A Comprehensive Comparison of C++ and R

November 2020

Contents

1 R

1.1 Purpose and Motivations

Fundamentally, R is a dialect of S. It was created to do away with the limitations of S, which is that it is only available commercially.

1.2 History (Authors, Revisions, Adoption)

1.2.1 S, the precursor of R

S is a language created by John Chambers and others at Bell Labs on 1976. The purpose of the language was to be an internal statistical analysis environment. The first version was implemented by using FORTRAN libraries. This was later changed to C at S version 3 (1988), which resembles the current systems of R.

On 1988, Bell labs provided StatSci (which was later named Insigthful Corp.) exclusive license to develop and sell the language. Insightful formally gained ownership of S when it purchased the language from Lucent for $ 2,000,000, and created the language as a product called S-PLUS. It was names so as there were additions to the features, most of which are GUIs.

1.2.2 R

R was created on 1991 by Ross Ihaka and Robert Gentleman of the University of Auckland as an implementation of the S language. It was presented to the 1996 issue of the *Journal of Computational and Graphical Statistics* as a "language for data analysis and graphics". It was made free source when Martin Machler convinced Ross and Robert to include R under the GNU General Public License.

The first R developer groups were in 1996 with the establishment of R-help and R-devel mailing lists. The R Core Group was formed in 1997 which included associates which come from S and S-PLUS. The group is in charge of controlling the source code for the language and checking changes made to the R source tree.

R version 1.0.0 was publicly released in 2000. As of the moment of writing this paper, the R is in version 4.0.3.

1.3 Language Features

R as a language follows the philosophy of S, which was primarily developed for data analysis rather than programming. Both S and R have interactive environment that could easily service both data analysis (skewed to command-line commands) and longer programs (following traditional programming). R has the following features:

- Runs in almost every standard computing platform and operating systems
- Open-source
- An effective data handling and storage facility
- A suite of operators for calculations on array, in particular matrices
- A large, coherent, integrated collection of intermediate tools for data analysis
- Graphical facilities for data analysis and display either on-screen or on hardcopy u
- Well-developed, simple, and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.
- Can be linked to C, C++, and Fortran for computationally-intensive tasks
- A broad selection of packages available in the CRAN sites which cater a wide variety of modern statistics
- An own LaTex-like documentation format to supply comprehensive documentation
- Active community support

1.4 Paradigm(s)

Functional

1.5 Language Evaluation Criteria

1.5.1 Data Types

R abides by the principle that everything is an object which is stored in physical memory, there are no "data types", per se. Hence this section shall cover and critique the fundamental data objects (which may be called variables throughout the paper) and values found in R.

Atomic Objects There are five basic objects which serve as the building blocks of other complex objects in the language. The objects are given by the table below:

Table 1: Atomic Classes of Objects in R

Object Name	**Sample Values**	**Stored as**
character	"R", "another char"	character
numeric (real)	2, 1.0	numeric
integer	1L, 22L	integer
complex	1 + 0i, 2 - 11i	complex
logical	TRUE, T, FALSE, F	logical
raw	"Hello" which is stored as 48 65 6c 6c 6f	raw

It must also be noted that vectors are the most basic type of objects in R. Hence, these atomic objects are actually vectors of length 1. A more detailed analysis on the effects of vectors will be discussed further sections.

Majority of the atomic objects are intuitive in nature, however some are affected by the peculiarities of the language. By default, R treats numbers as numeric types, which are implemented as double precision real. This can be very useful if one is dealing with large numbers as the memory allocated to double precision is suitable, however it would be too much if numbers which fall within the short or integer range are used. To declare an integer, one must add L as a suffix to the number, as seen in the table - this may positively affect readability as one can distinctly separate the integers and the non-integers, however writability suffers as forgetting the suffix means that the number is type casted into numeric, which may happen when one writes long code in the language. Another issue on readability and writability is the allowing of T and F, which also has other issues to be discussed in the next section, as native variables which contain TRUE and FALSE values - this causes ambiguity in the sense that a single construct is implemented in two way, and it may be confused with a variable,

Additionally, R was also designed with no separate string object, thereby eroding the distinction between characters (which are implemented generally as single characters) and strings (which may contain zero or more characters).

For composite objects, R has vectors, matrices, names, lists, and data frames.

Vectors. As stated earlier, vectors are the most basic objects in R. Vectors, much like the implementations of languages such as C and Java, are collections of objects with the same type. Some special vectors included the atomic objects and vectors with a length of 0. If type-checked, a vector will reflect the data type of its values. Implementations of vectors can be done using the following syntax:

```
> vec <- c(1,2,3) #using c()
> vec
[1] 1 2 3
> class(vec)
[1] numeric
> vec2 <- vector(mode=''numeric'', length= 3L) #by vector()
> vec2 #uniform values vector
[1] 0 0 0
> class(vec2)
[1] numeric
> vals <- c(4,5,6)
> vec3 <- as.vector(vals) #explicit coercion
> vec3
[1] 4 5 6
> class(vec)
[1] numeric
```

This again raises the issue of ambiguity, as vectors can be implemented in three different ways. Writability suffers as even though programmers may choose a single implementation, the three methods' use cases do not directly intersect with each other (vector() creates a vector of uniform values, c() is the most generic implementation but does not directly handle uniform values, and as.vector() is an explicit coercion to a vector). Readability also suffers with the usage of c(), as the function names are not self-documenting.

Matrices. Matrices are two-dimensional vectors, with the dimension as an attribute of length 2 comprised of the number of rows and number of columns. The implementation is done using the following syntax:

```
> matr <- matrix(data=1:4, nrow=2, ncol=3) #using matrix()
> matr #matrix of NAs
    [,1] [,2]
[1,] 1 3
[2,] 2 4
> matr2 <- 1:4
> dim(matr2) <- c(2,2) #adding dims to vector
> matr2
    [,1] [,2]
[1,] 1 3
[2,] 2 4
> x <- 1:2
> y <- 3:4
> matr3 <- rbind(x,y) #row binding vectors
> matr3
    [,1] [,2]
x 1 2
y 3 4
> matr4 <- cbind(x,y) #column binding vectors
> matr4
      x y
[1,] 1 2
[2,] 3 4
```

Similar to vectors, the multiple implementations with different use cases negatively affects both readability and writability as the programmer needs to remember each implementation.

Lists. Lists are special vectors that can hold values of different classes. The implementation is done using the following syntax:

```
> list1 <- list(1, 2L, True, ‘‘list’’) #using list()
> list1
[[1]]
```

```
[1] 1

[[2]]
[1] 2

[[3]]
[1] True

[[4]]
[1] ``list''
> list2 <- vector(``list'', length=4) #using vector()
> list2 #empty list of specified length
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

In this case, one can see heavy ambiguity with the use of vector(). Although a list is a type of vector, using vector() to create a NULL list defeats the purpose of having a separate list function. In turn, it aids somehow aids writability as a programmer can use one function, but negatively affects readability, even with the passing of the "list" argument.

Factors. Factors represent categorical data, with or without order. This data class is important for statistical modeling. The sole implementation is given by the syntax:

```
> x <- factor(c(``red'', ''blue'', ``red'', ``red''))
> x
[1] red blue red red
Levels: blue red
```

Only one implementation enhances writability as a programmer would not need to memorize multiple syntax to create factors; it negatively affects readability as the programmer needs to memorize yet another data class, albeit necessary.

Data Frames. Data frames are implemented as a special type of list with each element having the same length, which is intuitive as it is used to read tabular data. Each element (specifically, column) only has one class, but the columns may have different classes from each other. This data class is implemented by the given syntax:

```
> df <- data.frame(nums=1:4, letters=c(‘‘a’’,’’b’’,’’c’’,
                  ’’d’’))
> df
  nums letters
1 1 a
2 2 b
3 3 c
4 4 d
```

Only one implementation enhances writability as a programmer would not need to memorize multiple syntax to create data frames; it negatively affects readability as the programmer needs to memorize yet another data class, albeit necessary.

Special Data Values. As R is fundamentally a statistical language, it contains other values which are integral in processing data, such as:

NA stands for “Not Available” as an indicator for missing values. It can have classes to (except raw)

NaN stands for “Not a Number” which applies to numerical, and complex (real, imaginary) values, but not to integer vector values.

NULL is an object which is returned when an expression/function returns an undefined value.

Inf/-Inf stands for infinity which entails very large values or the quotient of dividing a number by 0.

In strengthens readability as each value covers distinct contexts, making them very readable for the programmer. On the other hand, writability suffers as programmers must memorize more values to suit their needed cases.

1.5.2 Binding, Scoping, Referencing

This section covers names, variable lifetimes, scope, coercion, and an introduction on how R implements scoping.

Names. Names in R are case sensitive. Such a feature is critiqued for being less readable as similar-looking names which may only differ in capitalization entails confusion. Length has not been seen as an issue as the language provides no limit. Valid variable names (formally symbols) in R can be represented by this BNF:
<variable>::= <first><second><succeeding>*
<first>::= A-Z — a-z — .
<second>::= A-Z — a-z — . — _
<succeeding>::= A-Z — a-z — . — _ — 0-9

It can be noticed that variable names like ..var is possible - this is because the dot character in R bears no major significance (except for the implications in UNIX-adapted commands such as ls() and the ... syntax used in functions). $ is used a manner similar to the dot in other languages. These feature are quite problematic for programmers who have experience in Object Oriented programming as variable names such as sample.var has an actual implication in the OO paradigm, especially as this is the recommended naming style of Google's R Style Guide. Other naming conventions use underscores (sample_var), which may cause writability issues for programmers who use Emacs Speaks Statistics (ESS) as the underscore is mapped to the ¡-operator, and camelcase (sampleVar), which also suffers the readability issue of having similar-looking variable names.

Reserved Words. R has some reserved words such as if, else, repeat, while, function, for, in, next, break, TRUE, FALSE, NULL, Inf, NaN, NA, NA_integer_, NA_real_, NA_complex_, NA_character_, ..., ..1, ..2, ..3, ..4, ..5, ..6, ..7, ..8, and ..9. Furthermore, the language has several one-letter "reserved" words: c, q, s, t, C, D, F, I, and T. However, native variables like T and F (corresponding to True and False, respectively) can be overwritten without producing any warning messages hence they are not as rigid as the formally reserved words:

```
> T #T as a logical value
[1] True
> T <- 22 #declaring some value to T ,does not raise
   ↪ warnings
> T
[1] 22
```

Also, R separates the namespaces for functions and nonfunctions so a variable c and the vector-creating function c() can coexist, which is disadvantageous for readability as programmers have to be aware of the different uses of a name.

On Variables. Variable aliasing does not exist in R as the language allows only for copying objects and does not directly give the user access to pointers. This is beneficial for reliability as the programmer is assured that no other variable hold the same address with each other hence there is no risk for unintentional changes. On the other hand, as variables are objects which are stored into physical memory, excessive copying of variables may be detrimental to large systems. Most R object can be bound to variables, as well as other statements (such as conditional statements) and functions. Type definitions are not required for variable declarations as R uses dynamic binding. This feature is advantageous to writability as the programmer does not need to explicitly specify the data types, however the burden is on the interpreter to dynamically type check and interpret the variables during runtime. Furthermore, issues on type error detection may be difficult, also due to the language's coercion rules.

Coercion. R checks type compatability during runtime and performs implicit coercion, when possible. It follows certain coercion rules: - Logical values are converted to numbers: TRUE is converted to 1 and FALSE to 0 - Values are converted to the simplest type to represent all information - The orderning is roughly logical < integer < numeric < complex < character < list - Objects of type raw are not converted to other types - Object attributes are dropped when an object is coerced from one type to another

This greatly affects reliability as some of the features are not as intuitive - examples would include the following:

‘

```
> vec1 <- c(1, 'a', 3.11, TRUE) # direct coercion of vectors
> vec1
[1] ''1'' ''a'' ''3.11'' ''TRUE''
```

```
> class(vec1)
[1] ‘‘character’’
> TRUE + 2 # direct coercion during addition
[1] 3
```

In addition, the language does not yield any exceptions when an object of the wrong type is passed in functions, rather it directly converts it. Such non-strongly typed features can cause ambiguous and unreliable results. Albeit the coercion rules try to represent as much information, the risk of wrongly applying expressions between two object types and the direct dropping of certain attributes of a coerced object does more harm than good.

Scoping. In the context of R as part of the functional paradigm, lexical scoping means that a free variable's declaration within a function (i.e. variables that are used in a function but not defined in the function) are looked up in the enclosing static scopes or parent environment of the function, as opposed to the environment of the caller (also referred to as the parent frame). This means that the referencing environment spans from the local environment to its enclosing environments. This will be better discussed in the functions (ADD REFERENCE HERE) part. Arguments against static scoping indicate the "looseness" of access of variable declarations and nested subprograms (and environments) are mimicking a kind of global scope.

Additionally, a variable's scope is bounded by the environment it is declared into, which is basically a collection of (symbol/variable name, value) pairs. Variables within blocks have a scope starting from the beginning of the declaration to the end of the scope, which is advantageous to readability as the reader can directly trace the origin of the variable in a top-down manner (except for free variables, which may be hidden in packages).

1.5.3 Expressions

This section discusses expressions, precedence rules, and type conversion.

General Syntax. Separating expressions can be done in either through whitespace or semicolons (critique here). Given two ways, writability suffers as the programmer must be aware of the implications of using both styles (although it has been a practice in other languages to use only one way).

Precedence rules. Precedence rules for R is provided by Table 1:

Related to this, R also does left to right associativity in expressions with operators of equal rank with the exception of the exponent and the leftward assignment operators (<-, <<-, =). Associativity can be changed using

Table 2: Operator Precedence

Description	**Operators**
Function Calls and grouping expression	({
Index and lookup operators	[[[
Arithmetic$\hat{**}$	ˆ, + (unary), - (unary), %% (modulus), *, /, +, -
Comparison$\hat{**}$	<, >, <=, >=, ==, !=, !, &, &&, |, ||
Formulas	~
Assignment$\hat{**}$	->, ->>, =, <-, <<-
Help	?

$\hat{**}$*Listed operators are also hierarchical on a left-to-right basis.*

parenthesis, which takes the highest priority, thereby overriding the default order of operations in a certain expression. This also intuitively reflects how expressions (especially in mathematics) are evaluated.

On how other operators work. In relation to this, arithmetic expressions are written in infix notation. This is advantageous in both readability and writability as it directly reflects how people usually write mathematical expressions. R also utilizes element recycling in arithmetic expression as such expressions are evaluated element-wise (technically 1 + 1 means the addition of two vectors of length 1):

```
> 1 + c(1,2,3)
[1] 2 3 4
```

Such a feature is good for data analysis as the feature allows for a cleaner syntax without having to do looping and much faster when vectors are long. However, it can negatively affect readability as such a feature may not be as intuitive to beginners and may be cause issues in larger systems and the programmer must take into consideration such a feature as arithmetic operators accept uneven length vectors, which is also detrimental to writability.

The operators !, &, and | apply element-wise on vectors similar to arithmetic operators. The operators && and || are often used in conditional statements and use lazy evaluation as in C: the operators will not evaluate their second argument if the return value is determined by the first argument. Having two different ways of writing and implementing logical operators is beneficial to readability as the reader can directly notice the difference and the contexts in which each is used, but is slightly disadvantageous to writability as the programmer needs to take note of how many symbols are needed for a specific use case as both element-wise and lazy-evaluated operators use the same character.

Operator Overloading. R does support operator overloading in the sense that you can override what a specific operator does. But as the language is not heavily inclined into the Object Oriented paradigm, use cases for operator overloading in C/C++ do not directly apply to the language. John Chambers (creator of S) and the R core team also stand on the perspective that certain operators such as "+" can only be used in their respective contexts only (in this case, in commutative operators, hence concatenation via "+" is not supported) as doing so might break related functionalities of that certain opeator. That is why R has provided the mechanism for user-defined operators, such as the following implementation for concatenation using "+":

```
> # using %<character>% as a user defined operator
> '%+%' <- function(x,y) paste(x,y,sep='')
> ``con'' %+% ``caten'' %+% ``ation''
[1] ``concatenation''
```

This preference to user-defined operators greatly benefits readability and reliability as such operators are distinct from the built-in operators as they are enclosed in % and it is assured that a certain operator works according to the context set either by the language rules or by the user which avoids the risk of misusing overloaded operators. However, this would negatively affect writability as the programmer must take into consideration further user-defined operators in writing code.

Type Conversion. As discussed earlier, R applies implicit coercion in most applications. To explicit convert a variable/object into a specific type, one can use the AsIs functions:

```
> class(3)
[1] ``numeric''
> as.complex(3) # conversion
[1] 3+0i
> as.integer(``a'') # warnings only
[1] NA
```

```
Warning message:
NAs introduced by coercion
> is.numeric(3) # explicit type-checking
[1] TRUE
> is.numeric(3L)
[1] FALSE
```

One can see that for certain cases, explicit conversions yield NAs and a warning message only. This negatively affects reliability as such features entail that the code is continuously executed with possible unwanted data which can greatly affect large systems of code.

1.5.4 Statements and Control Structures

Assignment Operators. R uses the following operators for assignments:

<- Local assignment operator; introduces new symbols/updates exsiting in the current frame.

<<- Operates on the rest of the environment, ignores local values, updates the first value found anywhere in the environment, or defines a new binding at the top-level.

-> or ->> Righthand-side assignment operators corresponding to the operators above.

= Lefthand-side assignment; rarely used as the style guide prefers arrow operators.

The presence of the equal sign as a possible way of assigning introduces readability issues as it may be confused with the equality sign used in mathematics, since the language is mostly tuned to a mathematically-inclined user base. Being able to declare on the righthand-side may also cause readability issues as most programming languages implement assignment statements in a variable-value order. Such a feature also negatively affects writability as the presence of multiple ways of assigning means that the programmer must be knowledgeable of each. The implication of a seemingly similar single (<) and a double(<<) arrow head is very distinct, hence the programmer must also be careful of using them

Compound and Single-operator Assignment Operators. R does not provide common compound assignment operators such as +=, -=, *=, /=, as well as single-operator assignments such as the iterator (e.g. ++) operators. Writability and readability benefits as the programmer needs to explicitly specify the operation as a more formal assignment statement, thereby lessening the constructs to remember and improving how the code is read. Issues on post-increment/pre-increment due to the placement of iteration operators are also avoided. On the other hand, as such operators are prevalent in modern programming languages, some programmers may the lack to be a disadvantage.

Assignment as Statement R supports assignments as part of a statement. However, it may be problematic as programmers should exercise caution in using it as the statements can change values, and might change in other expressions, might cause unintended side-effects.

Multiple Assignments R also allows for multiple assignments. Writability is benefited slightly as a programmer can directly chain assignments. However readability suffers the chaining may be too long and it may be confusing to read it especially with the different ways of declaring (although the interpreter makes sure that assignment operators cannot be used alternatively in multiple assignments.)

Conditional Statements. For two-way selection, R uses the if-else statement similar to C and C++. The statement also supports both single and compound clause forms:

```
# supports compound boolean expressions
# curly braces can be omitted in single clauses

if(condtion){
  statements
}
else if(condition){
  # else if is optional
  statements
}
else{
  # else is optional
  statements
}
```

Conditional statements are not vector operations. If the condition statement is a vector of more than one logical value, then only the first item will be used (recall that && and || are used). To evaluate multiple logical values, we used ifelse. On the other hand, if-else statements can be directly assigned to a symbol in a manner similar to a ternary operator but with the generic syntax of the statement:

```
> x <- if(5 == 0) 10 else 3
> x
[1] 3
```

One major readability issue on if-else statements is that nested single clauses are allowed. As tabs do not bear any significance to the language, the statement

```
if(condition)
    if(second.condition)
        statement
else
    statement # else statement of second.condition
```

introduces ambiguity as the format and implementation are different, especially in deeply nested conditional statements. An issue against the else if argument also point to its feature to implement a multi-way selection in what is supposed to be a two-way statement. Although (as you will see in the next expression), the else if construct is effective as the switch statement is quite limited as to what it can do.

Multiway selection is done using the switch function, which allows a variable to be tested for equality against a list of values. The snippet below demonstrates the syntax of a switch statement:

```
switch(expression, case1, case2, case3, ...)
```

The switch statement ma have any number of case statements which are implemented based on the following forms:

```
# based on integer value's position (1 to the maximum number
# of arguments), if value exceeds it nothing is returned
> switch(3,''one'', ``two'', ``three'', ``four'' )
[1] ``three''

# based on exact character value matching
```

```
> switch(‘‘a’’, a = ‘‘alpha’’, b = ‘‘bravo’’, c = ‘‘charlie’’
    ↪ , ‘‘default value’’)
[1] ‘‘alpha’’
```

The switch statement follows certain rules such as (a) if the value of an expression is not a character, it is coerced into an integer, (b) only the first match is returned in the case of multiple matches, and (c) for characer-related switch statements the language does not automatically provide a Default argument thus the user must define it or it raises an error.

As one may notice, the limited choice of data types to be used and having two ways of evaluating the same function (positional or exact matching). may cause writability issues as programmers need to be especially aware of the context of how they are going to use the switch statement. Also, case conditions are very limited as no comparative expressions can be used, which might also be seen as a disadvantage for users of modern programming languages. Furthermore, having the only the first result as output indicates user is not given the freedom to select where the statement is supposed to stop, which is advantageous in terms of reliability as one is assured that the switch statement yields a single value only. However, for users of languages such as C, C++, and Java, not having stopwords such as break could be an issue for readability. Default value are not required for integer-based switch statements but not for character-based, thereby negatively affecting writability as the programmer needs to be aware of the two types of implementations and the rules for each.

Iterative Statements. Similar to conditional statements, R adapts a more imperative-oriented approach to loops than a recursive style followed by functional languages. Although there are other special looping functions such as lapply, sapply, tapply, apply, mapply, the section shall exclusively discuss the fundamental iterative statements which are for, while, and repeat.

For loop. R’s counter-controlled loop is the for loop. The syntax is quite similar to C++’s, except that the iteration statement loops through the each value of a list or vector whose values can be integer, character, and logical.

```
for (value in vector){
  expressions
}
```

Such a design prevents the issue of possibly changing the loop variable as the for loop iterates over the elements of a vector/list rather than an integer-based loop variable. However, since there is no way of explicitly specifying initial and terminal values within the iteration statement, it is assumed that the beginning and the end of the vector/list to be iterated are the initial and terminal values. This negatively affects writability as the programmer must always make sure that the vector/list to be iterated has the correct initial and terminal values.

The for loop's loop variable still exists after the loop terminates and has the last values of the vector that was used in the looping process as a side effect. In common context, such a side effect is extraneous, especially as the variable has not been declared prior (indicating that it is only used in looping), and as R stores it in physical memory, multiple instances of for loops may affect how the code interacts with the hardware in terms of storage, especially if one considers that R users often deal with large datasets. If the loop variable has been declared prior, it is changed in the environment where the looping was done, hence the programmer needs to be aware of the loop variable to be used.

While loop. R's logically-controlled loop is the while loop. Similar to the for loop, its syntax is similar to C++'s. Test expressions are evaluated in a left to right manner:

```
while(test_expression){
  statement
}
```

It can be noticed that the while loop has pretest iteration statement, which is advantageous to reliability as it avoid the problem of unintentionally executing the loop body of posttest loops. One can also use the `break` and `exit` within the loop (which will be discussed in later).

Repeat loop. Repeat is a special loop in R which is used to generate an infinite loop, which are used when one does not know when to terminate (e.g. iterative algorithms):

```
repeat{
  statements
}
```

The only way to exit from the loop is to use the `break` statement. In relation to this, the statements in the loop body must be in a ablock, as at least two statements are needed to both perform some computation and test whether or not to break from the loop. One glaring issue here is that there is no guarantee whether the loop stops or not, which may be very troublesome in systems which utilize the loop.

On Object-Oriented Looping. R does not natively support object-oriented looping mechanisms such as iterators and foreach. There are packages which contain such mechanisms, however given that the scope of the critique is for the language itself, we do not consider them.

other looping statements

User-defined Stopping Mechanisms. R has two stopwords which are independent of conditional mechanisms. Only the loop body in which the stopword is located is exited/skipped:

break Exits from the innermost loop (the loop in which the break is defined).

next Continue to the next iteration of the loop and does not execute anything below it.

Sequential Statements. Sequential statements in R are separated either by a semicolon or new line. A semicolon always indicates the end of a statement whereas a new line may or may not indicate the end of a statement. In cases where statements are not finished (e.g. a statement with an open parenthesis only) and a new line is introduced, the interpreter picks up on the preceeding lines until an indicator that the statement is finished or a semicolon is introduced. If the statement is made in the interactive session, the prompt changes from $' >'$ to '+'. This has a similar critique to how expressions are separated, as writability suffers as the programmer needs to ensure that the appropriate separating construct is implemented. Readability can also be negatively affected as multiple statements in a single line can be posssible. A way of dealing with this is through following the style guide where new lines are used.

1.5.5 Functions (Procedures or Subprograms)

Functions in R are objects which take in input and return an output. In the language, functions are responsible for all actions, may it be arithmetic, assignment, looping, and others. As functions are first class objects, they can be bound to symbols (or variables), can be passed as arguments, and can be given different names. However, binding to variables are not necessary, as R functions are work like anonymous functions. Furthermore, a call to the function (that is, calling the function without parenthesis e.g. `> some.fxn`) returns a function. The syntax of functions is:

```
# functions with a single statement within the body may omit
    ↪  the curly braces
function(arguments) body

# assigning functions to variables
> square <- function(some.number) some.number^2
> square(10)
[1] 100
```

This section will further discuss the different parts of the function and how R implements each of them.

Parameter/Argument Passing R uses the call-by-value in passing arguments. Generally, supplied arguments behave like local variables initialized with the value supplied and the name of their corresponding formal argument. Changing the value of a certain supplied argument within the conext of some function does not induce any change or side effect of the same variable in the calling frame. The language also uses more than one parameter-passing method.

Formally defined function arguments are matched according to this order of priority:

1. **Exact Names**. The arguments will be assigned to full names explicitly given in the argument list. Full argument names are matched first.

2. **Partially Matching Names**. The arguments will be assigned to partial names explicitly given in the arguments list.

3. **Argument Order**. The arguments will be assigned to names in the order in which they were given in the function call.

Such a feature combines the advantages of positional and keyword matching. Exact names are effective as arguments can be passed in any order, and the disadvantage of having to know the exact name is remedied by the partial matching feature. However, partial matching is also dependent on which is similar on the ordering of characters lexically (left to right), so partially matched arguments might raise errors as the interpreter indicates two or more similar matches to a partially matched argument, get matched to an undesired formal argument, or get matched to the exact formal argument. Furthermore, one can mix named and positional argument, and the rule states that positional arguments are matched first and what is left is decided through argument order. In all of these cases, writability still suffers as the programmer still needs to create distinct variable names or, in the case of built-in functions, have knowledge on the parameter names and the order of parameters.

In addition to this, arguments with default values, if placed at the last parts of the function definition, can be ignored if arguments passed function calls are either incomplete or unnamed:

```
print.four <- function(first, second, third='3rd', fourth='4
   ↪ th'){
  cat(first, second, third, fourth)
}
> print.four('1st','2nd') # providing values for non-default
1st 2nd 3rd 4th
> print.four('1st', '2nd', 'third') # providing a new value
   ↪ for third
1st 2nd third 4th
> print.four('1st', '2nd', fourth='fourth') # exact name
   ↪ matching to fourth
1st 2nd 3rd fourth
```

Similar to the critiques on the order of priority, the programmer has to take note of which arguments have default values if they are to provide arguments for those without defaults, be aware of the risks of unintentionally overwriting the value of a default variable, and know the names of certain arguments with default values if they are to overwrite them. These issues are disadvantageous to writability and reliability.

R also supports a variable number of arguments with the formal argument being the ellipsis (. . .). Unlike other programming languages, the arguments passed can be of different types. Arguments with default values are initally omitted, and a valid function call with the ellipsis as a formal parameter must have at least one positional argument. The ellipsis argument can also be used when the arguments will be passed on to another function. In this context, arguments after ellipsis must be passed by name as they cannot be matched positionally or through partial matching. This may not be as intuitive in function calls as the arguments passed do not directly relate to named variables (a loop for the contents of the ... is usually used). Parameter passing is also position-based, and unless if there is an explicit binding process within the function, the provided values directly depend on how the function works.

As one can see in the function definition, argument types are not explicitly stated, hence type checcking during parameter passing with respect to the programmer's desired data type is not possible. This becomes an issue in both writability and reliability. The programmer must ensure that during function calls, the arguments to be passed coincide with the order and the type in which they are to be used inside the function, which also means that one must also take note of how the function is written and how it works. Also, as certain R features such as implicit coercion may not even raise errors if the incorrect data type is passed thereby possibly causing unexpected results.

Parameter Evaluation lazy implementations of function arguments arguments are boxed into promises which contain the expression passed as an argument and a reference to the environment values are evaluated transparently when their value is required

r does not state when promises are forced. order of evaluation affects side effects (promises) promises are evaluated aggresively as they do not outlive the function they are passed to name lookup will force promises in order to determine if the symbol is bound to the function

referential transparency referentail if any expression is replaced by its result, which does not induce side effects on other values in r, all function arguments are passed by value, all updates are only visible to specific functions

Functions as Arguments It is quite common for R functions to take other functions as arguments, as the language is heavily used in mathematical and statistical analysis and modelling.

A function's environment is the environment that was active at the time that the function was created. Any symbols bound in that environment are captured and available to the function. This combination of the code of the function and the bindings in its environment is called a 'function closure', a term from functional programming theory. In this document we generally use the term 'function', but use 'closure' to emphasize the importance of the attached environment.

When a function is called, a new environment (called the evaluation environment) is created, whose enclosure is the environment from the function closure. This new environment is initially populated with the unevaluated arguments to the function; as evaluation proceeds, local variables are created within it.

Functions as Arguments Deep bindings

```
> fxn1 <- function(){
+ val <- 1
+ fxn2 <- function() print(val)
+ fxn3 <- function(){
+ val <- 3
+ fxn4(fxn2)
+ }
+ fxn4 <- function(f){
+ val <- 4
+ f()
+ }
+ fxn3()
+ }
> fxn1()
[1] 1
```

since R function argument lists only entail the name and an optional default value, type checking the parameters cannot be done.

Local Variables, Free Variables, and Lexical Scoping. As mentioned in the previous sections, R is primarily a lexically scoped language. The implications of such scoping can be seen in how variables inside a function, namely local and free variables, are handled. Local variables are variables which are not part of the formal arguments that are declared within the function. Based on the lack of methods to allocate/create a static local variable, it can be said that local variables are dynamically allocated. Free variables are variables which are not part of the formal parameters and are not declared within the function. The following snippet of code demonstrates the two variables:

```
print.name <- function(pet.name){
# local variables
name.text <- ‘‘is my name.’’
age.text <- ‘‘is my pet’s␣name.’’
#␣concatenates␣the␣parameters␣provided
#␣name␣is␣a␣free␣variable
cat(name,␣name.text,␣pet.name,␣age.text)
}
```

Lexical scoping is used to determine the value of the free variables. A search for the value of the variable is conducted from the environment in which the function is defined, its parent environment, and goes on until the top-level environment (usually the global environment or the namespace of a package) or it reaches the empty environment. If a value is found, the search stops, else an error is thrown. Such a procedure is the reason why objects in R must be stored in memory, as functions carry a pointer to their respective defining environments. This goes bak to the S language, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment' of all " functions is the same.

One possible downside in terms of reliability is that searching is affected by the order of environments and packages, hence searching with many unordered packages may be quite slow. Hence one must be careful of the order of how the packages are included, with the frequently used or highly relevant at the top.

Return values. All functions in R return values. R returns any object based on last evaluated expression as the result of a function. Also, `return()` returns a value explicitly from a function, but it is rarely used in R. Functions can have multiple return values in a list with names associated with each. Accessing each can be done using `attach()`, which creates an environment that contains all the returned values:

```
> sample.fxn <- function() {
+  # returning multiple name-value pairs in a list
+ list(a=11, b=22, addfxn= function(x,y) x+y)
+ }
> return.env <- attach(sample.fxn())
> return.env$a
[1] 11
> return.env$b
[1] 22
> return.env$addfxn(1,2)
[1] 3
```

Functional Side Effects. In practice, it is bad style to use arguments to functions to cause side-effects -harder to debug -harder to prove that the program is correct

rediscuss <<- operator The operator will casue the interpreter to first search through the current environment to find the symbol. if the intepreter does not find the symbol in the current environment, it will next search throught the parent environment. the interpreter will recursively search through environments until it either finds the symbol or reaches the global environment. if it reaches the global environment before the symbol is found, then R will assign the value to the symbol in the global environment

Recursive Procedures R supports recursive procedures and/or functions. However caution must be exercised in using such procedures, especially in solving large problems. As the language treats functions as objects and objects are stored into physical memory, recursive functions may be memory intensive especially if there are deep nested function calls.

overloaded subprograms same name in the same referencing environment, different number, order, types of parameters or in its return if a function

(I don't think it exists natively, and existing implementations for truly overriden functions are not as simple as other programming languages)

generic subprograms

In R, methods for different classes can share the same name. These are called generic functions. Generic functions serve two purposes. First, they make it easy to guess the right function name for an unfamiliar class. Second, generic functions make it possible to use the same code for objects of different types.

By the way, the R interpreter calls the generic function print on any object returned on the R console. Suppose that you define x as: ¿ x ¡- 1 + 2 + 3 + 4 When you type: ¿ x [1] 10 the interpreter actually calls the function print(x) to print the results. This means that if you define a new class, you can define a print method to specify how objects from that new class are printed on the console. Some functions take advantage of this functionality to do other things when you enter an expression on the console

(wala pa ni) User defined overloaded operator

Exceptions. R includes the ability to signal exceptions when unusual events occur and catch to exceptions when they occur. It might seem strange to talk about exception handling in the context of environments, but exception handling and envirnments are closely linked. When an exception occurs, the R interpreter may need to abandon the current function and signal the exception in the calling environment.

try function tryCatch

2 C++

2.1 Purpose and Motivations

The purpose and motivation for the development of C++ programming language stems from the Ph.D thesis of Bjarne Stroustrup, the inventor of C++, where he used the Simula 67 language which is accredited as the first programming language to implement object-oriented programming. He found it incredibly useful for software development but was deterred from continued use due to the language being far too slow to be practical. Thus the development of *C with Classes* later to be renamed *C++*.

2.2 History (Authors, Revisions, Adoption)

2.2.1 Early Development

C with Classes was developed from the ideas of Bjarne Stroustrup with the intent of adding object-oriented programming to the C language. C was chosen because it is portable and does not sacrifice speed and low-level functionality. In this stage, the language already included classes, constructors, destructors, basic inheritance, inlining, default function arguments, strong type checking as well as all the features of the C language.

C with Classes was developed with its first compiler known as Cfront, which is derived from another compiler called CPre. It was initially designed to translate C with Classes code into ordinary C language. Cfront was also written in mostly C with Classes which made it a self-hosting compiler capable of compiling itself. Its first full version release was on 1985.

In 1983, C with Classes was renamed to C++ which was a direct reference to the increment operator that exists in C, emphasizing the enhancement and the addition functionality of functionality of the C language. During this time, virtual functions, function and operator overloading, referencing with the & symbol, the const keyword, single-line comments using two forward slashes, new and delete operators, and scope resolution operator were in development around this time.

In 1985, *The C++ Programming Language 1st Edition* reference book was published and was an incredible resource in order to learn and program in C++. In 1987, C++ support was added to GNU Compiler Collection version 1.15.3. In 1989, both the language and the compiler, Cfront, were updated and multiple features were added such as multiple inheritance, pointers to members, protected access, type-safe linage, abstraction and abstract classes, static and const member functions and class-specific new and delete functions.

2.2.2 International Recognition and Standardization

In 1990, *The Annotated C++ Reference Manual* was published and served as the standard before International Organization for Standardization (ISO) was used and added new features such as namespaces, exception handling, nested classes, and templates. In the same year the American National Standards Institute (ANSI) C++ committee was founded. In 1991, Cfront 3,0 was released; *The C++ Programming Language 2nd edition* was published, and the ISO C++ committee was founded. In 1992, the Standard Template Library (STL) was implemented for C++. In 1998, the C++ standards committee published the first C++ ISO/IEC 14882:1998, known colloquially as C++98. Problems were reported on the newly created standard and in 2003, they were revised and fixed accordingly. This change would be known as C++03.

In 2011, a new C++ standard was released. The standard was designed to help programmers on existing practices and improve upon abstractions in C++. The ideas for this standard were developed as early as 2005, creating an 8-year gap between standards. The Boost Library Project heavily impacted this revision and added in regular expression support, a comprehensive randomness library, a new C++ time library, atomics support, standard threading library, a new for loop syntax, the auto keyword, new container classes, better support for unions and array-initialization list, and variadic templates. This standard is known as C++11.

In 2014, another standard was released but is considered a minor revision of the C++11 standard. Following the naming convention, this is known as C++14. It included variable templates, generic lambdas, lambda init-captures, new/delete elision, relaxed restrictions on constexpr functions, binary literals, digit separators, return type deduction for functions, aggregate classes with default non-static member initializers, among many others.

In 2017, another standard was published and added in the following features: fold-expressions, class template argument deduction, non-type template parameters declared with auto, initializers for if and switch, u8 character literal, simplified nested namespaces, using-declaration declaring multiple names, made noexcept part of type system, new order of evaluation rules, guaranteed copy elision, lambda capture of *this, constexpr lambda, __has_include, and among others. The standard is known as C++17.

At the time of writing, the C++ Standards Committee, is processing the finalization of another standard for the year 2020, known as C++20.

2.3 Language Features

2.4 Paradigm(s)

2.5 Language Evaluation Criteria

3 Workarounds

```
# used to execute functions defined within the "Struct"
execute <- function(func ,env=parent.frame()){
    # print(e)
    # print(f)
    environment(func) <- env
    func
}

# "Struct" constructor, creates an environment with the
   ↪ contents with or without values
constructor <- function(env.data, val.list=NULL){
    with(env.data,
         {
             final.env <- new.env()
             # final.list <- list()
             iterval <- 1
             for(i in 1:length(env.data)){
                 var <- env.data[[i]]
                 print(var)
                 if(length(var) == 1){
                     print(var)
                     if(!is.null(val.list)){
                         final.env[[env.data[[i]]]] <- val.list[[
                             ↪ iterval]]
                         iterval <- iterval+1
                     }
                     else
                         final.env[[env.data[[i]]]] <-NULL
                 }
                 else{
                     final.env[[env.data[[i]][[1]]]] <- env.data
                         ↪ [[i]][[2]]
                 }
             }
             # print(env.data)
             # for(pair in final.list){
             # final.env[[pair[[1]] ]] ->pair[[2]]
             # }
             return(final.env)
         }
    )
}

# Defining function for "Structs". This is where the
    ↪ programmer provides variable names and functions to be
    ↪ used.
Struct <- function(...){
    varlist <- list()
    fxnlist <- list()
    for(var in list(...)){
         if(length(var)>1){
             if(length(fxnlist) == 0)
                 fxnlist <- list(var)
             else
                 fxnlist <- list(fxnlist, var)
         }
         else{
             varlist <- c(varlist,var)
         }
    }
    if(length(fxnlist) > 0){
        c(varlist, fxnlist)
    }
    else{
        varlist
    }
}
```