5.1
One-Dimensional Arrays

1

Review

- What have you learnt so far:
  - Basic Sequential Programming (data, operators, simple I/O)
  - Branching (if-else, switch, conditional operator)
  - Looping (for, while, do…while)
  - Functions (function definition and call by value)
  - **Pointers (call by reference for function communication)**
- With **Pointers**, you may use **call by reference** to pass more than one value to the calling function…..
- Apart from using it to pass more than one parameter to the calling functions, pointers can also be used in other data structures such as **arrays**, **strings** and **structures**.
- In this lecture, we will discuss **… 1D Arrays …** and the use of pointers in 1D arrays.

2

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. If not using array, you will need to define many variables instead of just **one** array variable.
- Python provides the **list** structure, two major differences from array:
  - Arrays have only limited operations while lists have a large number of operations;
  - Size of arrays cannot be changed while lists can grow and shrink.

3

This Lecture

- At the end of this lecture, you will be able to understand the following on 1D arrays:
  - Array Declaration, Initialization and Operations
  - Pointers and Arrays
  - Arrays as Function Arguments

4

Arrays

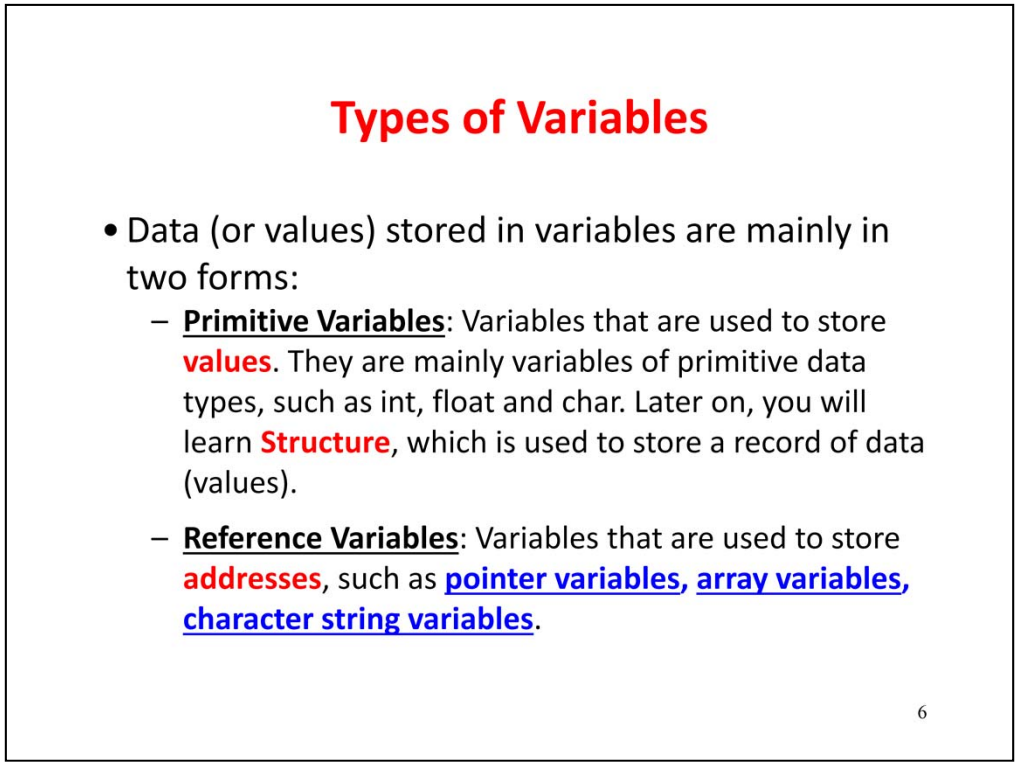
In the previous lectures, we have discussed the various data types such as **char**, **int**, **float**, etc. When we define a variable of one of these types, the computer will reserve a memory location for the variable. Only one value is stored for each variable at any one time. However, there are applications which require storing related data items under one variable name. For example, if we have different items with similar nature, such as examination marks for the programming course, we might need to declare different variables such as mark1, mark2, etc. to represent the mark for each student. This is quite cumbersome if the number of students is very large. Instead, we can declare a single variable called mark as an array, and each element of the array can be used to store the mark for each student. In arrays, we can use a single **variable** to collect a **group** of data items of the ***same data type***.

In this lecture, we introduce this important topic on data structure that can be used to organize and store related data items. *Arrays* are used to store related data items of the same data type. In arrays, we can categorize them as one-dimensional arrays and multi-dimensional arrays. In this lecture, we focus on discussing one-dimensional arrays, and in the next lecture, we will discuss multi-dimensional arrays.

One-Dimensional Arrays

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

5



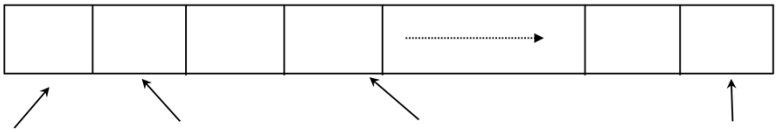
Types of Variables

As discussed before, there are mainly two types of variables: primitive type variables and reference (pointer) variables. Primitive variable is of data type such as **int**, **float**, **char**, etc. which store the data directly in its memory. Reference variable such as pointer variable is used to store the address, in 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.
- Index always starts with **0 (zero)**.

Array of size 12



1st element index=**0**

2nd element index=1

Nth element index=N-1

12th element index=**11**

7

What is an Array?

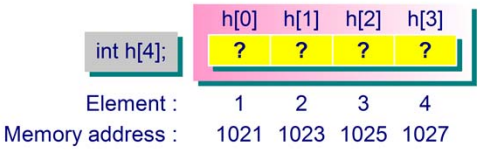
An array is a list of values with the same (one) data type. Each value is stored at a specific, numbered position in the array. An array uses an integer called index to reference an element in the array. The size of an array is fixed once it is created. The index always starts with 0 (zero) and the last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

Array Declaration

- Declaration of arrays **without initialization**:

```
float sales[365];        /*array of 365 floats */
char name[12];           /*array of 12 characters*/
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 for an integer depending on machine)



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

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

8

Array Declaration

The syntax for an array declaration is as follows:

type_specifier array_name[array_size];

where **type_specifier** specifies the type of data to be stored in the array, **array_name** is the name given to the array, and **array_size** specifies the number of elements in the array.

For example, the declaration

char name[12];

defines an array of 12 elements, each element of the array stores data of type **char**. The elements are stored sequentially in memory. Each memory location in an array is accessed by a relative address called an *index* (or *subscript*).

Arrays can be declared without initialization:

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, **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. For example, if a system uses 2 (or 4) bytes to store an integer, the declaration for the array

int h[4];

will result in a total of 8 (or 16) bytes allocated for the array. 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.

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

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

are illegal.

Initialization of Arrays

- **Initialize array** variables at declaration:

```
#define MTHS 12        /* define a constant */
int days[MTHS]={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)

```
#define MTHS 12
int days[MTHS]={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

9

Array Initialization

After an array has been declared, it must be initialized. Arrays can be initialized at compile time after declaring them. The format for initializing an array is

type_specifier array_name[array_size]={list_of_values};

The statements

```
#define MTHS 12          /* define a constant */
int days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
```

initialize the array **days** with 12 data items.

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 following example, only the first 7 elements of the array are initialized.

```
#define MTHS 12
int days[MTHS]={31,28,31,30,31,30,31};
```

After the first 7 array elements are initialized, the remaining array elements will be initialized to 0.

Initialization of Arrays (Cont'd.)

- **Omitting the size** in array initialization:

```
int days[ ]= {31,28,31,30,31,30,31};
             /* an array of 7 elements */
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
days	31	28	31	30	31	30	31

10

Array Initialization

In addition, an array can also be declared and initialized without explicitly indicating the array size. The declaration

```
int days[]={31,28,31,30,31,30,31};      /* an array of 7 elements */
```

is valid. It declares **days** as an array of 7 elements as there are 7 elements in the list.

Operations on Arrays

- **Accessing** array elements:

	[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

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

- **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:

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

Operations on Arrays

We can access array elements and perform operations on the array elements. If the array variable **sales** is declared as

```
float sales[365];          /* array of 365 floats */
```

array elements can then be processed. Values can be assigned into each array element. The array can also be used in conditional expressions and looping constructs as follows:

```
sales[0]=143.50;
if (sales[23]==50.0) {...}
while (sales[364]!= 0.0) {...}
```

The elements are indexed from **0** to **n-1** where **n** is the declared size of the array. Therefore, the following statements:

```
char name[12];
name[12]='c';        /* invalid - index out of range */
```

are 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.

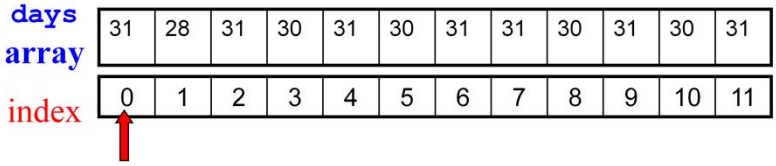
Similarly, the following initialization:

```
int days[2]={2,3,4,5,6};        /* invalid */
```

is also invalid.

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 to display every element's content:



12

Traversing an Array – Using Array Index

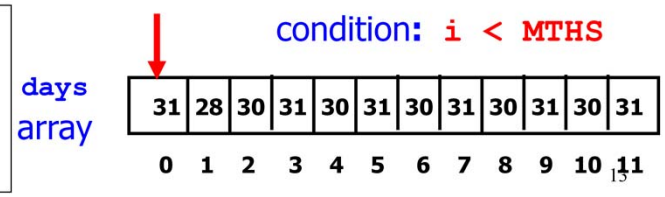
Since array elements can be accessed individually, the most efficient way of manipulating array elements is to use a **for** or **while** loop. The loop control variable is used as the index for the array. Thus, each element of the array can be accessed as the value of the loop control variable changes when the loop is executed. Also note that array values are printed using the corresponding indexes.

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 */


    return 0;
}
```

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



13

Traversing an Array – Printing Values

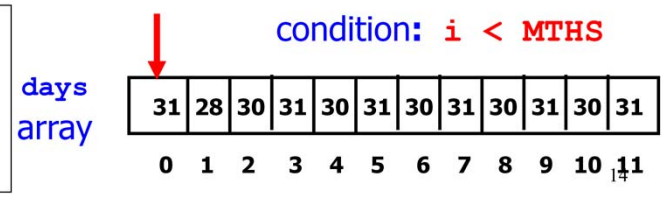
In the program, the array **days** is first initialized using a list of integers. The number in the list should match the size of the array. However, if the list is shorter than the size of the array, then the remaining elements are initialized to 0.

Example 1: Printing Values (Cont'd.)

```
#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.
```



Traversing an Array – Printing Values

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




15  return 0;
}
```

Output

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

Traversing an Array – Searching for a Value

When working with arrays, it may be necessary to search for the presence of a particular element. The element that needs to be found is called a *search key*. 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.

Example 2: Searching for a Value (Cont'd.)

```
#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
        }
    }
16  return 0;
}
```

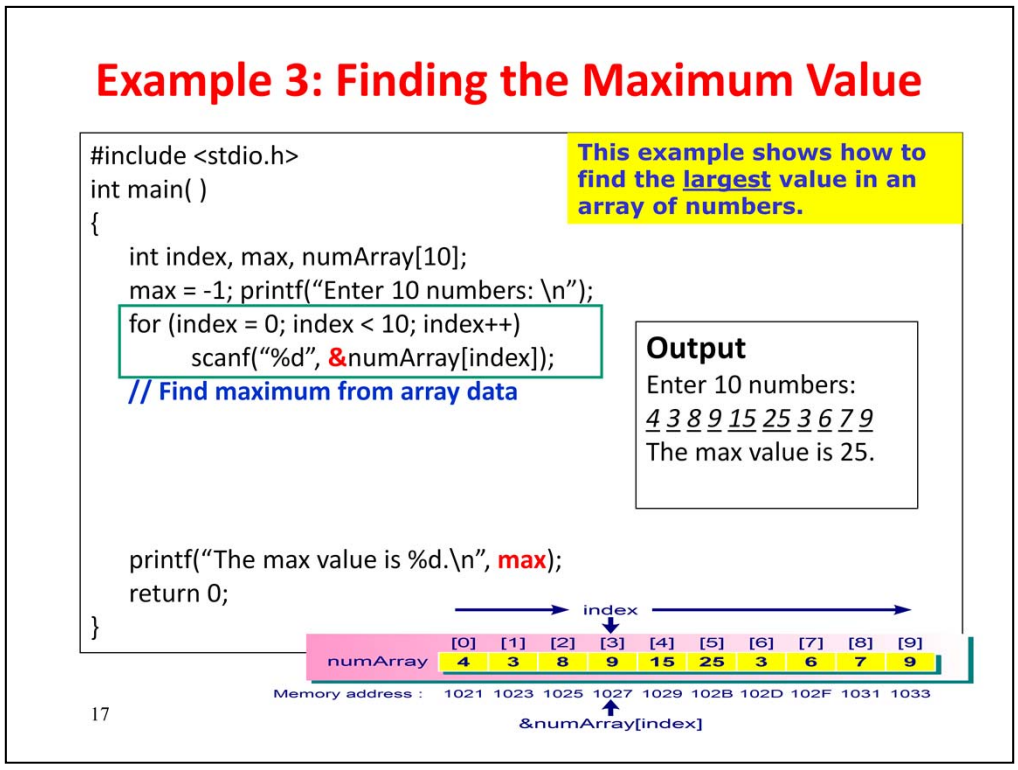
Output

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

Traversing an Array – Searching for a Value

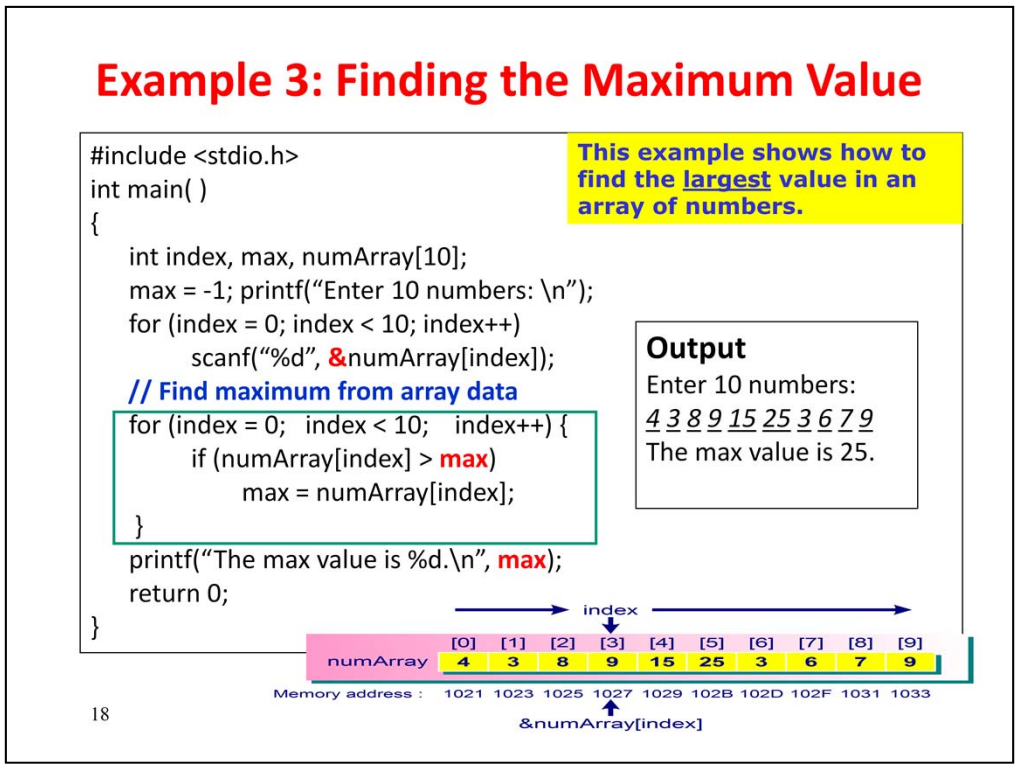
The program searches for the search key from the array **myChar** and returns the corresponding index position if found. In the program, the target character is firstly read from the user. Then, the character values stored in the array are checked one by one using a **for** loop. If the character value 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.

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.



Traversing an Array – Finding the Maximum Value

The program finds 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.



Traversing an Array – Finding the Maximum Value

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. 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

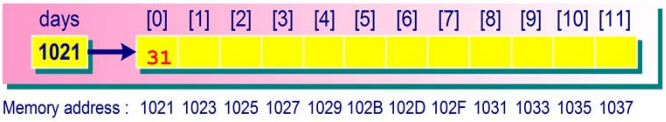
- Array Declaration, Initialization and Operations
- **Pointers and Arrays**
- Arrays as Function Arguments

19

Pointer Constants

- The **array name** is actually a ***pointer constant***:
- An integer can be represented by 4 bytes (or 2 bytes – in older machines (as in this illustration)) and the array **days** begins at memory location 1021.

e.g. int **days**[12]; // **days – pointer constant**



days[0] = 31; /* days[0] contains the value assigned to it */

20

Pointer Constants

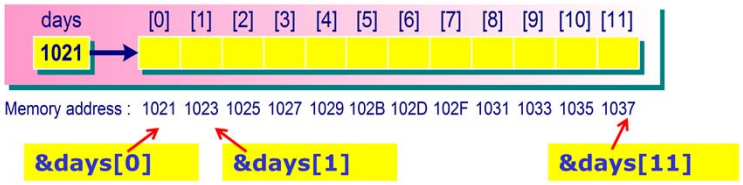
There is a strong relationship between pointers and arrays. The array name is actually a pointer constant. When the following array is created:

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 actually the address (or pointer) of the first element of the array. Assume an integer is represented by 2 bytes (in 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

21

Pointer Constants

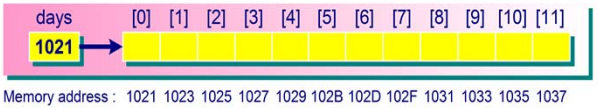
Note that to access the address of an array element, we can use the address operator. For example, if we declare an array as

int days[12];

then **&days[0]** is the address of the 1st element; and **&days[i]** is the address of the (i+1)th element. 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**



- What have you observed?
  - Array variable **days** – contains a pointer constant (i.e. 1021) (the value cannot be changed)
  - Array 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.
- Aim – to be able to use the pointer **days** for accessing each array element. How to do that?

22

Pointer Constants

Thus, the array name by itself, **days**, is really the address (or pointer) of the 1st element of the array.

Since the array name is the pointer to the first element of the array, we have

days == &days[0]

days+1 == &days[1]

days+i == &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**

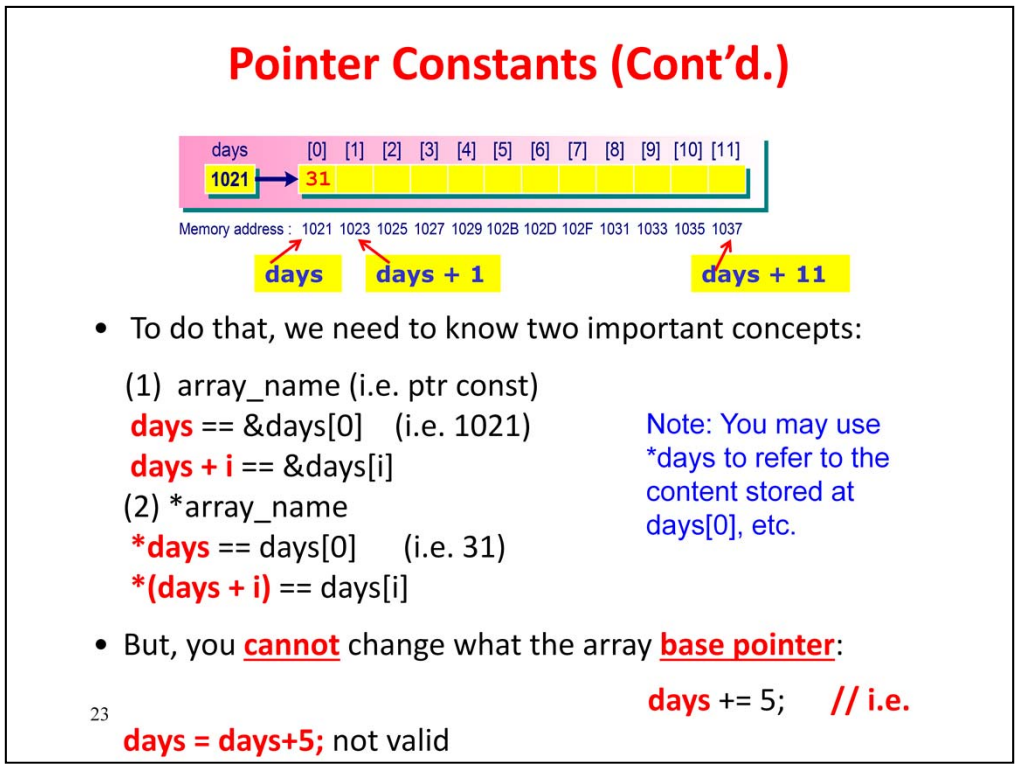
For example, we can write ***(days+1)** to access the array element **days[1]**. Similarly, ***(days+2)** is used to access array element **days[2]**, etc.

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:

days += 5;

days++;

are invalid.



Pointer Constants

Thus, the array name by itself, **days**, is really the address (or pointer) of the 1st element of the array.

Since the array name is the pointer to the first element of the array, we have

days == &days[0]

days+1 == &days[1]

days+i == &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**

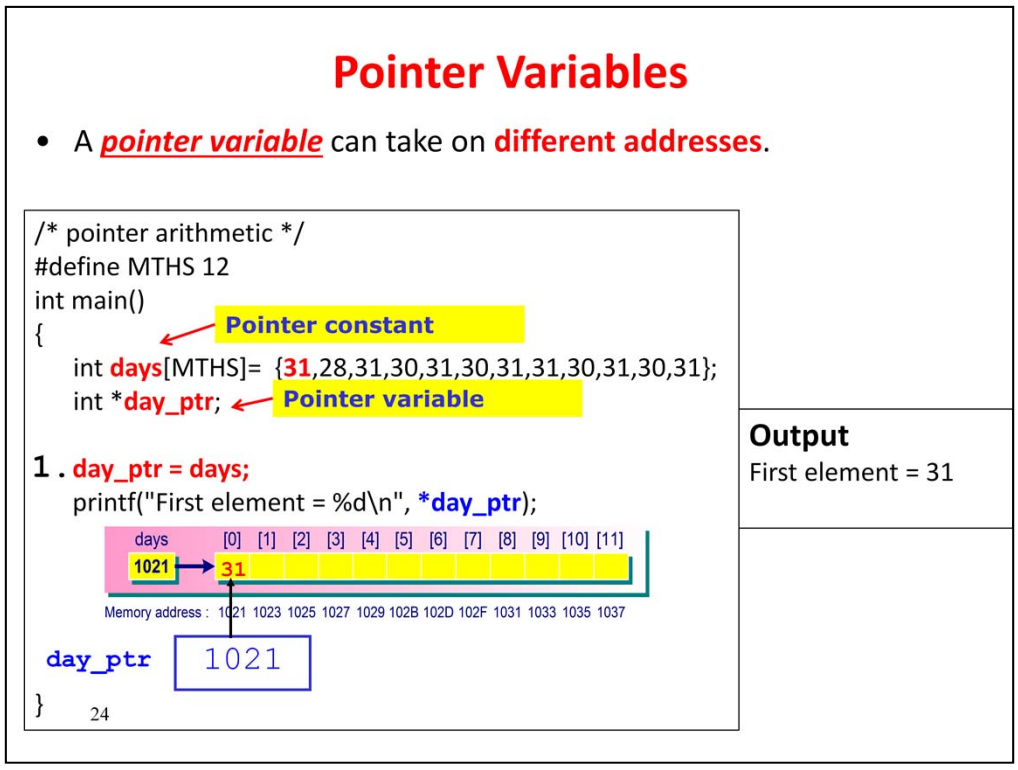
For example, we can write ***(days+1)** to access the array element **days[1]**. Similarly, ***(days+2)** is used to access array element **days[2]**, etc.

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:

days += 5;

days++;

are invalid.



Pointer Variables

A *pointer variable* can take on different addresses.

In the program, we declare

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.

If we declare

int ***day_ptr**;

where **day_ptr** is a pointer variable.

The statement

day_ptr = days;

assigns the value 1021 stored in **days** to the pointer variable **day_ptr**. This causes the pointer variable to point to the first element of the array. After that, we can use the pointer variable **day_ptr** to access each element of the array.

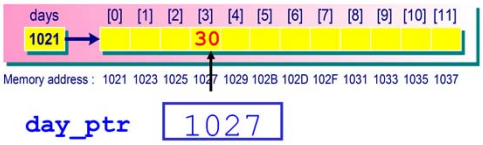
Pointer Variables (Cont'd.)

- 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);
}
```

Pointer constant

Pointer variable



day_ptr 1027

25

Output

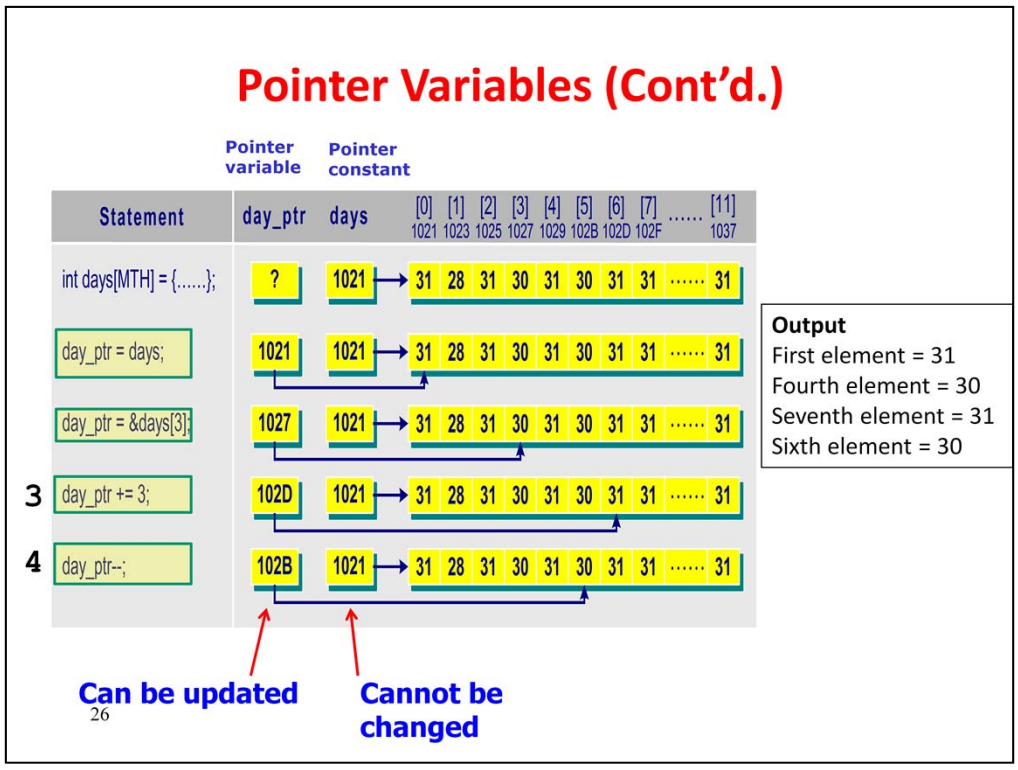
First element = 31
Fourth element = 30

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

We can also 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. The **day_ptr** contains the value of 102D, which is the address of the seventh element of the array **days[6]**.

The pointer variable can also be decremented as:

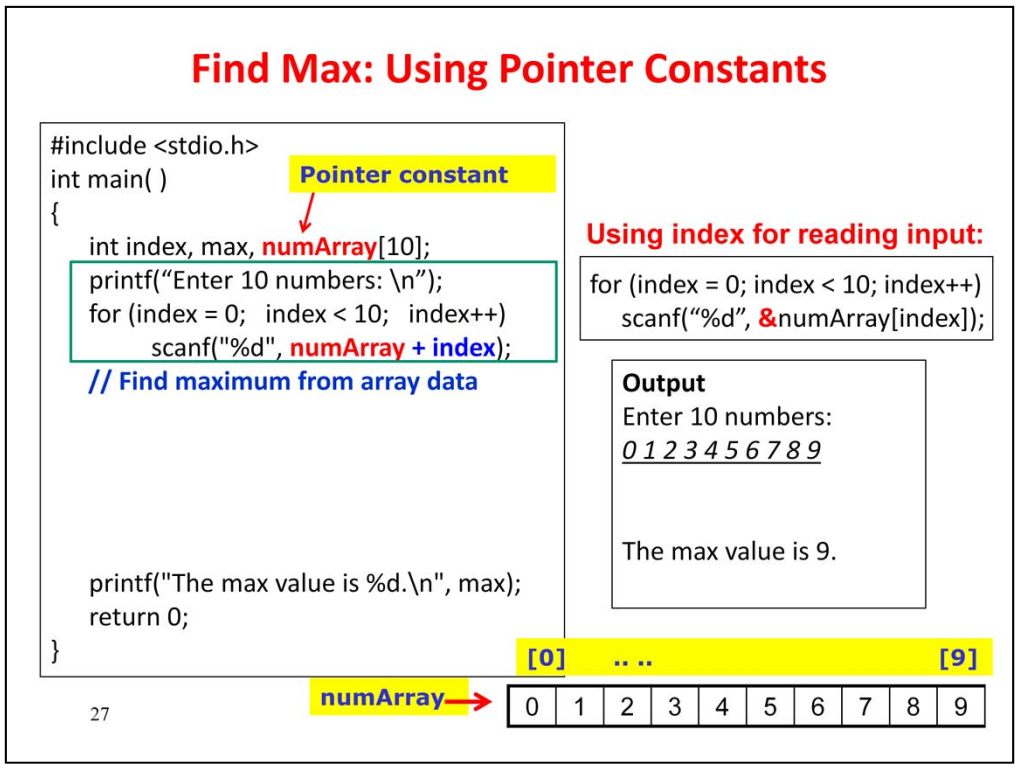
day_ptr--;

The **day_ptr** moves back one element in the array. It points to the sixth element of the array **days[5]**.

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 (assuming that **int** takes 2 bytes) will be added for every increment of one.

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. 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.

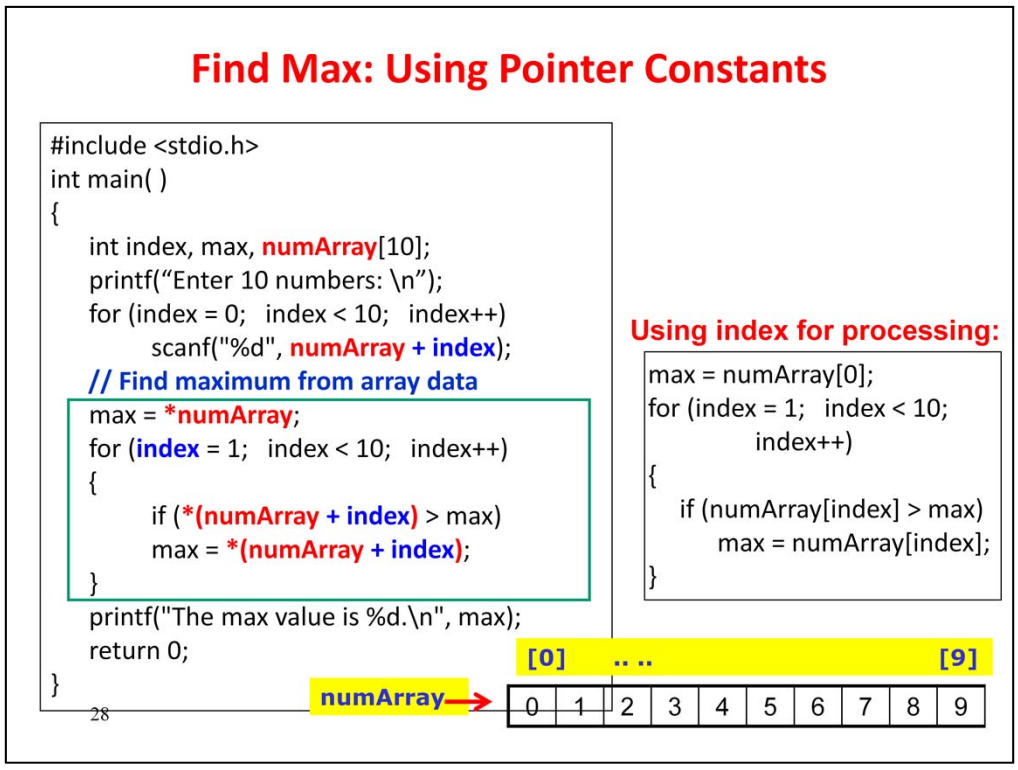
However, note that the array variable cannot be changes as it is a pointer constant.



Example: Using Pointer Constants

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. 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]**. In addition, you may also use the array notation such as **numArray[index]** to access directly each element of the array.

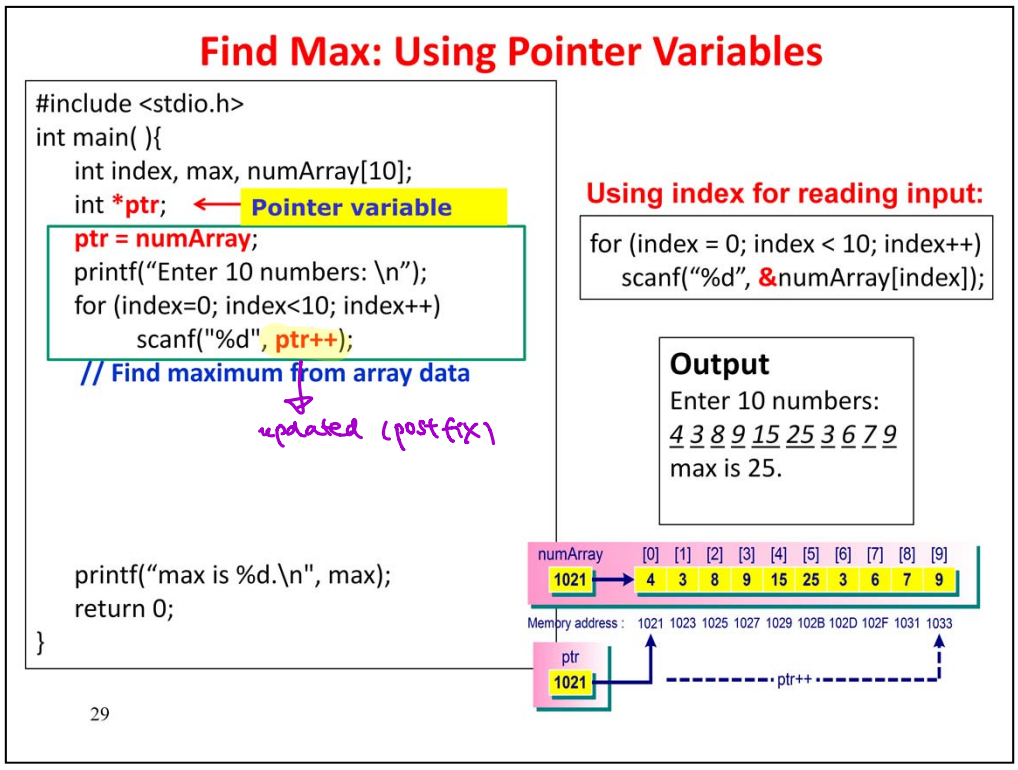
Note that the address operator (**&**) is needed in the **scanf()** statement.



Example: Using Pointer Constants

The **for** loop accesses each element of the array, and compares it with **max** in order to determine the maximum value. The maximum value is then assigned to **max**.
***(numArray+index)** is the value of the element **numArray[index]**.

In addition, you may also use the array notation such as **numArray[index]** to access directly each element of the array.

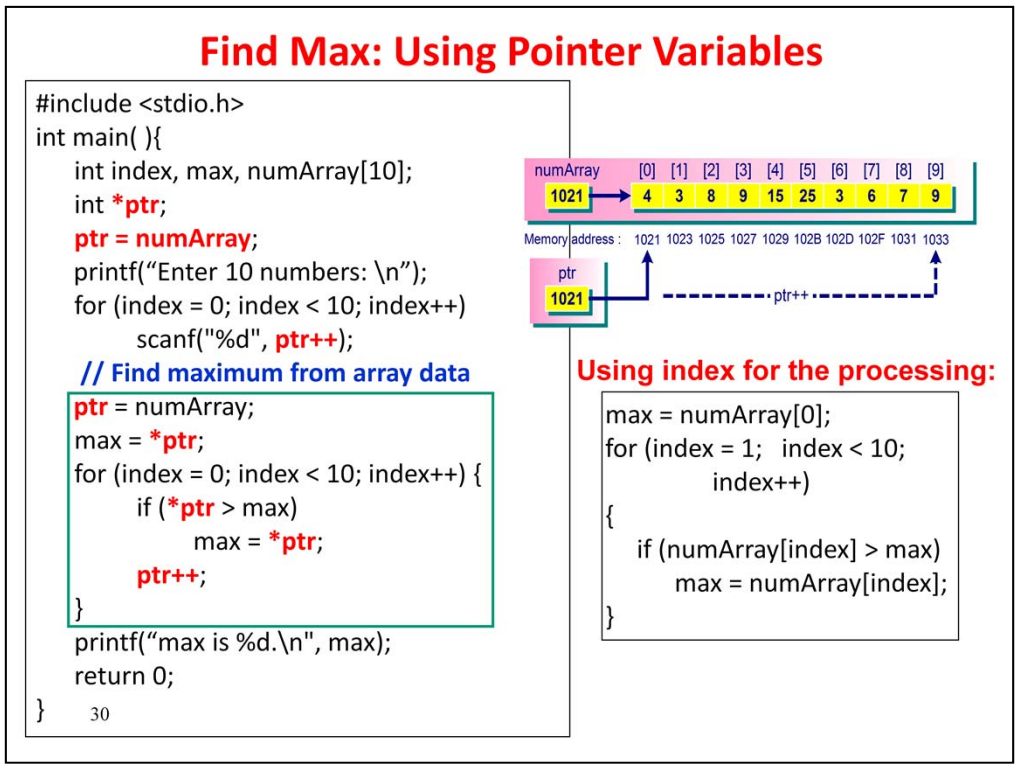


Example: Using Pointer Variables

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. This program gives an example using a pointer variable to find the maximum element of the array. To achieve this, it is important to assign **numArray** to **ptr**: **ptr = numArray;** After that, we can read in the array data via the pointer variable **ptr**. In the **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.



Example: Using Pointer Variables

To find the maximum value stored in the array, we use a **for** loop. In the **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. 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 Ideas

- Array is declared as **pointer constant**: In this case, we cannot change the base pointer address.
  - Example: int **numArray**[10];
  - Can use the index notation to access each element of the array, e.g. numArray[0] refers to the first element.
  - Can also use the pointer constant to access each element of the array, e.g. ***(numArray+1)** refers to **numArray[1]**, etc.
- In addition, we can also declare **pointer variables** to access the array.
  - Example: declare a pointer variable and assign the array to the pointer variable:

    int ***ptr**;

    ptr = numArray;
  - Then we can use the pointer notation to access each element of the array, e.g. ***ptr** refers to the first element of the array **numArray[0]**, etc.

31

Arrays and Pointers – Key Ideas

Array is declared as pointer constant. For pointer constant declaration (e.g. int **numArray**[10];), the base pointer address stored in the array variable cannot be changed. For pointer variable declaration (e.g. int ***ptr**;), we can then 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.

As such, both the use of array notation and pointer variable can be adopted for accessing individual elements of an array:

- Using array notation: e.g. **numArray[index]**
- Using pointer constant: e.g. ***(numArray+index)**
- Using pointer variable: e.g. ***ptr**

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**

32

Arrays as Function Arguments: Function Header

Function header

```
void fn(int table[ ], int n)
{
   ....
}
```

The prototype of the function:

```
void fn(int table[ ], int n);   or
```

```
or  void fn(int table[TABLESIZE])
{
   ....
}
```

```
void fn(int table[TABLESIZE]); or
```

```
or  void fn(int *table, int n)
{
   ....
}
```

```
void fn(int *table, int n);
```

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

55

Arrays as Function Arguments

For example, assume a function called **maximum()** that finds the maximum value of the following array elements:

int table[10] = {34,21,65,54,17,48,29,93,49,23};

has been written. We may call the function as follows:

maximum(table, 10);

To receive the array argument, the function must be defined with a parameter that is an array. There are three ways to define a function with a one-dimensional array as the argument. The first way is to define the function as

int maximum(int table[], int n)

The parameter list includes an array **table** and an integer **n**. The data type of the array is specified, and empty square brackets follow the array name. The integer **n** is used to indicate the size of the array.

Another way is to define the function as

int maximum(int table[TABLESIZE])

The parameter list includes an array only. The array size **TABLESIZE** is also specified in the square brackets of the array **table**.

The third way is to define the function as

int maximum(int *table, int n)

The parameter list includes a pointer **table** of type **int**, and an integer **n**. The integer **n** is used to indicate the size of the array.

The function prototypes of the function become

int maximum(int table[], int n);

or **int maximum(int table[TABLESIZE]);**

or **int maximum(int *table, int n);**

Arrays as Function Arguments: Calling the Function

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

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

where **fn()** is a function and **table** is a 1-D 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.

34

Arrays as Function Arguments

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 access individual element, or pass an element to a function. To do this, the index of the array (e.g. **days[3]**) is used to specify the element that will be passed to the function.

An array can also be passed to a function as an argument, e.g.,

fn(table);

where **fn()** is a function and **table** is a 1-D array.

When we pass an array as a function argument, the original array is passed to the function. 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. In fact, 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.

Array as a Function Argument: Maximum

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

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

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

    return 0 ;
}
```

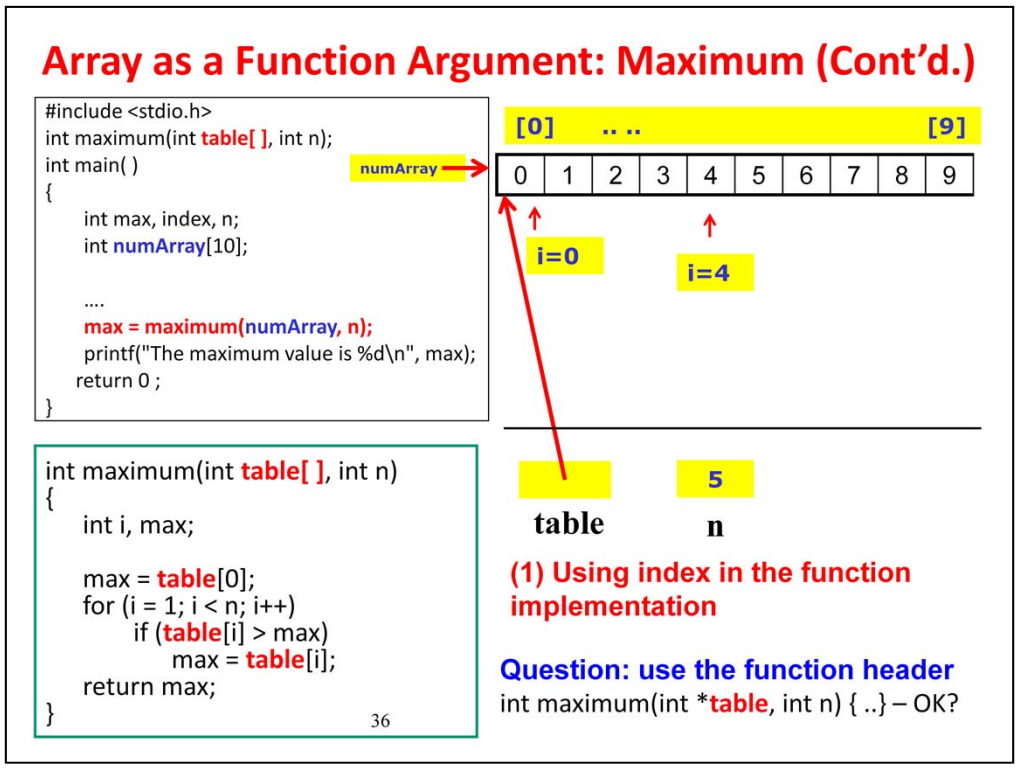
Output

Enter the number of values: 5
Enter 5 values: 1 2 3 4 5
The maximum value is 5

35

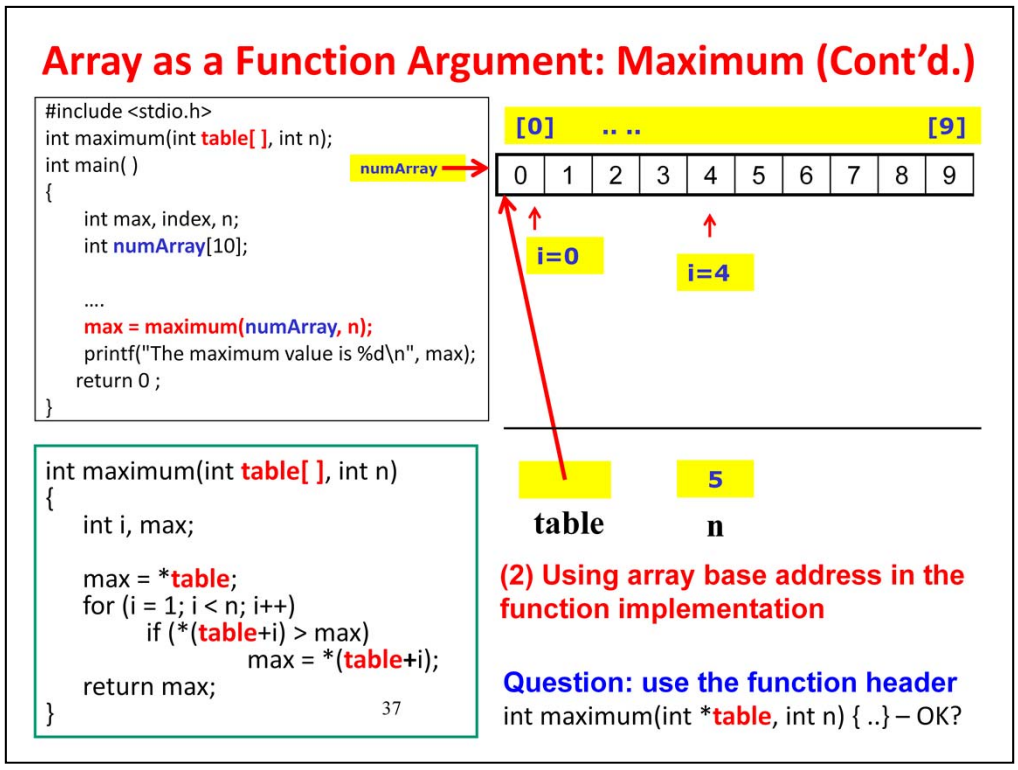
Arrays as Function Arguments: Example

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. 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**.



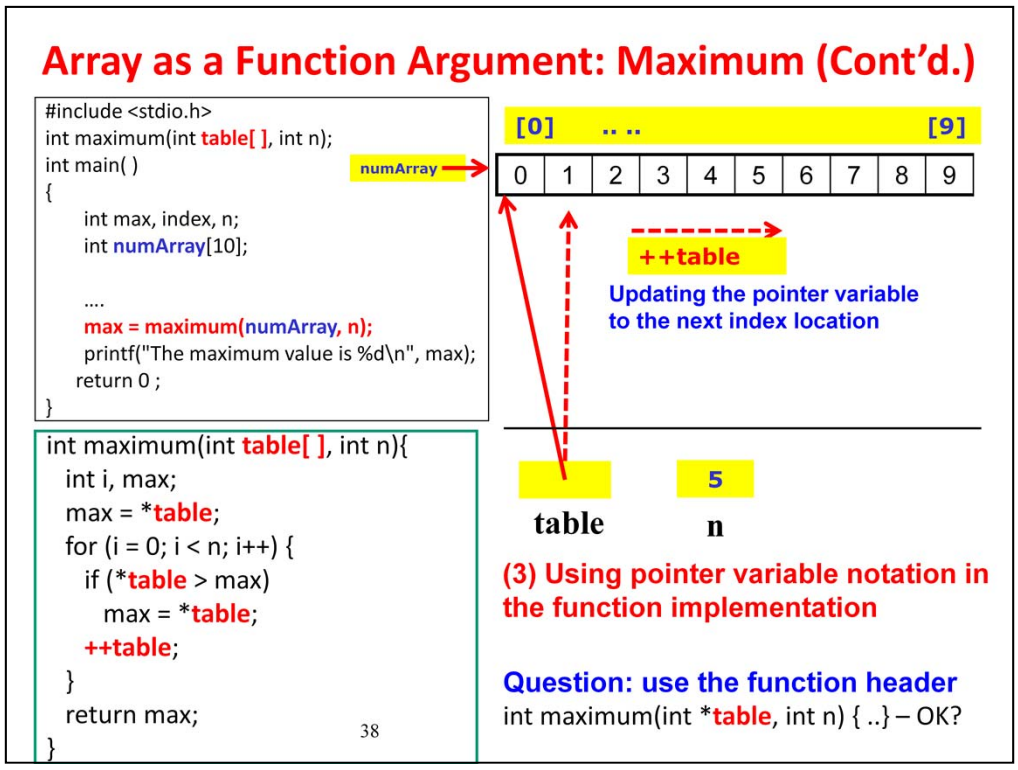
Arrays as Function Arguments: Version 1

The implementation of the function **maximum()** uses array indexes. It has two parameters: **table** and **n**. 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. At the end of the function, the maximum number stored in **max** is passed back to the calling function.



Arrays as Function Arguments: Version 1

The implementation of the function **maximum()** uses array indexes. It has two parameters: **table** and **n**. 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. At the end of the function, the maximum number stored in **max** is passed back to the calling function.



Arrays as Function Arguments: Version 1

The implementation of the function **maximum()** uses array indexes. It has two parameters: **table** and **n**. 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. At the end of the function, the maximum number stored in **max** is passed back to the calling function.

Quiz: What is the output?

39

Quiz: What is the Output?

```
#include <stdio.h>
int main( )
{
    int a[ ] = {1,2,3,4,5,6,7,8,9,0};
    int *p = a;

     printf ("%p\n", p);
     printf ("%p\n", p+9);
     printf ("%d\n", *p+9);
     printf ("%d\n", *(p+9));
     printf ("%d\n", *++p+9);
    return 0 ;
}
```

40

Quiz: What is the output?

Determine the output of the program.

C Operator Precedence Table

Operator	Description	Associativity
( )	Parentheses (function call)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	**Postfix increment/decrement**	
++ --	**Prefix increment/decrement**	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*	**Dereference**	
&	**Address (of operand)**	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	

Quiz: What is the output?

To find the answer, you may need to understand the operator precedence table in C. Note that the precedence of dereferencing and address operators, and their relative positions when compared with the increment/decrement operators. As shown in the table, the precedence of the increment/decrement operators is higher than the dereferencing operator.

What is the Output?

```
#include <stdio.h>
int main( )
{
    int a[ ] = {1,2,3,4,5,6,7,8,9,0};
    int *p = a;

     printf ("%p\n", p);
     printf ("%p\n", p+9);
     printf ("%d\n", *p+9);
     printf ("%d\n", *(p+9));
     printf ("%d\n", *++p+9);
    return 0 ;
}
```

42

0022FEE4 (address a[0])
0022FF08 (address a[9])
10
0
11

Quiz: What is the output?

The outputs are:

0022FEE4 (address a[0])
0022FF08 (address a[9])
10
0
11