Chapter 5

Objects

Everything in R is an object; that is, a dynamically created, self-describing container for data. This chapter presents techniques for managing objects. Section 5.1 introduces the fundamental reference technique: assigning a name in an environment. Section 5.2, page 115, discusses the replacement operation, by which assigned objects are modified. Section 5.3, page 119, discusses the environments, in which objects are assigned. R allows assignments to nonlocal environments, discussed in Section 5.4, page 125, and including the technique known as *closures*. The final two sections discuss the transfer of R data and objects to and from external media: Section 5.5, page 131, describes connections, the R technique for dealing with an external medium; Section 5.6, page 135, covers the techniques for transferring data and objects.

5.1 Objects, Names, and References

The central computation in R is a function call, defined by the function object itself and the objects that are supplied as the arguments. In the functional programming model, the result is defined by another object, the value of the call. Hence the traditional motto of the S language: *everything is an object*—the arguments, the value, and in fact the function and the call itself: All of these are defined as objects.

Think of objects as collections of data of all kinds. The data contained and the way the data is organized depend on the class from which the object was generated. R provides many classes, both in the basic system and in

various packages. Defining new classes is an important part of programming with R. Chapter 6 discusses existing classes and the functions that compute on them. Chapters 9 and 10 discuss new classes and new functional computational methods. The present chapter explores computations to create and organize objects, regardless of their class or contents. The fundamental dualism in all aspects of R and the S language, the dualism between function calls and objects, is reflected in all these discussions.

As in any programming language, it's essential to be able to refer to objects, in a particular context, in a way that is consistent and clear. In the S language, there is one and only one way to refer to objects: by name. More precisely, the combination of a name (that is, a non-empty character string) and an environment (or context) in which the name is evaluated is the fundamental reference to an object in R. So, the value of the expressions `pi` or `lm` in the global environment, or the value of `x` inside a particular function call, will refer to a specific object (or generate an error, if no corresponding object can be found). The next section elaborates on environments and related ideas: basically, any computation in R takes place in an environment that defines how the evaluator will search for an object by name.

Whenever we talk about a reference to an object, in any language, the key is that we expect to use that reference repeatedly, in the confidence that it continues to refer to the same object. References do usually include the ability to change the object, what is sometimes called a *mutable object reference*, but which in R we can reduce to an assignment. Unless some explicit assignment has occurred, using an object reference means we can be confident that successive computations will see consistent data in the object. It's essentially a sanity requirement for computing: otherwise, there is no way to understand what our computations mean.

A name, with an associated environment, provides a reference in exactly this sense in R, for normal objects and programming style. As for that qualification, "normal", it excludes two kinds of abnormality. R permits some non-standard functions that explicitly reach out to perform non-local assignments. They have their place, and are discussed in section 5.4, but we'll exclude them from the current discussion. In addition, there are some non-standard classes of objects whose behavior also breaks the general model, as discussed beginning on page 114. These too are excluded by the term "normal". (Notice again the duality of functions and objects in the exceptions to normal behavior.)

The reference of a name to an object is made by an assignment, for example:

```
lmFit <- lm(survival ~  ., study2004)
```

This expression creates an object named `lmFit` in the current environment. Having created the object, we can now use it, perhaps to generate some printed or plotted summaries, or to create some further named objects:

```
lmResid <- lmFit$residuals
```

As long as no second assignment for the name `lmFit` took place in the same context, we can be confident that the new object was computed from the `lmFit` object created above—the same object in all respects, regardless of what other computations took place involving `lmFit`.

The assurance of consistency is key for providing clear and valid software. Suppose, between the two assignments you saw an expression such as

```
verySubtleAnalysis(lmFit)
```

Suppose you had no clue what this function was doing internally, except that all its computations are normal in our current sense, and that `lmfit` is a normal object. You can then be quite confident that the intermediate computations will not have modified `lmFit`. Such confidence allows a top-down analysis of the computations, contributing directly to trustworthy software and to our *Prime Directive*.

We said that names are the only general form of reference in R, and that statement is important to understand. In the second assignment above, `lmFit$residuals` extracts a component of the `lmFit` object. To emphasize, the computation extracts the information, as a new object, rather than creating a reference to the portion of `lmFit` that contains this information. If a following computation changes `lmFit`, there will be no change in `lmResid`.

The statement that nearly all object references in R start from assignments needs some elaboration, too. As later sections in this chapter discuss, there are many ways to get access to objects in R: from packages, saved images, and other files. However, these objects were nearly always created by assignments, and then saved in other forms.

The most important objects not created by an assignment are the arguments in a function call. The R evaluator creates an association between the name of the argument and the expression supplied in the actual call. If you are writing a function with an argument named `x`, then inside the function definition, you can use the name `x` and be confident that it refers to the corresponding argument in the call. The mechanism involved is extremely important in the way R works, and is somewhat different from an assignment. Section 13.3, page 460, discusses the details. For the most part,

however, you just use the argument names in the body of the function in the same way as any other names.

Exceptions to the object model

Most classes of objects in R behave according to the model described in this section, but a few do not. You need to be careful in using such objects, because they do not give you the usual safety of knowing that local changes really are local. Three classes of such exceptional objects are connections, environments, and external pointers. The discussion here summarizes how and why these objects are exceptions to the normal object behavior.

Connections: The class of connection objects represents streams of bytes (characters, usually). Files on disc and other data streams that behave similarly can be used in R by creating a connection object that refers to the data stream. See Section 5.5, page 131, for a general discussion of connections.

The connection refers to a data stream that often has some sort of physical reality in the computer; as a result, any computation that uses the connection object will deal with the same data stream. Reading from a connection in one function call will alter the state of the stream (for example, the current position for reading from a file). As a result, computations in other functions will be affected. Connection objects in a function call are not local. Ignoring the non-local aspect of a connection object leads to obvious, but easy-to-make, errors such as the following.

```
wRead <- function (con) {
  w <- scan(con, numeric(), n=1)
  if(w > 0)
    w * scan(con, numeric(), n=1)
  else
    NA
}
```

The function `wread()` is intended to read a weight `w` from connection `con` and then to return either the weight times the following data value on the connection, if the weight is positive, or `NA` otherwise. The danger is that `wRead` sometimes reads one field from the connection, and sometimes two. If connections were ordinary objects (if, say, we were just picking items from a list), the difference would not matter because the effect would be local to the single call to `wRead`. But `con` is a connection. If it contained pairs of numbers, as it likely would, then the first non-positive value of `w` will cause

`wRead` to leave the following field on the connection. From then on, disaster is likely.

The recommended fix, here and in general, is that all computations on a connection should leave the connection in a well-defined, consistent state. Usually that means reading (or writing) a specific sequence of fields. Each function's specification should include a description of what it does to the connection. Unfortunately, most of the base functions dealing with connections are implemented as internal `C` code. Their definition is not easily understood, and different functions can behave inconsistently.

Environments: As discussed in section 5.3, one can access a reference to the environment containing objects as if it were itself an object. In detailed programming tasks, you may need to pass such objects to other functions, so they can search in the right place for a particular name, for example. But environments are not copied or made local to a particular function. Any changes made to the environment will be seen by all software using that environment from now on.

Given that environment objects have this highly non-standard behavior, it might have been better if standard `R` computations were not allowed for them. Unfortunately a number of basic functions do appear to work normally with environments, including replacement functions for components (`"$"`) and attributes (`attr`). Don't be fooled: the effects are very different. Avoid using these replacement functions with environments.

External pointers: These are a much more specialized kind of object, so the temptation to misuse them arises less often. As the name suggests, they point to something external to `R`, or at least something that the `R` evaluator treats that way. As a result, the evaluator does none of the automatic copying or other safeguards applied to normal objects. External pointers are usually supplied from some code, typically written in `C`, and then passed along to other such code. Stick to such passive use of the objects.

For all such non-standard objects, one important current restriction in programming is that they should not be extended by new class definitions. They can, with care, be used as slots in class definitions.

5.2 Replacement Expressions

In discussing names as references, we stated that an object assigned in an environment would only be changed by another assignment. But `R` computations frequently have replacement expressions such as:

```
diag(x) <- diag(x) + epsilon
```

```
z[[i]] <- lowerBound
lmFit$resid[large] <- maxR
```

Don't these modify the objects referred to by `x`, `z` and `lmFit`? No, technically they do not: A replacement creates a new assignment of an object to the current name. The distinction usually makes little difference to a user, but it is the basis for a powerful programming technique and affects computational efficiency, so we should examine it here.

The expressions above are examples of a *replacement expression* in the S language; that is, an assignment where the left side is not a name but an expression, identifying some aspect of the object we want to change. By definition, any replacement expression is evaluated as a simple assignment (or several such assignments, for complex replacement expressions), with the right side of the assignment being a call to a *replacement function* corresponding to the expression. The first example above is equivalent to:

```
x <- `diag<-`(x, value = diag(x) + epsilon)
```

The mechanism is completely general, applying to any function on the left side of the assignment defined to return the modified object. The implication is that a new complete object replaces the existing object each time a replacement expression is evaluated.

It may be important to remember how replacements work when replacing portions of large objects. Each replacement expression evaluates to a new assignment of the complete object, regardless of how small a portion of the object has changed. Sometimes, this matters for efficiency, but as with most such issues, it's wise not to worry prematurely, until you know that the computation in question is important enough for its efficiency to matter. The classic "culprit" is an expression of the form:

```
for(i in undefinedElements(z))
  z[[i]] <- lowerBound
```

The loop in the example will call the function for replacing a single element some number of times, possibly many times, and on each call a new version of `z` will be assigned, or at least that is the model. In this example, there is no doubt that the programmer should have used a computation that is both simpler and more efficient:

```
z[undefinedElements(z)] <- lowerBound
```

In the jargon that has grown up around S-language programming the distinction is often referred to as "vectorizing": the second computation deals with

the whole object (in this case, a vector). Some suggestions and examples are provided in Section 6.4, page 157.

However, as is often the case, predicting the actual effect on efficiency requires considerable knowledge of the details, another reason to delay such considerations in many applications. The example above, in fact, will usually prove to be little more efficient in the vectorized form. The replacement function `` `[[<-` `` is one of a number of basic replacements that are defined as primitives; these can, sometimes, perform a replacement in place. The distinction is relevant for efficiency but does not contradict the general model. Primitive replacement functions generally will modify the object in place, without duplication, if it is local. If so, then no difference to the overall result will occur from modification in place.

As a result, a simple loop over primitive replacements will at most tend to produce one duplicate copy of the object. Even if the object is not local, the first copy made and assigned will be, so later iterations will omit the duplication.

The argument for this particular vectorizing is still convincing, but because the revised code is a clearer statement of the computation. It's also likely to be slightly faster, because it eliminates the setup and execution of some number of function calls. Even this distinction is not likely to be very noticeable because the replacement function is a primitive.

Replacement functions

The ability to write new replacement functions provides an important programming tool. Suppose you want to define an entirely new form of replacement expression, say:

```
undefined(z) <- lowerBound
```

No problem: just define a function named `` `undefined<-` ``. For an existing replacement function, you may often want to define a new replacement method to replace parts of objects from a class you are designing; for example, methods for replacements using `` `[` `` or `` `[[` `` on the left of the assignment. Again, no special mechanism is needed: just define methods for the corresponding replacement function, `` `[<-` `` or `` `[[<-` ``.

To work correctly, replacement functions have two requirements. They must always return the complete object with suitable changes made, and the final argument of the function, corresponding to the replacement data on the right of the assignment, must be named `"value"`.

The second requirement comes because the evaluator always turns a replacement into a call with the right-hand side supplied by name, `value=`, and that convention is used so that replacement functions can have optional arguments. The right-hand side value is never optional, and needs to be supplied by name if other arguments are missing.

Let's define a replacement function for `undefined()`, assuming it wants to replace missing values with the data on the right-hand side. As an extra feature, it takes an optional argument `codes` that can be supplied as one or more numerical values to be interpreted as undefined.

```
`undefined<-` <- function(x, codes = numeric(), value) {
      if(length(codes) > 0)
        x[ x %in% codes] <- NA
      x[is.na(x)] <- value
      x
  }
```

If the optional `codes` are supplied, the `` `%in%` `` operator will set all the elements that match any of the codes to `NA`.

Notice that one implication of the mechanism for evaluating replacement expressions is that replacement functions can be defined whether or not the ordinary function of the same name exists. We have not shown a function `undefined()` and no such function exists in the core packages for `R`. The validity of the replacement function is not affected in any case. However, in a nested replacement, where the first argument is not a simple name, both functions must exist; see Section 13.5, page 466.

Replacement methods

Methods can be written for replacement functions, both for existing functions and for new generic functions. When a class naturally has methods for functions that describe its conceptual structure, it usually should have corresponding methods for replacing the same structure. Methods for `` `[` ``, `` `[[` ``, `length()`, `dim()`, and many other similar functions suggest methods for `` `[<-` ``, `` `[[<-` ``, etc.

New replacement functions can also be made generic. To create a generic function similar to the `` `undefined<-` `` example:

```
setGeneric("undefined<-",
    function(x, ..., value) standardGeneric("undefined<-"),
    useAsDefault = FALSE)
```

The argument, `code`, in the original function was specific to the particular method that function implemented. When turning a function into a generic, it often pays to generalize such arguments into `"..."`.

We chose not to use the previous function as the default method. The original function above was fine for casual use, but the operator `` `%in%` `` calls the `match()` function, which is only defined for vectors. So a slightly better view of the function is as a method when both `x` and `value` inherit from class `"vector"`. A default value of `NULL` for `code` is more natural when we don't assume that `x` contains numeric data.

```
setMethod("undefined<-",
          signature(x="vector", value = "vector"),
          function(x, codes = NULL, value) {
              if(length(codes) > 0)
                  x[x %in% codes] <- NA
              x[is.na(x)] <- value
              x
          })
```

Class `"vector"` is the union of all the vector data types in R: the numeric types plus `"logical"`, `"character"`, `"list"`, and `"raw"`. A method for class `"vector"` needs to be checked against each of these, unless it's obvious that it works for all of them (it was not obvious to me in this case). I leave it as an exercise to verify the answer: it works for all types except `"raw"`, and does work for `"list"`, somewhat surprisingly. A separate method should be defined for class `"raw"`, another exercise.

A convenience function, `setReplaceMethod()`, sets the method from the name of the non-replacement function. It's just a convenience, to hide the addition `"<-"` to the name of the replacement function.

5.3 Environments

An environment consists of two things. First, it is a collection of objects each with an associated name (an arbitrary non-empty character string). Second, an environment contains a reference to another environment, technically called the *enclosure* of that environment, but also referred to as the *parent*, and returned by the function `parent.env()`.

Environments are created by several mechanisms. The global environment contains all the objects assigned there during the session, plus possibly objects created in a few other ways (such as by restoring some saved data).

The environment of a function call contains objects corresponding to the arguments in the function call, plus any objects assigned so far during the evaluation of the call. Environments associated with packages contain the objects exported to the session or, in the package's namespace, the objects visible to functions in the package. Generic functions have environments created specially to store information needed for computations with methods. Environments created explicitly by `new.env()` can contain any objects assigned there by the user.

When the R evaluator looks for an object by name, it looks first in the local environment and then through the successive enclosing environments. The enclosing environment for a function call is the environment of the function. What that is varies with the circumstances (see page 123), but in the ordinary situation of assigning a function definition, it is the environment where the assignment takes place. In particular, for interactive assignments and ordinary source files, it is the global environment.

The chain of enclosing environments for any computation determines what functions and other objects are visible, so you may need to understand how the chaining works, in order to fully understand how computations will work.

In this section we give some details of environments in various contexts, and also discuss some special programming techniques using environments. A general warning applies to these techniques. As mentioned earlier in the chapter, the combination of a name and an environment is the essential object reference in R. But functional programming, which is central to R (section 3.2), generally avoids computing with references. Given that, it's not surprising that computing directly with environments tends to go outside the functional programming model. The techniques may still be useful, but one needs to proceed with caution if the results are to be understandable and trustworthy.

Environments and the R session

An R session always has an associated environment, the *global* environment. An assignment entered by a user in the session creates an object with the corresponding name in the global environment:

```
sAids <- summary(Aids2)
```

Expressions evaluated directly in the session are also evaluated in the global environment. For the expression above, the evaluator needs to find a function named `"summary"` and then, later, an object named `"Aids2"`. As always,

the evaluator looks up objects by name first in the current environment (here the global environment) and then successively in the enclosing or parent environments.

The chain of environments for the session depends on what packages and other environments are attached. The function `search()` returns the names of these environments, traditionally called the "search list" in the S language. It's not a list in the usual sense. The best way of thinking of the search list is as a chain of environments (and thus, conceptually a list).

At the start of a session the search list might look as follows:

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

The global environment comes first. Its enclosing environment is the second environment on the search list, which has the third environment as its parent, and so on. We can see this by calling `parent.env()`:

```
> ev2 <- parent.env(.GlobalEnv); environmentName(ev2)
[1] "package:stats"
> ev3 <- parent.env(ev2); environmentName(ev3)
[1] "package:graphics"
```

(If you wonder why the call to `environmentName()`, it's because the printed version of packages as environments is confusingly messy; `environmentName()` gets us back to the name used by `search()`.)

The arrangement of enclosing environments, whereby each package has the next package in the search list as its parent, exists so that R can follow the original rule of the S language that the evaluator searches for names in the search list elements, in order.

In evaluating `summary(Aids2)`, the evaluator finds the function object `summary` in the `base` package. However, object `"Aids2"` is not found in any of the elements of the search list:

```
> find("summary")
[1] "package:base"
> find("Aids2")
character(0)
```

That object is contained in the package `MASS`. To obtain it, the package must be attached to the search list, or the object must be explicitly extracted from the package. Attaching the package, say by calling `require()`, alters the search list, and therefore the pattern of enclosing environments.

```
> require(MASS)
Loading required package: MASS
[1] TRUE
> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
> ev2 <- parent.env(.GlobalEnv); environmentName(ev2)
[1] "package:MASS"
> ev3 <- parent.env(ev2); environmentName(ev3)
[1] "package:stats"
```

The search by name for objects now looks in the environment for package `MASS` before the previous environments in the search list. If there happened to be a function `summary()` in that package, it would be chosen rather than the function in the `base` package. The function `require()` would have warned the user if attaching the package introduced any name conflicts.

However, possible conflicts between packages are a worry; with the very large number of packages available, some conflicts are inevitable. Package `mgcv` and package `gam` on CRAN both have a function `gam()`. The two functions are similar in purpose but not identical, so one might want to compare their results. To do so, one needs to be explicit about which function is being called. The `` `::` `` operator prepends the name of the package to the function name, so that `mgcv::gam()` and `gam::gam()` refer uniquely to the two functions.

For programming rather than interactive analysis, the problem and the approach are slightly different. If your function calls functions from other packages, you would like to be assured that the intended function is called no matter what other packages might be used in some future session. If the function was loaded into the global environment, say by using `source()`, such assurance is not available. In our previous example, you cannot ensure that a future user has the intended package in the search list, ahead of the unintended one, when you call `gam()`, and similarly for every other function called from a package. The problem remains when your function is in a simple package, because the original R model for package software is basically that of `source`-ing the code in the package when the package is attached. In either case, the environment of the function is the global environment. If a name is encountered in a call to any such function, then by the general rule on page 120, the evaluator searches first in the call, then in the global environment, and then in its enclosing environments. So the object found can change depending on what packages are attached.

Using `:::` on every call is clearly unreasonable, so a more general mechanism is needed to clarify what software your software expects. This is one of the main motivations for introducing the NAMESPACE mechanism for R packages. A "NAMESPACE" file in the package source contains explicit directives declaring what other packages are imported, potentially even what individual objects are imported from those packages. The mechanism implementing the imports can be understood in terms of the current discussion of environments. If the package SoDA had no namespace file, then a function from the package, say binaryRep() would have the global environment as its environment. But SoDA does have a namespace file and:

```
 > environment(binaryRep)
<environment: namespace:SoDA>
```

The namespace environment constructed for the package restricts the visible objects to those in the namespace itself, those explicitly imported, and the base package. To implement this rule, the parent of the package's namespace is an environment containing all the imports; its parent is the base package's namespace.

In most circumstances, the namespace mechanism makes for more trustworthy code, and should be used in serious programming with R. See Section 4.6, page 103 for the techniques needed.

Environments for functions (continued)

Functions are usually created by evaluating an expression of the form:

```
function ( formal arguments ) body
```

As discussed in Section 3.3, page 50, the evaluation creates a function object, defined by its formal arguments, body, and environment. The function is basically just what you see: the same definition always produces the same object, with one important exception. When it is created, the function object gets a reference to the environment in which the defining expression was evaluated. That reference is a built-in property of the function.

If the expression is evaluated at the command level of the session or in a file sourced in from there, the environment is the global environment. This environment is overridden when packages have a namespace, and replaced by the namespace environment. There are two other common situations in programming that generate function environments other than the global environment.

Function definitions can be evaluated inside a call to another function. The general rule applies: the function is given a reference to the environment created for the evaluation of that call. Ordinarily, the environment of the call disappears after the call is complete, whenever storage is cleaned up by a garbage collection.

However, there is an R programming technique that deliberately creates functions that share a more persistent version of such an environment. The goal is usually to go beyond a purely functional approach to programming by sharing other objects, within the same environment, among several functions. The functions can then update the objects, by using non-local assignments.

For a discussion of programming this way, and of alternatives, see Section 5.4, page 125. Software that is used by calling functions from a list of functions (in the style of `z$f(···)`), or that discusses R closures, likely makes use of this mechanism.

The other commonly encountered exception is in *generic* functions (those for which methods are defined). These mainly exist for the purpose of selecting methods, and are created with a special environment, whose enclosure is then the function's usual environment (typically the namespace of the package where the function is defined). The special environment is used to store some information for rapid selection of methods and for other calculations. A few other objects involved in method dispatch, such as methods including a `callNextMethod()`, also have specialized environments to amortize the cost of searches. Unlike package namespaces, the special environments for method dispatch don't change the fundamental rules for finding names. The specialized environments are an implementation detail, and might in principle disappear in later versions of R.

Computing with environment objects

Environments arise mostly in the background when expressions are evaluated, providing the basic mechanism for storing and finding objects. They can themselves be created (by `new.env()`) and used as objects, however. Doing so carries risks because environments are not standard R objects. An environment is a reference. Every computation that modifies the environment changes the same object, unlike the usual functional model for computing with R.

If you do want to use environments directly, consider using the following basic functions to manipulate them, in order to make your programming intentions clear. The functions actually predate environments and R itself,

and form the traditional set of techniques in the S language for manipulating "database" objects. A 1991 Bell Labs technical report [4] proposed them for database classes. Explicit computation with environments often treats them essentially as database objects. For a more modern approach to a database interface to R, see the DBI package, and Section 12.7, page 446.

The five basic computations, in their R form with environments, are:

`assign(x, value, envir =)` Store the object `value` in the environment, as the character string name, `x`.

`get(x, envir =)` Return the object associated with the name from the environment..

`exists(x, envir = )` Test whether an object exists associated with the name.

`objects(envir = )` Return the vector of names for the objects in the environment.

`remove(list = , envir = )` Remove the objects named as `list` from the environment.

The five functions are widely used, but are presented here with somewhat specialized arguments, needed in order to use them consistently with environments. In addition, both functions `get()` and `exists()` should be called with the optional argument `inherits = FALSE`, if you want to search only in the specified environment and not in its enclosures.

If your programming includes defining new classes, it's natural to embed computations with environments in a special class, to clarify the intentions and hide confusing details. Be warned however: You cannot make class `"environment"` a superclass of a new class, such as by `contains = "environment"` in the call to `setClass()`. Because environment objects are references, objects from the new class will actually have the same reference, including all slots and other properties.

You can use an environment as a slot in a new class, provided as always that your computations take account of the environment's non-standard behavior.

5.4 Non-local Assignments; Closures

Many computational objects are naturally thought of as being repeatedly updated as relevant changes occur. Whenever an object represents a summary of an ongoing process, it requires computations to change the object

when new data arrives in the process. Other objects that represent physical or visual "real things" also lend themselves to updating; for example, an object representing a window or other component of a user interface will be updated when some preference or other internal setting is changed.

The S language provides a very general mechanism for updating a local object, via replacement expressions (Section 5.2, page 115).

R introduces an alternative mechanism, in which functions share a common environment and update non-local objects in that environment. The mechanism is inspired by other languages; in particular, it has something in common with reference-based object-oriented programming systems, but it does not use formal class definitions. As such, it departs significantly from a functional programming style. All the same, it does enable some useful computations, so let's examine it, show an example, along with a more functional alternative, and then assess the pros and cons.

The trick is made possible by two techniques: non-local assignments and the environment created by a function call. Any assignment or replacement with the `<-` or `=` operator can be made non-local by using the operator `<<-` instead. The meaning is quite different, however, and also different from the same operator in S-Plus. Consider the assignment:

```
dataBuf <<- numeric(0)
```

The rule for such assignments in R is to search for the name through all the enclosing environments, starting from the environment of the function in which the assignment is evaluated. If an existing object of this name is found, the assignment takes place there; otherwise, the object is assigned in the global environment. This is an unusual rule and can have strange consequences (for example, if the name is first encountered in one of the attached packages, an attempt is made to assign in that package, usually failing because the package environment is locked). The intended use in most cases is that an object will have been initialized with this name in an enclosing environment; the `<<-` operator then updates this object.

The other part of the trick involves assigning one or more functions inside a function call, by evaluating an ordinary definition, but inside another call. The primitive code that evaluates the `function` expression sets the environment of the function object to the environment where the evaluation takes place, in this case the local environment of the call. Because the assignment is local, both function and environment normally disappear when the call is completed, but not if the function is returned as part of the value of the call. In that case, the object returned preserves both the function and its environment. If several functions are included in the object returned, they

all share the same environment. The R programming mechanism referred to as a *closure* uses that environment to keep references to objects that can then be updated by calling functions created and returned from the original function call.

Here is an example that illustrates the idea. Suppose a large quantity of data arrives in a stream over time, and we would like to maintain an estimate of some quantiles of the data stream, without accumulating an arbitrarily large buffer of data. The paper [7] describes a technique, called *Incremental Quantile estimation* (IQ), for doing this: a fixed-size data buffer is used to accumulate data; when the buffer is full, an estimate of the quantiles is made and the data buffer is emptied. When the buffer fills again, the existing quantile estimates are merged with the new data to create a revised estimate. Thus a fixed amount of storage accumulates a running estimate of the quantiles for an arbitrarily large amount of data arriving in batches over time.

Here's an implementation of the updating involved, using closures in R.

```
newIQ <- function(nData = 1000, probs = seq(0, 1, 0.25)) {
   dataBuf <- numeric(0)
   qBuf <- numeric(0)

   addData <- function(newdata) {
       n <- length(newdata);
       if(n + length(dataBuf) > nData)
         recompute(newdata)
       else
         dataBuf <<- c(dataBuf, newdata)
   }

   recompute <- function(newdata = numeric(0)) {
       qBuf <<- doQuantile(qBuf, c(dataBuf, newdata), probs)
       dataBuf <<- numeric(0)
   }

   getq <- function() {
       if(length(dataBuf) > 0)
         recompute()
       qBuf
   }

   list(addData = addData, getQ = getQ)
}
```

Our implementation is trivial and doesn't in fact illustrate the only technically interesting part of the computation, the actual combination of the current quantile estimate with new data using a fixed buffer, but that's not our department; see the reference. We're interested in the programming for updating.

For each separate data stream, a user would create an IQ "object":

```
myData <- newIQ()
```

The actual returned object consists of a list of two functions. Every call to `newIQ()` returns an identical list of functions, except that the environment of the functions is unique to each call, and indeed is the environment created dynamically for that call. The shared environment is the business end of the object. It contains all the local objects, including `dataBuf` and `qBuf`, which act as buffers for data and for estimated quantiles respectively, and also three functions. Whenever data arrives on the stream, a call to one of the functions in the list adds that data to the objects in the environment:

```
> myData$addData(newdata)
```

When the amount of data exceeds the pre-specified maximum buffer size, quantiles are estimated and the function `recompute()`, conveniently stored in the environment, clears the data buffer. Whenever the user wants the current quantile estimate, this is returned by the other function in the list:

```
> quants <- myData$getQ()
```

This returns the internal quantile buffer, first updating that if data is waiting to be included.

Because the computation is characteristic of programming with closures, it is worth examining why it works. The call to `newIQ()` assigns the two buffers, in the environment of the call. That environment is preserved because the functions in the returned list have a reference to it, and therefore garbage collection can't release it.

When the `addData()` function does a non-local assignment of `dataBuf`, it applies the rule on page 126 by looking for an object of that name, and finds one in the function's environment. As a result, it updates `dataBuf` there; similarly, function `recompute()` updates both `dataBuf` and `qBuf` in the same environment. Notice that `recompute()` shares the environment even though it is not a user-callable function and so was not returned as part of the list.

It's helpful to compare the closures implementation to one using replacement functions. In the replacement version, the buffers are contained explicitly in the object returned by `newIQ()` and a replacement function updates

them appropriately, returning the revised object. Here's an implementation similar to the closure version.

```
newIQ <- function(nData = 1000, probs = seq(0, 1, 0.25))
  list(nData = nData, probs = probs,
       dataBuf = numeric(0), qBuf = numeric(0))

`addData<-` <- function(IQ, update = FALSE, value) {
    n <- length(value);
    if(update || (n + length(IQ$dataBuf) > IQ$nData))
      recompute(IQ, value)
    else {
        IQ$dataBuf <- c(IQ$dataBuf, value)
        IQ
    }
}

recompute  <- function(IQ, newdata = numeric(0)) {
    IQ$qBuf <- doQuantile(qBuf, c(IQ$dataBuf, newdata), probs)
    IQ$dataBuf <- numeric(0)
    IQ
}

getq <- function(IQ) {
    if(length(IQ$dataBuf) > 0)
      IQ <- recompute(IQ)
    IQ$qBuf
}
```

This version of `addData()` is a replacement function, with an option to update the quantile estimates unconditionally. The logic of the computation is nearly the same, with the relevant objects now extracted from the `IQ` object, not found in the environment. Typical use would be:

```
> myData <- newIQ()
  .......
> addData(myData) <- newdata
  .......
> getq(myData)
```

The user types apparently similar commands in either case, mainly distinguished by using the `` `$` `` operator to invoke component functions of the IQ object in the closure form, versus an explicit replacement expression in the alternate version. Even the implementations are quite parallel, or at least can be, as we have shown here.

What happens, however, follows a very different concept. Closures create a number of object references (always the same names, but in unique environments), which allow the component functions to alter the object invisibly. The component functions correspond to methods in languages such as C++, where objects are generally *mutable*, that is, they can be changed by methods via object references.

The replacement function form follows standard S-language behavior. General replacement functions have often perplexed those used to other languages, but as noted in section 5.2, they conform to the concept of local assignments in a functional language.

Are there practical distinctions? Closures and other uses of references can be more efficient in memory allocation, but how much that matters may be hard to predict in examples.

The replacement version requires more decisions about keeping the quantile estimates up to date, because only an assignment can change the object. For example, although `getq()` always returns an up-to-date estimate, it cannot modify the non-local object (fortunately for trustworthy software). To avoid extra work in recomputing estimates, the user would need to reassign the object explicitly, for example by:

```
myData <- recompute(myData)
```

Another difference between the versions arises if someone wants to add functionality to the software; say, a summary of the current state of the estimation. The replacement version can be modified in an ordinary way, using the components of any IQ object. But notice that a new function in the closure version must be created by `newIQ()` for it to have access to the actual objects in the created environment. So any changes can only apply to objects created after the change, in contrast to the usual emphasis on gradual improvement in R programming.

Finally, I think both versions of the software want to evolve towards a class-and-method concept. The IQ objects really ought to belong to a class, so that the data involved is well-defined, trustworthy, and open to extension and inheritance. The replacement version could evolve this way obviously; what are currently components of a list really want to be slots in a class.

The closure version could evolve to a class concept also, but only in a class system where the slots are in fact references; again, this has much of the flavor of languages such as C++ or Java.

5.5 Connections

Connection objects and the functions that create them and manipulate them allow R functions to read and interpret data from outside of R, when the data can come from a variety of sources. When an argument to the R function is interpreted as a connection, the function will work essentially the same way whether the data is coming from a local file, a location on the web, or an R character vector. To some extent, the same flexibility is available when an R function wants to write non-R information to some outside file.

Connections are used as an argument to functions that read or write; the argument is usually the one named `file=` or `connection=`. In most cases, the argument can be a character string that provides the path name for a file.

This section discusses programming with connection objects, in terms of specifying and manipulating them. Section 5.6 discusses the functions most frequently used with connections.

Programming with connections

For programming with R, the most essential fact about connections may be that they are not normal R objects. Treating them in the usual way (for example, saving a connection object somewhere, expecting it to be self-describing, reusable, and independent of other computations) can lead to disaster. The essential concept is that connections are references to a data stream. A paradigm for defensive programming with connections has the form:

```
con <- create(description, open)
## now do whatever input or output is needed using con
close(con)
```

where *create* is one of the functions (`file()`, etc.) that create connections, *description* is the description of the file or command, or the object to be used as a text connection, and *open* is the string defining the mode of the connection, as discussed on page 134.

Two common and related problems when programming with connections arise from not explicitly closing them and not explicitly opening them (when writing). The paradigm shown is not always needed, but is the safest approach, particularly when manipulating connections inside other functions.

Connections opened for reading implement the concept of some entity that can be the source of a stream of bytes. Similarly, connections opened for writing represent the corresponding concept of sending some bytes to

the connection. Actually, hardly any R operations on connections work at such a low level. The various functions described in this chapter and elsewhere are expressed in terms of patterns of data coming from or going to the connection. The lower level of serial input/output takes place in the underlying C code that implements operations on connections.

Connections in R implement computations found at a lower level in C. The most useful property of a connection as an object is its (S3) class. There exist S3 methods for connection objects, for functions `print()` and `summary()`, as well as for a collection of functions that are largely meaningful only for connection-like objects (`open()`, `close()`, `seek()`, and others).

However, connections are distinctly nonstandard R objects. As noted on page 114, connections are not just objects, but in fact references to an internal table containing the current state of active connections. Use the reference only with great caution; the connection object is only usable while the connection is in the table, which will not be the case after `close()` is called. Although a connection can be defined without opening it, you have no guarantee that the R object so created continues to refer to the internal connection. If the connection was closed by another function, the reference could be invalid. Worse still, if the connection was closed and another connection opened, the object could silently refer to a connection totally unrelated to the one we expected. From the view of trustworthy software, of the *Prime Directive*, connection objects should be opened, used and closed, with no chance for conflicting use by other software.

Even when open and therefore presumably valid, connections are nonstandard objects. For example, the function `seek()` returns a "position" on the connection and for files allows the position to be set. Such position information is a reference, in that all R function calls that make use of the same connection see the same position. It is also not part of the object itself, but only obtained from the internal implementation. If the position is changed, it changes globally, not just in the function calling `seek()`.

Two aspects of connections are relevant in programming with them: what they are and how information is to be transferred. These are, respectively, associated with the *connection class* of the object, an enumeration of the kinds of entities that can act as suitable sources or sinks for input or output; and with what is known as the *connection mode*, as specified by the `open` argument to the functions that create a connection object.

Connection classes

Connections come from the concept of file-like entities, in the C programming tradition and specifically from the Posix standards. Some classes of connections are exactly analogous to corresponding kinds of file structures in the Posix view, other are extensions or analogs specific to R. The first group includes `"file"`, `"fifo"`, `"pipe"`, and `"socket"` connection objects. Files are the most common connections, the others are specialized and likely to be familiar only to those accustomed to programming at the C level in Linux or UNIX. Files are normally either specified by their path in the file system or created as temporary files. Paths are shown UNIX-style, separated by `"/"`, even on Windows. There are no temporary files in the low-level sense that the file disappears when closed; instead, the `tempfile()` function provides paths that can be used with little danger of conflicting with any other use of the same name.

Three classes of connections extend files to include compression on input or output: . They differ in the kind of compression done. Classes `"gzfile"` and `"bzfile"` read and write through a compression filter, corresponding to the shell commands `gzip` and `bzip2`. The `"unz"` connections are designed to read a single file from an archive created by the `zip` command. All of these are useful in compressing voluminous output or in reading data previously compressed without explicitly uncompressing it first. But they are not particularly relevant for general programming and we won't look at examples here.

The `"url"` class of connections allow input from locations on the Web (not output, because that would be a violation of security and not allowed). So, for example, the "State of the Union" summary data offered by the `swivel.com` Web site is located by a URL:

```
http://www.swivel.com/data_sets/download_file/1002460
```

Software in R can read this remote data directly by using the connection:

```
url("http://www.swivel.com/data_sets/download_file/1002460")
```

Text connections (class `"textConnection"`) use character vectors for input or output, treating the elements of the character vector like lines of text. These connections operate somewhat differently from file-like connections. They don't support seeking but do support `pushBack()` (see that function's documentation). When used for output, the connections write into an object whose name is given in creating the connection. So writing to a text connection has a side effect (and what's more, supports the idea of a non-local side effect, via option `local=FALSE`).

Modes and operations on connections

The modes and operations on connections, like the objects themselves, come largely from the C programming world, as implemented in Posix-style software. The operation of opening a connection and the character string arguments to define the mode of the connection when opened were inspired originally by corresponding routines and arguments in C. You don't need to know the C version to use connections in R; indeed, because the R version has evolved considerably, knowing too much about the original might be a disadvantage.

Connections have a state of being *open* or *closed*. While a connection is open, successive input operations start where the previous operation left off. Similarly, successive output operations on an open connection append bytes just after the last byte resulting from the previous operation.

The mode of the connection is specified by a character-string code supplied when the connection is opened. A connection can be opened when it is created, by giving the `open=` argument to the generating function. The connection classes have generating functions of the name of the class (`file()`, `url()`, etc.) A connection can also be opened (if it is not currently open) by a call to the `open()` function, taking an `open=` argument with the same meaning. Connections are closed by a call to `close()` (and not just by running out of input data, for example).

The mode supplied in the `open=` argument is a character string encoding several properties of the connection in one or two characters each. In its most general form, it's rather a mess, and not one of the happier borrowings from the Posix world. The user needs to answer two questions:

- Is the connection to be used for reading or writing, or both? Character `"r"` means open for reading, `"w"` means open for writing (at the beginning) and `"a"` means open for appending (writing after the current contents).

  Confusion increases if you want to open the connection for both reading and writing. The general notion is to add the character `"+"` to one of the previous. Roughly, you end up reading from the file with and without initially truncating it by using `"w+"` and `"a+"`.

- Does the connection contain text or binary data? (Fortunately, if you are not running on Windows you can usually ignore this.) Text is the default, but you can add `"t"` to the mode if you want. For binary input/output append `"b"` to the string you ended up with from the first property.

So, for example, `open="a+b"` opens the connection for both appending and reading, for binary data.

The recommended rules for functions that read or write from connections are:

1. If the connection is initially closed, open it and close it on exiting from the function.

2. If the connection is initially open, leave it open after the input/output operations.

As the paradigm on page 131 stated, you should therefore explicitly open a connection if you hope to operate on it in more than one operation.

Consider the following piece of code, which writes the elements of a character vector `myText`, one element per line, to a file connection, to the file `"myText.txt"` in the local working directory:

```
txt <- file("./myText.txt")
writeLines(myText, txt)
```

The output is written as expected, and the connection is left closed, but with mode `"w"`. As a result, the connection would have to be explicitly re-opened in read mode to read the results back. The default mode for connections is read-only (`"r"`), but `writeLines()` set the mode to `"wt"` and did not revert it; therefore, a call to a read operation or to `open()` with a read mode would fail. Following the paradigm, the first expression should be:

```
txt <- file("./myText.txt", "w+")
```

Now the connection stays open after the call to `writeLines()`, and data can be read from it, before explicitly closing the connection.

5.6 Reading and Writing Objects and Data

R has a number of functions that read from external media to create objects or write data to external media. The external media are often files, specified by a character string representing the file's name. Generally, however, the media can be any connection objects as described in Section 5.5.

In programming with these functions, the first and most essential distinction is between those designed to work with any R object and those designed for specific classes of objects or other restricted kinds of data. The first approach is based on the notion of *serializing*, meaning the conversion

of an arbitrary object to and from a stream of bytes. The content of the file is not expected to be meaningful for any purpose other than serializing and unserializing, but the important property for programming is that any object will be serialized. The second type of function usually deals with files that have some particular format, usually text but sometimes binary. Other software, outside of R, may have produced the file or may be suitable to deal with the file.

Serializing: Saving and restoring objects

The serializing functions write and read whole R objects, using an internal coding format. Writing objects this way and then reading them back should produce an object identical to the original, in so far as the objects written behave as normal R objects. The coding format used is platform-independent, for all current implementations of R. So although the data written may be technically "binary", it is suitable for moving objects between machines, even between operating systems. For that reason, files of this form can be used in a source package, for example in the `"data"` directory (see Section 4.3, page 87).

There are two different approaches currently implemented. One, represented by the `save()` and `load()` functions, writes a file containing one or more named objects (`save()`). Restoring these objects via `load()` creates objects of the same names in some specified R environment. The data format and functions are essentially those used to save R workspaces. However, the same mechanism can be used to save any collection of named objects from a specified environment.

The lower-level version of the same mechanism is to `serialize()` a single object, using the same internal coding. To read the corresponding object back use `unserialize()`. Conceptually, saving and loading are equivalent to serializing and unserializing a named list of objects.

By converting arbitrary R objects, the `serialize()` function and its relatives become an important resource for trustworthy programming. Not only do they handle arbitrary objects, but they consider special objects that behave differently from standard R objects, such as environments. To the extent reasonable, this means that such objects should be properly preserved and restored; for example, if there are multiple references to a single environment in the object(s) being serialized, these should be restored by `unserialize()` to refer to one environment, not to several. Functions built on the serializing techniques can largely ignore details needed to handle a variety of objects. For example, the `digest` package implements a hash-

style table indexed by the contents of the objects, not their name. Using `serialize()` is the key to the technique: rather than having to deal with different types of data to create a hash from the object, one uses `serialize()` to convert an object to a string of bytes. (See Section 11.2, page 416, for an example based on `digest`.)

Two caveats are needed. First, references are only preserved uniquely within a single call to one of the serializing functions. Second, some objects are only meaningful within the particular session or context, and no magic on the part of `serialize()` will save all the relevant context. An example is an open connection object: serializing and then unserializing in a later process will not work, because the information in the object will not be valid for the current session.

Reading and writing data

The serializing techniques use an internal coding of R objects to write to a file or connection. The content of the file mattered only in that it had to be consistent between serializing and unserializing. (For this reason, serializing includes version information in the external file.)

A different situation arises when data is being transferred to or from some software outside of R. In the case of reading such data and constructing an R object, the full information about the R object has to be inferred from the form of the data, perhaps helped by other information. General-purpose functions for such tasks use information about the format of character-string data to infer fairly simple object structure (typically vectors, lists, or data-frame-like objects). Many applications can export data in such formats, including spreadsheet programs, database software, and reasonably simple programs written in scripting, text manipulation, or general programming languages. In the other direction, R functions can write text files of a similar form that can be read by these applications or programs.

Functions `scan()` and `read.table()` read fields of text data and interpret them as values to be returned in an R object. Calls to `scan()` typically return either a vector of some basic class (`numeric` or `character` in most cases), or a list whose components are such vectors. A call to `read.table()` expects to read a rectangular table of data, and to return a `data.frame` object, with columns of the object corresponding to columns of the table. Such tables can be generated by the export commands of most spreadsheet and database systems. Section 8.2, page 294, has an example of importing such data.

A variety of functions can reverse the process to write similar files: `cat()` is the low-level correspondence to `scan()`, and `write.table()` corresponds to

`read.table()`.

These functions traditionally assume that file arguments are ordinary text files, but they can in fact read or write essentially any connection. Also, functions exist to deal with binary, `raw`, data on the connection rather than text fields. See the documentation for functions `readBin()` and `writeBin()`.

For many applications, these functions can be used with modest human effort. However, there are limitations, particularly if you need an interface to other software that deals with highly structured or very large objects. In principle, specialized inter-system interfaces provide a better way to deal with such data. Some interfaces are simple (but useful) functions that read the specialized files used by other systems to save data. At the other extreme, inter-system interfaces can provide a model in one language or system for computing in another, in a fully general sense. If a suitable general inter-system interface is available and properly installed, some extra work to adapt it to your particular problem can pay off in a more powerful, general, and accurate way of dealing with objects in one system when computing in another. See Chapter 12 for a discussion.