2D Arrays

- 2 dimensional array is a collection of rows and columns, which is declared as follows

Syntax:

datatype array_name[rows][columns]

How to calculate the total memory?

Number of elements = rows * columns
Total bytes = number of elements * sizeof(datatype)
Or
Total bytes = rows * columns * sizeof(datatype)

E.g.,
int arr[2][3] = {1,2,3,4,5,6};
Or
int arr[2][3] = { {1,2,3}, {4,5,6}};

Total bytes = 2 * 3 * sizeof(int)
= 2 * 3 * 4
= 24 bytes

Memory layout:

[0][0] R0-C0	[0][1] R0-C1	[0][2] R0-C2	[1][0] R1-C0	[1][1] R1-C1	[1][2] R1-C2
1	2	3	4	5	6
1000	1004	1008	1012	1016	1020

How to read and print the values of a 2D array?

→ 2D array elements will be accessed by using array_name[row][column]

E.g., arr[0][0] → 1   arr[1][0] → 4
arr[0][1] → 2   arr[1][1] → 5
arr[0][2] → 3   arr[1][2] → 6

Printing array elements using loop:

→ Because of 2 dimension 2d array elements will be fetched by using nested for loop
→ Outer loop is for rows and inner loop is to track the columns of each row

```
int arr[2][3] = {1,2,3,4,5,6};

int i,j;
for(i = 0; i < 2 ; i++)
{
    for(j = 0; j < 3; j++)
    {
          printf(“%d\n”,arr[i][j]);
    }
}
```

Reading array elements through user:

```
int arr[2][3];
int i,j;

for(i = 0; i < 2 ; i++)
{
    for(j = 0; j < 3; j++)
    {
          scanf(“%d\n”,&arr[i][j]);
    }
}
```

Interpretation of 2d array:

arr[i][j]

- Replace a[i] with x
  arr[i][j] ⇒ x[j]

- From 1d array we know that x[j] can be interpreted as follow:
  x[j] = *(x + j)
       = *(x + j * sizeof(datatype))

- So, 2d array can be interpreted as,
  *(arr[i] + j)

- Further it can interpreted as,
  *(arr[i] + j) ⇒ *(*(arr + i) + j)

- At last 2d array interpretation looks like below
  ((arr + i * sizeof(1D)) + j * sizeof(datatype_array))

	1000	1004	1008	
Row 0	1	2	3	1d array
Row 1	4	5	5	1d array
	1012	1016	1020	

- With the above image, as a summary we can define 2D array as a combination of several 1D array
- It can be said that the base address of 1st row is 1000 and base address of 2nd row is 1012

E.g.,
Consider i = 1, j = 1 and base address of array is 1000
Arr[1][1]
= *(arr[1] + 1)
=*(arr[1] + 1 * sizeof(int))
=*(arr[1] + 1 * 4) = *(arr[1] + 4)
=*(*(arr + 1 * sizeof(1d)) + 4)
=*(*(1000 + 1 * 12) + 4
=*(*(1012) + 4)
=*(1012+4)
=*1016 ⇒ 5

With the above interpretation it is clear that 2d array can be interpreted in the following ways

1. arr[i][j]
2. *(arr[i] + j)
3. *(*(arr + i) + j)
4. (*(arr+i))[j]

Array of Pointers

→ array of pointers is a collection of address
→ syntax:

datatype *pointer_name[size];

→ e.g., int *ptr[3];

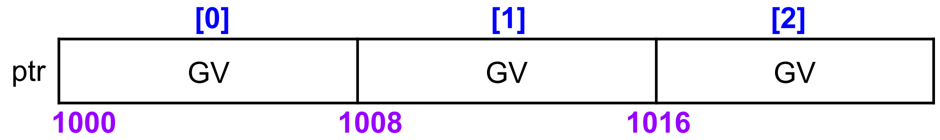
- Meaning of this declaration is ptr is a pointer which is capable of holding reference of 3 variable / memory location

- Total memory will be dependant on the bitness of system
  - If 32-bit system
    Total memory = size * sizeof(pointer)
    = 3 * 4
    = 12 bytes
  - If 64-bit system,
    Total Memory = 3 * 8 = 24 bytes

- Memory layout: 32-bit

	[0]	[1]	[2]
ptr	GV	GV	GV
	1000	1004	1008

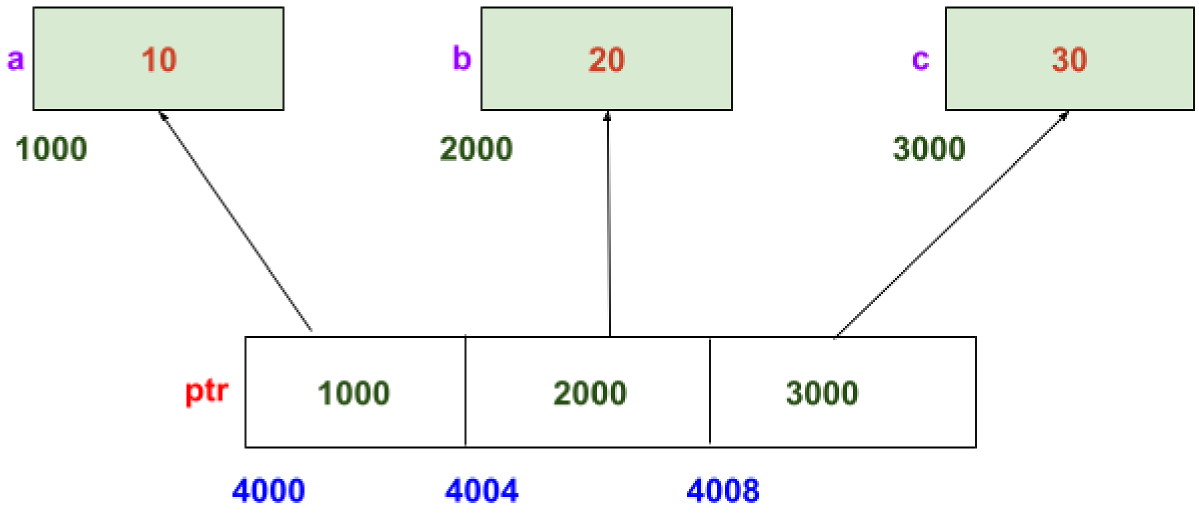
In 64-bit



Initialising and accessing array of pointers elements

```
int a = 10, b = 30, c = 40;
int *ptr[3];

ptr[0] = &a;
ptr[1] = &b;
Ptr[1] = &c;
OR
int *ptr = {&a, &b, &c};
```



Accessing array of pointers elements:

```
*ptr[0] = *(*ptr+0)
        =*(*4000 + 0 * sizeof(pointer))
        =*(*4000 + 0)
        =*1000
        = 10

*ptr[1] = *(*ptr+1)
        = *(*4000 + 1 * 4)
        =*(*4004)
        =*2000
        =20

*ptr[2] = *(*ptr+2)
        = *(*4000 + 2 * 4)
```

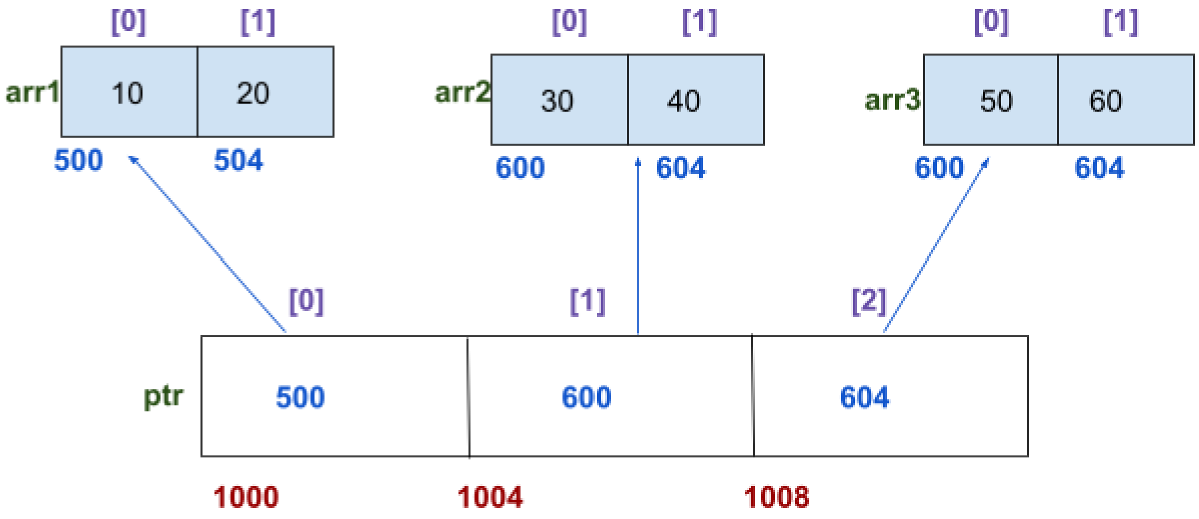
=*(*4008)

=*3000 = 30

→ array of pointers can be used to hold the address of 2 or more arrays

E.g.,

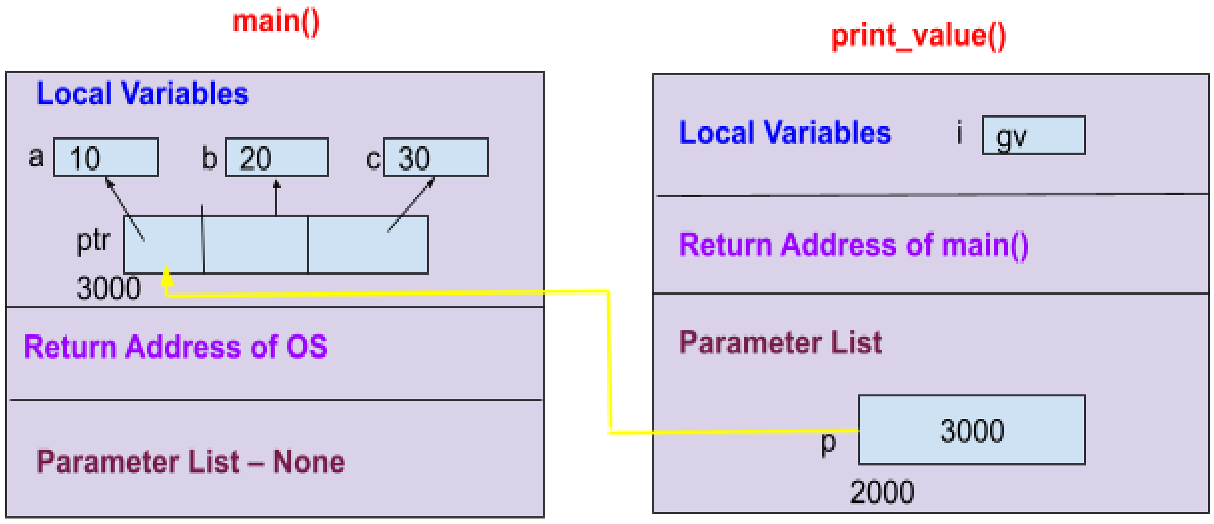
```
int arr1[2] = {10,20};
int arr2[2] = {30,40};
int arr3[2] = {50,60};
Int *ptr[3] = {arr1,arr2,arr3};
```



Passing array of pointer to function:

```
int main()
{
   int a=10,b=20,c=30;
   int *ptr[3] = {&a, &b, &c};
   print_value(ptr);
}
```

```
void print_value(int **p)
{
         int i;
         for(i = 0; i < 3 ; i++)
             printf("%d\n",*ptr[i]);
}
```



Array of strings

→ Array of string is a collection of strings, which is a 2 dimensional array
→ the first dimension says how many strings there are and the second dimension says about the maximum length of each string.
E.g., char s[3][8] = {“Array”,”Of”,”Strings”};
In the above example 3 is the number of strings and 8 is the length of each string.

Interpretation of array of strings:

Consider base address of s is 1000,

```
s[0] => *(s+0)
      = *(1000 + 0 * sizeof(1D))
      = *(1000 + 0 * 8)
      =*(1000)
      = 1000

s[1] => *(s+1)
      = *(1000 + 1 * 8)
      = *(1008)
      = 1008

s[2] = *(s + 2)
      = *(1000 + 2 * 8)
      = *(1016)
      = 1016
```

Pointer to an array (explicitly used in 2d arrays)

- Pointer to an array is a pointer which holds the whole address of an array
  E.g., int (*ptr)[3];
- Above example, ptr is a pointer which is pointing to array of 3 integer elements
- Pointer arithmetic on pointer to an array will be,
  ptr + 1 = ptr + 1 * sizeof(1D array)

- sizeof(*ptr) = 3 * sizeof(datatype)

Passing 2D array to function

1. The way array is declared
   → We can pass 2D array as a way its declared like,
   void print_array(int arr[2][3]);

2. Pointer to an array
   → Next way of passing 2d array to function is by using pointer to an array
   void print_array(int (*ptr)[3]);

3. Array of pointer
   → 2D arrays are also passed by using array of pointers
   void print_array(int *ptr[]);

4. By passing size along with array address
→ One of the recommended way of passing 2d array along with number of rows and Columns

```
void print_array(int row, int col, int arr[row][col]);
```

→ The order of the arguments should be in the above order else it will be an error if the arguments are like below:

```
void print_array(int arr[row][col],int row,int col);//compile time error
```

5. By normal integer pointer

```
void print_array(int row, int col, int *ptr);
```

—> one of complex way of passing 2d array is using normal pointer
→ To access the 2d array in this method need to use pointer arithmetic
→ accessing array element
*((p+i+number_of_columns) +j))

E.g., i = 1, j = 1, columns = 3, base address = 1000
*((1000+1*3*sizeof(int)) + 1)
*((1012)+1)
*(1012 + 1 * sizeof(int))
*(1012+4)
*1016

2D array Creations:

1. Both Static

- In this method both rows and columns are fixed.
- This kind of array is also known as Rectangular array
- E.g., **int arr[2][3] = {10, 20, 30, 40, 50, 60};**

2. First Static Second Dynamic(FSSD)

- Here, the number of rows will be fixed but columns will be variable.
- To create this type of array, will make use of array of pointers
- For example consider the number of rows are 2,
  int *ptr[2]; //rows

  To create the columns for each row we need to use dynamic memory allocation method,

  Read col value from user

```
for(i = 0; i < 2;i++)
{
        ptr[i] = malloc(col * sizeof(int));
}
```

3. First Dynamic Second Static(FDSS)

- Number of rows are variable but columns are fixed.
- To create such kind of array will use pointer to an array concept

- For example, consider u need 3 columns for each row, so

  ```
  int (*ptr)[3]; //number of columns
  ```

  Create rows using dynamic memory allocation
  //read value for row from user

  ```
  ptr = malloc(sizeof(*ptr) * row)
  ```

4. Both Dynamic

- Last way of creating 2d array is both rows and columns are dynamic
- It will achieved by using 2d level pointer that is **ptr.

  ```
  int **ptr;
  ```

- First create number of rows like,

  ```
  ptr = malloc(row * sizeof(int *));
  ```

- Then create columns for each row,

  ```
  for(i = 0; i < row ; i++)
  {
          ptr[i] = malloc(col * sizeof(int));
  }
  ```