5.2
Multi-dimensional Arrays

1

Review

- What have you learnt so far:
  - Basic Sequential Programming (data, operators, simple I/O)
  - Branching (if-else, switch, conditional operator)
  - Looping (for, while, do…while)
  - Functions (function definition and call by value)
  - Pointers (call by reference for function communication)
  - **1-D Arrays (+ using pointers for 1D arrays)**
- In this lecture, we will discuss **… 2D (Multi-dimensional) Arrays …** and the use of pointers in 2D arrays.

2

This Lecture

- At the end of this lecture, you will be able to understand the following:
  - Multi-dimensional Arrays Declaration, Initialization and Operations
  - Multi-dimensional Arrays and Pointers
  - Multi-dimensional Arrays as Function Arguments
  - Applying 1-D Array to 2-D Arrays
  - Sizeof Operator and Arrays

3

Multi-dimensional Arrays

In the last lecture, 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 the subsequent discussion, 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 in matrix manipulation.

Multi-dimensional Arrays

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

4

Multi-dimensional Arrays Declaration

- Declared as **consecutive** pairs of brackets.
- E.g. **2-dimensional**; 3-element array of 5-element arrays :

  int x[3][5];

- E.g. **3-dimensional**; 3-element array of 4-element arrays of 5-element arrays

  char x[3][4][5];

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

5

Multi-dimensional Arrays Declaration

A two-dimensional array can be declared as

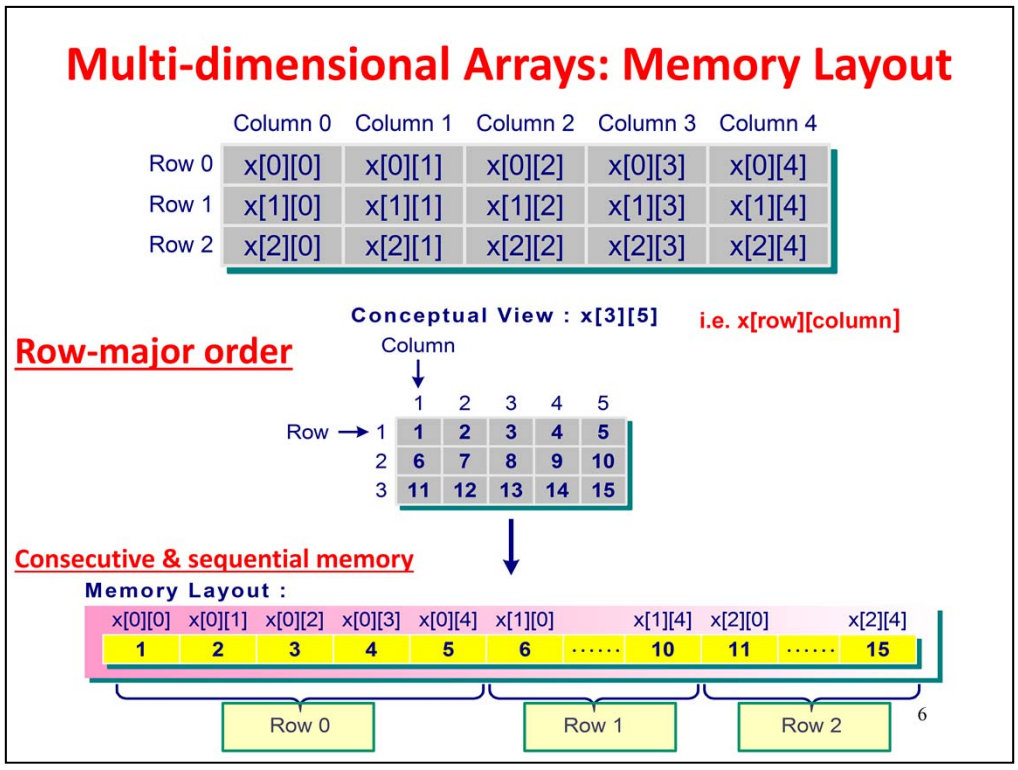
```
int x[3][5];          /* 3-element array of 5-element arrays */
```

Two indexes are needed to access each element of the array.

Similarly, a three-dimensional array can be declared as

```
char x[3][4][5];
```

Three indexes are used to access a specific element of the array. ANSI C standard supports arrays with a maximum of 6 dimensions.



Multi-dimensional Arrays: Memory Layout

The statement

int x[3][5];

declares a two-dimensional array **x[][]** of type **int** having three rows and five columns. The compiler will set aside the memory for storing the elements of the array. The two-dimensional array can also be viewed as a table made up of rows and columns. For example, the array **x[3][5]** can be represented as a table.

The array consists of three rows and five columns. The array name and two indexes are used to represent each individual element of the array. The first index is used for the row, and the second index is used for column ordering. **x[0][0]** represents the first row and first column, and **x[1][0]** represents the second row and first column, and **x[1][3]** represents second row and fourth column, etc. A two-dimensional array is stored in *row-major* order in the memory.

Note that the memory storage of the two-dimensional array **x[3][5]** is consecutive and sequential.

Initializing Multi-dimensional Arrays

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

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

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}};
```

7

Initializing Multi-dimensional Arrays

For the initialization of multi-dimensional arrays, each row of data is enclosed in braces:

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

The data in the first interior set of braces is assigned to the first row of the array, the data in the second interior set goes to the second row, etc. If the size of the list in the first row is less than the array size of the first row, then the remaining elements of the row are initialized to zero. If there are too many data, then it will be an error.

Since the inner braces are optional, a multi-dimensional array can be initialized as

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

An array can also be initialized partially:

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

This statement initializes the first two elements in the first row, the first element in the second row, and the first two elements in the third row. All elements that are not initialized are set to zero by default.

The following statement

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

is valid, but the two-dimensional array is initialized as

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

If the number of rows and columns are not specified in the declaration statement, the compiler will determine the size of the array based on information about the initialization

of the array. For example, the following statement

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

will create an array having two rows and two columns. The array is also initialized according to the values specified in the initialization of the array.

Initializing Multidimensional Arrays (Cont'd.)

- Can **omit** the **outermost dimension** because compiler can figure that out. E.g.

```
int arr[ ][3][2] = {
        { {1,1},{0,0},{1,1}  },
        { {0,0},{1,2},{0,1}  }
        };
```

gives a **[2]**[3][2] dimensioned array.

- The following is not correct. Why?

```
int wrong_arr[ ][ ] = {1,2,3,4};
```

8

Initializing Multi-dimensional Arrays

For higher dimensional arrays, we can omit the outermost dimension because the compiler can determine the dimension automatically. For example, the following array declaration

```
int array[][3][2]= {    {{1,1}, {0,0}, {1,1}},
                        {{0,0}, {1,2}, {0,1}}   };
```

creates a [2][3][2] dimensioned array.

However, the statement

```
int wrong_array[][] = { 1,2,3,4 };
```

is incorrect, as the compiler is unable to determine the size of the array.

Operations on 2-D Arrays - using Array Index

```
#include <stdio.h>
int main()
{   // declare an array with initialization
    int array[3][3]={
            {5, 10, 15},
            {10, 20, 30},
            {20, 40, 60}
            };
    int row, column, sum;
    /* compute sum of row - traverse each row first */
```

row ↓ column →

Output

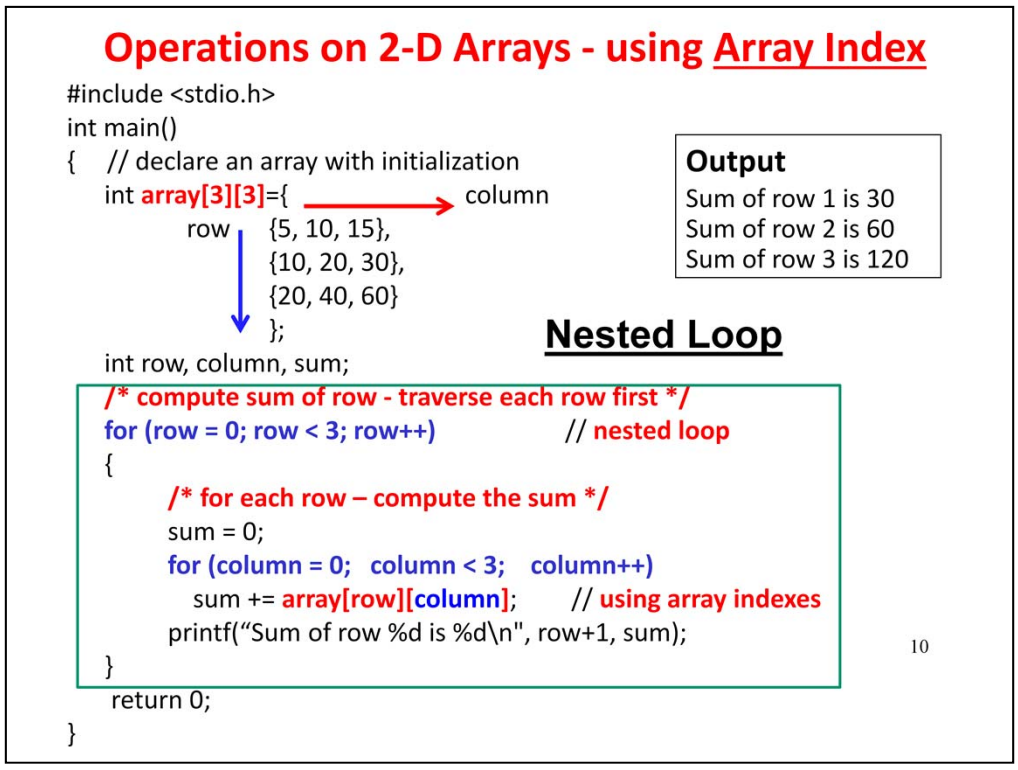
```
Sum of row 1 is 30
Sum of row 2 is 60
Sum of row 3 is 120
```

9

```
    return 0;
}
```

Operations on 2-D Arrays

The program determines the sum of rows of two-dimensional arrays. It uses indexes to traverse each element of the 2-D **array**. In the program, the array is first initialized.



Operations on 2-D Arrays

When accessing two-dimensional arrays using indexes, we use an index variable **row** to refer to the row number and another index variable **column** to refer to the column number. A nested **for** loop is used to access the individual elements of the array. To process the sum of rows, we use the index variable **row** as the outer **for** loop. Then, it traverses each element of each row and add them up to give the sum of rows. Note that the first dimension of an array is row and the second dimension is column. It is row-major.

Operations on 2-D Arrays - using Array Index

```
#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 */




    return 0;
}
```

Output

```
Sum of column 1 is 35
Sum of column 2 is 70
Sum of column 3 is 105
```

11

Operations on 2-D Arrays

The program determines the sum of columns of two-dimensional arrays. It uses indexes to traverse each element of the 2-D **array**. In the program, the array is first initialized.

Operations on 2-D Arrays - using Array Index

```
#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+1, sum);
    }
    return 0;
}
```

Output

```
Sum of column 1 is 35
Sum of column 2 is 70
Sum of column 3 is 105
```

12

Operations on 2-D Arrays

To process the sum of columns, a nested **for** loop is used. We use the index variable **column** as the outer **for** loop. Then, it traverses each element of each column and add them up to give the sum of columns. Again note that the first dimension of an array is row and the second dimension is column. It is row-major.

Multi-dimensional Arrays

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

13

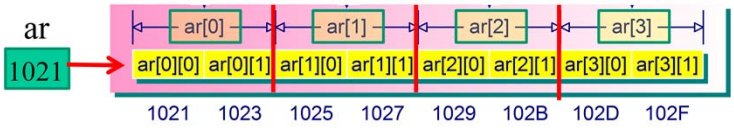
Multi-dimensional Arrays and Pointers (Cont'd.)

- Multidimensional arrays - stored sequentially in memory.

```
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}
                };
```



14

Multi-dimensional Arrays and Pointers

Multi-dimensional arrays are also stored sequentially in the memory. For example, consider the following two-dimensional array:

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

where **ar** is also the address of the first element of the array.

The memory layout of the two-dimensional array is shown with its associated pointers. We will discuss the association between the array elements and the associated pointers in the next few slides.

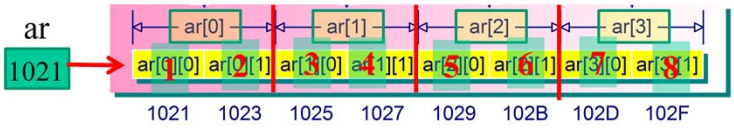
Multi-dimensional Arrays and Pointers (Cont'd.)

- Multidimensional arrays - stored sequentially in memory.

```
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}
                };
```



15

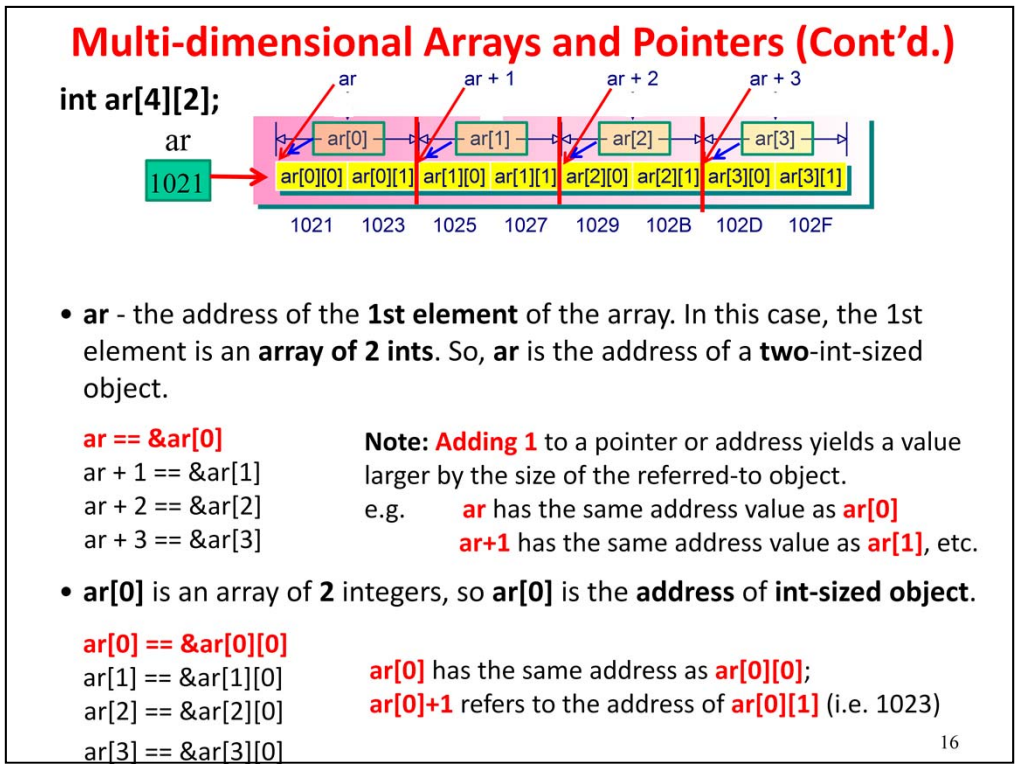
Multi-dimensional Arrays and Pointers

Multi-dimensional arrays are also stored sequentially in the memory. For example, consider the following two-dimensional array:

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

where **ar** is also the address of the first element of the array.

The memory layout of the two-dimensional array is shown with its associated pointers. We will discuss the association between the array elements and the associated pointers in the next few slides.



Multi-dimensional Arrays and Pointers

The first element is an array of 2 integers. **ar** is the address of a two-integer sized object. We have

ar == &ar[0]

ar + 1 == &ar[1]

ar + 2 == &ar[2]

ar + 3 == &ar[3]

ar[0] is an array of 2 integers, so **ar[0]** is the address of an integer sized object. Therefore, we have

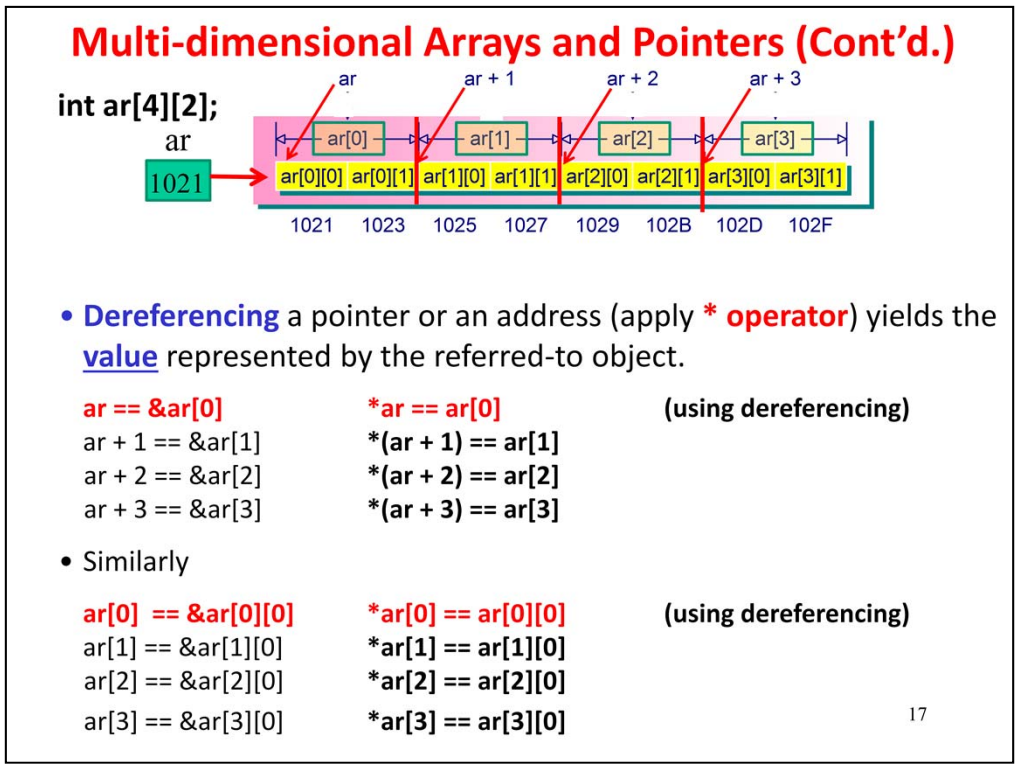
ar[0] == &ar[0][0]

ar[1] == &ar[1][0]

ar[2] == &ar[2][0]

ar[3] == &ar[3][0]

Note that adding 1 to a pointer or address yields a value larger by the size of the referred-to object. For example, although **ar** has the same address value as **ar[0]**, **ar+1** (i.e. 1025) is different from **ar[0]+1** (i.e. 1023). This is due to the fact that **ar** is a two-integer sized object, while **ar[0]** is an integer sized object. Adding 1 to **ar** increases by 4 bytes. **ar[0]** refers to ***ar**, which is the address of an integer, adding 1 to it increases by 2 bytes.



Multi-dimensional Arrays and Pointers

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

ar == &ar[0], using dereferencing we have ***ar == ar[0]**

ar + 1 == &ar[1], using dereferencing we have ***(ar + 1) == ar[1]**

ar + 2 == &ar[2], using dereferencing we have ***(ar + 2) == ar[2]**

ar + 3 == &ar[3], using dereferencing we have ***(ar + 3) == ar[3]**

Similarly, we have

ar[0] == &ar[0][0], using dereferencing we have ***ar[0] == ar[0][0]**

ar[1] == &ar[1][0], using dereferencing we have ***ar[1] == ar[1][0]**

ar[2] == &ar[2][0], using dereferencing we have ***ar[2] == ar[2][0]**

ar[3] == &ar[3][0], using dereferencing we have ***ar[3] == ar[3][0]**

Multidimensional Arrays and Pointers (Cont'd.)

- 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***)

18

Multi-dimensional Arrays and Pointers

We can use the * operator to dereference a pointer or an address to yield the value represented by the referred-to object:

***ar[0] == ar[0][0]**

since

***ar == ar[0]**

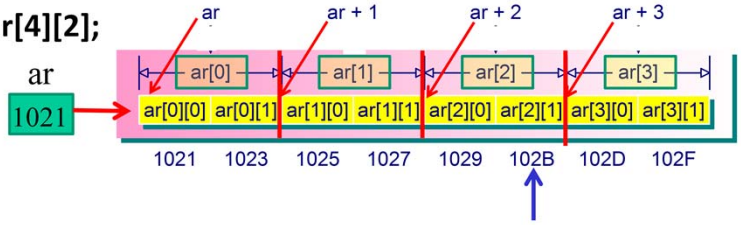
we have

****ar == *(ar[0]) == ar[0][0]**

This is called *double indirection*. Therefore, to obtain **ar[0][0]**, we can achieve it through ****ar** via double dereferencing. Similarly, we can obtain the general formula of the array element at index location **ar[m][n]**, which is given in the next slide.

Multi-dimensional Arrays and Pointers (Cont'd.)

int ar[4][2];



- 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.

19

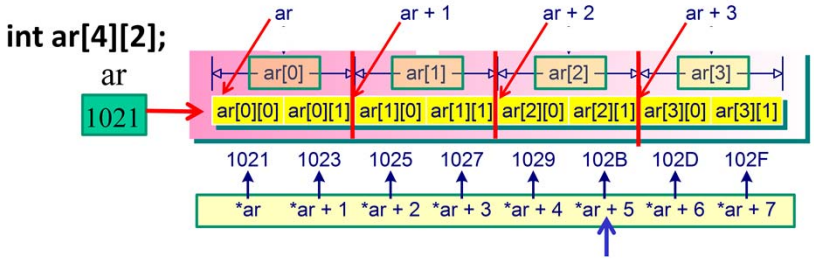
Multi-dimensional Arrays and Pointers

In general, we can represent individual elements of a two-dimensional array as

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

where **m** is the index associated with **ar**, and **n** is the index associated with the sub-array **ar[m]**. When **n** is added to ***(ar+m)**, i.e. ***(ar+m)+n**, it becomes the address of the element of **ar[m][n]**. Applying the * operator to that address gives the content at that address location, i.e. the array element content.

Multi-dimensional Arrays and Pointers (Cont'd.)



- On the other hand, you may also use the following way to access each element of a 2-D array. For example, in the array declaration
  int ar[4][2];
  Let's redefine it as **ar**[**D1**][**D2**], then **D1**=4, **D2**=2:

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

  with **index = m*D2+n**.

  e.g. ar[2][1] = *(*ar + (2*D2 + 1))) = *(*ar + (2***2**+1)) = *(*ar + 5)
  ar[3][0] = *(*ar + (3*D2 + 0))) = *(*ar + (3***2**+0)) = *(*ar + 6)

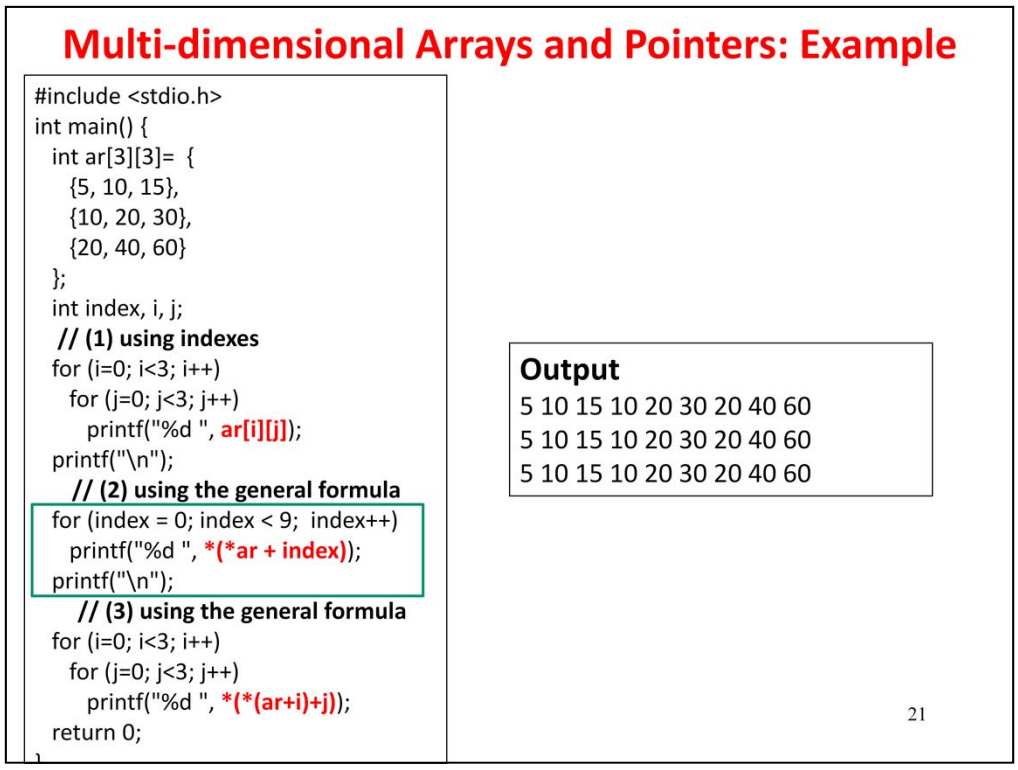
20

Multi-dimensional Arrays and Pointers

By using the similar idea, you may also use the following way to access each element of 2-D array **ar[D1][D2]** (where **ar[4][2]**):

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

with **index = m*D2+n**. ***ar** refers to the array base address. The **index** indicates the relative position of the memory location of the array element from the base address in the memory. **(*ar + index)** refers to the memory location of the element of the array. Using dereferencing, we will then be able to obtain the content of the array element accordingly.



Multi-dimensional Arrays and Pointers: Example

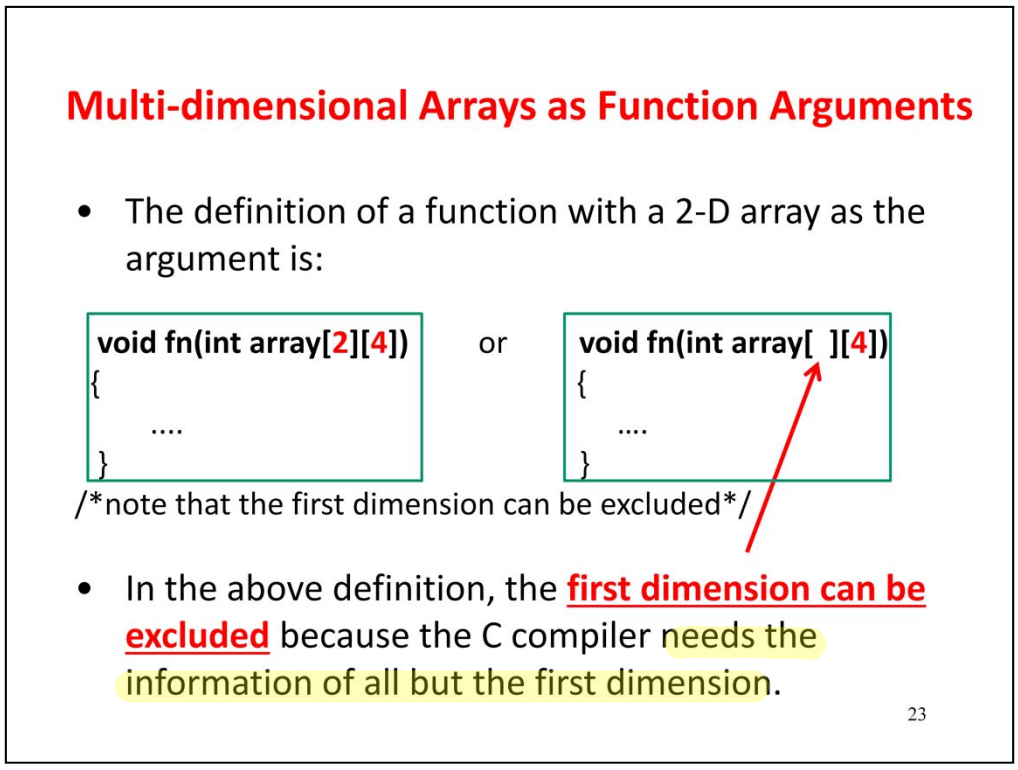
The program aims to print the value of each array element in a 2-D array. In the program, it first initializes each array element of the array **ar[3][3]**. There are three ways to access each element of the array with a **for** loop:

- Using the index approach **ar[i][j]**;
- Using the general formula: ***(*ar + index)**;
- Using the general formula: ***(*(ar+i)+j)**.

Multi-dimensional Arrays

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

22



Multi-dimensional Arrays as Function Arguments

The individual element of a two-dimensional array can be passed as an argument to a function. This can be done by specifying the array name with the corresponding row number and column number.

If an entire multi-dimensional array is to be passed as an argument to a function, this can be done in a similar manner to a one-dimensional array. The definition of a function with a two-dimensional array argument is given as follows:

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

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

Notice that the first dimension of the array can be omitted in the function definition.

Why the First Dimension can be Omitted?

- For example, 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 must be given to the compiler.
- Let's redefine array (i.e. **int array[2][4]**) as

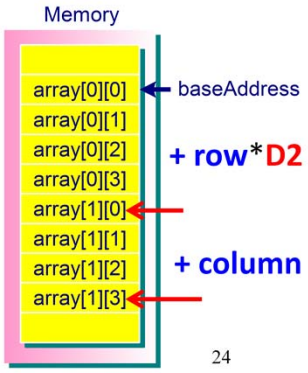
  int array[D1][D2]; // D1=2, D2=4

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

  baseAddress + row * D2 + column

  ==> baseAddress + 1 * 4 + 3

  ==> baseAddress + 7



24

Why the First Dimension can be Omitted?

The first dimension (i.e. the row information) of the array can be excluded in the function definition because C compiler can determine the first dimension automatically. However, the number of columns must be specified.

For example, the assignment statement

array[1][3] = 100;

requests the compiler to compute the address of **array[1][3]** and then places a value of 100 to that address. In order to compute the address, the dimension information must be given to the compiler. Let us redefine **array** as:

int array[D1][D2];

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

baseAddress + row * D2 + column

=> baseAddress + 1 * 4 + 3

=> baseAddress + 7

where the baseAddress is the address pointing to the beginning of **array**. Notice that **D1** is not needed in computing the address.

Therefore, we can omit the value of the first dimension of an array in defining a function, which takes arrays as its arguments.

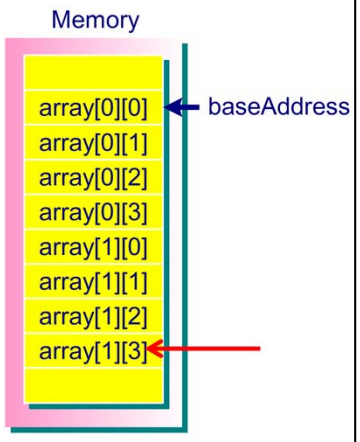
Why the First Dimension can be Omitted? (Cont'd.)

- The **baseAddress** is the address pointing to the beginning of array.
- Because **D1** is **not needed in computing the address**, one 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]**);



25

Why the First Dimension can be Omitted?

Therefore, the function prototype of the function **fn()** can be

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

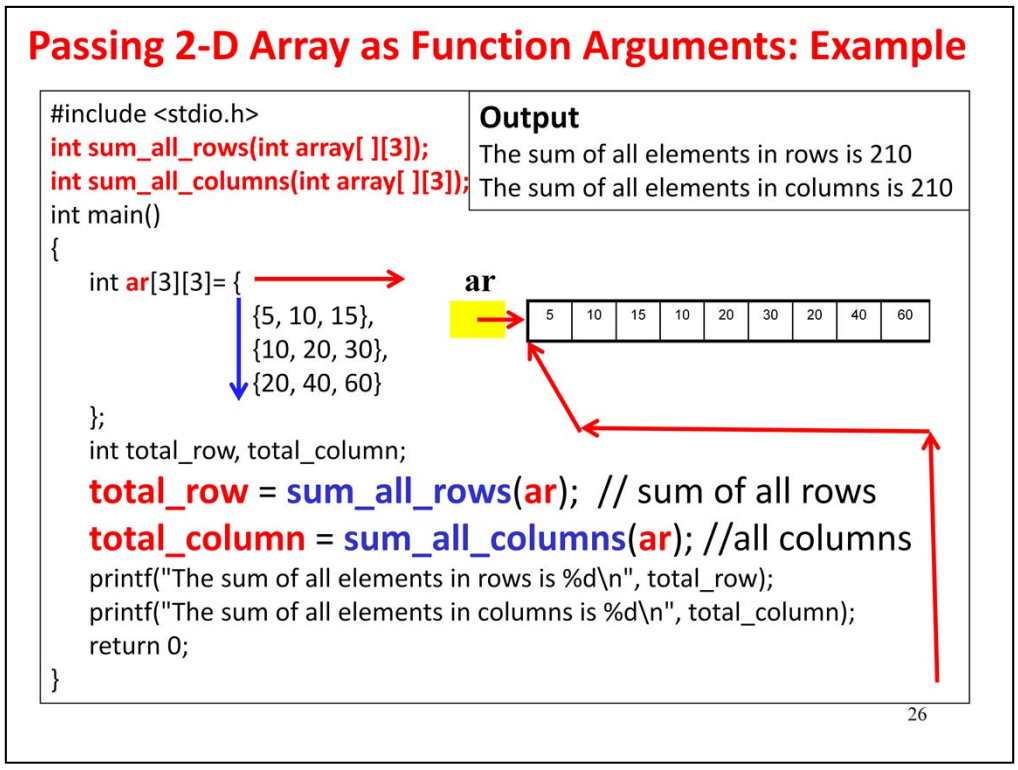
or **void fn(int array[][4]);**

In the **main()** function, it calls **fn()** as follows:

```
int table[2][4];
....
fn(table);
```

Similar to one-dimensional array, the name of the array **table** is specified as the argument without any subscripts in the function call.

The above discussion and definition of two-dimensional arrays can be generalized to the arrays of higher dimensions.



Passing 2-D Array as Function Arguments: Example

The program determines the total sum of all the rows and the total sum of all the columns of a two-dimensional array. Two functions **sum_all_rows()** and **sum_all_columns()** are written to compute the total sums. Both functions take an array as its argument:

int sum_all_rows(int array[][3])

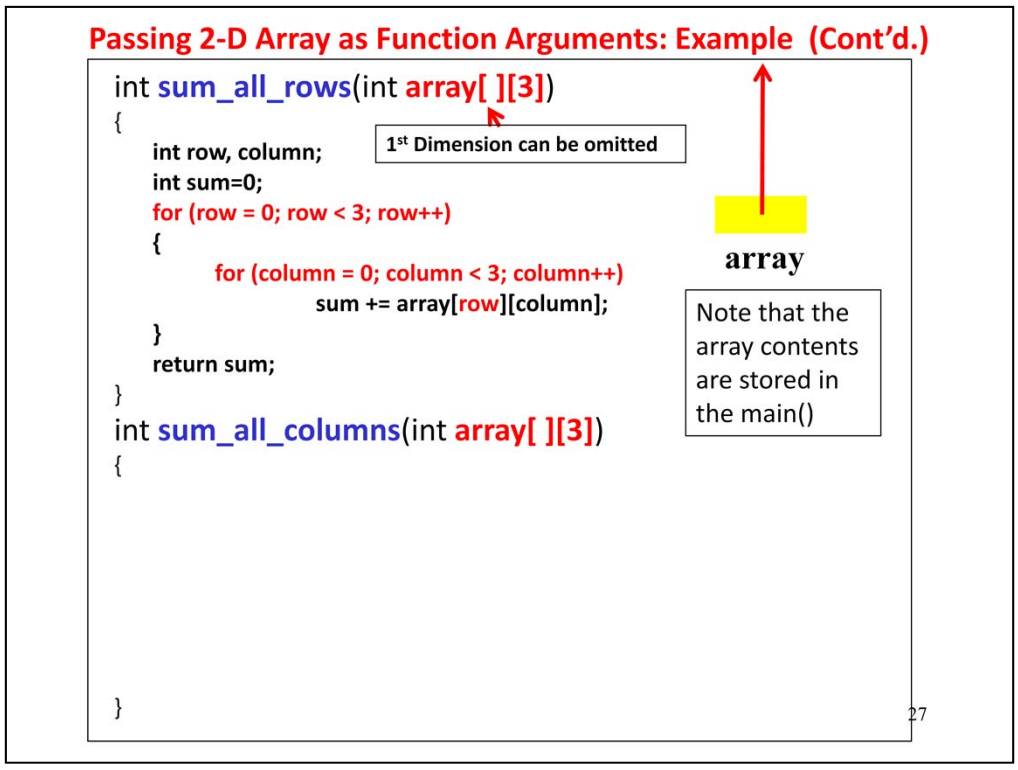
int sum_all_columns(int array[][3])

Note that the first dimension of the array parameter in the function prototype can be omitted. When calling the functions, the name of the array is passed to the calling functions:

total_row = sum_all_rows(ar);

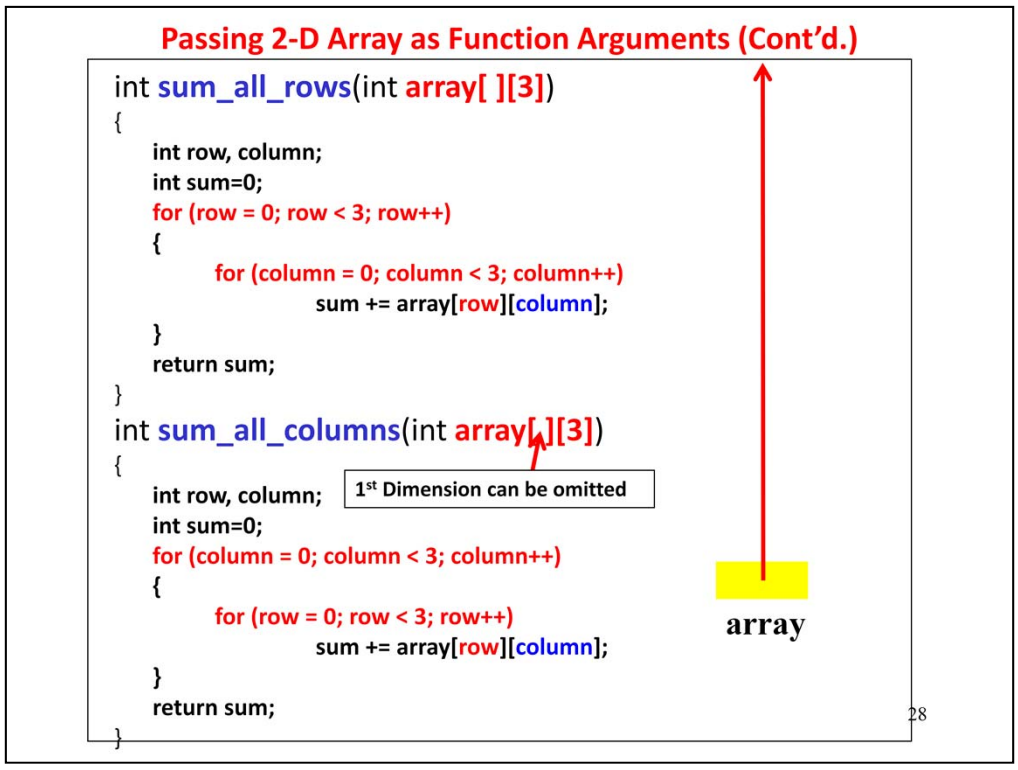
total_column = sum_all_columns(ar);

The total values are computed in the two functions and placed in the two variables **total_row** and **total_column** respectively.



Passing 2-D Array as Function Arguments: Example

Note that the first dimension of the array parameter **array** in the function **sum_all_rows()** can be omitted. A nested **for** loop is used to traverse the 2-dimensional array in order to compute the sum of all rows. The result **sum** is then returned to the calling **main()** function.



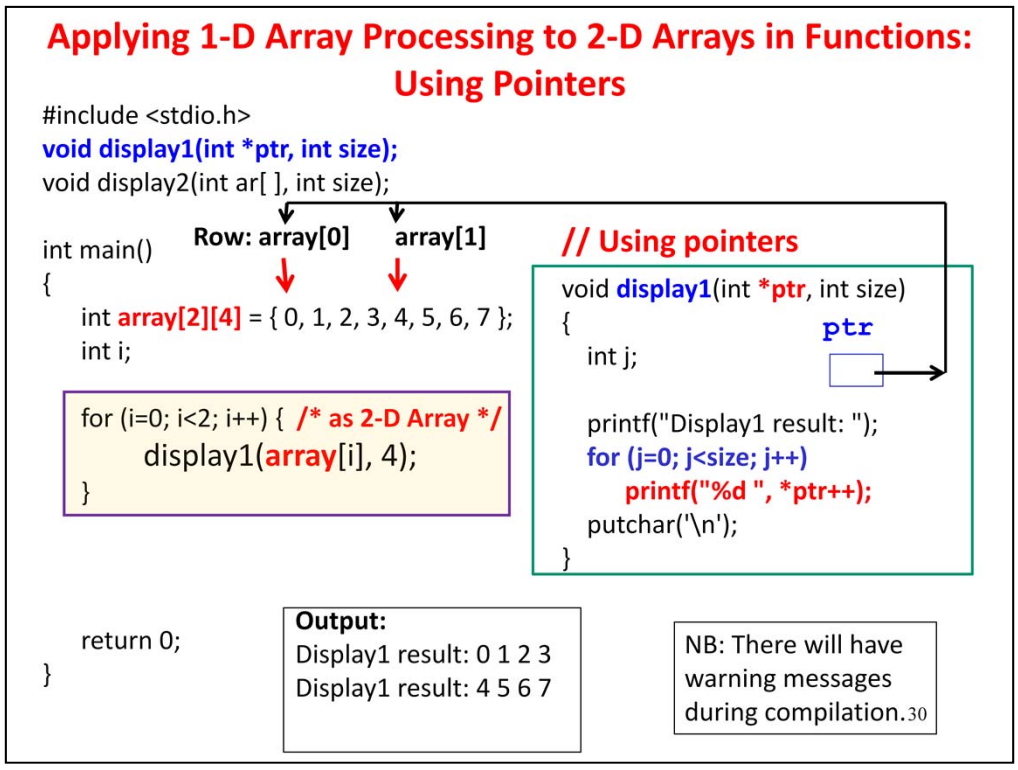
Passing 2-D Array as Function Arguments: Example

Note that the first dimension of the array parameter **array** in the function **sum_all_columns()** can be omitted. A nested **for** loop is used to traverse the 2-dimensional array in order to compute the sum of all columns. The result **sum** is then returned to the calling **main()** function.

Multi-dimensional Arrays

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

29

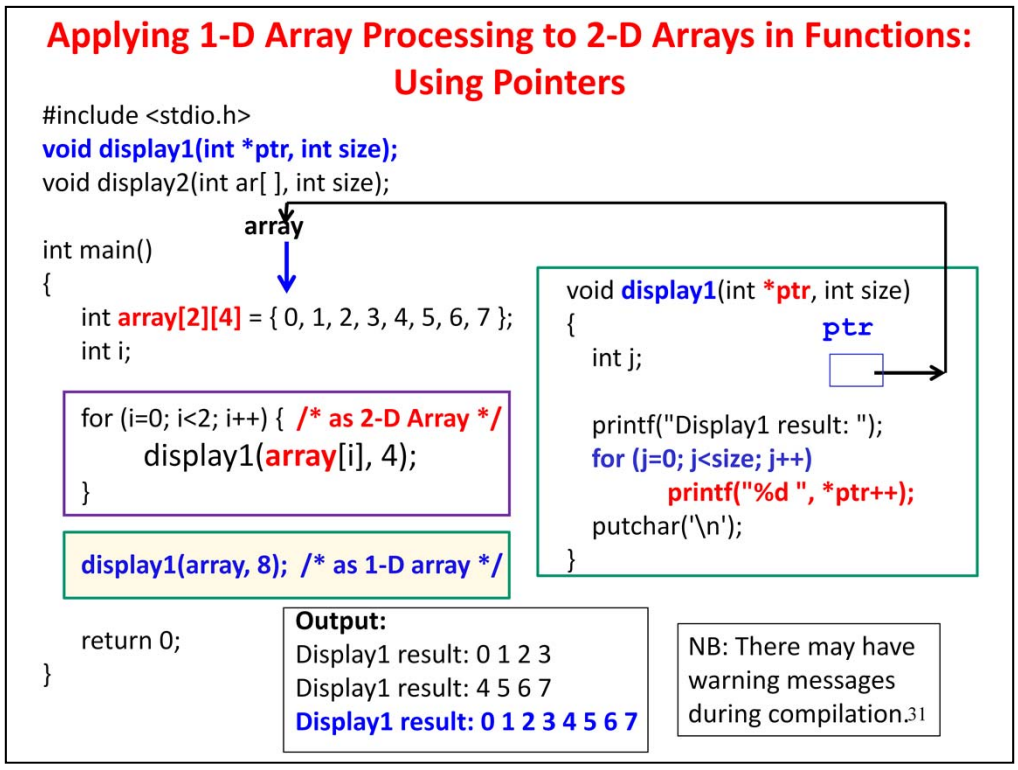


Applying 1-D Array Processing to 2-D Arrays – using Pointers

A function that is written for processing one-dimensional arrays can be used to deal with two-dimensional arrays.

In the program, **array** is an array of 2x4 integers. The function **display1()** is written to access the elements of the array with the specified **size** and prints the contents to the screen.

In **display1()**, it accepts a pointer variable and accesses the elements of the array using the pointer variable. In the **for** loop of the **main()** function, when **i=0**, we pass **array[0]** to **display1()**. **array[0]** corresponds to the address of **array[0][0]** (i.e. **&array[0][0]**). The function then accesses the array starting from the location **array[0][0]**, and prints the 4 elements out to the display as specified in the function. When **i=1**, **array[1]** is passed to **display1()**. Now, **array[1]** corresponds to the address of **array[1][0]** (i.e. **&array[1][0]**). The function then accesses the 4 elements starting from **array[1][0]**, and print the contents of the 4 elements.

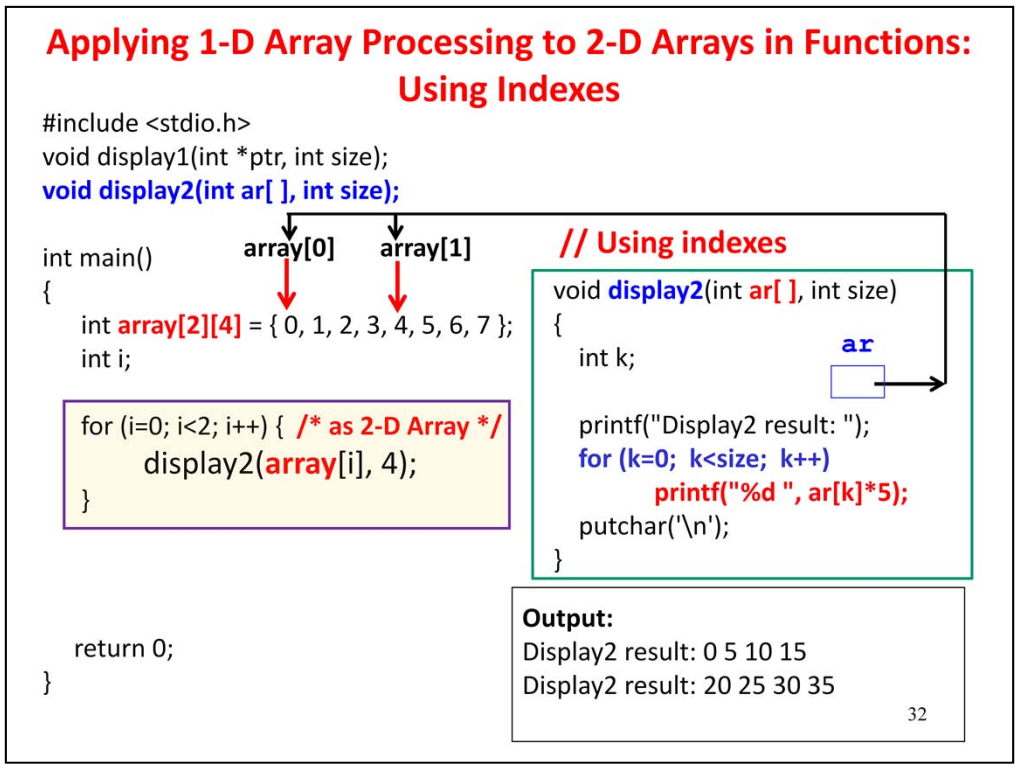


Applying 1-D Array Processing to 2-D Arrays – using Pointers

We can also view **array** as an array of 8 integers. When we pass **array** as an argument to the functions **display1()** with

display1(array, 8);

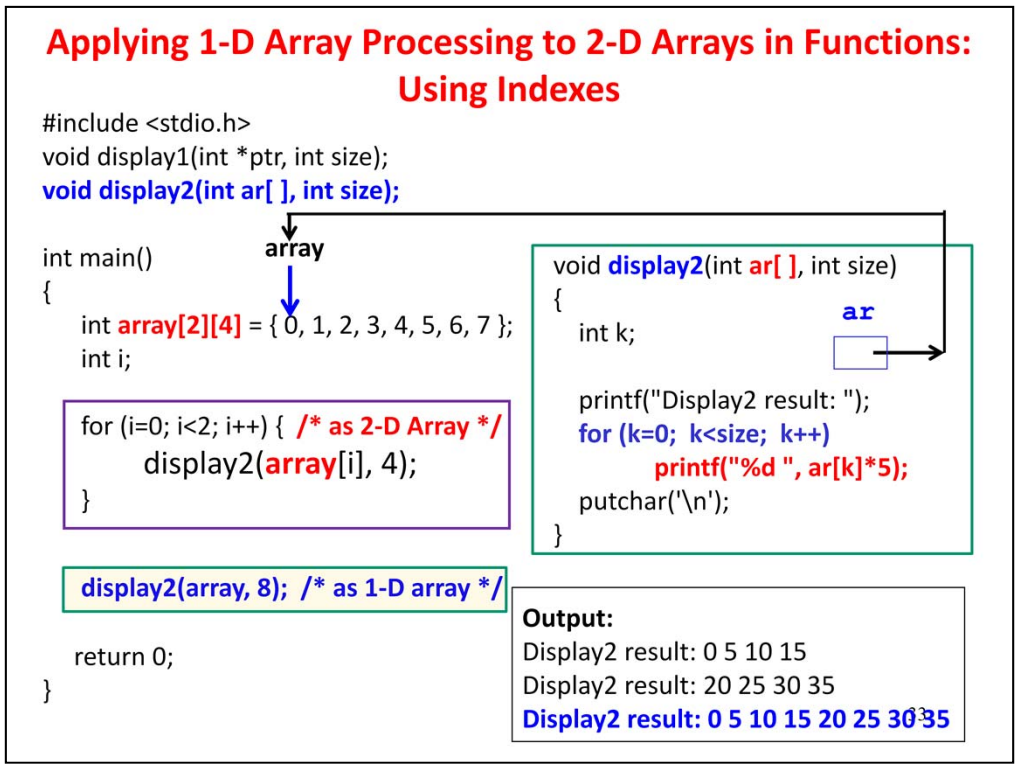
the pointer **ptr** in the function **display1()** is initialized to the address of **array[0][0]**. As a result, ***ptr** corresponds to **array[0][0]**, while ***(ptr+1)** corresponds to **array[0][1]**. Similarly, ***(ptr+4)** corresponds to **array[1][0]**, and so on. Therefore, all the elements of the two-dimensional array can be accessed and printed to the screen.



Applying 1-D Array Processing to 2-D Arrays – using Indexes

In the program, **array** is an array of 2x4 integers. The function **display2()** is written to access the elements of the array with the specified **size** and prints the contents to the screen.

In **display2()**, it accepts the array pointer and uses array index to access the elements of the array. In the **for** loop of the **main()** function, when **i=0**, we pass **array[0]** to **display2()**. **array[0]** corresponds to the address of **array[0][0]** (i.e. **&array[0][0]**). The function then accesses the 4 elements of the array starting from the location **array[0][0]**, and prints the results to the display as specified in the function. When **i=1**, **array[1]** is passed to **display2()**. Now, **array[1]** corresponds to the address of **array[1][0]** (i.e. **&array[1][0]**). The function then accesses the 4 elements starting from **array[1][0]**, and prints the results according to the function.

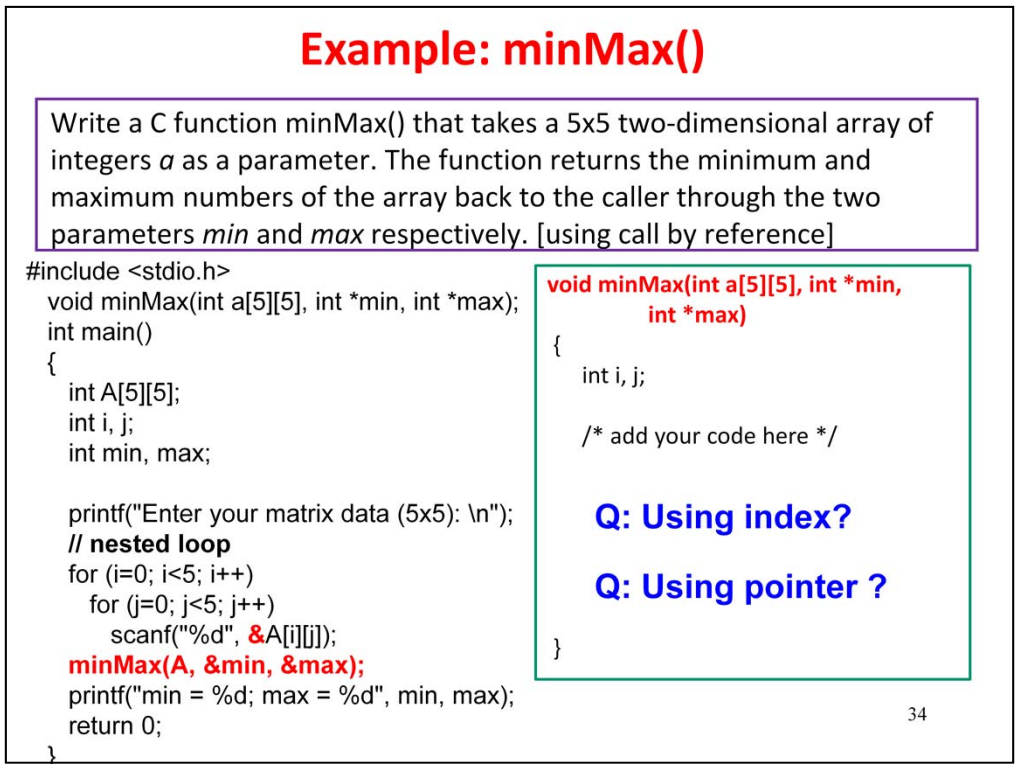


Applying 1-D Array Processing to 2-D Arrays – using Indexes

We can also view **array** as an array of 8 integers. When we pass **array** as an argument to the function **display2()** with

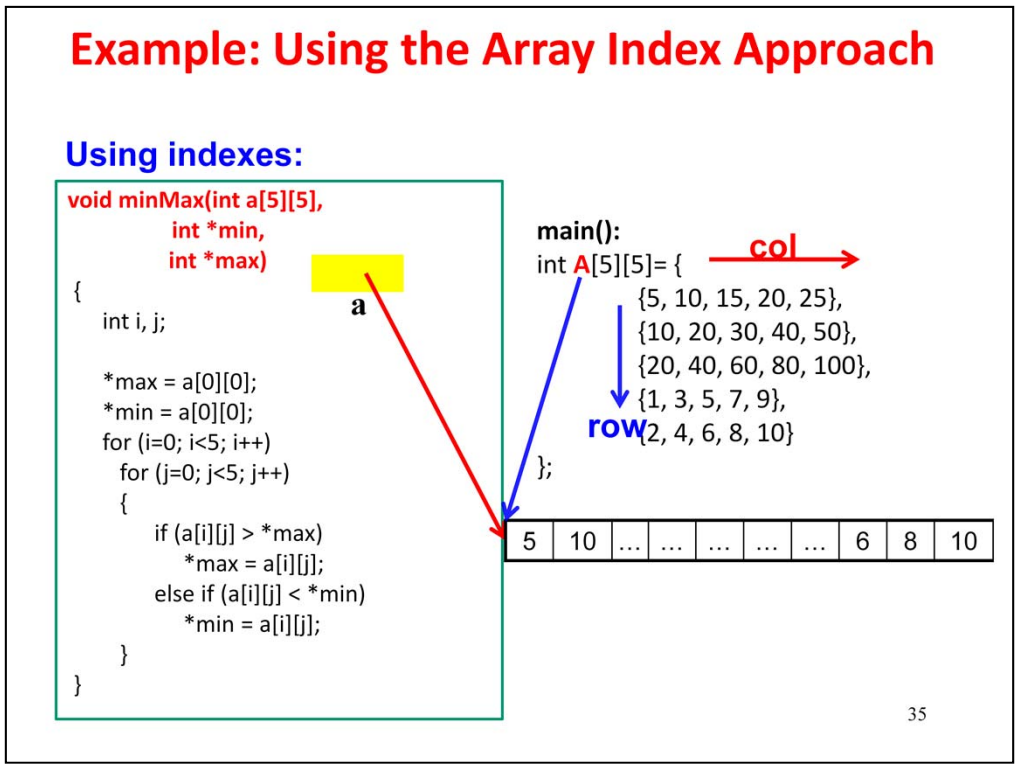
display2(array, 8);

the array **ar** in the function **display2()** is initialized to the address of **array[0][0]**. As a result, **ar[0]** corresponds to **array[0][0]**, while **ar[1]** corresponds to **array[0][1]**. Similarly, **ar[4]** correspond to **ar[1][0]**, and so on. Therefore, all the elements of the two-dimensional array can be accessed and printed to the screen.



Applying 1-D Array Processing to 2-D Arrays – Example

In this application example, you are required to 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 back to the caller through the two parameters *min* and *max* respectively. Call by reference is used for passing the results on maximum and minimum numbers to the calling function. You may use the array index approach or pointer variable approach for processing the 2-D array.



Example: Using the Array Index Approach

In this implementation using the array index approach, a nested **for** loop is used to process the 2-D array in the function. It first initializes the ***max** and ***min** to contain the first array element number. The 2-D array **a** is processed using indexes to access and comapre all the elements stored in the array with ***max** and ***min**. After the processing of the 2-D array, the maximum and minimum numbers are determined and stored at ***max** and ***min** respectively. The implementation using indexes is quite straightforward.

Example: Using Pointer Variable Approach

Using pointers

```
void minMax2(int a[5][5], int *min,
int *max)
{
  int i;
  int *p;
  p=a;

  *max = *p;
  *min = *p;


   /* Write your code here */


}
```

36

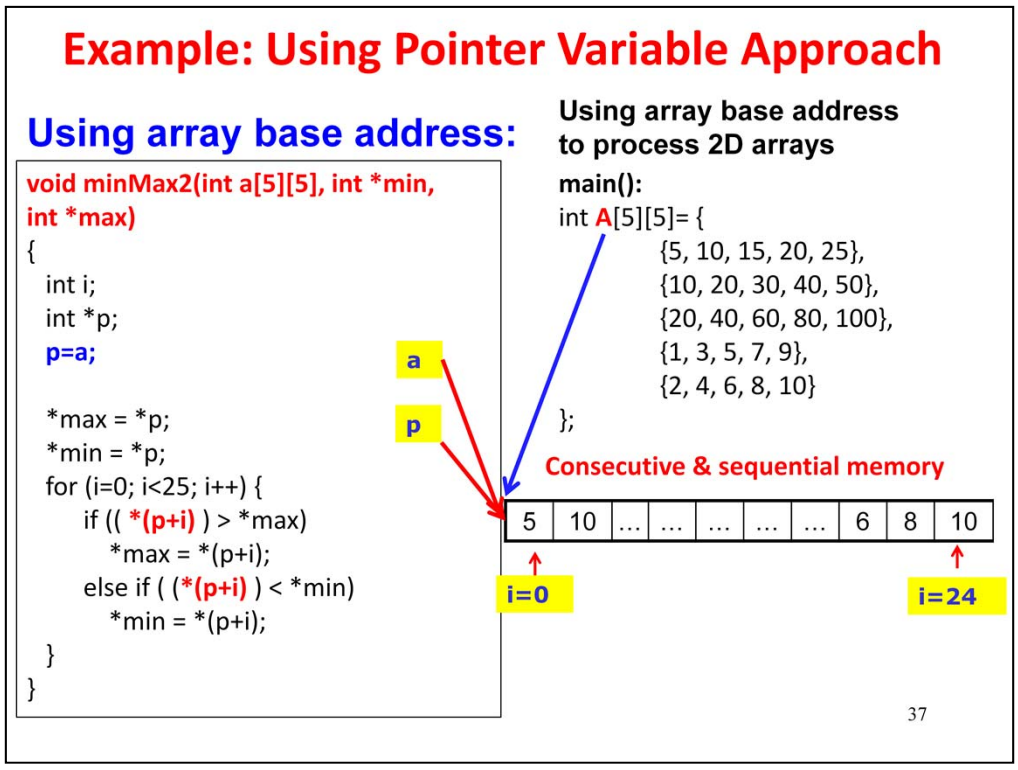
Example: Using the Pointer Variable Approach

In this implementation, it uses the pointer variable approach. It first initializes the ***max** and ***min** to contain the first array element number. It is implemented using the pointer variable **p** by using the following statement:

```
int *p;
```

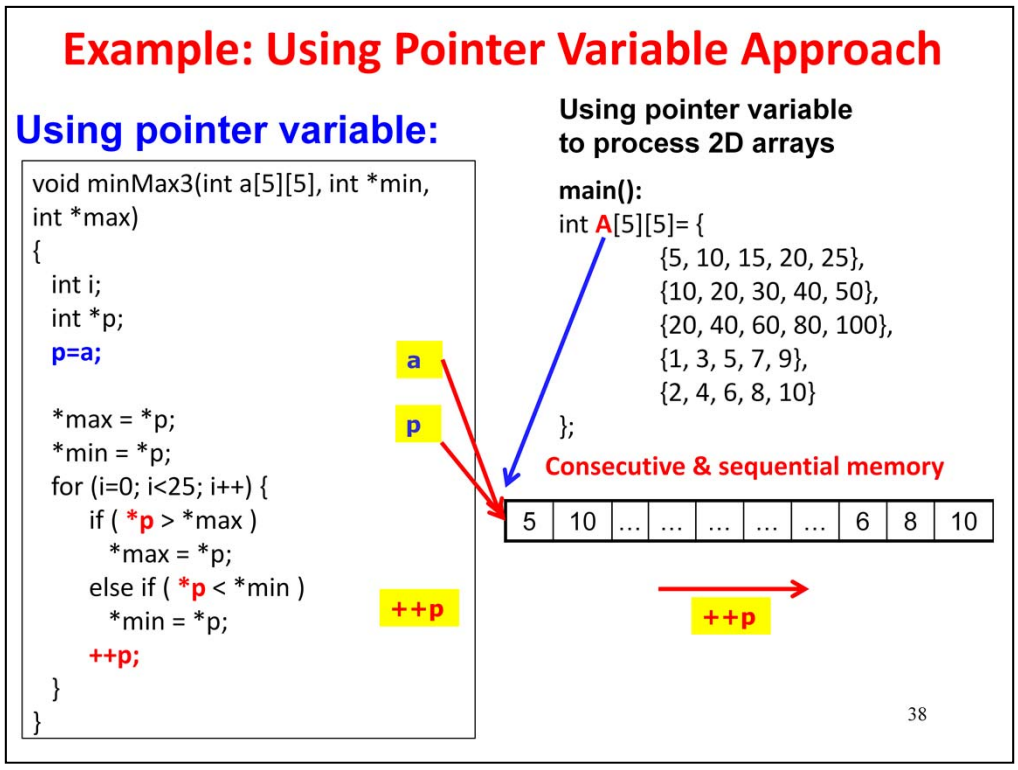
Then, assign the array **a** to the pointer variable **p**, and initializes the values for ***max** and ***min** by the first element of the array:

```
p=a;
*max = *p;
*min = *p;
```



Example: Using the Pointer Variable Approach

A **for** loop is used to traverse and process the 2-D array by treating it as a 1-D array. In this approach, we use the **base address** as the reference. The index variable **i** is used to update the pointer variable to the corresponding array memory location **p+i**, and retrieves the array element content via ***(p+i)**. Each array element content will be compared with the ***max** and ***min** to determine the maximum and minimum numbers respectively. At the end of the processing, the maximum and minimum numbers are determined and stored at ***max** and ***min** respectively. The values are returned to the calling function via call by reference.



Example: Using the Pointer Variable Approach

Different from the previous approach using the base address, we can also use the pointer variable approach by updating the **pointer variable** directly. Similarly, a **for** loop is used to traverse and process the 2-D array by treating it as a 1-D array. The index variable is not needed in this approach. We can update the pointer variable to the corresponding array memory location using **p++**, and retrieves the array element content via ***p**. Each array element content will be compared with the ***max** and ***min** to determine the maximum and minimum numbers respectively. At the end of the processing, the maximum and minimum numbers are determined and stored at ***max** and ***min** respectively. The values are returned to the calling function via call by reference.

Multi-dimensional Arrays

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

39

Sizeof Operator and Array

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

  sizeof (**operand**)

  or

  sizeof operand

- The **operand** can be:

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

40

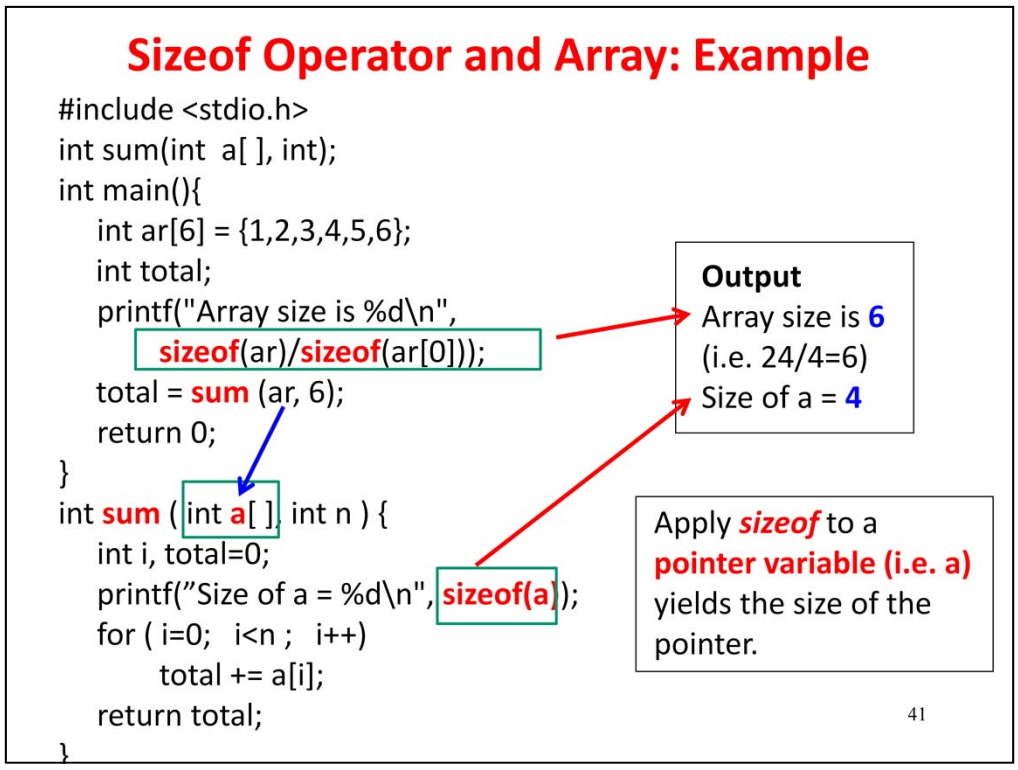
Sizeof Operator and Array

sizeof is an operator which gives the size (in bytes) of its operand. Its syntax is

sizeof(operand)

or **sizeof operand**

The **operand** can either be a type enclosed in parenthesis or an expression. We can also use it with arrays.



Sizeof Operator and Array: Example

In the **main()** function of the program, the **sizeof** operator returns the number of bytes of the array. The second **sizeof** operator returns the number of bytes of each element in the array. Therefore, the number of elements can be calculated by dividing the size of the array by the size of each element in the array. In this case, the array size is 24/6 which gives the value of 6.

However, in the function **sum()**, the **sizeof** operator returns the number of bytes for the array **a**. It is in fact a pointer which contains the address of the argument passed in from the calling function. As a pointer has 4 bytes, the size of **a** is 4.

Quiz: What is the output?

42

Quiz: What is the Output?

```
#include <stdio.h>
int main( ){
  char ar1[4][6] = {
            {'P', 'e', 't', 'e', 'r'},
            {'J', 'o', 'h', 'n', 'n', 'y'},
            {'M', 'a', 'r', 'y'},
            {'K', 'e', 'n', 'n', 'y'}
          };
  char *ar2;
  char *ar3;
  ar2 = &ar1[1][1];
  ar3 = ar1[1];
  printf("Output1 = %c\n", ar1[1][4]);
  printf("Output2 = %c\n", *(ar2-3));
  printf("Output3 = %c\n", *(ar3+1));
  printf("Output4 = %c\n", *(ar3+6));
  return 0;
}
```

empty → n → r → o → M

43

Quiz: What is the output?

Determine the output of the program.

Quiz: What is the Output?

```
#include <stdio.h>
int main( ){
  char ar1[4][6] = {
            {'P', 'e', 't', 'e', 'r'},
            {'J', 'o', 'h', 'n', 'n', 'y'},
            {'M', 'a', 'r', 'y'},
            {'K', 'e', 'n', 'n', 'y'}
          };
  char *ar2;
  char *ar3;
  ar2 = &ar1[1][1];
  ar3 = ar1[1];
  printf("Output1 = %c\n", ar1[1][4]);
  printf("Output2 = %c\n", *(ar2-3));
  printf("Output3 = %c\n", *(ar3+1));
  printf("Output4 = %c\n", *(ar3+6));
  return 0;
}
```

44

Note:
ar1[0][5] is '\0'.
ar1[2][4] is '\0'.
This will be discussed in Character String on array of strings.

Output1 = n
Output2 = r
Output3 = o
Output4 = M

Quiz: What is the output?

The output are:

Output1 = n
Output2 = r
Output3 = o
Output4 = M

Self-Practice Question

45

Q: Explain how the addition of 1 to every element of the two dimensional array 'array' is done in the following program.

```
#include <stdio.h>
void  add1(int ar[], int size);
int main(){
     int  array[3][4];
     int h,k;
     for (h = 0; h < 3; h++)
        for (k = 0; k < 4; k++)
           scanf("%d", &array[h][k]);

     for (h = 0; h < 3; h++) /* line a */
        add1(array[h], 4);

     for (h = 0; h < 3; h++) {
        for (k = 0; k < 4; k++)
           printf("%10d", array[h][k]);
           putchar('\n');
     }
     return 0;
}
void add1(int ar[], int size)
{
     int k;
     for (k = 0; k < size; k++)
        ar[k]++;
}
```

Q: What if the for statement at 'line a' is replaced by this statement:

add1(array[0], 3*4);

```
#include  <stdio.h>
void  add1(int ar[], int size);
int main(){
    int  array[3][4];
    int h,k;
    for (h = 0; h < 3; h++)
       for (k = 0; k < 4; k++)
          scanf("%d", &array[h][k]);

    for (h = 0; h < 3; h++)    /* line a */
       add1(array[h], 4);
    // replaced by add1(array[0], 3*4);

    for (h = 0; h < 3; h++) {
       for (k = 0; k < 4; k++)
          printf("%10d", array[h][k]);
          putchar('\n');
    }
    return 0;
}
void add1(int ar[], int size)
{
    int k;
    for (k = 0; k < size; k++)
       ar[k]++;
}
```

Q: Explain how the addition of 1 to every element of the two dimensional array 'array' is done in the following program.

```
#include <stdio.h>
void  add1(int ar[], int size);
int main(){
     int  array[3][4];
     int h,k;
     for (h = 0; h < 3; h++)
        for (k = 0; k < 4; k++)
           scanf("%d", &array[h][k]);

     for (h = 0; h < 3; h++) /* line a */
        add1(array[h], 4);

     for (h = 0; h < 3; h++) {
        for (k = 0; k < 4; k++)
           printf("%10d", array[h][k]);
           putchar('\n');
     }
     return 0;
}
void add1(int ar[], int size)
{
     int k;
     for (k = 0; k < size; k++)
        ar[k]++;
}
```

array[3][4]={1,2,3,4, 5,6,7,8, 9,10,11,12}

Output:

```
2    3    4    5
6    7    8    9
10   11   12   13
```

The function add1() have two parameters. The first one is an array address and the second one is the size of the array. So the function adds 1 to every element of the one dimensional array.

When the function is called in the for statement at line a by

add1(array[h], 4);

array[h] is an one dimensional array of 4 integers. It is the (h+1)th row of the two dimensional array 'array'. In fact, array[h] is the address of the first element of the (h+1)th row. So every function call works on one row of the two dimensional array.

Q: What if the for statement at 'line a' is replaced by this statement:

add1(array[0], 3*4);

```
#include  <stdio.h>
void  add1(int ar[], int size);
int main(){
    int  array[3][4];
    int h,k;
    for (h = 0; h < 3; h++)
       for (k = 0; k < 4; k++)
          scanf("%d", &array[h][k]);

    for (h = 0; h < 3; h++)    /* line a */
       add1(array[h], 4);
    // replaced by add1(array[0], 3*4);

    for (h = 0; h < 3; h++) {
       for (k = 0; k < 4; k++)
          printf("%10d", array[h][k]);
          putchar('\n');
    }
    return 0;
}
void add1(int ar[], int size)
{
    int k;
    for (k = 0; k < size; k++)
       ar[k]++;
}
```

array[0]

↓

array[3][4]={1,2,3,4,5,6,7,8,9,10,11,12}

Output:

```
2    3    4    5
6    7    8    9
10   11   12   13
```

When the for statement at line a is replaced by add1(array[0], 3*4), it is passing the address of the first element of the first row to add1() and telling the function that the array size is 12. So add1() works on an one dimensional array starting at array[0] and with 12 elements.