

A Comprehensive Comparison of C++ and R
©2020 by Cang, K. and Navarez, A.
November 2020

Contents

1	R	2
1.1	Purpose and Motivations	2
1.2	History (Authors, Revisions, Adoption)	2
1.2.1	S, the precursor of R	2
1.2.2	R	2
1.3	Language Features	3
1.4	Paradigm(s)	3
1.5	Language Evaluation Criteria	3
1.5.1	Data Types	3
1.5.2	Binding, Scoping, Referencing	9
1.5.3	Expressions	10
1.5.4	Statements and Control Structures	11
1.5.5	Procedures and Subprograms	11
2	C++	12
2.1	Purpose and Motivations	12
2.2	History (Authors, Revisions, Adoption)	12
2.2.1	Early Development	12
2.2.2	International Recognition and Standardization	13
2.3	Language Features	14
2.4	Paradigm(s)	14
2.5	Language Evaluation Criteria	14

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 Insightful 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 named 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
- 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 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(vec3)
[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 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

WHERE TO PLACE? objects must generally be stored in physical memory

Because the underscore was not allowed as a variable character, the convention arose to use dot as a name separator. Unlike its use in many object oriented languages, the dot character in R has no special significance, with two exceptions. First, the `ls()` function in R lists active variables much as the `ls` Unix shell command lists directory contents. As the `ls` shell command does not list files that begin with a dot, neither does the `ls()` function in R function show variables that begin with dot by default. Second, `...` is used to indicate a variable number of function R uses `$` in a manner analogous to the way other languages use dot.

R has several one-letter reserved words: `c`, `q`, `s`, `t`, `C`, `D`, `F`, `I`, and `T`.

(Actually, these are not reserved, but it's best to think of them as reserved. For example, `c` is a built-in function for creating vectors, though you could also create a variable named `c`. Worse, `T` and `F` are not synonyms for `TRUE` and `FALSE` but variables that have the expected values by default. So someone could include the code `T <- FALSE`; `F <- TRUE` and reverse their meanings!) names case sensitive, keywords? mixed case special words - to aid readability

Variable names are declared based on this BNF:

```
<variable> ::= <first><second><succeeding>*
<first> ::= A-Z — a-z — .
<second> ::= A-Z — a-z — . — _
<succeeding> ::= A-Z — a-z — . — _ — 0-9
```

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

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

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?

Dynamic Binding – occurs during execution or can change during execution of the program

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 Precedence rules 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
```

Iterative Statements Concurrent Statements Sequential Statements

`j-` local assignment introduces new symbols/updates existing in the current frame; `is local j-` 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 `-j` can be done `=` can be done sometimes

1.5.5 Procedures and Subprograms

Subparameters Simple Call returns Recursive Procedures Coroutines

Exception Handling Parameter-Passing Methods Parameters which are subprograms

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

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

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 aggressively 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 referential 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 is done by position or by name omits parameters with default values a valid call must have at least one positional parameter

lookup is context sensitive

... argument arguments after ellipsis must be passed by name 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

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 lineage, 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