5.1
One-dimensional
Arrays

One-dimensional Arrays

1. In the previous chapters, 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.

2. 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.

3. Instead, we can declare a 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**.

4. In this chapter, 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.

5. In arrays, we can categorize them as one-dimensional arrays and two-dimensional (or multi-dimensional) arrays. In this chapter, we focus on discussing one-dimensional arrays.

One-dimensional Arrays

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

2

One-dimensional Arrays

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

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 many operations;
  - Size of arrays cannot be changed while lists can grow and shrink.

3

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.

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, string variables**.


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)**.

Array of size 12

| 1st element	2nd element	Nth element	12th element
index=**0**	index=1	index=N-1	index=**11**
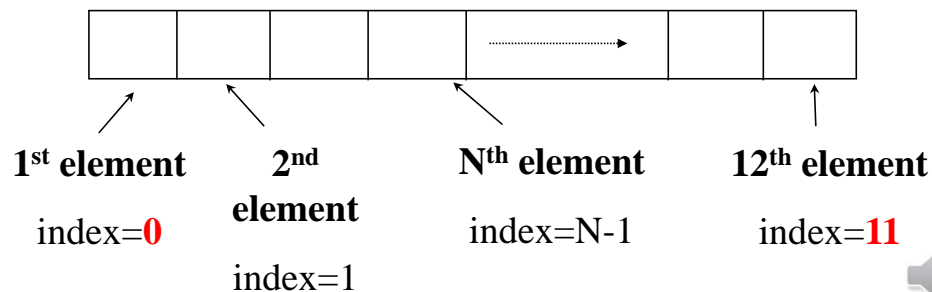
5

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 for an integer depending on machine): **total_memory = sizeof(type_specifier)*array_size;**

	h[0]	h[1]	h[2]	h[3]
int h[4];	?	?	?	?

Element :	1	2	3	4
Memory address :	1021	1023	1025	1027
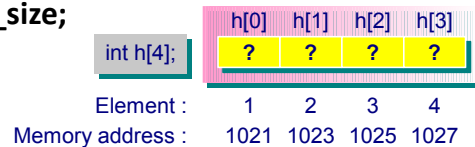
- The size of array must be <u>integer constant</u> or <u>constant expression</u> in declaration:

  e.g. char name[**i**];   // **i** variable ==> illegal

  int states[**i***6];   // **i** 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;
  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**

- <u>Working on array values</u>:
  (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 to display every element's content:

days array	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

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.

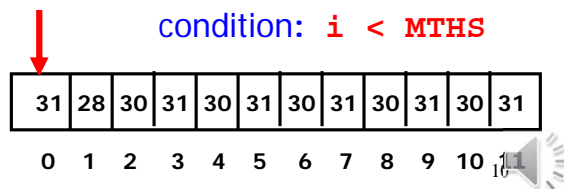3. This is illustrated in this example on using the array **days[]**.

Example 1: Printing Values

```c
#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.

condition: `i < MTHS`

`days`
array

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

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

```c
#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.
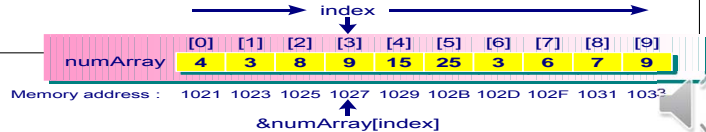
Example 3: Finding the Maximum Value

```
#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.

index

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
numArray	4	3	8	9	15	25	3	6	7	9

Memory address : 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033

&numArray[index]

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_**:
- 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**

Memory address : 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033 1035 1037
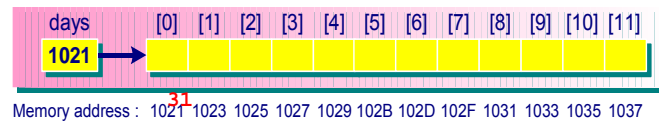
days[0] = 31; /* days[0] contains the value assigned to it */

14

Pointer Constants

1. There is a strong relationship between pointers and arrays. The array name is in fact a pointer constant.

2. When the array **days[]** is declared: **int days[12];** a **pointer constant** with the same name as the array is also created.

3. 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.

4. Assume an integer is represented by 2 bytes (in some older machines) and the array **days** begins at memory location 1021.

5. 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:
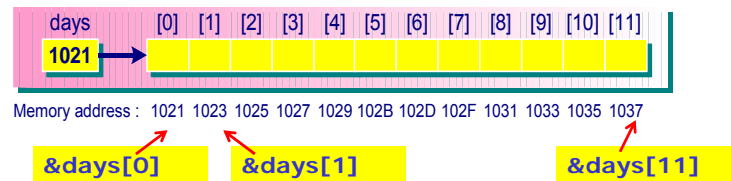
&days[0] - is the **address** of the **1st** element [i.e. 1021]
&days[1] - is the **address** of the **2nd** element [i.e. 1023]
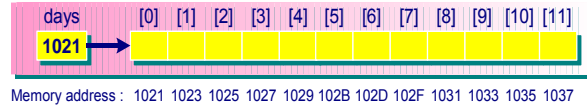**&days[i]** - is the **address** of the **(i+1)th** element

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**

days	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
1021												

Memory address : 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033 1035 1037

- 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.
- Goal – to be able to use the pointer **days** for accessing each array element. How to do that?

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
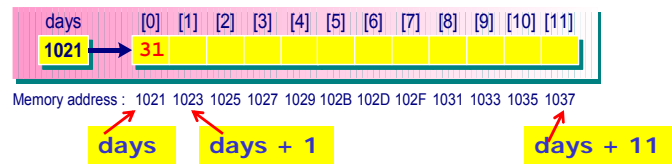
1. The array name **days** is a pointer constant.

2. 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]**

3. 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)**.

4. 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.

5. 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

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()
{                    Pointer constant
    int days[MTHS]= {31,28,31,30,31,30,31,31,30,31,30,31};
    int *day_ptr;
    day_ptr = days;       Pointer variable

    printf("First element = %d\n", *day_ptr);
2.
    day_ptr = &days[3];  /* points to the fourth element */




    printf("Fourth element = %d\n", *day_ptr);
}
```
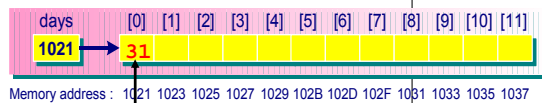
Output
First element = 31
Fourth element = 30

```
days    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
1021            30

Memory address : 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033 1035 1037
            day_ptr   1027
```
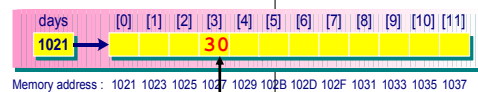
19

Pointer Variables

1. 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.

2. The value stored in **day_ptr** becomes 1027.

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.

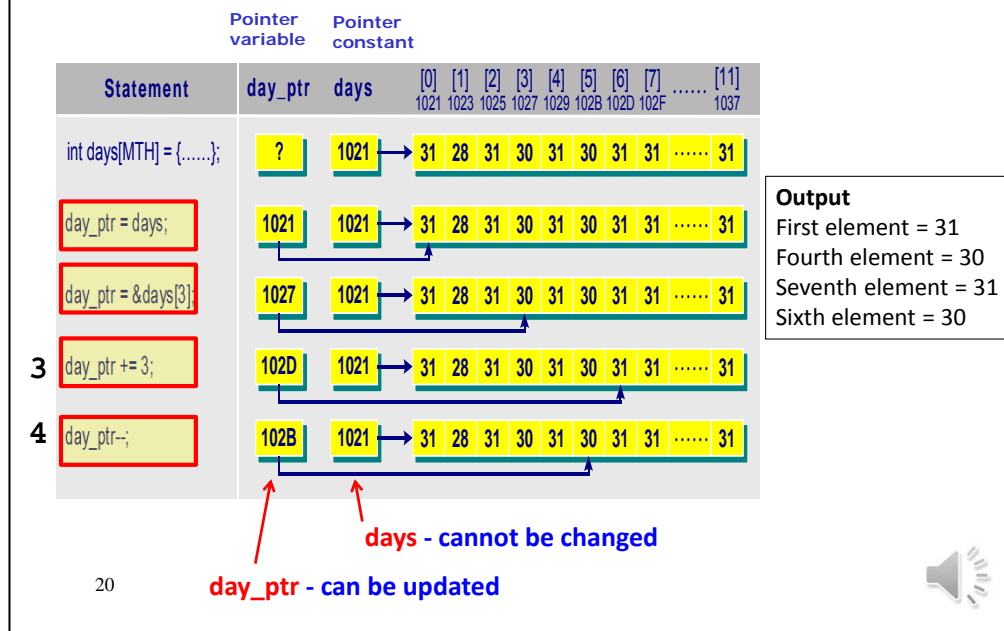2. The **day_ptr** contains the value of 102D, which is the address of the seventh element of the array **days[6]**.

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( )                    Pointer constant
{
    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;
}
    21
numArray
```

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];
}
```

[0]	.. ..	[9]

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

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

```c
#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;
}
```
22

Using index for reading input:

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

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

numArray [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
1021 → 4 3 8 9 15 25 3 6 7 9
Memory address : 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033
ptr
1021 ┤ --------- ptr++ ---------

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 **<u>pointer constant</u>**: 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.
  - We 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 **<u>pointer variables</u>** 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.

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

The **prototype** of the function:

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

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

```
or   void fn2(int table[TABLESIZE])
{
    ....
}
```

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

```
or   void fn3(int *table, int size)
{
    ....
}
```

```
void fn3(int *table, int size);
```

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

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

1. 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.

2. 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.

3. When we pass an array as a function argument, the array is passed using **call by reference** to the function.

4. 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

```c
#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

27

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

i=0   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;
}
```
28

table   n   10

(1) Using array indexing in the function implementation

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: Using Array 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=0

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;
}                                    29
```

table   n   10

(2) Using array base address in the function implementation

29

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: Using Pointer Variable

```c
#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 ;
}
```

```c
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;
}
```

numArray →

[O]   .. ..   [9]

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

++table

Updating the pointer variable to the next index location

table   **n**   10

(3) Using pointer variable notation in the function implementation

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

1. Thank you.