Data Structures (CS 21001)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School Of Computer Engineering

Dr. Amiya Ranjan Panda
Assistant Professor [II]
School of Computer Engineering,
Kalinga Institute of Industrial Technology (KIIT),
Deemed to be University, Odisha

4 Credit

Lecture Note 02

Array

- Collection of similar types of data items stored at contiguous memory locations.
- Considered as derived data type.
- Simplest data structure where each data element can be randomly accessed by using its index number.

- Example: To store the marks in 10 subjects, need not define different variables
- Define an array to store the marks in each subject
- The array marks[10] defines the marks of the student in 10 different subjects

Properties of the Array

- Each element of array are same data type and carries a same size i.e. int = 4 bytes
- Elements of the array are stored at contiguous memory locations
- Elements of the array can be randomly accessed since the address of each element of the array is calculated with the given base address and the size of data element
- Example, in C language, the syntax of declaring an array:
  - int iarr[10];
  - char carr[10];
  - float farr[5]

Types of Arrays

- One - Dimensional Array
- Two - Dimensional Array
- Multi - Dimensional Array

Need of using Array

- Require to store a large number of data of similar type
- To store such an amount of data, a large number of variables need to be defined
- It would be very difficult to remember the names of all the variables while writing the programs
- Instead of naming all the variables with a different name, it is better to define an array and store all the elements in it.

Program

- **Program without array:**

```
#include <stdio.h>
void main ()  {
    int marks_1 = 56, marks_2 =
    78, marks_3 = 88, marks_4 =
    76, marks_5 = 56, marks_6 =
    89;
    float avg = (marks_1 +
    marks_2 + marks_3 + marks_4
    + marks_5 +marks_6) / 6.0 ;
    printf(“%f”, avg);
}
```

Program by using array:

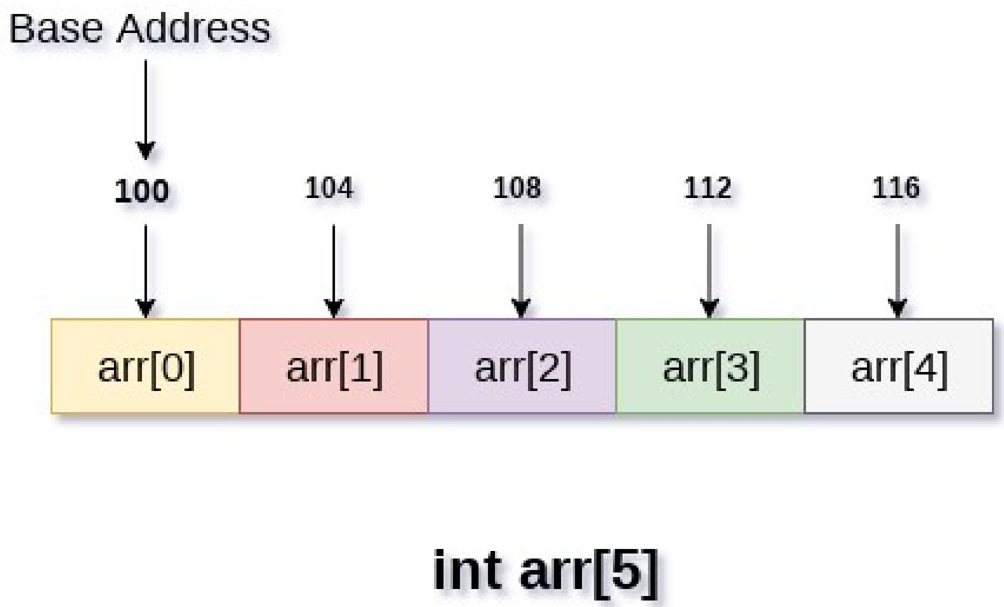
```
#include <stdio.h>
int main ()  {
    int marks[6] = {56, 78, 88, 76, 56, 89};
    int i;
    float avg;
    for (i=0; i<6; i++ ) {
        avg = avg + marks[i];
    }
    avg = avg/6.0;
    printf(“%f”, avg);
    return 0;
}
```

Advantages of Array

- Array provides the single name for the group of variables of the same type
- Easy to remember the name of all the elements of an array
- Traversing an array is a very simple process
- Any element in the array can be directly accessed by using the index.

Memory Allocation of the Array

- All the data elements of an array are stored at contiguous locations in the main memory
- Name of the array represents the base address or the address of first element in the main memory
- Each element of the array is represented by a proper indexing



One-Dimensional Array

- A collections of data elements given by one variable name using only one subscript

Declaration:

```
data_type  Array_name[size]
```

data_type : valid data type like int, float or char

Arrayname : valid identifier

size : maximum number of elements that can be stored in array

Example:

```
int arr[5];
```

One-Dimensional Array

- An elements of an array must be initialized, otherwise they may contain “garbage” value.
- An array can be initialized at either of the following stages
  - At compile time
  - At run time

Compile Time Initialization

- We can initialize the elements of arrays as:

```
type array_name[size] = {list of values};
```

For example,

```
int number[3] = {5,10,15};
float total[5] ={0.0, 15.75, -10.9};
```

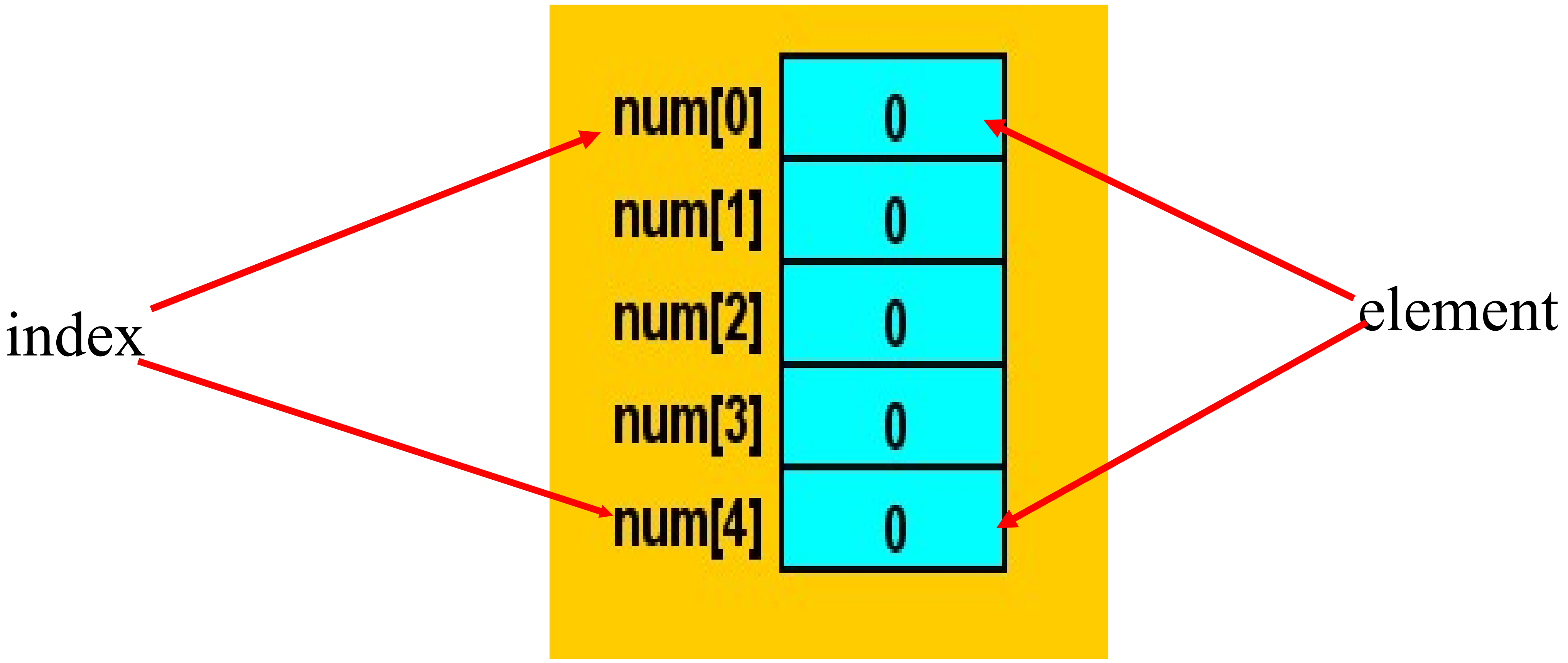
- The size may be omitted

```
int counter[] = {1,1,1,1};
```

- The character array may be initialized in the similar manner

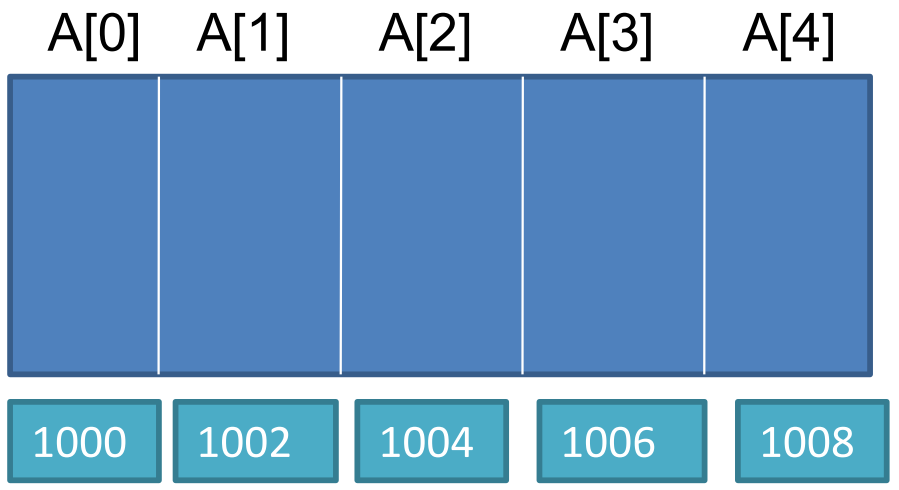
```
char name[] = {‘M’,’O’,’H’,’I’,’T’,’\0’};
char name[] = “MOHIT”;
```

Accessing of 1D Array Elements



Memory Representation of an Array

- The array elements are stored in contigous memory location



Address of the ith element(Address(A[i]))=B+w*(i-LB)

where B=Base address of the array A

w=Size of the each element

LB =Lower index of the array

Accessing Elements of an Array

- To access any random element of an array it needs the following information:
  - Base Address of the array.
  - Size of an element in bytes.

- Address of any element of a 1D array can be calculated by using the following formula:
  - Byte address of element A[i] = base address + size * (i - first index) [General]

  - Byte address of element A[i] = base address + size * (i) [if array index start with 0]
  - Byte address of element A[i] = base address + size * (i -1) [if array index start with 1]

- Example: In an array, A[-10 ..... +2], Base address (BA) = 1001, size of an element = 2 bytes, find the location of A[-1].

  $L(A[-1]) = 1001 + [(-1) - (-10)] \times 2$

  $= 1001 + 18$

  $= 1019$

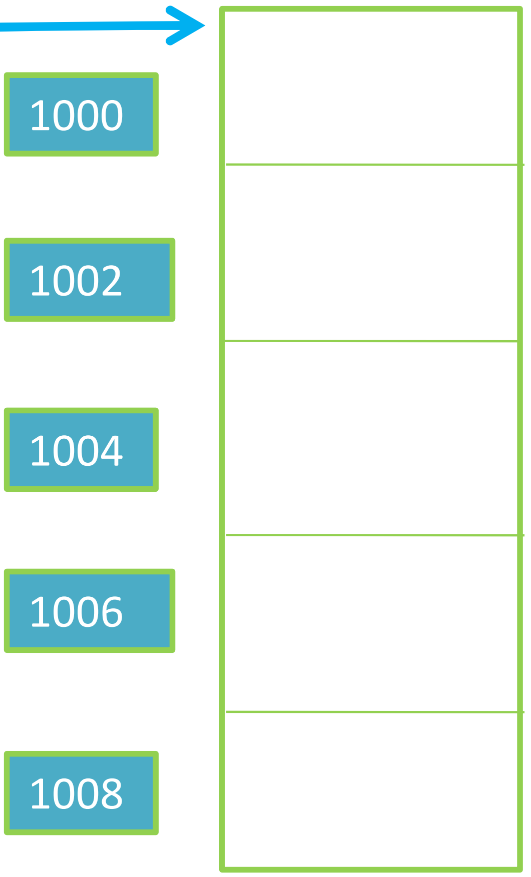
Run-Time initialization

- An array can be explicitly initialized at run time.

```
void main()
{
  int arr[5],i;
  prin 5 Enter array elements:”);
  for(i=0;i<5;i++)
  {
      scanf(“%d”,&arr[i]);
  }
}
```

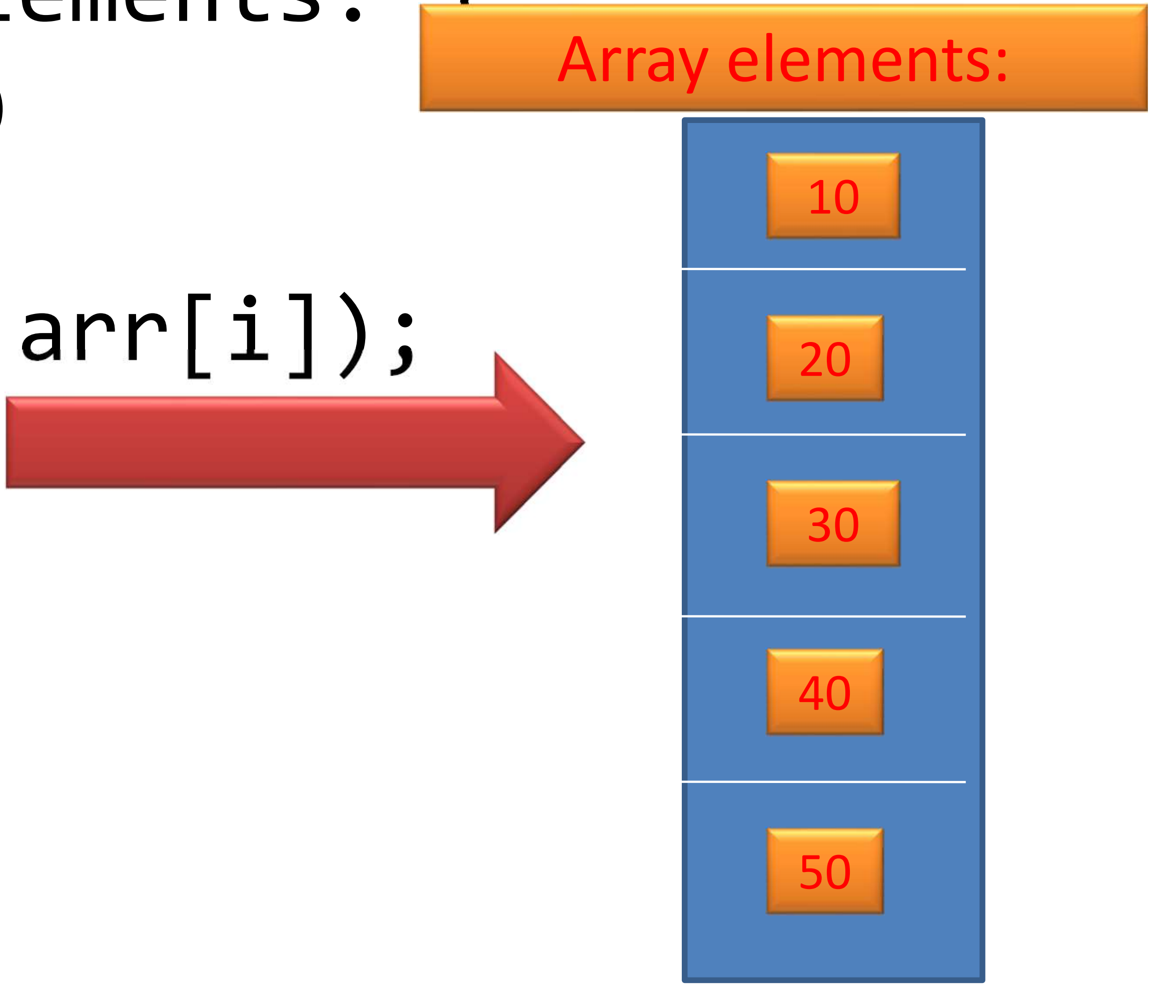
Enter Array elements:

10 20 30 40 50



Run-Time initialization

```
printf(“Array elements:”);
for(i=0;i<5;i++)
{
        printf(“%d”,arr[i]);
}
}
```



PROGRAMS

Program to input elements for an array and display the array elements.

```
#include<stdio.h>
void main( ){
    int a[5],i;
    printf("Enter the elements  for the array:");
    for(i=0;i<5;i++)
            scanf("%d",&a[i]);           //reading the array elements
    printf("The array elements are:\t");
    for(i=0;i<5;i++)
            printf("%d\t",a[i]);         //displaying the array elements
}
```

Output:

```
Enter the elements for the array: 1 2 3 4 5
    The array elements are:      1 2 3 4 5
```

PROGRAMS

Program to input elements for an array and calculating the sum of all array elements.

```
#include<stdio.h>
void main( ){
        int a[5],I,sum=0;
        printf("\nEnter the elements  of the array:");
        for(i=0;i<5;i++)
                        scanf("%d",&a[i]);                  //reading the array elements
        for(i=0;i<5;i++)
                        sum=sum+a[i];
        printf("\nThe array elements are:\t");
        for(i=0;i<5;i++)
                        printf("%d\t",a[i]);                //displaying the array elements
        printf("\nThe sum of elements =%d",sum);
}
```

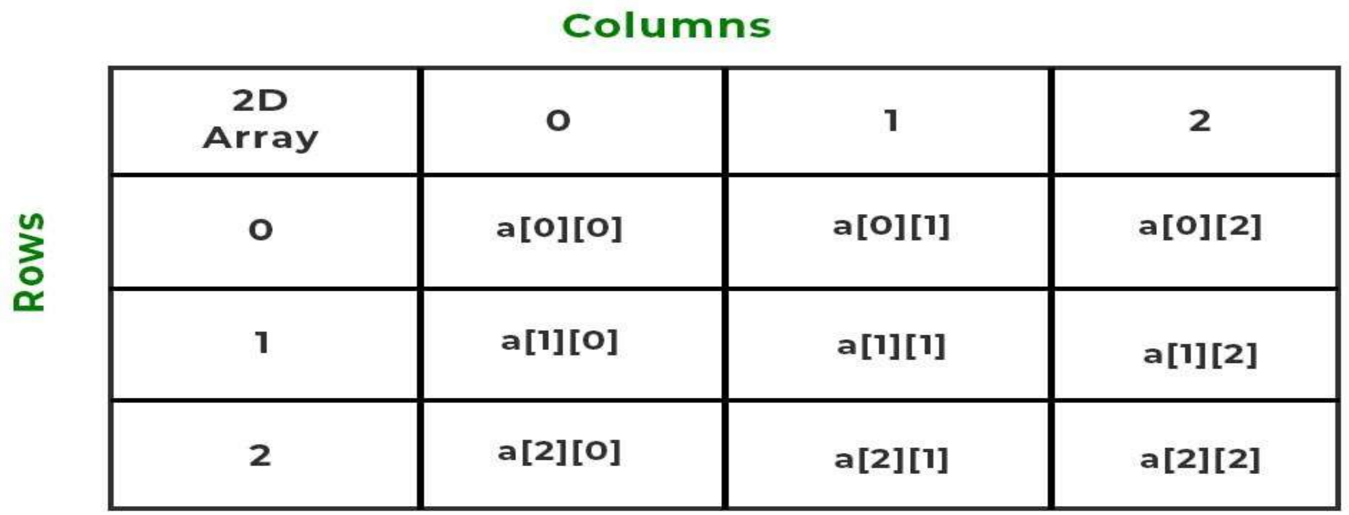
```
Output:  Enter the elements of the array: 1 2 3 4 5
                The array elements are:1 2 3 4 5
                Sum of elements=15
```

Memory layout of multi-dimensional arrays

- what memory layout to use for storing the data, and
- how to access such data in the most efficient manner

- Computer memory is inherently linear: A one-dimensional structure, mapping multi-dimensional data on it can be done in several ways.
- Programmer notation for matrices: rows and columns start with zero, at the top-left corner of the matrix.
  - Row indices go over rows from top to bottom
  - column indices go over columns from left to right

2D Array



Columns

Rows

2D Array	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]

- We cannot put this 2-D array in to memory, because the random access will fail.
- To put this 2-D array into memory , we mapped it into 1-D array.

Mapping (2-D Array to 1-D Array)

- Row-major
  - Puts the first row in contiguous memory, then the second row right after it, then the third, and so on.
  - In row-major layout, column indices change faster.

- Column-major
  - Puts the first column in contiguous memory, then the second, etc.
  - In column-major layout, row indices change the faster.

Address calculation of the element using row-major order

- **Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))**
- *I = Row Subset of an element whose address to be found,*
  J = Column Subset of an element whose address to be found,
  B = Base address,
  W = Storage size of one element store in an array(in byte),
  LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
  LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
  N = Number of column given in the matrix.

Example

- Given an array, **arr[1………10][1………15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order.

Solution:

- **Given:**
  Base address B = 100
  Storage size of one element store in any array W = 1 Bytes
  Row Subset of an element whose address to be found I = 8
  Column Subset of an element whose address to be found J = 6
  Lower Limit of row/start row index of matrix LR = 1
  Lower Limit of column/start column index of matrix = 1
  Number of column given in the matrix N = Upper Bound – Lower Bound + 1
  = 15 – 1 + 1
  = 15
- **Formula:**
  Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))
- **Solution:**
  Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))
  = 100 + 1 * ((7) * 15 + (5))
  = 100 + 1 * (110)
  Address of A[I][J] = 210

Address calculation of the element using Column-major order

- **Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))**
- *I = Row Subset of an element whose address to be found,*
  J = Column Subset of an element whose address to be found,
  B = Base address,
  W = Storage size of one element store in any array(in byte),
  LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
  LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
  M = Number of rows given in the matrix.

Example

- Given an array **arr[1………10][1………15]** with a base value of **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.
- Solution
- ***Given:***
  Base address B = 100
  Storage size of one element store in any array W = 1 Bytes
  Row Subset of an element whose address to be found I = 8
  Column Subset of an element whose address to be found J = 6
  Lower Limit of row/start row index of matrix LR = 1
  Lower Limit of column/start column index of matrix = 1
  Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1
  = 10 – 1 + 1
  = 10
- ***Formula: used***
  *Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))*
  *Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))*
  *= 100 + 1 * ((5) * 10 + (7))*
  *= 100 + 1 * (57)*
  Address of A[I][J] = 157

Explanation of both answer

- *From the above examples, it can be observed that for the same position two different address locations are obtained that's because in row-major order movement is done across the rows and then down to the next row, and in column-major order, first move down to the first column and then next column. So both the answers are right.*

- *So it's all based on the position of the element whose address is to be found for some cases the same answers is also obtained with row-major order and column-major order and for some cases, different answers are obtained.*

Problems

Each element of an array arr[15][20] requires 'W' bytes of storage. If the address of arr[6][8] is 4440 and the base address at arr[1][1] is 4000, find the width 'W' of each cell in the array arr[][] when the array is stored as column major wise.

Address of [I, J]th element in column-major = $B + W[R(J - Lc) + (I - Lr)]$

$\Rightarrow 4440 = 4000 + W[15(8 - 1) + (6 - 1)]$

$\Rightarrow 4440 = 4000 + W[15(7) + 5]$

$\Rightarrow 4440 = 4000 + W[105 + 5]$

$\Rightarrow 4440 = 4000 + W[110]$

$\Rightarrow W[110] = 440$

$\Rightarrow W = 4$

Problems

A matrix ARR[-4…6, 3…8] is stored in the memory with each element requiring 4 bytes of storage. If the base address is 1430, find the address of ARR[3][6] when the matrix is stored in Row Major Wise.

Number of columns, C = 8 – 3 + 1 = 6.

Address of [I, J]th element in row-major = B + W[C(I – Lr) + (J – Lc)]

⇒ Address of ARR[3][6] = 1430 + 4[6(3 – (-4)) + (6 – 3)]

⇒ Address of ARR[3][6] = 1430 + 4[6(3 + 4) + 3]

⇒ Address of ARR[3][6] = 1430 + 4[6(7) + 3]

⇒ Address of ARR[3][6] = 1430 + 4[42 + 3]

⇒ Address of ARR[3][6] = 1430 + 4[45]

⇒ Address of ARR[3][6] = 1430 + 180

⇒ Address of ARR[3][6] = 1610

ASSIGNMENTS

1.Program to take a matrix and calculate the sum of all the matrix elements.

2.Program for the addition of two matrices of order m*n and display the resultant matrix.

3.Program for the multiplication of two matrices of order m*n and n*p and display the resultant matrix.

4.Program to find out the transpose of a given matrix of order m*n.

5.Program to check whether a matrix is symmetric or not.

Array of Pointer

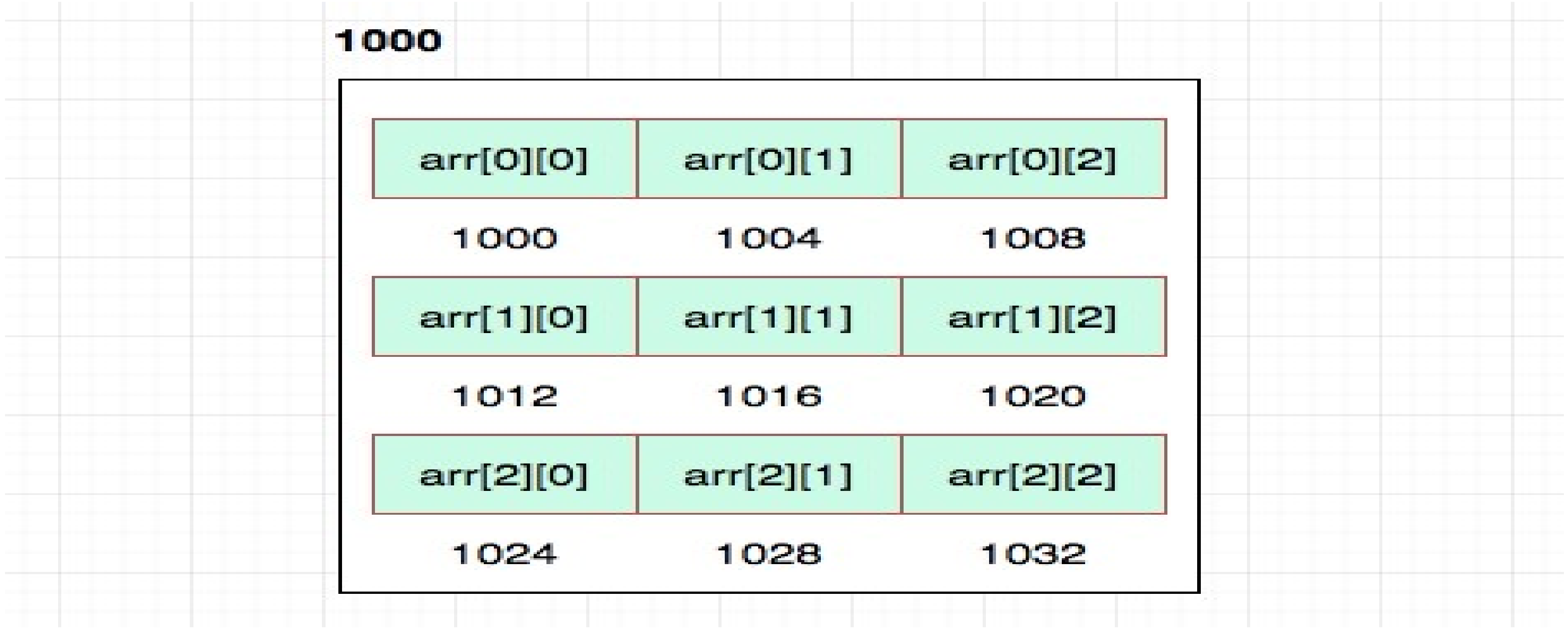
2D array

Let's declare a 3x3 array.

int arr[3][3];

Let's assume the starting address of the above 2D array as 1000.

The Below array's memory address is an arithmetic progression with common difference of 4 as the size of an integer is 4.

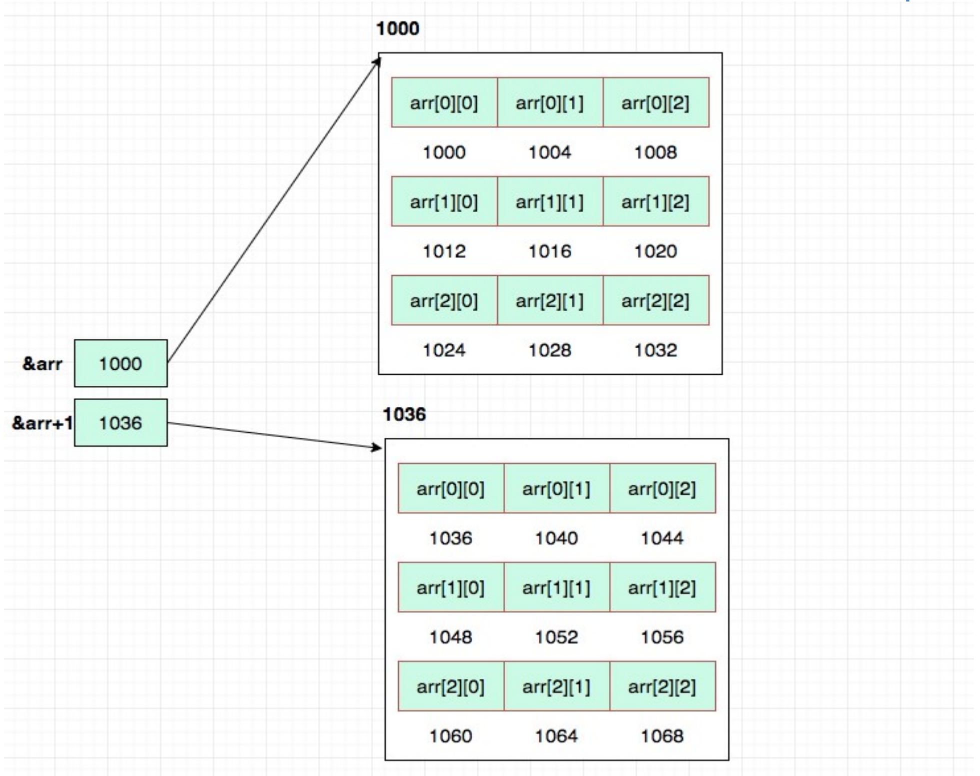


&Arr is a whole 2D array pointer

&arr is a pointer to the entire 2D(3x3) array. i.e. (int*)[3][3]
If we move &arr by 1 position(&arr+1), it will point to the next 2D block(3X3).

In our case, the base address of the 2D array is 1000. So, &arr value will be 1000.
What will be the value of &arr+1?

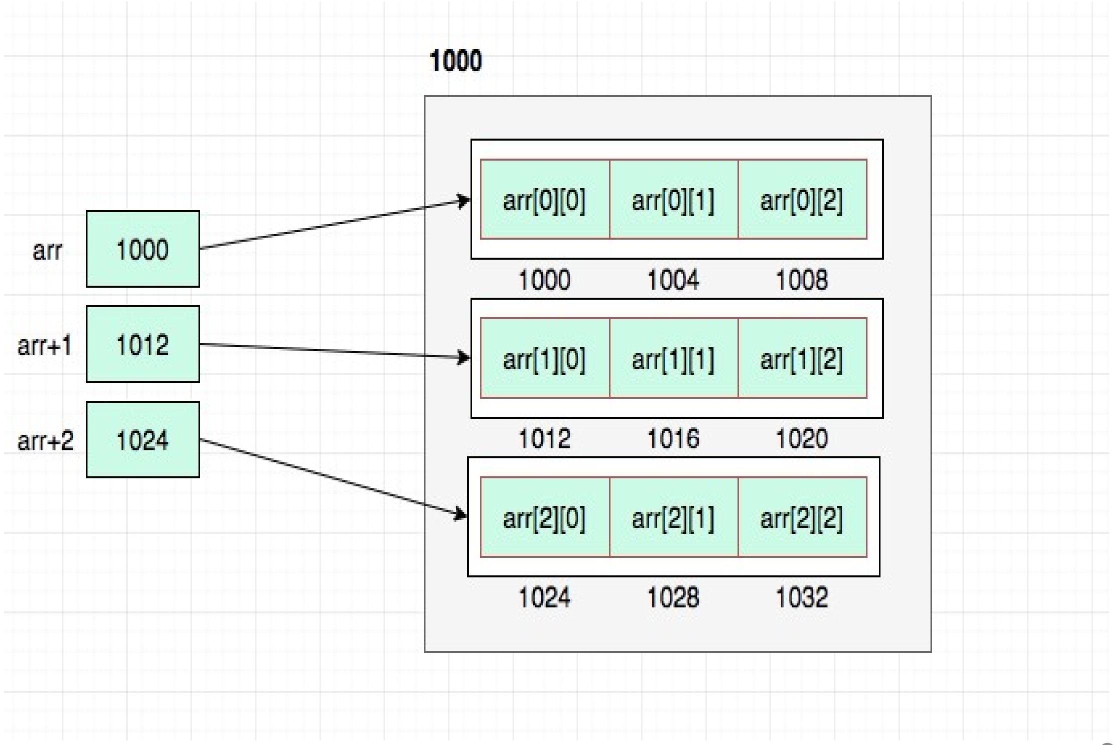
In general, for Row x Col array the formula will be
datatype)
In our case, Row = 3, Col = 3, sizeof(int) = 4.
It is base address + (3 * 3)* 4 => 1000 + 36 => 1036



Arr is a 1D array pointer

arr is a pointer to the first 1D array. i.e. (int*)[3]
If we move arr by 1 position(arr+1), it will point to the next 1D block(3 elements).

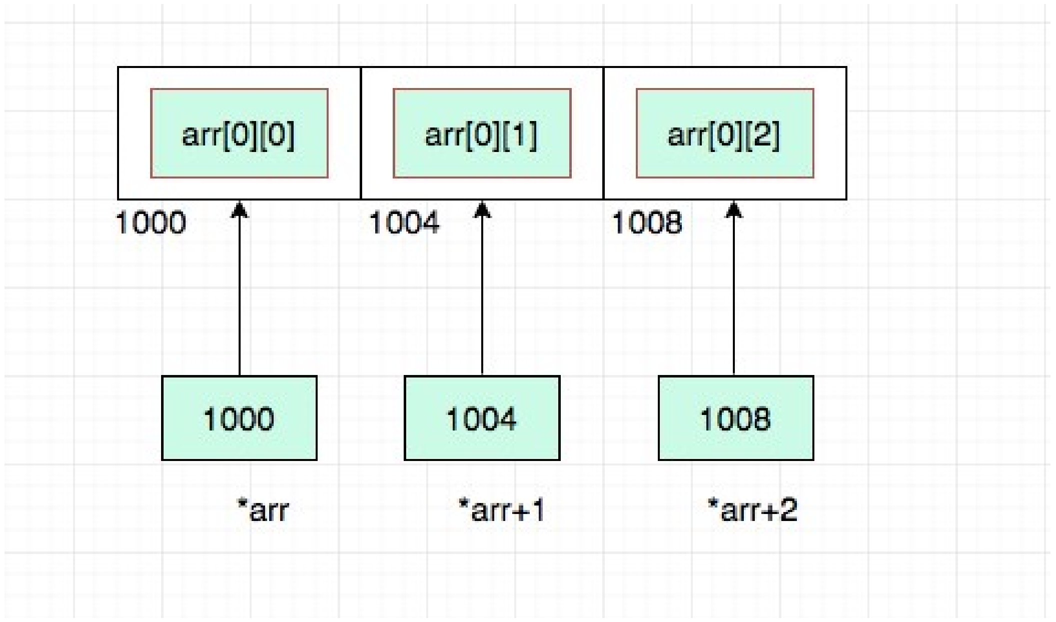
The base address of first 1D array also 1000. So, arr value will be 1000.
What will be the value of arr+1?
It will be base address + Row * sizeof(datatype).
1000 + 3 * 4 =1012



*Arr is a pointer to the first element of the 2D array.

arr is a pointer to the first element of the 2D array. i.e. (int)
If we move *arr by 1 position(*arr+1), it will point to the next element.

The base address of the first element also 1000.So, *arr value will be 1000.
What will be the value of *arr+1?
It will be base address + sizeof(datatype).
1000 + 4=1004



**Arr will be the value of the first element.

Since *arr holds the address of the first element, **arr will give the value stored in the first element.

If we move **arr by 1 position(**arr+1), the value will be incremented by 1.
If the array first element is 10, **arr+1 will be 11.

Summary

1. &arr is a 2D array pointer (int*)[row][col]. So, &arr+1 will point the next 2D block.
2. arr is a 1D array pointer (int*)[row]. So, arr+1 will point the next 1D array in the 2D array.
3. *arr is a single element pointer (int*). So, *arr+1 will point the next element in the array.
4. **arr is the value of the first element. **arr+1 will increment the element value by 1.

Pointer to Array

- Pointer to an array is also known as array pointer.
- Using the pointer the elements of the array are accessed.
- Example:

```
int arr[3] = {30, 40, 50};
int *ptr = arr;
```

- pointer *ptr* that holds address of 0th element of the array.
- Likewise, it can be declared a pointer that can point to whole array rather than just a single element of the array.
- Syntax:

```
data type (*var name)[size of array];
```

- Declaration of the pointer to an array:

```
int (* ptr)[5] = NULL;   // pointer to an array of five numbers
```

- subscript has higher priority than indirection

Pointer to array

Pointer to 1-D array

```
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a;
for (i=0; i<5; i++)
{
 printf("%d,", a[i]);
 printf("%d\n", *(p+i));
}
```

Pointer to 2-D array

```
int main()
{
   int arr1[5][5] = { { 0, 1, 2, 3, 4 },
                      { 2, 3, 4, 5, 6 },
                      { 4, 5, 6, 7, 8 },
                      { 5, 4, 3, 2, 6 },
                      { 2, 5, 4, 3, 1 } };
   int* arr2[5][5];
 for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
         arr2[i][j] = &arr1[i][j];
      }
   }
printf("The values are\n");
   for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
         printf("%d ", *arr2[i][j]);
      }
      printf("\n");
   }
    return 0;
}
```

Array of Pointer

```
// C program to demonstrate the use of array of pointers
#include <stdio.h>

int main()
{
    // declaring some temp variables
    int var1 = 10;
    int var2 = 20;
    int var3 = 30;

// array of pointers to integers
    int* ptr_arr[3] = { &var1, &var2, &var3 };

    // traversing using loop
    for (int i = 0; i < 3; i++)
    {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
    }

    return 0;
}
```

Pointer to Array

Example:
// C program to demonstrate pointer to an array.

```
#include <stdio.h>
int main() {
    // Pointer to an array of five numbers
    int(*ptr)[5];
    int arr[5] = {10, 20, 30, 40, 50};
    int i = 0;
    // Points to the whole array b
    ptr = &arr;
    for (i = 0; i < 5; i++)
        printf("%d\n", *(*ptr + i));
    return 0;
}
```

O/P : **Output**:

```
10
20
30
40
50
```

Pointer to Array

```
// C program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include<stdio.h>
int main() {
    int *p;    // Pointer to an integer
    int (*ptr)[5];  // Pointer to an array of 5 integers
    int arr[5];
     p = arr;    // Points to 0th element of the arr.
    ptr = &arr;    // Points to the whole array arr.
    printf("p = %p, ptr = %p\n", p,  ptr);
    p++;       ptr++;
    printf("p = %p, ptr = %p\n", p,  ptr);
    return 0;
}
```

Output:

```
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64
```

- p: is pointer to 0th element of the array arr
- ptr is a pointer that points to the whole array arr.
- base type of p is int
- base type of ptr is 'an array of 5 integers'.
- pointer arithmetic is performed relative to the base size,
- ptr++, the pointer ptr will be shifted forward by 20 bytes.

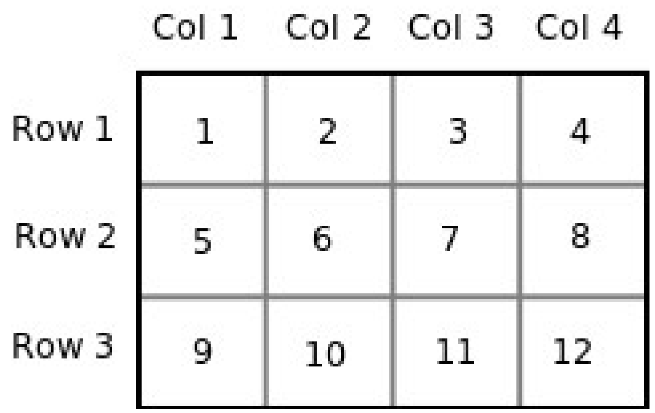
Pointers and 2-D Arrays

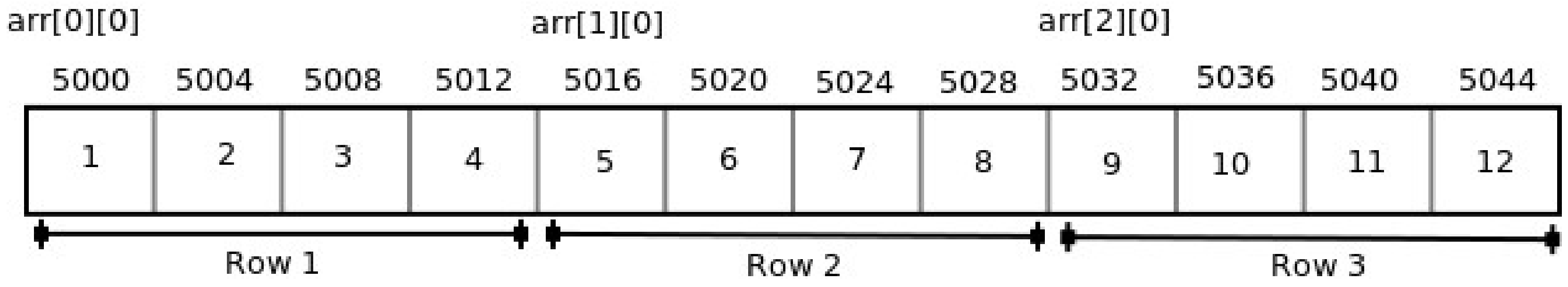
- Access each element by using two subscripts
  - first subscript represents the row number and
  - second subscript represents the column number
- The elements of 2-D array can be accessed with the help of pointer notation also
- Suppose arr is a 2-D array, can access any element arr[i][j] of the array using the pointer expression *(*(arr + i) + j)

Pointers and 2-D Arrays

- int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };

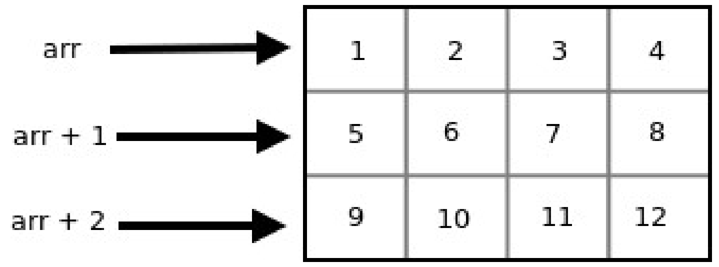
- Memory in a computer is organized linearly
- Not possible to store the 2-D array in rows and columns
- The concept of rows and columns is only theoretical
- Actually, a 2-D array is stored in row-major order i.e rows are placed next to each other





Pointers and 2-D Arrays

- Each row can be considered as a 1-D array
- A two-dimensional array can be considered as an array of one-dimensional arrays
- *arr* is an array of 3 elements where each element is a 1-D array of 4 integers
- Name of an array is a constant pointer that points to 0th 1-D array and contains address 5000
- Since arr is a 'pointer to an array of 4 integers',
  - according to pointer arithmetic the expression arr + 1 will represent the address 5016 and
  - expression arr + 2 will represent address 5032.
- arr points to the 0th 1-D array, arr + 1 points to the 1st 1-D array and arr + 2 points to the 2nd 1-D array

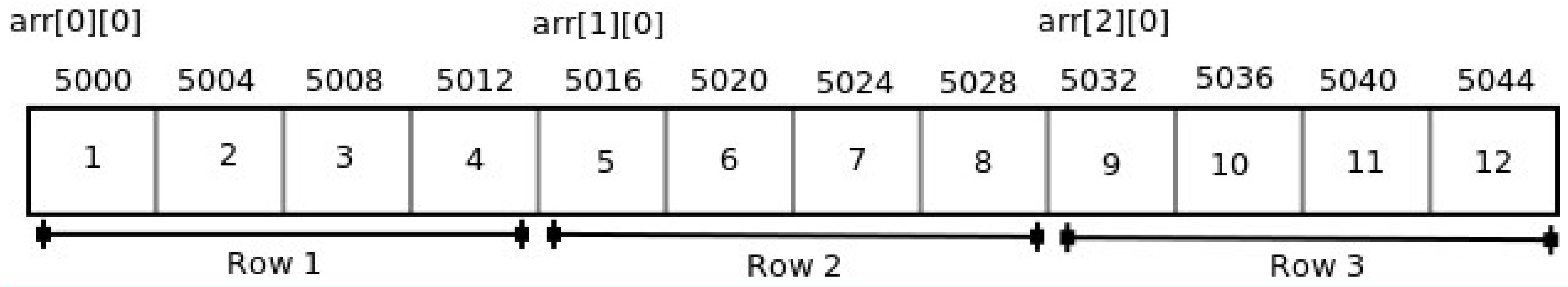


arr	-	Points to 0th element of arr	-	Points to 0th 1-D array	-	5000
arr + 1	-	Points to 1th element of arr	-	Points to 1nd 1-D array	-	5016
arr + 2	-	Points to 2th element of arr	-	Points to 2nd 1-D array	-	5032

Pointers and 2-D Arrays

In general:

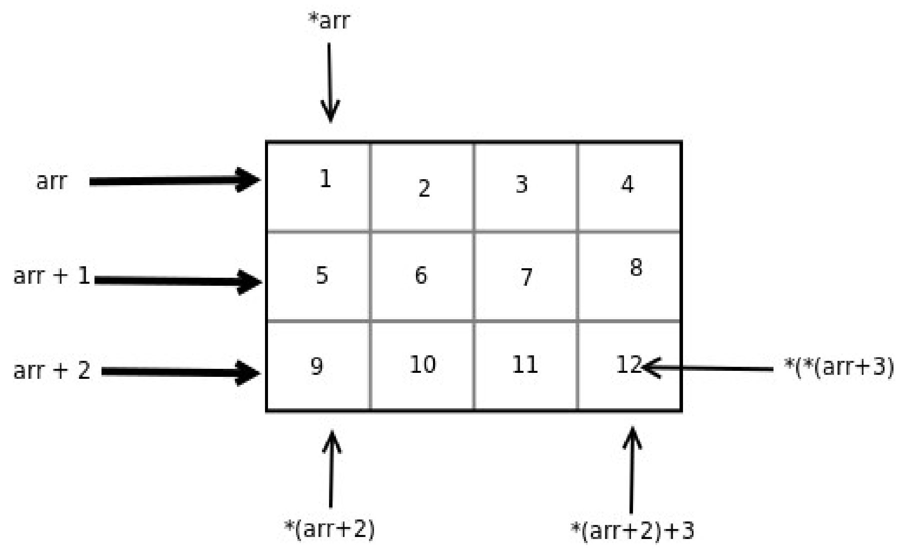
- arr + i points to ith element of arr
- on dereferencing, it will get ith element of arr which is of course a 1-D array
- expression *(arr + i) gives the base address of ith 1-D array
- pointer expression *(arr + i) is equivalent to the subscript expression arr[i]
- So, *(arr + i) which is same as arr[i] gives us the base address of ith 1-D array



*(arr + 0) - arr[0] - Base address of 0th 1-D array - Points to 0th element of 0th 1-D array - 5000
*(arr + 1) - arr[1] - Base address of 1st 1-D array - Points to 0th element of 1st 1-D array - 5016
*(arr + 2) - arr[2] - Base address of 2nd 1-D array - Points to 0th element of 2nd 1-D array - 5032

Pointers and 2-D Arrays

- To access an element of 2-D array, access any jth element of ith 1-D array
- base type of *(arr + i) is *int* and it contains the address of 0th element of ith 1-D array
- get the addresses of subsequent elements in the ith 1-D array by adding integer values to *(arr + i)
- Example:
  - *(arr + i) + 1 will represent the address of 1st element of ith 1-D array and
  - *(arr+i)+2 will represent the address of 2nd element of ith 1-D array
  - *(arr + i) + j will represent the address of jth element of ith 1-D array
- On dereferencing this expression, can get the jth element of the ith 1-D array



Pointers and 2-D Arrays

Print the values and address of elements of a 2-D array

```
#include<stdio.h>
int main() {
  int arr[3][4] = {{ 10, 11, 12, 13},  {20, 21, 22, 23},
                   {30, 31, 32, 33}
              };
  int i, j;
  for (i = 0; i < 3; i++) {
    printf("Address of %dth array = %p %p\n",
              i, arr[i], *(arr + i));
    for (j = 0; j < 4; j++)
      printf("%d %d ", arr[i][j], *(*(arr + i) + j));
    printf("\n");
  }
  return 0;
}
```

Output:

```
Address of 0th array =
0x7ffe50edd580    0x7ffe50edd580
10 10 11 11 12 12 13 13
Address of 1th array =
0x7ffe50edd590    0x7ffe50edd590
20 20 21 21 22 22 23 23
Address of 2th array =
0x7ffe50edd5a0    0x7ffe50edd5a0
30 30 31 31 32 32 33 33
```

Array of pointers

- "Array of pointers" is an array of the pointer variables
- Also known as pointer arrays.
- Syntax:
  - int *var_name[array_size];

- Declaration:
  - int *ptr[3];

Array of pointers

```
#include <stdio.h>
const int SIZE = 3;
int main() {
    int arr[] = { 10, 20, 30 };
    int i, *ptr[SIZE];
    for (i = 0; i < SIZE; i++) {
        ptr[i] = &arr[i];
    }
    for (i = 0; i < SIZE; i++) {
        printf("Value of arr[%d] = %d\n", i, *ptr[i]);
    }
    return 0;
}
```

Output:

```
Value of arr[0] = 10
Value of arr[1] = 20
Value of arr[2] = 30
```

Array of pointers

```
#include <stdio.h>
const int size = 4;
int main() {
    char* names[] = {
        "Amit",
        "Amar",
        "Ankit",
        "Ashish"
    };
    int i = 0;
    for (i = 0; i < size; i++) {
        printf("%s\n", names[i]);
    }
    return 0;
}
```

Output:

```
Amit
Amar
Ankit
Ashish
```

String and Pointer

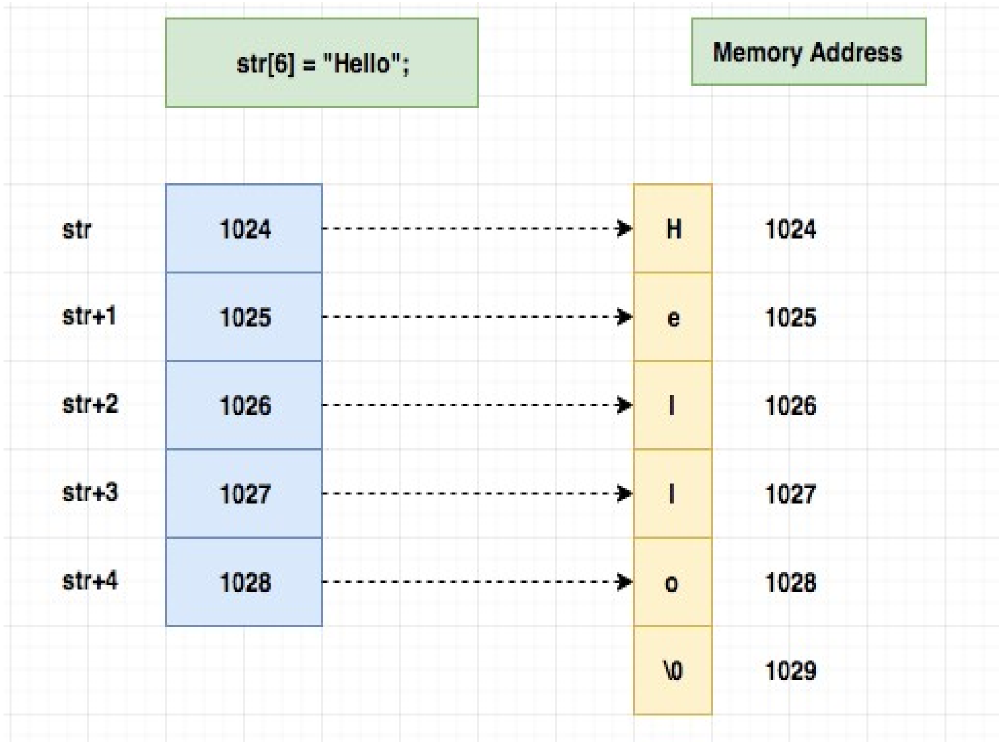
String name == &string[0]
As we all know, the string is a character array which is terminated by the null '\0' character.

```
char str[6]="Hello";
```

Where,
•str is the string name and it is a pointer to the first character in the string. i.e. &str[0].
•Similarly, str+1 holds the address of the second character of the string. i.e. &str[1]
•To store "Hello", we need to allocate 6 bytes of memory.
•5 byte for "Hello"
•1 byte for null '\0' character.

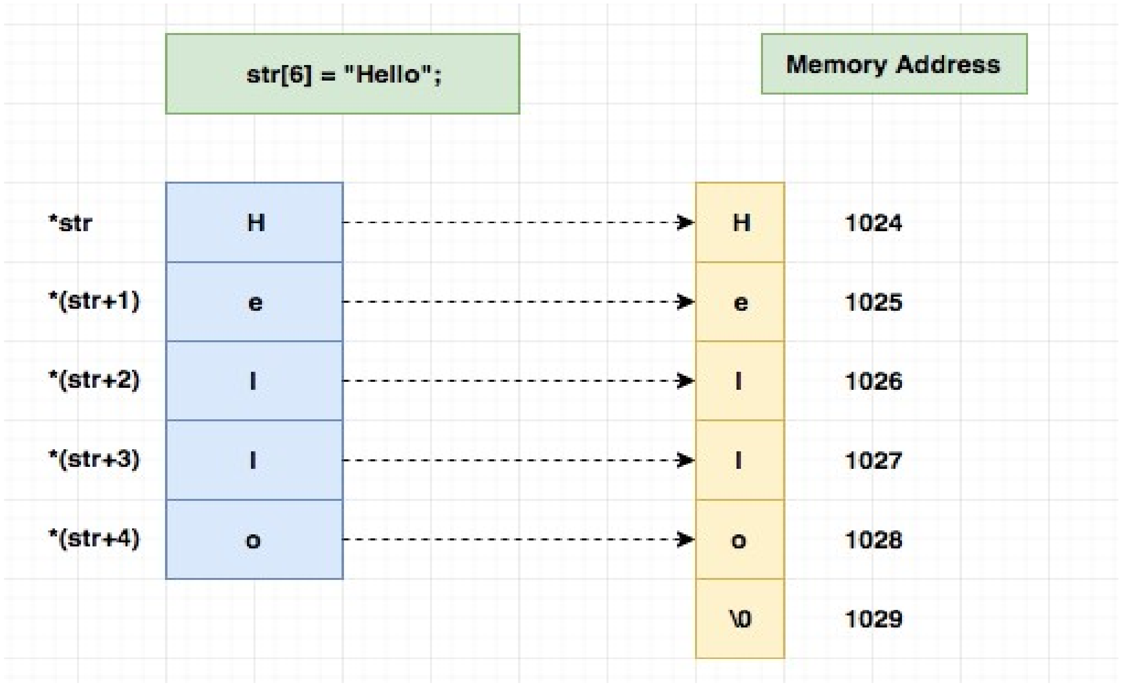
Character Array and Pointer

```
#include<stdio.h>
int main()
 {
char str[6] = "Hello";
 int i; //printing each char address
 for(i = 0; str[i]; i++)
      printf("&str[%d] = %p\n",i,str+i);
return 0;
 }
```



Character Array and Pointer

Str[i] == *(str+i)
str[i] will give the character stored at the string index i.
str[i] is a shortend version of *(str+i).
*(str+i) - value stored at the memory address str+i. (base address + index)



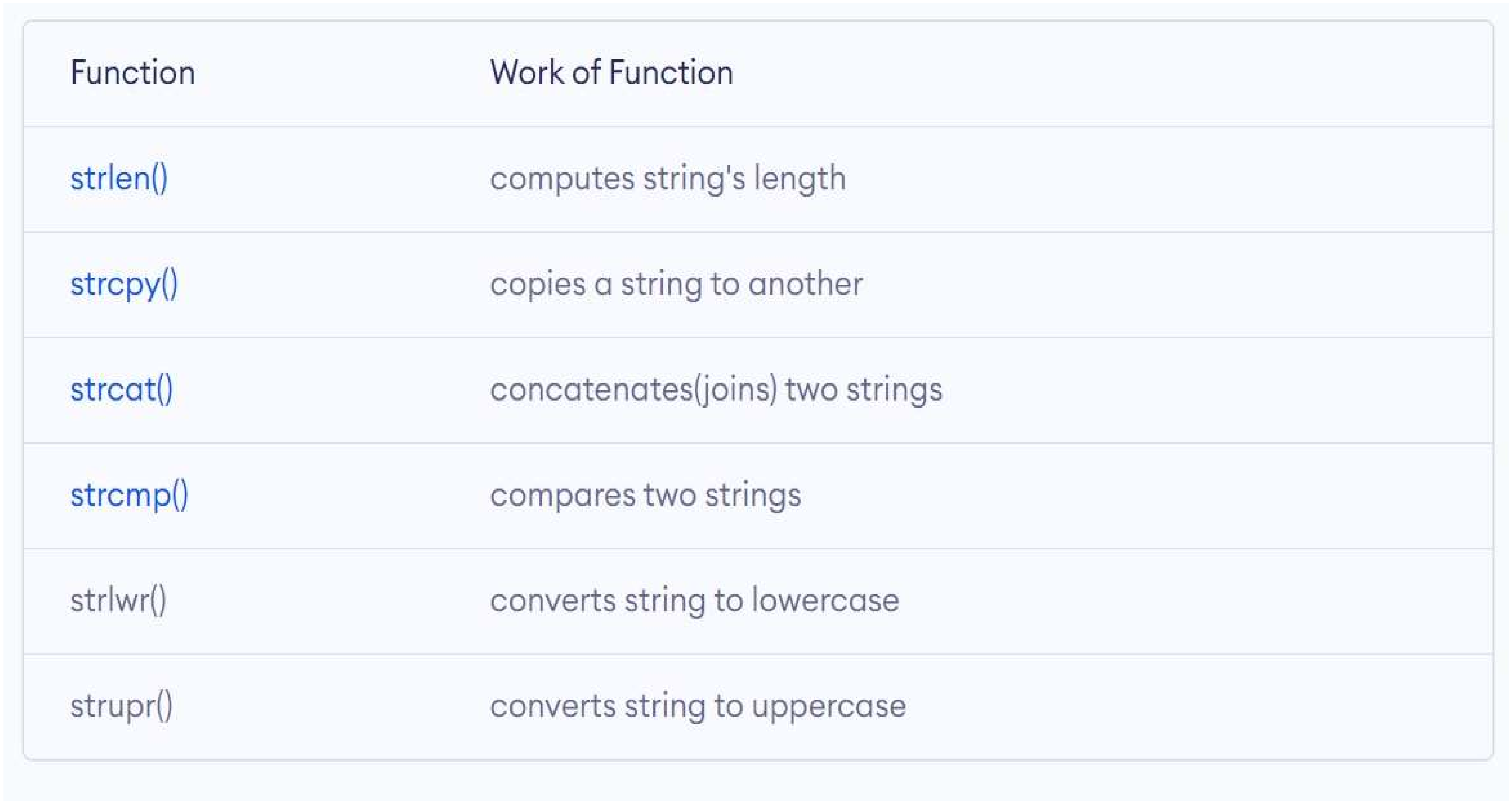
Character Array and Pointer

```
#include<stdio.h>
int main()
 {
                char str[6] = "Hello"; int i;    //printing each char value
                for(i = 0; str[i]; i++)
                                printf("str[%d] = %c\n",i,*(str+i));  //str[i] == *(str+i)
                return 0;
 }
```

Accessing string using pointer

```
#include<stdio.h>
 int main()
 {
          char str[6] = "Hello";
           char *ptr; int i; //string name itself a base address of the
          string ptr = str; //ptr references str
           for(i = 0; ptr[i] != '\0'; i++)
                    printf("&str[%d] = %p\n",i,ptr+i);
           return 0;
 }
```

Accessing string using pointer



Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

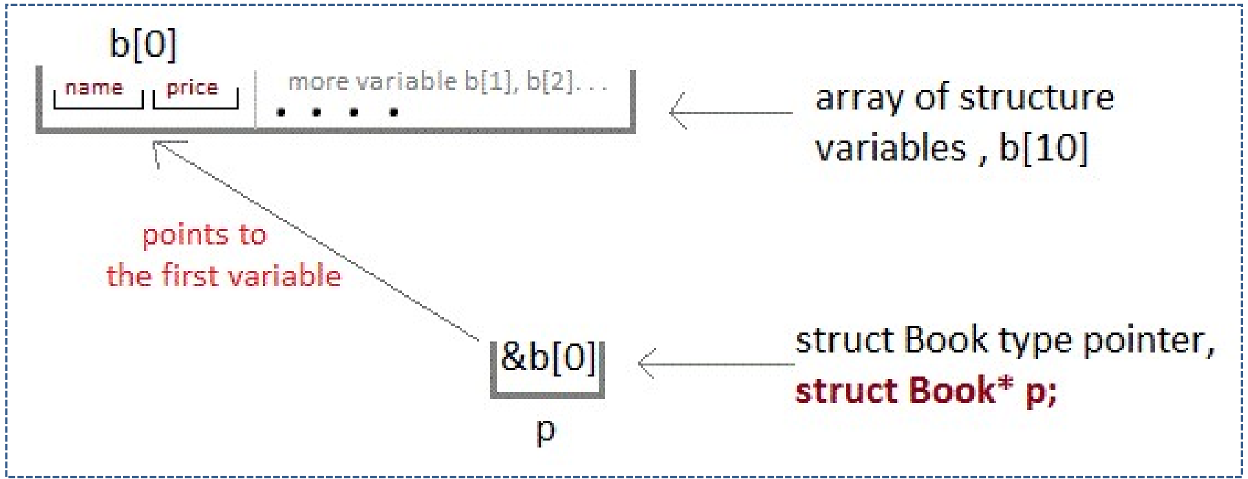
Pointer to Structure

```
struct Book
{
 char name[10];
 int price;
}

int main()
{
 struct Book a;      //Single structure variable
 struct Book* ptr;   //Pointer of Structure type
 ptr = &a;

ptr->name = "Dan Brown";     //Accessing Structure Members
ptr->price = 500;

 struct Book b[10];    //Array of structure variables
 struct Book* p;       //Pointer of Structure type
 p = &b;
}
```


b[0]
name
price
more variable b[1], b[2]. . .
array of structure variables , b[10]
points to the first variable
&b[0]
p
struct Book type pointer,
struct Book* p;

DMA – 1-D and 2-Darrays

- Array in C is static in nature, so its size should be known at compile time and we can't change the size of the array after its declaration.
- Due to this, we may encounter situations where our array doesn't have enough space left for required elements or we allotted more than the required memory leading to memory wastage. To solve this problem, dynamic arrays come into the picture.
- A Dynamic Array is allocated memory at runtime and its size can be changed later in the program.
- We can create a dynamic array in C by using the following methods:
  - Using malloc() Function
  - Using calloc() Function
  - Resizing Array Using realloc() Function

Dynamic Array Using malloc() Function

- The "**malloc**" or "**memory allocation**" method in C is used to dynamically allocate a single large block of memory with the specified size.

- It returns a pointer of type void which can be cast into a pointer of any form.

- It is defined inside **<stdlib.h>** header file.

Syntax:

```
ptr = (cast-type*) malloc(byte-size);
```

Example:

- ptr = (int*) malloc(100 * sizeof(int));

In the above example, we have created a dynamic array of type **int** and size **100 elements.**

Example

```
// C program to create dynamic array using malloc()
      function

#include <stdio.h>
#include <stdlib.h>

int main()
{

   // address of the block created hold by this pointer
   int* ptr;
   int size;

   // Size of the array
   printf("Enter size of elements:");
   scanf("%d", &size);

   //  Memory allocates dynamically using malloc()

ptr = (int*)malloc(size * sizeof(int));

   // Checking for memory allocation
     if (ptr == NULL) {
      printf("Memory not allocated.\n");
    }
    else {

       // Memory allocated
       printf("Memory successfully allocated using "
              "malloc.\n");
        // Get the elements of the array
       for (int j = 0; j < size; ++j) {
          ptr[j] = j + 1;
       }

       // Print the elements of the array
       printf("The elements of the array are: ");
       for (int k = 0; k < size; ++k) {
          printf("%d, ", ptr[k]);
       }
    }

    return 0;
}
```

O/P: Enter size of elements:5 Memory successfully allocated using malloc. The elements of the array are: 1, 2, 3, 4, 5,

Dynamic Array Using calloc() Function

- The “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type and initialized each block with a default value of 0.
- The process of creating a dynamic array using calloc() is similar to the malloc() method. The difference is that calloc() takes arguments instead of one as compared to malloc().
- Here, we provide the size of each element and the number of elements required in the dynamic array. Also, each element in the array is initialized to zero.

Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

Example:

```
ptr = (int*) calloc(5, sizeof(float));
```

In the above example, we have created a dynamic array of type float having five elements.

Example

```
// C program to create dynamic array using calloc()
      function

#include <stdio.h>
#include <stdlib.h>

int main()
{

   // address of the block created hold by this pointer
   int* ptr;
   int size;

   // Size of the array
   printf("Enter size of elements:");
   scanf("%d", &size);

   //  Memory allocates dynamically using calloc()
   ptr = (int*)calloc(size, sizeof(int));
  // Checking for memory allocation
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
   }
     else {

      // Memory allocated
      printf("Memory successfully allocated using "
           "malloc.\n");

      // Get the elements of the array
      for (int j = 0; j < size; ++j) {
         ptr[j] = j + 1;
      }

      // Print the elements of the array
      printf("The elements of the array are: ");
      for (int k = 0; k < size; ++k) {
         printf("%d, ", ptr[k]);
      }
   }

   return 0;
}
```

```
O/P: Enter size of elements:6
Memory successfully allocated using malloc.
 The elements of the array are: 1, 2, 3, 4, 5, 6,
```

Dynamically Resizing Array Using realloc() Function

The **“realloc”** or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory.

Using this function we can create a new array or change the size of an already existing array.

Syntax:

```
ptr = realloc(ptr, newSize);
```

Example

```
// C program to resize dynamic array using realloc()
// function

#include <stdio.h>
#include <stdlib.h>

int main()
{

  // address of the block created hold by this pointer
  int* ptr;
  int size = 5;


  //  Memory allocates dynamically using calloc()
  ptr = (int*)calloc(size, sizeof(int));

  if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
  }
  else {
    printf("Memory successfully allocated using "
        "calloc.\n");
  }

  // inserting elements
  for (int j = 0; j < size; ++j) {
    ptr[j] = j + 1;
  }
```

```
printf("The elements of the array are: ");
for (int k = 0; k < size; ++k) {
   printf("%d, ", ptr[k]);
}

printf("\n");

size = 10;

int *temp = ptr;

//  using realloc
ptr = realloc(ptr, size * sizeof(int));
if (!ptr) {
   printf("Memory Re-allocation failed.");
   ptr = temp;
}
else {
   printf("Memory successfully re-allocated using "
        "realloc.\n");
}
```

Example

```
// inserting new elements
   for (int j = 5; j < size; ++j) {
      ptr[j] = j + 10;
   }

   printf("The new elements of the array are: ");
   for (int k = 0; k < size; ++k) {
      printf("%d, ", ptr[k]);
   }
   return 0;
}
```

Output

```
Memory successfully allocated using calloc.
 The elements of the array are: 1, 2, 3, 4, 5,
Memory successfully re-allocated using realloc.
 The new elements of the array are: 1, 2, 3, 4, 5,
15, 16, 17, 18, 19,
```

Array ADT

An abstract data type (**ADT**) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the **point of view of** a **user of the data**, specifically in terms of **possible values**, **possible operations on data of this type**, and t**he behavior of these operations**. When considering **Array ADT** we are more concerned with the **operations** that can be performed on array.

Basic Operations

- **Traversal –** print all the array elements one by one.
- **Insertion –** add an element at given index.
- **Deletion** – delete an element at given index.
- **Search –** search an element using given index or by value.
- **Sorting** – arranging the elements in some type of order.
- **Merging** – Combining two arrays into a single array
- **Reversing** – Reversing the elements

Default Value Initialization

In C, when an array is initialized with size, then it assigns following defaults values to its elements

- **bool –** false
- **char–** 0
- **float–** 0.0
- **double–** 0.0f
- **int–** 0

Basic Operation cont…

Traverse

Write a program to processing each element in an array (traversing) .

```
#include<stdio.h>
#include<conio.h>
void main()
{
		int a[100],i,n;
		clrscr();
		printf("Enter the number of element
to be insert in the array  :");
scanf("%d",&n);
printf("\nEnter the array element \n");
		for(i=0;i<n;i++)
		{
				scanf("%d",&a[i]);
}
/* traversing block */
for(i=0;i<n;i++)
		{
				a[i]=a[i]+2;
}
printf("After traversing the array  is :\n");
for(i=0;i<n;i++)
{
		printf("%d\t",a[i]);
}
}
```

OUTPUT:

```
Enter the number of element to be insert in the
array : 6

Enter the array element
10
20
30
40
50
60

After traversing the array is:
22		32		42		52		62
```

Basic Operation cont...

Insertion

Write a program to insert an element in an array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[5], n , p, i, ub=4 ;
printf("Enter the elements of the array :\n");
for(i=0;i<5 ;i++)
{
        scanf("%d",&a[i]);
}
printf("\nEnter the inserting element:");
scanf("%d",&n);
printf("\nEnter the position where the element
to be entered :");
scanf("%d",&p);
p--;
while(ub>=p)
{
        a[ub+1]=a[ub];
        ub--;

}
a[p]=n;
printf("After insertion the array is:\n");
for(i=0;i<6;i++)
{
        printf("%d\t",a[i]);
}

}
```

OUTPUT:

```
Enter the elements of the array:
10
11
12
13
14
Enter the inserting element: 20
Enter the position where the element to be
entered: 3

After insertion the array is:
11        20        12        13        14
```

Basic Operation cont…

Deletion

```
/* program to remove the specific elements
from an array in C. */
#include <stdio.h>  #include <conio.h>
 int main ()
{
     int arr[50], pos, i, num;
    printf (" \n Enter the number of elements in
an array: \n ");
    scanf (" %d", &num);
    printf (" \n Enter %d elements in array: \n ",
 num);
    for (i = 0; i < num; i++ )
    {   printf (" arr[%d] = ", i);
        scanf (" %d", &arr[i]);
    }
    // enter the position of the element to be del
eted
    printf( " Define the position of the array elem
ent where you want to delete: \n ");
    scanf (" %d", &pos);
// check whether the deletion is possible or not
    if (pos >= num+1)
    {
        printf (" \n Deletion is not possible in the ar
ray.");     }
    else
    {
// use for loop to delete the element and update t
he index
        for (i = pos - 1; i < num -1; i++)
        {
            arr[i] = arr[i+1]; // assign arr[i+1] to arr[i]
         }
         printf (" \n The resultant array is: \n");
         for (i = 0; i< num - 1; i++)
        {
            printf (" arr[%d] = ", i);
            printf (" %d \n", arr[i]);
        }
    }
    return 0;
}
```

Basic Operation cont...

Binary Search

Write a program to search an element is present or not in a given array using linear search .

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num[10],n,i,item;
clrscr();
printf("Enter the array size :");
scanf("%d",&n);
printf("\nEnter the searching element to be
search :");
scanf("%d",&item);
printf("\nEnter the array element\n");
for(i=0;i<n;i++)
{
    scanf("%d",&num[i]);
}
/* searching block */
for(i=0;i<n; i++)
{
                        if(num[i]== item)
                        {
                                    printf("\n The
element is found at position : %d",i+1);
                                    break;
}
}
if(i==n)
{
                        printf(" \n The element is
not found ");
}
```

OUTPUT:

```
Enter the array size: 7
Enter the searching element to be search: 13
Enter the array element
10  20 30 13 15 19 56
The element is found at position: 4
```

Basic Operation cont…

Sorting

Write a program to sort the array element using selection sort .

```
#include<stdio.h>
#include<conio.h>
void main()
{
		int a[100],i,j,temp,n;
		clrscr();
		printf("Enter the number of element to be insert in the array  :");
scanf("%d",&n);
printf("\nEnter the array element \n");
		for(i=0;i<n;i++)
		{
				scanf("%d",&a[i]);
}
/* sorting  block*/
for(i=0;i<n;i++)
{
		for(j=i+1;j<n-1; j++)
		{
				if(a[i]>a[j])
				{
						temp=a[i];
						a[i]=a[j];
						a[j]=temp;
				}
		}
}
printf("After sorting the array  is :\n");
for(i=0;i<n;i++)
{		printf("%d\t",a[i]);
}
```

OUTPUT:

```
Enter the number of element to be insert in the array  : 5
 Enter the array element
54		2		6		1		8
 After sorting the array is:
2		6		8		54
```

Basic Operation cont…

Insertion

Insert operation is to insert one or more data elements into an array. Based on the requirement, new element can be added at the beginning or end or any given position of array.

Let LA is a filled linear array (unordered) with N elements and K is a positive integer such that K<=N. Below is the algorithm where ITEM is inserted into the Kth position of LA. Procedure – **INSERT(LA, N, K, ITEM)**

```
1. Start
2. Set J=N
3. Set N = N+1 /* Increase the array length by 1*/
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J] /* Move the element downward*/
6. Set J = J-1 /* Decrease counter*/
7. Set LA[K] = ITEM
8. Stop
```

Deletion

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Below is the algorithm to delete an element available at the Kth position of LA. Procedure - **DELETE(LA, N, K)**

```
1. Start
2. Set J = K
3. Repeat step 4 while J < N
4. Set LA[J] = LA[J+1] /* Move the element upward*/
5. Set N = N-1 /* Reduce the array length by 1 */
6. Stop
```

Basic Operation cont…

Search

You can perform a search for array element based on its value or position.

Consider LA is a linear array with N elements. Below is the algorithm to find an element with a value of ITEM using sequential search. Procedure - **SEARCH(LA, N, ITEM)**

```
1. Start
2. Set J=1 and LOC = 0
3. Repeat steps 4 and 5 while J < N
4. IF (LA[J] = ITEM) THEN LOC = J AND GOTO STEP 6
5. Set J = J +1
6. IF (LOC > 0) PRINT J, ITEM ELSE PRINT 'Item not found'
7. Stop
```

Updation

Update operation refers to updating an existing element from the array at a given position.

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Below is the algorithm to update an ITEM available at the Kth position of LA. Procedure - **UPDATE(LA, N, K, ITEM)**

```
1. Start
2. Set LA[K] = ITEM
3. Stop
```

Basic Operation cont…

Traversa

Traversal operation refers to printing the contents of each element or to count the number of elements with a given property

Consider LA is a linear array with N elements. Below is the algorithm to print each element. Procedure - **TRAVERSE(LA, N)**

```
1. Start
2. Set J=1
3. Repeat steps 4 and 5 while J ≤ N
4. PRINT LA[J]
5. Set J = J +1
6. Stop
```

Sortin

Sorting operation refers to arranging the elements either in ascending or descending way.

Consider LA is a linear array with N elements. Below is the **Bubble Sort algorithm** to sort the elements in ascending order. Procedure - **SORT(LA, N)**

```
1. Start
2. Set I = 1
3. Set J = 1
4. Repeat steps 5 to 11 while I ≤ N
5. J = 1
6. Repeat steps 7 to 10 while J ≤  N - I
7. IF LA[I] is > LA[J] THEN
8. Set TEMP = LA[I]; LA[I] = LA[J]; LA[J] = TEMP;
9. END IF
10. Set J = J+1
11. Set I = I+1
12. Stop
```

Basic Operation cont…

Merging - Assignment 1

Merging refers to combining two sorted arrays into one sorted array. It involves 2 steps –

- Sorting the arrays that are to be merged
- Adding the sorted elements of both arrays to a new array in sorted order

LA1 is a linear array with N elements, LA2 is a liner array with M elements and LA3 is a liner array with M+N elements. Write the algorithm to sort LA1 & LA2 and merge LA1 & LA2 into LA3 & sort LA3 and print each element of LA3

```
1. Start
/* Assignment1 */
2. Stop
```

Reversing - Assignment 2

Reversing refers to reversing the elements in the array by swapping the elements. Swapping should be done only half times of the array size

Consider LA is a linear array with N elements. Write the algorithm to reverse the elements and print each element of LA

```
1. Start
/* Assignment 2 */
2. Stop
```

Assignments

Assignment 3

LA is a linear array with N elements. Write the algorithm to finds the largest number and counts the occurrence of the largest number

1. Start
/* Assignment 3 steps */
2. Stop

Assignment 4

LA is a linear array with N elements. Write the algorithm to copy the elements from LA to a new array LB

1. Start
/* Assignment 4 steps*/
2. Stop

Assignment 5

LA is a linear array with N elements. Write the algorithm to transpose the array

1. Start
/* Assignment 5 steps */
2. Stop

Assignment 6

LA is a linear **sorted** array with N elements. Write the algorithm to insert ITEM to the array

1. Start
/* Assignment 6 steps */
2. Stop

THANK YOU!