Data Structures (CSC-154)

B. Sc. IT Computer Science, 2nd Semester

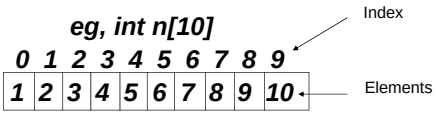
Tribhuvan University

Lecture 03: Arrays

Data Structures

Array

- A ***finite ordered*** set of ***homogeneous*** elements
- Finite - specific numbers of elements
- Ordered - arranged one after another
- Homogeneous - similar types
  - All integers, all floating points etc.



Array

- Array name is actually a pointer to the first location of memory block
- Array can be initialized only during declaration time

  eg, int n[10] = {1, 2, 3 …9};

  eg, int n[ ] = {1, 2, 3 …9};
- There is no bound checking concept for arrays in C (any no. of values can be entered)

Array Bound Checking

- Array Bounds (index) are not verified neither at compile-time nor at run-time
- Index must be within 0 and arraysize-1
- If not others data may be overwritten

```
int a=100,b[5],c=200;
int i;
for (i=0; i < 6; i++)
   b[i] = i;
printf("a=%d, c=%d\n", a, c);
```


Array Operations

- Two basic operations are defined
- **Extraction** – accepts an array **a**, an index **i**, and returns an element of the array
- **Storing** – accepts an array **a**, an index **i**, and an element **x** (a[i] = x)
- Operations in arrays are vary fast

Limits in Array

- Lower bound - smallest array's index i.e 0
- Upper bound - highest array's index
- Range - Upper bound – Lower bound + 1
- Lower and upper bound never changes during program execution

1 D Array

- Only one subscription is needed to specify a parameter element of the array
- eg. datatype variable[expression]

  eg, int n[10] ← ***Only one subscript i.e, 10***
- Size in bytes = size of array * size of (base type)
- eg, float var[5]
- Size of var = 5 * 4 bytes = 20 bytes

Implementation in memory

- Address of an element n[k] = B + W*k
- B – Base address
- W – size of datatype
- k- array index
- Eg. n[5]
  - Let B = 2000, W = 2 bytes (int) and k = 5(6^{th} value)
  - So, address of n[5] is 2000 + 2*5 = 2010

Traversing and Merging

- Traversing
  - Accessing all the elements of the array from first element up to the last one by one
- Merging
  - Adding elements of one array (a) into another array (b)
  - Merging can be done in the new array (c)

Arrays as Parameter

- Arrays can be passed to function in two ways;
  - Passing as the array
    - Passing the whole array to the function makes another copy of the same array, this causes unwanted duplication
  - Passing as pointer to the array
    - Since an array variable is a pointer, array parameters are passed by reference, i.e, base address of the array is passed

Example

- Passing by reference
  - Float avg(float a[], int size)

```
{
  int i;
  float sum = 0;
  for(i=0; i<size; i++)
        sum+=a[i];
  return (sum/size);
}
Function call:
  average = avg(a, size);
```


Array as an ADT

ADT:

- A useful tool for specifying the ***logical properties*** of a data type.
- Type defined for the ***data*** that contains values and set of operations

  An ADT consists of
- **Data definition** (Data Holder)
  - Pre – conditions
  - Post – conditions (Situation of the data holder after calculation)
- **Operator definition**

Array as an ADT

<ADT definition> Array
Dataholder[items] //Holds data
Index //serial number of items

<process>Process name:<returns none>**Store** (Array, I, element)
Pre-condition: the index should be within the range
Method:
Post-condition:Array[i]=element
<end process>

<process>Process name:<returns item>**Extract** (Array, i)
Pre-condition:the index should be within the range
Post-condition: Extract=Array[i]
<end process>

<end definition>

2-D Array

- Defined as arrays of array
- The component type of an array can be another array
- eg, int a[3][4];
  - Array containing 3 elements, and each of these elements is itself an array containing 4 elements

2-D Array

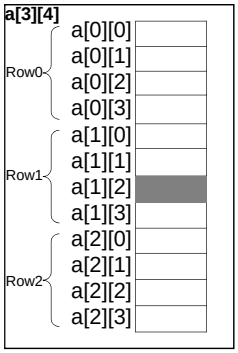
- int a[3][4]

	Column 0	Column 1	Column 2	Column 3
Row 0				
Row 1			a[1][2]	
Row 2				

- It is only a logical data structure that is useful in programming and problem solving

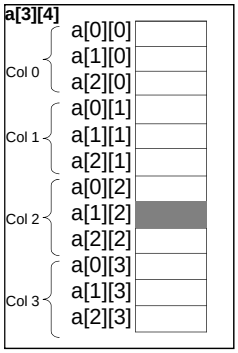
Representation of 2-D Array

- **Row-major representation**
  - First row of the array occupies first set of memory locations, second occupies second and so on.
    eg arr[r][c]
  - Address of arr[i][j] is given by
    base(arr)+(i*c+j)*esize
    i.e base(arr)+(1*4+2)*esize



Representation of 2-D Array

- **Column-major representation**
  - For column major representation, memory allocation is done column by column, ie, first the element of the complete first column is stored, then elements of second and so on.
    eg arr[r][c]
  - Address of arr[i][j] is given by
    base(arr)+(j*r+i)*esize
    i.e base(arr)+(2*3+1)*esize



Lab Work (Include in the lab report)

1. Write a program to implement linear list using array. It should have following functions, calculating the position using the formula **base(arr)+(index)*esize;**
   1. Store (array, position, value)
   2. Extract (array, position)
   3. Display (array)
   4. ClearAll(array)
   5. Exit
2. Make another similar type of program for 2 Dimensional Array.
3. WAP of calculator for RATIONAL numbers.

Multi-dimensional Arrays

- Recall: An array is a sequence of data items that are of the same type, that can be indexed, and that are stored contiguously
- Each element of an array is like a single item of a particular type
- But an array itself is an item of a particular type
  - So, an array element could be another array
- An "array-of-arrays" is called "multi-dimensional" arrays whose elements are themselves arrays
  - No of subscript determines the dimension of the array


```
int a[3][4];
int i, j;

for (i = 0; i < 3; ++i)
   for (j = 0; j < 4; ++j)
      a[i][j] = i+j;

for (i =0; i < 3; ++i)
{
   for (j = 0; j < 4; ++j)
      printf("a[%d][%d] = %d  ", i, j, a[i][j]);
   printf("\n");
}

printf("%d\n", a[2][1]/2);
printf("%d\n", a[1][1] * (a[0][0]+2));
printf("%d\n", a[3][1]/2);
```


Initialization

- List the values separated by commas and enclosed in braces
  - `int a[2][3] = { 1, 2, 3, 4, 5, 6};`
  - The values will be assigned in the order they appear
- Initializers can be grouped with braces
  - `int a[2][3] = { {1, 2, 3}, {4, 5, 6}};`

1	2	3
4	5	6

- If not enough, unspecified elements set to zero
  - `int a[2][3] = { {1, 2}, {3, 4}};`
- You can leave the size for first subscript
  - `int a[][3] = { {1, 2}, {3, 4}};`

1	2	0
3	4	0

Passing Multidimensional Arrays to Function

- Specify the array variable name, while passing it to a function
  - only the address of the first element is actually passed
- The parameter receiving the array must define the size of all dimension, except the first one
- Any changes to array elements within the function affects the "original" array elements

```
int a[3][4];
func(a);
```

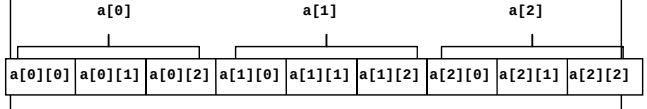
Function Call

```
void func(int x[][4])
{
}
```

Multidimensional array in parameter

Multidimensional Arrays in Memory

- Each array within a multidimensional array stored sequentially in memory as with one-dimensional array
- For two-dimensional array, all elements in first row is stored, then the elements of second row and so on



Array of Strings

- You can create array of strings using a two-dimensional character array

  ```
  char months[12][10];
  ```

  – Left dimension determines the number of strings, and right dimension specifies the maximum length of each string
  – Now you can use the array `months` to store 12 strings each of which can have a maximum of 10 characters (including the null)
  – To access an individual string, you specify only the left subscript

  ```
  puts(months[2]);
  ```

  prints the third month

Example

```
char months[12][10] =
{
"January",
"February",
"March",
"April",
"May",
"June",
"July",
"August",
"September",
"October",
"November",
"December"
};
printf("%s \n", months[5]);
```

months[0]	J	a	n	u	a	r	y	\0		
months[1]	F	e	b	r	u	a	r	y	\0	
months[2]	M	a	r	c	h	\0				
months[3]	A	p	r	i	l	\0				
months[4]	M	a	y	\0						
months[5]	J	u	n	e	\0					
months[6]	J	u	l	y	\0					
months[7]	A	u	g	u	s	t	\0			
months[8]	S	e	p	t	e	m	b	e	r	\0
months[9]	O	c	t	o	b	e	r	\0		
months[10]	N	o	v	e	m	b	e	r	\0	
months[11]	D	e	c	e	m	b	e	r	\0	

The End