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)

## 1.5 Language Evaluation Criteria

### 1.5.1 Data Types

TAs R abides by the principle that everything is an object, there are no "data types", per se. Hence this section shall cover and critique the fundamental data objects 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 | char, "another char" | character |
| numeric (real) | 2, 1.0 | numeric |
| integer | 1L, 22L | integer |
| complex | 1 + 2i, 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 CROSS REFERENCE TO RESERVED WORDS 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 is 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

Objects must generally be stored in physical memory

**Names.** Names in R are case sensitive, which 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.

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.

!!!!!!!!!!!!!!!!!!!!1 address - aliasing SO FAR WALA MAN, value, type, lifetime, scope R associates attributes to all data structures, used as books for many purposes !!!!!!!!!!!!!!!!!!!!!!!!!!

Aliasing does not exist in R as the language allows for copying objects

Explicit type definitions are not required for variables and functions - reliability, issue with katong vector INFERENCE DURING CONVERSION R checks type compatability during runtime, and performs conversions, when possible

R is statically/lexically scoped until not

the values of free variables are searched for in the environment in which the function was defined

Normally when discussed in the context of R lexical scoping means that free variables in a function (i.e. variables that are used in a function but not defined in the function) are looked up in the parent environment of the function, as opposed to the environment of the caller (also referred to as the parent frame) but there are no free variables in with.default so the example does not illustrate a violation of lexical scoping in that sense. With the undertones of functional programming languages, scoping is lexical,

```
> x <- 5
> y <- function() x
> z <- function(){
+   x <- 0
+   y()
+ }
> z
[1] 5
```

HAS SUBPROGRAM DEFINITIONS TO CREATE NESTED STATIC SCOPES?

Implicit coercion is done during operations. This can be seen on the given example:

```
> vec1 <- c(1, 'a', 3.11, TRUE)
> vec1
[1] ``1''    ``a''    ``3.11''    ``TRUE''
> class(vec1)
[1] ``character''
```

Dynamic Binding, hence binding occurs during execution or can change during execution of the program. This has advan

dynamic typed high cost in dynamic type checking and interpretation type error detection by the compiler is difficult

dynamic evaluation text to unevaluated expressions using quote() variable substitution substitute() partial substitution bquote() expression to text substitute()/deparse()

### 1.5.3 Expressions

Syntax Separating expressions can be done in either through whitespace or semicolons (critique here)

Precedence rules

Table 2: Operator Precedence

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

Associativity can be done using parenthesis, which takes the highest priority, thereby overriding the default order of operations in a certain expression.

Curly braces are used to evaluate a series of expressions and return only the last expression, very much similar to a function scope in languages such as C, C++, and Java. However, braced expressions can be implemented independently. THe contents of the curly braces are evaluated inside the current environment; a new environment is created by a function all but not by the use of curly braces. (!!!!!!!!!!!!)

Overloaded Operations Type conversions, relational, and boolean operators && & —— —, implicit conversions are done most of the time, but explicit conversions can be done

element-wise operation, whether arithmetic or boolean operators due to the functional nature of R

element recycling is also done, e.g. 1 + c(1,2,3) = c(2,3,4) good for data analysis purposes as the functional feature allows for a cleaner syntax without having to do looping and much faster when vectors are long

The operators & and — apply element-wise on vectors. 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.

Expressions

### 1.5.4 Statements and Control Structures

Assignment Statements Procedure Calls Conditional Statements if() else if() else

can be declared, a la ternary operator

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

in R, 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. To evaluate multiple logical values, we used ifelse.

Multi way selection is done using the switch function (insert some code here, with default and all)

```
> switching.function <- function(x){
+   switch(x,
+     a = ``alpha'',
+     b = ``bravo'',
+     c = ``charlie'',
+     ``default value''
+  )
+ }
```

Iterative Statements for - generic range, a la python 1:10 naa puy katong object oriented seq_along (based on the length of the object) a la c++ nga for(letter in x)

looping statements lapply, sapply, tapply, apply, mapply

while - generic conditions are evaluated LR

repeat - generates an infinite loop; used when one does not know when to terminate e.g. iterative algorithm only way to exit is to use BREAK CANNOT BE GUARANTEED IF LOOP STOPS

stop words break - exit from the loop next - continue to the next iteration of the loop

a variable var that is set in a for loop environment is changed in the calling environment

No inherent support for object-oriented looping such as iterators or foreach, but has through additional packages. (do we consider vanilla R?? WE NEED TO DECLARE A SCOPE INTO THIS)

Concurrent Statements Sequential Statements

<- local assignment introduces new symbols/updates exsiting in the current frame; is local <<- operates on the rest of the environment, ignores local values, updates the first value anywhere in the environment, or define a new binding for x at the top-level -> ->> can be done = can be done sometimes

### 1.5.5 Procedures and Subprograms

Subparameters Simple Call returns Recursive Procedures Coroutines

Exception Handling Paramater-Passing Methods Parameters which are subprograms

Design Issues and Functions Overloaded and Generic Subprograms User-defined Overloaded operators

RELATED TO FUNCTIONS and lexical scoping environment in the context of R is a collection of (symbol, value) pairs function + environment = closure

how? if the value of a symbol is not found in the environment in which a function is defined, then the search is continued in the parent environment the search continues down the sequence of parent environments until we hit the top-level environment; this is usually the global environment (workspace) or the namespace of a package after the top-level environment, the search continues down the search list until we hit the empty environment

if symbol is not found once the empty environment is arrived at, then an error is thrown

downside is searching is affected by the order and order of packages

when a function is defined in the global environment and is subsequently called from the global environment, then the defining environment and the caling environment are the same. This can sometimes give the appearance of dynamic scoping.

Lexical Scoping in R has consequences beyond how free variables are looked up. In particular, it is the reason that all objects in R must be stored in memory in R. THis is beause all functions must carry a pointer to therir respective defining environments, which could be anywhere. in 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.

FUNCTIONS INSIDE FUNCTIONCS (ELABORATE)

functions basic structure function(arguments), which can be stored in a variable functions are first class objects, hence they can be bound to symbols (or variables), can be passed as arguments, and can be given different names

return() returns a value explicitly from a function, but it is rarely used in R

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

parameters - specifying default parameter values and a variable number of parameters passing AND MATCHING is done by position or by name (EXPOUND ON THIS) PARTIAL MATCHING ALSO omits parameters with default values a valid call must have at least one positional parameter

order of priority in parameters: 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 give nin the arguments list 3. Argument order. the arguments wi.l be assigned to names in the order in which they were given.

lookup is context sensitive

... argument arguments after ellipsis must be passed by name as they cannot be matched positionally or through partial matching default arguments values can be expressions and evaluated within the same environment as the function body, can use internal variables in the function's body It is often convenient to specify a variable-length argument list, hence the use of the ellipsis

As a way of addressing the issue of not having a clear knowledge as to what the arguments are, R stores function properties such as the list of arguments used. This can be accessed by the args() function.

Return values R, will simply return the last evaluated expression as the result of a function.

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.

Anonymous functions Are usually passed as arguemts to other functions, similar to other functional programming laguages, but can also be directly implemented with parameter/s.

Exception

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

Generic functions when using generic functions, you cannot specify the argument name of the object on which the generic function is being called. you can still specify the names for other arguments

Side effects all functions in R return a value. some functions also change the variables in the current or other environment 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

# 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 comprehen-

sive 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