www.JntukMaterials.com

UNIT II
R Programming Structures

Control Statements

Control statements in R look very similar to those of the ALGOL-descendant family languages mentioned above. Here, we'll look at loops and if-else statements

Loops

In Section 1.3, we defined the oddcount() function. In that function, the following line should have been instantly recognized by Python programmers:

for (n in x) {

It means that there will be one iteration of the loop for each component of the vector x, with n taking on the values of those components—in the first iteration, n = x[1]; in the second iteration, n = x[2]; and so on. For example, the following code uses this structure to output the square of every element in a vector:

```
> x <- c(5,12,13)
> for (n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

C-style looping with while and repeat is also available, complete with break, a statement that causes control to leave the loop. Here is an example that uses all three:

```
> i <- 1
> while (i <= 10) i <- i+4
> i
[1] 13
>
> i <- 1
> while(TRUE) { # similar loop to above
+ i <- i+4
+ if (i > 10) break
+ }
> i
[1] 13
>
> i <- 1
> repeat { # again similar
+ i <- i+4
+ if (i > 10) break
+ }
> i
[1] 13
```

In the first code snippet, the variable i took on the values 1, 5, 9, and 13 as the loop went through its iterations. In that last case, the condition i <= 10 failed, so the break took hold and we left the loop.

This code shows three different ways of accomplishing the same thing,with break playing a key role in the second and third ways. Note that repeat has no Boolean exit condition. You must use break (or something like return()). Of course, break can be used with for loops, too.Another useful statement is next, which instructs the

www.JntukMaterials.com
interpreter to skip the remainder of the current iteration of the loop and proceed directly to the next one. This provides a way to avoid using complexly nested if-thenelse constructs, which can make the code confusing. Let's take a look at an example that uses next.

```
1 sim <- function(nreps) {
2 commdata <- list()
3 commdata$countabsamecomm <- 0
4 for (rep in 1:nreps) {
5 commdata$whosleft <- 1:20
6 commdata$numabchosen <- 0
7 commdata <- choosecomm(commdata,5)
8 if (commdata$numabchosen > 0) next
9 commdata <- choosecomm(commdata,4)
10 if (commdata$numabchosen > 0) next
11 commdata <- choosecomm(commdata,3)
12 }
13 print(commdata$countabsamecomm/nreps)
14 }
```

There are next statements in lines 8 and 10. Let's see how they work and how they improve on the alternatives. The two next statements occur within the loop that starts at line 4. Thus, when the if condition holds in line 8, lines 9 through 11 will be skipped, and control will transfer to line 4.

The situation in line 10 is similar. Without using next, we would need to resort to nested if statements,something like these:

```
1 sim <- function(nreps) {
2 commdata <- list()
3 commdata$countabsamecomm <- 0
4 for (rep in 1:nreps) {
5 commdata$whosleft <- 1:20
6 commdata$numabchosen <- 0
7 commdata <- choosecomm(commdata,5)
8 if (commdata$numabchosen == 0) {
9 commdata <- choosecomm(commdata,4)
10 if (commdata$numabchosen == 0)
11 commdata <- choosecomm(commdata,3)
12 }
13 }
print(commdata$countabsamecomm/nreps)
15 }
```

Because this simple example has just two levels, it's not too bad. However, nested if statements can become confusing when you have more levels. The for construct works on any vector, regardless of mode. You can loop over a vector of filenames, for instance. Say we have a file named *file1* with the following contents:

```
1
2
3
4
5
6
```

We also have a file named *file2* with these contents:

```
5
```

www.JntukMaterials.com
12
13
The following loop reads and prints each of these files. We use the scan() function here to read in a file of numbers and store those values in a vector.

```
> for (fn in c("file1","file2")) print(scan(fn))
Read 6 items
[1] 1 2 3 4 5 6
Read 3 items
[1] 5 12 13
```

So, fn is first set to *file1*, and the file of that name is read in and printed out. Then the same thing happens for *file2*.

7.1.2 Looping Over Nonvector Sets

R does not directly support iteration over nonvector sets, but there are a couple of indirect yet easy ways to accomplish it:

- Use lapply(), assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order.
- Use get(). As its name implies, this function takes as an argument a character string representing the name of some object and returns the object of that name. It sounds simple, but get() is a very powerful function.

Let's look at an example of using get(). Say we have two matrices, u and v, containing statistical data, and we wish to apply R's linear regression function lm() to each of them.

```
> u
[,1] [,2]
[1,] 1 1
[2,] 2 2
[3,] 3 4
> v
[,1] [,2]
[1,] 8 15
[2,] 12 10
[3,] 20 2
> for (m in c("u","v")) {
+ z <- get(m)
+ print(lm(z[,2] ~ z[,1]))
+ }
Call:
lm(formula = z[, 2] ~ z[, 1])
Coefficients:
(Intercept) z[, 1]
-0.6667 1.5000
Call:
lm(formula = z[, 2] ~ z[, 1])
Coefficients:
(Intercept) z[, 1]
23.286 -1.071
```

Here, m was first set to u. Then these lines assign the matrix u to z, which allows the call to lm() on u:

```
z <- get(m)
print(lm(z[,2] ~ z[,1]))
```

The same then occurs with v.

www.JntukMaterials.com

if-else

The syntax for if-else looks like this:

```
if (r == 4) {
x <- 1
} else {
x <- 3
y <- 4
}
```

It looks simple, but there is an important subtlety here. The if section consists of just a single statement:

```
x <- 1
```

So, you might guess that the braces around that statement are not necessary. However, they are indeed needed. The right brace before the else is used by the R parser to deduce that this is an if-else rather than just an if. In interactive mode, without braces, the parser would mistakenly think the latter and act accordingly, which is not what we want.

An if-else statement works as a function call, and as such, it returns the last value assigned.

```
v <- if (cond) expression1 else expression2
```

This will set v to the result of expression1 or expression2, depending on whether cond is true. You can use this fact to compact your code. Here's a simple example:

```
> x <- 2
> y <- if(x == 2) x else x+1
> y
[1] 2
> x <- 3
> y <- if(x == 2) x else x+1
> y
[1] 4
```

Without taking this tack, the code

```
y <- if(x == 2) x else x+1
```

would instead consist of the somewhat more cluttered

```
if(x == 2) y <- x else y <- x+1
```

In more complex examples, expression1 and/or expression2 could be function calls. On the other hand, you probably should not let compactness take priority over clarity. When working with vectors, use the ifelse() function

Arithmetic and Boolean Operators and Values

Table 7-1 lists the basic operators.

Table 7-1: Basic R Operators :**Operation Description**

x + y Addition
x - y Subtraction
x * y Multiplication
x / y Division
x ^ y Exponentiation
x %% y Modular arithmetic
x %/% y Integer division
x == y Test for equality

www.JntukMaterials.com
x <= y Test for less than or equal to
x >= y Test for greater than or equal to
x && y Boolean AND for scalars
x || y Boolean OR for scalars
x & y Boolean AND for vectors (vector x,y,result)
x | y Boolean OR for vectors (vector x,y,result)
!x Boolean negation
Though R ostensibly has no scalar types, with scalars being treated as one-element vectors, we see the exception in Table 7-1: There are different Boolean operators for the scalar and vector cases. This may seem odd, but a simple example will demonstrate the need for such a distinction.

```
> x
[1] TRUE FALSE TRUE
> y
[1] TRUE TRUE FALSE
> x & y
[1] TRUE FALSE FALSE
> x[1] && y[1]
[1] TRUE
> x && y # looks at just the first elements of each vector
[1] TRUE
> if (x[1] && y[1]) print("both TRUE")
[1] "both TRUE"
> if (x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
```

the condition has length > 1 and only the first element will be used The central point is that in evaluating an if, we need a single Boolean, not a vector of Booleans, hence the warning seen in the preceding example,
as well as the need for having both the & and && operators.

The Boolean values TRUE and FALSE can be abbreviated as T and F (both must be capitalized). These values change to 1 and 0 in arithmetic expressions:

```
> 1 < 2
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
> (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
> (1 < 2) == TRUE
[1] TRUE
> (1 < 2) == 1
[1] TRUE
```

In the second computation, for instance, the comparison 1 < 2 returns TRUE, and 3 < 4 yields TRUE as well. Both values are treated as 1 values, so the product is 1.

On the surface, R functions look similar to those of C, Java, and so on. However, they have much more of a functional programming flavor, which has direct implications for the R programmer.

Default Values for Arguments :
In Section 5.1.2, we read in a data set from a file named exams:

www.JntukMaterials.com

```
> testscores <- read.table("exams",header=TRUE)
```

The argument header=TRUE tells R that we have a header line, so R should not count that first line in the file as data.
This is an example of the use of *named arguments*. Here are the first few lines of the function:

```
> read.table
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
comment.char = "#", allowEscapes = FALSE, flush = FALSE,
stringsAsFactors = default.stringsAsFactors(), encoding = "unknown")
{
if (is.character(file)) {
file <- file(file, "r")
on.exit(close(file))
...
...
```

The second formal argument is named header. The = FALSE field means that this argument is optional, and if we don't specify it, the default value will be FALSE. If we don't want the default value, we must name the argument in our call:

```
> testscores <- read.table("exams",header=TRUE)
```

Hence the terminology *named argument*.

Return Values
The return value of a function can be any R object. Although the return value is often a list, it could even be another function. You can transmit a value back to the caller by explicitly calling return(). Without this call, the value of the last executed statement will be returned by default.

```
> oddcount
function(x) {
k <- 0 # assign 0 to k
for (n in x) {
if (n %% 2 == 1) k <- k+1 # %% is the modulo operator
}
return(k)
}
```

This function returns the count of odd numbers in the argument. We could slightly simplify the code by eliminating the call to return(). To do this, we evaluate the expression to be returned, k, as our last statement in the code:

```
oddcount <- function(x) {
k <- 0
pagebreak
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
```

www.JntukMaterials.com

```
k
}
```

On the other hand, consider this code:

```
oddcount <- function(x) {
k <- 0

for (n in x) {
if (n %% 2 == 1) k <- k+1
}
}
```

It wouldn't work, for a rather subtle reason: The last executed statement here is the call to for(), which returns the value NULL (and does so, in R parlance, *invisibly*, meaning that it is discarded if not stored by assignment). Thus, there would be no return value at all.

7.4.1 Deciding Whether to Explicitly Call return()

The prevailing R idiom is to avoid explicit calls to return(). One of the reasons cited for this approach is that calling that function lengthens execution time. However, unless the function is very short, the time saved is negligible,
so this might not be the most compelling reason to refrain from using return(). But it usually isn't needed nonetheless.

Consider our second example from the preceding section:

```
oddcount <- function(x) {
k <- 0
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
k
}
```

Here, we simply ended with a statement listing the expression to be returned—in this case, k. A call to return() wasn't necessary. Code in this book usually does include a call to return(), for clarity for beginners, but it is customary to omit it.
Good software design, however, should be mean that you can glance through a function's code and immediately spot the various points at which control is returned to the caller. The easiest way to accomplish this is to use an explicit return() call in all lines in the middle of the code that cause a return. (You can still omit a return() call at the end of the function if
you wish.)

Returning Complex Objects

Since the return value can be any R object, you can return complex objects. Here is an example of a function being returned:

```
> g
function() {
t <- function(x) return(x^2)
return(t)
}
> g()
function(x) return(x^2)
<environment: 0x8aafbc0>
```

If your function has multiple return values, place them in a list or other container.

www.JntukMaterials.com

Functions Are Objects

R functions are *first-class objects* (of the class "function", of course), meaning that they can be used for the most part just like other objects. This is seen in the syntax of function creation:

```
> g <- function(x) {
+ return(x+1)
+ }
```

Here, function() is a built-in R function whose job is to create functions! On the right-hand side, there are really two arguments to function(): The first is the formal argument list for the function we're creating—here, just x—and the second is the body of that function—here, just the single statement return(x+1). That second argument must be of class "expression". So, the point is that the right-hand side creates a function object, which is then assigned to g.

By the way, even the "{" is a function, as you can verify by typing this:

```
> ?"{"
```

Its job is the make a single unit of what could be several statements.

These two arguments to function() can later be accessed via the R functions formals() and body(), as follows:

```
> formals(g)
$x
> body(g)
{
return(x + 1)
}
```

Recall that when using R in interactive mode, simply typing the name of an object results in printing that object to the screen. Functions are no exception, since they are objects just like anything else.

```
> g
function(x) {
return(x+1)
```

This is handy if you're using a function that you wrote but which you've forgotten the details of. Printing out a function is also useful if you are not quite sure what an R library function does. By looking at the code, you may understand it better. For example, if you are not sure as to the exact behavior of the graphics function abline(), you could browse through its code to better understand how to use it.

```
> abline
function (a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
coef = NULL, untf = FALSE, ...)
{
int_abline <- function(a, b, h, v, untf, col = par("col"),
lty = par("lty"), lwd = par("lwd"), ...) .Internal(abline(a,
b, h, v, untf, col, lty, lwd, ...))
if (!is.null(reg)) {
if (!is.null(a))
warning("'a' is overridden by 'reg'")
a <- reg
}
if (is.object(a) || is.list(a)) {
p <- length(coefa <- as.vector(coef(a)))
```

www.JntukMaterials.com
...
...
If you wish to view a lengthy function in this way, run it through page():

```
> page(abline)
```

Note, though, that some of R's most fundamental built-in functions are written directly in C, and thus they are not viewable in this manner. Here's an example:

```
> sum
function (..., na.rm = FALSE) .Primitive("sum")
```

Since functions are objects, you can also assign them, use them as arguments to other functions, and so on.

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)
> f <- f1
> f(3,2)
[1] 5
> f <- f2}

> f(3,2)
[1] 1
> g <- function(h,a,b) h(a,b)
> g(f1,3,2)
[1] 5
> g(f2,3,2)
[1] 1
```

And since functions are objects, you can loop through a list consisting of several functions. This would be useful, for instance, if you wished to write a loop to plot a number of functions on the same graph, as follows:

```
> g1 <- function(x) return(sin(x))
> g2 <- function(x) return(sqrt(x^2+1))
> g3 <- function(x) return(2*x-1)
> plot(c(0,1),c(-1,1.5)) # prepare the graph, specifying X and Y ranges
> for (f in c(g1,g2,g3)) plot(f,0,1,add=T) # add plot to existing graph
```

The functions formals() and body() can even be used as replacement functions. We'll discuss replacement functions in Section 7.10, but for now, consider how you could change the body of a function by assignment:

```
> g <- function(h,a,b) h(a,b)
> body(g) <- quote(2*x + 3)
> g
function (x)
2 * x + 3
> g(3)
[1] 9
```

The reason quote() was needed is that technically, the body of a function has the class "call", which is the class produced by quote(). Without the call to quote(), R would try to evaluate the quantity 2*x+3. So if x had been defined and equal to 3, for example, we would assign 9 to the body of g(), certainly not what we want. By the way, since * and + are functions (as discussed in Section 2.4.1), as a language object, 2*x+3 is indeed a call—in fact, it is one function call nested within another.

www.JntukMaterials.com

Environment and Scope Issues

A function—formally referred to as a *closure* in the R documentation— consists not only of its arguments and body but also of its *environment*. The latter is made up of the collection of objects present at the time the function is created. An understanding of how environments work in R is essential for writing effective R functions.

The Top-Level Environment

Consider this example:

```
> w <- 12
> f <- function(y) {
+ d <- 8
+ h <- function() {
+ return(d*(w+y))
+ }
+ return(h())
+ }
> environment(f)
<environment: R_GlobalEnv>
```

Here, the function f() is created at the *top level*—that is, at the interpreter command prompt—and thus has the top-level environment, which in R output is referred to as R_GlobalEnv but which confusingly you refer to in R code as .GlobalEnv. If you run an R program as a batch file, that is considered top level, too.

The function ls() lists the objects of an environment. If you call it at the top level, you get the top-level environment. Let's try it with our example

code:

```
> ls()
[1] "f" "w"
```

As you can see, the top-level environment here includes the variable w, which is actually used within f(). Note that f() is here too, as functions are indeed objects and we did create it at the top level. At levels other than the top, ls() works a little differently,

You get a bit more information from ls.str():

```
> ls.str()
f : function (y)
w : num 12
```

Next, we'll look at how w and other variables come into play within f().

The Scope Hierarchy

Let's first get an intuitive overview of how scope works in R and then relate it to environments. If we were working with the C language (as usual, background in C is not assumed), we would say that the variable w in the previous section is *global* to f(), while d is *local* to f(). Things are similar in R, but R is more hierarchical.

In C, we would not have functions defined within functions, as we have with h() inside f() in our example. Yet, since functions are objects, it is possible—and sometimes desirable from the point of view of the encapsulation goal of object-oriented programming—to define a function within a function; we are simply creating an object, which we can do anywhere.

Here, we have h() being local to f(), just like d. In such a situation, it makes sense for scope to be hierarchical. Thus, R is set up so that d, which is local to f(), is in turn global to h(). The same is true for y, as arguments are considered locals in R.

Similarly, the hierarchical nature of scope implies that since w is global to f(), it is global to h() as well. Indeed, we do use w within h(). In terms of environments then, h()'s environment consists of whatever objects are defined at the time h() comes into existence; that is, at the time

www.JntukMaterials.com
that this assignment is executed:

```
h <- function() {
return(d*(w+y))
}
```

(If f() is called multiple times, h() will come into existence multiple times, going out of existence each time f() returns.) What, then, will be in h()'s environment? Well, at the time h() is created, there are the objects d and y created within f(), *plus* f()'s environment (w). In other words, if one function is defined within another, then that inner function's environment consists of the environment of the outer one, plus whatever locals have been created so far within the outer one. With multiple nesting of functions, you have a nested sequence of larger and larger environments,with the "root" consisting of the top-level objects.

Let's try out the code:

```
> f(2)
[1] 112
```

What happened? The call f(2) resulted in setting the local variable d to 8, followed by the call h(). The latter evaluated d*(w+y)—that is, 8*(12+2)—giving us 112.

Note carefully the role of w. The R interpreter found that there was no local variable of that name, so it ascended to the next higher level—in this case, the top level—where it found a variable w with value 12.

Keep in mind that h() is local to f() and invisible at the top level.

```
> h
Error: object 'h' not found
```

It's possible (though not desirable) to deliberately allow name conflicts in this hierarchy. In our example, for instance, we could have a local variable d within h(), conflicting with the one in f(). In such a situation, the innermost environment is used first. In this case, a reference to d within h() would refer to h()'s d, not f()'s.

Environments created by inheritance in this manner are generally referred to by their memory locations. Here is what happened after adding a print statement to f() (using edit(), not shown here) and then running the code:

```
> f
function(y) {
d <- 8
h <- function() {
return(d*(w+y))
}
print(environment(h))
return(h())
}
> f(2)
<environment: 0x875753c>
[1] 112
```

Compare all this to the situation in which the functions are not nested:

```
> f
function(y) {
d <- 8
return(h())
}
> h
function() {
return(d*(w+y))
}
```

The result is as follows:

www.JntukMaterials.com

```
> f(5)
Error in h() : object 'd' not found
```

This does not work, as d is no longer in the environment of h(), because h() is defined at the top level. Thus, an error is generated. Worse, if by happenstance there had been some unrelated variable d in the top-level environment, we would not get an error message but instead would have incorrect results.

You might wonder why R didn't complain about the lack of y in the alternate definition of h() in the preceding example. As mentioned earlier, R doesn't evaluate a variable until it needs it under a policy called lazy evaluation. In this case, R had already encountered an error with d and thus never got to the point where it would try to evaluate y.

The fix is to pass d and y as arguments:

```
> f
function(y) {
d <- 8
return(h(d,y))
}
> h
function(dee,yyy) {
return(dee*(w+yyy))
}
> f(2)
[1] 88
```

Okay, let's look at one last variation:

```
> f
function(y,ftn) {
d <- 8
print(environment(ftn))
return(ftn(d,y))
}
> h
function(dee,yyy) {
return(dee*(w+yyy))
}
> w <- 12
> f(3,h)
<environment: R_GlobalEnv>
[1] 120
```

When f() executed, the formal argument ftn was matched by the actual argument h. Since arguments are treated as locals, you might guess that ftn could have a different environment than top level. But as discussed, a closure includes environment, and thus ftn has h's environment.

More on ls() :

Without arguments, a call to ls() from within a function returns the names of the current local variables (including arguments). With the envir argument, it will print the names of the locals of any frame in the call chain

Here's an example:

```
> f
function(y) {
d <- 8
return(h(d,y))
```

www.JntukMaterials.com

```
}
> h
function(dee,yyy) {
print(ls())
print(ls(envir=parent.frame(n=1)))
return(dee*(w+yyy))
}
> f(2)
[1] "dee" "yyy"
[1] "d" "y"
[1] 112
```

With parent.frame(), the argument n specifies how many frames to go up in the call chain. Here, we were in the midst of executing h(), which had been called from f(), so specifying n = 1 gives us f()'s frame, and thus we get its locals.

7.6.4 Functions Have (Almost) No Side Effects

Yet another influence of the functional programming philosophy is that functions do not change nonlocal variables; that is, generally, there are no *side effects*. Roughly speaking, the code in a function has read access to its nonlocal variables, but it does not have write access to them. Our code can appear to reassign those variables, but the action will affect only copies, not the variables themselves. Let's demonstrate this by adding some more code to our previous example.

```
> w <- 12
> f
function(y) {
d <- 8
w <- w + 1
y <- y - 2
print(w)
h <- function() {
return(d*(w+y))
}
return(h())
}

> t <- 4
> f(t)
[1] 13
[1] 120
> w
[1] 12
> t
[1] 4
```

So, w at the top level did not change, even though it appeared to change within f(). Only a local *copy* of w, within f(), changed. Similarly, the top-level variable t didn't change, even though its associated formal argument y did change.

Extended Example: A Function to Display the Contents of a Call Frame

In single-stepping through your code in a debugging setting, you often want to know the values of the local variables in your current function. You may also want to know the values of the locals in the parent function—that is, the one from which the current function was called. Here, we will develop code to display these values, thereby further demonstrating access to the environment hierarchy. (The code is adapted from my edtdbg debugging tool in R's CRAN code repository.)

For example, consider the following code:

www.JntukMaterials.com

```
f <- function() {
a <- 1
return(g(a)+a)
}
g <- function(aa) {
b <- 2
aab <- h(aa+b)
return(aab)
}
h <- function(aaa) {
c <- 3
return(aaa+c)
}
```

When we call f(), it in turn calls g(), which then calls h(). In the debugging setting, say we are currently about to execute the return() within g(). We want to know the values of the local variables of the current function, say the variables aa, b, and aab. And while we're in g(), we also wish to know the values of the locals in f() at the time of the call to g(), as well as the values of the global variables. Our function showframe() will do all this. The showframe() function has one argument, upn, which is the number of frames to go up the call stack. A negative value of the argument signals that we want to view the globals—the top-level variables. Here's the code:

```
# shows the values of the local variables (including arguments) of the
# frame upn frames above the one from which showframe() is called; if
# upn < 0, the globals are shown; function objects are not shown
showframe <- function(upn) {
# determine the proper environment
if (upn < 0) {
env <- .GlobalEnv
} else {
env <- parent.frame(n=upn+1)
}
# get the list of variable names
vars <- ls(envir=env)
# for each variable name, print its value
for (vr in vars) {
vrg <- get(vr,envir=env)
if (!is.function(vrg)) {
cat(vr,":\n",sep="")
print(vrg)
}
}
}
```

Let's try it out. Insert some calls into g():

```
> g
function(aa) {
b <- 2
showframe(0)
showframe(1)
aab <- h(aa+b)
return(aab)
```

www.JntukMaterials.com
}
Now run it:
> f()
aa:
[1] 1
b:
[1] 2
a:
[1] 1
To see how this works, we'll first look at the get() function, one of the most useful utilities in R. Its job is quite simple: Given the name of an object, it fetches the object itself. Here's an example:

> m <- rbind(1:3,20:22)
> m
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 20 21 22
> get("m")
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 20 21 22
This example with m involves the current call frame, but in our showframe() function, we deal with various levels in the environment hierarchy. So, we need to specify the level via the envir argument of get():

vrg <- get(vr,envir=env)
The level itself is determined largely by calling parent.frame():
if (upn < 0) {
env <- .GlobalEnv
} else {
env <- parent.frame(n=upn+1)
}
Note that ls() can also be called in the context of a particular level, thus enabling you to determine which variables exist at the level of interest and then inspect them. Here's an example:

vars <- ls(envir=env)
for (vr in vars) {
This code picks up the names of all the local variables in the given frame and then loops through them, setting things up for get() to do its work.

No Pointers in R
R does not have variables corresponding to *pointers* or *references* like those of, say, the C language. This can make programming more difficult in some cases. (As of this writing, the current version of R has an experimental feature
called *reference classes*, which may reduce the difficulty.)

For example, you cannot write a function that directly changes its arguments.
In Python, for instance, you can do this:
>>> x = [13,5,12]
>>> x.sort()
>>> x
[5, 12, 13]

www.JntukMaterials.com
Here, the value of x, the argument to sort(), changed. By contrast, here's how it works in R:

```
> x <- c(13,5,12)
> sort(x)
[1] 5 12 13
> x
[1] 13 5 12
```

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

```
> x <- sort(x)
> x
[1] 5 12 13
```

What if our function has several variables of output? A solution is to gather them together into a list, call the function with this list as an argument, have the function return the list, and then reassign to the original list. An example is the following function, which determines the indices of odd and even numbers in a vector of integers:

```
> oddsevens
function(v){
odds <- which(v %% 2 == 1)
evens <- which(v %% 2 == 1)
list(o=odds,e=evens)
}
```

In general, our function f() changes variables x and y. We might store them in a list lxy, which would then be our argument to f(). The code, both called and calling, might have a pattern like this:

```
f <- function(lxxyy) {
...
lxxyy$x <- ...
lxxyy$y <- ...
return(lxxyy)
}

# set x and y
lxy$x <- ...
lxy$y <- ...
lxy <- f(lxy)
# use new x and y
... <- lxy$x
... <- lxy$y
```

However, this may become unwieldy if your function will change many variables. It can be especially awkward if your variables, say x and y in the example, are themselves lists, resulting in a return value consisting of lists within a list. This can still be handled, but it makes the code more syntactically complex and harder to read. Alternatives include the use of global variables, which we will look at in Section 7.8.4, and the new R reference classes mentioned earlier.

Another class of applications in which lack of pointers causes difficulties is that of treelike data structures. C code normally makes heavy use of pointers for these kinds of structures. One solution for R is to revert to what was done in the "good old days" before C, when programmers formed their own "pointers" as vector indices.

www.JntukMaterials.com

Writing Upstairs

As mentioned earlier, code that exists at a certain level of the environment hierarchy has at least read access to all the variables at the levels above it. On the other hand, direct write access to variables at higher levels via the standard <- operator is not possible.

If you wish to write to a global variable—or more generally, to any variable higher in the environment hierarchy than the level at which your write statement exists—you can use the superassignment operator, <<-, or the assign() function. Let's discuss the superassignment operator first.

7.8.1 Writing to Nonlocals with the Superassignment Operator

Consider the following code:

```
> two <- function(u) {
+ u <<- 2*u
+ z <- 2*z
+ }
> x <- 1
> z <- 3
> u
Error: object "u" not found
> two(x)
> x
[1] 1
> z
[1] 3
> u
[1] 2
```

Let's look at the impact (or not) on the three top-level variables x, z, and u:

- x: Even though x was the actual argument to two() in the example, it retained the value 1 after the call. This is because its value 1 was copied to the formal argument u, which is treated as a local variable within the function. Thus, when u changed, x did not change with it.
- z: The two z values are entirely unrelated to each other—one is top level, and the other is local to two(). The change in the local variable has no effect on the global variable. Of course, having two variables with the same name is probably not good programming practice.
- u: The u value did not even exist as a top-level variable prior to our calling two(), hence the "not found" error message. However, it was created as a top-level variable by the superassignment operator within two(), as confirmed after the call.

Though <<- is typically used to write to top-level variables, as in our example, technically, it does something a bit different. Use of this operator to write to a variable w will result in a search up the environment hierarchy, stopping at the first level at which a variable of that name is encountered. If none is found, then the level selected will be global. Look what happens in this little example:

```
> f
function() {
inc <- function() {x <<- x + 1}
x <- 3
inc()
return(x)
}
> f()
[1] 4
> x
Error: object 'x' not found
```

www.IntukMaterials.com
Here, inc() is defined within f(). When inc() is executing, and the R interpreter sees a superassignment to x, it starts going up the hierarchy. At the first level up—the environment within f()—it does find an x, and so that x is the one that is written to, not x at the top level.

Writing to Nonlocals with assign()

You can also use the assign() function to write to upper-level variables. Here's an altered version of the previous example:

```
> two
function(u) {
assign("u",2*u,pos=.GlobalEnv)
z <- 2*z
}
> two(x)
> x
[1] 1
> u
[1] 2
```

Here, we replaced the superassignment operator with a call to assign(). That call instructs R to assign the value 2*u (this is the local u) to a variable u further up the call stack, specifically in the top-level environment. In this case, that environment is only one call level higher, but if we had a chain of calls, it could be much further up.

The fact that you reference variables using character strings in assign() can come in handy. Recall the example in Chapter 5 concerning analysis of hiring patterns of various large corporations. We wanted to form a subdata frame for each firm, extracted from the overall data frame, all2006. For instance, consider this call:

```
makecorpdfs(c("MICROSOFT CORPORATION","ms","INTEL CORPORATION","intel","
SUN MICROSYSTEMS, INC.","sun","GOOGLE INC.","google")
```

This would first extract all Microsoft records from the overall data frame, naming the resulting subdata frame ms2006. It would then create intel2006 for Intel, and so on. Here is the code (changed to function form, for clarity):

```
makecorpdfs <- function(corplist) {
for (i in 1:(length(corplist)/2)) {
corp <- corplist[2*i-1]
newdtf <- paste(corplist[2*i],"2006",sep="")
assign(newdtf,makecorp(corp),pos=.GlobalEnv)
}
}
```

In the iteration i = 1, the code uses paste() to splice together the strings "ms" and "2006", resulting in "ms2006", the desired name.

When Should You Use Global Variables?

Use of global variables is a subject of controversy in the programming community. Here, the term *global variable*, or just *global*, will be used to include any variable located higher in the environment hierarchy than the level of the given code of interest. The use of global variables in R is more common than you may have guessed. You might be surprised to learn that R itself makes very substantial use of globals internally, both in its C code and in its R routines. The superassignment operator <<-, for instance, is used in many of the R library functions (albeit typically in writing to a variable just one level up in the environment hierarchy). *Threaded* code and *GPU* code, which are used for writing fast programs (as described in Chapter 16), tend to make heavy use of global variables, which provide the main avenue of communication between parallel actors.

www.IntukMaterials.com
Now, to make our discussion concrete, let's return to the earlier example
from Section 7.7:

```
f <- function(lxxyy) { # lxxyy is a list containing x and y
...
lxxyy$x <- ...
lxxyy$y <- ...
return(lxxyy)
}
R
# set x and y
lxy$x <- ...
lxy$y <- ...
lxy <- f(lxy)
# use new x and y
... <- lxy$x
... <- lxy$y
```

As noted earlier, this code might be a bit unwieldy, especially if x and y
are themselves lists.
By contrast, here is an alternate pattern that uses globals:

```
f <- function() {
...
x <<- ...
y <<- ...
}
# set x and y
x <- ...
y <- ...
f() # x and y are changed in here
# use new x and y
... <- x
... <- y
```

We chose to use globals, rather than to return lists, in the DES code earlier in this chapter. Let's explore that example further. We had two global variables (both lists, encapsulating various information): sim, associated with the library code, and mm1glbls, associated with our M/M/1 application code. Let's consider sim first. Even many programmers who have reservations about using globals agree that such variables may be justified if they are truly global, in the sense that they are used broadly in the program. This is the case for sim in our DES example. It is used both in the library code (in schedevnt(), getnextevnt(), and dosim()) and in in our M/M/1 application code (in mm1reactevnt()).
So, using sim as a global seems justified. Nevertheless, if we were bound and determined to avoid using globals, we could have placed sim as a local within dosim(). This function would then pass sim as an argument to all of the functions mentioned in the previous paragraph (schedevnt(), getnextevnt(), and so on), and each of these functions would return the modified sim.
for example, would change from this: reactevnt(head) to this: sim <- reactevnt(head)
We would then need to add a line like the following to our applicationspecific
function mm1reactevnt():

```
return(sim)
```

We could do something similar with mm1glbls, placing a variable called, say, appvars as a local within dosim(). However, if we did this with sim as well, we would need to place them together in a list so that both would be returned, as in our earlier example function f().

www.JntukMaterials.com
On the other hand, critics of the use of global variables counter that the simplicity of the code comes at a price. They worry that it may be difficult during debugging to track down locations at which a global variable changes value, since such a change could occur anywhere in the program.
Another concern raised by critics involves situations in which a function is called in several unrelated parts of the overall program using different values. For example, consider using our example f() function in different parts of our program, each call with its own values of x and y, rather than just a single value of each, as assumed earlier. This could be solved by setting up vectors of x and y values, with one element for each instance of f() in your program.
Within dosim(), the line
sim <<- list()
would be replaced by
assign("simenv",new.env(),envir=.GlobalEnv)
This would create a new environment, pointed to by simenv at the top level. It would serve as a package in which to encapsulate our globals. We would access them via get() and assign(). For instance, the lines

if (is.null(sim$evnts)) {
sim$evnts <<- newevnt
in schedevnt() would become
if (is.null(get("evnts",envir=simenv))) {
assign("evnts",newevnt,envir=simenv)

Yes, this is cluttered too, but at least it is not complex like lists of lists of lists. And it does protect against unwittingly writing to an unrelated variable the user has at the top level. Using the superassignment operator still yields the least cluttered code, but this compromise is worth considering

Recursion
A *recursive* function calls itself. If you have not encountered this concept before, it may sound odd, but the idea is actually simple. In rough terms, the idea is this:
To solve a problem of type X by writing a recursive function f():
1. Break the original problem of type X into one or more smaller problems of type X.
2. Within f(), call f() on each of the smaller problems.
3. Within f(), piece together the results of (b) to solve the original problem.

A Quicksort Implementation
A classic example is Quicksort, an algorithm used to sort a vector of numbers from smallest to largest. For instance, suppose we wish to sort the vector (5,4,12,13,3,8,88). We first compare everything to the first element, 5, to form two subvectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us subvectors (4,3) and (12,13,8,88).
We then call the function on the subvectors, returning (3,4) and (8,12,13,88). We string those together with the 5,
yielding (3,4,5,8,12,13,88), as desired.
R's vector-filtering capability and its c() function make implementation of Quicksort quite easy.
NOTE *This example is for the purpose of demonstrating recursion. R's own sort function, sort(), is much faster, as it is written in C.*

```
qs <- function(x) {
if (length(x) <= 1) return(x)
pivot <- x[1]
therest <- x[-1]
sv1 <- therest[therest < pivot]
sv2 <- therest[therest >= pivot]
sv1 <- qs(sv1)
sv2 <- qs(sv2)
```

www.JntukMaterials.com

```
return(c(sv1,pivot,sv2))
}
```

Note carefully the *termination condition*:
if (length(x) <= 1) return(x)
Without this, the function would keep calling itself repeatedly on empty vectors, executing forever. (Actually, the R interpreter would eventually refuse to go any further, but you get the idea.)
Sounds like magic? Recursion certainly is an elegant way to solve many problems. But recursion has two potential drawbacks:
• It's fairly abstract. I knew that the graduate student, as a fine mathematician, would take to recursion like a fish to water, because recursion is really just the inverse of proof by mathematical induction. But many programmers find it tough.
• Recursion is very lavish in its use of memory, which may be an issue in R if applied to large problems.

Quick sort another example

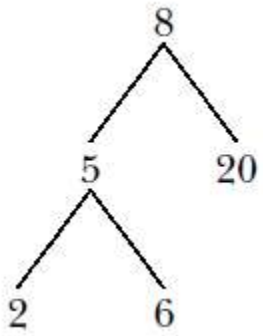
```
quicksort <- function(x) {
  if(length(x) <= 1) return(x)

  pivot      <- x[1]
  remainder  <- x[-1]
  remainder_1 <- remainder[remainder < pivot]
  remainder_2 <- remainder[remainder >= pivot]

  remainder_1 <- quicksort(remainder_1)
  remainder_2 <- quicksort(remainder_2)

  return(c(remainder_1, pivot, remainder_2))
}

if (require("testthat")) {
  test_that("quicksort function sorts properly", {
    x <- c(1, 6, 2, 10, -9, 12, 4, -12)
    y <- c(-12, -9, 1, 2, 4, 6, 10, 12)
    expect_that(quicksort(x), equals(y))
    expect_that(quicksort(x), equals(sort(x)))
  })
}
```

Extended Example: A Binary Search Tree
Treelike data structures are common in both computer science and statistics. In R, for example, the rpart library for a recursive portioning approach to regression and classification is very popular. Trees obviously have applications
in genealogy, and more generally, graphs form the basis of analysis of social networks.

For the sake of simplicity, our example here will be a binary search tree, a classic computer science data structure that has the following property: In each node of the tree, the value at the left link, if any, is less than or equal to that of the parent, while the value at the right link, if any, is greater than that of the parent.

www.JntukMaterials.com

Here is an example:



We've stored 8 in the *root*—that is, the head—of the tree. Its two child nodes contain 5 and 20, and the former itself has two child nodes, which store 2 and 6.
Note that the nature of binary search trees implies that at any node, all of the elements in the node's left subtree are less than or equal to the value stored in this node, while the right subtree stores the elements that are larger than the value in this mode. In our example tree, where the root node contains 8, all of the values in the left subtree—5, 2 and 6—are less
than 8, while 20 is greater than 8.

If implemented in C, a tree node would be represented by a C struct, similar to an R list, whose contents are the stored value, a pointer to the left child, and a pointer to the right child. But since R lacks pointer variables,

Before discussing the code, let's run through a quick session of tree building using its routines.

```
> x <- newtree(8,3)
> x
$mat
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NA NA
[3,] NA NA NA
$nxt
[1] 2
$inc
[1] 3
> x <- ins(1,x,5)
> x
$mat

[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NA NA
$nxt
[1] 3
$inc
[1] 3
> x <- ins(1,x,6)
> x
$mat
```

www.JntukMaterials.com

```
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA 3 5
[3,] NA NA 6
$nxt
[1] 4
$inc
[1] 3
> x <- ins(1,x,2)
> x
$mat
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA NA
[6,] NA NA NA
$nxt
[1] 5
$inc
[1] 3
> x <- ins(1,x,20)
> x
$mat
[,1] [,2] [,3]

[1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA 20
[6,] NA NA NA
$nxt
[1] 6
$inc
[1] 3
```

What happened here? First, the command containing our call newtree(8,3) creates a new tree, assigned to x, storing the number 8. The argument 3 specifies that we allocate storage room three rows at a time.

The result is that the matrix component of the list x is now as follows:

```
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NA NA
[3,] NA NA NA
```

Three rows of storage are indeed allocated, and our data now consists just of the number 8. The two NA values in that first row indicate that this node of the tree currently has no children.

We then make the call ins(1,x,5) to insert a second value, 5, into the tree x. The argument 1 specifies the root. In other words, the call says,

www.JntukMaterials.com
"Insert 5 in the subtree of x whose root is in row 1." Note that we need to reassign the return value of this call back to x. Again, this is due to the lack of pointer variables in R. The matrix now looks like this:

```
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NA NA
```

The element 2 means that the left link out of the node containing 8 is meant to point to row 2, where our new element 5 is stored.

The session continues in this manner. Note that when our initial allotment of three rows is full, ins() allocates three new rows, for a total of six. In the end, the matrix is as follows:

```
[,1] [,2] [,3]
[1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6

[4,] NA NA 2
[5,] NA NA 20
[6,] NA NA NA
```

This represents the tree we graphed for this example.

The code follows. Note that it includes only routines to insert new items and to traverse the tree. The code for deleting a node is somewhat more complex, but it follows a similar pattern

```
# routines to create trees and insert items into them are included
# below; a deletion routine is left to the reader as an exercise

# storage is in a matrix, say m, one row per node of the tree; if row
# i contains (u,v,w), then node i stores the value w, and has left and
# right links to rows u and v; null links have the value NA

# the tree is represented as a list (mat,nxt,inc), where mat is the
# matrix, nxt is the next empty row to be used, and inc is the number of
# rows of expansion to be allocated whenever the matrix becomes full

# print sorted tree via in-order traversal
printtree <- function(hdidx,tr) {
left <- tr$mat[hdidx,1]
if (!is.na(left)) printtree(left,tr)
print(tr$mat[hdidx,3]) # print root
right <- tr$mat[hdidx,2]
if (!is.na(right)) printtree(right,tr)
}

# initializes a storage matrix, with initial stored value firstval
newtree <- function(firstval,inc) {
m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
m[1,3] <- firstval
return(list(mat=m,nxt=2,inc=inc))
}

# inserts newval into the subtree of tr, with the subtree's root being
# at index hdidx; note that return value must be reassigned to tr by the
```

www.JntukMaterials.com

```
30 # caller (including ins() itself, due to recursion)
31 ins <- function(hdidx,tr,newval) {
32 # which direction will this new node go, left or right?
33 dir <- if (newval <= tr$mat[hdidx,3]) 1 else 2
34 # if null link in that direction, place the new node here, otherwise
35 # recurse
36 if (is.na(tr$mat[hdidx,dir])) {
37 newidx <- tr$nxt # where new node goes
38 # check for room to add a new element
39 if (tr$nxt == nrow(tr$mat) + 1) {
tr$mat <-
41 rbind(tr$mat, matrix(rep(NA,tr$inc*3),nrow=tr$inc,ncol=3))
42 }
43 # insert new tree node
44 tr$mat[newidx,3] <- newval
45 # link to the new node
46 tr$mat[hdidx,dir] <- newidx
47 tr$nxt <- tr$nxt + 1 # ready for next insert
48 return(tr)
49 } else tr <- ins(tr$mat[hdidx,dir],tr,newval)
50 }
```

There is recursion in both printtree() and ins(). The former is definitely the easier of the two, so let's look at that first. It prints out the tree, in sorted order. Recall our description of a recursive function f() that solves a problem of category X: We have f() split the original X problem into one or more smaller X problems, call f() on them, and combine the results. In this case, our problem's category X is to print a tree, which could be a subtree of a larger one. The role of the function on line 13 is to print the given tree, which it does by calling itself in lines 15 and 18. This mechanism is seen in the if() statements in lines 15 and 18. When we come to a null link, we do not continue to recurse. The recursion in ins() follows the same principles but is considerably more delicate. Here, our "category X" is an insertion of a value into a subtree.