Data Structure: Array

Alok Kumar Jagadev

Array

- Collection of similar type 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]

Need of using Array

- Require to store the large number of data of similar type
- To store such amount of data, large number of variables need to be defined
- It would be very difficult to remember 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 into 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:

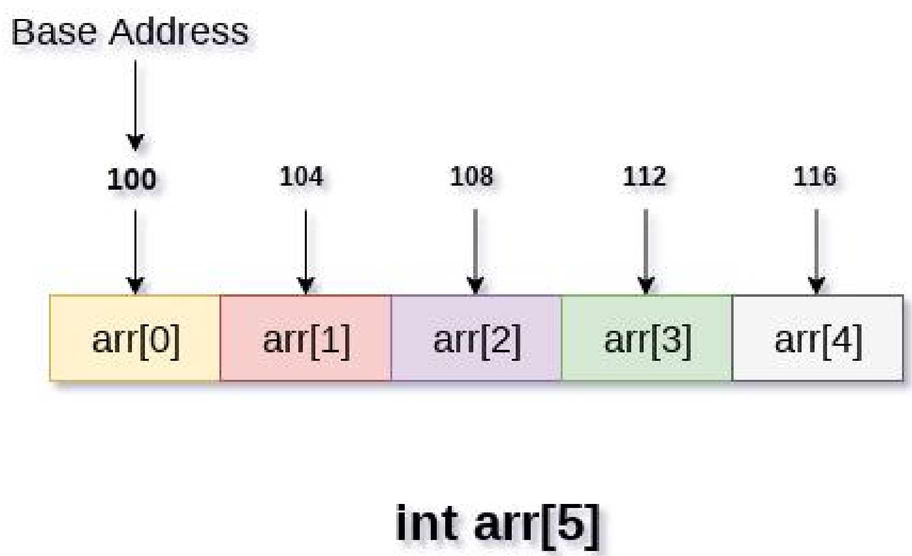
```
#include <stdio.h>
int main ()  {
    int marks[6] = {56, 78, 88, 76, 56, 89};
    int i;
    float avg;
    for (i=0; i<6; i++ ) {
        avg = avg + marks[i];
    }
    avg = avg/6.0;
    printf("%f", avg);
    return 0;
}
```

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



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)

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

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

  $= 1001 + 18$

  $= 1019$

Program

Function to reverse an array

```
void rvereseArray(int arr[], int start, int end)  {
    int temp;
    while (start < end)      {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
```

Input : arr[] = {4, 5, 1, 2}

Output : arr[] = {2, 1, 5, 4}

Program

Move all zeroes to the end of array

```
void moveZerosToEnd(int arr[], int n) {
    // Count of non-zero elements
    int count = 0;
    for (int i = 0; i < n; i++)
        if (arr[i] != 0)
            swap(arr[count++], arr[i]);
}
```

Input : arr[] = {1, 2, 0, 0, 0, 3, 6}

Output : 1 2 3 6 0 0 0

```
void pushZerosToEnd(int arr[], int n) {
    // Count of non-zero elements
    int count = 0;
    for (int i = 0; i < n; i++)
        if (arr[i] != 0)
            arr[count++] = arr[i];
    while (count < n)
        arr[count++] = 0;
}
```

Input : arr[] = {1, 2, 0, 0, 0, 3, 6}

Output : arr[] = {1, 2, 3, 6, 0, 0, 0}

Program

Rearrange array such that even index elements are smaller and odd index elements are greater

```
void rearrange(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        if (i % 2 == 0 && arr[i] > arr[i + 1])
            swap(arr[i], arr[i + 1]);
        if (i % 2 != 0 && arr[i] < arr[i + 1])
            swap(arr[i], arr[i + 1]);
    }
}
```

Input :arr[] = {6, 4, 2, 1, 8, 3}

Output :arr[] = {4, 6, 1, 8, 2, 3}

Program

Given an array of integers, update every element with multiplication of previous and next elements with following exceptions.

a) First element is replaced by multiplication of first and second.

b) Last element is replaced by multiplication of last and second last.

```
void modify(int arr[], int n) {
    if (n <= 1) return;
    int prev = arr[0];
    arr[0] = arr[0] * arr[1];
    for (int i=1; i<n-1; i++) {
        int curr = arr[i];
        arr[i] = prev * arr[i+1];
        prev = curr;
    }
    arr[n-1] = prev * arr[n-1];
}
```

Example:

Input: arr[] = {2, 3, 4, 5, 6}

Output: arr[] = {6, 8, 15, 24, 30}

arr[] = {2*3, 2*4, 3*5, 4*6, 5*6}

Program

Input: arr[] = {10, 4, 3, 50, 23, 90}
Output: 90, 50, 23

Find the three largest elements in an array

```
void print3largest(int arr[], int size) {
    int i, first, second, third;
    if (size < 3) {
        printf(" Invalid Input ");
        return;
    }
    third = first = second = INT_MIN;
    for (i = 0; i < size ; i ++) {
        if (arr[i] > first)   {
            third = second;
            second = first;
            first = arr[i];
        }
else if (arr[i] > second) {
            third = second;
            second = arr[i];
        }
        else if (arr[i] > third)
            third = arr[i];
    }
    printf("Three largest elements are %d
%d   %d\n", first, second, third);
}
```

Program

Find minimum difference between any two elements in a given array

```
int findMinDiff(int arr[], int n) {
   int diff = INT_MAX;
   for (int i=0; i<n-1; i++)
      for (int j=i+1; j<n; j++)
          if (abs(arr[i] - arr[j]) < diff)
                diff = abs(arr[i] - arr[j]);
   printf("%d", diff);
}
```

Example:

Input : {1, 5, 3, 19, 18, 25};

Output : 1

Minimum difference is between 18 and 19

Input : {30, 5, 20, 9};

Output : 4

Minimum difference is between 5 and 9

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

Memory layout of multi-dimensional arrays

- 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 the faster

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

Calculate Memory Addresses in 2D-Arrays

- The 2-dimensional arrays are stored as 1-dimensional arrays in the computer's memory.
- There are two ways to achieve this:
  - Row-major Implementation
  - Column-major Implementation

Calculate Memory Addresses in 2D-Arrays

- The 2-dimensional arrays are stored as 1-dimensional arrays in the computer's memory.
- There are two ways to achieve this:
  - Row-major Implementation
  - Column-major Implementation

Calculate Memory Addresses in 2D-Arrays

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

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

- Note that:
  - B is the base address (address of the first block in the array).
  - W is the width in bytes (size in bytes for each element in the array).
  - Lr is the index of the first row.
  - Lc is the index of the first column.
  - R is the total number of rows.
  - C is the total number of columns.

Problem

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 = $B + W[R(J - Lc) + (I - Lr)]$

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

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

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

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

$\Rightarrow W[110] = 440$

$\Rightarrow W = 4$

Problem

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 = $B + W[C(I - Lr) + (J - Lc)]$

$\Rightarrow$ Address of ARR[3][6] = $1430 + 4[6(3 - (-4)) + (6 - 3)]$

$\Rightarrow$ Address of ARR[3][6] = $1430 + 4[6(3 + 4) + 3]$

$\Rightarrow$ Address of ARR[3][6] = $1430 + 4[6(7) + 3]$

$\Rightarrow$ Address of ARR[3][6] = $1430 + 4[42 + 3]$

$\Rightarrow$ Address of ARR[3][6] = $1430 + 4[45]$

$\Rightarrow$ Address of ARR[3][6] = $1430 + 180$

$\Rightarrow$ Address of ARR[3][6] = 1610

Problem

A matrix A[m][m] is stored in the memory with each element requiring 4 bytes of storage. If the base address at A[1][1] is 1500 and the address of A[4][5] is 1608, determine the order of the matrix when it is stored in Column Major Wise.

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

$\Rightarrow 1608 = 1500 + 4[m(5 - 1) + (4 - 1)]$

$\Rightarrow 1608 = 1500 + 4[m(4) + 3]$

$\Rightarrow 1608 = 1500 + 16m + 12$

$\Rightarrow 1608 = 1512 + 16m$

$\Rightarrow 16m = 96$

$\Rightarrow m = 6$

Problem

A matrix P[15][10] is stored with each element requiring 8 bytes of storage. If the base address at P[0][0] is 1400, determine the address at P[10][7] when the matrix is stored in Row Major Wise.

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

⇒ Address at P[10][7] = 1400 + 8[10(10 – 0) + (7 – 0)]

⇒ Address at P[10][7] = 1400 + 8[10(10) + 7]

⇒ Address at P[10][7] = 1400 + 8[100 + 7]

⇒ Address at P[10][7] = 1400 + 8[107]

⇒ Address at P[10][7] = 1400 + 856

⇒ Address at P[10][7] = 2256

Problem

A matrix A[m][n] is stored with each element requiring 4 bytes of storage. If the base address at A[1][1] is 1500 and the address at A[4][5] is 1608, determine the number of rows of the matrix when the matrix is stored in Column Major Wise.

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

$\Rightarrow$ 1608 = 1500 + 4[R(5 – 1) + (4 – 1)]

$\Rightarrow$ 1608 = 1500 + 4[4R + 3]

$\Rightarrow$ 1608 = 1500 + 16R + 12

$\Rightarrow$ 1608 = 1512 + 16R

$\Rightarrow$ 16R = 96

$\Rightarrow$ R = 6

Problem

The array D[-2…10][3…8] contains double type elements. If the base address is 4110, find the address of D[4][5], when the array is stored in Column Major Wise.

Number of rows, $R = 10 – (-2) + 1 = 13$.

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

$\Rightarrow$ Address of $D[4][5] = 4110 + 8[13(5 – 3) + (4 – (-2))]$

$\Rightarrow$ Address of $D[4][5] = 4110 + 8[13(2) + (4 + 2)]$

$\Rightarrow$ Address of $D[4][5] = 4110 + 8[26 + 6]$

$\Rightarrow$ Address of $D[4][5] = 4110 + 8[32]$

$\Rightarrow$ Address of $D[4][5] = 4110 + 256$

$\Rightarrow$ Address of $D[4][5] = 4366$

Problem

An array AR[-4 … 6, -2 … 12], stores elements in Row Major Wise, with the address AR[2][3] as 4142. If each element requires 2 bytes of storage, find the Base address.

Number of columns, $C = 12 - (-2) + 1 = 12 + 2 + 1 = 15$.

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

$\Rightarrow 4142 = B + 2[15(2 - (-4)) + (3 - (-2))]$

$\Rightarrow 4142 = B + 2[15(2 + 4) + (3 + 2)]$

$\Rightarrow 4142 = B + 2[15(6) + 5]$

$\Rightarrow 4142 = B + 2[90 + 5]$

$\Rightarrow 4142 = B + 2[95]$

$\Rightarrow 4142 = B + 190$

$\Rightarrow B = 3952$

Problem

A square matrix M[][] of size 10 is stored in the memory with each element requiring 4 bytes of storage. If the base address at M[0][0] is 1840, determine the address at M[4][8] when the matrix is stored in Row Major Wise.

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

⇒ Address at M[4][8] = 1840 + 4[10(4 – 0) + (8 – 0)]

⇒ Address at M[4][8] = 1840 + 4[10(4) + 8]

⇒ Address at M[4][8] = 1840 + 4[40 + 8]

⇒ Address at M[4][8] = 1840 + 4[48]

⇒ Address at M[4][8] = 1840 + 192

⇒ Address at M[4][8] = 2032

Problem

A matrix B[10][7] is stored in the memory with each element requiring 2 bytes of storage. If the base address at B[x][1] is 1012 and the address at B[7][3] is 1060, determine the value 'x' where the matrix is stored in Column Major Wise.

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

$\Rightarrow$ 1060 = 1012 + 2[10(3 – 1) + (7 – x)]

$\Rightarrow$ 1060 = 1012 + 2[10(2) + 7 – x]

$\Rightarrow$ 1060 = 1012 + 2[20 + 7 – x]

$\Rightarrow$ 1060 = 1012 + 2[27 – x]

$\Rightarrow$ 1060 = 1012 + 54 – 2x

$\Rightarrow$ 1060 = 1066 – 2x

$\Rightarrow$ -2x = -6

$\Rightarrow$ x = 3

Problem

A square matrix A [m × m] is stored in the memory with each element requiring 2 bytes of storage. If the base address at A[1][1] is 1098 and the address at A[4][5] is 1144, determine the order of the matrix A[m × m] when the matrix is stored in Column Major Wise.

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

$\Rightarrow 1144 = 1098 + 2[m(5 - 1) + (4 - 1)]$

$\Rightarrow 1144 = 1098 + 2[m(4) + 3]$

$\Rightarrow 1144 = 1098 + 8m + 6$

$\Rightarrow 1144 = 1104 + 8m$

$\Rightarrow 8m = 40$

$\Rightarrow m = 5$

Problem

A character array B[7][6] has a base address 1046 at 0, 0. Calculate the address at B[2][3] if the array is stored in Column Major Wise. Each character requires 2 bytes of storage.

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

⇒ Address at B[2][3] = 1046 + 2[7(3 – 0) + (2 – 0)]

⇒ Address at B[2][3] = 1046 + 2[7(3) + 2]

⇒ Address at B[2][3] = 1046 + 2[21 + 2]

⇒ Address at B[2][3] = 1046 + 2[23]

⇒ Address at B[2][3] = 1046 + 46

⇒ Address at B[2][3] = 1092

Problem

Each element of an array A[20][10] requires 2 bytes of storage. If the address of A[6][8] is 4000, find the base address at A[0][0] when the array is stored in Row Major Wise.

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

$\Rightarrow$ 4000 = B + 2[10(6 – 0) + (8 – 0)]

$\Rightarrow$ 4000 = B + 2[10(6) + 8]

$\Rightarrow$ 4000 = B + 2[60 + 8]

$\Rightarrow$ 4000 = B + 2[68]

$\Rightarrow$ 4000 = B + 136

$\Rightarrow$ B = 3864

Problem

A two-dimensional array defined as X[3…6, -2…2] requires 2 bytes of storage space for each element. If the array is stored in Row Major Wise order, determine the address of X[5][1], given the base address as 1200.

Number of columns, $C = 2 - (-2) + 1 = 5$.

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

$\Rightarrow$ Address of $X[5][1] = 1200 + 2[5(5 - 3) + (1 - (-2))]$

$\Rightarrow$ Address of $X[5][1] = 1200 + 2[5(2) + (3)]$

$\Rightarrow$ Address of $X[5][1] = 1200 + 2[13]$

$\Rightarrow$ Address of $X[5][1] = 1200 + 26$

$\Rightarrow$ Address of $X[5][1] = 1226$

Program

Interchange any two Rows & Columns in the given Matrix

```
void interchangerow(mat[][n], col)  {
    scanf("%d %d", &r1, &r2);
    for (i = 0; i < col; ++i) {
        /*  first row has index is 0 */
        temp = mat[r1 - 1][i];
        mat[r1 - 1][i] = mat[r2 - 1][i];
        mat[r2 - 1][i] = temp;
    }
}
```

```
void interchangecol(mat[][n], row)  {
    scanf("%d %d", &c1, &c2);
    for (i = 0; i < row; ++i) {
        /*  first column index is 0 */
        temp = mat[i][c1 - 1];
        mat[i][c1 - 1] = mat[i][c2 - 1];
        mat[i][c2 - 1] = temp;
    }
}
```

Program

Sort Rows of the Matrix in Ascending & Columns in Descending Order

Arranging rows in ascending order

```
for (i = 0; i < m; ++i)
        for (j = 0; j < n; ++j)
            for (k =(j + 1); k < n; ++k)
                if (mat[i][j] > mat[i][k])   {
                    tmp = mat[i][j];
                    mat[i][j] = mat[i][k];
                    mat[i][k] = tmp;
                }
```

Arranging the columns in descending order

```
for (j = 0; j < n; ++j)
    for (i = 0; i < m; ++i)
        for (k = i + 1; k < m; ++k)
            if (mat[i][j] < mat[k][j])  {
                tmp = mat[i][j];
                mat[i][j] = mat[k][j];
                mat[k][j] = tmp;
            }
```

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 0^{th} 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

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;
}
```

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.

Pointer to Array

```
// C program to illustrate sizes of
// pointer of array
#include<stdio.h>
int main() {
    int arr[] = { 30, 50, 60, 70, 90 };
    int *p = arr;
    int (*ptr)[5] = &arr;
    printf("p = %p, ptr = %p\n", p, ptr);
    printf("*p = %d, *ptr = %p\n", *p, *ptr);
     printf("sizeof(p) = %lu, sizeof(*p) = %lu\n",
                      sizeof(p), sizeof(*p));
    printf("sizeof(ptr) = %lu, sizeof(*ptr) = %lu\n",
                    sizeof(ptr), sizeof(*ptr));
    return 0;
}
```

Output:

```
p = 0x7ffde1ee5010, ptr =
0x7ffde1ee5010
*p = 30, *ptr = 0x7ffde1ee5010
sizeof(p) = 8, sizeof(*p) = 4
sizeof(ptr) = 8, sizeof(*ptr) = 20
```

Pointers and 2-D Arrays

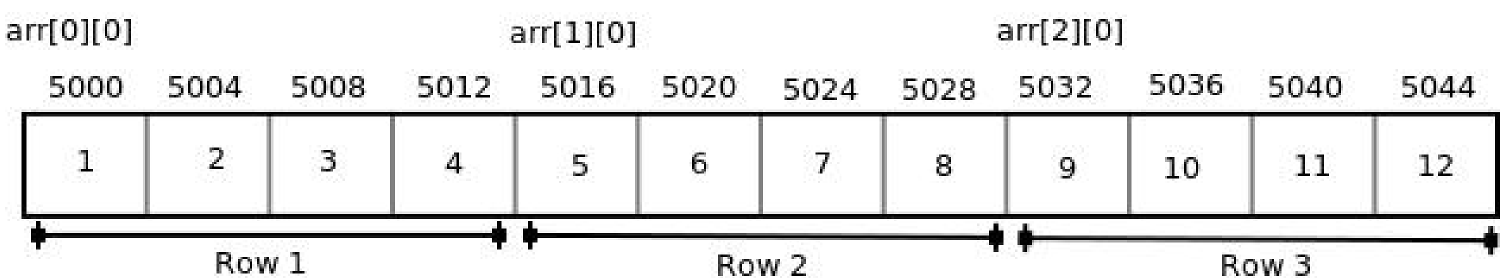
- Access each element by using two subscripts
  - first subscript represents the row number and
  - second subscript represents the column number
- The elements of 2-D array can be accessed with the help of pointer notation also
- Suppose arr is a 2-D array, can access any element arr[i][j] of the array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

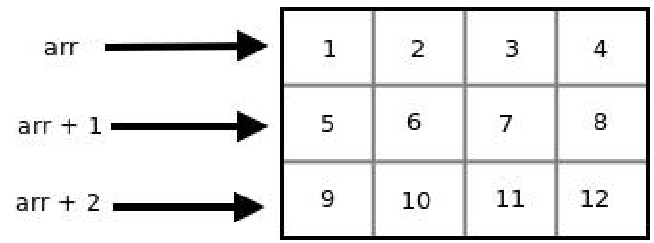
	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

- 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 0^{th} 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 0^{th} 1-D array, arr + 1 points to the 1^{st} 1-D array and arr + 2 points to the 2^{nd} 1-D array

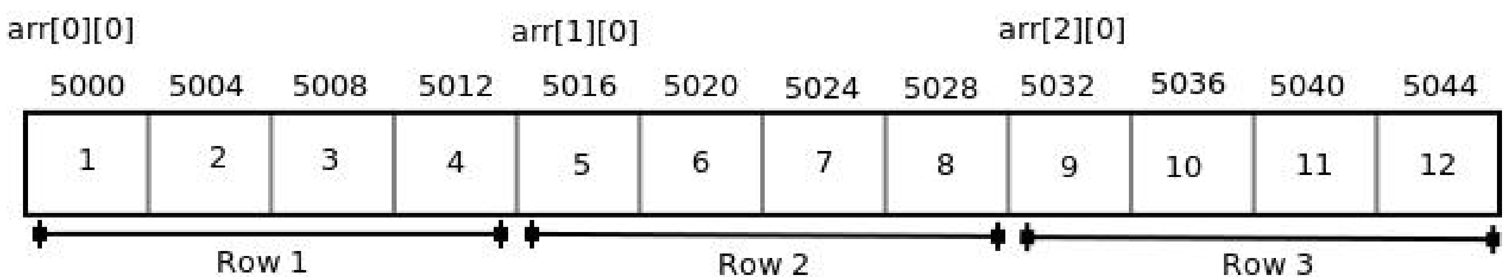


arr	-	**Points to 0^{th} element of arr**	-	**Points to 0^{th} 1-D array**	-	**5000**
arr + 1	-	**Points to 1^{th} element of arr**	-	**Points to 1^{nd} 1-D array**	-	**5016**
arr + 2	-	**Points to 2^{th} element of arr**	-	**Points to 2^{nd} 1-D array**	-	**5032**

Pointers and 2-D Arrays

In general:

- arr + i points to i^{th} element of arr
- on dereferencing, it will get i^{th} element of arr which is of course a 1-D array
- expression *(arr + i) gives the base address of i^{th} 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 i^{th} 1-D array



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

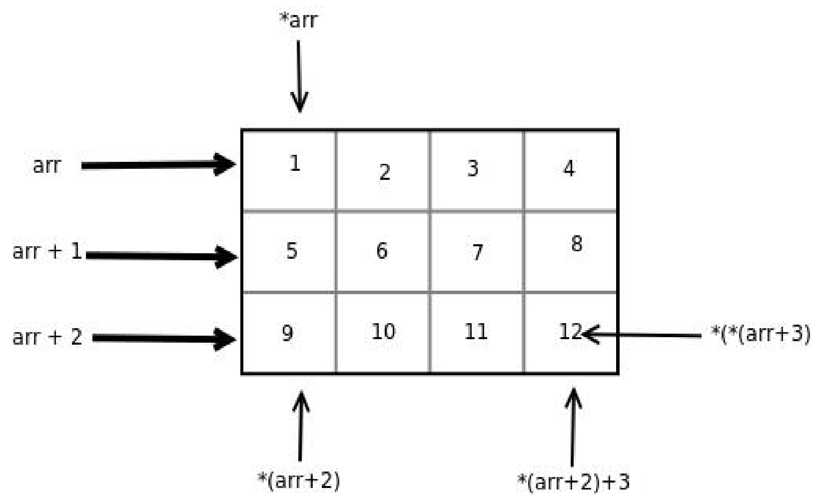
Pointers and 2-D Arrays

In general:

- *(arr + i) - arr[i] - Base address of ith 1-D array -> Points to 0th element of ith 1-D array
- Both the expressions (arr + i) and *(arr + i) are pointers, but their base type are different
  - base type of (arr + i) is ‘an array of 4 elements’
  - while the base type of *(arr + i) or arr[i] is *int*

Pointers and 2-D Arrays

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

Pointer to Structure

- To access members of a structure using pointers
  - use the -> operator.

```
#include <stdio.h>
struct person  {
  int age;
  float wt;
};
```

```
int main()  {
    struct person *pPtr, p1;
    pPtr = &p1;
    printf("Enter age: ");
    scanf("%d", &pPtr->age);
    printf("Enter weight: ");
    scanf("%f", &pPtr->wt);
    printf("Displaying...\n");
    printf("Age: %d\n", pPtr->age);
    printf("Weight: %f", pPtr->wt);
    return 0;
}
```