Comparison of J and C Programming Languages

What are J and C Programming Languages

Both C and J are programming languages, but they are as different as English and Chinese are.

C is an imperative programming language that was first created in 1972 by Dennis Ritche and Ken Thompson. Prior to C Thompson created B programming language for the development of Unix operating system but it was too slow. Then Ritche started improving B in 1971 and called the improved version New B, NB was used for developing Unix kernel and the needs of doing that shaped the direction of the language development. Through one year of development a new compiler was written, and the language was renamed to C.

J is an array-oriented programming language that was first developed in early 1990s by Kenneth E. Iverson and Roger Hui which is based on primarily on APL, which was also developed by Iverson and originally was developed as a mathematical notation system. J's main domain is mathematical processes that have multidimensional arrays as central datatypes. Instead of being compiled J programs are interpreted. Compared to APL J only uses ASCII characters to not have special character problem.

What are imperative and array-oriented programming languages

Imperative programming is a programming paradigm that uses statements to change a program's state. An imperative program consists of commands for computer to perform step by step.

Array-oriented programming is a programming paradigm that operates on an entire set of values at once. An array-oriented program processes differently dimensioned arrays transparently and with no change in the structure of the process.

Comparison criteria

Simplicity

Simplicity of a programing language depends on number of unique features and redundancy of features. Compared to C J has more basic constructs due its expressiveness but it doesn't have feature multiplicity as much as C does, C and J example side by side:

```
// increase by one
counter = counter + 1;
counter += 1;
counter++;
++counter;
```

```
NB. both return _2 _3 _4
-@>:1 2 3
-@:>:1 2 3
```

The pre and post increment in C are not the same but neither "@" and "@:" are same in J even though their effect in given examples are same, respectively. Though number of verbs in J that result same are less than the number of equal operations that result the same in C, this is because verbs are already short in J and no further abbreviation is needed for ease.

This uniqueness of features of J makes it "more readable" in this aspect when compared to C. But due to a greater number of unique features it takes more time to learn all of them.

Orthogonality

Orthogonality is the program's features' independency of context of their appearance. There are number of context dependent changes in features in C, for example:

- Using keyword "static" in global namespace to declare a variable makes that variable unreachable from other C files of the same program but using the same keyword inside a function would result in allocating an independent memory to store variable's value and use same address for every call of that function.

- Using binary addition operator plus on a pointer and an integer give results depending on the type of the pointer. The addition of integer is basically multiplied by the size of the type of the pointer.

Although C is simpler and thus more orthogonal than its counterpart C++ and other popular languages in this aspect, J is inherently more orthogonal since its design principle is to process data independently of its size and shape. Even then there are some differences in behaviors of verbs (operators/functions) of J depending on if they are called monadically (with one operand) or dyadically (with two operands), for example:

- Using asterisk with two integers would give the result of their multiplication but if it was to be used with only one integer (as right operand) it would return the sign of the given integer.

This is the case for almost every verb in J language, although monadic and dyadic operations of verbs are similar or related this doesn't change the fact that they are not the same. But this is a design choice of J and simply a feature of it, nothing like the special cases of C mentioned above. A fairer comparison would be a verb behaving differently even though it's used monadically or dyadically in both different contexts, for example:

- Using left curly bracket with an integer **x** as left operand and a list of integers **y** as a right operand would return the item of **y** at the index **x** but if left operand was not an integer but a list of integers then the result would be list of integers of **y** at given indexes that are items of left operand.

This example can be said to show the context dependent behavior of the left curly bracket verb. But like dyadic and monadic design difference of verbs this also is a design choice and has to do with rank of the verb. What the left curly bracket does is looking scalar items of left operand and getting the corresponding item of right operand for each, it neither cares the dimension of left operand nor the right operand, it always behaves on **0 rank** (scalar/singular) items for left operand and the entirety of right operand. Again, every J verb has specific ranks and always behave depending on those but not the size or

shape of its operands. So is there no similar case of context dependent behavior in J? Well, there are few, for example:

- The code snippet “p. 0 _24 44 _24 4” (_x is basically -x) will return “4” and list of four integers “3 2 1 0” in the form of “4;3 2 1 0” (; is a verb). What happens here is verb **p.** is “polynomial” verb and it returns the roots of polynomial “0+(-24)x+44x^2+(-24)x^3+4x^4”. And the exact opposite happens when you run “p. 4;3 2 1 0” which returns “0 _24 44 _24 4”

As you can see here the verb **p.** behaves differently when given input has not different size or shape but different structure, verb “;” won’t be explained here but it simply results in a different form/structure with values “4” and “3 2 1 0” which is not a **two-dimensional** array in which case **p.** would behave the as it did for input “0 _24 44 _24 4” which is **one dimensional** numeric array.

There are other verbs that behave like this, but their number is low. As it is shown J verbs don’t behave like they have special cases that were thought afterwards, instead different context dependent behaviors are related to each other and more like two faces of the same coin. Therefore J is “more readable” and more writable compared to C in this aspect.

Control

Unlike C J doesn’t necessarily use loops, it supports it with keywords such as “for.”, “while.” etc. but this goes against the design philosophy of J and you can execute equivalent processes with verbs already, for example:

- To get minimum integer of a list of integers you simply do “>./1 7 2 9” which would return 9. There’s no loop here, the verb **>.** returns the greater one of its left and right operands and the adverb **/** executes the verb on its left as if it was in between of every item of the right operand

which would be “1 >. 7 >. 2 >. 9” and since J executes its verbs always from right to left without precedence difference this is equal to “(1 >. (7 >. (2 >. 9)))”

To get the maximum integer from an array of integers in C you would create a variable for max value and initialize it to some big number, write a for loop to visit all elements, compare them with the variable that holds the max value and if currently visited element is greater then assign the compared variable to current element and finally return the max variable. Being pointed out that J doesn’t necessarily have loops a similar process can be simulated by an adverb that accepts a value that determines the amount of execution of its left verb on its operands, to give an example without getting into details “1 +^:2 (5)” is equal to “1 + 1 + 5”. J also has a combination of verbs that can be similar to switch control, for example:

- “x +`-@.(2&|) y” will return the result of “x+y” or “x-y” if “2|y” (y mod 2) returns either 0 or 1, respectively. To explain what it does without getting into details, the ` verb puts given verbs into a list and the verb **@.** runs the verb at given index of that list with operands **x** and **y**.

Again, J also supports conditional controls with keywords such as “if.”, “else.”, “elseif.” Which C counterparts are “if”, “else”, and “else if”.

Data types and structures

C has four distinct basic types that are **char**, **int**, **float**, and **double** with the option of adding **short** or **long** before them to modify the memory size of the value. And it has five derived types that are **array**, **pointer**, **function**, **structure**, and **union**. Arrays are structures that contain several values of same type, pointer is a type that points the memory address of a data, functions are processes that have inputs and an output, structures are user defined types that can contain more than one data that are different types, and finally unions are types that can only contain a singular value, but it can be number of different types. And there’s **void** which represents “no value”.

J has only three main types that are **numeric**, **character**, and **boxed**. But numeric type encapsulates Boolean values (0 and 1), integer values, rational values (represented as XrY e.g 4r5 which is 4/5), float values, and complex values (represented as XjY e.g 3j4 which is 3+4i). You can have struct and union equivalents in J but they will be discussed later.

In this aspect it can be said that C has clearer types for handling data than J, which are structs.

Syntax considerations

Identifier form

J has all the restrictions that C has for identifiers of variables and alongside those J also indicates that you can't put and underscore (_) at the end of the identifier and they also can't have double underscore (__) in them due to **locale** (namespace) identification design. So, J is more restricted in this case.

Special words

Contrary to C J does not have "words" whatsoever (except "if.", "for." etc.), instead every verb consists of few characters. So, for someone with natural Latin based language background this is going to be less readable.

Form and meaning

C has English words for its control keywords and this helps programmers that are familiar with English to understand the semantics from the syntax but since J only has verbs made up of few characters no natural human language background will help, other than well known algebraic symbols such as plus and minus.

Abstraction

Process abstraction

Bot C and J lets the programmer to define functions (verbs in J's case) to use many times in different places without knowing what it does on the inside.

Data abstraction

As mentioned above C has arrays, structs, and unions to help data abstraction. One example would be a point struct that has an x and a y value to represent its coordinates.

But for data abstraction J only has arrays and boxes, arrays can only contain items of the same type, but a box is a type itself and a box can contain a value of any type (like a union in C). So, one can have a list of boxes that have different types of values in them. Though J still doesn't have a struct type so it's up to the programmer to create these lists of boxes having boxes in the correct order etc.

Expressivity

Expressivity is being able to express the design and thought process of the program easily and compactly. C is a simple and clear language, and you can do a lot by using what it offers to you but since it was developed with the intent of using if to developing an operating system it only has special words for machine data and state handling, mainly loops, controls, and pointers. J on the other hand embodies a mathematical structure and thinks of everything as arrays so it takes only three characters to get the mean value of a list of numbers in J whereas in C it would take several lines of code.

Type checking

In C whenever a variable is declared the programmer must specify its type, so the compiler knows if any type mismatch happens and if there is aborts compiling and reports the issue to the programmer. J is an interpreted language, and it does not have type specifying but unlike C it lets you to check data type in runtime and if an unchecked mismatch happens the program can continue for J is interpreted.

Error handling

C does not support error handling, to prevent errors you must use condition control statements. But J has "::" which catches errors, for example "+/::'msg' 'string'" will return "msg" since string characters cant be summed in J, instead of "msg" a verb could have been put which would be executed with the same operand "string". This can be chained so there isn't a limit. Also you can get the error number of the last error with "13!:11 ''" which won't be explained here.

Aliasing

C has pointers to point at the same memory cell(s) that can be addressed from different variables. J does not have this and pointers and always creates a new memory block for the result of a verb. Except when a file content is "mapped" to a variable which then another variable can be assigned to it and will behave as an alias where a change on any of them will result in change of file's content. And the other exception is when a variable has a boxed item which will be stored by its pointer. Again J does not have pointers as data type therefore programmers can't explicitly use pointers as variable values.

Cost

Performance

Since J is interpreted its performance is worse than C which is compiled.

Writing

Since J is a very expressive language writing a program is generally several times easier and faster compared to C.

Compiling

It has been a long time since the first C compiler was created, currently C compilers are very fast and therefore compiling cost is almost negligible. And J is interpreted so it has none.

Language

Both languages are available online free.

Reliability

Since C is a compiled language, it needs to be compiled in the environment it is desired to be run, whereas J being interpreted solves the problem.

Maintenance

C takes longer to write and is less expressive, but it is a simple and clear language, and it is closer to human language than J is. J is easier to write but it is clear it is nowhere near to C when it comes to readability for the untrained eye. So if its needed to read and maintain someone else's work it will be more costly in J.

Summary and comments

Even though that J and C are compared in this essay they were designed to be used for different purposes. In main aspects the differences are:

- Readability: C is more benefits from familiarity and naturalness of human language and Latin alphabet. Even though J is consistent within itself it is as different as math is to an average person, and if the programmer has bias such as being used thinking step by step as in imperative languages, then readability will be harder.
- Writability: C has less keywords that can constitute a foundation for anything but to be able to do so lacks expressiveness. J has numerous non overlapping expressive operators, but this results in search of the exact operator for the job and checking their description occasionally, throughout the learning process. But when you know what you're doing J will speed up writing process.
- Reliability: As for the machine and environment perspective J is more reliable since it is interpreted but the inherent difficulty of readability may reduce reliability of the programmer who writes in J.

Both having their advantages on one another, I would say that if you are a C programmer or any other imperative language programmer, even if you're not going to use it for serious developments you can still learn J for fun and to explore a new perspective of thinking and programming. The core idea and design of J is very mathematical and poetic whereas C's is efficient and multipurpose based.

References

Concepts of Programming Languages – Rober W. Sebesta

J for C Programmers – Henry Rich

https://www.arraycast.com/

https://code.jsoftware.com/wiki/Main_Page

https://en.wikipedia.org/wiki/J_(programming_language)

https://en.wikipedia.org/wiki/Array_programming

https://en.wikipedia.org/wiki/Imperative_programming

https://en.wikipedia.org/wiki/APL_(programming_language)

https://en.wikipedia.org/wiki/C_(programming_language)