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 hard-copy 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

This section shall cover and critique the fundamental data objects and values found in R.

**Atomic Objects** As R abides by the principle that everything is an object, there are no "data types", per se. However, 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 to stand for True and False in logical values - this causes ambiguity in the sense that a single construct is implemented in two way, and it may be confused with a variable, much like this snippet of code which does not produce errors when T is given a value:

```
> T #T as a logical value
[1] True
> T <- 22 #T as a variable, does not raise errors/warnings
> T
```

```
[1] 22
```

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.

# 2 C++

## 2.1 Purpose and Motivations

## 2.2 History (Authors, Revisions, Adoption)

## 2.3 Language Features

## 2.4 Paradigm(s)

## 2.5 Language Evaluation Criteria