

5.1 One-dimensional Arrays

1

One-dimensional Arrays

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

Why Learning Arrays?

- Most programming languages provide array data structure as built-in data structure.
- An array is a list of values with the same data type that can be used to organize and store related data items. If not using array, you will need to define many variables instead of just one array variable.
- Python provides the list structure, which has two major differences from the array data structure in C:
 - Arrays have only limited operations while lists have many operations.
 - Size of arrays cannot be changed while lists can grow and shrink.
- In arrays, we can categorize them as one-dimensional arrays and two-dimensional (or multi-dimensional) arrays. In this lecture, we

2

Why Learning Arrays?

1. Most programming languages provide array data structure as built-in data structure.
2. An array is a list of values with the **same** data type. If not using array, you will need to define many variables instead of just **one** array variable.
3. Python provides the **list** structure, there are two major differences between array and list:
 - Arrays have only limited operations while lists can have many operations.
 - The size of arrays cannot be changed while lists can grow and shrink.
4. In arrays, we can categorize them as one-dimensional arrays and two-dimensional (or multi-dimensional) arrays. In this lecture, we focus on discussing one-dimensional arrays.

One-dimensional Arrays

- **Array Declaration, Initialization and Operations**
- Pointers and Arrays
- Arrays as Function Arguments

3

One-dimensional Arrays

1. Here, we discuss array declaration, initialization and operations in one-dimensional arrays.

Types of Variables

- Data (or values) stored in variables are mainly in two forms:
 - **Primitive Variables**: Variables that are used to store **values**. They are mainly variables of primitive data types, such as **int**, **float** and **char**. Later on, you will learn **Structure**, which is used to store a record of data (values).
 - **Reference (or Pointer) Variables**: Variables that are used to store **addresses**, such as pointer variables, array variables and string variables.

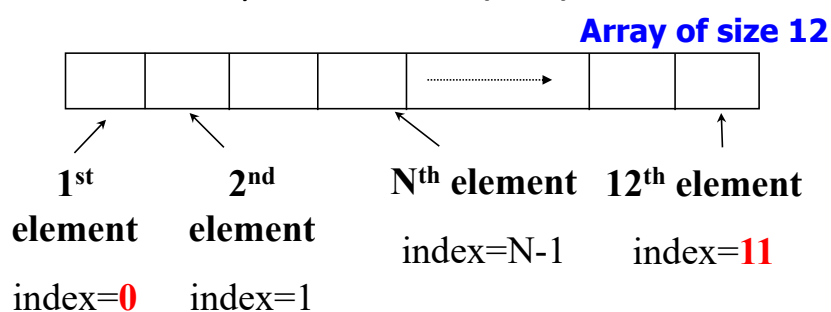
4

Types of Variables

1. There are mainly two types of variables: primitive type variables and reference (pointer) variables.
2. **Primitive variable** is of data type such as **int**, **float**, **char**, etc. which stores the data directly in its memory.
3. **Reference (or pointer) variable** is used to store the **address**, from which the actual data is stored. Apart from pointer variables, arrays and strings are also reference variables. The content stored in an array variable is an address, not the actual data.

What is an Array?

- An **array** is a list of values with the same data type. Each value is stored at a specific, numbered position in the array.
- An array uses an **integer** called index to reference an element in the array.
- The **size** of an array is **fixed** once it is created. Could the size be created dynamically? Yes by using **malloc()**, you will learn that later in data structures.
- Index always starts with **0 (zero)**.



5

What is an Array?

1. An array is a list of values with the same (i.e. one) data type. Each value is stored at a specific, numbered position in the array.
2. An array uses an integer called index to reference an element in the array.
3. The size of an array is fixed once it is created.
4. The index always starts with 0 (zero) and the last element will have an index of length minus 1.

Array Declaration

- Declaration of arrays without initialization:

```
char name[12];           /* array of 12 characters */
float sales[365];        /* array of 365 floats */
int states[50];          /* array of 50 integers */
int *pointers[5];        /* array of 5 pointers to integers */
```

- When an array is declared, some consecutive memory locations are allocated by the compiler for the whole array (**2 or 4** bytes will be allocated for an integer depending on machine):

total_memory = sizeof(type_specifier)*array_size;
e.g. char name[12]; - total_memory = 1*12 = 12 bytes

- The size of array must be integer constant or constant expression in declaration:

e.g. char name[i]; // i is a variable ==> illegal
int states[i*6]; // i is a variable ==> illegal

6

Array Declaration

1. The syntax for an array declaration is **type_specifier array_name[array_size]**;
2. For example, the declaration **char name[12]**; defines an array of 12 elements, each element of the array stores data of type **char**.
3. The elements are stored sequentially in memory. Each memory location in an array is accessed by a relative address called an *index* (or *subscript*).
4. Arrays can be declared without initialization, for examples:

```
float sales[365]; /* array of 365 floats */
int states[50];   /* array of 50 integers */
int *pointers[5]; /* array of 5 pointers to integers */
```

5. When an array is declared, **consecutive memory locations** for the number of elements specified in the array are allocated by the compiler for the whole array. The total number of bytes of storage allocated to an array will depend on the size of the array and the type of data items. The size of memory required can be calculated using the following equation: **total_memory = sizeof(type_specifier)*array_size**; where **sizeof** operator gives the size of the specified data type and **array_size** is the total number of elements specified in the array.
6. For example, in an older system, if it uses 2 bytes to store an integer, and the declaration for the array is **int h[4]**; then a total of 8 bytes is allocated for the

array.

7. An integer constant or constant expression must be used to declare the size of the array. Variables or expressions containing a variable cannot be used for the declaration of the size of the array. The following declarations are illegal:

```
char name[i];    /* where i is a variable */  
int states[i*6];
```

Initialization of Arrays

- Initialize array variables at declaration:

int days[12]={31,28,31,30,31,30,31,31,30,31,30,31};

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	31	30	31	30	31

- Partial array initialization: E.g. (initialize first 7 elements)

int days[12]={31,28,31,30,31,30,31};

/* remaining elements are initialized to zero */

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	0	0	0	0	0

7

Array Initialization

1. After an array has been declared, it can be initialized. Arrays can be initialized at compile time after declaring them. It is done by specifying a list of values after array declaration.
2. The following statement initializes an array **days** with 12 data items: **int days[12]={31,28,31,30,31,30,31,31,30,31,30,31};**
3. An array can also be declared and initialized partially in which the number of elements in the list **{}** is less than the number of array elements. In the given example, only the first 7 elements of the array are initialized: **int days[12]={31,28,31,30,31,30,31};** After the first 7 array elements are initialized, the remaining array elements will be initialized to 0.

Operations on Arrays

- **Accessing** array elements:

```
sales[0] = 143.50;    // using array index
if (sales[23] == 50.0) ...
```

- **Subscripting**: The element indices range from **0** to **n-1** where n is the declared size of the array.

```
char name[12];
name[12] = 'c';    // index out of range – common error
```

- **Working on array values**:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	31	30	31	30	31

- (1) days[1] = 29; - OK ??
- (2) days[2] = days[2] + 4; - OK ??
- (3) days[3] = days[2] + days[3]; - OK ??
- (4) days[1] = {2,3,4,5,6}; - OK? NOT OK!!

8

Operations on Arrays


1. We can access array elements and perform operations on the array elements. The array variable **sales** is declared as an array of 365 floating point numbers: **float sales[365]**; Values can then be assigned into each array element using indexes, e.g. **sales[0]=143.50**;
2. The array can also be used in conditional expressions and looping constructs as follows:


```
if (sales[23]==50.0) {...}
while (sales[364]!= 0.0) {...}
```
4. The elements are indexed from **0** to **n-1** where **n** is the declared size of the array. Therefore, if **char name[12]**; then the following statement: **name[12]='c'**; is invalid since the array elements can only range from **name[0]** to **name[11]**. It is a common mistake to specify an index that is one value more than the largest valid index.
5. Note that in statement (4), **days[1]={2,3,4,5,6}**; is invalid when a list of values is assigned to an array index location.

Traversing an Array – Using Array Index

- One of the **most common actions** in dealing with arrays is to examine every array element in order to perform an operation or assignment.
- This action is also known as **traversing** an array.
- Example:
 - Traverse the **days[]** array using a **for** or **while** loop to access each array element individually with array index, and then process each array element's content accordingly.

days	31	28	31	30	31	30	31	31	30	31	30	31
index	0	1	2	3	4	5	6	7	8	9	10	11



9

Traversing an Array – Using Array Index

1. One of the most common actions in dealing with arrays is to examine every array element in order to perform an operation or assignment. This action is also known as traversing an array.
2. Since array elements can be accessed individually, the most efficient way of manipulating array elements is to use a **for** or **while** loop. The loop control variable is used as the index for the array. Thus, each element of the array can be accessed as the value of the loop control variable changes when the loop is executed. Also note that array values are printed using the corresponding indexes.
3. This is illustrated in this example on using the array **days[]**.

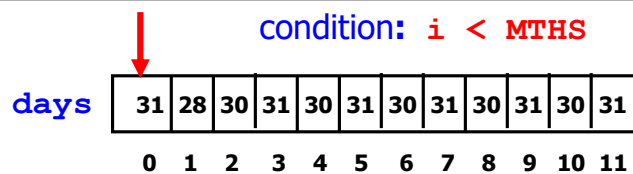
Example 1: Printing Values

```
#include <stdio.h>
#define MTHS 12      /* define a constant */
int main( )
{
    int i;
    int days[MTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};

    /* print the number of days in each month */
    for (i = 0 ; i < MTHS ; i++)
        printf("Month %d has %d days\n", i+1, days[i]);
    return 0;
}
```

Output

Month 1 has 31 days.
 Month 2 has 28 days.
 ...
 Month 12 has 31 days.



10

Example 1 – Printing Values

1. In the program, the array **days** is first initialized using a list of integers.
2. Note that the number in the list should match the size of the array in array initialization. However, if the list is shorter than the size of the array, then the remaining elements are initialized to 0.
3. After that, a **for** loop is used as the control construct to print each element of the **days** array.

Example 2: Searching for a Value

```
#include <stdio.h>
#define SIZE 5      /* define a constant */
int main ( )
{
    char myChar[SIZE] = {'b', 'a', 'c', 'k', 's'};
    int i;
    char searchChar;
    // Reading in user's input to search
    printf("Enter a char to search: ");
    scanf("%c", &searchChar);
    // Traverse myChar array and output character if found
    for (i = 0; i < SIZE; i++) {
        if (myChar[i] == searchChar){
            printf ("Found %c at index %d", myChar[i], i);
            break;    //break out of the loop
        }
    }
    return 0;
}
```

Output

Enter a char to
search: a
Found a at index 1

Example 2 – Searching for a Value

1. When working with arrays, it may be necessary to search for the presence of a specified element. The element that needs to be found is called a *search key*.
2. In the program, the array **myChar** is first initialized using a list of characters. The user can then enter the target character to search. The program will then traverse the array to find the index position of the target character.
3. The program searches for the search key from the array **myChar** and returns the corresponding index position if found.
4. In the program, the target character is firstly read from the user. Then, the character values stored in the array are checked one by one using a **for** loop. If the character value of the checked item is the same as the target character, the corresponding index position is then printed on the screen. And the **break** statement is executed to exit the loop.
5. This **linear search algorithm** compares each element of the array with the search key until a match is found or the end of the array is reached. The program uses linear search by comparing each element of the array with the target character. On average, the linear search algorithm requires to compare the search key with half of the elements stored in an array. Linear search is sufficient for small arrays. However, it is inefficient for large and sorted arrays. Therefore, a more efficient technique such as binary search should be used for large arrays.

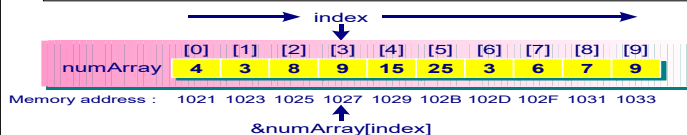
Example 3: Finding the Maximum Value

```
#include <stdio.h>
int main( )
{
    int index, max, numArray[10];
    max = -1; printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", &numArray[index]);
    // Find maximum from array data
    for (index = 0; index < 10; index++) {
        if (numArray[index] > max)
            max = numArray[index];
    }
    printf("The max value is %d.\n", max);
    return 0;
}
```

This example shows how to find the **largest** value in an array of numbers.

Output

Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
 The max value is 25.



12

Example 3 – Finding the Maximum Value

1. The program aims to find the maximum non-negative value in an array. The value for each item in an array is read from the user and stored in the array. Then, the array is traversed element by element in order to find the maximum value in the array.
2. In the program, the value for each item in an array is firstly read from the user and stored in the array. The value **-1** is assigned to the variable **max**, which is defined as the current maximum.
3. Then, the items in the array are checked one by one using a **for** loop. If the value of the next item is larger than the current maximum, it becomes the current maximum. If the value of the next item is less than the current maximum, the current value of **max** is retained. The maximum value in the array is then printed on the screen.

One-dimensional Arrays

- Array Declaration, Initialization and Operations
- **Pointers and Arrays**
- Arrays as Function Arguments

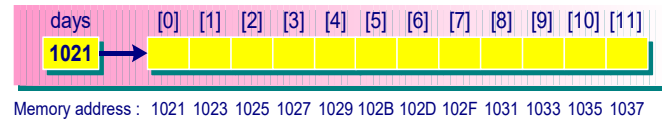
13

Arrays – One-dimensional Arrays

1. Here, we discuss pointers and arrays.

Pointer Constants

- The array name is actually a pointer constant.
e.g. `int days[12];` `// days – pointer constant`



- The array days begins at memory location 1021. Here, we use 2 bytes to represent an integer value (for older machines) for illustration purpose. Note that most current systems represent an integer using 4 bytes.

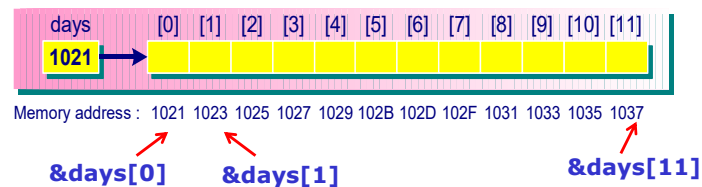
14

Pointer Constants

- There is a strong relationship between pointers and arrays. The array name is in fact a pointer constant.
- When the array `days[]` is declared: `int days[12];` a **pointer constant** with the same name as the array is also created.
- The pointer constant points to the first element of the array. Therefore, the array name by itself, `days`, is containing the address (or pointer) of the first element of the array.
- Assume an integer is represented by 2 bytes (in some older machines) and the array `days` begins at memory location 1021.
- In this array declaration, the array consists of 12 elements. The value stored at `days` is 1021, which corresponds to the address of the first element of the array.

Pointer Constants (Cont'd.)

- Address of an array element:



&days[0] - is the **address** of the **1st** element [i.e. 1021]
&days[1] - is the **address** of the **2nd** element [i.e. 1023]
&days[i] - is the **address** of the **(i+1)th** element

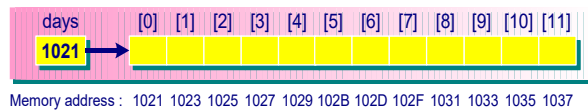
15

Pointer Constants

1. Note that to access the address of an array element, we can use the address operator.
2. For example, for the array **days[]**, **&days[0]** is the address of the 1st element; **&days[1]** is the address of the 2nd element; and **&days[i]** is the address of the (i+1)th element.
3. The address of an array element is important when performing pointer arithmetic with array.

Pointer Constants (Cont'd.)

- **days** - is the **address (or pointer)** of the **1st element of the array**



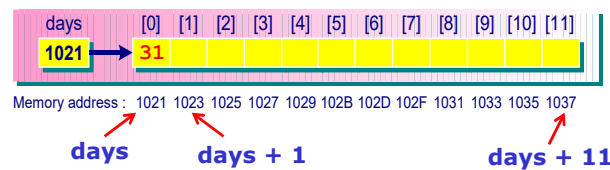
- Note:
 - Array variable: **days** – contains a pointer constant (i.e. 1021) (the value cannot be changed)
 - Array with index: **days[0]**, days[1], etc. – contains the array value at that index location
 - Array element address: **&days[0] (i.e. 1021)**, &days[1], etc. - days[0] has the address of 1021, days[1] has the address of 1023, etc.
- Can we use the pointer **days** for accessing each array element?

16

Pointer Constants

1. The array name by itself, **days**, is the address (or pointer) of the 1st element of the array.
2. What have you observed here?
 - The array variable **days** contains a pointer constant (i.e. 1021), in which the value cannot be changed.
 - The array index **days[0]**, days[1], etc. contains the array value at that index location.
 - The array element address is **&days[0] (i.e. 1021)**, &days[1], etc. That is, days[0] has the address of 1021, days[1] has the address of 1023, etc.
3. The goal is to use the pointer constant **days** for accessing each array element.

Pointer Constants (Cont'd.)



- To do that, we need to know two important concepts:

- (1) **array_name** (i.e. pointer constant)

days == &days[0] (i.e. 1021)

days + i == &days[i]

- (2) ***array_name (dereferencing)**

***days** == days[0] (i.e. 31)

***(days + i)** == days[i]

Note: You may also use ***days** to refer to the content stored at **days[0]**, etc.

- But, you **cannot** change the array **base pointer**:

days += 5; // i.e. **days = days+5;** not valid

days++; // i.e. **days = days+1;** not valid

17

Pointer Constants

- The array name **days** is a pointer constant.
- Since the array name is the pointer to the first element of the array, we have:
 - days** refers to the address of **days[0]**, i.e. **&days[0]**
 - days+1** refers to the address of **days[1]**, i.e. **&days[1]**
 - days+i** refers to the address of **days[i]**, i.e. **&days[i]**
- Therefore, there are two ways to retrieve the content of the element of the arrays. For example, if we want to get the value of the first element, we can use either **days[0]** (using index notation) or ***days** (using pointer notation).
- For example, we can write ***(days+1)** to access the array element **days[1]**. Similarly, ***(days+2)** is used to access array element **days[2]**, etc.
- However, it is important to note that the array name is a **pointer constant**, not a pointer variable. It means that the value stored in **days** cannot be changed by any statements. As such, the following assignment statements are invalid: **days** += 5; and **days**++;

Pointer Variables

- A pointer variable can take on **different addresses**.

```

/* pointer arithmetic */
#define MTHS 12
int main()
{
    int days[MTHS]= {31,28,31,30,31,30,31,31,30,31,30,31};
    int *day_ptr;
    1. day_ptr = days;

    printf("First element = %d\n", *day_ptr);
}

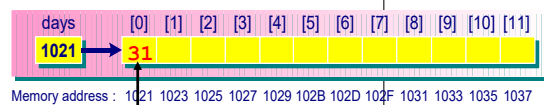
```

← **Pointer constant**

← **Pointer variable**

Output

First element = 31



`day_ptr` 1021

18

Pointer Variables

1. A pointer variable can take on different addresses.
2. In the program, we declare an array variable `days[]` of 12 elements and initialized it with 12 values: `int days[MTHS]= {31,28,31,30,31,30,31,31,30,31,30,31};` where `days` is a pointer constant which is declared as an array of 12 elements.
3. Then, we declare an integer pointer variable `day_ptr`: `int *day_ptr;`
4. The statement `day_ptr = days;` assigns the value 1021 from the array variable `days` to the pointer variable `day_ptr`. This causes the pointer variable to point to the first element of the array.
5. After that, we can use the pointer variable `day_ptr` to access each element of the array.

Pointer Variables (Cont'd.)

- A pointer variable can take on **different addresses**.

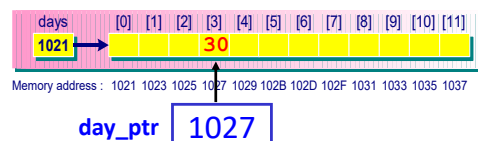
```

/* pointer arithmetic */
#define MTHS 12
int main()
{
    int days[MTHS]= {31,28,31,30,31,30,31,31,30,31,30,31};
    int *day_ptr;
    day_ptr = days;
    printf("First element = %d\n", *day_ptr);

    2. day_ptr = &days[3]; /* points to the fourth element */

    printf("Fourth element = %d\n", *day_ptr);
}

```



Output

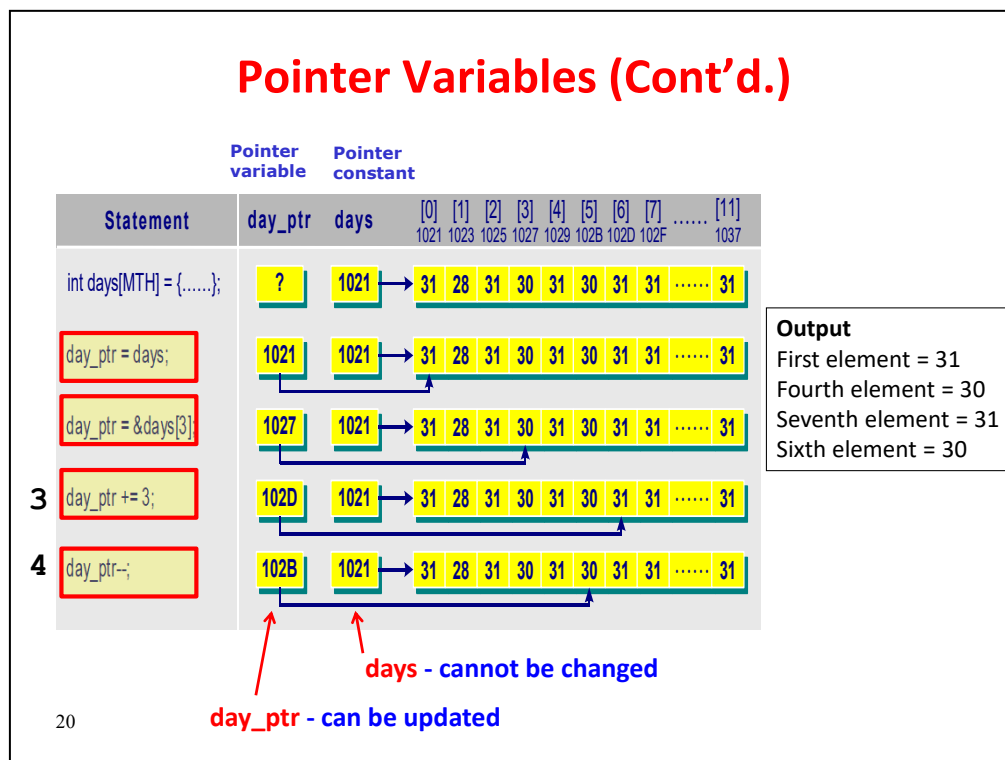
First element = 31
Fourth element = 30

19

Pointer Variables

- The statement `day_ptr = &days[3];` assigns the address of `days[3]` to the `day_ptr`. It updates the `day_ptr` to point to the fourth element of the array.
- The value stored in `day_ptr` becomes 1027.

Pointer Variables (Cont'd.)



Pointer Variables

1. We can add an integer value of 3 to the pointer variable **day_ptr** as follows: **day_ptr += 3**; The **day_ptr** will move forward three elements.
2. The **day_ptr** contains the value of 102D, which is the address of the seventh element of the array **days[6]**.
3. The pointer variable can also be decremented as **day_ptr--**; The **day_ptr** moves back one element in the array. It points to the sixth element of the array **days[5]**.
4. When we perform pointer arithmetic, it is carried out according to the size of the data object that the pointer refers to. If **day_ptr** is declared as a pointer variable of type **int**, then every two bytes (assume that **int** takes 2 bytes) will be added for every increment of one.
5. Therefore, after assigning the array variable to the pointer variable, we can either use the array variable **days** to access each element of the array, or we can use the pointer variable **day_ptr** to access each element.
6. As such, there are two possible ways to process an array: (1) use the array variable directly, or (2) use a pointer variable and assign the array variable to the pointer variable.
7. However, note that the array variable **days** cannot be changed as it is a pointer constant.

Finding Maximum: Using Pointer Constants

```

#include <stdio.h>
int main( )
{
    int index, max, numArray[10];
    printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", numArray + index);
    // Find maximum from array data
    max = *numArray;
    for (index = 1; index < 10; index++)
    {
        if (*(numArray + index) > max)
            max = *(numArray + index);
    }
    printf("The max value is %d.\n", max);
    return 0;
}

```

Pointer constant

numArray [0] .. [9]

Using index for reading input:

```

for (index = 0; index < 10; index++)
    scanf("%d", &numArray[index]);

```

Using index for processing:

```

max = numArray[0];
for (index = 1; index < 10; index++)
{
    if (numArray[index] > max)
        max = numArray[index];
}

```

21

Finding Maximum: Using Pointer Constants

1. In this program, it shows the use of array variable (i.e. pointer constant) to access each element of the array to find the maximum number from an array.
2. The program first reads in 10 integers from the user and stores them into the array variable **numArray**. The **numArray** is the address of the first element of the array, and **numArray+index** is the address of element **numArray[index]**.
3. In addition, you may also use the array notation such as **numArray[index]** to access directly each element of the array. Note that the address operator (**&**) is needed in the **scanf()** statement.
4. The **for** loop accesses each element of the array, and then compares it with **max** in order to determine the maximum value. The maximum value is then assigned to **max**.
5. Note that ***(numArray+index)** is the value of the element **numArray[index]**.
6. Finally, the program prints the maximum value to the screen.

Finding Maximum: Using Pointer Variables

```
#include <stdio.h>
int main( ){
    int index, max, numArray[10];
    int *ptr;
    ptr = numArray;
    printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", ptr++);
    // Find maximum from array data
    ptr = numArray;
    max = *ptr;
    for (index = 0; index < 10; index++) {
        if (*ptr > max)
            max = *ptr;
        ptr++;
    }
    printf("max is %d.\n", max);
    return 0;
}
```

Output
Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
max is 25.

Finding Maximum: Using Pointer Variables

1. The previous program uses the pointer constant **numArray** to access all the elements of the array. Another way to access the elements of an array is to use a pointer variable.
2. This program gives an example using a pointer variable to find the maximum element of the array.
3. To achieve this, it is important to assign **numArray** to **ptr**: that is **ptr = numArray**; After that, we can read in the array data via the pointer variable **ptr**.
4. In the first **for** loop, we use **scanf()** to read in user input. We increment the **ptr** as **ptr++**; to access each element of the array in order to store the input integer into the corresponding index location of the array. The first input will be stored at index location **numArray[0]**, after increasing the pointer **ptr** by 1, the next input integer will be stored at location **numArray[1]**, etc.
5. To find the maximum value stored in the array, we also use a **for** loop. In the second **for** loop, it traverses each element in the array using the pointer variable **ptr**. The value stored at the location of the array is referred to as ***ptr**. The content of each element of the array is compared with the current maximum value.
6. After executing the loop, the maximum value in the array is determined. And the variable **max** will store the maximum value.

Arrays and Pointers – Key Points

- Array is declared as **pointer constant**: In this case, we cannot change the base pointer address.
 - Example: `int numArray[10];`
 - Generally, we can use the **index notation** to access each element of the array, e.g. `numArray[0]` refers to the first element, etc.
 - We can also use the pointer constant to access each element of the array, e.g. `*(numArray+1)` refers to `numArray[1]`, etc. in order to access each element of the array.
- In addition, we can also declare **pointer variables** to access the array.
 - Declare a pointer variable and assign the array to the pointer variable.
Example: `int *ptr; ptr = numArray;`
 - Then we can use `ptr` to access each element of the array.
 - For example, by dereferencing the pointer variable, `*ptr` refers to the first element of the array `numArray[0]`, etc. By updating the pointer variable (`ptr++`) to point to the next array element, we can then access each element of the array.

23

Arrays and Pointers – Key Points

1. Array is declared as pointer constant. For pointer constant declaration, e.g. `int numArray[10];` We can use the **index notation** to access each element of the array, e.g. `numArray[0]` refers to the first element. We can also use the **pointer constant** to access each element of the array, e.g. `*(numArray+1)` refers to `numArray[1]`, etc.
2. However, the base pointer address stored in the array variable cannot be changed.
3. In addition, we can also use **pointer variable** (e.g. `int *ptr;`) to access an array. After declaring a pointer variable, we can assign the pointer variable with the array address, i.e. `ptr = numArray;` we can then use the pointer variable to access each element of the array.
4. As such, both the use of array notation and pointer variable can be adopted for accessing individual elements of an array:
 - Using array index notation: e.g. `numArray[index]`
 - Using pointer constant: e.g. `*(numArray+index)`
 - Using pointer variable: e.g. `*ptr++`
5. However, the use of pointer variable will be more efficient than the array notation, and it is also more convenient when working with strings.

One-Dimensional Arrays

- Array Declaration, Initialization and Operations
- Pointers and Arrays
- **Arrays as Function Arguments**

24

Arrays – One-dimensional Arrays

1. Here, we discuss using arrays as function arguments.

Arrays as Function Arguments: Function Header

Function header

```
void fn1(int table[ ], int size)
{
    ....
}
```

or void fn2(int table[**TABLESIZE**])

```
{
    ....
}
```

or void fn3(int *table, int size)

```
{
    ....
}
```

The prototype of the function:

void fn1(int table[], int size); or

void fn2(int table[**TABLESIZE**]); or

void fn3(int *table, int size);

Note: **size** and **TABLESIZE** are the **data size** to be processed in the array

25

Arrays as Function Arguments: Function Header

1. There are three ways to define a function with an one-dimensional array as the argument.
2. The first way is to define the function as **void function1(int table[], int size)** where **table** is an array and **size** is an integer. The data type of the array is specified, and empty square brackets follow the array name. The integer **size** is used to indicate the size of the array.
3. Another way is to define the function as **void function2(int table[**TABLESIZE**])** where the parameter list includes an array only. The array size **TABLESIZE** is also specified in the square brackets of the array **table**.
4. The third way is to define the function as **void function3(int *table, int size)** where **table** is a pointer of type **int**, and **size** is an integer.

Arrays as Function Arguments: Calling the Function

- Any dimensional array can be passed as a function argument, e.g. we can call the function:

```
fn1(table, n);    /* calling a function */
```

where **fn1()** is a function and **table** is an one-dimensional array, and **n** is the size of the array **table**.

- An **array table** is passed in using call by reference to a function.
- This means the address of the first element of the array is passed to the function.

26

Arrays as Function Arguments: Calling the Function

- We can use an array in a function's body. We may also use an array as a function argument. An array consists of a number of elements. We may pass an element to a function.
- An array can also be passed to a function as an argument, e.g., **fn1(table, n);** where **fn1()** is a function and **table** is an one-dimensional array.
- When we pass an array as a function argument, the array is passed using **call by reference** to the function.
- This means that the address of the first element of the array is passed to the function. Since the function has the address of the array, any changes to the array are made to the original array. There is no local copy of the array to be maintained in the function. This is mainly due to efficiency as arrays can be quite large and thereby taking a considerably large storage space if a local copy is stored.

Array as a Function Argument: Maximum

```
#include <stdio.h>
int maximum(int table[ ], int n);
int main( )
{
    int max, index, n;
    int numArray[10]; // Using index for input

    printf("Enter the number of values: ");
    scanf("%d", &n);
    printf("Enter %d values: ", n);
    for (index = 0; index < n; index++)
        scanf("%d", &numArray[index]);

    // find maximum // Calling the function
    max = maximum(numArray, n);
    printf("The maximum value is %d\n", max);

    return 0 ;
}
```

Output

Enter the number of values: 10
 Enter 10 values: 0 1 2 3 4
5 6 7 8 9
 The maximum value is 9

Arrays as Function Arguments: Maximum

1. In the program, the **main()** function calls the function **maximum()** to compute the maximum value in an array. When the function **maximum()** is called, it passes an array as the function argument.
2. The function **maximum()** determines the maximum value stored in the array. Apart from the array argument **numArray**, the number of elements stored in the array is also passed as an integer argument **n**.

Implementing Maximum: (1) Using Array Indexing

```

#include <stdio.h>
int maximum(int table[ ], int n);
int main( )
{
    int max, index, n;
    int numArray[10];

    ....
    max = maximum(numArray, n);
    printf("The maximum value is %d\n", max);
    return 0 ;
}

```

numArray

[0]	1	2	3	4	5	6	7	8	9
-----	---	---	---	---	---	---	---	---	---

↑
i=4

```

int maximum(int table[ ], int n)
{
    int i, max;

    max = table[0];
    for (i = 1; i < n; i++)
        if (table[i] > max)
            max = table[i];
    return max;
}

```

Using array indexing

table

n

10

28

Implementing Maximum: Using Array Indexing

1. The implementation of the function **maximum()** uses **array indexing**. It has two parameters: **table** and **n**.
2. The array is traversed element by element using indexing with **table[i]**, where **i** is the index from 0 to **n-1**, in order to find the maximum number.
3. At the end of the function, the maximum number stored in **max** is passed to the calling function.

Implementing Maximum: (2) Using Ar Base Address

```

#include <stdio.h>
int maximum(int table[], int n);
int main()
{
    int max, index, n;
    int numArray[10];

    ....
    max = maximum(numArray, n);
    printf("The maximum value is %d\n", max);
    return 0;
}

```

numArray

[0]	[9]
0	1	2
3	4	5
6	7	8
9		

↑
i=4

```

int maximum(int table[], int n)
{
    int i, max;

    max = *table;
    for (i = 1; i < n; i++)
        if (*(table+i) > max)
            max = *(table+i);
    return max;
}

```

Using array base address

table

10
n

Implementing Maximum: Using Array Base Address

1. The implementation of the function **maximum()** uses array base address.
2. As shown, the base address of the array **table** is used.
3. When traversing the array, the array element is accessed via ***(table+i)**, where **i** is the index from 0 to **n-1**.
4. The maximum number is then determined at the end of the loop.

Implementing Maximum: (3) Using Pointer Variable

```

#include <stdio.h>
int maximum(int table[], int n);
int main()
{
    int max, index, n;
    int numArray[10];

    ....
    max = maximum(numArray, n);
    printf("The maximum value is %d\n", max);
    return 0;
}

```

numArray →

[0]	1	2	3	4	5	6	7	8	9
-----	---	---	---	---	---	---	---	---	---

-- -- --
++table

Updating the pointer variable to the next index location

↑

10
n

```

int maximum(int table[], int n){
    int i, max;
    max = *table;
    for (i = 0; i < n; i++) {
        if (*table > max)
            max = *table;
        ++table;
    }
    return max;
}

```

30

Implementing Maximum: Using Pointer Variable

1. The implementation of the function **maximum()** uses pointer variable notation.
2. In this version of implementation, the array **table** is used as a pointer variable. When traversing the array, the array element is accessed via ***table**, and the variable **table** is incremented by 1 using **table++** in order to access each element of the array.
3. The maximum number is then determined at the end of the loop.



Thank You!

31

Thank You

1. Thanks for watching the lecture video.

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.

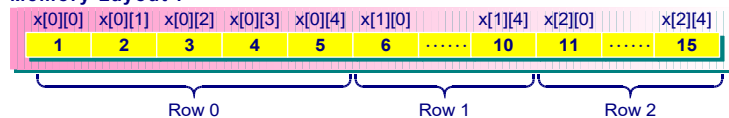
Two-dimensional Arrays: Memory Layout

```
int x[3][5];
```

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

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>	<code>x[0][3]</code>	<code>x[0][4]</code>
Row 1	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>	<code>x[1][3]</code>	<code>x[1][4]</code>
Row 2	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>	<code>x[2][3]</code>	<code>x[2][4]</code>

Memory Layout :



Consecutive & sequential memory

5

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]={
    {5, 10, 15},
    {10, 20, 30},
    {20, 40, 60}
  };
  row
  ↓
  column
  →
```

Output

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

Nested Loop

```
/* 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;
7 }
```

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]={
        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
```

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.

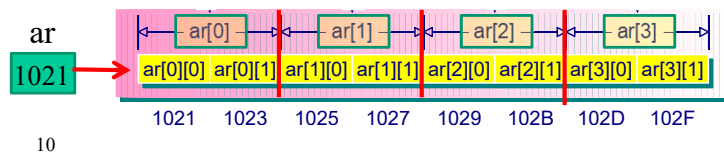
Two-dimensional Arrays and Pointers

- 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



Two-dimensional Arrays and Pointers

- 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 */
```

- The array variable `ar` is the address of the first element of the array.

Two-dimensional Arrays and Pointers

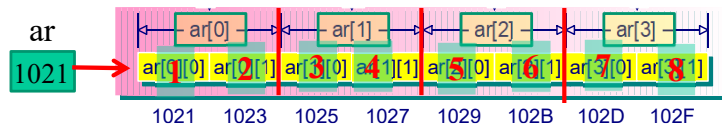
- Two-dimensional array variable declaration:

```
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

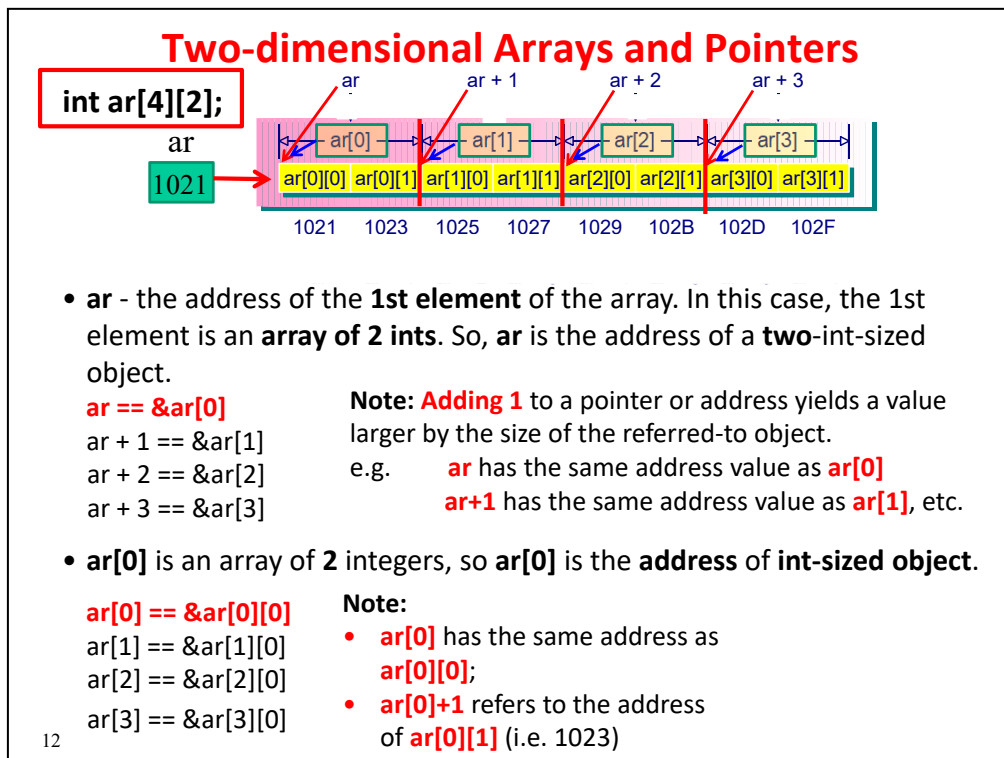
- After array declaration, memory locations are allocated and used to store the initial values of the array.



11

Two-dimensional Arrays and Pointers

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



Two-dimensional Arrays and Pointers

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

Two-dimensional Arrays and Pointers

int ar[4][2];

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

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

- Similarly

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

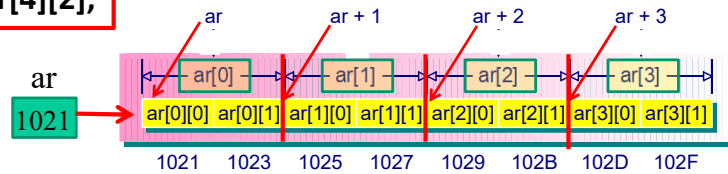
ar == &ar[0],	using dereferencing we have *ar == ar[0]
ar + 1 == &ar[1],	using dereferencing we have *(ar + 1) == ar[1]
ar + 2 == &ar[2],	using dereferencing we have *(ar + 2) == ar[2]
ar + 3 == &ar[3],	using dereferencing we have *(ar + 3) == ar[3]

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

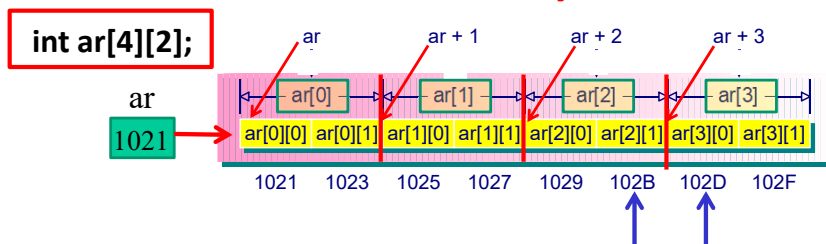
***ar** refers to the value of its first element, **ar[0]**

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:

$$\mathbf{ar[m][n]} == * (* (\mathbf{ar + m}) + \mathbf{n})$$

$$\text{e.g. } \mathbf{ar[2][1]} = * (* (\mathbf{ar + 2}) + \mathbf{1}) \quad [m=2, n=1]$$

$$\mathbf{ar[3][0]} = * (* (\mathbf{ar + 3}) + \mathbf{0}) \quad [m=3, n=0]$$

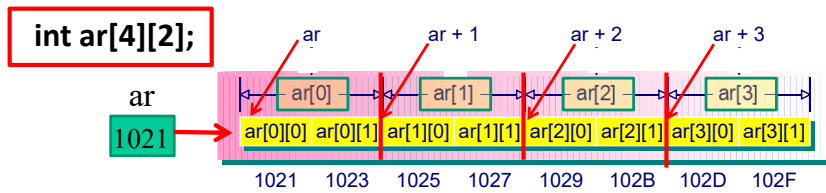
Note: you are not required to remember the calculation on deriving the general formula.

15

Two-dimensional Arrays and Pointers

- 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]**.
- 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.
- 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 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;
}
```

Diagram illustrating array processing:

Array **ar** is shown as a row of boxes containing: 5, 10, 15, .., .., .., .., 60.

An arrow labeled **ptr** points to the first element (5).

An arrow labeled **ptr++** points to the next element (10).

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:

<pre>void fn(int array[2][4]) { }</pre>	or	<pre>void fn(int array[][4]) { }</pre>
--	----	--

/*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

- 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 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.
- 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])**.
- 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`

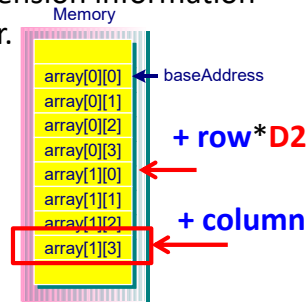
The address of `array[1][3]` is computed as:

$\text{baseAddress} + \text{row} * D2 + \text{column}$

$\Rightarrow \text{baseAddress} + 1 * 4 + 3$

$\Rightarrow \text{baseAddress} + 7$

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:

$$\text{baseAddress} + \text{row} * D2 + \text{column}$$

$$\Rightarrow \text{baseAddress} + 1 * 4 + 3$$

$$\Rightarrow \text{baseAddress} + 7$$
 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.)

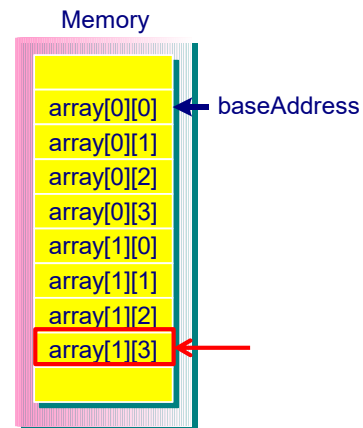
- Since **D1** is not needed in computing the address, we can omit the first dimension value in defining a function which takes arrays as its formal arguments.

- 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 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;
}

```

main():

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

24

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');
}

```

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;
}

```

```

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

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

```

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');
}

```

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;
}
```

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

    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
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

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
---	----	-----	-----	-----	-----	-----	---	---	----

31

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 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++;
    }
}

```

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++

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

```
#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.

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.