A Comparison of C++ and R

Cang, K. and Navarez, A.

Submitted in partial fulfilment of the requirements of
CMSC 124

Department of Computer Science
University of the Philippines Cebu
Philippines
December 31, 2020

Contents

1 R

1.1 Purpose and Motivations

Fundamentally, R is a dialect of S. R is a programming language and software environment that is focused on statistical analysis, graphics representation and reporting. 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 writing this paper, 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)

R is a multiparadigm language. Websites such as wikipedia ascribe multiple paradigms to R, however the most distinct and verified paradigms are object-oriented and functional as stated a paper made by John Chambers (2014), the creator of S. As we are not able to verify the claims of such sites (as they did not provide any external resource on their reasons), we shall only considered the two mentioned paradigms in this paper.

R's functional characteristics:

- First-class functions
- Writing and implementing pure functions are encouraged, although impure functions can be done
- Lazy evaluation of arguments

R's object-oriented features:

- S3 and S4, which are objects that adhere to the OO paradigm (immutable; supports generic functions, inheritance, etc.). S4, in particular, is influence by Common Lisp and Dylan's object-oriented style.
- Packages which provide other types of object-oriented feature (e.g. R.oo, mutatr).

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 but objects with different types/classes. However, in the paper we use the term data type to refer to these types. 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 table 1:

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.

Table 1: Atomic Classes of Objects in R

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

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 coerced 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 ways.

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).

Among R's composite objects, the most prominent are vectors, matrices, 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
> mat <- matrix(1:4, nrow=2, ncol=2)
> vec3 <- as.vector(vec) #explicit coercion
> vec3
[1] 1 2 3 4
> 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 name is not self-documenting and the need to remember each implementation.

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 the use case for 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=2) #using vector()
> list2 #empty list of specified length
[[1]]
```

```
NULL

[[2]]
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 negatively affects writability as the programmer needs to know two ways of implementing lists (especially if one needs a list of NULLs), as well as affects readability due to the usage of a function of another data type even with the passing of the "list" argument.

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.

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, so `someobject.value` in C or C++ becomes `someobject$value` in R. These features 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. This negatively affects writability as the programmer needs to know what these words are in naming variables. 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, thereby reducing reliability:

```
> 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 that could exist in both function and variable forms.

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 ordering 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 section (1.5.5). 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. 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). Readability also suffers as multiple expressions can be written in a single line, which leads to obscure-looking code.

Precedence rules. Precedence rules for R are provided by Table 2:

Table 2: Operator Precedence

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

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

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 which follow the right to left associativity(<-, <<-, =). Associativity can be changed using 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 and reliability as such a feature may not be as intuitive to beginners and may be cause issues in larger systems. 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 `as()` for converting and `is()` to verify if a certain variable/object is of a certain type:

```
> 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 an Expression R supports assignments as part of an expression. 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 single-claused 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 may 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). This 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.

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.

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` 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 block, 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)

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

Functions. Functions in R are objects which take in input and return an output. In the language, functions are responsible for all actions and computations, 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
function(some.number) some.number^2
> square(10) # function call with variable
[1] 100
```

One can see that the function definition does not need to specify any data types for the arguments and the return type, as all functions in R return some value. This is advantageous for writability as the programmer does not need to specify data types, and disadvantageous as function calls

may receive various objects. Furthermore, not specifying the return type can be detrimental to readability as the reading the code does not really indicate what it would return (the function name may have an indication, but does not exactly tell). Furthemore, reliability may also be an issue due to some features of R, which will be explained in the next parts.

Parameter/Argument Passing and Matching 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, and all actual arguments that are not matched into the formal arguments are included into the ... argument. 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. In a nutshell, R does lazy implementations of function arguments. This means that values are not evaluated when needed. This is done by boxing arguments into promises which contain the expression passed as an argument and a reference to the environment. Values are then evaluated transparently when their value is required.

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. The language uses deep bindings in its implementation.

When a function is called, a new environment (called the evaluation environment) is created, directly reflecting the environment that was active at the time that the function was created. Hence, variables within that environment are also available to the function. A demonstration is provided by the following code snippet:

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

> fxn1()
[1] 1 # deep binding; 4 if shallow binding
```

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. This may cause readability issues for programmers who are used to adding the return

word in functions. 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. Recall the <<- operator. The operator can cause changes in the parent environments as it causes the interpreter to search for the symbol from the function's environment towards the global environment. It binds the new value to the symbol that is found first, else it assigns the value to the symbol in the global environment. Such a feature may be detrimental to reliability as unintentional changes may occur, and as the searching process goes through environments, it may be difficult to trace where the changes are made. Relating on the early critique, writability suffers as the programmer needs to be careful in using such operators provided that it is very similar to the local <- operator with very different implications. In practice, it is bad style to use arguments to functions to cause side-effects.

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.

Generic functions in R. Generic functions in R serve two purposes: firstly, it makes it easy to determine the correct function name for a certain unfamiliar class, secondly, it promotes code reusability for objects of different tyles. An example of which is when the interpreters calls the print function on any object return on the console. Contrary to its name, creating generic functions are more similar to creating overloaded functions. We take the print function for example - a definition to a print function (called in this

context as method) for a new class can be done:

```
# creating an environment with contents
> envi <- new.env()
> envi$name <- "environment1"

# creating a new class for the environment
> class(envi) <- "customenvironment"

# creating a print method for objects of customenvironment
   ↪ class
> print.customenvironment <- function(some.envi) some.envi$
   ↪ name
> print(envi)
[1] "environment1"
```

Exceptions. R includes the ability to signal exceptions when unusual events occur and catch to exceptions when they occur. When an exception occurs, the R interpreter may need to abandon the current function and signal the exception in the calling environment. Certain functions used for exception catching are the `try()` and `tryCatch()` functions. User-defined errors can also be done using `stop()`, which stops the execution of the function and displays an error message.

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

As a language, C++ as a language has the following features:

- C++ is an open ISO-standardized language - standardization helps with teaching and learning the code

- C++ is a compiled language - compilation allows for direct translation into machine code allowing for faster and more effecient programs
- C++ is a strongly-typed unsafe language - the programmer is expected to know what they are doing and allows for a higher degree of control
- C++ supports both manifest and inferred typing - allows for flexibility and avoids unnecessary verbosity
- C++ supports static and dynamic type checking - allows for flexibility although, most C++ type checking is static
- C++ encompasses multiple programming paradigms - offers support for a variety of paradigms
- C++ is portable - so long as there is a compiler for the device, C++ is capable of running with little to no problems
- C++ is upwards comapatible with C - directly built with the C in mind, all libraries in C are compatible with C++
- C++ has library support. - extensive libraries have been built for C++, are constantly updated, and have incredible support

2.4 Paradigm(s)

As stated in the language features, C++ is capable of a multitude of programming paradigms or simply, a multi-paradigm language. The most notable being Object-oriented, Procedural, Imperative, and Generic programming paradigms:

- C++ was initially deveoped with the OOP paradigm in mind, and so contains all the characteristics of OOP: Encapsulation, Inheritance, and Polymorphism. OOP also focuses on the use of objects that attempt to manifest the behaviors, and characteristics of real life objects. C++ contain all of these features and more.
- C++ is considered to be an Imperative programming language because it allows programmers to give an ordered list of instructions to perform a task. The program is told what to do, when to do it, and in what order to them to achieve the solution needed for the problem at hand.

- C++ is also considered to be a Procedural programming language due to its ability and the support of the use of the concept of procedures and subroutines familiarly known as functions in C or C++.
- C++ is also considered to be a Generic programming language as it is capable of abstraction and the use of creating skelton algorithms.

2.5 Language Evaluation Criteria

2.5.1 Data Types

C++ has a vast collection of primitive data types that a programmer is able to use. The table 3 represents a short summary of all the primitive data types available to the user.

Table 3: Primitive Data Types in C++

Data Type	**Keyword**	**Size**
Integer	int	Between 2-8 bytes
Character	char	1 byte
Boolean	bool	Typically 1 byte
Floating Point	float	4 bytes
Double Floating Point	double	Between 8-12 bytes
Void	void	N/A
Wide Character	wchar_t	2 or 4 bytes

C++ also allows for the use of modifiers to explicitly call data types of a variety of ranges. The available modifiers in C++ include:

- signed
- unsigned
- long
- long long
- short

Table 4: Modified Data Types, their ranges, and their size allocation in C++

Modifier	**Range**	**Size Allocation**
signed char	-(2^7) - (2^7)-1	1 byte
unsigned char	0 - (2^8)	1 byte
signed short int	-(2^15) - (2^15)-1	2 bytes
unsigned short int	0 - (2^16)	2 bytes
signed int	-(2^31) - (2^31)-1	4 bytes
unsigned int	0 - (2^32)	4 bytes
signed long int	-(2^31) - (2^31)-1	8 bytes
unsigned long int	0 - (2^32)	8 bytes
signed long long int	-(2^63) - (2^63)-1	8 bytes
unsigned long long int	0 - (2^64)	8 bytes
float	N/A	4 bytes
double	N/A	8 bytes
long double	N/A	12 bytes

The table 4 shows the ranges of values of modified data types as well as their size allocation.

It is important to note that it is possible to call various data types based on specific modifiers and have them behave and be saved as one universally accepted data type. For example, the standard format of initializing an integer is:

int x;

The text above will be read by the compiler as a call for a signed integer with the variable name of x. But it is also possible to do this with the following:

signed x;
signed int x;

C++ has made initializations of various data types as flexible as possible but has also affected the readability of code. It presents much more freedom in the ability for a user to write what they intend to write but because of

the multitude of ways for even a single variable to be called it makes it that much more hard to read and fully understand what the code is meant to be doing.

Another example to this problem is the ability to write modifiers in any combination. In most standards, writings, and even in online references the declaration of a variable in a code should be written as:

unsigned long long int x;

This will be read as a call for an unsigned long long integer the variable name of **x**. But the following is also possible:

long unsigned int long x;

This too will be accepted as a call for an unsigned long long integer with the variable name of x. This shows that C++ allows for a great versatility in allowing the user to choose the methods that they wish to implement for their code but is also causes problems with the code's readability for other programmers. Although it is a testament to how effective the C++ compiler is at identifying the keywords and how they should be processed and used, but the standard of *datatype variable_name* must always be followed, so the following code:

long unsigned int x long;

will not be read as an unsigned long long integer and will instead produce a syntax error.

Other available data types that are built upon the primitive data types (derived data types) include Functions, Arrays, Pointers, and References. C++ also allows for user defined abstract data types such as Classes, Structures, Unions, Enumerations, and the Typedef-defined data type. All these will be further discussed in the following sections.

2.5.2 Binding, Scoping, Referencing

This section covers the C++ language's ability of binding, scoping, and referencing.

Binding C++ covers the ability for both compile-time binding, and execution-time binding. By default, and because C++ is a compiled language, the binds are all made during compile-time (early time compilation)

this means that most, if not all, binds must be declared explicitly in the code wherever required.
Binds to variable names in C++ also have very specific rules. They include the following:

- The variable name must begin with a letter or an underscore.
- The variable name may contain letters, numbers, and underscores.
- Spaces and special characters are not allowed and will result in an error.
- Variable names are case sensitive.
- Variable names can range between 1-255 characters.
- Reserved words or keyword are not allowed to be variable names.

It is also a given that when binding a variable name, one must also include the data type of said bind.

These rules allow a degree of freedom for the naming of variables by the programmer. The writabilty of the code is preserved because there is only one way of declaring variables. The only downside to variable naming is how the programmer decides to name their variables, so this does affect readability to a certain extent, especially if the programmer does not follow naming conventions and suggestions.

Reserved Words All programming languages have certain reserved words that are made available to them in order to create the codes they need to create. Words such as *if, for, while, do, continue, break, return, True, False* are only among of the few keywords that have a specific role in C++. These reserved words are case-sensitive and will not be recognized by the compiler if used any differently. The downside of having so many reserved words is not being able to memorize all of them and their specific uses as well as the possibility of collisions in code.

It also worthy to note that when a user-defined function is declared or defined, one cannot use the name of said function as a variable name (case-sensitive) in the same scope, further improving the readability of a specific line of code in C++. Following this idea, it also gives full control to the programmer to decide the method to be performed by the program. Unfortunately, this does affect the writability since this does limit the ideal

situation of the programmer to bind a specfic name that they may be comfortable with.

Type Conversion C++ is able to perform both implicit type conversion (coercion), and explicit type conversion (casts). The following is an example of such these two processes:

```
#include <iostream>

using namespace std;

int main(){
  double x;
  int y;

  x = 12; // a coercion of an integer type to a double type
  y = (int) 2.25; //a cast of an float type to an integer
     ↪ type

  cout<< x << "\n" << y << \n << typeid(x).name();

  return 0; //safely ending main()
}
----------------------------------------
expected output:
12
2
d
```

As seen above, C++ has the ability to convert various data types during compile time. This allows for a great flexibility for the programmer to write programs that are strongly-typed and have full control over what they want the program to perform. The example above of *y = (int) 2.25* is a type of conversion called a *narrowing conversion* wherein the accuracy of a value is lost. This process can be done both implicitly and explicitly. Generally, when coding in C++ one should attempt to avoid narrowing conversions, but the code will continue to do so when explicitly stated such as the example above. When accuracy is important, it is best to avoid narrowing conversions either as coercion, or cast. On the other hand, *widening conversions* also known as *promotions* or conversions that place a lower sized data type into a higher

sized data type, allow for a greater control over accuracy as well as being able to represent a higher value than the previous size. C++ allows for both widening coercions and widening casts. As the process is considered safer, loss of information is not an inherent problem. The table 5 shows all the possible promotions among the various primitive data types and their modifiers.

Table 5: Possible Widening Conversions without loss of information

From	**To**
Any signed or unsigned numeric type except long long & __int64	double
bool or char	Any other type
short or wchar_t	int, long, long long
int, long	long long
float	double

Scoping C++ has a variety of defined scopes, and the extent of lifetimes in a scope, as well as intuitively knowing what can access what and where they are according to its scope. C++ generally have six forms of scopes. They are

- Global scope - has the longest lifetime; and exists implicitly in the global space and extends from the beginning of its declaration until the end of the file; a common example being global variables
- Namespace scope - it is the scope within a namespace that is outside any class, enum definition, or function block; visible from its declaration until the end of the namespace; a common example being *using namespace std;*
- Local scope - names declared within a function or lambda; visible only at the point of declaration until the end of the function or lambda; also commonly known as block scopes denoted by code found in between brackets ;
- Class scope - anything and everything found and defined in a class has a class scope (class members); they exist regardless of point of

declaration and may be *public, protected, or private*; only public and protected members can be accessed from member-selction operators (. & ->)

- Statement scope - statement scopes exist in control statements and loop statements such as *for, if, while, & switch* statements.

- Function scope - function scopes are visible even before their point of declaration; mostly known as labels; common in *switch case statements & goto labels*; usually denoted by a colon (:)

These specific scopes allow for the ease of processing information throughout the entirety of code and allows the programmer to write code that behaves in accordance to what is needed and specified. Processes involving the use of scopes usually involve the use of the scope-resolution operator, double colon (::). This allows for greater control of definitions especially involving classes, global variables with the same name as local variables, and even namespace variable names with the same name as local variable names. The simplicity of the operator makes it easier to write the code for its specific purpose and deals with problems in regards to scoping.

References In C++, there are two methods of creating references on various data types, the use of pointers and the use of references. Pointers are variables that store memory adresses of an object. They are mainly used for

1. Allocation of new objects to the heap

2. Iteration over elements in arrays or other data structures

3. Passing functions to other functions

References are similar to pointers in which they too hold the address of an object in the memory but unlike a pointer, a reference cannot be changed to refer to another object or set to null. They must be initialized with the object they are referring to. References present a convenience especially when creating references to complicated structures such as classes, structs, and other user-defined data structures. The following C++ code shows an example of the use of raw pointers and references:

```
#include <iostream>

using namespace std;
```

```
void toTen(int *x){ //a function written to receive a
    ↪ reference to a variable
    *x = 10; //a reference that changes the value the
        ↪ argument to 10
}
int main(){
    int x; //integer named x
    x = 52; //assigning 52 to x;
    int *y = &x; //an integer pointer named y pointing at
        ↪ the address of integer x
    int &z = x; //an integer reference to integer x
    toTen(y); //passing a pointer into the toTen function
    cout<< x <<endl; //x should now have the value its
        ↪ holding changed to 10
    z = 25; // changing the value of integer x through the z
        ↪  reference
    cout<< x <<endl; //printing x must give the new changed
        ↪ value through the reference
    x = 36; //changing the value of x to 36
    cout<< z <<endl; //printing reference z in order to show
        ↪  the referencing

    return 0; //safely ending main()
}
  ----------------------------------------
  expected output:
  10
  25
  36
```

As shown in the code example above, references and pointers are capable of causing changes to the data they are pointing at or referencing. This allows a particularly powerful tool for programmers to update data even through different scopes or processes. The downside of such a convenient process is the readability of the code. As it only uses simple symbols (that also have various meanings or contexts), understanding its behavior becomes much harder and may become prone to reading mistakes or misunderstandings in

behavior and potential use.

2.5.3 Expressions

General Syntax In C++, all declarative sentences or instructions must end in a semicolon (;) as seen in earlier examples. Scoping is generally indicated through the use of curly brackets ({}). C++ is strict with these concepts in all versions and will fail to compile whenever there is a missing semicolon in areas where it is required. This strict rule does affect the writability of code, especially for programmers coming from languages that do not require these indicators. However, because it is an excellent notation for the separation of instructions, it makes code more readable and generally understandable. It also increases the language's reliability because it is clearly denotes the end of an expression and prevents other syntactical mistakes. The use of a comma (,) also denotes a separation of similar statements but are not immediately related to one another. A example is the use in the code

int x,y,z;

In the example, the comma is used to separate variable names but all are related to being a declaration of an integer data type. The idea in the ability to process this data is intuitive, much like in the english language, the comma is used to separate clauses that are related to one central idea and is applied similarly in C++. This allows for a very writable and readable snippet of code.

Precedence rules C++ precedence rules are extensive because of how many features are found in the language. Being a multi-paradigm language, there is a variety of methods to do certain tasks, but they still have to follow the precedence rules in order to avoid ambiguity in all its forms. Table 6 and its continuation on Table 7 shows the precedence rules found in C++:

As seen from the tables, the precedence found in C++ is facet of the language that allows for great control and maneuverability of the programming language and the code one is able to make. All these in an effor to avoid problems in regards to ambiguity of the language and to isolate certain processes from others. One is also able to see operators, and their overloads, and what behaviors they perform that a programmer is able to use an any given time. For example the operator ++ is an single increment operator added in postfix or prefix to a variable name to increase its value by one. This can also be written as **variable_name+=1; or variable_name =**

Table 6: Precedence rules of C++

Precedence	**Operator**	**Description**	**Associativity**
17	::	Scope resolution	Left-to-Right
16	a++, a– type(), type{} a() a[] . , ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member acess	Left-to-Right
15	++a,–a +a,-a !, (type) &a *a sizeof co_await new, new[] delete, deletep[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Address-of Dereference Size-of await-expression Dynamic memory allocation Dynamic memory deallocation	Right-to-Left
14	.*, ->*	Pointer-memeber	Left-to-Right
13	a*b, a/b, a%b,	Multiplication, Division, and Remainder,	Left-to-Right
12	a+b, a-b	Addition and Subtraction	Left-to-Right
11	>>, <<	Bitwise Left-shift and Right-shift	Left-to-Right
10	<= >	Three-way comparison operator	Left-to-Right

variable_name + 1;. There is some nuance to the implementation of each operator such as those that affect runtime, but generally, all three operators perfrom the same task, add a value of one to the current value found in the variable and assigning this new value to the variable. C++ is full of these

Table 7: Continuation of Precedence Rules

Precedence	**Operator**	**Description**	**Associativity**
9	<, <= >, >=	Relational operators Relational operators	Left-to- Right
8	==, !=	Relational Operators	Left-to-Right
7	&	Bitwise AND	Left-to-Right
6	^	Bitwise XOR	Left-to-Right
5	—	Bitwise OR	Left-to-Right
4	&&	Logical AND	Left-to-Right
3	——	Logical OR	Left-to-Right
2	a?b:c throw co_yield = +=, -= *=, /=, %= >>= , <<= &=, ^=, —=	Ternary conditional Throw operator Yield-expression Direct assignment Compound assignment by sum and difference Compound assignment by multiplication, division, and remainder Compound assignment by bitwise left-shift and right-shift Compound assignment by bitwise AND, XOR, and OR	 Right-to- Left
1	,	Comma	Left-to-Right

types of operations and overloading is not uncommon. Unfortunately, having so many methods available does make it difficult to learn a programming language but having it too simple, also affects how readable it is. It also allows for greater control using less steps or instructions, which make it very convenient for programmers.

2.5.4 Statements and Control Structures

Assignment Operations C++ assignment is fairly straightforward. The syntax is always the same in all situations.

variable_name = some_accepted_value;

The equal sign (=) being the assignment operator. As seen above, has a very low precedence as well as being read from right-to-left, so will almost always be done last. Assignment is also a statement and so must always have a semicolon to denote the end of the statement. It is also worthy of note that multiple assignment is allowed in C++ as well as multiple assignments in one line. The code below exemplifies that:

```
#include <iostream>
using namespace std;
int main(){
  int x = 52, y = 31, z = 40;
  int t1, t2;
  t1 = t2 = 52;
  cout<< t1<<"␣"<<t2<<endl;
  t1 = y, t2 = z;
  cout<< t1<<"␣"<<t2<<endl;

  return 0; //safely ending main()
}
--------------------------------
Expected Output:
52 52
31 40
```

The above use of assignments is a convenience when writing code as it allows for short but concise instructions but because of the use of arbitrary symbols, the readability of the code suffers especially if the programmer is fond of daisy-chaning many assignments at once. The facet that makes the readability process a lot less difficult is the use of comma because it will always denote two separate statements that are similar in concept or process. In this case, it is used for assignment.

Conditional Statements Conditional statements in C++ involve the use of the reserved words *if, else if, else,* and *switch.* the if-statement asks

a question that is determined True or False by logical conditions. When determined True, the code block under the it is run. When False, proceeds to check for other else if-statements or the else-statement or terminate. The else if-statement is an alternative question that is determined True or False by logical conditions, designed to be another logical check that is different or an alternative to the original if-statement and will only run its code block when the condition is determined to be True. The else statement automatically runs its code block when the if statement and all other else if statements have determined to be False. It is not necessary for a control statement to have else if statements or the else statement. The logical statements can be seen in the precedence table 6 and 7. The switch statement works in tandem with the case-label. A switch works by taking an input and passing it to the code block with the appropriate case. The following code snippet is an application of C++ conditional statements:

```
#include <iostream>

using namespace std;

int main(){
    int a, b, c;
    a = b = 2; //giving a value of 2 to variables a and b
    if(a==2){ //an if statement with the condition of is the
       ↪ value found in a equal to 2

        cout<<"In␣if␣statement"<<endl; //the code snippet run
           ↪ if the if statement is true

    }else if(b!=9){ //an else if statement with the condition
       ↪ of is the value found in b not equal to 9

        cout<<"In␣else-if␣statement"<<endl; //the code snippet
           ↪ run if the else if statement is true

    }else{ //the else statement

        cout<<"In␣else␣statement"<<endl; //runs if all of the
           ↪ above is deteremined to be false
```

```
    }
    c = 3; //setting variable c to have a value of 3

    switch(c){ //switch statement with the input of c

        case 1: //runs if the switch statement input is 1

            cout<< "switch␣Case␣1"<<endl;
            break;
        case 2: //runs if the switch statement input is 2

            cout<< "switch␣Case␣2"<<endl;
            break;
        case 3: //runs if the switch statement input is 3

            cout<< "switch␣Case␣3"<<endl;
            break;
        default: //runs if the switch statement input is none
            ↪ of the above

            cout<< "switch␣default"<<endl;
            break;
    }
    return 0; //safely ending main()
}
-------------------------------
Expected Output:
In if statement
switch Case 3
```

The snippet above shows the use of all control statements available of use. Notice how even though the else-if statement is True, the code block under it does not run. This is because the if, else-if, and else statements only run the first code block that passes as True in its statement and ignore the rest. In this case, the if-statement is True, so the program chooses to run only the if-statement and ignores the rest. Similarly, if the if-statement was False, and the else-if statement was True, it would only run the code block under the else-if statement and ignore the rest and proceed. The switch statement

works by accepting an input and choosing among the cases available and running the one with the exact case value. In this code, the value passed to the switch was three (3). So the switch statement pushes the code to be run to everything under case 3. The break; statement found in each case is important as it avoids running snippets of code it isn't supposed to because cases are really only jump labels and not actual scopes for code runs. Hypothetically, if case 3 did not have the break; statement, the line *cout¡¡ "switch default"¡¡endl;* would run, and that would be a behavior a programmer should prevent.

A common problem when writing if-statements are the nested if-statements that cause an ambiguous behavior. Exemplified in the following snippet

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

The question becomes, which else statement is that pointing towards? Is it the else statement of the first if-statement or the nested if-statement? Although a very common problem, it can be simply solved through the use of scope denoted by the use of curly braces {}.

```
if(condition){
  if(second.condition)
    statement
}else
  statement
```

or

```
  if(condition)
    if(second.condition){
      statement
  }else
    statement
```

Both methods allow for the prevention of ambiguity as well as making things more readable but perform completely different processes. The first scope shown is for when the else-statement is for the first if-statement while the second scope shown is for when the else-statement is for the nested if-statement.

This ability to decrease ambiguity takes more effort on the part of the programmer to scope out certain behaviors. Ultimately, it is for the best to use scopes to limit ambiguity.

Iterative Statements In C++, iterative statements or commonly known as loops, are very powerful tools that allow the performance of a specific snippet of code multiple times. There are three primary loops in C++, *for-loop, while-loop, and the do-while loop.* A common use for a loop is to go through various data structures such as arrays, lists, vectors, among many others available to C++. For loops have the following syntax:

```
for(initial\_value; conditionl; increment/decrement){
    statement;
}
```

While loops syntax is the following:

```
while(condition){
  statement;
  increment/decrement;
}
```

Do-while loops syntax is the following:

```
do{
  statement;
  increment/decrement;
}while(condition)
```

The main difference between the three loops is their usage and what they are designed to do. For loops were designed to iterate through data structures and have it check a specific set of instructions. For-loops are usually used when the programmer knows what it should be doing, and when it should stop. Do-While loops are used when the inner statements of the loop must be performed before checking the condition in which to stop. Its postfix nature allows for at least one iteration of the process to be performed. While loops are prefix in nature and check the conditions first before proceeding with the code block. It is imperative that all loops must have a condition which actually terminates else the problem of infinite looping occurs. For reference, all the conditions must be determine True for the code blocks within them to be processed and when deteremined False or the use of the break keyword exits the loop in its entirety. Another keyword found in loops is the continue

keyword which at the point of reading, ignores every other statement after it and proceeds with the next iteration of the loop. Be very careful of the placement of the continue statement, as this may also cause an infinite loop when put before the increment/decrement of the loop (does not affect for-loops). It is possible to have control statements and nested iterations in all loops. This allows for a great degree of freedom for the programmer to use the loop in the ways that they require. Unfortunately, because the loops have a very general syntax, it is easy to misread certain loops especially when nested multiple times. The reliability of the loops also depends on how the programmer wants it to perform but in general, well-coded loops will always work as intended. It is truly the fault of the programmer if an infinite loop is created.

2.5.5 Procedures and Subprograms

Functions In C++, the distinction between procedure, subprograms and functions very hazy and are generally accepted to be similar in meaning. Many programmers say that the meaning between the three has become more of a semantic problem. To help understand the distinction, the following definitions will be used for the paper. A *procedure* is a group of instructions that are performed step-by step. A *subprogram* is a procedure that is made because it is a commonly used process that can be called and performed at any time so long as it is fully defined. A *function* is the manifestation of a subprogram that the user defines and may call and use, similarly to any other subprogram. It may or may not return a specific entity. With the distinction out of the way, C++ allows for the creation, definition, and use of user-defined functions. Generally, C++ follows the following syntax to instantiate a function:

```
data\_type\_to\_return optional_scope::function\_name(
    ↪ argument1,argument2,...,argumentn){
  statements;
  return data\_type\_to\_return;
}
```

The *data type to return* may be any primitive data type or even data types that the user defines such as structs, classes, typedefs and more. Functions are one of the many tools every programmer must be capable of creating since it allows for code reusability and shows the extent of the knowledge the pro-

grammer has in the language. Functions are very much the bread and butter of good code. The part that indicates *optional_scope* is important because it denotes where this function can be found, either in a struct, in a class, or other scopes. Generally, if a function is defined in the same file without it being a part of a struct or class, this and the scope resolution operator are ommitted. Following on is the *function_name*. The function name follows the same rules as a variable name as found in section 2.5.2. Following the variable name in parenthesis are the function arguments. These arguments are generally used in the performance of the function itself. They are the values, objects, or entities, that help with what the function is instructed to do. One can have many arguments, as many as necessary. Most of the time, you have to declare the data type of each argument and works with all primitive data types as well as user-defined types, along with objects.

Simple Call Returns These are procedures refer how a program proceeds when there there is function call in between statements. The following code will demonstrate what this entails.

```
#include <iostream>

using namespace std;

int retAdd(int a,int b){
  return a+b; // the function returns sum of the two
     ↪ arguments
}

int main(){
  int x, y, z; //initialization of variables
  x = 5, y = 7; // setting value of x to 5 an y to 7

  x = 42; //assigned 42 to x
  z = retAdd(x,y); //function first proceeds to retAdd
     ↪ function, performs its process, and its return value
     ↪  is then assigned to z
  cout<< z <<endl; //print the value found in z

  return 0; //safely ending main()
}
```

```
------------------------------
Expected Output:
49
```

The simple snippet of code above shows that whenever a function call is performed, the program knows where it "stopped" and whenever the function has finished its process, will return to where it was called and follows through with the code or instruction in the area. In this case, it is assigning the return value to z. This is the concept of simple call returns, and it is an intuitive process that allows for programmers to know the behaviors of all the functions they code, as well process that is being undergone in each procedure or in each step of the code. This concept also works in nested functions, and is imperative in the concept of Recursive procedures.

Recursive Procedures C++ functions allow the process of recursive calls and processes. A recursive call is a procedure wherein a process is called into itself to create a smaller and more manageable version of itself that will eventually solve the problem or have performed the task it is required to do. An example of recursion in C++ is in the following code:

```
#include <iostream>

using namespace std;

int powersOfTwo(int a){ //the recursive function
  if(a==0) //the stop condition
    return 1;
  else
    return 2 * powersOfTwo(a-1); //the recursive call
}

int main(){
  int b=4;
  b = powersOfTwo(b); //the call to a recursive function
  cout<<b<<endl; //outputting the value

  return 0; //safely ending main()
}
------------------------------
Expected Output:
```

```
16
```

The recursive function above uses the concept of the simple call return to allow for a recursive call of the function powersOfTwo(). To make sure the recursion is not infinite, one must be able to know when the recursion must end or know the value to be returned at that specific instance is also known as the *base case*. In this case, the recursion must end when the argument passed is currently 1. The recursive call of *return 2 * powersOfTwo(a-1)* continuously multiplies 2 to whatever the result of the recursive call is and performs this over and over until it reaches the base case, in which it returns 1. Because the recursive call multiplies 2 for every function called, the amount of times the function is called will be a-1 and the result of the function would be 2^a (where a is the input of the function). The best way to understand this process is through following the calls, like this:

```
call to powersOfTwo(4) //initial call
  return 2 * powersOfTwo(3) //the argument inputted is not 1
     ↪ therefore run the else
    return 2 * (2 * powersOfTwo(2)) //this is a visualization
       ↪  of all the processes performed
      return 2 * (2 * (2 * powersOfTwo(1))) // runs the else
         ↪ again
        return 2 * (2 * (2 * (2 * powersOfTwo(0)))) //runs the
           ↪  else again
        return 2 * (2 * (2 * (2 * 1))) //reached the base case
           ↪  so the function returns 1
      return 2 * (2 * (2 * 2)) //returning the recursive
         ↪ calls
    return 2 * (2 * 4) //returning the recursive calls
  return 2 * 8 //returning the recursive calls
return 16; //returning the final answer
```

Exception Handling C++ has implemented methods for exception handling. Exceptions are problems that the code has during execution that may or may not cause unknown behaviors or are technically impossible to process such as the dividing by zero. Exception handling in C++ is done through the three keywords *try, catch, throw*. The *try* block is the block of code that has to run which may cause exceptions. The *catch* block is the part of the code that is run when the code in the try block does throw an exception. The *throw* block throws an exception when it encounters one

during runtime. An example of C++ exception handling below:

```
#include <iostream>
using namespace std;

double DivInt(int a, int b){
  double x;
  if(b==0){ //if the divisor is zero
    throw "Division␣by␣Zero␣is␣not␣allowed.";
  }else{
    x = (double) a / (double) b; //cast of two integers to
       ↪ doubles
    return x;
  }
}

int main(){
    int x = 5, y = 0;
    double z;
    try{
      z = DivInt(x,y); //division function call in the form of
         ↪  x / y
      cout<< z <<endl;
    }catch(const char* err){
        cout<< err<< endl; //the catch block process
    }

    return 0; //safely ending main()
}
-------------------------------
Expected Output:
Division by Zero is not allowed.
```

The code above shows how the try-catch block works in tandem with the throw code from within the DivInt() function. The try block is executed first which contains the DivInt() function which also contains the throw block. The throw runs only when the second argument is found to be zero. It throws the string "Division by Zero is not allowed". Whenever something is thrown, it also must be caught. So, the catch block waits to catch any

thrown exceptions and in this case, waits for a throw which is strictly a *const char** type. This catches our thrown string and proceeds to run the catch block. C++ also has a list of standard exceptions which one can use and implement into their code and is found on table 8.

Table 8: Exceptions in C++

Exception	**Short Description**
std::exception	An exception and parent clas of all standard C++ exceptions
std::bad_alloc	This can be thrown by the New method
std::bad_cast	This can be thrown by the dynamic_cast method
std::bad_exception	This is useful when handling unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by the typeid method
std::logic_error	An exception that can theoretically be detected by reading the code
std::domain_error	An exception thrown when a mathematically invalid domain is used
std::invalid_argument	An exception thrown due to invalid arguments
std::length_error	An exception thrown when an std::string that is too large is created
std::out_of_range	This is can be thrown by the at method, used generally with an std::vector
std::runtime_error	An exception that cannot theoretically be detected by reading the code
std::overflow_error	An exception thrown when a mathematical overflow occurs
std::range_error	An exception thrown when once tries to store a value which is out of range
std::underflow_error	An exception thrown when a mathematical underflow occurs

The use of all these exceptions allows a great degree of control for the programmer to issue many methods whenever cases of exceptions do happen within their code. It is unfortunate, however, that the code blocks require a great deal of writing especially if you need multiple catches for various exceptions coupled wwith syntaxes that are not very easily readable due to the confusion between the try-catch blocks and where the throws are going to be. However, because exception handling in C++ is well documented and has been developed over the course of many iterations throughout its development and standardizations, it is assured that all exception handling is reliable, works very well for all types of implementations and applications and is a very useful tool for programmers.

Parameters for Subprograms In C++, user-defined functions (subprograms) are able to accept various arguments (parameters). Throughout this paper, various examples have been made that include the use of user-defined functions. In order to understand how these functions interact with the parameters, we must understand how they are passed. C++ allows for pass-by-value parameter passing, and pass-by-reference parameter passing. Pass-by-value parameter passing means that the value-holder (the memory dedicated to holding the value of the variable) passed onto the function cannot be changed/accessed within the function itself and does not affect the actual value found in the caller environment. Pass-by-reference parameter passing passing is the opposite, wherein the value-holder is able to be changed within the function and does reflect that change in the original calling environment. The following code shows the difference between the two:

```
#include <iostream>

using namespace std;

void passVal(int x, int y){ //sample pass-by-value function
  x = 5;
  y = 20;
}

void passRef(int *x, int*y){ //sample pass-by-reference
   ↪ function
  *x = 35;
  *y = 69;
```

```
}

int main(){
  int a = 70, b = 54;

  passVal(a,b); //calling pass-by-value function

  cout<< a << "␣" <<b<<endl; //printing out current values of
     ↪ a and b

  passRef(&a,&b); // calling pass-by-reference function

  cout<< a << "␣" <<b<<endl; //printing out current values of
     ↪ a and b

  return 0; //safely ending main()
}
-------------------------------
Expected Output:
70 54
35 69
```

One may notice that even though the values were passed into the passVal() function their values did not change, but when passed onto passRef() their original values did change. The writability of such statements is fairly simple and the compiler is also able to read the errors done in writing, but because only simple and overloaded operators are used, reading becomes difficult especially for a code that is long, complex, and involves many header files. Another disadvantage of having to know the difference is the ability of pass-by-reference to be able to change the original values that it is pointing to, and may even cause problems when involving arrays. An advantage of knowing between the two is that it allows for a great amount of freedom especially when one needs to be able to cause sideffects such as using the same value in both the calling environment and the function's environment (such as a forced increment to ignore certain array accesses). Parameters are not only limited to passing variables, it is also able to pass entities that are results of other function processes. An example of this in the code below.

```
#include <iostream>
```

```
using namespace std;

int addin(int x, int y){ //function that returns the sum of
    ↪ the two arguments
  return x+y;
}
int multin(int x, int y){//function that returns the product
    ↪ of the two arguments
  return x*y;
}

int main(){
  int a = 70, b = 54, c = 32, d;
  d = addin(multin(b,c),a); //calling multin() as an argument
     ↪ for addin()
  cout<< d <<endl;
  d = multin(addin(a,b),c) //calling addin() as an argument
     ↪ for multin()
  cout<< d <<endl;

  return 0; //safely ending main()
}
--------------------------------
Expected Output:
1768
3968
```

As one can see, it is very much possible to use functions as parameters, and according to how functions are processed, they must first be able to evaluate the arguments passed onto them as well their proper data types. This means that functions in the arguments are done first, before the execution of the function proper. The outputs above reflect that behavior. This allows the programmer for freedom in the code that they write. This is especially useful for areas of code that need the result of a function but are not necessary to save or assign to a variable, or are just temporary. Easily written, and somewhat easy to read as functions in arguments can easily be denoted by the existence of the parentheses. The only difficulty that may be encountered

is the return type of the function found inside the argument (when reading). The process is also reliable since the processing of the inner function is always performed first and may go through type coercion even if the function called may not return the correct data type (follows type coercion rules in C++).

Overloaded Subprograms C++ has the innate ability to created overloaded subprograms. This means that the core of software reusability is applied in the language. The rules and restrictions of writing overloaded functions are as follows:

- Any two functions in the set of overloaded functions must have different argument lists
- Overloading functions with argument lists of the same type, based on return type alone, is an error
- Member functions cannot be overloaded solely on one being static and the other non-static
- Typedef declarations must be different and not synonymous to other overloaded functions
- Enumerated types are distinct and therefore distinguisable between other overloaded functions
- The type "array of" and "pointer of" are considered identical but only for single-dimension arrays; other dimensions are considered distinguishable from others

All the rules stated above are strictly followed and the program will not compile if done any other way. A very simple example of overloaded functions is as follows:

```
  #include <iostream>

using namespace std;

int addnums(int x, int y){ //function that returns the sum of
    ↪ two ints
  return x+y;
}
```

```
double addnums(double x, double y){//function that returns the
   ↪  sum of two doubles
  return x+y;
}

int main(){
  int a = 70, b = 54, c;
  double d = 1.2214, e = 4.4532, f;

  c = addnums(a,b); //call for addnums for ints
  cout<< c <<endl; //displays the value
  f = addnums(d,e); //call for addnums for doubles
  cout<< f <<endl; //displays the value

  return 0; //safely ending main()
}
-------------------------------
Expected Output:
124
5.6746
```

As seen in the example above, it is possible to create an overloaded function that does return another data type so long as the argument list differs in either amount or in the data types required. Overall, function overloading is both difficult to write, and to read, and most programmers tend to avoid hard coding overloaded functions (other than for use in constructors and destructors). The problem may stem from the similar names of functions as well as the possible returns. Another problem in writing is that one must know what the end result to return as well as managing to implement all possible types and combinations that could occur with an overloaded function. In the example above, it is not possible to perform the instruction *f = addnums(a,d)*. One would think the coercion would allow for such a step, but because overloaded functions are strict with arguments evaluation and data types, it does its best to avoid ambiguity and avoids processing these situations. Reading it is even harder, especially when the same function call is used for a variety of processes and the differences of what they may return as a result of their processes. In terms of reliability, so long as the programmer has coded the functions properly and in the behavior that they require,

there should not be any problems with creating overloaded functions.

Generic Subprograms C++ does allow for the use of generic functions (subprograms) seeing as how this is an important tool in object-oriented programming. The way it is mostly implemented in C++ is through the use of *templates*. Generic functions are important because they help us avoid using overloaded functions (its problems discussed earlier), allows for effecient code reusability, and allows for it to be used in a variety of cases (thus being called "generic"). A reflection of the code for overloaded functions but instead implemented through generic functions is shown below:

```
#include <iostream>

using namespace std;

template <typename T> //the template call

T addnums(T x, T y){ //a function using the template that
    ↪ returns
  return x + y;
}

int main(){
  int a = 70, b = 54, c;
  double d = 1.2214, e = 4.4532, f;

  c = addnums<int>(a,b); //call for addnums for ints
  cout<< c <<endl; //displays the value
  f = addnums<double>(d,e); //call for addnums for doubles
  cout<< f <<endl; //displays the value
  f = addnums<double>(a,d); //call for addnums for doubles but
     ↪  with an int and a double
  cout<< f <<endl; //displays the value
  c = addnums<int>(b,e); //call for addnums for ints but with
     ↪ an int and a double
  cout<< c <<endl; //displays the value

  return 0; //safely ending main()
}
```

```
------------------------------
Expected Output:
124
5.6746
71.2214
58
```

In comparison to overloaded functions, generic functions have indeed allowed the addition between doubles and ints through the use of the template as well as type coercion. The same function was able to perform the same processes but with less code. Writing the code is far easier than that of overloaded functions and is generally more reliable seeing as how it accepts multiple data types and performs implicit type conversions. There is a slight problem that this causes, it makes it harder to read and discern the behavior especially for large scale code. It is slight because everytime the function is called, it requires the type to be declared at its instance. Notice how each function call has the data types in angled brackets that denote the data type that the template will use. In terms of reliability, templates are able to be used in the implementation of various functions that can be found in structs, classes, and even data structure manipulation methods and are a part of standard practices in the current industry. This means that it is highly reliable and is at the top of skills a successful C++ programmer must have.

User-defined Overloaded Operators C++ also allows for user-defined overloaded operators. This is because C++ has the ability to have user-defined data types as well as user-defined objects such as structs and classes. A simple implementation can be found below.

```
  #include <iostream>

using namespace std;

class point{ //class called point
  private:
    int x, y; //attributes of the class

  public:
  point operator+(const point& line){ //the user-defined
     ↪ overloaded operator
    point l1;
```

```
    l1.x = this->x + line.x;
    l1.y = this->y + line.y;
    return l1;
  }

  point(){ //class default constructor
    x = 0;
    y = 0;
  }

  point(int inx, int iny){ //class overloaded constructor
    x = inx;
    y = iny;
  }

  void print(){ //print
    cout<< this->x << "␣" << this->y << endl;
  }
};

int main(){
  point x(25,21), y(10,15), z; //creating objects
  z = x + y; //user-defined overloaded operator call
  z.print(); //prints out the values of x and y

  return 0; //safely ending main()
}
--------------------------------
Expected Output:
35 36
```

The code above shows the use of user-defined overloaded operators through the use of a class with integer attributes x and y with the class being called *point*. Its constructors are there to further simplify the intialization of the object. The overloaded operator is the + operator which has been defined to add the attributes x and y to another point object's x and y attributes. Without the overloaded definition, this process would have had to be done manually, taking up more memory, and needing more time to process and

code. The upside to having user-defined overloaded operators is that it makes future code much simpler to use and implement, as well as make the code shorter and more concise, and it decreases time spend on code production due to software reuse. The downsides include having to know what each operator does for which specific process. This affects both readability and writability, especially when dealing with multiple objects. In terms of reliability, similar to generic subprograms, so long as the programmer defines the operators well, they will work as intended, sideffects and all.

All in all C++ is a very powerful programming language with a vast array of features that has allowed it to be one of the best programming languages to use and learn. Both writability and readability are equally balanced for even a beginner. Due to it being a compiled language, it has allowed for a variety of good features such as fast execution times, error-catching at compilation, and having expected behaviors during runtime. Its focus on object-oriented programming has given it a plethora of features that allow the programmer to code anything and everything. It truly is an industry-level programming language, and it has such a great online presence that references, documentation, and library support are among the best. C++ is a language that anyone can learn.

3 Feature 1: Struct

3.1 Description of the feature and Code in C++

A struct is a blueprint used to create a lightweight instance of a class. It is comprised of data elements called members. Structs are important pieces of code that allow for user-defined objects that have specific attributes and processes that change these attributes to suite the purpose of programmer to solve specific problems in a well compacted way. They are usually built around the primitive data types found in C++. In C++ one can define a struct using the struct keyword followed by the name. The code found inside of the scope are its associated attributes and methods.

3.2 Code in C++

```
#include <iostream>
```

```
using namespace std;

// the struct feature
struct Rectangle{
  int length;
  int width;

  // struct constructor
  Rectangle(int l, int w){
    length = l;
    width = w;
  }

  int Rectangle_area(){
    return length * width;
  }
};

int main(){
  // defining an instance of a rectangle
  struct Rectangle newrec = Rectangle(10,20);
  cout << ``Length: '' << newrec.length << ``, Width: '' <<
     ↪ newrec.width << endl;
  cout << ``The area of the rectangle is '' << newrec.
     ↪ Rectangle_area() << endl;
}

----------------------------------------
Output:
Length: 10, Width: 20
The area of the rectangle is 200
```

3.3 Advantages of having Structs

Similar to classes, structs are very useful in object-oriented programming. It provides another way of grouping data elements under one name. It also provides for a means to implement plain-old-data (POD) types, which are

classes without constructors/destructors. A clear advantage of a struct is the ability to create an object that describes an entity in real life as well as the methods one can do with such an object. It also allows for definition and processing of data, separate from other processes. Another advantage is that structs, when defined, have a long lifetimes during runtime that allows for dynamic allocation of memory and greater control of data.

3.4 Advantages of not having Structs

As structs are very similar to classes except for their default element access and inheritance manner (public for structs, private for classes), one can say that choosing either of them is a matter of preference. Hence in most contexts, structs can be seen as a kind of duplicate to classes, hence not having them slightly improves readability and writability as one is assured that all object-oriented actions are done using classes only. Also, not having structs do not greatly affect a language with classes like R.

An advantage of not having structs is that there is no confusion as to how to process certain data. Data would also not be isolated in its own scope and may be accessed throughout the code wherever possible.

3.5 Workaround in R

We used R's feature of declaring new environments and declaring objects within those environments as a way of implementing a similar "Struct" feature in R. We created the code below to simulate what structs can do:

```
# used to execute functions defined within the "Struct"
execute <- function(func ,env=parent.frame()){
    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()
            iterval <- 1
            for(i in 1:length(env.data)){
                var <- env.data[[i]]
                if(length(var) == 1){
                    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]]
                }
            }
            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
    }
}
```

To emulate the same Rectangle struct in C++, the following can be done:

```
# creation of functions to be added into the Rectangle
    ↪ Struct
> Rectangle_area <- function() length*width

# ‘‘Creating’’ a struct, that is, providing the variable
    ↪ names and functions inside the struct
# variable names are characters while functions are vectors
    ↪ which contain the name and the function
> Rectangle <- Struct(‘‘length’’, ‘‘width’’, c(‘‘Rectangle_
    ↪ area’’,Rectangle_area))

# Creating an instance of rectangle using constructor()
# values are passed positionally as a vector
> newrec <- constructor(Rectangle, c(10,20))
> newrec$length
[1] 10
> newrec$width
[1] 20

# calling a struct function using execute(), which takes the
    ↪  struct and the function as arguments
> execute(newrec$Rectangle_area, newrec)()
[1] 200

# like C++ structs, one can also alter the values using a
    ↪ dot ($ in R)
> newrec$length <- 20
```

```
> execute(newrec$Rectangle_area, newrec)()
[1] 400
```

4 Feature 2: Overloaded functions

4.1 Description of the feature and Code in C++

In the context of C++, overloaded functions are used within classes in order to create functions that do similar things with different inputs. A common example are constructors for classes being that classes with multiple attributes may be initialized with or without these attributes. Another great use of overloaded functions is having functions with the same name, that perform the same process, but return varying data types. In the part on which discussed R's generic function, one can see that constructing methods are quite similar to constructing overloaded functions. However, one also see that such methods only apply in a class-to-function basis, not necessarily a function-to-function overloading.

```
#include <iostream>

int mul(int a, int b){
  return a * b;
}

float mul(float a, float b, float c){
  return a * b * c;
}

int main(){
  cout << ‘‘Int mul ’’ << mul(10,20);
  cout << ‘‘Float mul ’’ << mul(1.0, 2.0, 3.0);
}

-------------------------------------
Output:
Int mul 200
Float mul 6.0
```

4.2 Advantages of having Overloaded functions

One major advantage of function overloading is having the same name for functions which do the same things, thereby improving readability. Another great advantage is the saving of memory because the programmer need not create new functions for behaviors they have already defined.

4.3 Advantages of not having Overloaded functions

Function overloading introduces ambiguity on how the function is exactly implemented which may be due to language's type coercion rules on arguments, presence of implementations with default values, and function which uses pass by reference (although this is not the case in R). R prevents such issues without overloaded functions; furthemore, if the function to be overloaded is class-based, generic functions/methods can be made.

4.4 Workaround in R

The following can be a possible solution to emulate function overloading, given the following assumptions:

1. Similar to creating overloaded functions, the programmers knows the minimum and maximum number of arguments.

2. Argument names are decided beforehand for all possible arguments of the said functions

The approach uses a single function with all the possible parameters within it. We exploit the feature of having assigning FALSE to default argument values.

```
# method 1: using FALSE as default values
mul <- function(a=FALSE, b=FALSE, c=FALSE){
    if(!isFALSE(a)){
        if(is.integer(a)){
            if(!isFALSE(b)){
                # explicit conversion based on the return type
                    ↪ (int)
                a * as.integer(b)
            }
```

```
            else
                stop(``Variable b is not initialized b is not
                    ↪ initialized'')
        }
        else{
            # coercion from integer to numeric according to the
                ↪  return type (float)
            if(isFALSE(b))
                stop(``Variable b is not initialized'')
            else{
                if(!isFALSE(c))
                    a * b * c # returns numeric
                else
                    as.integer(a) * as.integer(b) # coerce to
                        ↪ int
            }
        }
    }
    else{
        # emulate errors if named variables are passed
        stop(``Variable a is not initialized'')
    }
}

> val1 <- mul(1L, 2L) # int return
> val1
[1] 2
> class(val1)
[1] ``integer''
> val2 <- mul(1.5, 4) # int return
> val2
[1] 4
> class(val2)
[1] ``integer''
> class(mul(1,2,3)) # float/numeric return
[1] ``numeric''
```

5 Feature 3: Do-while loops

5.1 Description of feature and Code in C++

A do-while loop repeats a statement at least once before evaluating the iteration statement. In contrast to a while loop, a do-while loop employs a posttest checking.

```
#include <iostream>

using namespace std;

int main(){
  int x = 0;

  do{
    x++;
    cout << x << endl;
  }while(x < 3 );

}

------------------------
Output:
1
2
3
```

5.2 Advantages of having Do-while loops

An advantage of having a do-while loop is having another looping mechanism within the language.

5.3 Advantages of not having Do-while loops

As posttest-based loops are more prone to usage error, not having do-while loops prevents such errors from happening. Also, the said loop is fundamentally a reversed while loop, so not having a do-while would prevent an less reliable duplicate of the while loop.

5.4 Workaround in R

We used R's repeat loop and a conditional statement at the end of the loop body. This ensures that the code runs at least once before checking the conditions.

```
> x <- 0
> repeat{
+ x <- x + 1
+ print(x)
+
+ # acts as the while of the loop
+ if(x >= 3)
+ break
+ }
[1] 1
[1] 2
[1] 3
```

6 Conclusion

After the extensive critique of both languages, one can see the advantages and disadvantages of C++ and R as programming languages. This section will summarize and contextualize the critiques of the features of the languages.

R as a language assumes as a role between those who use programming as a tool in their field (non-programmers) and those who are programmers. Fundamentally similar to S, R caters to mathematicians and data scientists as a language and software where they do data analysis. Hence, some of the features may not be as intuitive for heavy programmers. Environments are very useful for non-programmers who almost exclusively use the interactive environment as they are able to easily store and access the data that they need anytime. Features such as implicit coercion and special data values are more applicable to such mathematical needs which mostly deals with dataframes. The functional features are also effective and intentional due to fact that exploratory data analysis do not necessary mutate data and that such work is done on the interactive environment provided by the software; they also emulate the mathematical concepts, which does not allow for mutation especially in functions. Furthermore, some functions like lapply, sapply,

and plotting functions are also optimized to run over large data. On the other hand, programmers are also able to enjoy the features of R as the language also allows for the creation of larger programs. The syntax of most discussed programming constructs and features can be found in other programming languages as well, thereby transitional from some commonly used language to R is not that difficult. Robust programs can be made as most of the commonly used programming constructs are present in the language. R also supports object-oriented features with the S3 and S4 objects which appeals to programmers who are used to the OO paradigm. Being open-source and having a large user base, it is easy to find packages to most features not found in the base R language.

In comparison to more general C++, R is a niche-oriented language. As mentioned, the language design is heavily tailored towards data analysis and graphical representation, which means that it is more limited than C++. The recurring critique of having multiple implementations of the same feature/-construct of R versus the more distinct implementations of C++ indicates that R is splitting their implementations for specific use cases, presumably to give non-programmers more options on how they do some things. the way R stores data in physical memory in contrast to how C++ stores variables in the RAM can be attributed to the presence of an interactive environment where users can create and interact with data without caring what the interpreter does in the background. C++ allows for direct memory access via pointers and functions such as `malloc()` whereas R does not give direct access to them means that programs. In terms of features revolving around type binding and function definitions, C++ requires for types to be declared explicitly as the language implemented using static binding whereas R is the opposite. C++ also has more robust object-oriented features than R. Furthemore, the former language is compiled while the latter is interpreted. Such program designs mean that C++ is faster than R, which directly reflects their use cases: C++ (as stated earlier) is used in various fields such as in applications, compilers, operating systems, and animation thus speed and reliability of the language is prioritized, R is mainly used today in statistical modelling, implementing machine learning models, and other data-oriented uses hence operability and accessibility to a non-programming user base is prioritized. It is also worth noting that R is written in C and has interfaces for other languages such as C, C++, and Fortran for more computationally-intensive work, thus the two languages are not that distant from each other.

Ultimately, there is no "better language" between C++ and R. For math-

ematically intensive work which deals with large data, R is the language to go. For a more general purpose language which makes heavy use of object-oriented features, C++ is a great language. The choice of using one language against the other boils down to use case, needed features, and one's personal preference.

References

[1] Peng, R. D. (2016). R Programming for data science. Victoria, British Columbia, Canada: Leanpub.

[2] Matloff, N. (2013). The art of R programming: A tour of statistical software design. San Francisco: No Starch Press.

[3] R Programming. (2016). Tutorials Point.

[4] Blair, E. (2004, February 28). A critique of R and S-PLUS [PDF].

[5] Adler, J. (2012). R in a Nutshell. Beijing: O'Reilly.

[6] Chambers, J. M. (2014). Object-Oriented Programming, Functional Programming and R. Statistical Science, 29(2), 167–180. https://doi.org/10.1214/13-sts452

[7] The R language, for programmers. (2017, July 31). Retrieved December 2, 2020, from https://www.johndcook.com/blog/r_language_for_programmers/

[8] R Recursion (Recursive Function) With Example. (2018, October 08). Retrieved December 27, 2020, from https://www.datamentor.io/r-programming/recursion/

[9] (n.d.). Retrieved December 28, 2020, from https://cran.r-project.org/doc/manuals/R-lang.html

[10] C++ Structs - javatpoint. (n.d.). Retrieved December 29, 2020, from https://www.javatpoint.com/cpp-structs

[11] Wickham, H. (n.d.). Advanced R. Retrieved December 31, 2020, from https://adv-r.hadley.nz/fp.html

[12] C++ Overloading - javatpoint. (n.d.). Retrieved December 29, 2020, from https://www.javatpoint.com/cpp-overloading

[13] Uses of C++: Top 10 Reasons Why You Should Use C++. (2020, August 10). Retrieved December 31, 2020, from https://www.educba.com/uses-of-c-plus-plus/

[14] C++ Resources Network (n.d.) History of C++ Retrieved December 07, 2020, from https://www.cplusplus.com/info/history/

[15] CPPReference (n.d.) History of C++ Retrieved December 07, 2020, from https://en.cppreference.com/w/cpp/language/history

[16] C++ Resources Network (n.d.) Description of C++ Retrieved December 07, 2020, from https://www.cplusplus.com/info/description/

[17] CPPReference (n.d.) C++11 Retrieved December 07, 2020, from https://en.cppreference.com/w/cpp/11

[18] CPPReference (n.d.) C++14 Retrieved December 07, 2020, from https://en.cppreference.com/w/cpp/14

[19] CPPReference (n.d.) C++17 Retrieved December 07, 2020, from https://en.cppreference.com/w/cpp/17

[20] CPPReference (n.d.) C++20 Retrieved December 07, 2020, from https://en.cppreference.com/w/cpp/20

[21] Agarwal, H. (2020, October 14) C++ Data Types Retrieved December 29, 2020, from https://www.geeksforgeeks.org/c-data-types/

[22] Robertson, C. (2019, October 21) Retrieved December 29, 2020, from https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-160

[23] Fundamental types (n.d.) Retrieved December 29, 2020, from https://en.cppreference.com/w/cpp/language/types

[24] Christine, S. (n.d.) What are the rules to declare variables in C++? Retrieved Dec 29, 2020, from https://www.tutorialspoint.com/What-are-the-rules-to-declare-variables-in-Cplusplus

[25] C++ Variables (n.d.) Retrieved Dec 29, 2020, from https://www.w3schools.in/cplusplus-tutorial/variables/

[26] Robertson, C.(2018, November 19) Scope (C++) Retrieved December 29, 2020, from https://docs.microsoft.com/en-us/cpp/cpp/scope-visual-cpp?view=msvc-160

[27] Robertson, C. (2019, November 19) Pointers (C++) Retrieved December 29, 2020, from https://docs.microsoft.com/en-us/cpp/cpp/pointers-cpp?view=msvc-160

[28] Robertson, C. (2016, November 04) References (C++) Retrieved December 29, 2020, from https://docs.microsoft.com/en-us/cpp/cpp/references-cpp?view=msvc-160

[29] Pillai, P. (2004, June 18) Introduction to Static and Dynamic Typing Retrieved December 29, 2020, from https://www.sitepoint.com/typing-versus-dynamic-typing/

[30] Singh, H. (2018, February 05) Early binding and Late binding in C++ Retrieved December 29, 2020, from https://www.geeksforgeeks.org/early-binding-late-binding-c/

[31] Alex (2007, June 19) 6.15 — Implicit type conversion (coercion) Retrieved on December 29, 2020, from https://www.learncpp.com/cpp-tutorial/implicit-type-conversion-coercion/

[32] Alex (2015, April 16) 6.16 — Explicit type conversion (casting) and static_cast Retrieved on December 29, 2020, from https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/

[33] Robertson, C. (2019, November 11) Type conversions and type safety Retrieved on December 29, 2020, from https://docs.microsoft.com/en-us/cpp/cpp/type-conversions-and-type-safety-modern-cpp?view=msvc-160

[34] Grimm, R. (2018, February 12) C++ Core Guidelines: Rules for Conversions and Casts Retrieved on December 29, 2020, from https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-conversions-and-casts

[35] CPPReference (n.d.) C++ Operator Precedence Retrieved on December 29, 2020, from https://en.cppreference.com/w/cpp/language/operator_precedence

[36] CPPReference (n.d.) Other operators Retrieved on December 29, 2020, from

https://en.cppreference.com/w/cpp/language/operator_other#Built-in_comma_operator

[37] W3schools (n.d.) C++ If ... Else Retrieved on December 29, 2020, from https://www.w3schools.com/cpp/cpp_conditions.asp

[38] C++ Resources Network (n.d.) Functions Retrieved on December 29, 2020, from https://www.cplusplus.com/doc/tutorial/functions/

[39] tutorialspoint (n.d.) C++ Exception Handling Retrieved on December 31, 2020, from https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

[40] GeeksforGeeks (2018, December 18) Parameter Passing Techniques in C/C++ Retrieved on December 31, 2020, from https://www.geeksforgeeks.org/parameter-passing-techniques-in-c-cpp/

[41] Robertson, C. (2019, March 27) Function Overloading Retrieved on December 31, 2020, from https://docs.microsoft.com/en-us/cpp/cpp/function-overloading?view=msvc-160#restrictions-on-overloading

[42] Robertson, C. (2018, October 12) Generic Functions Retrieved on December 31, 2020, from https://docs.microsoft.com/en-us/cpp/extensions/generic-functions-cpp-cli?view=msvc-160

[43] Patil, R. (2019, April 01) Generics in C++ Retrieved on December 31, 2020, from https://www.geeksforgeeks.org/generics-in-c/

[44] tutorialspoint (n.d.) C++ Overloading (Operator and Function) Retrieved on December 31, 2020, from https://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

Forums on Stackoverflow:

[45] https://stackoverflow.com/questions/9734646/using-generic-functions-of-r-when-and-why

[46] https://stackoverflow.com/questions/9266194/r-function-overloading

[47] https://stackoverflow.com/questions/1944910/what-is-your-preferred-style-for-naming-variables-in-r

[48] https://stackoverflow.com/questions/6048344/what-is-the-difference-between-a-function-and-a-subroutine