ARRAY DATA STRUCTURE

Isha Batra, Divya Raheja
Information Technology
Dronacharya College Of Engineering, Farukhnagar,Gurgaon

***Abstract*- In computer science, an array data structure or simply an array is a data structure consisting of a collection of *elements* (values or variables), each identified by at least one *array index* or *key*. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array. For example, an array of 10 32-bit integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, … 2036, so that the element with index *i* has the address 2000 + 4 × *i*. The term *array* is often used to mean array data type, a kind of data type provided by most high-level programming languages that consists of a collection of values or variables that can be selected by one or more indices computed at run-time. Array types are often implemented by array structures; however, in some languages they may be implemented by hash tables, linked lists, search trees, or other data structures. Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.**

***Index Terms*- data type , structures ,vectors, dynamic memory allocation, identifiers**

I. INTRODUCTION

Arrays are among the oldest and most important data structures, and are used by almost every program. They are also used to implement many other data structures, such as lists and strings. They effectively exploit the addressing logic of computers. In most modern computers and many external storage devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially vector processors, are often optimized for array operations.Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually but not always fixed while the array is in use. The term *array* is often used to mean array data type, a kind of data type provided by most high-level programming languages that consists of a collection of values or variables that can be selected by one or more indices computed at run-time. Array types are often implemented by array structures; however, in some languages they may be implemented by hash tables, linked lists, search trees, or other data structures. The term is also used, especially in the description of algorithms, to mean associative array or "abstract array", a theoretical computer science model (an abstract data type or ADT) intended to capture the essential properties of arrays.

- **APPLICATIONS**

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many structures small and large, consist of (or include) one-dimensional arrays whose elements are records.Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists. One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably.

Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array. The array may contain subroutine pointers (or relative subroutine numbers that can be acted upon by SWITCH statements) that direct the path of the execution.

- **ELEMENTS IDENTIFIER AND ADDRESSING FORMULAS**

When data objects are stored in an array, individual objects are selected by an index that is usually a non-

negative scalar integer. Indices are also called subscripts. An index *maps* the array value to a stored object. There are three ways in which the elements of an array can be indexed:

- **0** (*zero-based indexing*): The first element of the array is indexed by subscript of 0.
- **1** (*one-based indexing*): The first element of the array is indexed by subscript of 1.
- **n** (*n-based indexing*): The base index of an array can be freely chosen. Usually programming languages allowing *n-based indexing* also allow negative index values and other scalar data types like enumerations, or characters may be used as an array index.

Arrays can have multiple dimensions, thus it is not uncommon to access an array using multiple indices. For example a two-dimensional array A with three rows and four columns might provide access to the element at the 2nd row and 4th column by the expression A[1, 3] (in a row major language) or A[3, 1] (in a column major language) in the case of a zero-based indexing system. Thus two indices are used for a two-dimensional array, three for a three-dimensional array, and *n* for an *n*-dimensional array.The number of indices needed to specify an element is called the dimension, dimensionality, or rank of the array.In standard arrays, each index is restricted to a certain range of consecutive integers (or consecutive values of some enumerated type), and the address of an element is computed by a "linear" formula on the indices.

- **One-dimensional arrays**

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.As an example consider the C declaration int anArrayName[10];Syntax : datatype anArrayname[sizeofArray];In the given example the array can contain 10 elements of any value available to the int type. In C, the array element indices are 0-9 inclusive in this case. For example, the expressions anArrayName[0] and anArrayName[9] are the first and last elements respectively. For a vector with linear addressing, the element with index i is located at the address $B + c \times i$, where B is a fixed *base address* and c a fixed constant, sometimes called the *address increment* or *stride*. If the valid element indices begin at 0, the constant B is simply the address of the first element of the array. For this reason, the C programming language specifies that array indices always begin at 0; and many programmers will call that element "zeroth" rather than "first".

However, one can choose the index of the first element by an appropriate choice of the base address B. For example, if the array has five elements, indexed 1 through 5, and the base address B is replaced by $B + 30c$, then the indices of those same elements will be 31 to 35. If the numbering does not start at 0, the constant B may not be the address of any element.

- **Multidimensional arrays**

For a two-dimensional array, the element with indices i,j would have address $B + c \cdot i + d \cdot j$, where the coefficients c and d are the *row* and *column address increments*, respectively.More generally, in a k-dimensional array, the address of an element with indices $i_1, i_2, \ldots, i_k$ is

$B + c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + c_k \cdot i_k$.For example: int a[3][2];

This means that array a has 3 rows and 2 columns, and the array is of integer type. Here we can store 6 elements they are stored linearly but starting from first row linear then continuing with second row. The above array will be stored as a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}. This formula requires only k multiplications and k additions, for any array that can fit in memory. Moreover, if any coefficient is a fixed power of 2, the multiplication can be replaced by bit shifting.

The coefficients c_k must be chosen so that every valid index tuple maps to the address of a distinct element. If the minimum legal value for every index is 0, then B is the address of the element whose indices are all zero. As in the one-dimensional case, the element indices may be changed by changing the base address B. Thus, if a two-dimensional array has rows and columns indexed from 1 to 10 and 1 to 20, respectively, then replacing B by $B + c_1 - - 3\ c_1$ will cause them to be renumbered from 0 through 9 and 4 through 23, respectively. Taking advantage of this feature, some languages (like FORTRAN 77) specify that array indices begin at 1, as in mathematical tradition; while other languages (like Fortran 90, Pascal and Algol) let the user choose the minimum value for each index.

II. COMPACT LAYOUTS

Often the coefficients are chosen so that the elements occupy a contiguous area of memory. However, that is not necessary. Even if arrays are always created with contiguous elements, some array slicing operations may create non-contiguous sub-arrays from them.

There are two systematic compact layouts for a two-dimensional array. For example, consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

In the row-major order layout (adopted by C for statically declared arrays), the elements in each row are stored in consecutive positions and all of the elements of a row have a lower address than any of the elements of a consecutive row:

1	2	3	4	5	6	7	8	9

In column-major order (traditionally used by Fortran), the elements in each column are consecutive in memory and all of the elements of a column have a lower address than any of the elements of a consecutive column:

1	4	7	2	5	8	3	6	9

For arrays with three or more indices, "row major order" puts in consecutive positions any two elements whose index tuples differ only by one in the *last* index. "Column major order" is analogous with respect to the *first* index.

In systems which use processor cache or virtual memory, scanning an array is much faster if successive elements are stored in consecutive positions in memory, rather than sparsely scattered. Many algorithms that use multidimensional arrays will scan them in a predictable order. A programmer (or a sophisticated compiler) may use this information to choose between row- or column-major layout for each array. For example, when computing the product $A \cdot B$ of two matrices, it would be best to have A stored in row-major order, and B in column-major order.

III. RESIZING

Static arrays have a size that is fixed when they are created and consequently do not allow elements to be inserted or removed. However, by allocating a new array and copying the contents of the old array to it, it is possible to effectively implement a *dynamic* version of an array; see dynamic array. If this operation is done infrequently, insertions at the end of the array require only amortized constant time.Some array data structures do not reallocate storage, but do store a count of the number of elements of the array in use, called the count or size. This effectively makes the array a dynamic array with a fixed maximum size or capacity; Pascal strings are examples of this.

IV. NON-LINEAR FORMULAS

More complicated (non-linear) formulas are occasionally used. For a compact two-dimensional triangular array, for instance, the addressing formula is a polynomial of degree 2.

V. EFFICIENCY

Both *store* and *select* take (deterministic worst case) constant time. Arrays take linear (O(*n*)) space in the number of elements *n*that they hold. In an array with element size *k* and on a machine with a cache line size of B bytes, iterating through an array of *n* elements requires the minimum of ceiling(*nk*/B) cache misses, because its elements occupy contiguous memory locations. This is roughly a factor of B/*k* better than the number of cache misses needed to access *n* elements at random memory locations. As a consequence, sequential iteration over an array is noticeably faster in practice than iteration over many other data structures, a property called locality of reference (this does *not* mean however, that using a perfect hash or trivial hash within the same (local) array, will not be even faster - and achievable in constant time). Libraries provide low-level optimized facilities for copying ranges of memory (such as memcpy) which can be used to move contiguous blocks of array elements significantly faster than can be achieved through individual element access. The speedup of such optimized routines varies by array element size, architecture, and implementation. Memory-wise, arrays are compact data structures with no per-element overhead. There may be a per-array overhead, e.g. to store index bounds, but this is language-dependent. It can also happen that elements stored in an array require *less* memory than the same elements stored in individual variables, because

several array elements can be stored in a single word; such arrays are often called *packed* arrays. An extreme (but commonly used) case is the bit array, where every bit represents a single element. A single octet can thus hold up to 256 different combinations of up to 8 different conditions, in the most compact form.Array accesses with static.

VI. COMPARISON WITH OTHER DATA STRUCTURES

Comparison of list data structures

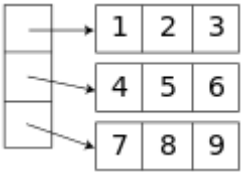
	Linked list	Array	Dynamic array	Balanced tree	Random access list
Indexing	Θ(*n*)	Θ(1)	Θ(1)	Θ(log n)	Θ(log n)
Insert/delete at beginning	Θ(1)	N/A	Θ(*n*)	Θ(log n)	Θ(1)
Insert/delete at end	Θ(*n*) when last element is unknown; Θ(1) when last element is known	N/A	Θ(1) amortized	Θ(log *n*)	Θ(log *n*) updating
Insert/delete in middle	search time + Θ(1)	N/A	Θ(*n*)	Θ(log *n*)	Θ(log *n*) updating
Wasted space (average)	Θ(*n*)	0	Θ(*n*)	Θ(*n*)	Θ(*n*)

Growable arrays are similar to arrays but add the ability to insert and delete elements; adding and deleting at the end is particularly efficient. However, they reserve linear (Θ(*n*)) additional storage, whereas arrays do not reserve additional storage.

Associative arrays provide a mechanism for array-like functionality without huge storage overheads when the index values are sparse. For example, an array that contains values only at indexes 1 and 2 billion may benefit from using such a structure. Specialized associative arrays with integer keys include Patricia tries, Judy arrays, and van Emde Boas trees.

Balanced trees require O(log *n*) time for indexed access, but also permit inserting or deleting elements in O(log *n*) time, whereas growable arrays require linear (Θ(*n*)) time to insert or delete elements at an arbitrary position.

Linked lists allow constant time removal and insertion in the middle but take linear time for indexed access. Their memory use is typically worse than arrays, but is still linear.



An Iliffe vector is an alternative to a multidimensional array structure. It uses a one-dimensional array of references to arrays of one dimension less. For two dimensions, in particular, this alternative structure would be a vector of pointers to vectors, one for each row. 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 alternative structure allows *ragged* or *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. It also saves one multiplication (by the column address increment) replacing it by a bit shift (to index the vector of row pointers) and one extra memory access (fetching the row address), which may be worthwhile in some architectures.

REFRENCES

[1] Black, Paul E. (13 November 2008). "arraY". Dictionary of Algorithms and Data Structures. National Institute of Standards and Technology. Retrieved 22 August 2010.

[2] Bjoern Andres; Ullrich Koethe; Thorben Kroeger; Hamprecht (2010). "Runtime-Flexible Multi-dimensional Arrays and Views for C++98 and C++0x".

[3] Garcia, Ronald; Lumsdaine, Andrew (2005). "MultiArray: a C++ library for generic programming with arrays". Software: Practice and Experience **35** (2): 159–188.

[4] David R. Richardson (2002), The Book on Data Structures. iUniverse, 112 pages,

[5] T. Veldhuizen. Arrays in Blitz++. In Proc. of the 2nd Int. Conf. on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), LNCS 1505, pages 223-220. Springer, 1998.

[6] http://en.wikipedia.org/wiki/Array_data_structure