

MODULE 3

ARRAYS AND STRINGS

The basic data types in C such as **char, int, float, and double** are very useful, but they have one major limitation:

- A variable of these types can store **only one value at a time**.

This means they are suitable only for storing **small amounts of data**.

However, in many real-life applications, we often need to **store and process a large collection of data**, such as:

- marks of many students
- list of numbers
- list of names
- readings from a sensor, etc.

Handling such large volumes of data using single variables becomes **difficult and inefficient**.

To overcome this problem, C provides a **derived data type** called **array**.

What is Array?

An array is a fixed-size collection of elements of the same data type stored in contiguous memory locations.

Example uses of arrays:

- A list of 10 integers
- Names of students
- Marks in different subjects

Thus, an array allows us to store **multiple values** of the **same type** under **one common name**, making the program easier to manage and faster to execute.

Example:

```
int Salary [10];
```

ONE DIMENSIONAL ARRAY

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one dimensional array. The subscripts of an array can be integer constants.

Example If we want to represent a set of five numbers, say (35,40,20,57,19), by an array variable number, then we may declare the variable number as :

```
int number[5];
```

	Number[0]
	Number[1]
	Number[2]
	Number[3]
	Number[4]

The values to the array elements can be assigned as follows:

```
Number[0]=35;
```

```
Number[1]=40;
```

```
Number[2]=20;
```

```
Number[3]=57;
```

```
Number[4]=19;
```

Declaration of one-dimensional arrays

Arrays, must be declared before they are used so that the compiler can allocate space for them in memory.

Syntax:

One-dimensional arrays are declared as follows:

type variable-name[size];

Where type can be any valid C data type, variable-name will be a valid C identifier and size indicates the maximum number of elements that can be stored inside the array.

Example: float height[50];

declares the height to be an array containing 50 real elements. The C language treats character strings simply as arrays of characters. The size in a character string represents the maximum number of characters that the string can hold.

Example: char name[10];

declares the name as a character array (string) variable can hold a maximum of 10 characters. Suppose the value of name is “WELL DONE”, then the each character of the string is treated as an element of the array name and is stored in the memory as follows:

'W'
'E'
'L'
'L'
'D'
'O'
'N'
'E'
'\0'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element name[10] holds the null character ‘\0’

Initialization of one-dimensional arrays

After an array is declared, its elements must be initialized. Otherwise, they will contain “garbage”. An array can be initialized at either of the following stage:

1. At compile time
2. At run time

Compile time initialization: We can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

Syntax: `type array-name[size] = {list of values};`

The values in the list are separated by commas.

Example `int number[3]={0,0,0};`

will declare the variable number as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically

```
float total[5] = {0.0, 15.75, -10};
```

will initialize the first 3 elements to **0.0, 15.75 and -10.0** and the remaining 2 elements to **zero**.

Character arrays may be initialized in a similar manner. Thus, the statement

```
char name[] = {'J', 'O', 'H', 'N', '\0'};
```

or

```
char name[] = "JOHN";
```

declares *name* to be an array of five characters, initialized with the string **"JOHN"** ending with the null character.

Run-time Initialization

An array can be explicitly initialized at run time. We can use the function `scanf()` to initialize an array.

Example:

```
int x[3];
scanf("%d %d %d", &x[0], &x[1], &x[2]);
```

Another method to initialize an array is by using for loop.

Example:

```
#include <stdio.h>
int main() {
    int marks[5];
    int i;
    printf("Enter 5 marks:\n");
    for(i = 0; i < 5; i++) {
        scanf("%d", &marks[i]); // reading each element
    }
    printf("You entered:\n");
    for(i = 0; i < 5; i++) {
        printf("%d ", marks[i]);
    }
    return 0;
}
```

Program

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int a[50], i, j, n, temp;
    clrscr();

    printf("Enter the array boundary\n");
    scanf("%d", &n);

    printf("Enter %d array elements\n", n);
    for(i = 1; i <= n; i++)
        scanf("%d", &a[i]);

    for(i = 1; i <= n; i++)
    {
        for(j = i + 1; j <= n; j++)
        {
            if(a[i] > a[j])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }

    printf("Sorted Array\n");
    for(i = 1; i <= n; i++)
        printf("%d\n", a[i]);

    getch();
}
```

OUTPUT:

```
Enter the array boundary
5
Enter 5 array element
45
67
3
90
2
Sorted array
2
3
45
67
90
_
```

TWO-DIMENSIONAL ARRAYS

A two-dimensional array is an array used to store data in **table form** (rows and columns). It is mainly used to represent **matrices** or data arranged like a grid.

A 2D array can be visualized as a **collection of one-dimensional arrays**, where each row is itself an array.

Syntax

```
type array_name[row_size][column_size];
```

- **type** → any valid C data type (int, float, char, etc.)
- **array_name** → valid C identifier
- **row_size** → number of rows
- **column_size** → number of columns

Example

```
int marks[3][4];
```

This declares a 2D array with **3 rows** and **4 columns**, i.e., 12 integer elements.

Representation of a 2D Array in Memory

A two-dimensional array is stored in **row-major order**, meaning:

- First the entire **row 0** is stored,
- Then **row 1**,
- Then **row 2**, and so on.

Example for `int a[2][3]`:

| `a[0][0]` | `a[0][1]` | `a[0][2]` | `a[1][0]` | `a[1][1]` | `a[1][2]` |

Stored continuously in memory.

Column1	Column2	Column3	
↓	↓	↓	
<code>[0][0]</code>	<code>[0][1]</code>	<code>[0][2]</code>	
310	275	365	--→Row1
<code>[1][0]</code>	<code>[1][1]</code>	<code>[1][2]</code>	
10	190	325	--→Row2
<code>[2][0]</code>	<code>[2][1]</code>	<code>[2][2]</code>	
405	235	240	--→Row3
<code>[3][0]</code>	<code>[3][1]</code>	<code>[3][2]</code>	
310	275	365	--→Row4

Initializing Two-Dimensional Arrays

Two-dimensional arrays may be initialized by writing a list of initial values inside braces `{ }` after the declaration.

Examples

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

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

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

If some values are missing in the initializer list, the remaining elements are **automatically set to zero**.

Example Program – Addition of Two Matrices

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int a[10][10], b[10][10], d[10][10];
    int r, c, i, j;

    clrscr();

    printf("Enter the row and column: ");
    scanf("%d%d", &r, &c);

    printf("Enter the first matrix\n");
    for(i = 0; i < r; i++)
    {
        for(j = 0; j < c; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }

    printf("Enter the second matrix\n");
    for(i = 0; i < r; i++)
    {
        for(j = 0; j < c; j++)
        {
            scanf("%d", &b[i][j]);
        }
    }

    printf("The Result matrix\n");
    for(i = 0; i < r; i++)
    {
        for(j = 0; j < c; j++)
        {
            d[i][j] = a[i][j] + b[i][j];
            printf("%d\t", d[i][j]);
        }
        printf("\n");
    }
}
```

```
    }  
  
    getch();  
}
```

OUTPUT:

```
Enter the row and column      3 3  
Enter the first matrix  
1 2 3  
4 5 6  
7 8 9  
Enter the second matrix  
1 2 3  
4 5 6  
7 8 9  
Result matrix  
2      4      6  
8      10     12  
14     16     18  
_
```

MULTIDIMENSIONAL ARRAY

C allows arrays of **three or more dimensions**.
The exact limit depends on the compiler.

Syntax

```
type array_name[s1][s2][s3] ... [sm];
```

Where:

- **type** → any valid C data type
- **array_name** → a valid C identifier
- **s1, s2, s3 ... sm** → sizes of each dimension

A multidimensional array can store large amounts of data in multiple levels.

Examples

```
int survey[3][5][12];
```

($3 \times 5 \times 12$ storage locations)

```
float table[5][4][5][3];
```

($5 \times 4 \times 5 \times 3$ storage locations)

CHARACTER ARRAYS AND STRINGS

A **string** is a sequence of characters treated as a single data item.

Any characters written inside **double quotes** form a **string constant**.

Example

```
"well done"
```

Strings are stored in **character arrays**, ending with a **null character** `'\0'`.

Common Operations on Strings

1. Reading and Writing strings
2. Combining strings together (Concatenation)
3. Copying one string to another
4. Comparing strings for equality
5. Extracting a part of a string

Declaring and Initializing String Variables

C does not support strings as a separate data type. However, it allows us to represent strings using **character arrays**.

Therefore, in C, a *string variable* is simply a character array declared with any valid C identifier.

Syntax

```
char string_name[size];
```

The **size** specifies the maximum number of characters the string can hold. C automatically adds a **null character** (`'\0'`) at the end of every string. So, the size should be the number of characters **plus one** for the null terminator.

Ways to Initialize a Character Array

C permits two forms of initialization:

1. Using a string constant

```
char city[9] = "NEW YORK";
```

2. Using a list of characters

```
char city[9] = {'N','E','W',' ','Y','O','R','K','\0'};
```

The size must be **9**, because "NEW YORK" has **8 characters** and one space is reserved for the null terminator.

Automatic Size Determination

C also allows initialization without specifying the size.

In such cases, the size is determined automatically based on the number of characters:

```
char city[] = "NEW YORK";
```

Declaring a Larger Size

You can also declare an array larger than the string you assign:

```
char word[10] = "GOOD";
```

Here, the compiler:

- Creates an array of size 10
- Stores "GOOD" followed by `'\0'`
- Sets all remaining elements to **NULL (0)**

G	O	O	D	\0	\0	\0	\0	\0	\0
---	---	---	---	----	----	----	----	----	----

Reading Strings from Terminal

The familiar input function **scanf** can be used with the **%s** format specifier to read a string.

```
char address[10];
scanf("%s", address);
```

The `scanf` function automatically terminates the string with a **null character** (`'\0'`).

Therefore, the character array must be large enough to hold the input string **plus** the null character.

Using Field Width with %s

We can also specify the field width using the form **%ws** in the `scanf` statement to read a fixed number of characters.

```
scanf("%ws", name);
```

Problems with this method

1. **If the width (w) is equal to or larger than the number of characters typed,**
→ the entire string will be stored in the variable.
2. **If the width (w) is smaller than the length of the input string,**
→ extra characters will be **truncated** and left unread.

Example:

```
char name[10];
scanf("%5s", name);
```

If the input is **KRISHNA**, it will be stored as:

K	R	I	S	H	\0	?	?	?
---	---	---	---	---	----	---	---	---

(Only the first 5 characters are stored.)

Reading a Line of Text

C supports a format specification called the **edit set conversion code** `%[...]` which can read a line containing various characters, including **whitespace**.

Example:

```
char line[80];
scanf("%[^\n]", line);
printf("%s", line);
```

This reads an entire line from the keyboard (until newline) and prints it back on the screen.

Using `getchar()` and `gets()` Functions

`getchar()`

The function `getchar()` is used to read **one character at a time** from the input. When used repeatedly inside a loop, it can read an entire line of text into a character array.

- Reading stops when the **newline** (`'\n'`) is entered.
- After this, a **null character** (`'\0'`) is added at the end of the string.

Example:

```
char ch;
ch = getchar();
```

`gets()`

The function `gets()` is used to read an entire string (a line of text):

```
gets(str);
```

Here, **str** is a properly declared character array.

`gets()` reads characters from the keyboard **until a newline is encountered**, then it automatically adds a **null character** at the end.

Example:

```
char line[80];
gets(line);
printf("%s", line);
```

This reads a complete line from the keyboard and prints it on the screen.

Writing Strings to Screen

We commonly use `printf()` with the `%s` format specifier to display strings.

Example:

```
printf("%s", name);
```

Features of %s Format Specification

1. If the **field width** is **less** than the string length, the **entire string** is printed.
2. The integer value **after the decimal point** specifies how many characters to print.
3. If the number of characters to be printed is **zero**, nothing is printed.
4. A **minus sign (-)** causes the string to be **left-justified**.
5. The specification **%.ns** prints only the **first n characters** of the string.

Example Program

```
main()
{
    char country[15];
    printf("*123456789*");

    printf("\nEnter the string: ");
    scanf("%s", country);

    printf("%15s\n", country);    // Right-justified, width 15
    printf("%5s\n", country);    // Width 5
    printf("%15.6s\n", country); // First 6 chars, width 15
    printf("%-15.6s\n", country); // First 6 chars, left-justified
    printf("%15.0s\n", country); // Prints nothing (0 characters)
    printf("%.3s\n", country);   // First 3 characters
    printf("%s\n", country);     // Entire string
}
```

Sample Output

```
*123456789*
```

```
Enter the string: Unitedkingdom
```

```
    Unitedkingdom
```

```
Unitedkingdom
```

```
    United
```

```
United
```

```
(prints nothing)
```

```
Uni
```

```
Unitedkingdom
```

Using putchar() and puts() Functions

putchar()

This function is used to output the value of a **single character**.

Example:

```
char ch = 'A';  
putchar(ch);
```

puts()

The puts() function is used to print **a string**.

```
puts(str);
```

Here, **str** is a character array containing a string.

puts() prints the string and then automatically moves the cursor to the **next line**.

String Handling Functions

Function	Action
strcat()	Concatenates two strings
strcmp()	Compares two strings
strcpy()	Copies one string to another
strlen()	Finds the length of a string

strcat() – String Concatenation

This function joins two strings together:

```
strcat(string1, string2);
```

- string2 is appended to string1.
- The null character at the end of string1 is removed and replaced by the first character of string2.
- string2 **remains unchanged**.

You can also append a **string constant**:

```
strcat(name, "City");
```

C also allows **nesting**:

```
strcat(strcat(string1, string2), string3);
```

strcmp() – String Comparison

This function compares two strings:

```
strcmp(string1, string2);
```

- Returns **0** if both strings are equal.
- Returns a **positive** or **negative** value based on the difference between the *first unmatched character*.

Arguments may be string variables or string constants.

strcpy() – String Copy

This function works like a string assignment operator:

```
strcpy(string1, string2);
```

Copies the contents of **string2** into **string1**.

After execution, **string1** becomes an exact copy of **string2**.

strlen() – String Length

This function counts and returns the number of characters in a string:

```
n = strlen(string);
```

- **n** receives the length of the string.
- Counting stops at the **first null character** (`'\0'`).
- The argument may also be a string constant.

Example Program Using String Functions

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main()
{
    char s1[30] = "Hello";
    char s2[30] = "Welcome";
    char s3[30];

    clrscr();

    printf("The first string: %s", s1);
    printf("\nThe second string: %s", s2);

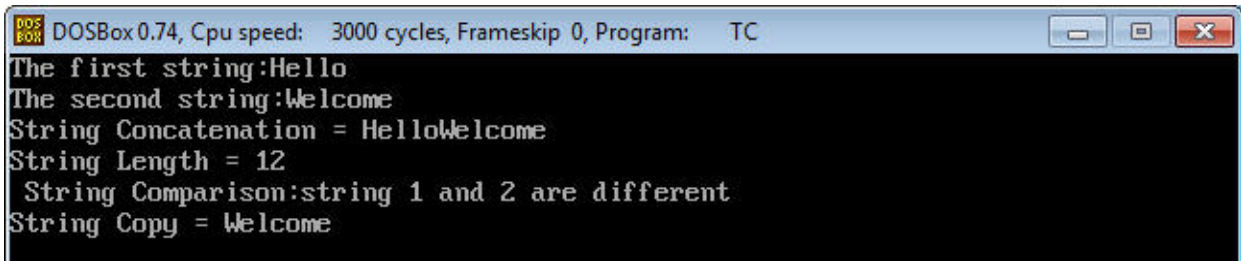
    // String concatenation
    strcat(s1, s2);
    printf("\nString Concatenation = %s", s1);

    // String length
    printf("\nString Length = %d", strlen(s1));

    // String comparison
    printf("\nString Comparison: ");
    if (strcmp(s1, s2) == 0)
    {
        printf("string1 and string2 are equal");
    }
    else
    {
        printf("string1 and string2 are different");
    }

    // String copy
    strcpy(s3, s2);
    printf("\nString Copy = %s", s3);

    getch();
}
```

A screenshot of a DOSBox window. The title bar reads "DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: TC". The window contains a black terminal area with white text. The text shows the output of a C program: "The first string:Hello", "The second string:Welcome", "String Concatenation = HelloWelcome", "String Length = 12", "String Comparison:string 1 and 2 are different", and "String Copy = Welcome".

```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: TC
The first string:Hello
The second string:Welcome
String Concatenation = HelloWelcome
String Length = 12
String Comparison:string 1 and 2 are different
String Copy = Welcome
```

Other String Functions

1. strncpy()

- Used to copy **only the left-most n characters** of the source string into the target string.
- It is a **three-parameter** function.

Syntax:

```
strncpy(s1, s2, n);
```

Example meaning:

```
strncpy(s1, s2, 5); → Copies the first 5 characters of string s2 into string s1.
```

2. strncmp()

- Compares the **first n characters** of $s1$ and $s2$.
- Returns:
 1. **0** → if both substrings are equal
 2. **Negative value** → if substring of $s1 <$ substring of $s2$
 3. **Positive value** → if substring of $s1 >$ substring of $s2$

Syntax:

```
strncmp(s1, s2, n);
```

3. strncat()

- Concatenates the **left-most n characters** of $s2$ to the end of $s1$.
- Three-parameter function.

Syntax:

```
strncat(s1, s2, n);
```

4. strstr()

- Used to **locate a substring inside another string**.

Syntax:

```
strstr(s1, s2);
```

- Searches `s1` to check if `s2` exists inside it.
- Returns:
 - **Pointer to the first occurrence** of `s2` in `s1` (if found)
 - **NULL pointer** (if not found)

5. strchr()

- Finds the **first occurrence** of a character in a string.

Syntax:

```
strchr(s1, 'm');
```

This locates the **first 'm'** in string `s1`.

6. strrchr()

- Finds the **last occurrence** of a character in a string.

Syntax:

```
strrchr(s1, 'm');
```

This locates the **last 'm'** in string `s1`.