5.2
Two-dimensional Arrays

Why Learning Two-dimensional Arrays?

- We have discussed one-dimensional arrays in which only a single index (or subscript) is needed to access a specific element of the array.
- The number of indexes that are used to access a specific element in an array is called the **dimension** of the array.
- Arrays that have more than one dimension are called multi-dimensional arrays.
- In this lecture, we focus mainly on two-dimensional arrays. We may use two-dimensional arrays to represent data stored in tabular form.
- Two-dimensional arrays are particularly useful for matrix manipulation.

Two-dimensional Arrays

- **Two-dimensional Arrays Declaration, Initialization and Operations**
- Two-dimensional Arrays and Pointers
- Two-dimensional Arrays as Function Arguments
- Applying 1-D Array to Process 2-D Arrays
- Sizeof Operator and Arrays

Two-dimensional (or Multi-dimensional) Arrays Declaration

- Declared as **consecutive** pairs of brackets.
- E.g. a **2-dimensional** array is declared as follows:

```
int x[3][5];    // a 3-element array of 5-element arrays
```

- E.g. a **3-dimensional** array is declared as follows:

```
char x[3][4][5]; // a 3-element array of 4-element arrays of 5-element arrays
```

- ANSI C standard requires a minimum of **6 dimensions** to be supported.

Two-dimensional Arrays: Memory Layout

int x[3][5];

Row-major order   i.e. x[row][column]

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]
Row 1	x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]
Row 2	x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]

↓

Memory Layout :

x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]	x[1][0]		x[1][4]	x[2][0]		x[2][4]
1	**2**	**3**	**4**	**5**	**6**	......	**10**	**11**	......	**15**

Row 0   Row 1   Row 2

Consecutive & sequential memory

Initializing Two-dimensional Arrays

- **Initializing** multidimensional arrays: enclose each row in braces.

```
int x[2][2] = {  { 1, 2},    /* row 0 */
                 { 6, 7}  }; /* row 1 */
```

or

```
int x[2][2] = { 1, 2, 6, 7};
```

- **Partial** initialization:

```
int exam[3][3] = { {1,2}, {4}, {5,7} };

int exam[3][3] = { 1,2,4,5,7 };
        i.e. = { {1,2,4}, {5,7}};
```

Operations on 2-D Arrays – Sum of Rows

```
#include <stdio.h>
int main()
{    // declare an array with initialization
     int array[3][3]={                    column
          row      {5, 10, 15},
                   {10, 20, 30},
                   {20, 40, 60}
                   };
     int row, column, sum;
     /* compute sum of row - traverse each row first */
     for (row = 0; row < 3; row++)              // nested loop
     {
          /* for each row – compute the sum */
          sum = 0;
          for (column = 0;  column < 3;   column++)
             sum += array[row][column];
          printf("Sum of row %d is %d\n", row, sum);
     }
      return 0;
}
```

Output

Sum of row 0 is 30
Sum of row 1 is 60
Sum of row 2 is 120

Nested Loop

Operations on 2-D Arrays – Sum of Columns

```
#include <stdio.h>
int main()
{   // declare an array with initialization
    int array[3][3]={                    column
          row     {5, 10, 15},
                  {10, 20, 30},
                  {20, 40, 60}
                  };
    int row, column, sum;
    /* compute sum of each column */
    for (column = 0; column < 3; column++)
    {
        sum = 0;
        for (row = 0; row < 3; row++)
            sum += array[row][column];
        printf("Sum of column %d is %d\n", column, sum);
    }
    return 0;
}
```

Output

```
Sum of column 0 is 35
Sum of column 1 is 70
Sum of column 2 is 105
```

Two-dimensional Arrays

- Two-dimensional Arrays Declaration, Initialization and Operations
- **Two-dimensional Arrays and Pointers**
- Two-dimensional Arrays as Function Arguments
- Applying 1-D Array to Process 2-D Arrays
- Sizeof Operator and Arrays

Two-dimensional Arrays and Pointers

- Two-dimensional array variable declaration:

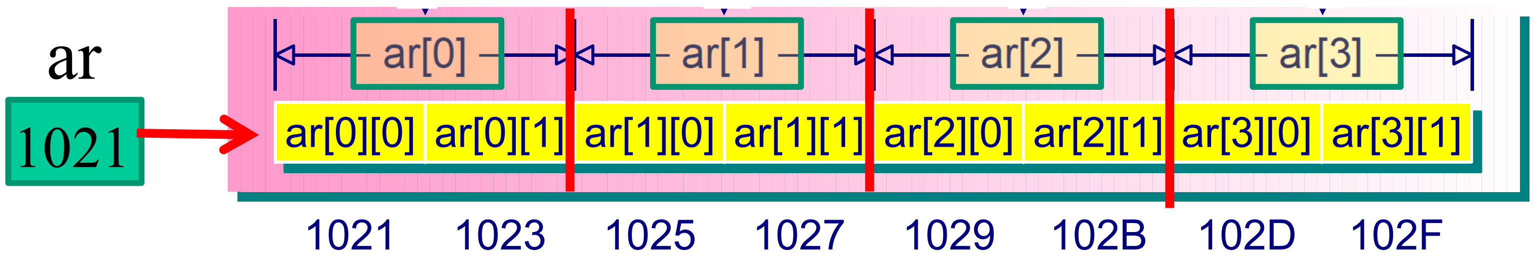
```
int ar[4][2];
```

/* **ar** is an array of **4** elements; each element is an array of **2** ints */

or

```
int ar[4][2] = {
    {1, 2},
    {3, 4},
    {5, 6},
    {7, 8}
};
```

2-D array data are stored sequentially in the memory



Two-dimensional Arrays and Pointers

- Two-dimensional array variable declaration:

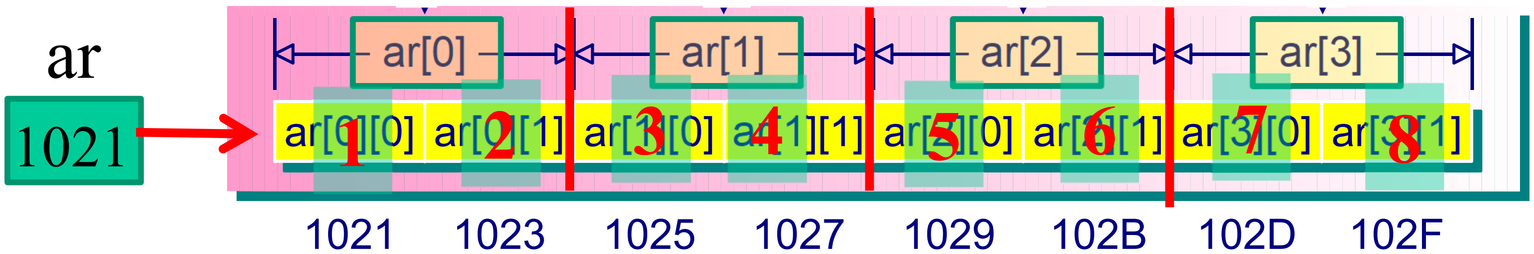
int ar[4][2]; /* **ar** is an array of **4** elements; each element is an array of **2** ints */

or

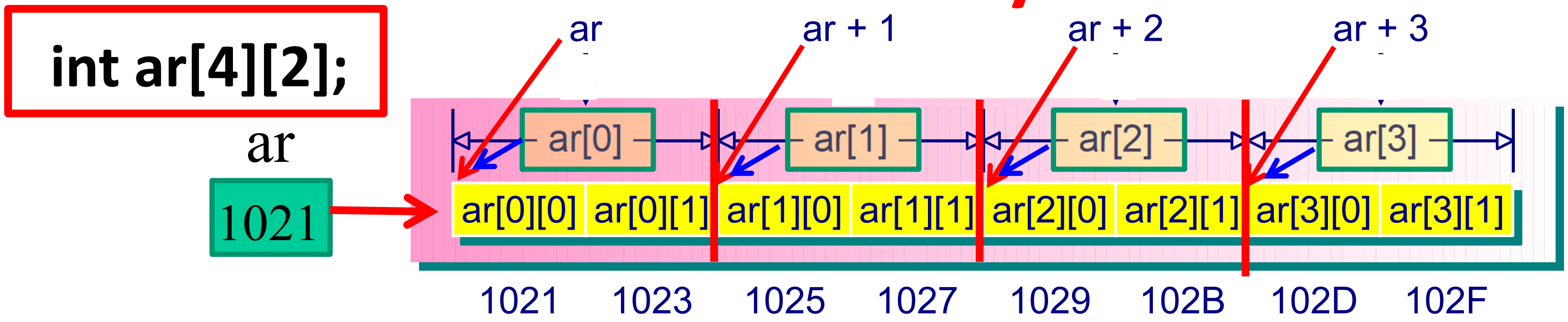
```
int ar[4][2] = {
              {1, 2},
              {3, 4},
              {5, 6},
              {7, 8}
          };
```

2-D array data are stored sequentially in the memory

- After array declaration, memory locations are allocated and used to store the initial values of the array.



Two-dimensional Arrays and Pointers



- **ar** - the address of the **1st element** of the array. In this case, the 1st element is an **array of 2 ints**. So, **ar** is the address of a **two**-int-sized object.

ar == &ar[0]
ar + 1 == &ar[1]
ar + 2 == &ar[2]
ar + 3 == &ar[3]

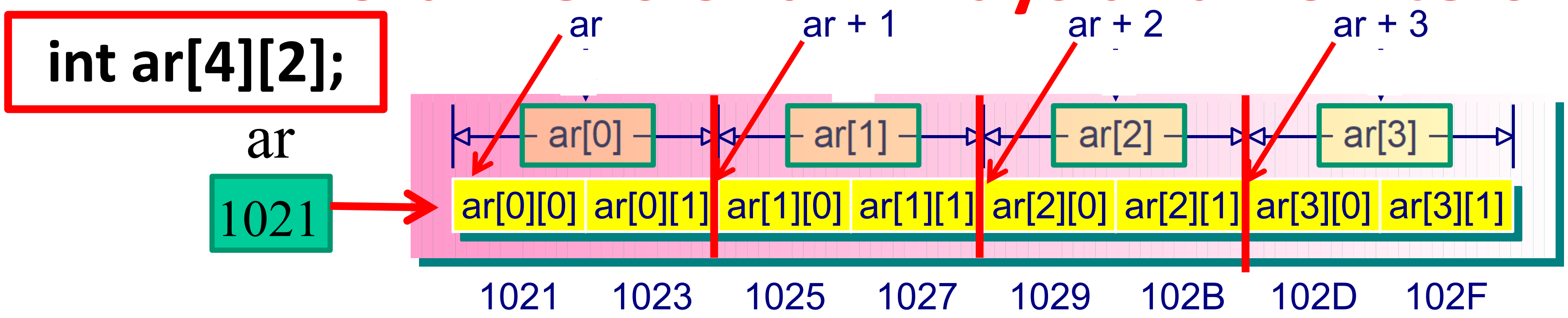
Note: **Adding 1** to a pointer or address yields a value larger by the size of the referred-to object.
e.g. **ar** has the same address value as **ar[0]**
ar+1 has the same address value as **ar[1]**, etc.

- **ar[0]** is an array of **2** integers, so **ar[0]** is the **address** of **int-sized object**.

ar[0] == &ar[0][0]
ar[1] == &ar[1][0]
ar[2] == &ar[2][0]
ar[3] == &ar[3][0]

Note:
- **ar[0]** has the same address as **ar[0][0]**;
- **ar[0]+1** refers to the address of **ar[0][1]** (i.e. 1023)

Two-dimensional Arrays and Pointers



- **Dereferencing** a pointer or an address (apply *** operator**) yields the **value** represented by the referred-to object.

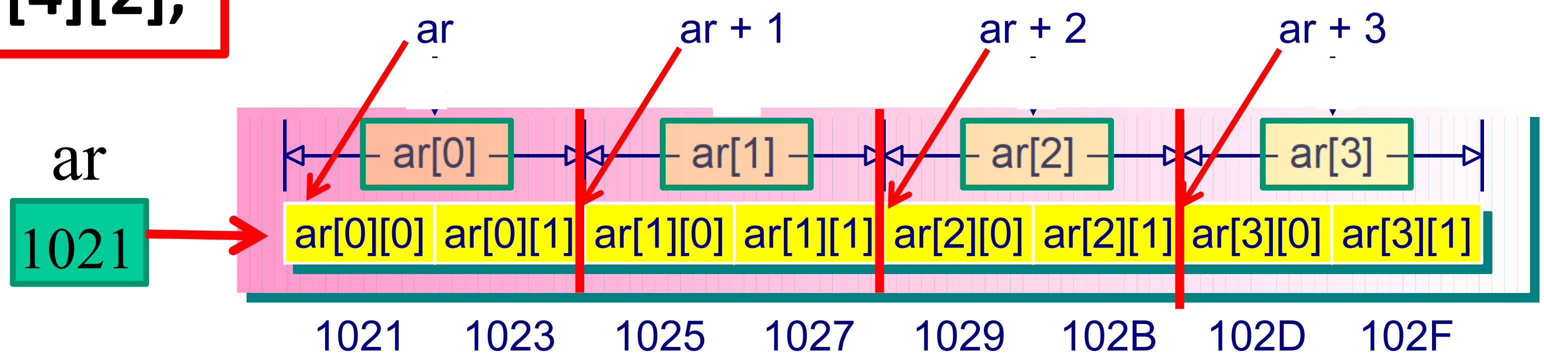
ar == &ar[0]	***ar == ar[0]**	**(by dereferencing)**
ar + 1 == &ar[1]	***(ar + 1) == ar[1]**	
ar + 2 == &ar[2]	***(ar + 2) == ar[2]**	
ar + 3 == &ar[3]	***(ar + 3) == ar[3]**	

- Similarly

ar[0] == &ar[0][0]	***ar[0] == ar[0][0]**	**(dereferencing)**
ar[1] == &ar[1][0]	***ar[1] == ar[1][0]**	
ar[2] == &ar[2][0]	***ar[2] == ar[2][0]**	
ar[3] == &ar[3][0]	***ar[3] == ar[3][0]**	

Two-dimensional Arrays and Pointers

int ar[4][2];



- Therefore:

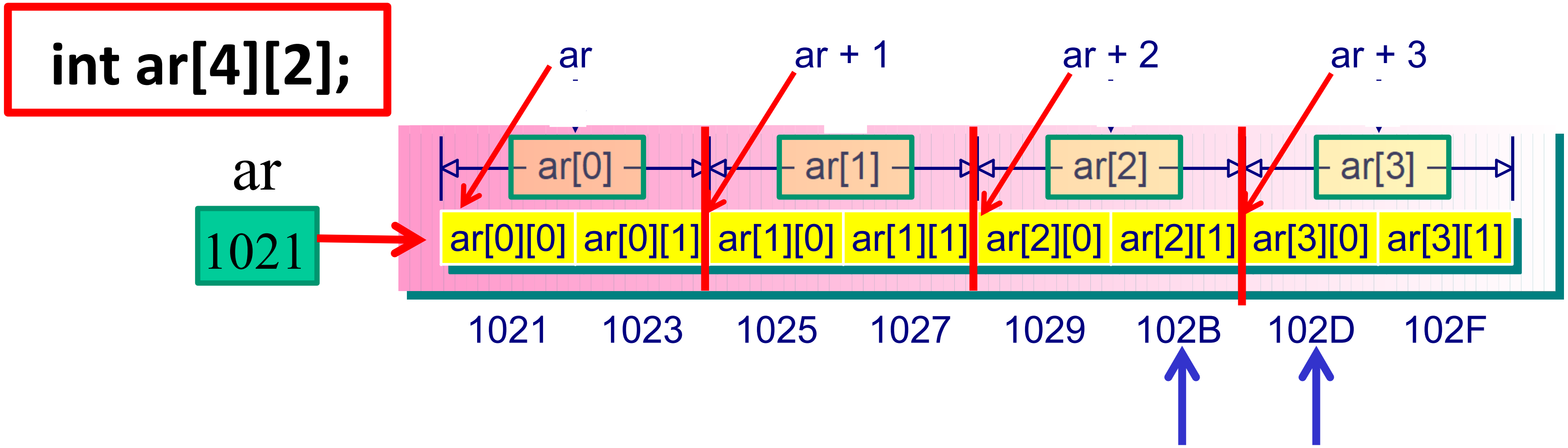
  ***ar[0]** == the value stored in **ar[0][0]**.

  ***ar** == the value of its first element, **ar[0]**.

  we have

  ****ar** == the value of **ar[0][0]** (**double indirection**)

Two-dimensional Arrays and Pointers



- After some calculations using **double** dereferencing as shown above, we will get the general formula for using pointer to access each element of a 2-D array **ar** with row=**m**, column=**n**, as follows:

ar[m][n] == *(*(ar + m) + n)

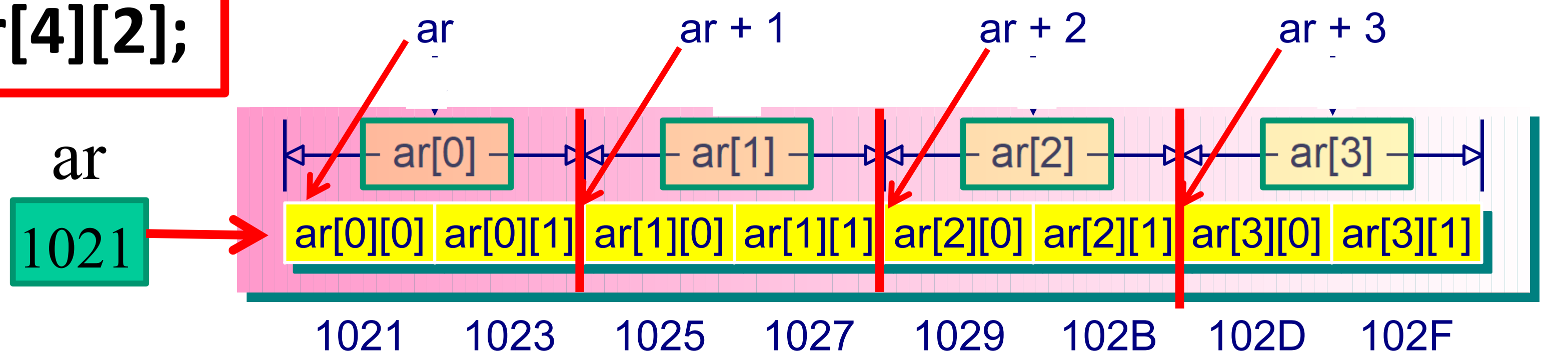
e.g. **ar[2][1]** = *(*(ar + **2**) + **1**) [m=2, n=1]

ar[3][0] = *(*(ar + **3**) + **0**) [m=3, n=0]

Note: you are not required to remember the calculation on deriving the general formula.

Two-dimensional Arrays and Pointers

int ar[4][2];



Two ways to access two-dimensional Array:

- Using the two indexes (e.g. **m** and **n**):

  e.g. **ar[m][n]**

- Using pointers and the general formula for two-dimensional array:

  ar[m][n] == *(*(ar + m) + n)

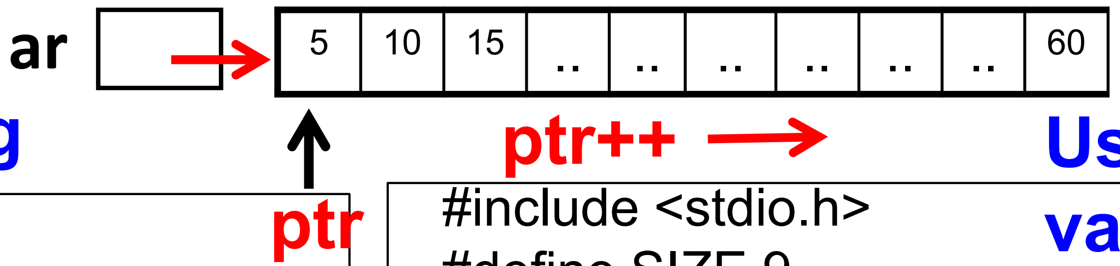
Processing Two-dimensional Arrays: Example

```
#include <stdio.h>
int main() {
  int ar[3][3]=  {
    {5, 10, 15},
    {10, 20, 30},
    {20, 40, 60}
  };
  int i, j;
    // (1) using indexing approach
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      printf("%d ", ar[i][j]);
  printf("\n");
    // (2) using the pointer formula
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      printf("%d ", *(*(ar+i)+j));
  return 0;
}
```

Output

5 10 15 10 20 30 20 40 60
5 10 15 10 20 30 20 40 60

Processing 2-D Arrays (Indexing vs Pointer Variable)



Using indexing

```
#include <stdio.h>
int main ( ) {
   int ar[3][3]=  {
      {5, 10, 15},
      {10, 20, 30},
      {20, 40, 60}
      };
   int i, j;

   /* using index – nested loop*/
   printf("\n");
   for (i=0; i<3; i++)
      for (j=0; j<3; j++)
         printf("%d ", ar[i][j]);
   printf("\n");
   return 0;
}
```

Using pointer variable

```
#include <stdio.h>
#define SIZE 9
int main ( ) {
   int ar[3][3]=  {
      {5, 10, 15},
      {10, 20, 30},
      {20, 40, 60}
      };
   int i;
   int *ptr;
   ptr = *ar;
   /* using pointer - looping */
   for (i=0; i<SIZE; i++)
      printf("%d ", *ptr++);
   printf("\n");
   return 0;
}
```

Two-dimensional Arrays

- Two-dimensional Arrays Declaration, Initialization and Operations
- Two-dimensional Arrays and Pointers
- **Two-dimensional Arrays as Function Arguments**
- Applying 1-D Array to Process 2-D Arrays
- Sizeof Operator and Arrays

Two-dimensional Arrays as Function Arguments

- The definition of a function with a 2-D array as the argument is:

```
void fn(int array[2][4])
{
      ....
}
```

or

```
void fn(int array[ ][4])
{
      ....
}
```

/*note that the first dimension can be excluded*/

In the above definition, the **first dimension can be excluded** because the C compiler does not need the information of the first dimension.

Why the First Dimension can be Omitted?

- For example, in the assignment operation: **array[1][3]** = 100; requests the **compiler** to compute the address of **array[1][3]** and then place 100 to that address.
- In order to compute the address, the dimension information of the array must be given to the compiler.
- Let's redefine **array** as

int **array[D1][D2]**; // with **D1**=**2**, **D2**=**4**

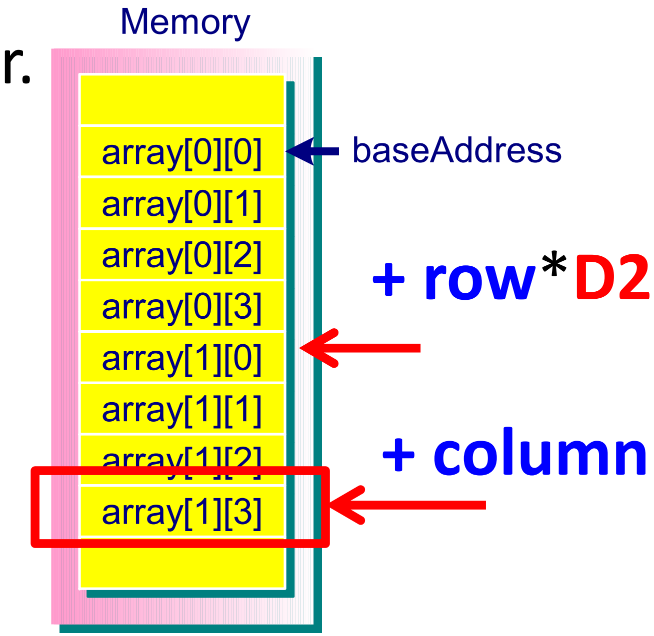
The address of array[**1**][**3**] is computed as:

baseAddress + **row** * **D2** + **column**

==>   baseAddress + **1** * **4** + **3**

==>   baseAddress + 7

The **baseAddress** is the address pointing to the beginning of array.



Why the First Dimension can be Omitted? (Cont'd.)

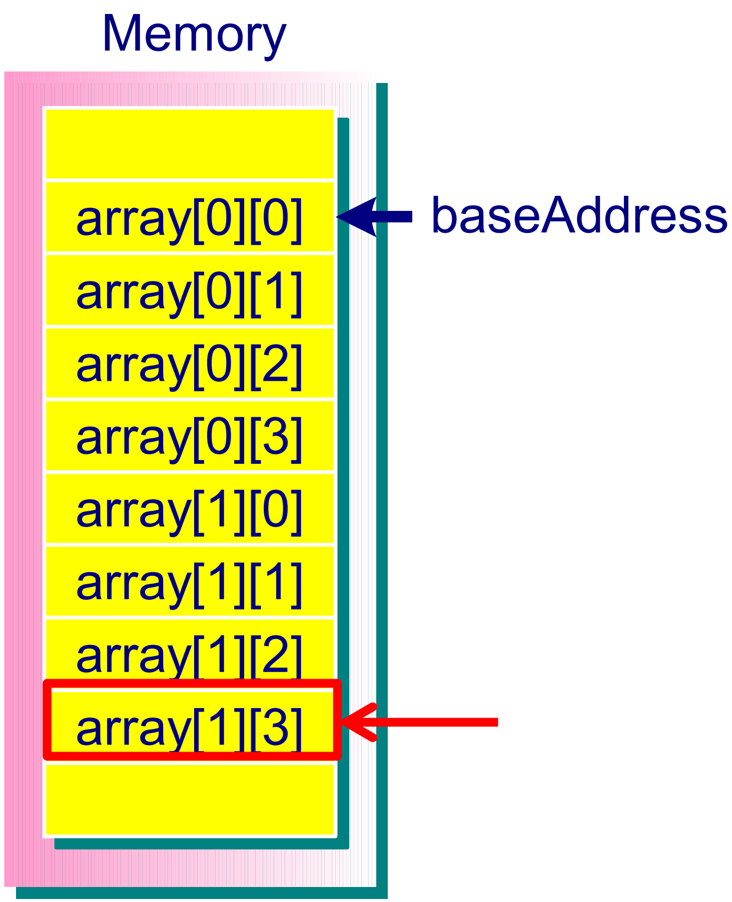
- Since **D1** is **not needed in computing the address**, we can omit the first dimension value in defining a function which takes arrays as its formal arguments.

- Therefore, the prototype of the function could be:

  void fn(int **array[2][4]**);

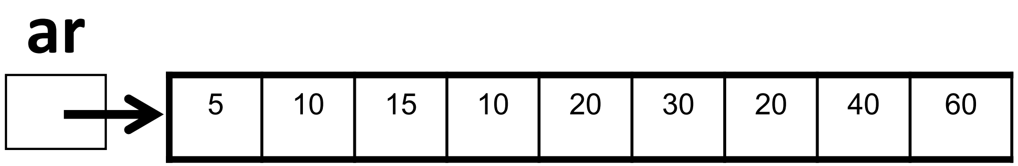
  or

  void fn(int **array[ ][4]**);



Passing 2-D Array as Function Arguments: Example

```c
#include <stdio.h>
int sum_all_rows(int array[ ][3]);
int sum_all_columns(int array[ ][3]);
int main()
{
    int ar[3][3]= {
                   {5, 10, 15},
                   {10, 20, 30},
                   {20, 40, 60}
    };
    int total_row, total_column;
    total_row = sum_all_rows(ar);  // sum of all rows
    total_column = sum_all_columns(ar); //all columns
    printf("The sum of all elements in rows is %d\n", total_row);
    printf("The sum of all elements in columns is %d\n", total_column);
    return 0;
}
```



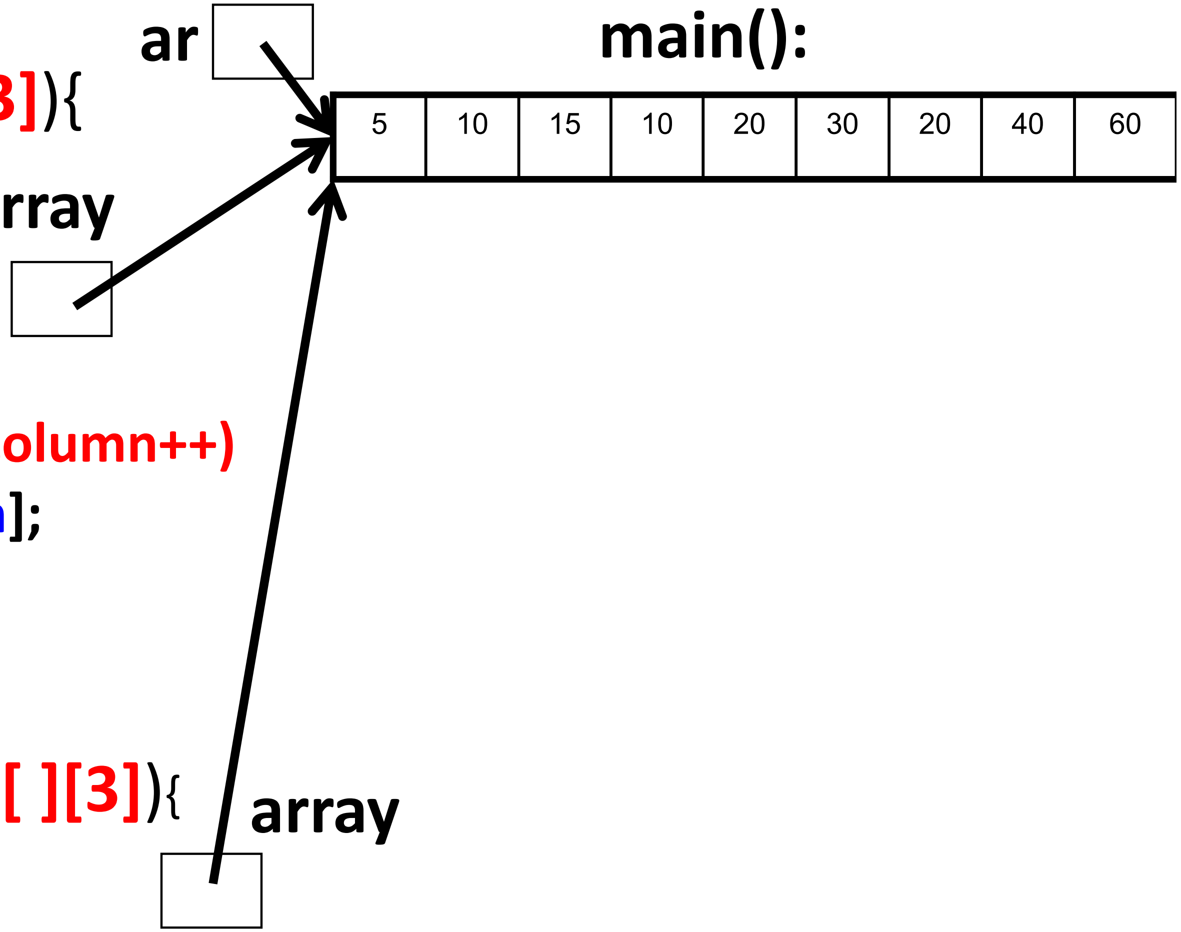
Output

The sum of all elements in rows is 210

The sum of all elements in columns is 210

Passing 2-D Array as Function Arguments: Example

```
int sum_all_rows(int array[ ][3]){
    int row, column;
    int sum=0;
    for (row = 0; row < 3; row++)
    {
        for (column = 0; column < 3; column++)
            sum += array[row][column];
    }
    return sum;
}
int sum_all_columns(int array[ ][3]){
    int row, column;
    int sum=0;
    for (column = 0; column < 3; column++)
    {
        for (row = 0; row < 3; row++)
            sum += array[row][column];
    }
    return sum;
}
```



Two-dimensional Arrays

- Two-dimensional Arrays Declaration, Initialization and Operations
- Two-dimensional Arrays and Pointers
- Two-dimensional Arrays as Function Arguments
- **Applying 1-D Array to Process 2-D Arrays**
- Sizeof Operator and Arrays

Applying 1-D Array to Process 2-D Arrays in Functions: Using Pointers

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[ ], int size);

int main()
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;

    for (i=0; i<2; i++) {  /* as 2-D Array */
        display1(array[i], 4);
    }

    return 0;
}
```

Row: array[0]   array[1]

// Using pointers

```
void display1(int *ptr, int size)
{
    int j;

    printf("Display1 result: ");
    for (j=0; j<size; j++)
        printf("%d ", *ptr++);
    putchar('\n');
}
```

ptr

Output:

```
Display1 result: 0 1 2 3
Display1 result: 4 5 6 7
```

Applying 1-D Array to Process 2-D Arrays in Functions: Using Pointers

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[ ], int size);

int main()
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;

    for (i=0; i<2; i++) {  /* as 2-D Array */
        display1(array[i], 4);
    }

    display1(array, 8);  /* as 1-D array */

    return 0;
}
```

```
void display1(int *ptr, int size)
{
    int j;

    printf("Display1 result: ");
    for (j=0; j<size; j++)
        printf("%d ", *ptr++);
    putchar('\n');
}
```

array

ptr

Output:
Display1 result: 0 1 2 3
Display1 result: 4 5 6 7
Display1 result: 0 1 2 3 4 5 6 7

Applying 1-D Array to Process 2-D Arrays in Functions: Using Indexing

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[ ], int size);

int main()                  array[0]     array[1]
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;

    for (i=0; i<2; i++) {  /* as 2-D Array */
        display2(array[i], 4);
    }

    return 0;
}
```

// Using indexes

```
void display2(int ar[ ], int size)
{
    int k;                         ar

    printf("Display2 result: ");
    for (k=0;  k<size;  k++)
        printf("%d ", ar[k]*5);
    putchar('\n');
}
```

Output:
Display2 result: 0 5 10 15
Display2 result: 20 25 30 35

Applying 1-D Array to Process 2-D Arrays in Functions: Using Indexing

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[ ], int size);

int main()
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;

    for (i=0; i<2; i++) {  /* as 2-D Array */
        display2(array[i], 4);
    }
    display2(array, 8);  /* as 1-D array */

    return 0;
}
```

```
void display2(int ar[ ], int size)
{
    int k;

    printf("Display2 result: ");
    for (k=0;  k<size;  k++)
        printf("%d ", ar[k]*5);
    putchar('\n');
}
```

array

ar

Output:
Display2 result: 0 5 10 15
Display2 result: 20 25 30 35
Display2 result: 0 5 10 15 20 25 30 35

Example: minMax()

Write a C function minMax() that takes a 5x5 two-dimensional array of integers *a* as a parameter. The function returns the minimum and maximum numbers of the array to the caller through the two parameters *min* and *max* respectively. [using call by reference]

```
#include <stdio.h>
  void minMax(int a[5][5], int *min, int *max);
  int main()
  {
    int A[5][5];
    int i, j;
    int min, max;

    printf("Enter your matrix data (5x5): \n");
    // nested loop
    for (i=0; i<5; i++)
      for (j=0; j<5; j++)
        scanf("%d", &A[i][j]);
    minMax(A, &min, &max);
    printf("min = %d; max = %d", min, max);
    return 0;
  }
```

```
void minMax(int a[5][5], int *min,
            int *max)
{
    int i, j;
    /* add your code here */
```

Q: Using indexing?

Q: Using pointer?

```
}
```

minMax: Using the Array Indexing Approach

Using indexing:

```
void minMax(int a[5][5],
            int *min,
            int *max)
{
    int i, j;

    *max = a[0][0];
    *min = a[0][0];
    for (i=0; i<5; i++)
      for (j=0; j<5; j++)
      {
          if (a[i][j] > *max)
              *max = a[i][j];
          else if (a[i][j] < *min)
              *min = a[i][j];
      }
}
```

a

main():

```
int A[5][5]= {
        {5, 10, 15, 20, 25},
        {10, 20, 30, 40, 50},
        {20, 40, 60, 80, 100},
        {1, 3, 5, 7, 9},
        {2, 4, 6, 8, 10}
};
```

col

row

5	10	…	…	…	…	…	6	8	10

minMax: Using Pointer Variable Approach

Using pointer variable:

```
void minMax(int a[5][5], int *min, int
*max)
{
   int i;
   int *p;
   p=*a;

   *max = *p;
   *min = *p;
   for (i=0; i<25; i++) {
        if ( *p > *max )
           *max = *p;
        else if ( *p < *min )
           *min = *p;
        p++;
   }
}
```

a

p

Using pointer variable to process 2D arrays

main():

```
int A[5][5]= {
            {5, 10, 15, 20, 25},
            {10, 20, 30, 40, 50},
            {20, 40, 60, 80, 100},
            {1, 3, 5, 7, 9},
            {2, 4, 6, 8, 10}
};
```

Consecutive & sequential memory

5	10	…	…	…	…	…	6	8	10

p++

Two-dimensional Arrays

- Two-dimensional Arrays Declaration, Initialization and Operations
- Two-dimensional Arrays and Pointers
- Two-dimensional Arrays as Function Arguments
- Applying 1-D Array to Process 2-D Arrays
- **Sizeof Operator and Arrays**

Sizeof Operator and Array

- **sizeof**(operand) is an operator which gives the **size** (i.e. how many bytes) of its operand. Its syntax is

  sizeof (operand)

  or

  sizeof operand

- The **operand** can be:

  int, float, ..., complexDataTypeName, variableName, arrayName

Sizeof Operator and Array: Example

```
#include <stdio.h>
int sum(int  a[ ], int n);
int main(){
    int ar[6] = {1,2,3,4,5,6};
    int total;
    printf("Array size is %d\n",
        sizeof(ar)/sizeof(ar[0]));
    total = sum (ar, 6);
    return 0;
}
int sum ( int a[ ], int n ) {
    int i, total=0;
    printf("Size of a = %d\n", sizeof(a));
    for ( i=0;   i<n ;   i++)
        total += a[i];
    return total;
}
```

Output
Array size is **6**
(i.e. 24/4=6)
Size of a = **4**

Apply ***sizeof*** to a **pointer variable (e.g. a)** yields the size of the pointer.

Thank You!