Indexing of Arrays: 0 vs 1

Photo by James Wheeler from Pexels

1.1: Why Zero?

The majority of programming languages use 0-based indexing i.e. arrays in that language start at index 0. One major reason for this is convention. All the way back in 1966, Martin Richards - the creator of BPCL for IBM - used 0 as the initial index so that a pointer p would access the memory location at p[1]. Carrying this forward, C established this 0-based indexing even more due to its widespread use and portability. From then on, most programming languages, many of which are derivatives of C, started using 0-based indexing.

Later, the famous computer scientist Edsgar W. Dijkstra went on to write a note called "Why numbering should start at 0" enumerating and defending all the reasons for this "strange" convention[2].

According to Dijkstra, to denote the set of numbers 1, 2, 3 … 5, we could use the following 4 conventions:

A. $0 < i < 6$
B. $0 \leq i < 6$
C. $0 < i \leq 5$
D. $0 \leq i \leq 5$

Dijkstra laid out a rule for the *ideal convention*.

- The convention should be able to show an empty set

Also, Dijkstra ruled out A and C since to denote starting from 0, we would have to use -1 in the notation. Dijkstra called this "ugly".

We are now left with options B and D. From the rule that Dijkstra laid out, to denote an empty set, we would write it in the following way:

B. $0 \leq i < 0$

D. $0 \leq i \leq -1$

Again, Dijkstra called the option D “ugly” not only since it used a negative number, but also because it used a smaller number as an upper limit and a lower number as a lower limit. Naturally, we are left to prefer the option B.

Aside from this subjective “ugliness”, many recursive algorithms use 0 as a base case and would be harder to implement without 0-based indexing. In many combinatorial algorithms - such as getting all the subsets of a set - empty sets appear often and simple notation for this would be preferred.

Additionally, for cyclic lists, a modulo operator could be used to send the iterator back to 0 after the last element.

For a cyclic list with n elements and a variable k that iterates itself without stopping, the iterator i that denotes the index of the element being accessed could iterate in the following manner:

$$i = (k \% n)$$

Note that after accessing the (n-1)th element, we fall back to the ‘zeroth’ element.

Again, C and a number of other low-level languages use 0 so that the data can be accessed in a simple formulaic manner[3]:

If a is the memory address of the first (initial, not the one in position 1) element in an array, where s is the size of each element in the array, and i is the iterator, the memory address of an element in the ‘i’th position can be expressed by:

$$a + (i * s)$$

1.2: Why One?

Some languages such as Lua, Fortran and COBOL and Julia use 1-based indexing. The common argument for this convention-contrary indexing practice is "It is natural to count from one" and is correct to an extent. Since the dawn of math, humans learned to count from 1 since it is impossible to 'count' nothing.

In the case of Julia, which is used mainly in the field of research in science and math, it is more intuitive for scientists to start counting from 1 and is hence preferred.

Julia was also initially designed for a better programming language in the field of math and to iterate over MATLAB and use common programming features absent in similar languages.

However, it is important to note that it is possible to change the indexing in Julia[4].

Another reason is beginner-friendliness. When novice programmers lear indexing of a list, it is much more intuitive to think of 1:3 as 1, 2, 3 than the usual 0:3 which denotes the numbers 0, 1, 2.

Some languages like Pascal and Perl allow the programmer to define the indices of the array to their choosing[5]. This option can usually make solving problems much easier (such as the sum of 3 consecutive terms of an Arithmetic Progression be taken as (n - d) + (n) + (n + d) = 3n as opposed to (n) + (n + d) + (n + 2d) = (3n + 3d)

Another reason why high-level languages should use 1-based indexing since they do not allow the programmer to access memory locations individually and do not require 0-based indexing, which would instead convolute the code.

Also, the 'indexing optimisation' that 0-based indexing provides is not a true optimisation[6] since the accessing of memory addresses could be simplified by defining a term 'b' with:

$$b = a - s$$

This would then simplify the memory address formula to:

$$b + (i * s)$$

This does not require calculating (i - 1) every iteration as shown by this formula which is commonly used as an argument against 1-based indexing:

$$a + (s * (i - 1))$$

1.3: Conclusion: Which one is better?

Since this is written from a Julia standpoint, we will examine the advantages and disadvantages of each system with respect to Julia.

Since Julia's community is deeply rooted in pure math and science, 1-based indexing is preferred. Also, if Julia were to switch so that it follows the general programming norm, many scripts would cease to work and cause failure in critical research projects.

Also, the concept of indexing 1:3 as 1, 2, 3 is much more intuitive than the normative 0:3 as 0, 1, 2. Though Dijkstra has shown the inherent beauty of half open intervals, it is natural and intuitive to use 1 as a base count. We have also seen that 0-based indexing offers no real memory access optimisation as commonly argued.

The only drawback of 1-based indexing is cyclic lists and clunky setups required for these lists in 1-indexed languages.

In conclusion, it boils down to personal preference, application of language, and just indexing where not only problem solving, but debugging and writing code is easier.

In my opinion, languages should offer flexibility in indexing options to the programmer similar to the way Perl and Pascal do. However, this can make code hard to read if not properly written, or if the flexible indexing is not implemented properly.

For Julia, it is again important to note that the index can be changed.

Given Julia's community, usage, current standards, and beginner friendliness, 1-based indexing makes sense and is here to stay unless Julia is overhauled in a major update similar to that between Python 2 and 3.

1.4: Citations and References

[1]- Information about BPCL referenced from:
https://en.wikipedia.org/wiki/Zero-based_numbering#Origin

[2]- Dijkstra's Note can be found at:
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html

[3]- Numerical Properties of Memory Accessing referenced from:
https://en.wikipedia.org/wiki/Zero-based_numbering#Numerical_properties

[4]- More information on changing the index of Julia arrays:
https://docs.julialang.org/en/v1/devdocs/offset-arrays/

[5]- Custom nature of indices in Pascal:

https://en.wikipedia.org/wiki/Zero-based_numbering#Usage_in_programming_languages

[6]- Proof that 0-based indexing offers no optimisation:
https://en.wikipedia.org/wiki/Zero-based_numbering#Numerical_properties

Cover Image "Lake and Mountain" covered under the Free to Use license by "James Wheeler" on Pexels:
https://www.pexels.com/photo/alberta-amazing-attraction-banff-417074/