Dr. B.S.R.V. Prasad
Department of Mathematics

Array (Data Structure)

Arrays (data structures) are a type of linear data structure that can hold an ordered collection of values. Unlike the Array (ADT), the array data structure specifies an implementation in which the values are homogeneous and stored in contiguous memory. They are highly ubiquitous and among the oldest, most widely used data structures in programming.

Unless otherwise specified, the word "array" will refer to the data structure and not the abstract data type for the remainder of these notes.

Arrays are often used to implement abstract data types such as arrays (ADT) and stacks. They could also implement other structures, such as queues and double-ended queues.

Properties of the Array

The most straightforward implementation of an array only stores the memory address of its first index. Some languages also store the size of the array to prevent buffer overflow errors. Insertion and retrieval of an item at a given index is performed by taking the index ii, multiplying by the item size s in the number of bytes, and adding it to the base memory address b.

For zero-based numbering languages, the memory address mm of an index is given by

$$m = b + s \cdot i$$

To store heterogeneously sized items in an array in C, pointers to those items, instead of the items themselves, would have to be explicitly stored. This is how arrays work in higher-level languages like Java and Python. A reference to the actual item is stored instead of storing a raw item directly in the array. This happens implicitly, as Java and Python don't have a pointer type. A consequence is that traversing the array sequentially might mean accessing non-contiguous portions of memory. Instead of all the values existing in a contiguous portion of memory, the values are stored in arbitrary locations in memory that may be far apart from each other. The data's locality and the CPU's ability to cache it efficiently are lost.

Arrays have a fixed size, which is declared when they are created. A more complex data structure, the dynamic array, allows for resizing an array after creation.

Time Complexity

Arrays are among the fastest data structures because the index lookup process is simply an integer multiplication operation followed by an integer addition operation—both very fast operations on modern CPUs. Below are the time complexities of the basic functions the array (ADT) requires.

Operation	Complexity
get	$O(1)$
set	$O(1)$

Space Complexity

Arrays take up linear space with respect to the number of elements n they hold. They also have the benefit that no space is wasted in memory (because it is initialised with a certain size):

$$Space = O(n).$$

Two-dimensional Arrays

Two-dimensional arrays are also stored in contiguous memory. For zero-based numbering languages, the memory address of an item at given indices i and $j$in an array of item size s bytes, r rows, c columns, and base memory address b is given by

$$m = b + s \cdot (c \cdot i + j)$$

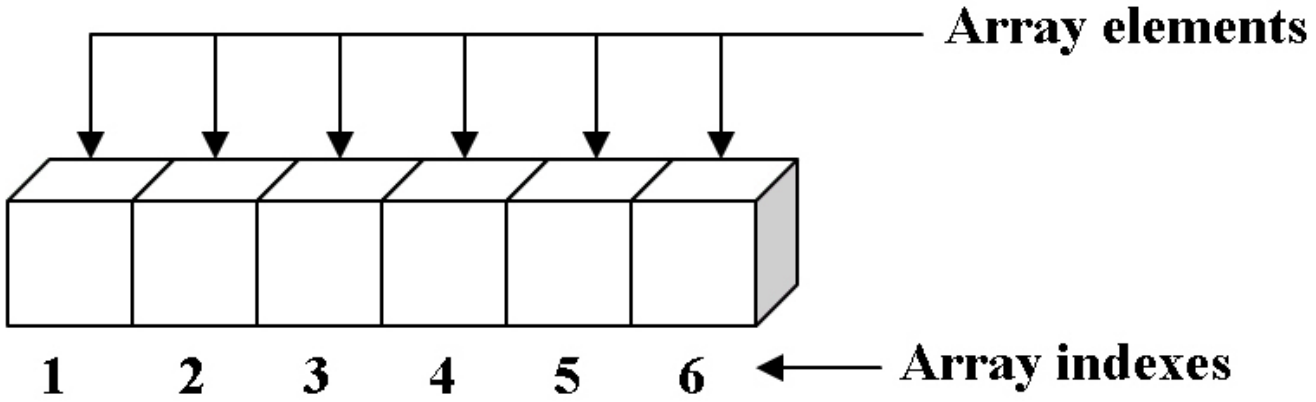
This technique can be generalised for storing n-dimensional arrays in contiguous memory.

Array (ADT)

The **array** is a basic abstract data type with an ordered collection of items accessible by an integer index. These items can be anything from primitive types, such as integers, to more complex types, like instances of classes. Since it's an ADT, it doesn't specify an implementation but is almost always implemented by an array (data structure) or dynamic array.

Unless otherwise specified, the word "array" will refer to the abstract data type and not the data structure for the remainder of this notes.

Arrays have one property: they store and retrieve items using an integer index. An item is stored in a given index and can be retrieved later by specifying the same index. The way these indices work is specific to the implementation, but you can usually just think of them as the slot number in the array that the value occupies. Take a look at the image below:



One-dimensional array with six elements

In this visual representation of an array, you can see that it has a size of six. Once six values have been added, no more can be added until the array is resized or something is removed. You can also see that each "slot" in the array has an associated number. The programmer can directly index into the array to get specific values using these numbers. Note that this image uses one-based numbering. Most languages use zero-based numbering where the first index in an array is 0, not 1.

Python's built-in list provides array functionality. C, C++, C#, Java, and a few other languages have similar syntaxes for working with arrays.

You can think of arrays as a group of numbered boxes holding only one item each. Say we have 10 boxes. That means we can't store 11 items because that's more than the number of boxes we have. If I want to store my phone in box number 5, I can go directly there and place it in the box. Later, if I want to see what's in box number 5, I can again go directly there to look inside and see my phone.

Minimal Required Functionality

Arrays have vastly different functionality across various implementations, but the following operations are common. They form the basis for other, more complicated operations.

DEFINITION

Function Name*	Provided Functionality
`set(i, v)`	Sets the value of index *i* to *v*
`get(i)`	Returns the value of index *i*

Exact names are not required. In fact, some functionality may be provided by a given language directly via special syntax.