Paper TS03

Techniques for writing robust R programs

Martin Gregory , Merck Serono

1 Abstract

In complex applications it is important that individual components are robust and interact in a predictable manner. A robust program should be well documented and tested, but also validate its inputs, handle failure in a controlled and elegant way, appropriately notify the calling programmer or program of the outcome of execution. Furthermore a good program does not pollute its environment. This paper describes specific techniques for implementing these concepts when building applications using the R Environment for Statistical Computing. While a certain investment is involved in applying the techniques, they result in more reliable and easier to maintain systems with a lower support requirement.

2 Introduction

When we write software, especially for data analysis, we want to ensure that it can be trusted to accurately carry out the tasks for which it was intended. The term robust is often used and can be defined in various ways. The *Good Programming Practice* section of the *PhUSE Wiki* describes it as software that

> "performs well not only under normal circumstances but also under conditions that stress its original purpose and assumptions" [1]

Performing well should also be understood to mean that the program results are reliable, not only that it reacts appropriately in stress situations. Chambers[2], in using the term *Trustworthy Software* goes a step further and suggests that the programmer not only has a responsibility to produce a result that is trustworthy, but also one that can be shown to be trustworthy.

The goal of this paper is to discuss how the R language features help the programmer in delivering robust and trustworthy programs which are seen to be so. The paper will, more specifically, cover defining function parameters, validation of inputs, handling errors in a controlled way and restoring environment after the function has completed. While R provides some interesting features for standardising documentation and automating testing when building packages, a separate paper would be required in order to treat the topic of packages adequately.

Achieving robustness requires additional effort in design and implementation so this needs to be weighed against the benefits using a risk-based approach. For programs which will be run only one time, with know inputs a different approach may be appropriate that for a package intended for use by many programmers.

Section 3 is dedicated to some key aspects of the language which might affect robustness while the subsequent sections 4 to 7 provide practical solutions to ensure robustness of programs.

3 Some relevant aspects of R

We start by highlighting some idiosyncrasies of the R language where behaviour differs from other data analysis languages.

3.1 R is a functional language

R belongs to a class of programming languages known as functional languages. In purely functional languages, a program is a series of function calls. Furthermore, functions obey two important rules:

1. the result of a function is determined entirely by the inputs to the function; and
2. functions have no side effects, i.e. the only result of a function call is the value returned by the function

In a way, these rules are two sides of the same coin. The first rule may also be stated that nothing in the environment has an influence on the result of the function. The second rule implies that apart from producing a result, no other changes to the environment are made. Note that the second rule means that the values of inputs cannot be changed by a function call[1]. The idea behind this approach is to reduce or eliminate the effect of external influences on the program and therefore make is more reliable.

Functional programming is primarily a programming paradigm and can be implemented in any language which provides for the definition of functions. Some languages are designed to support the functional programming paradigm and have a syntax which does this to a greater or lesser extent. One should not, however, confuse the syntax with the paradigm. An extreme example is Common Lisp in which the only type of statement available is the function call. For example adding two numbers and assigning a value to a variable are function calls:

```
;; add two numbers
> (+ 2 3)
5
;; assign to a variable
> (setq a (+ 2 3))
5
```

As the *R Language Definition* [3] states, R derives many of its concepts from Lisp but is not as extreme as Lisp. R has statements for assignment and for flow control. Furthermore, operators appear as infix elements as in non-functional languages. Using the same example as above:

```
# add two numbers
> 2+3
[1] 5
# assign to a variable
> a <- 2 +3
> a
[1] 5
```

R operators are, in fact, functions and may be called as such:

```
# addition as function
> '+'(2,3)
[1] 5
```

Following the functional programming paradigm is one way to increase the reliability or robustness of your programs, whatever the syntax of the language. In the remainder of the paper, when we speak of robust programs we essentially mean robust functions.

[1] In some languages, e.g. Perl, SAS/SCL, call by reference allows the function to change the value of the variable passed as parameters

3.2 R is not purely functional

It is important to be aware that the R language itself breaks both rules in certain circumstances and allows you to break the rules too. There are good reasons for the exceptions and when you write your own functions you must decide whether these reasons apply or not.

3.2.1 Function result determined entirely by the inputs

This rule is broken by the fact there there are global options. For example, the option *digits*, in the absence of specific instructions, determines the number of digits to use when printing numeric data. It is not unreasonable to have such options but the danger lies in the fact that any function may change them by calling the *options()* function. A further option which is not so innocent is *contrasts*. This specifies functions to be used for contrasts in the modelling functions *aov()* and *lm()* when contrasts are not specified.

This rule has been broken in the language itself primarily for historical reasons and the amount of existing code depending on these global options. To make your own functions independent of arbitrary settings of those options your function uses, you can temporarily set them during function execution and restore on exit. Section 7 describes in detail how to do this.

3.2.2 Functions have no side-effects

An essential part of data analyis is viewing the results, whether tabular or graphical. Functions such as *summary()*, *plot()* and *print()* are used for this purpose and these function are called primarily for their side effects and do not return an object which is part of the language.

A further example is non-local assignment using the "<< −" or *assign()* function. These allow assignment of values to variables in environments outside that of the function itself.

Finally, the *options()* function can set values of globally available variables.

3.3 R is a vector language

There are no scalar data types in R, assigning a single value to a variable results in a vector with one element as can be seen by the index information shown in square brackets when the variable is displayed:

```
> a <- 5
> a
[1] 5
```

There are situations where a scalar value is required, in particular in conditions, both in *if* or *while* constructs:

```
> a <- c(1,2,3)
> if (a==3) print("hello")
Warning message:
In if (a == 3) print("hello") :
  the condition has length > 1 and only the first element will be used
```

Here only a warning message is printed, so if the caller does not read the message, or if warning messages are turned off, or the message comes from the depths of a series of nested function calls, there is a danger that the test may not produce the intended effect. In this particular case, the test is false, but if the first element of a were 3, the test would be true.

When an error occurs it is at least obvious. Suppose we want to test whether all elements of a vector are positive

```
> if (min(a)>0) print("all positive")
[1] "all positive"
```

A vector might contain only missing values:

```
> xNA <- c(NA,NA)
> if (min(xNA)>0) print("all positive")
Error in if (min(xNA) > 0) print("all positive") :
  missing value where TRUE/FALSE needed
```

The result of the expression *NA>0* is NA and the problem is that the condition requires that the expression resolve to a logical value.

A further potential issue is that R allows vectors of zero length, i.e. a vector may have no elements:

```
> xZero <- vector()
> length(xZero)
[1] 0
> if (min(xZero)>0) print("all positive")
[1] "all positive"
Warning message:
In min(xZero) : no non-missing arguments to min; returning Inf
```

For empty vectors, the function *min()* returns infinity, which is greater than zero and so the condition is true although the caller is warned of the potential problem.

3.4 Variable scope

For both of the rules governing functional programming, the question of variable scope is relevant. When a function executes, a reference to a variable which is neither a parameter nor previously defined in the function will cause a seach in enclosing environments[2] for the variable. The first value encountered will be used. If the variable is not found in any environment, an error is raised. To illustrate this behaviour consider the following trivial and unrealistic function:

```
plusone <- function(x){
  print(sprintf("in plusone: a=%d",a))
  a <- a+1
  print(sprintf("in plusone: a=%d",a))
  return(x+1)
}
```

which returns its argument augmented by 1. It also prints the value of a variable *a* which is not defined in the function, adds 1 to *a* and prints the new value. Now consider the following calls to the function and their results:

```
> a <- 1
> plusone(2)
[1] "in plusone: a=1"
[1] "in plusone: a=2"
[1] 3
> a
[1] 1
```

[2] each function executes in its own environment

The value of *a* in the enclosing environment is visible in the function environment, but the assignment to *a* in the function creates a new variable local to the function. So while the output of the function could be affected by values not passed to the function, side effects are not possible. If we now delete the variable and call the function again:

```
> rm(a)
> plusone(2)
Error in sprintf("in plusone: a=%d", a) : object 'a' not found
```

the first reference to it causes an error.

4 Defining function parameters

When defining a function there are three possibilities for specifying parameters:

1. named parameter with default value;
2. named parameter with no default;
3. ... for arbitrary parameters.

The following example shows all three possibilities:

```
myfun <- function(df,by,digits=4,...) { function body }
```

No declaration of parameter type is possible in the definition. When calling functions, the caller may choose between naming or not. The following are all valid ways of calling the function *myfun* above:

```
myfun(df=adae,by=usubjid,dig=3)
myfun(adae,usubjid,3)
myfun(usubjid,df=adae,3)
```

A three-step process is used to match arguments in the call to parameters in the definition:

1. exact matches for named parameters;
2. unambiguous partial matches for remaining named parameters;
3. positional matching for unnamed parameters.

This approach to matching arguments to formal parameters may seem arbitrary but is related to the use of the ... syntax which allows a function to be defined with arbitrary parameters. In the following example

```
mymatch <- function (a, b, c, d) {
   print(a)
   print(b)
   print(c)
   print(d)
}

> mymatch("one",a=2,b=3)
[1] 2
[1] 3
[1] "one"
Error in print(d) : argument "d" is missing, with no default
```

We see that the first argument is match to parameter *c*, the first not named in the call. For clarity and lack of ambiguity for those reading the function, it is advisable to use the exact named form in calls, i.e. according to the first matching rule.

5 Input validation

There are two different approaches to the validation of parameters:

reactive: we assume that if a problem arises as a result of an invalid parameter we can then handle the error.

proactive: we mean that the function checks that all required parameters have been supplied, that the supplied values are are of the type expected and that the values specified are appropriate.

In general, there are two major disadvantages to the reactive approach. First, when errors do occur, messages are provided by a function called by our function and these may be confusing to a caller not familiar with the internals of our function. Proactive validation allows the developer to provide more specific error messages to the caller. The second disadvantage of the reactive approach is that unnecessary computations may have been carried out before the error occurs. Depending on the parameters, there may be arguments in individual cases for reactive validation, but for robust programs the proactive approach is the preferred choice.

5.1 Checking a value has been specified

The case that a parameter is not supplied in the call can be detected by the *missing()* function[3]. This can be used to determine whether the caller has specified values for required parameters.

```
if (missing(df)) stop("parameter df is required.")
```

For optional parameters, it can also be used to provide a default value in a more flexible manner than is possible by specifying a fixed value in the definition of the function. The value could be based on conditions, such as the values of other parameters.

5.2 Checking the object class

The next step is to check that the objects supplied by argument values are the correct class. This is done using the *class()* function which, for vectors, returns one of logical, integer, numeric[4], character, list[5], complex or raw. More complex classes also exist, such as data.frame, matrix, survival or various object classes return by statistical analysis functions. Care needs to be taken here as an object may belong to multiple classes, so one of the following tests

```
if (is.element("data.frame",class(df)) ...
if (inherits(df,c("data.frame"))) ...
```

is safer than a direct comparison.

5.3 Checking the object content

Even though the class of the object may be correct, we may still need to check that the object actually contains appropriate data. As we saw in section 3.3, an object might be empty or contain only missing values, or values inappropriate for the purpose of the function.

[3] this function may only be called directly in the function whose parameters it is processing
[4] double precision floating point
[5] a vector in which elements may be of any data type

5.3.1 Is the object empty?

The first thing to check is whether the object is empty or not. For any object we can determine the number of elements using the *length()* function:

```
> a <- c(3,2,1)
> length(a)
[1] 3
```

For a data frame, we need a further test as the data frame is actually a list of vectors, each vector corresponding to a variable, the length is the number of variables. The function *nrow()* returns the number of rows in the data frame. Alternatively, both pieces of information can be retrieved by calling the *dim()* function:

```
> qrt
         r     theta
1 1.414214 0.7853982
2 2.828427 0.7853982
3 4.242641 0.7853982
> length(qrt)
[1] 2
> nrow(qrt)
[1] 3
> dim(qrt)
[1] 3 2
```

5.3.2 Does the object contain non-missing values?

To check for missing values (NA) we use the function *is.na()*. This returns a logical object of the same type and dimensions as the argument with a TRUE value at the location of missing values:

```
> qrtNA
         r     theta
1 1.414214 0.7853982
2 2.828427        NA
3 4.242641 0.7853982
> is.na(qrtNA)
         r theta
[1,] FALSE FALSE
[2,] FALSE  TRUE
[3,] FALSE FALSE
```

By combining this with the *any()* or *all()* functions we can obtain a single value indicating if any or all of the elements are missing.

```
> if (any(is.na(qrtNA))) print("some missing")
[1] "some missing"
```

The function *anyNA()* can be used in place of *any(is.na())* and is faster.

5.3.3 Comparing vector values

If we need to compare the values in a vector to another vector element by element, we can use the *any()* or *all()* functions to test whether any or all of the comparisons are true. To test if each element of one vector is greater than the corresponding element in a second:

```
if (all(a>b)) ...
```

This ensures that the condition resolves to a single logical value. If we need to keep the element by element comparison, avoiding an explicit loop we can use the *ifelse()* function:

```
> x <- c(6:-4)
> sqrt(x)
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000
 [7] 0.000000      NaN      NaN      NaN      NaN
Warning message:
In sqrt(x) : NaNs produced
> sqrt(ifelse(x >= 0, x, NA))
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000
 [7] 0.000000       NA       NA       NA       NA
```

5.4 External objects

In the preceding sections we considered validation of input where the input is an object in R. Functions may require access to external objects such as a data file or to the output of a command. As described in [4], R provides extensive support for access to operating system functions in a way which hides the operating system entirely, i.e. the R programmer does not need to query the operating system and have separate code to handle supported operating systems. For many types of external files, including databases, R also provides functions for reading and writing such files.

6 Error handling

In this section we consider two topics. One is the question of how we detect errors in functions we call, the other is how we can appropriately exit from a function when this is necessary.

6.1 Exiting from a function

There are three ways to exit a function:

implicitly: in this case, if the programmer of the function does not take any specific action to exit, the function will return the last evaluated expression. This approach affords no control is not recommended for robust programming.

return(): this explicit method provides the argument of the *return()* function as the result of the function. Multiple calls, from different parts of the function are possible, depending on the logic of the function. If the *on.exit()* function described in section 7 is used, clean-up code can be located in one place.

stop(): this explicit method signals an error. In interactive mode, by default control, is returned immediately to the interactive prompt. In batch mode, the R session is terminated. The resulting behaviour can be modified by defining error handlers as described in section 6.2 below.

6.2 Detecting problems

In contrast to non-functional languages, the value returned by a function cannot generally be interpreted as a return code. Functions which return a logical object might be interpreted as success or failure and used directly for this purpose. Examples of such functions are those that check existence (*exists(), file.exists()*)) or check that an object is of a particular type

(*is.numeric(), is.character(), is.NA(), ...*) or is a member of another object (*is.element()*). So testing whether a function was successful is not simply a case of comparing the function result to a number.

Two approaches are available. We can examine the result of the called function in order to see if the function succeeded. The documentation of the function should describe the return value for both normal and abnormal completion. This first approach does not, however, work when the called function uses *stop()* as there is no opportunity to examine the result.

Alternatively, R implements an error and warning handling facility similar to that provided by Lisp or Java. This consists of functions to trigger an error or warning condition and functions to trap or catch these conditions thus allowing the caller to take appropriate action. There are three functions for triggering a condition. First, *warning()* triggers a warning condition which by default is signalled only after the function has completed. Second, *stop()*, as we have seen above, triggers an error condition which by default immediately interrupts execution. Finally programmers can define their own conditions and signal them using *signalCondition()*. Let us examine the first two using a rather artificial example. We start by defining two functions. The first, *throw()*, takes a single parameter and signals an error when passed the value 1, a warning on 2 and returns the passed value otherwise:

```
throw <- function (a) {
  if (a==1) stop("a=1")
  else if (a==2) warning("a=2")
  else return(a)
}
```

The second function, *with.tryCatch()* shows how to catch a condition and take appropriate action, in particular, prevent an error from exiting the entire call stack:

```
with.tryCatch <- function(a) {
  result <- tryCatch(throw(a),
                     error=function(e) e,
                     warning=function(w) w)
  if (inherits(result, c("error","warning"))) return(result)
  print(result)
}
```

Instead of calling our *throw()* function directly, we use the *tryCatch()* function which defines functions[6] to execute when error or warning conditions are signalled. By calling this way, we have trapped or caught the conditions and can decide what to do. Conditions are actually R objects with a class of *error* or *warning*. So we decide to exit the function if the result has or inherits one of these classes. *tryCatch()* can also be used to evaluate any expression, not just function calls.

Clearly you will not use *tryCatch()* on every function call or expression in your program but will have to decide where it makes sense to do so. The same is true of the *stop()* function. If you use this, callers of your function who provide invalid inputs may have to set up condition handlers as described above.

7 Restoring the environment

The *on.exit()* function can be used to execute code when a function completes. The code is executed irrespective of whether the function completed normally or not and so is suitable for any cleanup operations such as resetting global options or deleting temporary files. Consider this illustrative example:

[6] R functions can be anonymous as, like Lisp, it supports lambda calculus

```
tidy <- function(df) {
  tmp <- tempfile(pattern=c("tidy"),tmpdir=".",fileext=c(".txt"))
  print(paste("Temp file is ",tmp))
  on.exit(if (file.exists(tmp)) unlink(tmp))
  ## save and set width option
  w <- getOption("width")
  options(width=40)
  on.exit(options(width=w),add=TRUE)
  ## write and read file
  sink(tmp)
  print(df)
  sink()
  fid <- file(tmp, "r")
  print(readLines(fid,-1,TRUE))
  close(fid)
}
> tidy(pxy)
[1] "Temp file is  ./tidy286d3863544c.txt"
[1] "  x y" "1 1 1" "2 2 2" "3 3 3"
> print(getOption("width"))
[1] 80
```

Note that temporary files are automatically deleted by R at the end of the session if you use the default value for the *tmpdir* parameter. In that case, if you want to clean up directly after the function, you need to remove them explicitly.

8 Conclusion

In this paper, we have discussed features of the R Environment for Statistical Computing which are of particular interest for writing robust functions. Support for the functional programming paradigm, combined with R's features for querying objects in the language and built-in, flexible error-handling functions provide the means for implementing functions which perform well not only under normal circumstances but also under conditions that stress their original purpose and assumptions.

Applying these techniques clearly has an associated cost in terms of programming effort which must be weighed against the benefits of being robust and trustworthy. Some of the techniques can, however, become part of the programmer's toolbox, thus reducing the effort.

What we have not shown in this paper, is that building R packages provides additional features including name spaces, standardised documentation and automating testing. The interested reader is encouraged to consult *Writing R Extensions* [5].

References

[1] Greg Nelson, Sascha Ahrweiler, and M Foxwell. Good programming practice: Robustness. http://www.phusewiki.org/wiki/index.php?title=Robustness, May 2012.

[2] John M Chambers. *Software for data analysis : programming with R.* New York ; London : Springer, 2008. Formerly CIP.

[3] R Core Team. R Language Definition. http://cran.r-project.org/doc/manuals/r-release/R-lang.html, 2014.

[4] Martin Gregory. Controlling opencdisc validator using r. *Pharmaceutical Programming*, 5(1-2):5–9, 2012-12-01T00:00:00.

[5] R Core Team. Writing R Extensions. http://cran.r-project.org/doc/manuals/r-release/R-exts.html, 2014.

Contact Information

Martin Gregory
Director Biostatistics Standards & Processes
Merck Serono
Frankfurter Str 250 64293 Darmstadt Germany
martin.gregory@merckgroup.com