeans(x), replications = 10, columns = c("test", "elapsed", "replicat

##                                                                      test
## 1 {\n    mns <- rep(NA, n)\n    for (i in 1:n) mns[i] <- mean(x[i, ])\n}
## 2                                                              rowMeans(x)
##   elapsed replications
## 1   0.221           10
## 2   0.025           10
```

1.1.2 Profiling

The *Rprof()* function will show you how much time is spent in different functions, which can help you pinpoint bottlenecks in your code.

Here's a function that works with a correlation matrix such as one might have for time series data.

```
makeTS <- function(param, len) {
    times <- seq(0, 1, length = len)
    dd <- rdist(times)
    C <- exp(-dd/param)
    U <- chol(C)
    white <- rnorm(len)
    return(crossprod(U, white))
}
```

```
library(fields)
Rprof("makeTS.prof")
out <- makeTS(0.1, 1000)
Rprof(NULL)
summaryRprof("makeTS.prof")
```

Here's the result for the *makeTS()* function from the demo code file:

```
$by.self   self.time self.pct total.time total.pct
".Call"         0.38    48.72       0.38     48.72
".Fortran"      0.22    28.21       0.22     28.21
"matrix"        0.08    10.26       0.30     38.46
"exp"           0.08    10.26       0.08     10.26
"/"             0.02     2.56       0.02      2.56
$by.total         total.time total.pct self.time self.pct
"makeTS"                0.78    100.00      0.00     0.00
".Call"                 0.38     48.72      0.38    48.72
"chol.default"          0.38     48.72      0.00     0.00
"chol"                  0.38     48.72      0.00     0.00
"standardGeneric"       0.38     48.72      0.00     0.00
"matrix"                0.30     38.46      0.08    10.26
"rdist"                 0.30     38.46      0.00     0.00
".Fortran"              0.22     28.21      0.22    28.21
"exp"                   0.08     10.26      0.08    10.26
"/"                     0.02      2.56      0.02     2.56
$sample.interval [1] 0.02
$sampling.time [1] 0.78
```

Rprof() tells how much time was spent in each function alone (the *"self"* columns) and aggregating the time spent in a function and all of the functions that it calls (the *"total"* columns). Usually the former is going to be more useful, but in some cases we need to decipher what is going on based on the latter.

Let's figure out what is going on here. The self time tells us that *.Call* (a call to C code), *.Fortran* (a call to Fortran code) and *matrix()* take up most of the time. Looking at the total time and seeing in *chol.default()* that *.Call* is used (you would have to go in and look at the *La_chol()* function to figure this out) and in *rdist()* that *.Fortran()* and *matrix()* are used we can infer that about 49% of the time is being spent in the Cholesky and 38% in the *rdist()* calculation, with 10% in *exp()*. As we increase the number of time points, the time taken up by the Cholesky would increase since that calculation is order of n^3 while the others are order n^2 (more in the linear algebra unit).

Apparently there is a memory profiler in R, *Rprofmem()*, but it needs to be enabled when R is compiled (i.e., installed on the machine), because it slows R down even when not used. So I've never gotten to the point of playing around with it.

Warning: *Rprof()* conflicts with threaded linear algebra, so you will need to set OMP_NUM_THREADS to 1 to disable threaded linear algebra if you profile code that involves linear algebra. More about this in the unit on parallel processing.

1.2 Strategies for improving efficiency

1.2.1 Fast initialization

It is very inefficient to iteratively add elements to a vector, matrix, data frame, array or list (e.g., iteratively using *c()*, *cbind()*, *rbind()*, etc.). Instead, create the full object in advance (this is equivalent to variable initialization in compiled languages) and then fill in the appropriate elements. The reason is that when R appends to an existing object, it creates a new copy and as the object gets big, this gets slow when one does it a lot of times. Here's an illustrative example, but of course we would not fill a vector like this because we would in practice use vectorized calculations.

```
n <- 1000
x <- 1
benchmark(for (i in 2:n) x <- c(x, i), {
    x <- rep(as.numeric(NA), n)
    for (i in 1:n) x[i] <- i
}, replications = 10, columns = c("test", "elapsed", "replications"))
```

```
##                                                                  test
## 1                                          for (i in 2:n) x <- c(x, i)
## 2 {\n    x <- rep(as.numeric(NA), n)\n    for (i in 1:n) x[i] <- i\n}
##   elapsed replications
## 1   0.129           10
## 2   0.014           10
```

It's not necessary to use *as.numeric()* above though it saves a bit of time. **Challenge**: figure out why I have `as.numeric(NA)` and not just `NA`.

We can actually speed up the initialization (though in most practical circumstances, the second approach here would be overkill):

```
n <- 1e+06
benchmark(x <- rep(as.numeric(NA), n), {
    x <- as.numeric(NA)
    length(x) <- n
}, replications = 10, columns = c("test", "elapsed", "replications"))
```

```
##                                          test elapsed replications
## 2 {\n    x <- as.numeric(NA)\n    length(x) <- n\n}   0.029           10
## 1                      x <- rep(as.numeric(NA), n)   0.066           10
```

For matrices, start with the right length vector and then change the dimensions

```
nr <- nc <- 2000
benchmark(x <- matrix(as.numeric(NA), nr, nc), {
    x <- as.numeric(NA)
    length(x) <- nr * nc
    dim(x) <- c(nr, nc)
}, replications = 10, columns = c("test", "elapsed", "replications"))

##
## 2 {\n    x <- as.numeric(NA)\n    length(x) <- nr * nc\n    dim(x) <- c(n
## 1                                                 x <- matrix(as.numeric(NA)
##   elapsed replications
## 2   0.038           10
## 1   0.402           10
```

For lists, we can do this

```
myList <- vector("list", length = n)
```

1.2.2 Vectorized calculations

One key way to write efficient R code is to take advantage of R's vectorized operations.

```
n <- 1e+06
x <- rnorm(n)
system.time(x2 <- x^2)

##    user  system elapsed
##   0.000   0.000   0.001

x2 <- as.numeric(NA)
system.time({
    length(x2) <- n
    for (i in 1:n) {
        x2[i] <- x[i]^2
    }
})  # how many orders of magnitude slower?
```

```
##    user  system elapsed
##   1.984   0.000   1.989
```

So what is different in how R handles the calculations above that explains the disparity? The vectorized calculation is being done natively in C in a for loop. The for loop above involves executing the for loop in R with repeated calls to C code at each iteration. You can usually get a sense for how quickly an R call will pass things along to C by looking at the body of the relevant function(s) being called and looking for *.Primitive*, *.Internal*, *.C,* *.Call*, or *.Fortran*. Let's take a look at the code for '+', *mean.default()*, and *chol.default()*.

Many R functions allow you to pass in vectors, and operate on those vectors in vectorized fashion. So before writing a for loop, look at the help information on the relevant function(s) to see if they operate in a vectorized fashion.

```
line <- c("Four score and 7 years ago, this nation")
startIndices = seq(1, by = 3, length = nchar(line)/3)
substring(line, startIndices, startIndices + 1)

##  [1] "Fo" "r " "co" "e " "nd" "7 " "ea" "s " "go" " t" "is" "na" "io"
```

Challenge: Consider the chi-squared statistic involved in a test of independence in a contingency table:

$$\chi^2 = \sum_i \sum_j \frac{(y_{ij} - e_{ij})^2}{e_{ij}}, \quad e_{ij} = \frac{y_{i\cdot} y_{\cdot j}}{y_{\cdot\cdot}}$$

where $f_{i\cdot} = \sum_j f_{ij}$. Write this in a vectorized way without any loops. Note that 'vectorized' calculations also work with matrices and arrays.

Vectorized operations can also be faster than built-in functions, and clever vectorized calculations even better, though sometimes the code is uglier:

```
x <- rnorm(1e+06)
benchmark(truncx <- ifelse(x > 0, x, 0), {
    truncx <- x
    truncx[x < 0] <- 0
}, truncx <- x * (x > 0), replications = 10, columns = c("test", "elapsed",
    "replications"))

##                                        test elapsed replications
## 1                truncx <- ifelse(x > 0, x, 0)   2.745           10
```

```
## 2 {\n    truncx <- x\n    truncx[x < 0] <- 0\n}    0.376          10
## 3                         truncx <- x * (x > 0)    0.094          10
```

The demo code (see also Section 4.2 of Unit 4) has a surprising example where combining vectorized calculations with a *for* loop is actually faster than using *apply()*. The goal is to remove rows of a large matrix that have any NAs in them.

Additional tips:

- If you do need to loop over dimensions of a matrix or array, if possible loop over the smallest dimension and use the vectorized calculation on the larger dimension(s).
- Looping over columns is likely to be faster than looping over rows given column-major ordering.
- You can use direct arithmetic operations to add/subtract/multiply/divide a vector by each column of a matrix, e.g. `A*b`, multiplies each column of A times a vector b. If you need to operate by row, you can do it by transposing the matrix.

Caution: relying on R's recycling rule in the context of vectorized operations, such as is done when direct-multiplying a matrix by a vector to scale the rows, can be dangerous as the code is not transparent and poses greater dangers of bugs. If it's needed to speed up a calculation, the best approach is to (1) first write the code transparently and then compare the efficient code to make sure the results are the same and (2) comment your code.

Question: What do the points above imply about how to choose to store values in a matrix? How would you choose what should be the row dimension and what should be the column dimension?

1.2.3 Using *apply()* and specialized functions

Another core efficiency strategy is to use the *apply()* functionality. Even better than *apply()* for calculating sums or means of columns or rows (it also can be used for arrays) is *{row,col}{Sums,Means}*:

```
n <- 3000
x <- matrix(rnorm(n * n), nr = n)
system.time(out <- apply(x, 1, mean))

##    user  system elapsed
##   0.212   0.012   0.226

system.time(out <- rowMeans(x))
```

```
##    user  system elapsed
##   0.024   0.000   0.023
```

We can '*sweep*' out a summary statistic, such as subtracting off a mean from each column, using *sweep()*

```
system.time(out <- sweep(x, 2, STATS = colMeans(x), FUN = "-"))

##    user  system elapsed
##   0.244   0.016   0.262
```

Here's a trick for doing it based on vectorized calculations, remembering that if we subtract a vector from a matrix, it subtracts each element of the vector from all the elements in the corresponding ROW.

```
system.time(out2 <- t(t(x) - colMeans(x)))

##    user  system elapsed
##   0.236   0.024   0.262

identical(out, out2)

## [1] TRUE
```

As we've discussed using versions of *apply()* with lists may or may not be faster than looping but generally produces cleaner code. Whether looping is slower will depend on whether a substantial part of the work is in the overhead of the looping or in the time required by the function evaluation on each of the elements. If you're worried about speed, it's a good idea to benchmark the *apply()* variant against looping.

1.2.4 Matrix algebra efficiency

Often calculations that are not explictly linear algebra calculations can be done as matrix algebra. The following can be done faster with *rowSums()*, so it's not a great example, but this sort of trick does come in handy in surprising places.

```
mat <- matrix(rnorm(500 * 500), 500)
benchmark(apply(mat, 1, sum), mat %*% rep(1, ncol(mat)), rowSums(mat), repli
    columns = c("test", "elapsed", "replications"))
```

```
##                          test elapsed replications
## 1        apply(mat, 1, sum)   0.064           10
## 2 mat %*% rep(1, ncol(mat))   0.013           10
## 3              rowSums(mat)   0.006           10
```

On the other hand, big matrix operations can be slow. Suppose you want a new matrix that computes the differences between successive columns of a matrix of arbitrary size. How would you do this as matrix algebra operations? [see demo code] Here it turns out that the *for* loop is much faster than matrix multiplication. However, there is a way to do it faster as matrix direct subtraction. Comment: the demo code also contains some exploration of different ways of creating patterned matrices. Note that this level of optimization is only worth it if you're doing something over and over again, or writing code that you will distribute.

When doing matrix algebra, the order in which you do operations can be critical for efficiency. How should I order the following calculation?

```
n <- 5000
A <- matrix(rnorm(5000 * 5000), 5000)
B <- matrix(rnorm(5000 * 5000), 5000)
x <- rnorm(5000)
res <- A %*% B %*% x
```

We can use the matrix direct product (i.e., `A*B`) to do some manipulations much more quickly than using matrix multiplication. **Challenge**: How can I use the direct product to find the trace of a matrix, XY?

You can generally get much faster results by being smart when using diagonal matrices. Here are some examples, where we avoid directly doing: $X + D$, DX, XD:

```
n <- 1000
X <- matrix(rnorm(n^2), n)
diagvals <- rnorm(n)
D = diag(diagvals)
diag(X) <- diag(X) + diagvals
tmp <- diagvals * X  # instead of D %*% X
tmp2 <- t(t(X) * diagvals)  # instead of X %*% D
```

More generally, sparse matrices and structured matrices (such as block diagonal matrices) can generally be worked with MUCH more efficiently than treating them as arbitrary matrices. The

spam (for arbitrary sparse matrices), *bdsmatrix* (for block-diagonal matrices), and *Matrix* (for a variety of sparse matrix types) packages in R can help, as can specialized code available in other languages, such as C and Fortran packages.

1.2.5 Fast mapping/lookup tables

Sometimes you need to map between two vectors. E.g., $y_{ij} \sim \mathcal{N}(\mu_j, \sigma^2)$ is a basic ANOVA type structure. Here are some efficient ways to aggregate to the cluster level and disaggregate to the observation level.

Disaggregate: Create a vector, *idVec*, that gives a numeric mapping of the observations to their cluster. Then you can access the μ value relevant for each observation as: `mus[idVec]`.

Aggregate: To find the sample means by cluster: `sapply(split(dat$obs, idVec), mean)`

As we've seen R allows you to look up elements of vector by name. For example:

```
vals <- rnorm(10)
names(vals) <- letters[1:10]
select <- c("h", "h", "a", "c")
vals[select]

##       h       h       a       c
## -1.4381 -1.4381 -0.4229  0.1217
```

You can do similar things in terms of looking up by name with dimension names of matrices/arrays, row and column names of dataframes, and named lists.

However, looking things up by name can be slow relative to looking up by index.

```
n = 1e+06
x <- 1:n
xL <- as.list(x)
nms <- as.character(x)
names(x) <- nms
names(xL) <- nms
benchmark(x[5e+05], x["500000"], xL[[5e+05]], xL[["500000"]], replications =

##             test replications elapsed relative user.self sys.self
## 2    x["500000"]           10   0.506       NA     0.484     0.02
## 1       x[5e+05]           10   0.000       NA     0.000     0.00
```

```
## 4 xL[["500000"]]           10   0.067       NA      0.068      0.00
## 3    xL[[5e+05]]           10   0.000       NA      0.000      0.00
##   user.child sys.child
## 2          0         0
## 1          0         0
## 4          0         0
## 3          0         0
```

Why do you think it is slow to look things up by name?

Hashing A hash function is a function that takes as input some data and maps it to a fixed-length output that can be used as a shortened reference to the data. We've seen this in the context of git commits where each commit was labeled with a long base-16 number. This also comes up when verifying files on the Internet. You can compute the hash value on the file you get and check that it is the same as the hash value associated with the legitimate copy of the file.

For our purposes here, hashing can allow one to look up values by their name via a hash table. The idea is that you have a set of key-value pairs (sometimes called a dictionary) where the key is the name associated with the value and the value is some arbitrary object. Hashing allows one to quickly determine an index associated with the key and therefore quickly find the relevant value based on the index. For example, one approach is to compute the hash as a function of the key and then take the remainder when dividing by the number of possible hash values to get the index. In general, the number of unique keys will be larger than the number of unique indices, so there will be collisions [I neglected to make this clear in Unit 2, where I said the result of the hash function would be unique], but usually there will be a small number of keys associated with a given index or slot, and determining the correct value within a given index/slot (also called a bucket) is fast. Put another way, the hash function distributes the keys amongst an array of buckets and allows one to look up the appropriate bucket quickly based on the computed index value. When the hash table is properly set up, the cost of looking up a value does not depend on the number of key-value pairs stored.

As can be inferred from `?environment` and according to Adler, looking up objects by name (i.e., looking up symbols) within an R environment is implemented using hashing, so it can be very fast (in the example here, it is only slightly slower than looking up by index, with some overhead associated with computing the hash value).

```
xEnv <- as.environment(xL)   # convert from a named list
xEnv$"500000"
```

```
## [1] 500000

# I need quotes above because numeric; otherwise xEnv$nameOfObject is fine
xEnv[["500000"]]

## [1] 500000

benchmark(x[5e+05], xL[[5e+05]], xEnv[["500000"]], xEnv$"500000", replicati

##               test replications elapsed relative user.self sys.self
## 1         x[5e+05]        10000   0.031    1.409     0.028        0
## 3 xEnv[["500000"]]        10000   0.035    1.591     0.032        0
## 4    xEnv$"500000"        10000   0.036    1.636     0.036        0
## 2      xL[[5e+05]]        10000   0.022    1.000     0.024        0
##   user.child sys.child
## 1          0         0
## 3          0         0
## 4          0         0
## 2          0         0
```

1.2.6 pqR and other R engines

Radford Neal, a prominent statistician/computer scientist has been working on a project called *pqR* (pretty quick R) to rewrite some aspects of R to make them faster. There are also a few other projects that aim to reimplement the "R engine" such that one could run one's R code with different back ends.

Here are some of the highlights of *pqR* in terms of efficiency, as discussed at http://radfordneal.github.io/pqR/:

1. When R runs code such as just below, it actually creates a vector 1,2,....,n, (which can be computationally and memory intensive for large n) and then iterates over the values in the vector. pqR avoids this vector creation.

   ```
   for(i in 1:n) { }
   vec[1:n]
   ```

2. As we'll see in Section 3.5, R often avoids making copies of objects when it is not necessary. However, the scheme used to do this can be improved so that even fewer copies are made.

3. pqR automatically uses multiple cores for some calculations.

4. pqR avoid some checks for NA and NaN and the like in matrix calculations in which such checking would be slow and it doesn't make sense to check for them anyway.

1.2.7 Byte compiling

R now allows you to compile R code, which goes by the name of byte compiling. Byte-compiled code is a special representation that can be executed more efficiently because it is in the form of compact codes that encode the results of parsing and semantic analysis of scoping and other complexities of the R source code. This byte code can be executed faster than the original R code because it skips the stage of having to be interpreted by the R interpreter.

The functions in the *base* and *stats* packages are now byte-compiled by default. (If you print out a function that is byte-compiled, you'll see something like <*bytecode: 0x243a368*> at the bottom.

We can byte compile our own functions using *cmpfun()*. Here's an example (silly since we we actually do this calculation using vectorized operations):

```
library(compiler)
library(rbenchmark)
f <- function(x) {
    for (i in 1:length(x)) x[i] <- x[i] + 1
    return(x)
}
fc <- cmpfun(f)
fc  # notice the indication that the function is byte compiled.

## function(x) {
##     for (i in 1:length(x)) x[i] <- x[i] + 1
##     return(x)
## }
## <bytecode: 0x24087548>

x <- rnorm(1e+05)
benchmark(f(x), fc(x), x <- x + 1, replications = 5)

##         test replications elapsed relative user.self sys.self user.child
## 2      fc(x)            5   0.147      147     0.144        0          0
## 1       f(x)            5  10.328    10328    10.297        0          0
## 3 x <- x + 1            5   0.001        1     0.004        0          0
```

```
##   sys.child
## 2         0
## 1         0
## 3         0
```

You can compile an entire source file with *cmpfile()*, which produces a *.Rc* file. You then need to use *loadcmp()* to load in the *.Rc* file, which runs the code.

Unfortunately, in my experience, byte compiling doesn't usually speed things up much. As experienced R programmers we would never write the unvectorized code above.

1.3 Challenges

One or more of these challenges may appear on a problem set.

Challenge 1: here's a calculation of the sort needed in mixture component modeling. I have a vector of n observations. I need to find the likelihood of each observation under each of p mixture components (i.e., what's the likelihood if it came from each of the components). So I should produce a matrix of n rows and p columns where the value in the ith row, jth column is the likelihood of the ith observation under the jth mixture component. The idea is that the likelihoods for a given observation are used in assigning observations to clusters. A naive implementation is:

```
> lik <- matrix(NA, nr = n, nc = p)
> for(j in 1:p) lik[ , j] <- dnorm(y, mns[j], sds[j])
```

Note that *dnorm()* can handle matrices and vectors as the observations **and** as the means and sds, so there are multiple ways to do this.

Challenge 2: Suppose you have $y_i \sim \mathcal{N}(\sum_{k=1}^{m_i} w_{i,k}\mu_{ID[i,k]}, \sigma^2)$ for a large number of observations, n. I give you a vector of μ values and a ragged list of weights and a ragged list of IDs identifying the cluster corresponding to each weight (note m_i varies by observation); this is a mixed membership type model. Figure out how to calculate the vector of means, $\sum_k w_{i,k}\mu_{ID[i,k]}$ as fast as possible. Suppose that m_i never gets too big (but μ might have many elements) - could this help you? Part of thinking this through involves thinking about how you want to store the information so that the calculations can be done quickly.

Challenge 3: Write code that simulates a random walk in two dimensions for n steps. First write out a straightforward implementation that involves looping. Then try to speed it up. The *cumsum()* function may be helpful.

Challenge 4: Determine if it's faster to subset based on vector of indices or a vector of logicals. Determine if it matters how big the original object is and how large the subset is, as well as whether the vector of indices is ordered.

2 Advanced topics in working with functions

2.1 Pointers

By way of contrast to R's pass by value system, I want to briefly discuss the idea of a pointer, common in compiled languages such as C.

```
int x = 3;
int* ptr;
ptr = &x;
*ptr * 7; // returns 21
```

Here *ptr* is the address of the integer *x*.

Vectors in C are really pointers to a block of memory:

```
int x[10];
```

In this case *x* will be the address of the first element of the vector. We can access the first element as `x[0]` or `*x`.

Why have we gone into this? In C, you can pass a pointer as an argument to a function. The result is that only the scalar address is copied and not the entire vector, and inside the function, one can modify the original vector, with the new value persisting on exit from the function. For example:

```
int myCal(int *ptr){
   *ptr = *ptr + *ptr;
}
```

When calling C or C++ from R, one (implicitly) passes pointers to the vectors into C. Let's see an example:

```
out <- rep(0, n)
out <- .C("logLik", out = as.double(out),
                    theta = as.double(theta))$out
```

In C, the function definition looks like this:

```
void logLik(double* out, double* theta)
```

2.2 Alternatives to pass by value in R

There are occasions we do not want to pass by value. The main reason is when we want a function to modify a complicated object without having to return it and re-assign it in the parent environment. There are several work-arounds:

1. We can use Reference Class objects. Reference classes are new in R. We'll discuss these in Section 4.

2. We can access the object in the enclosing environment as a 'global variable', as we've seen when discussing scoping. More generally we can access the object using *get()*, specifying the environment from which we want to obtain the variable. Recall that to specify the location of an object, we can generally specify (1) a position in the search path, (2) an explicit environment, or (3) a location in the call stack by using *sys.frame()*. However we cannot change the value of the object in the parent environment without some additional tools.

   (a) We can use the '<<-' operator to assign into an object in the parent environment (provided an object of that name exists in the parent environment).

   (b) We can also use *assign()*, specifying the environment in which we want the assignment to occur.

3. We can use replacement functions (Section 2.4), which hide the reassignment in the parent environment from the user. Note that a second copy is generally created in this case, but the original copy is quickly removed.

4. We can use a closure. This involves creating functions within a function call and returning the functions as a list (or a single function, as we saw when discussing scoping in Unit 4). When one executes the enclosing function, the list is created and one can call the functions of that object. Those functions then can access objects in the enclosing environment (the environment of the original function) and can use '<<-' to assign into the enclosing environment, to which all the functions have access. Chambers provides an example of this in Sec. 5.4.

```
x <- rnorm(10)
f <- function(input) {
    data <- input
    g <- function(param) return(param * data)
    return(g)
}
myFun <- f(x)
rm(x)  # to demonstrate we no longer need x
myFun(3)

##  [1]  5.59860 -3.17828  0.75196 -2.61164  2.91636  4.35508 -0.08599
##  [8] -1.57883  0.02341 -2.88443
```

```
x <- rnorm(1e+07)
myFun <- f(x)
object.size(myFun)  # hmmm

## 1560 bytes

object.size(environment(myFun)$data)

## 80000040 bytes
```

Here's a fun example. You might do this with an *apply()* variant, in particular *replicate()*, but this is slick:

```
make_container <- function(n) {
    x <- numeric(n)
    i <- 1

    function(value = NULL) {
        if (is.null(value)) {
            return(x)
        } else {
            x[i] <<- value
            i <<- i + 1
        }
    }
}
nboot <- 100
bootmeans <- make_container(nboot)
data <- faithful[, 1]  # length of Old Faithful geyser eruption times
for (i in 1:nboot) bootmeans(mean(sample(data, length(data), replace = TRUE)
# this will place results in x in the function env't and you can grab it
# out as
bootmeans()

##   [1] 3.488 3.701 3.356 3.545 3.526 3.494 3.503 3.620 3.393 3.463 3.526
##  [12] 3.513 3.533 3.544 3.468 3.518 3.507 3.520 3.369 3.430 3.504 3.586
##  [23] 3.419 3.477 3.353 3.513 3.436 3.532 3.498 3.548 3.525 3.425 3.548
##  [34] 3.431 3.432 3.559 3.603 3.481 3.517 3.452 3.453 3.467 3.464 3.531
```

```
##  [45] 3.429 3.438 3.424 3.433 3.468 3.511 3.556 3.497 3.461 3.533 3.382
##  [56] 3.503 3.514 3.429 3.412 3.481 3.571 3.536 3.362 3.542 3.448 3.405
##  [67] 3.501 3.563 3.454 3.444 3.518 3.472 3.468 3.389 3.293 3.417 3.526
##  [78] 3.450 3.576 3.516 3.439 3.477 3.574 3.478 3.452 3.549 3.477 3.474
##  [89] 3.494 3.490 3.583 3.512 3.465 3.539 3.505 3.484 3.512 3.411 3.530
## [100] 3.441
```

- A related approach is to wrap data with a function using *with()*.

```
x <- rnorm(10)
myFun2 <- with(list(data = x), function(param) return(param * data))
rm(x)
myFun2(3)

##  [1] -5.5296 -1.3205 -0.8530  2.8166 -6.3337 -5.3886  4.0034  3.0198
##  [9] -0.4103  4.8330

x <- rnorm(1e+07)
myFun2 <- with(list(data = x), function(param) return(param * data))
object.size(myFun2)

## 1560 bytes
```

Question: When would it be useful to have an object carried along with a function as done here?

2.3 Operators

Operators, such as '*+*', '*[*' are just functions, but their arguments can occur both before and after the function call:

```
a <- 7; b <- 3
# let's think about the following as a mathematical function
#  -- what's the function call?
a + b

## [1] 10
```

```
`+`(a, b)

## [1] 10
```

In general, you can use back-ticks to refer to the operators as operators instead of characters. In some cases single or double quotes also work. We can look at the code of an operator as follows using back-ticks to escape out of the standard R parsing, e.g., `` `%*%` ``.

Finally, since an operator is just a function, you can use it as an argument in various places:

```
myList = list(list(a = 1:5, b = "sdf"), list(a = 6:10, b = "wer"))
myMat = sapply(myList, `[[`, 1)
# note that the index '1' is the additional argument to the [[ function
x <- 1:3
y <- c(100, 200, 300)
outer(x, y, `+`)

##      [,1] [,2] [,3]
## [1,]  101  201  301
## [2,]  102  202  302
## [3,]  103  203  303
```

You can define your own *binary* operator (an operator taking two arguments) using a string inside % symbols:

```
`%2%` <- function(a, b) {
    2 * (a + b)
}
3 %2% 7

## [1] 20
```

Since operators are just functions, there are cases in which there are optional arguments that we might not expect. We've already briefly seen the *drop* argument to the '[' operator:

```
mat <- matrix(1:4, 2, 2)
mat[, 1]

## [1] 1 2
```

```
mat[, 1, drop = FALSE]  # what's the difference?

##      [,1]
## [1,]    1
## [2,]    2
```

2.4 Unexpected functions and replacement functions

All code in R can be viewed as a function call.

What do you think is the functional version of the following code? What are the arguments?

```
if (x > 27) {
    print(x)
} else {
    print("too small")
}
```

Assignments that involve functions or operators on the left-hand side (LHS) are called *replacement expressions* or *replacement functions.* These can be quite handy. Here are a few examples:

```
diag(mat) <- c(3, 2)
is.na(vec) <- 3
names(df) <- c("var1", "var2")
```

Replacement expressions are actually function calls. The R interpreter calls the replacement function (which often creates a new object that includes the replacement) and then assigns the result to the name of the original object.

```
mat <- matrix(rnorm(4), 2, 2)
diag(mat) <- c(3, 2)
mat <- `diag<-`(mat, c(10, 21))
base::`diag<-`

## function (x, value)
## {
##     dx <- dim(x)
##     if (length(dx) != 2L)
```

```
##         stop("only matrix diagonals can be replaced")
##     len.i <- min(dx)
##     len.v <- length(value)
##     if (len.v != 1L && len.v != len.i)
##         stop("replacement diagonal has wrong length")
##     if (len.i) {
##         i <- seq_len(len.i)
##         x[cbind(i, i)] <- value
##     }
##     x
## }
## <bytecode: 0x2c23f48>
## <environment: namespace:base>
```

The old version of *mat* still exists until R's memory management cleans it up, but it's no longer referred to by the symbol '*mat*'. Occasionally this sort of thing might cause memory usage to increase (for example it's possible if you're doing replacements on large objects within a loop), but in general things should be fine.

You can define your own replacement functions like this, with the requirements that the last argument be named '*value*' and that the function return the entire object:

```
me <- list(name = "Chris", age = 25)
`name<-` <- function(obj, value) {
    obj$name <- value
    return(obj)
}
name(me) <- "Christopher"
```

2.5 Functions as objects

Note that a function is just an object.

```
x <- 3
x(2)

## Error:  could not find function "x"
```

```
x <- function(z) z^2
x(2)

## [1] 4
```

We can call a function based on the text name of the function.

```
myFun = "mean"
x = rnorm(10)
eval(as.name(myFun))(x)

## [1] -0.02676
```

We can also pass a function into another function either as the actual function object or as a character vector of length one with the name of the function. Here *match.fun()* is a handy function that extracts a function when the function is passed in as an argument of a function. It looks in the calling environment for the function and can handle when the function is passed in as a function object or as a character vector of length 1 giving the function name.

```
f <- function(fxn, x) {
    match.fun(fxn)(x)
}
f("mean", x)

## [1] -0.02676
```

This allows us to write functions in which the user passes in the function (as an example, this works when using *outer()*). Caution: one may need to think carefully about scoping issues in such contexts.

Function objects contain three components: an argument list, a body (a parsed R statement), and an environment.

```
f <- function(x) x
f2 <- function(x) y <- x^2
f3 <- function(x) {
    y <- x^2
    z <- x^3
    return(list(y, z))
```

```
}
class(f)

## [1] "function"

typeof(body(f))

## [1] "symbol"

class(body(f))

## [1] "name"

typeof(body(f2))

## [1] "language"

class(body(f2))

## [1] "<-"

typeof(body(f3))

## [1] "language"

class(body(f3))

## [1] "{"
```

We'll see more about objects relating to the R language and parsed code in a later section. For now, just realize that the parsed code itself is treated as an object(s) with certain types and certain classes.

We can extract the argument object as

```
f4 <- function(x, y = 2, z = 1/y) {
    x + y + z
}
args <- formals(f4)
args

## $x
```

```
##
##
## $y
## [1] 2
##
## $z
## 1/y

class(args)

## [1] "pairlist"
```

A *pairlist* is like a list, but with pairing that in this case pairs argument names with default values.

2.6 Promises and lazy evaluation

One additional concept that it's useful to be aware of is the idea of a *promise* object. In function calls, when R matches user input arguments to formal argument names, it does not (usually) evaluate the arguments until they are needed, which is called *lazy evaluation*. Instead the formal arguments are of a special type called a *promise*. Let's see lazy evaluation in action. Do you think the following code will run?

```
f <- function(a, b = c) {
    c <- log(a)
    return(a * b)
}
f(7)
```

What's strange about that?

Another example:

```
f <- function(x) print("hi")
system.time(mean(rnorm(1e+06)))

##    user  system elapsed
##   0.080   0.000   0.079

system.time(f(3))
```

```
## [1] "hi"
##    user  system elapsed
##   0.004   0.000   0.003

system.time(f(mean(rnorm(1e+06))))

## [1] "hi"
##    user  system elapsed
##   0.000   0.000   0.001
```

3 Evaluating memory use

The main things to remember when thinking about memory use are: (1) numeric vectors take 8 bytes per element and (2) we need to keep track of when large objects are created, including in the frames of functions.

3.1 Allocating and freeing memory

Unlike compiled languages like C, in R we do not need to explicitly allocate storage for objects. However, we have seen that there are times that we do want to allocate storage in advance, rather than successively concatenating onto a larger object.

R automatically manages memory, releasing memory back to the operating system when it's not needed via garbage collection. Occasionally you will want to remove large objects as soon as they are not needed. *rm()* does not actually free up memory, it just disassociates the name from the memory used to store the object. In general R will clean up such objects without a reference (i.e., a name) but you may need to call *gc()* to force the garbage collection. This uses some computation so it's generally not recommended.

In a language like C in which the user allocates and frees up memory, memory leaks are a major cause of bugs. Basically if you are looping and you allocate memory at each iteration and forget to free it, the memory use builds up inexorably and eventually the machine runs out of memory. In R, with automatic garbage collection, this is generally not an issue, but occasionally memory leaks do occur.

3.2 Monitoring overall memory use

There are a number of ways to see how much memory is being used. When R is actively executing statements, you can use *top* from the UNIX shell. In R, *gc()* reports memory use and free memory as *Ncells* and *Vcells*. As far as I know, *Ncells* concerns the overhead of running R and *Vcells* relates to objects created by the user, so you'll want to focus on *Vcells*. You can see the number of Mb currently used (the "*used*" column of the output) and the maximum used in the session (the "*max used*" column)"

```
gc()

##            used  (Mb) gc trigger  (Mb)  max used   (Mb)
## Ncells  5498499 293.7    8125770  434   7904459  422.2
## Vcells 93304715 711.9  162187680 1237 157369192 1200.7

x <- rnorm(1e+08)  # should use about 800 Mb
object.size(x)

## 800000040 bytes

gc()

##             used   (Mb) gc trigger  (Mb)  max used   (Mb)
## Ncells   5498524  293.7    8125770  434   7904459  422.2
## Vcells 193304743 1474.8  213455863 1629 193322983 1475.0

rm(x)
gc()

##            used  (Mb) gc trigger  (Mb)  max used   (Mb)
## Ncells  5498538 293.7    8125770  434   7904459  422.2
## Vcells 93304777 711.9  213455863 1629 193322983 1475.0
```

In Windows only, *memory.size()* tells how much memory is being used.

You can check the amount of memory used by individual objects with *object.size()*.

Here is a useful function, *ls.sizes()*, that wraps *object.size()* to report the largest n objects in a given environment:

```
ls.sizes <- function(howMany = 10, minSize = 1) {
    pf <- parent.frame()
    obj <- ls(pf)  # or ls(sys.frame(-1))
    objSizes <- sapply(obj, function(x) {
        object.size(get(x, pf))
    })
    # or sys.frame(-4) to get out of FUN, lapply(), sapply() and sizes()
    objNames <- names(objSizes)
    howmany <- min(howMany, length(objSizes))
    ord <- order(objSizes, decreasing = TRUE)
    objSizes <- objSizes[ord][1:howMany]
    objSizes <- objSizes[objSizes > minSize]
    objSizes <- matrix(objSizes, ncol = 1)
    rownames(objSizes) <- objNames[ord][1:length(objSizes)]
    colnames(objSizes) <- "bytes"
    cat("object")
    print(format(objSizes, justify = "right", width = 11), quote = FALSE)
}
```

One frustration with memory management is that if your code bumps up against the memory limits of the machine, it can be very slow to respond even when you're trying to cancel the statement with *Ctrl-C*. You can impose memory limits in Linux by starting R (from the UNIX prompt) in a fashion such as this

```
> R --max-vsize=1000M
```

Then if you try to create an object that will push you over that limit or execute code that involves going over the limit, it will simply fail with the message "*Error: vector memory exhausted (limit reached?)*". So this approach may be a nice way to avoid paging/swapping by setting the maximum in relation to the physical memory of the machine. It might also help in debugging memory leaks because the program would fail at the point that memory use was increasing. I haven't played around with this much, so offer this with a note of caution.

We can use an internal function called *inspect()* to see where in memory an object is stored. We'll see that this can be a handy tool for seeing where copies are made and where they are not.

```
a <- rnorm(5)
.Internal(inspect(a))

## @30c0688 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) -0.522089,-0.245464,0.578
```

3.3 Hidden uses of memory

- Replacement functions can hide the use of additional memory. How much memory is used here?

```
x <- rnorm(1e+07)
gc()
dim(x) <- c(10000, 1000)
diag(x) <- 1
gc()
```

- Not all replacement functions actually involve creating a new object and replacing the original object.

```
x <- rnorm(1e+07)
.Internal(inspect(x))

## @7face0694010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.892061,1.25

x[5] <- 7
.Internal(inspect(x))

## @7facdba48010 14 REALSXP g0c7 [NAM(1)] (len=10000000, tl=0) 0.892061,1.25

gc()

##             used  (Mb) gc trigger (Mb)  max used   (Mb)
## Ncells   5498921 293.7    8125770  434   7904459  422.2
## Vcells 103455752 789.4  213455863 1629 193322983 1475.0
```

- Indexing large subsets can involve a lot of memory use.

```
x <- rnorm(1e+07)
gc()
y <- x[1:(length(x) - 1)]
gc()
```

Why was more memory used than just for x and y? Note that this is a limitation of R. Note that R could be designed to avoid this problem (see our discussion of *pqR* earlier in this Unit).

3.4 Passing objects to compiled code

As we've already discussed, when R objects are passed to compiled code (e.g., C or C++), they are passed as pointers and the compiled code uses the memory allocated by R (though it could also allocate additional memory if allocation is part of the code). However, a copy of the object is made, so when calling a C function from R there is some memory overhead.

Furthermore, we need to be aware of any casting that occurs, because the compiled code requires that the R object types match those that the function in the compiled code is expecting.

Here's an example of calling compiled code:

```
res <- .C("fastcount", PACKAGE="GCcorrect", tablex = as.integer(tablex),
tabley = as.integer(tabley), as.integer(xvar), as.integer(yvar),
as.integer(useline), as.integer(length(xvar)))
```

Let's consider when copies are made in casts:

```
f <- function(x) {
    print(.Internal(inspect(x)))
    return(mean(x))
}
x <- rnorm(1e+07)
class(x)
debug(f)
f(x)
f(as.numeric(x))
f(as.integer(x))
```

Next we'll see that C calls do involve a copy, even though it looks like we are just using the same object allocated by R. We'll use the *inline* package to work directly with C code in R and the .C functionality for interfacing with C.

```
library(inline)
src <- "\nfor (int i = 0; i < *n; i++) {\nx[i] = exp(x[i]);\n}\n"
sillyExp <- cfunction(signature(n = "integer", x = "numeric"), src, conventi
# sillyExp <- cfunction(signature(n = 'integer', x = 'numeric'), src,
# convention = '.C')
len <- as.integer(100)  # or 100L
vals <- rnorm(len)
vals[1]
```

```
## [1] 1.583

out1 <- sillyExp(n = len, x = vals)
.Internal(inspect(vals))

## @1f376090 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 1.58343,-0.231186,-0.5

.Internal(inspect(out1$x))

## @1f3763e0 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 4.87162,0.793592,0.571
```

3.5 Delayed copying (copy-on-change)

Next we'll see that something like lazy evaluation occurs outside of functions as well with some functionality called *delayed copying* or *copy-on-change*.

Let's see what goes on within a function in terms of memory use in different situations. Ignore the *gc()* results in the pdf, as we'll start R fresh to get a clean view of memory use during the class demo.

```
f <- function(x) {
    print(gc())
    z <- x[1]
    .Internal(inspect(x))
    return(x)
}
y <- rnorm(1e+07)
gc()

##             used  (Mb) gc trigger (Mb)  max used   (Mb)
## Ncells   5524541 295.1    8125770  434   7904459  422.2
## Vcells 113592934 866.7  213455863 1629 193322983 1475.0

.Internal(inspect(y))

## @7face0694010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) -0.1804

out <- f(y)
```

```
##             used  (Mb) gc trigger (Mb)  max used   (Mb)
## Ncells   5524579 295.1    8125770  434   7904459  422.2
## Vcells 113592896 866.7  213455863 1629 193322983 1475.0
## @7face0694010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) -0.1804
```

```
.Internal(inspect(out))
```

```
## @7face0694010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) -0.1804
```

In fact, this occurs outside function calls as well. Copies of objects are not made until one of the objects is actually modified. Initially, the copy points to the same memory location as the original object.

```
y <- rnorm(1e+07)
gc()
```

```
##             used  (Mb) gc trigger (Mb)  max used   (Mb)
## Ncells   5524597 295.1    8125770  434   7904459  422.2
## Vcells 114643303 874.7  213455863 1629 193322983 1475.0
```

```
.Internal(inspect(y))
```

```
## @7facb7e11010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1.96755
```

```
x <- y
gc()
```

```
##             used  (Mb) gc trigger (Mb)  max used   (Mb)
## Ncells   5524615 295.1    8125770  434   7904459  422.2
## Vcells 104643350 798.4  213455863 1629 193322983 1475.0
```

```
.Internal(inspect(x))
```

```
## @7facb7e11010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1.96755
```

```
x[1] <- 5
gc()
```

```
##             used  (Mb) gc trigger (Mb)  max used   (Mb)
## Ncells   5524640 295.1    8125770  434   7904459  422.2
## Vcells 114643391 874.7  213455863 1629 193322983 1475.0
```

```
.Internal(inspect(x))

## @7facdba48010 14 REALSXP g1c7 [MARK,NAM(1)] (len=10000000, tl=0) 5,0.376

rm(x)
x <- y
.Internal(inspect(x))

## @7facb7e11010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1.96755

.Internal(inspect(y))

## @7facb7e11010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1.96755

y[1] <- 5
.Internal(inspect(x))

## @7facb7e11010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1.96755

.Internal(inspect(y))

## @7facb31c5010 14 REALSXP g0c7 [NAM(1)] (len=10000000, tl=0) 5,0.376387,0
```

As discussed by Radford Neal, who is working to improve the efficiency of R in a project called pqR, *"So R doesn't copy all the time. Instead, it maintains a count, called NAMED, of how many "names" refer to an object, and copies only when an object that needs to be modified is also referred to by another name. Unfortunately, however, this scheme works rather poorly. Many unnecessary copies are still made, while many bugs have arisen in which copies aren't made when necessary.*"

Here are examples of how the NAMED count can be fooled into making a copy unnecessarily. Why do I say these copies are unnecessary and why is NAMED fooled?

```
rm(x, y)
f <- function(x) sum(x^2)
y <- rnorm(10)
f(y)

## [1] 6.536

.Internal(inspect(y))
```

```
## @226778c8 14 REALSXP g0c6 [NAM(2)] (len=10, tl=0) -0.0351136,1.68992,-1.2

y[3] <- 2
.Internal(inspect(y))

## @2266ca58 14 REALSXP g0c6 [NAM(1)] (len=10, tl=0) -0.0351136,1.68992,2,0


a <- 1:5
b <- a
.Internal(inspect(a))

## @860d420 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 1,2,3,4,5

.Internal(inspect(b))

## @860d420 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 1,2,3,4,5

a[2] <- 0
b[2] <- 4
.Internal(inspect(a))

## @2672c48 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,0,3,4,5

.Internal(inspect(b))

## @209d2f8 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,3,4,5
```

We can use the *tracemem()* function to assess what is going on without all those *inspect()* calls. Anything surprising in what you see?

```
a <- 1:10
tracemem(a)

## [1] "<0x1efff4f0>"

## b and a share memory
b <- a
b[1] <- 1
```

```
## tracemem[0x1efff4f0 -> 0x1eeef5e8]: eval eval withVisible withCallingHan
## tracemem[0x1eeef5e8 -> 0x2b82938]: eval eval withVisible withCallingHandl

untracemem(a)
.Internal(inspect(a))

## @1efff4f0 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

.Internal(inspect(b))

## @2b82938 14 REALSXP g0c6 [NAM(1),TR] (len=10, tl=0) 1,2,3,4,5,...
```

Given our understanding of copy-on-change, explain what happens here:

```
y <- 1:5
.Internal(inspect(y))

## @1e536258 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 1,2,3,4,5

x <- y
.Internal(inspect(x))

## @1e536258 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 1,2,3,4,5

y[2] <- 4
.Internal(inspect(y))

## @1f89a2f0 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,3,4,5

.Internal(inspect(x))

## @1e536258 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 1,2,3,4,5
```

Challenge: How much memory is used in the following calculation?

```
x <- rnorm(1e+07)
myfun <- function(y) {
    z <- y
    return(mean(z))
}
```

```
myfun(x)

## [1] -4.43e-07
```

How about here? What is going on?

```
x <- rnorm(1e+07)
x[1] <- NA
myfun <- function(y) {
    return(mean(y, na.rm = TRUE))
}
myfun(x)

## [1] -0.0003814
```

This makes sense if we look at *mean.default()*. Consider where additional memory is used.

3.6 Strategies for saving memory

A couple basic strategies for saving memory include:

- Avoiding unnecessary copies
- Removing objects that are not being used and, if necessary, do a *gc()* call.

If you're really trying to optimize memory use, you may also consider:

- Using reference classes and similar strategies to pass by reference
- Substituting integer and logical vectors for numeric vectors when possible

3.7 Example

Let's work through a real example where we keep a running tally of current memory in use and maximum memory used in a function call. We'll want to consider hidden uses of memory, passing objects to compiled code, and lazy evaluation. This code is courtesy of Yuval Benjamini. For our purposes here, let's assume that *xvar* and *yvar* are very long vectors using a lot of memory.

```
fastcount <- function(xvar, yvar) {
    naline <- is.na(xvar)
    naline[is.na(yvar)] = TRUE
    xvar[naline] <- 0
    yvar[naline] <- 0
    useline <- !naline
    # Table must be initialized for -1's
    tablex <- numeric(max(xvar) + 1)
    tabley <- numeric(max(xvar) + 1)
    stopifnot(length(xvar) == length(yvar))
    res <- .C("fastcount", PACKAGE = "GCcorrect", tablex = as.integer(tablex
        tabley = as.integer(tabley), as.integer(xvar), as.integer(yvar), as
        as.integer(length(xvar)))
    xuse <- which(res$tablex > 0)
    xnames <- xuse - 1
    resb <- rbind(res$tablex[xuse], res$tabley[xuse])
    colnames(resb) <- xnames
    return(resb)
}
```

4 Object-oriented programming (OOP)

Popular languages that use OOP include C++, Java, and Python. In fact C++ is the object-oriented version of C. Different languages implement OOP in different ways.

The idea of OOP is that all operations are built around objects, which have a class, and methods that operate on objects in the class. Classes are constructed to build on (inherit from) each other, so that one class may be a specialized form of another class, extending the components and methods of the simpler class (e.g., *lm* and *glm* objects).

Note that in more formal OOP languages, all functions are associated with a class, while in R, only some are.

Often when you get to the point of developing OOP code in R, you're doing more serious programming, and you're going to be acting as a software engineer. It's a good idea to think carefully in advance about the design of the classes and methods.

4.1 S3 approach

S3 classes are widely-used, in particular for statistical models in the *stats* package. S3 classes are very informal in that there's not a formal definition for an S3 class. Instead, an S3 object is just a primitive R object such as a list or vector with additional attributes including a class name.

Inheritance Let's look at the *lm* class, which builds on lists, and *glm* class, which builds on the *lm* class. Here *mod* is an object (an instance) of class *lm*. An analogy is the difference between a random variable and a realization of that random variable.

```
library(methods)
yb <- sample(c(0, 1), 10, replace = TRUE)
yc <- rnorm(10)
x <- rnorm(10)
mod1 <- lm(yc ~ x)
mod2 <- glm(yb ~ x, family = binomial)
class(mod1)

## [1] "lm"

class(mod2)

## [1] "glm" "lm"

is.list(mod1)

## [1] TRUE

names(mod1)

##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"

is(mod2, "lm")

## [1] TRUE

methods(class = "lm")
```

```
##  [1] add1.lm*            alias.lm*           anova.lm
##  [4] case.names.lm*      confint.lm*         cooks.distance.lm*
##  [7] deviance.lm*        dfbeta.lm*          dfbetas.lm*
## [10] drop1.lm*           dummy.coef.lm*      effects.lm*
## [13] extractAIC.lm*      family.lm*          formula.lm*
## [16] hatvalues.lm        influence.lm*       kappa.lm
## [19] labels.lm*          logLik.lm*          model.frame.lm
## [22] model.matrix.lm     nobs.lm*            plot.lm
## [25] predict.lm          print.lm            proj.lm*
## [28] qr.lm*              residuals.lm        rstandard.lm
## [31] rstudent.lm         simulate.lm*        summary.lm
## [34] variable.names.lm* vcov.lm*
##
##    Non-visible functions are asterisked
```

Often S3 classes inherit from lists (i.e., are special cases of lists), so you can obtain components of the object using the $ operator.

Creating our own class We can create an object with a new class as follows:

```
me <- list(firstname = "Chris", surname = "Paciorek", age = NA)
class(me) <- "indiv"  # there is already a 'person' class in R
```

Actually, if we want to create a new class that we'll use again, we want to create a *constructor* function that initializes new individuals:

```
indiv <- function(firstname = NA, surname = NA, age = NA) {
    # constructor for 'indiv' class
    obj <- list(firstname = firstname, surname = surname, age = age)
    class(obj) <- "indiv"
    return(obj)
}
me <- indiv("Chris", "Paciorek")
```

For those of you used to more formal OOP, the following is probably disconcerting:

```
class(me) <- "silly"
class(me) <- "indiv"
```

Methods The real power of OOP comes from defining *methods*. For example,

```
mod <- lm(yc ~ x)
summary(mod)
gmod <- glm(yb ~ x, family = "binomial")
summary(gmod)
```

Here *summary()* is a generic method (or generic function) that, based on the type of object given to it (the first argument), dispatches a class-specific function (method) that operates on the object. This is convenient for working with objects using familiar functions. Consider the generic methods *plot()*, *print()*, *summary()*, *'['*, and others. We can look at a function and easily see that it is a generic method. We can also see what classes have methods for a given generic method.

```
mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x27bea00>
## <environment: namespace:base>

methods(mean)

## [1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
```

In many cases there will be a default method (here, *mean.default()*), so if no method is defined for the class, R uses the default. Sidenote: arguments to a generic method are passed along to the selected method by passing along the calling environment.

We can define new generic methods:

```
summarize <- function(object, ...) UseMethod("summarize")
```

Once *UseMethod()* is called, R searches for the specific method associated with the class of *object* and calls that method, without ever returning to the generic method. Let's try this out on our *indiv* class. In reality, we'd write either *summary.indiv()* or *print.indiv()* (and of course the generics for *summary* and *print* already exist) but for illustration, I wanted to show how we would write both the generic and the specific method, so I'll write a *summarize* method.

```
summarize.indiv <- function(object) return(with(object, cat("Individual of a
    age, " whose name is ", firstname, " ", surname, ".\n", sep = "")))
summarize(me)

## Individual of age NA whose name is Chris Paciorek.
```

Note that the *print()* function is what is called when you simply type the name of the object, so we can have object information printed out in a structured way. Recall that the output when we type the name of an *lm* object is NOT simply a regurgitation of the elements of the list - rather *print.lm()* is called.

Similarly, when we used `print(object.size(x))` we were invoking the *object_size*-specific print method which gets the value of the size and then formats it. So there's actually a fair amount going on behind the scenes.

Surprisingly, the *summary()* method generally doesn't actually print out information; rather it computes things not stored in the original object and returns it as a new class (e.g., class *summary.lm*), which is then automatically printed, per my comment above, using *print.summary.lm()*, unless one assigns it to a new object. Note that *print.summary.lm()* is hidden from user view.

```
out <- summary(mod)
out
print(out)
getS3method(f = "print", class = "summary.lm")
```

More on inheritance As noted with *lm* and *glm* objects, we can assign more than one class to an object. Here *summarize()* still works, even though the primary class is *BerkeleyIndiv*.

```
class(me) <- c("BerkeleyIndiv", "indiv")
summarize(me)

## Individual of age NA whose name is Chris Paciorek.
```

The classes should nest within one another with the more specific classes to the left, e.g., here a *BerkeleyIndiv* would have some additional objects on top of those of an individual, perhaps *CalnetID*, and perhaps additional or modified methods. *BerkeleyIndiv* inherits from *indiv*, and R uses methods for the first class before methods for the next class(es), unless no such method is defined for the first class. If no methods are defined for any of the classes, R looks for *method.default()*, e.g., *print.default()*, *plot.default()*, etc..

Class-specific operators We can also use operators with our classes. The following example will be a bit silly (it would make more sense with a class that is a mathematical object) but indicates the power of having methods.

```
methods(`+`)

## [1] +.Date   +.POSIXt

`+.indiv` <- function(object, incr) {
    object$age <- object$age + incr
    return(object)
}
old.me <- me + 15
```

Class-specific replacement functions We can use replacement functions with our classes.

This is again a bit silly but we could do the following. We need to define the generic replacement function and then the class-specific one.

```
`age<-` <- function(x, ...) UseMethod("age<-")
`age<-.indiv` <- function(object, value) {
    object$age <- value
    return(object)
}
age(old.me) <- 60
```

Why use class-specific methods? We could have implemented different functionality (e.g., for *summary()*) for different objects using a bunch of *if* statements (or *switch()*) to figure out what class of object is the input, but then we need to have all that checking. Furthermore, we don't control the *summary()* function, so we would have no way of adding the additional conditions in a big if-else statement. The OOP framework makes things *extensible*, so we can build our own new functionality on what is already in R.

Final thoughts Consider the *Date* class discussed in the R bootcamp. This is another example of an S3 class, with methods such as *julian()*, *weekdays()*, etc.

Challenge: how would you get R to quit immediately, without asking for any more information, when you simply type '*q*' (no parentheses!)?

What we've just discussed are the old-style R (and S) object orientation, called S3 methods. The new style is called S4 and we'll discuss it next. S3 is still commonly used, in part because S4 can be slow (or at least it was when I last looked into this a few years ago). S4 is more structured than S3.

4.2 S4 approach

S4 methods are used a lot in *bioconductor*. They're also used in *lme4*, among other packages. Tools for working with S4 classes are in the *methods* package.

Note that components of S4 objects are obtained as `object@component` so they do not use the usual list syntax. The components are called *slots*, and there is careful checking that the slots are specified and valid when a new object of a class is created. You can use the *prototype* argument to *setClass()* to set default values for the slots. There is a default constructor (the method is actually called *initialize()*), but you can modify it. One can create methods for operators and for replacement functions too. For S4 classes, there is a default method invoked when *print()* is called on an object in the class (either explicitly or implicitly) - the method is actually called *show()* and it can also be modified. Let's reconsider our *indiv* class example in the S4 context.

```
library(methods)
setClass("indiv",
representation(
name = "character",

age = "numeric",

birthday = "Date"
)
)
me <- new("indiv", name = 'Chris', age = 20,
birthday = as.Date('91-08-03'))
# next notice the missing age slot
me <- new("indiv", name = 'Chris',
birthday = as.Date('91-08-03'))
# finally, apparently there's not a default object of class Date
me <- new("indiv", name = 'Chris', age = 20)

## Error:  invalid class "indiv" object:  invalid object for slot "birthday"
```

```
in class "indiv":  got class "S4", should be or extend class "Date"
```

S4 methods are designed to be more structured than S3, with careful checking of the slots.

```
setValidity("indiv", function(object) {
    if (!(object@age > 0 && object@age < 130))
        return("error: age must be between 0 and 130")
    if (length(grep("[0-9]", object@name)))
        return("error: name contains digits")
    return(TRUE)
    # what other validity check would make sense given the slots?
})
```

```
## Class "indiv" [in ".GlobalEnv"]
##
## Slots:
##
## Name:       name       age  birthday
## Class: character   numeric      Date
```

```
me <- new("indiv", name = "5z%a", age = 20, birthday = as.Date("91-08-03"))
```

```
## Error:  invalid class "indiv" object:  error:  name contains digits
```

```
me <- new("indiv", name = "Z%a B''*", age = 20, birthday = as.Date("91-08-0
me@age <- 150  # so our validity check is not foolproof
```

To deal with this latter issue of the user mucking with the slots, it's recommended when using OOP that slots only be accessible through methods that operate on the object, e.g., a *setAge()* method, and then check the validity of the supplied age within *setAge()*.

Here's how we create generic and class-specific methods. Note that in some cases the generic will already exist.

```
# generic method
setGeneric("isVoter", function(object, ...) {
    standardGeneric("isVoter")
})

## [1] "isVoter"

# class-specific method
isVoter.indiv <- function(object) {
    if (object@age > 17) {
        cat(object@name, "is of voting age.\n")
    } else cat(object@name, "is not of voting age.\n")
}
setMethod(isVoter, signature = c("indiv"), definition = isVoter.indiv)

## [1] "isVoter"
## attr(,"package")
## [1] ".GlobalEnv"

isVoter(me)

## Z%a B''* is of voting age.
```

We can have method signatures involve multiple objects. Here's some syntax where we'd fill in the function body with appropriate code - perhaps the plus operator would create a child.

```
setMethod(`+`, signature = c("indiv", "indiv"),
definition = function(indiv1, indiv2) { }
```

As with S3, classes can inherit from one or more other classes. Chambers calls the class that is being inherited from a *superclass*.

```
setClass("BerkeleyIndiv",
representation(
```

```
CalID = "character"
),

contains = "indiv"
)
me <- new("BerkeleyIndiv", name = "Chris", age = 20,
   birthday = as.Date('91-08-03'), CalID = "01349542")
isVoter(me)

## Chris is of voting age.

is(me, "indiv")

## [1] TRUE
```

For a more relevant example suppose we had spatially-indexed time series. We could have a time series class, a spatial location class, and a "location time series" class that inherits from both. Be careful that there are not conflicts in the slots or methods from the multiple classes. For conflicting methods, you can define a method specific to the new class to deal with this. Also, if you define your own *initialize()* method, you'll need to be careful that you account for any initialization of the superclass(es) and for any classes that might inherit from your class (see help on *new()* and Chambers, p. 360).

You can inherit from other S4 classes (which need to be defined or imported into the environment in which your class is created), but not S3 classes. You can inherit (at most one) of the basic R types, but not environments, symbols, or other non-standard types. You can use S3 classes in slots, but this requires that the S3 class be declared as an S4 class. To do this, you create S4 versions of S3 classes use *setOldClass()* - this creates a virtual class. This has been done, for example, for the *data.frame* class:

```
showClass("data.frame")

## Class "data.frame" [package "methods"]
##
## Slots:
##
## Name:                .Data               names           row.names
## Class:                list           character data.frameRowLabels
##
```

```
## Name:               .S3Class
## Class:             character
##
## Extends:
## Class "list", from data part
## Class "oldClass", directly
## Class "vector", by class "list", distance 2
```

You can use *setClassUnion()* to create what Adler calls *superclass* and what Chambers calls a *virtual class* that allows for methods that apply to multiple classes. So if you have a person class and a pet class, you could create a "named lifeform" virtual class that has methods for working with name and age slots, since both people and pets would have those slots. You can't directly create an object in the virtual class.

4.3 Reference classes

Reference classes are a new construct in R. They are classes somewhat similar to S4 that allow us to access their fields by reference. Importantly, they behave like pointers (the fields in the objects are 'mutable'). Let's work through an example where we set up the fields of the class (like S4 slots) and class methods, including a constructor. Note that one cannot add fields to an already existing class.

Here's the initial definition of the class.

```
tsSimClass <- setRefClass("tsSimClass",
fields = list(
   n = "numeric",
   times = "numeric",
   corMat = "matrix",
   lagMat = "matrix",
   corParam = "numeric",
   U = "matrix",
   currentU = "logical"),

methods = list(
initialize = function(times = 1:10, corParam = 1, ...){
   # we seem to need default values for the copy() method to function prope
   require(fields)
```

```
   times <<- times # field assignment requires using <<-
   n <<- length(times)
   corParam <<- corParam
   currentU <<- FALSE
   calcMats()
   callSuper(...) # calls initializer of base class (envRefClass)
},

calcMats = function(){
   # Python-style doc string
   ' calculates correlation matrix and Cholesky factor '
   lagMat <- rdist(times) # local variable
   corMat <<- exp(-lagMat / corParam) # field assignment
   U <<- chol(corMat) # field assignment
   cat("Done updating correlation matrix and Cholesky factor\n")
   currentU <<- TRUE
},

changeTimes = function(newTimes){
   times <<- newTimes
   calcMats()
},

show = function(){ # 'print' method
   cat("Object of class 'tsSimClass' with ", n, " time points.\n", sep = ''
}
)
)

## Warning:  local assignment to field name will not change the field:
##    lagMat <- rdist(times)
##  Did you mean to use "«-"?  ( in method "calcMats" for class "tsSimClass
```

We can add methods after defining the class.

```
tsSimClass$methods(list(

simulate = function(){
  ' simulates random processes from the model '
   if(!currentU)
    calcMats()
   return(crossprod(U, rnorm(n)))
})
)
```

Now let's see how we would use the class.

```
master <- tsSimClass$new(1:100, 10)
master
tsSimClass$help("calcMats")
devs <- master$simulate()
plot(master$times, devs, type = "l")
mycopy <- master
myDeepCopy <- master$copy()
master$changeTimes(seq(0, 1, length = 100))
mycopy$times[1:5]
myDeepCopy$times[1:5]
```

A few additional points:

- As we just saw, a copy of an object is just a pointer to the original object, unless we explicitly invoke the *copy()* method.
- As with S3 and S4, classes can inherit from other classes. E.g., if we had a *simClass* and we wanted the *tsSimClass* to inherit from it:

  ```
  setRefClass("tsSimClass", contains = "simClass")
  ```

  - We can call a method inherited from the superclass from within a method of the same name with *callSuper(...)*, as we saw for our *initialize()* method.
- If we need to refer to a field or change a field we can do so without hard-coding the field name as:

```
master$field("times")[1:5]
# the next line is dangerous in this case, since currentU will no longer
# be accurate
master$field("times", 1:10)
```

- Note that reference classes have Python style doc strings. We get help on a class with *class$help()*, e.g. `tsSimClass$help()`. This prints out information, including the doc strings.
- If you need to refer to the entire object within an object method, you refer to it as *.self*. E.g., with our *tsSimClass* object, `.self$U` would refer to the Cholesky factor. This is sometimes necessary to distinguish a class field from an argument to a method.

5 Creating and working in an environment

We've already talked extensively about the environments that R creates. Occasionally you may want to create an environment in which to store objects.

```
e <- new.env()
assign("x", 3, envir = e)  # same as e$x <- 3
e$x

## [1] 3

get("x", envir = e, inherits = FALSE)

## [1] 3

# the FALSE avoids looking for x in the enclosing environments
e$y <- 5
objects(e)

## [1] "x" "y"

rm("x", envir = e)
parent.env(e)

## <environment: R_GlobalEnv>
```

Before the existence of Reference Classes, using an environment was one way to pass objects by reference, avoiding having to re-assign the output. Here's an example where we iteratively update a random walk.

```
myWalk <- new.env()
myWalk$pos = 0
nextStep <- function(walk) walk$pos <- walk$pos + sample(c(-1, 1), size = 1)
nextStep(myWalk)
```

We can use *eval()* to evaluate some code within a specified environment. By default, it evaluates in the result of *parent.frame()*, which amounts to evaluating in the frame from which *eval()* was called. *evalq()* avoids having to use *quote()*.

```
eval(quote(pos <- pos + sample(c(-1, 1), 1)), envir = myWalk)
evalq(pos <- pos + sample(c(-1, 1), 1), envir = myWalk)
```

6 Computing on the language

6.1 The R interpreter

Parsing When you run R, the R interpreter takes the code you type or the lines of code that are read in a batch session and parses each statement, translating the text into functional form. It substitutes objects for the symbols (names) that represent those objects and evaluates the statement, returning the resulting object. For complicated R code, this may be recursive.

Since everything in R is an object, the result of parsing is an object that we'll be able to investigate, and the result of evaluating the parsed statement is an object.

We'll see more on parsing in the next section.

***.Primitive()* and *.Internal()* (and *.External*())** Some functionality is implemented internally within the C implementation that lies at the heart of R. If you see *.Internal()* or *.Primitive()* or *.External()*, in the code of a function, you know it's implemented internally (and therefore generally very quickly). Unfortunately, it also means that you don't get to see R code that implements the functionality, though Chambers p. 465 describes how you can look into the C source code. Basically you need to download the source code for the relevant package off of CRAN.

```
plot.xy  # plot.xy() is called by plot.default()

## function (xy, type, pch = par("pch"), lty = par("lty"), col = par("col"),
##     bg = NA, cex = 1, lwd = par("lwd"), ...)
## invisible(.External.graphics(C_plotXY, xy, type, pch, lty, col,
##     bg, cex, lwd, ...))
## <bytecode: 0x1d5a8a68>
## <environment: namespace:graphics>

print(`%*%`)

## function (x, y)  .Primitive("%*%")
```

6.2 Parsing code and understanding language objects

R code can be manipulated in text form and we can actually write R code that will create or manipulate R code. We can then evaluate that R code using *eval()*.

quote() will parse R code, but not evaluate it. This allows you to work with the code rather than the results of evaluating that code. The *print()* method for language objects is not very helpful! But we can see the parsed code by treating the result as a list.

```
obj <- quote(if (x > 1) "orange" else "apple")
as.list(obj)

## [[1]]
## `if`
##
## [[2]]
## x > 1
##
## [[3]]
## [1] "orange"
##
## [[4]]
## [1] "apple"

class(obj)
```

```
## [1] "if"

weirdObj <- quote(`if`(x > 1, 'orange', 'apple'))
identical(obj, weirdObj)

## [1] TRUE
```

Recall that to access symbols that involve special syntax (such as special characters), you use backquotes.

Officially, the name that you assign to an object (including functions) is a *symbol*.

```
x <- 3
typeof(quote(x))

## [1] "symbol"
```

We can create an *expression* object that contains R code as

```
myExpr <- expression(x <- 3)
eval(myExpr)
typeof(myExpr)

## [1] "expression"
```

The difference between *quote()* and *expression()* is basically that *quote()* works with a single statement, while *expression()* can deal with multiple statements, returning a list-like object of parsed statements. Both of them parse R code.

```
a <- quote(x <- 5)
b <- expression(x <- 5, y <- 3)
class(a)

## [1] "<-"

class(b)

## [1] "expression"

b[[1]]
```

```
## x <- 5

class(b[[1]])

## [1] "<-"

identical(a, b[[1]])

## [1] TRUE
```

The following table shows the *language* objects in R; note that there are three classes of language objects: *expressions*, *calls*, and *names*.

	Example syntax to create	Class	Type
object names	quote(x)	name	symbol (language)
expressions	expression(x <- 3)	expression	expression (language)
function calls	quote(f())	call	language
if statements	quote(if(x < 3) y=5)	if (call)	language
for statement	quote(for(i in 1:5) {})	for (call)	language
assignments	quote(x <- 3)	<- (call)	language
operators	quote(3 + 7)	call	language

Basically any standard function, operator, *if* statement, *for* statement, assignment, etc. are function calls and inherit from the *call* class.

Objects of type language are not officially lists, but they can be queried as such. You can convert between language objects and lists with *as.list()* and *as.call()*.

An official expression is one or more syntactically correct R statements. When we use *quote()*, we're working with a single statement, while *expression()* will create a list of separate statements (essentially separate call objects). I'm trying to use the term *statement* to refer colloquially to R code, rather than using the term *expression*, since that has formal definition in this context.

Let's take a look at some examples of language objects and parsing.

```
e0 <- quote(3)
e1 <- expression(x <- 3)
e1m <- expression({x <- 3; y <- 5})
e2 <- quote(x <- 3)
e3 <- quote(rnorm(3))
print(c(class(e0), typeof(e0)))

## [1] "numeric" "double"
```

```
print(c(class(e1), typeof(e1)))

## [1] "expression" "expression"

print(c(class(e1[[1]]), typeof(e1[[1]])))

## [1] "<-"       "language"

print(c(class(e1m), typeof(e1m)))

## [1] "expression" "expression"

print(c(class(e2), typeof(e2)))

## [1] "<-"       "language"

identical(e1[[1]], e2)

## [1] TRUE

print(c(class(e3), typeof(e3)))

## [1] "call"     "language"

e4 <- quote(-7)
print(c(class(e4), typeof(e4))) # huh? what does this imply?

## [1] "call"     "language"

as.list(e4)

## [[1]]
## `-`
##
## [[2]]
## [1] 7
```

We can evaluate language types using *eval()*:

```
rm(x)
eval(e1)
rm(x)
eval(e2)
e1mlist <- as.list(e1m)
e2list <- as.list(e2)
eval(as.call(e2list))
# here's how to do it if the language object is actually an expression
# (multiple statements)
eval(as.expression(e1mlist))
```

Now let's look in more detail at the components of R expressions. We'll be able to get a sense from this of how R evaluates code. We see that when R evaluates a parse tree, the first element says what function to use and the remaining elements are the arguments. But in many cases one or more arguments will themselves be call objects, so there's recursion.

```
e1 = expression(x <- 3)
# e1 is one-element list with the element an object of class '<-'
print(c(class(e1), typeof(e1)))

## [1] "expression" "expression"

e1[[1]]

## x <- 3

as.list(e1[[1]])

## [[1]]
## `<-`
##
## [[2]]
## x
##
## [[3]]
## [1] 3

lapply(e1[[1]], class)
```

```
## [[1]]
## [1] "name"
##
## [[2]]
## [1] "name"
##
## [[3]]
## [1] "numeric"
```

```
y = rnorm(5)
e3 = quote(mean(y))
print(c(class(e3), typeof(e3)))
```

```
## [1] "call"     "language"
```

```
e3[[1]]
```

```
## mean
```

```
print(c(class(e3[[1]]), typeof(e3[[1]])))
```

```
## [1] "name"   "symbol"
```

```
e3[[2]]
```

```
## y
```

```
print(c(class(e3[[2]]), typeof(e3[[2]])))
```

```
## [1] "name"   "symbol"
```

```
# we have recursion
e3 = quote(mean(c(12, 13, 15)))
as.list(e3)
```

```
## [[1]]
## mean
##
## [[2]]
## c(12, 13, 15)
```

```
as.list(e3[[2]])
```

```
## [[1]]
## c
##
## [[2]]
## [1] 12
##
## [[3]]
## [1] 13
##
## [[4]]
## [1] 15
```

6.3 Manipulating the parse tree

Of course since the parsed code is just an object, we can manipulate it, i.e., *compute on the language*:

```
out <- quote(y <- 3)
out[[3]] <- 4
eval(out)
y
```

```
## [1] 4
```

Here's another example:

```
e1 <- quote(4 + 5)
e2 <- quote(plot(x, y))
e2[[1]] <- `+`
eval(e2)
```

```
## [1] 7
```

```
e1[[3]] <- e2
e1
```

```
## 4 + .Primitive("+")(x, y)

class(e1[[3]])  # note the nesting

## [1] "call"

eval(e1)  # what should I get?

## [1] 11
```

We can also turn it back into standard R code, as a character, using *deparse()*, which turns the parse tree back into R code as text. *parse()* is like *quote()* but it takes the code in the form of a string rather than an actual expression:

```
codeText <- deparse(out)
parsedCode <- parse(text = codeText)
# parse() works like quote() except on the code in the form of a string
eval(parsedCode)
deparse(quote(if (x > 1) "orange" else "apple"))

## [1] "if (x > 1) \"orange\" else \"apple\""
```

Note that the quotes have been escaped since they're inside a string.

It can be very useful to be able to convert names of objects that are in the form of text to names that R interprets as symbols referring to objects:

```
x3 <- 7
i <- 3
as.name(paste("x", i, sep = ""))

## x3

eval(as.name(paste("x", i, sep = "")))

## [1] 7

assign(paste("x", i, sep = ""), 11)
x3

## [1] 11
```

6.4 Parsing replacement expressions

Let's consider replacement expressions.

```
animals = c("cat", "dog", "rat", "mouse")
out1 = quote(animals[4] <- "rat")
out2 = quote(animals[4] <- "rat")
out3 = quote(`[<-`(animals, 4, "rat"))
as.list(out1)

## [[1]]
## `<-`
##
## [[2]]
## animals[4]
##
## [[3]]
## [1] "rat"

as.list(out2)

## [[1]]
## `<-`
##
## [[2]]
## animals[4]
##
## [[3]]
## [1] "rat"

identical(out1, out2)

## [1] TRUE

as.list(out3)

## [[1]]
## `[<-`
##
```

```
## [[2]]
## animals
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] "rat"
```

```
identical(out1, out3)
```

```
## [1] FALSE
```

```
typeof(out1[[2]])  # language
```

```
## [1] "language"
```

```
class(out1[[2]])  # call
```

```
## [1] "call"
```

The parse tree for *out3* is different than those for *out1* and *out2*, but when *out3* is evaluated the result is the same as for *out1* and *out2*:

```
eval(out1)
animals
```

```
## [1] "cat" "dog" "rat" "rat"
```

```
animals[4] = "mouse"  # reset things to original state
eval(out3)
```

```
## [1] "cat" "dog" "rat" "rat"
```

```
animals  # both do the same thing
```

```
## [1] "cat" "dog" "rat" "rat"
```

Why? When R evaluates a call to '<-', if the first argument is a name, then it does the assignment, but if the first argument (i.e. what's on the left-hand side of the "assignment") is a call then

it calls the appropriate replacement function. The second argument (the value being assigned) is evaluated first. Ultimately in all of these cases, the replacement function is used.

6.5 substitute()

The substitute function acts like *quote()*:

```
identical(quote(z <- x^2), substitute(z <- x^2))

## [1] TRUE
```

But if you also pass *substitute()* an environment, it will replace symbols with their object values in that environment.

```
e <- new.env()
e$x <- 3
substitute(z <- x^2, e)

## z <- 3^2
```

This can do non-sensical stuff:

```
e$z <- 5
substitute(z <- x^2, e)

## 5 <- 3^2
```

Let's see a practical example of substituting for variables in statements:

```
plot(x = rnorm(5), y = rgamma(5, 1)) # how does plot get the axis label names?
```

In the *plot()* function, you can see this syntax:

```
xlabel <- if(!missing(x)) deparse(substitute(x))
```

So what's going on is that within *plot.default()*, it substitutes in for '*x*' with the statement that was passed in as the *x* argument, and then uses *deparse()* to convert to character. The fact that *x* still has *rnorm(5)* associated with it rather than the five numerical values from evaluating *rnorm()* has to do with lazy evaluation and promises. Here's the same idea in action in a stripped down example:

```
f <- function(obj) {
    objName <- deparse(substitute(obj))
    print(objName)
}
f(y)

## [1] "y"
```

More generally, we can substitute into *expression* and *call* objects by providing a named list (or an environment) - the substition happens within the context of this list.

```
substitute(a + b, list(a = 1, b = quote(x)))

## 1 + x
```

Things can get intricate quickly:

```
e1 <- quote(x + y)
e2 <- substitute(e1, list(x = 3))
```

The problem is that *substitute()* doesn't evaluate its first argument, "*e1*", so it can't replace the parsed elements in *e1*. Instead, we'd need to do the following, where we force the evaluation of *e1*:

```
e2 <- substitute(substitute(e, list(x = 3)), list(e = e1))
substitute(substitute(e, list(x = 3)), list(e = e1))

## substitute(x + y, list(x = 3))

# so e1 is substituted as an evaluated object, which then allows for
# substitution for 'x'
e2

## substitute(x + y, list(x = 3))

eval(e2)

## 3 + y
```

If this subsection is confusing, let me assure you that it has confused me too. The indirection going on here is very involved.

6.6 Final thoughts

Challenge: figure out how a *for* loop is parsed in R. See how a *for* loop with one statement within the loop differs from one with two or more statements.

We'll see *expression()* again when we talk about inserting mathematical notation in plots.