Array data type

From Wikipedia, the free encyclopedia

> *This article is about the abstract data type. For the byte-level structure, see Array data structure. For other uses, see Array.*

In computer science, an **array type** is a data type that is meant to describe a collection of *elements* (values or variables), each selected by one or more indices (identifying keys) that can be computed at run time by the program. Such a collection is usually called an **array variable**, **array value**, or simply **array**.[1] By analogy with the mathematical concepts of vector and matrix, array types with one and two indices are often called **vector type** and **matrix type**, respectively.

Language support for array types may include certain built-in array data types, some syntactic constructions (*array type constructors*) that the programmer may use to define such types and declare array variables, and special notation for indexing array elements.[1] For example, in the Pascal programming language, the declaration `type MyTable = array [1..4,1..2] of integer`, defines a new array data type called `MyTable`. The declaration `var A: MyTable` then defines a variable `A` of that type, which is an aggregate of eight elements, each being an integer variable identified by two indices. In the Pascal program, those elements are denoted `A[1,1]`, `A[1,2]`, `A[2,1]`,... `A[4,2]`.[2] Special array types are often defined by the language's standard libraries.

Arrays are distinguished from lists in that arrays allow random access, while lists only allow sequential access.[*citation needed*] Dynamic lists are also more common and easier to implement than dynamic arrays. Array types are distinguished from record types mainly because they allow the element indices to be computed at run time, as in the Pascal assignment `A[I,J] := A[N-I,2*J]`. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array variable.

In more theoretical contexts, especially in type theory and in the description of abstract algorithms, the terms "array" and "array type" sometimes refer to an abstract data type (ADT) also called *abstract array* or may refer to an *associative array*, a mathematical model with the basic operations and behavior of a typical array type in most languages — basically, a collection of elements that are selected by indices computed at run-time.

Depending on the language, array types may overlap (or be identified with) other data types that describe aggregates of values, such as lists and strings. Array types are often implemented by array data structures, but sometimes by other means, such as hash tables, linked lists, or search trees.

History [edit]

Wikipedia The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Esperanto
Tiếng Việt
Edit links

Assembly languages and low-level languages like BCPL[3] generally have no syntactic support for arrays.

Because of the importance of array structures for efficient computation, the earliest high-level programming languages, including FORTRAN (1957), COBOL (1960), and Algol 60 (1960), provided support for multi-dimensional arrays.

Abstract arrays [edit]

An array data structure can be mathematically modeled as an abstract data structure (an *abstract array*) with two operations

get(*A*, *I*): the data stored in the element of the array *A* whose indices are the integer tuple *I*.
set(*A*,*I*,*V*): the array that results by setting the value of that element to *V*.

These operations are required to satisfy the axioms[4]

get(*set*(*A*,*I*, *V*), *I*) = *V*
get(*set*(*A*,*I*, *V*), *J*) = *get*(*A*, *J*) if $I \neq J$

for any array state *A*, any value *V*, and any tuples *I*, *J* for which the operations are defined.

The first axiom means that each element behaves like a variable. The second axiom means that elements with distinct indices behave as disjoint variables, so that storing a value in one element does not affect the value of any other element.

These axioms do not place any constraints on the set of valid index tuples *I*, therefore this abstract model can be used for triangular matrices and other oddly-shaped arrays.

Implementations [edit]

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(May 2009)*

In order to effectively implement variables of such types as array structures (with indexing done by pointer arithmetic), many languages restrict the indices to integer data types (or other types that can be interpreted as integers, such as bytes and enumerated types), and require that all elements have the same data type and storage size. Most of those languages also restrict each index to a finite interval of integers, that remains fixed throughout the lifetime of the array variable. In some compiled languages, in fact, the index ranges may have to be known at compile time.

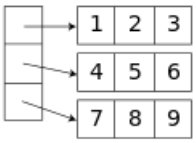
On the other hand, some programming languages provide more liberal array types, that allow indexing by arbitrary values, such as floating-point numbers, strings, objects, references, etc.. Such index values cannot be restricted to an interval, much less a fixed interval. So, these languages usually allow arbitrary new elements to be created at any time. This choice precludes the implementation of array types as array data structures. That is, those languages use array-like syntax to implement a more general associative array semantics, and must therefore be implemented by a hash table or some other search data structure.

Language support [edit]

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(May 2009)*

Multi-dimensional arrays [edit]

The number of indices needed to specify an element is called the *dimension*, *dimensionality*, or rank of the array type. (This nomenclature conflicts with the concept of dimension in linear algebra,[5] where it is the number of elements. Thus, an array of numbers with 5 rows and 4 columns, hence 20 elements, is said to have dimension 2 in computing contexts, but represents a matrix with dimension 4-by-5 or 20 in mathematics. Also, the computer science meaning of "rank" is similar to its meaning in tensor algebra but not to the linear algebra concept of rank of a matrix.)



Many languages support only one-dimensional arrays. In those languages, a multi-dimensional array is typically represented by an Iliffe vector, a one-dimensional array of references to arrays of one dimension less. A two-dimensional array, in particular, would be implemented as a vector of pointers to its rows. Thus an element in row *i* and column *j* of an

array *A* would be accessed by double indexing (*A*[*i*][*j*] in typical notation). This way of emulating multi-dimensional arrays allows the creation of jagged arrays, where each row may have a different size — or, in general, where the valid range of each index depends on the values of all preceding indices.

This representation for multi-dimensional arrays is quite prevalent in C and C++ software. However, C and C++ will use a linear indexing formula for multi-dimensional arrays that are declared as such, e.g. by `int A[10][20]` or `int A[m][n]`, instead of the traditional `int **A`.[6]:p.81

Indexing notation [edit]

Most programming languages that support arrays support the *store* and *select* operations, and have special syntax for indexing. Early languages used parentheses, e.g. `A(i,j)`, as in FORTRAN; others choose square brackets, e.g. `A[i,j]` or `A[i][j]`, as in Algol 60 and Pascal.

Index types [edit]

Array data types are most often implemented as array structures: with the indices restricted to integer (or totally ordered) values, index ranges fixed at array creation time, and multilinear element addressing. This was the case in most "third generation" languages, and is still the case of most systems programming languages such as Ada, C, and C++. In some languages, however, array data types have the semantics of associative arrays, with indices of arbitrary type and dynamic element creation. This is the case in some scripting languages such as Awk and Lua, and of some array types provided by standard C++ libraries.

Bounds checking [edit]

Some languages (like Pascal and Modula) perform bounds checking on every access, raising an exception or aborting the program when any index is out of its valid range. Compilers may allow these checks to be turned off to trade safety for speed. Other languages (like FORTRAN and C) trust the programmer and perform no checks. Good compilers may also analyze the program to determine the range of possible values that the index may have, and this analysis may lead to bounds-checking elimination.

Index origin [edit]

Some languages, such as C, provide only zero-based array types, for which the minimum valid value for any index is 0. This choice is convenient for array implementation and address computations. With a language such as C, a pointer to the interior of any array can be defined that will symbolically act as a pseudo-array that accommodates negative indices. This works only because C does not check an index against bounds when used.

Other languages provide only *one-based* array types, where each index starts at 1; this is the traditional convention in mathematics for matrices and mathematical sequences. A few languages, such as Pascal, support *n-based* array types, whose minimum legal indices are chosen by the programmer. The relative merits of each choice have been the subject of heated debate. Zero-based indexing has a natural advantage to one-based indexing in avoiding off-by-one or fencepost errors.[7]

See comparison of programming languages (array) for the base indices used by various languages.

Highest index [edit]

The relation between numbers appearing in an array declaration and the index of that array's last element also varies by language. In many languages (such as C), one should specify the number of elements contained in the array; whereas in others (such as Pascal and Visual Basic .NET) one should specify the numeric value of the index of the last element. Needless to say, this distinction is immaterial in languages where the indices start at 1.

Array algebra [edit]

Some programming languages support array programming, where operations and functions defined for certain data types are implicitly extended to arrays of elements of those types. Thus one can write *A*+*B* to add corresponding elements of two arrays *A* and *B*. Usually these languages provide both the element-by-element multiplication and the standard matrix product of linear algebra, and which of these is represented by the * operator varies by language.

Languages providing array programming capabilities have proliferated since the innovations in this area of APL. These are core capabilities of domain-specific languages such as GAUSS, IDL, Matlab, and Mathematica. They are a core facility in newer languages, such as Julia and recent versions of Fortran. These capabilities are also

provided via standard extension libraries for other general purpose programming languages (such as the widely used NumPy library for Python).

String types and arrays [edit]

Many languages provide a built-in string data type, with specialized notation ("string literals") to build values of that type. In some languages (such as C), a string is just an array of characters, or is handled in much the same way. Other languages, like Pascal, may provide vastly different operations for strings and arrays.

Array index range queries [edit]

Some programming languages provide operations that return the size (number of elements) of a vector, or, more generally, range of each index of an array. In C and C++ arrays do not support the *size* function, so programmers often have to declare separate variable to hold the size, and pass it to procedures as a separate parameter.

Elements of a newly created array may have undefined values (as in C), or may be defined to have a specific "default" value such as 0 or a null pointer (as in Java).

In C++ a std::vector object supports the *store*, *select*, and *append* operations with the performance characteristics discussed above. Vectors can be queried for their size and can be resized. Slower operations like inserting an element in the middle are also supported.

Slicing [edit]

An array slicing operation takes a subset of the elements of an array-typed entity (value or variable) and then assembles them as another array-typed entity, possibly with other indices. If array types are implemented as array structures, many useful slicing operations (such as selecting a sub-array, swapping indices, or reversing the direction of the indices) can be performed very efficiently by manipulating the dope vector of the structure. The possible slicings depend on the implementation details: for example, FORTRAN allows slicing off one column of a matrix variable, but not a row, and treat it as a vector; whereas C allow slicing off a row from a matrix, but not a column.

On the other hand, other slicing operations are possible when array types are implemented in other ways.

Resizing [edit]

Some languages allow *dynamic arrays* (also called *resizable*, *growable*, or *extensible*): array variables whose index ranges may be expanded at any time after creation, without changing the values of its current elements.

For one-dimensional arrays, this facility may be provided as an operation "`append`(*A*,*x*)" that increases the size of the array *A* by one and then sets the value of the last element to *x*. Other array types (such as Pascal strings) provide a concatenation operator, which can be used together with slicing to achieve that effect and more. In some languages, assigning a value to an element of an array automatically extends the array, if necessary, to include that element. In other array types, a slice can be replaced by an array of different size" with subsequent elements being renumbered accordingly — as in Python's list assignment "*A*[5:5] = [10,20,30]", that inserts three new elements (10,20, and 30) before element "*A*[5]". Resizable arrays are conceptually similar to lists, and the two concepts are synonymous in some languages.

An extensible array can be implemented as a fixed-size array, with a counter that records how many elements are actually in use. The `append` operation merely increments the counter; until the whole array is used, when the `append` operation may be defined to fail. This is an implementation of a dynamic array with a fixed capacity, as in the `string` type of Pascal. Alternatively, the `append` operation may re-allocate the underlying array with a larger size, and copy the old elements to the new area.

See also [edit]

- Array access analysis
- Array programming
- Array slicing
- Bounds checking and index checking
- Bounds checking elimination
- Delimiter-separated values
- Comparison of programming languages (array)
- Parallel array

Related types [edit]

- Variable-length array
- Dynamic array
- Sparse array

References [edit]

1. ^ [a] [b] Robert W. Sebesta (2001) Concepts of Programming Languages. Addison-Wesley. 4th edition (1998), 5th edition (2001), ISBN 9780201385960
2. ^ K. Jensen and Niklaus Wirth, PASCAL User Manual and Report. Springer. Paperback edition (2007) 184 pages, ISBN 978-3540069508
3. ^ John Mitchell, Concepts of Programming Languages. Cambridge University Press.
4. ^ Lukham, Suzuki (1979), "Verification of array, record, and pointer operations in Pascal". *ACM Transactions on Programming Languages and Systems* **1(2)**, 226–244.
5. ^ see the definition of a matrix
6. ^ Brian W. Kernighan and Dennis M. Ritchie (1988), *The C programming Language*. Prentice-Hall, 205 pages.
7. ^ Edsger W. Dijkstra, Why numbering should start at zero

External links [edit]

- NIST's Dictionary of Algorithms and Data Structures: Array

Wikibooks has a book on the topic of: ***Data Structures/Arrays***

Look up ***array*** in Wiktionary, the free dictionary.

Wikimedia Commons has media related to ***Array data structure***.

v · t · e	**Data types**	[hide]
Uninterpreted	Bit · Byte · Trit · Tryte · Word
Numeric	Bignum · Complex · Decimal · Fixed point · Floating point (Double precision · Extended precision · Half precision · Minifloat · Octuple precision · Quadruple precision · Single precision) · Integer (signedness) · Interval · Rational
Text	Character · String (null-terminated)
Pointer	Address (physical · virtual) · Reference
Composite	Algebraic data type (generalized) · **Array** · Associative array · Class · Dependent · Equality · Inductive · List · Object (metaobject) · Option type · Product · Record · Set · Union (tagged)
Other	Boolean · Bottom type · Collection · Enumerated type · Exception · Function type · Opaque data type · Recursive data type · Semaphore · Stream · Top type · Type class · Unit type · Void
Related topics	Abstract data type · Data structure · Generic · Kind (metaclass) · Parametric polymorphism · Primitive data type · Protocol (interface) · Subtyping · Type constructor · Type conversion · Type system

Categories: Arrays | Data types | Composite data types