


Data Structures/Arrays

 A Wikibookian has nominated this page for cleanup. You can help make it better (https://en.wikibooks.org/w/index.php?title=Data_Structures/Arrays&action=edit). Please review any relevant discussion.

Data Structures
Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs

Arrays

An array is a collection, mainly of similar data types, stored into a common variable. The collection forms a data structure where objects are stored linearly, one after another in memory. Sometimes arrays are even replicated into the memory hardware.

The structure can also be defined as a particular method of storing **elements** of indexed data. Elements of data are logically stored sequentially in blocks within the array. Each element is referenced by an **index**, or subscripts.

The index is usually a number used to address an element in the array. For example, if you were storing information about each day in August, you would create an array with an index capable of addressing 31 values—one for each day of the month. Indexing rules are language dependent, however most languages use either 0 or 1 as the first element of an array.

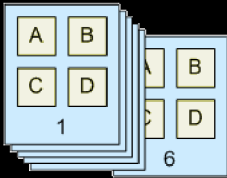
The concept of an array can be daunting to the uninitiated, but it is really quite simple. Think of a notebook with pages numbered 1 through 12. Each page may or may not contain information on it. The notebook is an *array* of pages. Each page is an *element* of the array 'notebook'. Programmatically, you would retrieve information from a page by referring to its number or *subscript*, i.e., notebook(4) would refer to the contents of page 4 of the array notebook.



The notebook (array) contains 12 pages (elements)

Arrays can also be multidimensional - instead of accessing an element of a one-dimensional list, elements are accessed by two or more indices, as from a matrix or tensor.

Multidimensional arrays are as simple as our notebook example above. To envision a multidimensional array, think of a calendar. Each page of the calendar, 1 through 12, is an element, representing a month, which contains approximately 30 elements, which represent days. Each day may or may not have information in it. Programmatically then, calendar(4,15) would refer to the 4th month, 15th day. Thus we have a two-dimensional array. To envision a three-dimensional array, break each day up into 24 hours. Now calendar(4,15,9) would refer to 4th month, 15th day, 9th hour.



A simple 6 element by 4 element array

Array<Element> Operations

make-array(integer *n*): Array

Create an array of elements indexed from to , inclusive. The number of elements in the array, also known as the size of the array, is *n*.

get-value-at(Array *a*, integer *index*): Element

Returns the value of the element at the given *index*. The value of *index* must be in bounds: *0* <= *index* <= (*n* - 1). This operation is also known as **subscripting**.

set-value-at(Array *a*, integer *index*, Element *new-value*)

Sets the element of the array at the given *index* to be equal to *new-value*.

Arrays guarantee **constant** time read and write access, , however many lookup operations (find_min, find_max, find_index) of an instance of an element are **linear** time, . Arrays are very efficient in most languages, as operations compute the address of an element via a simple formula based on the base address element of the array.

Array implementations differ greatly between languages: some languages allow arrays to be re-sized automatically, or to even contain elements of differing types (such as Perl). Other languages are very strict and require the type and length information of an array to be known at run time (such as C).

Arrays typically map directly to contiguous storage locations within your computer's memory and are therefore the "natural" storage structure for most higher level languages.

Simple linear arrays are the basis for most of the other data structures. Many languages do not allow you to allocate any structure except an array, everything else must be implemented on top of the array. The exception is the linked list, that is typically implemented as individually allocated objects, but it is possible to implement a linked list within an array.

Type

The array index needs to be of some type. Usually, the standard integer type of that language is used, but there are also languages such as [Ada](#) and [Pascal](#) which allow any discrete type as an array index. Scripting languages often allow any type as an index (associative array).

Bounds

The array index consists of a range of values with a lower bound and an upper bound.

In some programming languages only the upper bound can be chosen while the lower bound is fixed to be either 0 ([C](#), [C++](#), [C#](#), [Java](#)) or 1 ([FORTRAN 66](#), [R](#)).

In other programming languages ([Ada](#), [PL/I](#), [Pascal](#)) both the upper and lower bound can be freely chosen (even negative).

Bounds check

The third aspect of an array index is the check for valid ranges and what happens when an invalid index is accessed. This is a very important point since the majority of [computer worms](#) and [computer viruses](#) attack by using invalid array bounds.

There are three options open:

- 1. Most languages ([Ada](#), [PL/I](#), [Pascal](#), [Java](#), [C#](#)) will check the bounds and raise some error condition when an element is accessed which does not exist.
- 2. A few languages ([C](#), [C++](#)) will not check the bounds and return or set some arbitrary value when an element outside the valid range is accessed.
- 3. Scripting languages often automatically expand the array when data is written to an index which was not valid until then.

Declaring Array Types

The declaration of array type depends on how many features the array in a particular language has.

The easiest declaration is when the language has a fixed lower bound and fixed index type. If you need an array to store the monthly income you could declare in [C](#)

```
typedef double Income[12];
```

This gives you an array with in the range of 0 to 11. For a full description of arrays in C see [C Programming/Arrays](#).

If you use a language where you can choose both the lower bound as well as the index type, the declaration is—of course—more complex. Here are two examples in [Ada](#):

```
type Month is range 1 .. 12;
type Income is array(Month) of Float;
```

or shorter:

```
type Income is array(1 .. 12) of Float;
```

For a full description of arrays in Ada see [Ada Programming/Types/array](#).

Array Access

We generally write arrays with a name, followed by the index in some brackets, square '['] or round '()'. For example, `August[3]` is the method used in the C programming language to refer to a particular day in the month.

Because the C language starts the index at zero, `August[3]` is the 4th element in the array. `august[0]` actually refers to the first element of this array. Starting an index at zero is natural for computers, whose internal representations of numbers begin with zero, but for humans, this unnatural numbering system can lead to problems when accessing data in an array. When fetching an element in a language with zero-based indexes, keep in mind the *true* length of an array, lest you find yourself fetching the wrong data. This is the disadvantage of programming in languages with fixed lower bounds, the programmer must always remember that "[o]" means "1st" and, when appropriate, add or subtract one from the index. Languages with variable lower bounds will take that burden off the programmer's shoulder.

We use indexes to store *related* data. If our C language array is called `august`, and we wish to store that we're going to the supermarket on the 1st, we can say, for example

```
august[0] = "Going to the shops today"
```

In this way, we can go through the indexes from 0 to 30 and get the related tasks for each day in `august`.

Data Structures

[Introduction](#) - [Asymptotic Notation](#) - [Arrays](#) - [List Structures & Iterators](#)

[Stacks & Queues](#) - [Trees](#) - [Min & Max Heaps](#) - [Graphs](#)

[Hash Tables](#) - [Sets](#) - [Tradeoffs](#)

Retrieved from "https://en.wikibooks.org/w/index.php?title=Data_Structures/Arrays&oldid=3287559"

This page was last edited on 31 August 2017, at 06:44.

Text is available under the [Creative Commons Attribution-ShareAlike License](#).; additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#).

