5.2 Two-dimensional Arrays

1

Two-dimensional Arrays

1. In this lecture, we discuss the Two-dimensional Arrays data structure in C.

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.

2

Why Learning Two-dimensional (or Multi-dimensional) Arrays?

1. We have discussed one-dimensional arrays in which only a single index (or subscript) is needed to access a specific element of the array.
2. The number of indexes that are used to access a specific element in an array is called the **dimension** of the array.
3. Arrays that have more than one dimension are called multi-dimensional arrays.
4. Here, we focus mainly on two-dimensional arrays. We may use two-dimensional arrays to represent data stored in tabular form.
5. The concepts discussed in one-dimensional arrays can be extended to multi-dimensional arrays.
6. 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

3

Two-dimensional Arrays

1. We first discuss two-dimensional array declaration, initialization and operations.

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:

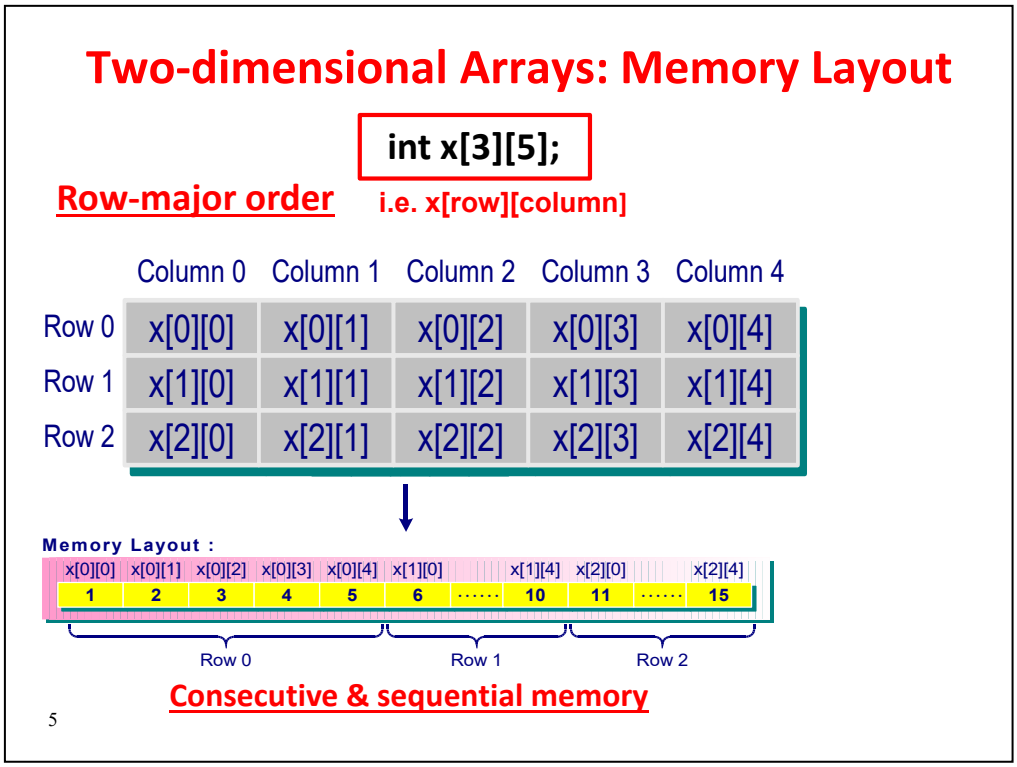
```
char x[3][4][5]; // a 3-element array of 4-element arrays of
                 5-element arrays
```

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

4

Two-dimensional (or Multi-dimensional) Arrays Declaration

1. A two-dimensional array can be declared as **int x[3][5];** which is a 3-element array of 5-element arrays.
2. Two indexes are needed to access each element of the array.
3. Similarly, a three-dimensional array can be declared as **char x[3][4][5];** In this case, three indexes are used to access a specific element of the array. ANSI C standard supports arrays with a minimum of 6 dimensions.



Two-dimensional Arrays: Memory Layout

1. 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.
2. 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.
3. 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. For example, **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.
4. A two-dimensional array is stored in **row-major** order in the memory.
5. Note that the memory storage of the two-dimensional array **x[3][5]** is consecutive and sequential.

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}};
```

6

Initializing Two-dimensional Arrays

1. For the initialization of two-dimensional arrays, each row of data is enclosed in braces as shown in the two-dimensional array **x**:

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

2. 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.
3. Since the inner braces are optional, a two-dimensional array can be initialized as **int x[2][2] = { 1,2,6,7 };**
4. An array can also be initialized partially, for example, **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.
5. For the following statement, **int exam[3][3] = { 1,2,4,5,7 };** the two-dimensional array will be initialized as **int exam[3][3] = { {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 Two-dimensional Arrays – Sum of Rows

1. The program determines the sum of rows of two-dimensional arrays. It uses indexes to traverse each element of the two-dimensional **array**.
2. In the program, the array is first initialized.
3. 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.
4. A nested **for** loop is used to access the individual elements of the array.
5. 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 with another **for** loop and add them up to give the sum of rows.
6. 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 – 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
```

8

Operations on Two-dimensional Arrays – Sum of Columns

1. The program determines the sum of columns of two-dimensional arrays.
2. It uses indexes to traverse each element of the two-dimensional **array**. In the program, the array is first initialized.
3. 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 with another **for** loop and add them up to give the sum of columns.
4. Again note that the first dimension of an array is row and the second dimension is column. It is row-major.

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

9

Two-dimensional Arrays

1. Here, we discuss two-dimensional arrays and pointers.

Two-dimensional Arrays and Pointers

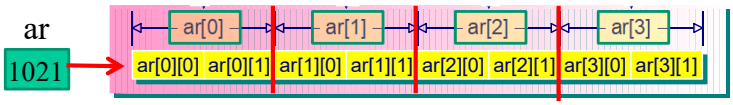
- Two-dimensional array variable declaration:

```
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



10

Two-dimensional Arrays and Pointers

1. 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 */
```

2. The array variable **ar** is the address of the first element of the array.

Two-dimensional Arrays and Pointers

- Two-dimensional array variable declaration:

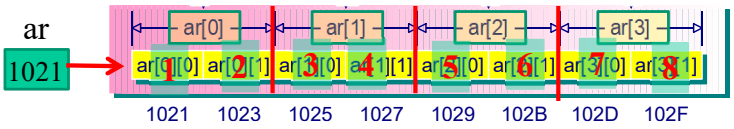
```
int ar[4][2];      /* ar is an array of 4 elements;
                      each element is an array of 2 ints */
```

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.

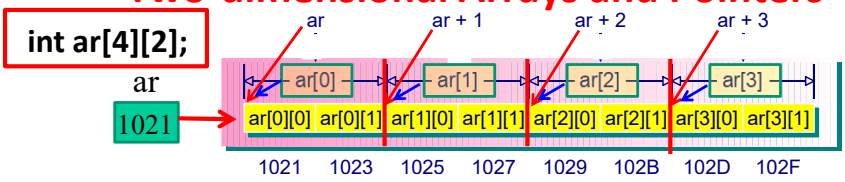


11

Two-dimensional Arrays and Pointers

1. The memory of the two-dimensional array is organized in a sequential manner.
2. As such, the values of the two-dimensional arrays are stored sequentially in the memory.

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)

12

Two-dimensional Arrays and Pointers

1. The memory layout of the two-dimensional array is shown with its associated pointers.
2. The first element is an array of 2 integers. **ar** is the address of a two-integer sized object.
3. Therefore, we have

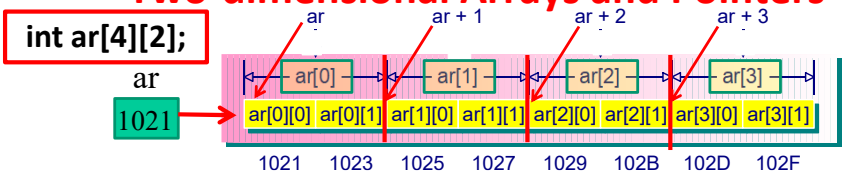
   ar == &ar[0] **ar + 1 == &ar[1]**
   ar + 2 == &ar[2] **ar + 3 == &ar[3]**

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

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

6. 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.
7. 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.

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]**

13

Two-dimensional Arrays and Pointers

1. **Dereferencing** a pointer or an address (by applying the dereferencing ***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]**

2. Similarly, we have

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

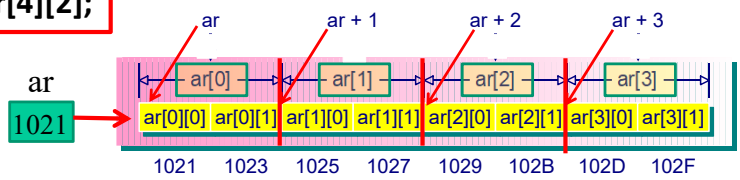
   ar[1] == &ar[1][0], using dereferencing we have ***ar[1] == ar[1][0]**

   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]**

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**)

14

Two-dimensional Arrays and Pointers

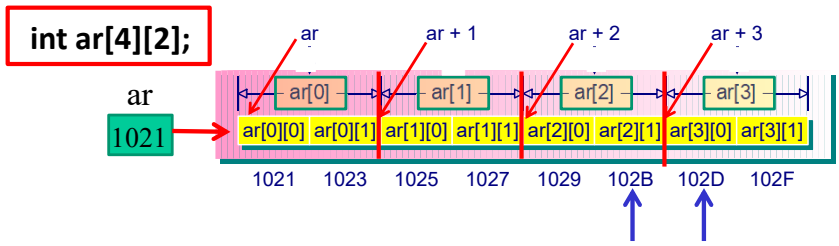
1. Therefore, we have

   ***ar[0]** refers to the value stored in **ar[0][0]**
2. Since

   ***ar** refers to the value of its first element, **ar[0]**
3. We have

   ****ar == *(ar[0]) == ar[0][0]**
4. This is called *double indirection*. Therefore, to obtain **ar[0][0]**, we can achieve it through ****ar** via double dereferencing.

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]

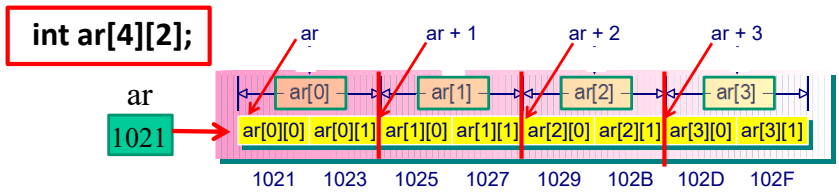
  Note: you are not required to remember the calculation on deriving the general formula.

15

Two-dimensional Arrays and Pointers

1. After some calculations using **double** dereferencing as shown above, we can represent each individual element 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]**.
2. This can be interpreted as follows: First, dereferencing ***(ar+m)** to get the address of the inner array of **ar** according to the row number **m**. Then, by adding the column number **n** 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.
3. Note that you are not required to remember the calculation on deriving the general formula.

Two-dimensional Arrays and Pointers



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)

16

Two-dimensional Arrays and Pointers

1. There are two ways to access each element of the array :
   a) First - using the indexing approach: **ar[m][n]** with indexes **m** and **n**;
   b) Second - using the general formula: ***(*(ar+m)+n)** for **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;
}
```

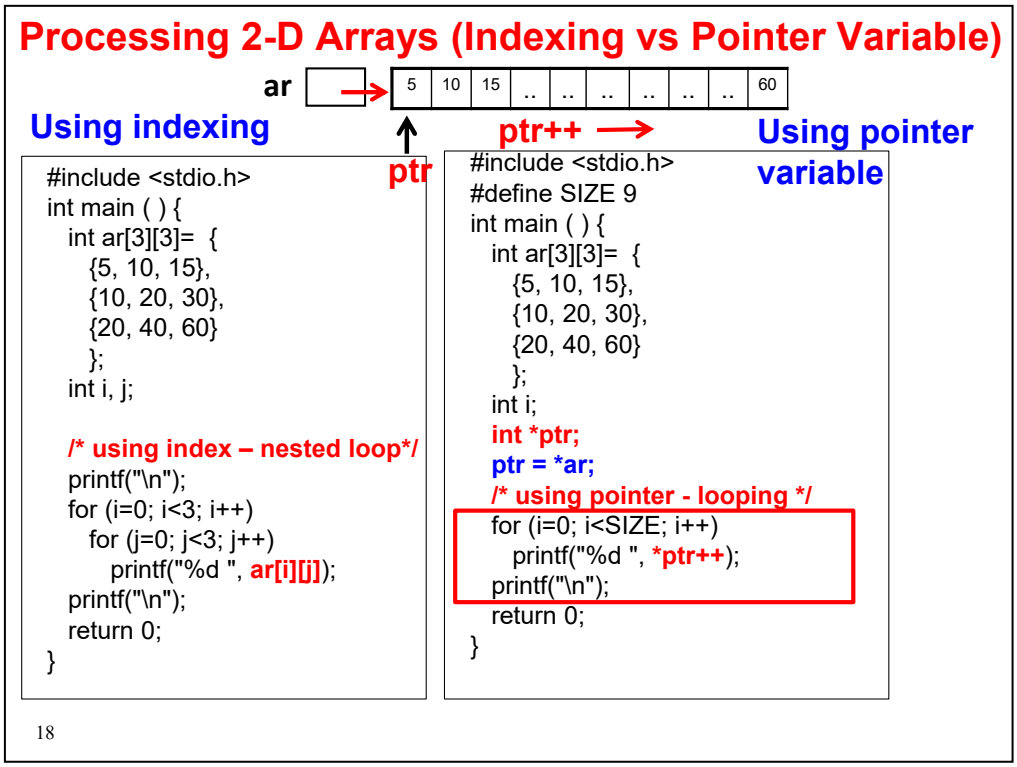
Output

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

17

Processing Two-dimensional Arrays: Example

1. The program aims to print the value of each array element in a two-dimensional array.
2. In the program, it first initializes each array element of the two-dimensional array **ar[3][3]**.
3. There are two ways to access each element of the array with a nested **for** loop:
    a) Using the indexing approach or
    b) Using pointers with the general formula



Processing Two-dimensional Arrays: Indexing vs Pointer Variable

1. For processing two-dimensional arrays, you may use array index or pointer variable for processing each element of the array.
2. When using the index approach, indexes (e.g. ar[i][j]) are used to access each individual element of a two-dimensional array.
3. When using pointer variable approach, a pointer variable is declared and assigned with the array address, i.e. **ptr = *ar;** It is then used to traverse each element of a two-dimensional array by incrementing the pointer variable to access the content of each element of the array.

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

19

Two-dimensional Arrays

1. Here, we discuss using two-dimensional arrays as function arguments.

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.

20

Two-dimensional Arrays as Function Arguments

1. 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.
2. If an entire two-dimensional array is to be passed as an argument to a function, this can be done in a similar manner to an one-dimensional array.
3. The definition of a function with a two-dimensional array argument is given as follows: **void function(int array[2][4])** or **void function(int array[][4])**.
4. Note that the first dimension of the array can be omitted in the function definition.

Why the First Dimension can be Omitted?

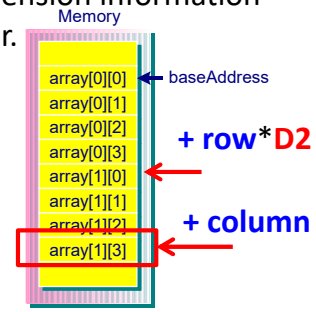
- For example, in the assignment operation: **array[1][3]** = 100; requests the **compiler** to compute the address of **array[1][3]** and then place 100 to that address.
- In order to compute the address, the dimension information of the array must be given to the compiler.
- Let's redefine **array**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



21

Why the First Dimension can be Omitted?

1. The first dimension (i.e. the row information) of an 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.
2. 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];**
3. 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**.
4. Note that **D1** is not needed in computing the address.

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

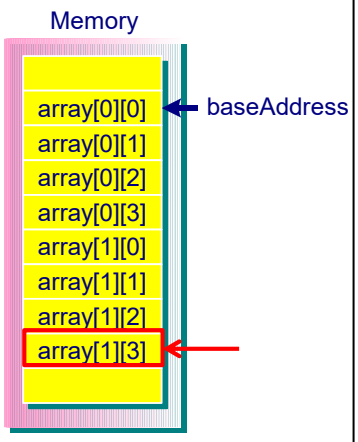
- Since **D1** is **not needed in computing the address**, we can omit the first dimension value in defining a function which takes arrays as its formal arguments.

- Therefore, the prototype of the function could be:

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

  or

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



22

22

Why the First Dimension can be Omitted?

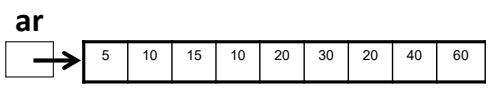
1. Since D1 is not needed in computing the address, we can omit the value of the first dimension of an array in defining a function, which takes arrays as its formal arguments.
2. Therefore, the function prototype of the function **function()** can be

   void function(int array[2][4]); or

   void function(int array[][4]);

Passing 2-D Array as Function Arguments: Example

```
#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;
}
```



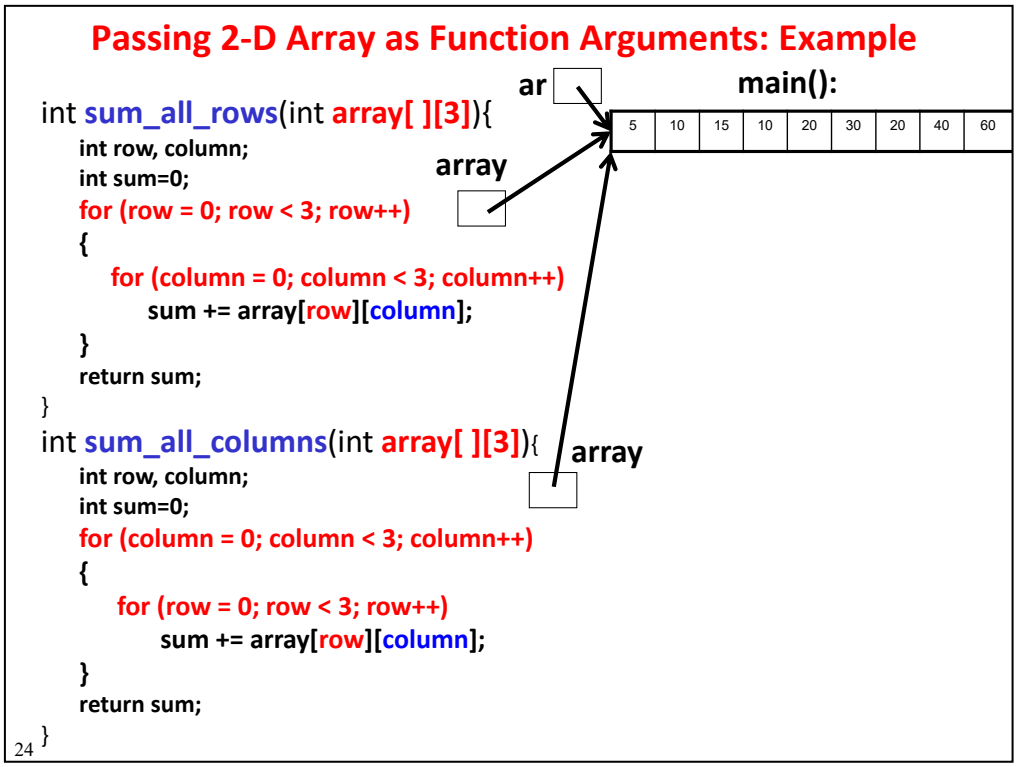
Output

The sum of all elements in rows is 210
The sum of all elements in columns is 210

23

Passing Two-dimensional Array as Function Arguments: Example

1. The program determines the total sum of all the rows and the total sum of all the columns of a two-dimensional array.
2. The 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]);** and **int sum_all_columns(int array[][3]);** Note that the first dimension of the array parameter in the function prototype can be omitted.
2. When calling the functions, the name of the array is passed to the calling functions: **total_row = sum_all_rows(ar);** and **total_column = sum_all_columns(ar);**
3. The total values are computed in the two functions and placed in the two variables **total_row** and **total_column** respectively.



Passing Two-dimensional Array as Function Arguments: Example

1. Note that the first dimension of the array parameter **array** in the function **sum_all_rows()** can be omitted.
2. 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.
3. Similarly, the first dimension of the array parameter **array** in the function **sum_all_columns()** can be omitted.
4. The function **sum_all_columns()** is implemented similarly to **sum_all_rows()**.

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

25

Two-dimensional Arrays

1. Here, we discuss how to apply one-dimensional array to process two-dimensional 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

26

Applying One-dimensional Array to Process Two-dimensional Arrays – using Pointers

1. A function is written for processing two-dimensional arrays using one-dimensional arrays.
2. 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.
3. In **display1()**, it accepts a pointer variable and accesses the elements of the array using the pointer variable.
4. 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 to the screen as specified in the function.
5. 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]**).
6. The function then accesses the 4 elements starting from **array[1][0]** and prints the contents of the 4 elements.
7. Note that the compilation of this program may generate a warning message.

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;
}
```

array

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

27

Applying One-dimensional Array to Process Two-dimensional Arrays – using Pointers

1. We can also view **array** as an array of 8 integers. When we pass **array** as an argument to the function **display1()** with **display1(array, 8);** the pointer **ptr** in the function **display1()** is referred to the address of **array[0][0]**.
2. In the function **display1()**, dereferencing the pointer variable ***ptr** corresponds to **array[0][0]**, dereferencing ***(ptr+1)** corresponds to **array[0][1]** and so on.
3. The function then accesses the 8 elements starting from **array[0][0]** and prints the contents of the 8 elements to the screen.
4. Therefore, all the elements of the two-dimensional array can be accessed via the pointer variable **ptr** and printed to the screen.

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);
    }

    return 0;
}
```

array[0] array[1]

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

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

ar

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

28

Applying One-dimensional Array to Process Two-dimensional Arrays – using Indexing

1. The function **display2()** is written to access the elements of the array with the specified **size** and prints the contents to the screen. It accepts the array pointer and uses array index to access the elements of the array.
2. 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 screen as specified in the function.
3. 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]**).
4. The function then accesses the 4 elements starting from **array[1][0]** and prints the results according to the function.

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;
}
```

array

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

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

ar

Output:

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

29

Applying One-dimensional Array to Process Two-dimensional Arrays – using Indexing

1. 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 referred to the address of **array[0][0]**.
2. In the function **display2()**, **ar[0]** corresponds to **array[0][0]**, **ar[1]** in the function **display2()** corresponds to **array[0][1]** and so on.
3. The function then accesses the 8 elements starting from **array[0][0]** and prints the contents of the 8 elements to the screen.
4. Therefore, all the elements of the two-dimensional array can be accessed and printed to the screen.

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;
30 }
```

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

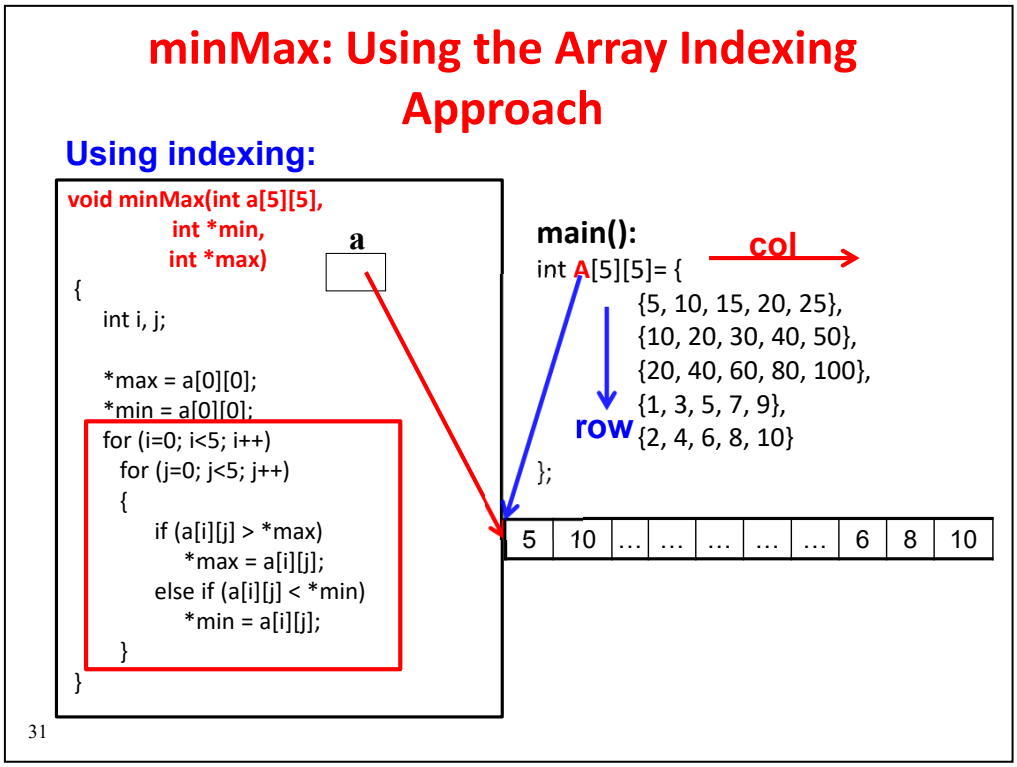
Q: Using indexing?

Q: Using pointer?

```
}
```

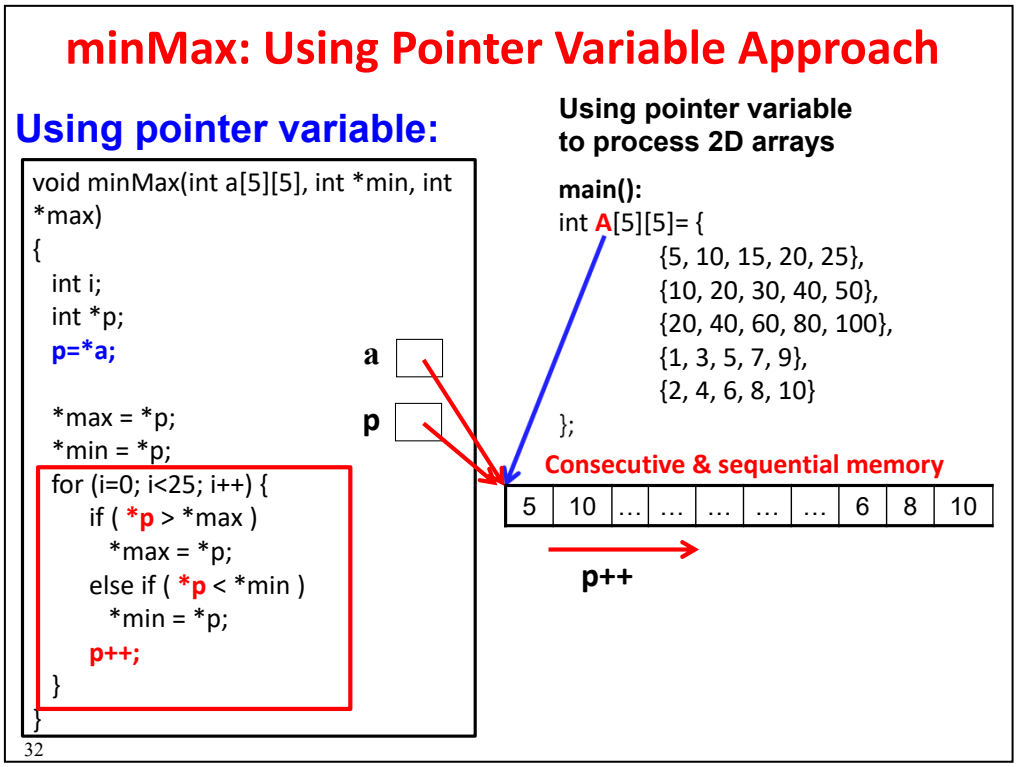
Example: minMax()

1. 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.
2. The function returns the minimum and maximum numbers of the array to the caller through the two parameters **min** and **max** respectively.
3. Call by reference is used for passing the results on maximum and minimum numbers to the calling function.
4. You may use the array indexing approach or pointer variable approach for processing the two-dimensional array.



minMax: Using the Array Indexing Approach

1. In the implementation using the array indexing approach, a nested **for** loop is used to process the two-dimensional array in the function.
2. In the **minMax()** function, it first initializes the ***max** and ***min** to contain the first array element number.
3. The two-dimensional array **a** is processed using indexes I and j to access and compare all the elements stored in the array with ***max** and ***min**.
4. After the processing of the two-dimensional array, the maximum and minimum numbers are determined and stored at ***max** and ***min** respectively.
5. The implementation using indexes is quite straightforward.



minMax: Using the Pointer Variable Approach

1. Different from the previous approach, we can also use the pointer variable approach by updating the **pointer variable** directly.
2. Similarly, a **for** loop is used to traverse and process the two-dimensional array by treating it as an one-dimensional array.
3. 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.
4. 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.

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**

33

Two-dimensional Arrays

1. Here, we discuss the 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

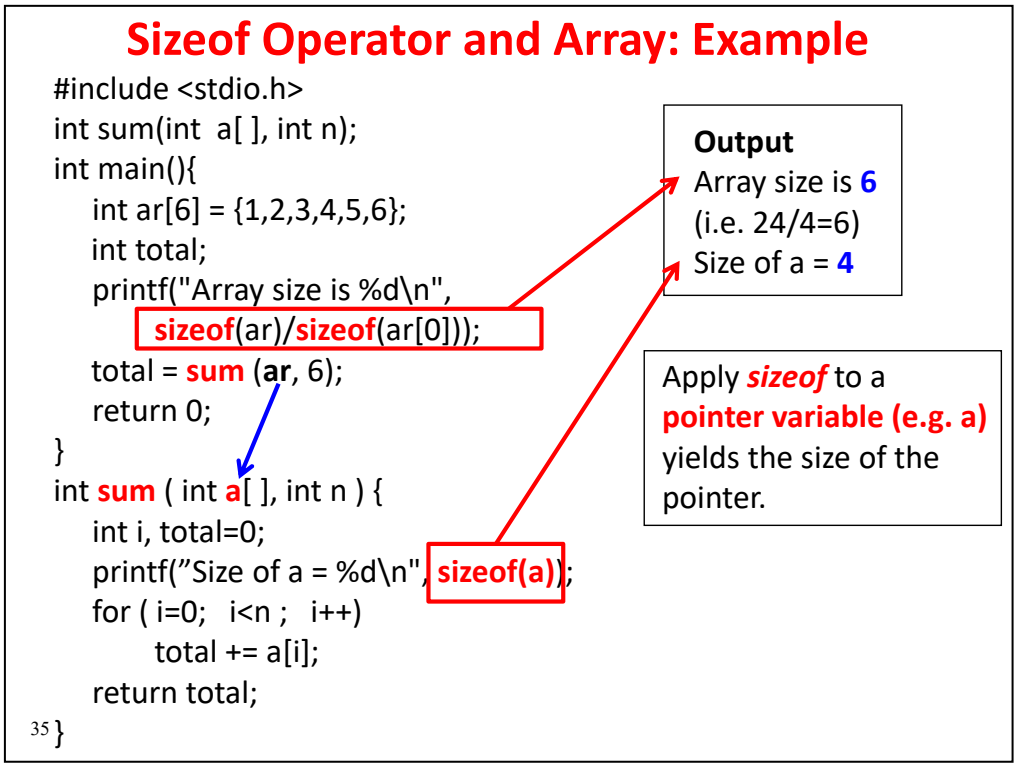
- The **operand** can be:

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

34

Sizeof Operator and Array

1. **sizeof** is an operator which gives the size (in bytes) of its operand. The syntax is **sizeof(operand)** or **sizeof operand.**
2. 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

1. In the **main()** function of the program, the **sizeof** operator returns the number of bytes of the array.
2. The second **sizeof** operator returns the number of bytes of each element in the array.
3. Therefore, the number of elements can be calculated by dividing the size of the array by the size of each element in the array.
4. In this case, the array size is 24/6 which gives the value of 6.
5. 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.

Thank You!

36

Thank You

1. Thanks for watching the lecture video.