

### Unit 3: Arrays and Strings

The type of variables that we declared so far were all **scalar variables**. The scalar variables are the ones that can be used only to store a single value. They cannot be further subdivided or separated into another data type.

Examples of scalar variables are

```
int a;  
char c;  
float interest;
```

The above mentioned scalar variables can be used to hold only a single value and cannot be further divided or updated. On the other hand we can have a derived data type that can hold a set of elements of the same data type within.

An **array** can be defined as a data structure of a fixed size that can hold a set of elements of same data type that are stored in an ordered fashion. All the elements of the array can be accessed using the same name but with different index values. Each of the values stored in an array can be individually retrieved or updated.

The advantages of using arrays are:

1. Multiple data items of same data type can be accessed using single name
2. Arrays can be used to implement matrices
3. Data structure like queue , linked list can be accessed using arrays

The limitations of using arrays are

1. We must know in advance that how many elements are to be stored in an array
2. Array is static structure. It means that array is of fixed size. The memory which is allocated to array can not be increased or reduced.
3. Insertion and deletion are quite difficult in array. As the elements are stored in consecutive memory locations.

#### One-Dimensional Array

The below example shows declaration and usage of a one-dimensional array.

The syntax for declaration of a one-dimensional array is as below

```
<data_type> <array_name>[array_size];
```

Example:

```
int number[10];
```

The above declaration states that **number** the name of the array which holds the elements which are of type integers. The size of the array is mentioned in the square brackets [10]. Here the size of the array is declared as 10, indicating that it can hold 10 elements.

The elements of an array can be accessed using the index values. A size of 10 indicates that the index of array elements has a range of 0 to 9.

The first element of the array is `number[0]`, the second element is `number[1]` and so on the tenth element in the array is `number[9]`.

The array subscripts range from 0 to  $(\text{array\_size} - 1)$ .

The lower bound = 0

Upper bound = size - 1

Size = upper bound + 1

Each element of the array can be accessed using the index constant by using the syntax `<array_name>[<index_value>]`

Example: the 5th element of the array **`number[10]`** can be accessed as `number[5]`.

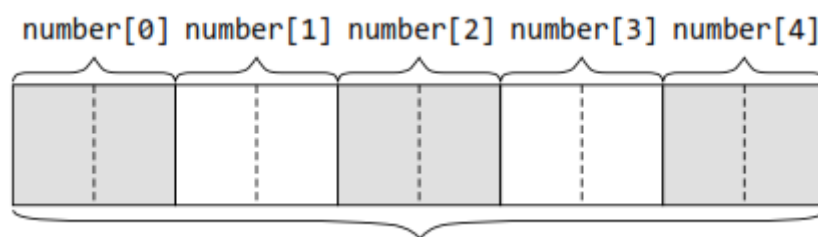


Figure 1: Identifying the array elements.

The individual array elements are stored in the memory location as shown in figure 1.

Important points to remember while using arrays.

1. Name of the array should be meaningful and naming an array follows the same rules as for naming a variable.
2. The size of the array should be known at compile time. Hence an array size either has to be defined as fixed by the programmer or should be declared as an integer constant.

Below two are the examples showing some of the errors made while declaring an array and are invalid.

The following declaration is invalid.

```
#include <stdio.h>
int main()
{
    double x[], y[];
    ...
    return 0;
}
```

No size is specified.

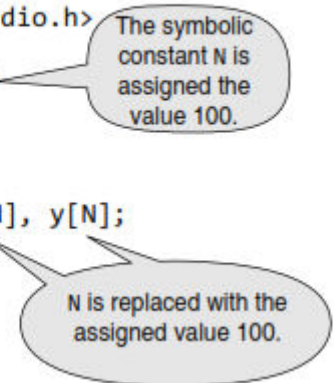
```
#include <stdio.h>
int main()
{
    int N;
    double x[N], y[N];
    ...
    scanf("%d",&N);
    return 0;
}
```

An integer variable is used as size of the array. It must be an integer constant or constant integer variable.

Please note that it is mandatory to specify the array size at compile time. An integer value cannot be used to define the size of an array, rather an integer constant can be used.

It is always convenient to declare an array size using a symbolic constant or declare an array of fixed size as shown in an example below.

```
/* Define a symbolic constant for the size of
the array */
#include <stdio.h>
#define N 100
int main()
{
    double x[N], y[N];
    ...
    return 0;
}
```



The diagram consists of two speech bubbles. The first bubble points to the line `#define N 100` and contains the text: "The symbolic constant N is assigned the value 100." The second bubble points to the array declarations `double x[N], y[N];` and contains the text: "N is replaced with the assigned value 100."

### Initializing the values of an array

An one-dimensional array can be initialized with values as shown below.

```
int mark[5] = {19, 10, 8, 17, 9};
```

Here an array by name mark of size 5 is declared which is holding the data of type integers consisting of the elements 19,10,8,17 and 9.

The elements are stored in the array locations as

```
mark[0] = 19
mark[1] = 10
mark[2] = 8
mark[3] = 17
mark[4] = 9
```

Another example showing an array consisting of elements of the data type float is shown below

```
double a[5] = {3.67, 1.21, 5.87, 7.45, 9.12};
```

An array can also be initialized without mentioning the size. In such cases the the C compiler will deduce the size automatically based on the number of elements initialized.

```
int arr[] = {3,1,5,7,9};
```

### Example to show how to load data into array using scanf()

```
include <stdio.h>
#define ARRAY_SIZE 10
int main()
```

```

{
    int index, a[ARRAY_SIZE];
    for(index = 0; index < ARRAY_SIZE; index++)
    {
        scanf("%d",&a[index]);
        printf("a[%d] = %d\n", index, a[index]);
    }
    printf("\n");
    return 0;
}

```

The output of the below program would look like  
Where the array elements entered are 35,45,65,70,15,60,10,50,40 and 30.

```

35
a[0] = 35
45
a[1] = 45
65
a[2] = 65
70
a[3] = 70
15
a[4] = 15
60
a[5] = 60
10
a[6] = 10
50
a[7] = 50
40
a[8] = 40
30
a[9] = 30

```

An array element should always be accessed with index.

For an array int a[10] writing a=0; is wrong or similarly copying an array int b[10] such as a=b; is also wrong.

Important points to remeber:

### **To load data into array:**

```

for(i=0 ; i< 10; i++)
    scanf("%d", a[i]);

```

### **Printing an array**

```

for(i=0 ; i< 10; i++)
    printf("%d", a[i]);

```

Few of the operations that can be done on an array are listed below.

These operations include the following, for an array named 'ar'.

1. To increment the *i*th element, the given statements can be used.

```
ar[i]++;  
ar[i] += 1;  
ar[i] = ar[i] + 1;
```

2. To add *n* to the *i*th element, the following statements may be used,

```
ar[i] += n;  
ar[i] = ar[i] + n;
```

3. To copy the contents of the *i*th element to the *k*th element, the following statement may be written.

```
ar[k] = ar[i];
```

4. To copy the contents of one array 'ar' to another array 'br', it must again be done one by one.

```
int ar[10],br[10];  
for(i = 0; i < 10; i = i + 1)  
br[i] = ar[i];
```

5. To exchange the values in *ar[i]* and *ar[k]*, a 'temporary' variable must be declared to hold one value, and it should be the same data type as the array elements being swapped. To perform this task, the following C statements are written

```
int temp;  
temp = ar[i]; /* save a copy of value in ar[i] */  
ar[i] = ar[j]; /* copy value from ar[j] to ar[i] */  
ar[j] = temp; /* copy saved value of ar[i] to ar[j] */
```

## References to elements outside of the array bounds

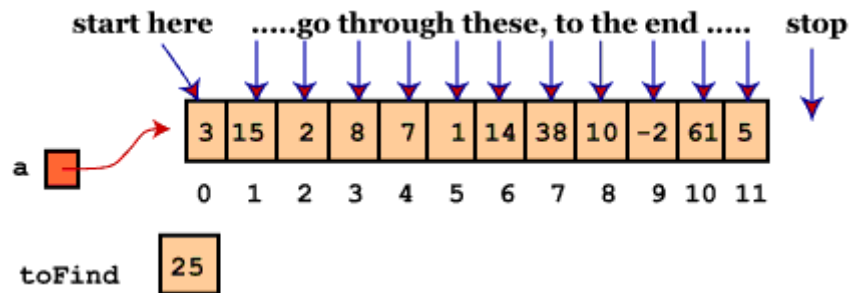
It is important to realize that there is no array bound checking in C. If an array *x* is declared to have 100 elements, the compiler will reserve 100 contiguous, appropriately sized slots in computer memory on its behalf. The contents of these slots can be accessed via expressions of the form *x[i]*, where the integer *i* should lie in the range 0 to 99. As seen, the compiler interprets *x[i]* as the contents of the memory slot which is *i* slots away from the beginning of the array. Obviously, accessing elements of an array that do not exist will produce some sort of error. Exactly what sort of error is very difficult to say—the program may crash, it may produce an absurdly incorrect output, it may produce plausible but incorrect output, it may even produce correct output—it all depends on exactly what information is being stored in the memory locations surrounding the block of memory reserved for *x*. This type of error can be extremely difficult to debug, since it may not be immediately apparent that something has gone wrong when the program is executed.

Example Programs using 1 dimensional arrays

## Linear Search

A simple approach is to do **linear search**, i.e

1. Start from the leftmost element of arr[] and one by one compare search with each element of arr[]
2. If search matches with an element, return the index.
3. If search doesn't match with any of elements, print as not found.



```
#include <stdio.h>
#define N 10
void main()
{
    int search;
    int array[N];
    int i, flag=0;
    for(i=0;i<N; i++)
    {
        printf("Enter the element at location %d \n", i);
        scanf("%d",&array[i]);
    }

    for(i=0;i<N; i++)
    {
        printf("element at location %d is %d \n", i, array[i]);
    }

    printf("Enter the number you would wish to search\n");
    scanf("%d", &search);

    for(i=0;i<N; i++)
    {
        if(array[i]==search)
        {
            printf("Number found at location %d\n", i);
            flag++;
        }
    }
    if(flag==0)
        printf("Number not found");
    else
        printf("Occurences = %d ", flag);
}
```

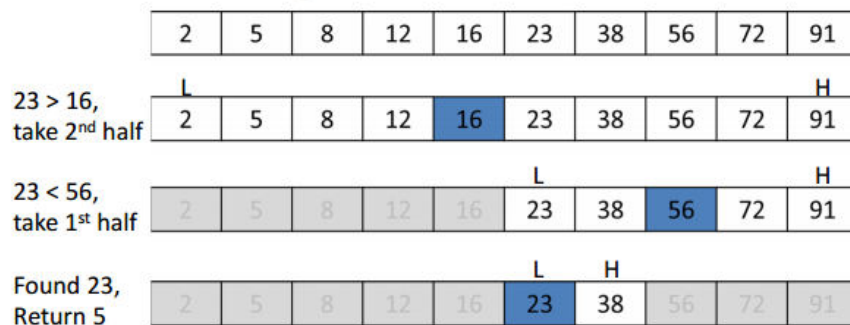
## Binary Search Using Arrays

**Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item

in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

1. Compare search with the middle element.
2. If search matches with middle element, we return the mid index.
3. Else If search is greater than the mid element, then search can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (search is smaller) recur for the left half.
5. If element is found, then print as found along with its index. If not found then print as not found.

**If searching for 23 in the 10-element array:**



```
#include <stdio.h>

int main()
{
    int c, first, last, middle, n, search, array[100];

    printf("Enter number of elements\n");
    scanf("%d",&n);

    printf("Enter %d integers\n", n);

    for (c = 0; c < n; c++)
        scanf("%d",&array[c]);

    printf("Enter value to find\n");
    scanf("%d", &search);

    first = 0;
    last = n - 1;
    middle = (first+last)/2;
    printf("%d is the middle location \n",middle);
    while (first <= last) {
        if (array[middle] < search)
            first = middle + 1;
        else if (array[middle] == search) {
            printf("%d found at location %d.\n", search, middle+1);
            break;
        }
        else
            last = middle - 1;

        middle = (first + last)/2;
    }
}
```

```

        if (first > last)
            printf("Not found! %d isn't present in the list.\n", search);

        return 0;
    }

```

### **Bubble Sort using one-dimensional array**

```

#include <stdio.h>

int main()
{
    int c, first, last, middle, n, search, array[100];

    printf("Enter number of elements\n");
    scanf("%d",&n);

    printf("Enter %d integers\n", n);

    for (c = 0; c < n; c++)
        scanf("%d",&array[c]);

    printf("Enter value to find\n");
    scanf("%d", &search);

    first = 0;
    last = n - 1;
    middle = (first+last)/2;
    printf("%d is the middle location \n",middle);
    while (first <= last) {
        if (array[middle] < search)
            first = middle + 1;
        else if (array[middle] == search) {
            printf("%d found at location %d.\n", search, middle+1);
            break;
        }
        else
            last = middle - 1;

        middle = (first + last)/2;
    }
    if (first > last)
        printf("Not found! %d isn't present in the list.\n", search);

    return 0;
}

```

### **Fibonacci Series using a one-dimensional array**

```

#include <stdio.h>
int main()
{
    int fib[15];
    int i;
    fib[0] = 0;
    fib[1] = 1;
    for(i = 2; i < 15; i++)
        fib[i] = fib[i-1] + fib[i-2];
    for(i = 0; i < 15; i++)
        printf("%d\n", fib[i]);
    return 0;
}

```



```
}
```

## Strings: One-Dimensional Character arrays

A string in C is represented as an array of characters. The end of each string is marked with a special character called as *null character* (NUL) which is a character whose all bits are zero.

Strings can be declared as one dimensional arrays

```
char str[30];  
char text[80];
```

### Initializing a string constant

Character arrays or strings allow a shorthand initialization, for example,

```
char str[9] = "I like C";
```

which is the same as

```
char str[9] = {'I',' ','l','i','k','e',' ','C','\0'};
```

Whenever a string, enclosed in double quotes, is written, C automatically creates an array of characters containing that string, terminated by the `\0` character. C language allows the alternative notation that is always used in practice. The rules for writing string constants are exactly the same as those that were discussed earlier when the use of `printf()` was introduced. It should be noted that the size of the aggregate 'msg' is six bytes, five for the letters and one for the terminating NUL.

```
char msg[] = "Hello";
```

There is one special case where the null character is not automatically appended to the array. This is when the array size is explicitly specified and the number characters during initialization completely fills the array size. For example,

```
char c[4] = "abcd";
```

Here, the array `c` holds only the four specified characters, `a`, `b`, `c`, and `d`. No null character terminates the array.

### Printing a string

A string can be printed using `%s` format/conversion specifier. Width and precision specifications can be used with `%s`.

example to print a string called `name`, we can write `printf()` statement as shown below

```
printf("%7.3s", name);
```

7 indicates that the total field width is 7, 3 indicates that only the first three characters should be printed. If we prefix 7 with a negative sign then the content is printed with left justification, else it is printed with right justification.

The following points should be noted.

1. When the field width is less than the length of the string, the entire string is printed.
2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed as left justified.

Example program:

<pre>#include &lt;stdio.h&gt; int main() {     char s[]="Hello, World";     printf("&gt;&gt;%s&lt;&lt;\n",s);     printf("&gt;&gt;%20s&lt;&lt;\n",s);     printf("&gt;&gt;%-20s&lt;&lt;\n",s);     printf("&gt;&gt;%.4s&lt;&lt;\n",s);     printf("&gt;&gt;%-20.4s&lt;&lt;\n",s);     printf("&gt;&gt;%20.4s&lt;&lt;\n",s);     return 0; }</pre>	<pre>Output: &gt;&gt;Hello, World&lt;&lt; &gt;&gt;      Hello, World&lt;&lt; &gt;&gt;Hello, World      &lt;&lt; &gt;&gt;Hell&lt;&lt; &gt;&gt;Hell              &lt;&lt; &gt;&gt;                  Hell&lt;&lt;</pre>
---	--

There is another way to print a string using non-formatted printing functions like puts().

```
#include <stdio.h>
int main()
{
    char s[]="Hello, World";
    puts(s);
    return 0;
}
```

Output of the program:

Hello, World

The library function sprintf() is similar to printf(). The only difference is that the formatted output is written to a memory area rather than directly to a standard output. It is particularly useful when it is necessary to construct formatted strings in memory for subsequent transmission over a communications channel or to a special device. Its relationship with printf() is similar to the relationship between sscanf() and scanf().

Example Program:

<pre>#include &lt;stdio.h&gt; int main() {     char buffer[50];     int a = 10, b = 20, c;     c = a + b;     sprintf(buffer, "Sum of %d and %d is %d", a, b, c);      // The string "sum of 10 and 20</pre>	<pre>Output: Sum of 10 and 20 is 30  (In the code note that the above statement is printed because of printf() statement in the last and not because of the sprintf(). sprintf() writes the string into a memory location rather than printing</pre>
--	--

<pre> is 30" is stored // into buffer instead of printing on stdout printf("%s", buffer);  return 0; } </pre>	<pre> it on the output display) </pre>
---	--

### Getting a string input:

A string input can be obtained from one of the below mentioned ways

1. Using %s and scanf()
2. Using %c and scanf()
3. Using %s along with a scanset
4. Using gets() function
5. Using getchar() and putchar() to read one character at a time.

One possible way to read in a string is by using scanf. However, the problem with this, is that if you were to enter a string which contains one or more spaces, scanf would finish reading when it reaches a space, or if return is pressed. As a result, the string would get cut off. So we could use the gets function

A gets takes just one argument - a char pointer, or the name of a char array, but don't forget to declare the array / pointer variable first.

A puts function is similar to gets function in the way that it takes one argument - a char pointer. This also automatically adds a newline character after printing out the string. Sometimes this can be a disadvantage, so printf could be used instead.

```

#include <stdio.h>

int main() {
    char array1[50];
    char *array2;

    printf("Now enter another string less than 50");
    printf(" characters with spaces: \n");
    gets(array1);

    printf("\nYou entered: ");
    puts(array1);

    printf("\nTry entering a string less than 50");
    printf(" characters, with spaces: \n");
    scanf("%s", array2);

    printf("\nYou entered: %s\n", array2);

    return 0;
}

```

```
}
```

This will produce following result:

```
Now enter another string less than 50 characters with spaces:
hello world
```

```
You entered: hello world
```

```
Try entering a string less than 50 characters, with spaces:
hello world
```

```
You entered: hello
```

### **Getting a string input using scanf.**

Using `%[^\n]%*c` inside scanf

Example : `scanf("%[^\n]%*c", str);`

```
#include <stdio.h>
int main()
{
    char str[20];
    scanf("%[^\n]%*c", str);
    printf("%s", str);

    return 0;
}
```

Explanation : Here, `[]` is the scanset character. `^\n` tells to take input until newline doesn't get encountered. Then, with this `%*c`, it reads newline character and here used `*` indicates that this newline character is discarded.

```
int main()
{
    char str[50];
    printf("Enter a string in lower case:");
    scanf("%[a-z]", str);
    printf("The string was : %s\n", str);
    return 0;
}
```

Three sample runs are given below.

- (a) Enter a string in lower case: hello world  
The string was: hello world
- (b) Enter a string in lower case: hello, world  
The string was: hello
- (c) Enter a string in lower case: abcd1234  
The string was : abcd

In the second case, the character, ‘,’ (comma) is not in the specified range. Note that in all cases, conversion is terminated by the input of something other than a space or lower-case letter.

### Printing a string into buffer using sprintf()

```
#include <stdio.h>
int main()
{
    char buffer[50];
    int a = 10, b = 20, c;
    c = a + b;
    sprintf(buffer, "Sum of %d and %d is %d", a, b, c);

    // The string "sum of 10 and 20 is 30" is stored
    // into buffer instead of printing on stdout
    printf("%s", buffer);

    return 0;
}
```

In the above program the function sprintf() prints the data into character array buffers rather than to the output display. The contents of the buffer are printed using the printf() statement in the end of the program.

### String manipulation functions available in string.h

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if str1 < str2. Returns >0 if str1 > str2
strcmpi ( )	Same as strcmp() function. But, this function negotiates case. “A” and “a” are treated as same.
strchr ( )	Returns pointer to first occurrence of char in str1
strrchr ( )	last occurrence of given character in a string is found
strstr ( )	Returns pointer to first occurrence of str2 in str1
strrstr ( )	Returns pointer to last occurrence of str2 in str1

strdup ( )	Duplicates the string
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string
strset ( )	Sets all character in a string to given character
strnset ( )	It sets the portion of characters in a string to given character
strtok ( )	Tokenizing given string using delimiter

---

## strlen

**strlen(s1) calculates the length of string s1.**

```
#include <stdio.h>
#include <string.h>
int main()
{
    char name[ ]= "Hello";
    int len1, len2;
    len1 = strlen(name);
    len2 = strlen("Hello World");
    printf("length of %s = %d\n", name, len1);
    printf("length of %s = %d\n", "Hello World", len2);
    return 0;
}
```

### Output

```
length of Hello = 5
length of Hello World = 11
```

strlen doesn't count '\0' while calculating the length of a string.

---

## strcat

**strcat(s1, s2) concatenates(joins) the second string s2 to the first string s1.**

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s2[ ]= "World";
    char s1[20]= "Hello";
    strcat(s1, s2);
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
```

### Output

```
Source string = World
Target string = HelloWorld
```

---

### **strncat**

**strncat(s1, s2, n) concatenates(joins) the first ‘n’ characters of the second string s2 to the first string s1.**

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s2[ ]= "World";
    char s1[20]= "Hello";
    strncat(s1, s2, 2);
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
Source string = World
Target string = HelloWo
```

---

### **strcpy**

**strcpy(s1, s2) copies the second string s2 to the first string s1.**

```
#include <string.h>
#include <stdio.h>
int main()
{
    char s2[ ]= "Hello";
    char s1[10];
    strcpy(s1, s2);
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
Output
```

```
Source string = Hello
Target string = Hello
```

---

### **strncpy**

**strncpy(s1, s2, n) copies the first ‘n’ characters of the second string s2 to the first string s1.**

```
#include <string.h>
#include <stdio.h>
int main()
{
    char s2[ ]= "Hello";
    char s1[10];
    strncpy(s1, s2, 2);
    s1[2] = '\0'; /* null character manually added */
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
```

Output

```
Source string = Hello
Target string = He
```

Please note that we have added the null character ('\0') manually.

---

### strcmp

**strcmp(s1, s2)** compares two strings and finds out whether they are same or different. It compares the two strings character by character till there is a mismatch. If the two strings are identical, it returns a 0. If not, then it returns the difference between the ASCII values of the first non-matching pair of characters.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "jkl";
    char s2[ ]= "jkq";
    int i, j;
    i = strcmp(s1, "Hello");
    printf("%d", i);
    return 0;
}
```

Output:

-1

Here we have two strings str1 = "jkl" and str2 = "jkq". Comparison starts off by comparing the first character from str1 and str2 i.e 'j' from "jkl" and 'j' from "jkm", as they are equal, the next two characters are compared i.e 'k' from "jkl" and 'k' from "jkm", as they are also equal, again the next two characters are compared i.e 'l' from "jkl" and 'q' from "jkm", as ASCII value of 'q' (113) is greater than that of 'l' (108), Therefore str2 is greater than str1 and strcmp() will return 5 ( i.e 113-108 = 5 ) .

It is important to note that not all systems return difference of the ASCII value of characters, On some systems if str1 is greater than str2 then 1 is returned. On the other hand, if str1 is smaller than str2 then -1 is returned.

---

### strncmp

**strncmp(s1, s2, n)** compares the first 'n' characters of s1 and s2.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hello";
    char s2[ ]= "World";
    int i, j;
    i = strncmp(s1, "He Game", 2);
    printf("%d\n", i);
    return 0;
}
```

Output

0

---



## strchr

**strchr(s1, c)** returns a pointer to the first occurrence of the character **c** in the string **s1** and returns **NULL** if the character **c** is not found in the string **s1**.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hello World";
    char c = 'W';
    char *p;
    p = strchr(s1, c);
    printf("%s\n", p);
    return 0;
}
```

Output

World

---

## strrchr

**strrchr(s1, c)** returns a pointer to the last occurrence of the character **c** in the string **s1** and returns **NULL** if the character **c** is not found in the string **s1**.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hello";
    char c = 'o';
    char *p;
    p = strrchr(s1, c);
    printf("%s\n", p);
    return 0;
}
```

Output

o

---

## strstr

**strstr(s1, s2)** finds the first occurrence of the string **s2** in the string **s1**.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "HelloWorld";
    char s2[ ] = "World";
    char *p;
    p = strstr(s1, s2);
    printf("%s\n", p);
    return 0;
}
```

```
}  
Output
```

```
World
```

---

### **strcspn**

**strcspn(s1, s2)** returns the length of the initial part of the string s1 not containing any of the characters of the string s2.

```
#include <stdio.h>  
#include <string.h>  
int main()  
{  
    char s1[ ] = "HelloWorld";  
    char s2[ ] = "lo";  
    int i = strcspn(s1, s2);  
    printf("The first character is matched at %d\n", i+1);  
    return 0;  
}  
Output
```

```
The first character is matched at 3
```

---

### **strspn**

**strspn(s1, s2)** returns the length of the initial part of the string s1 only containing the characters of the string s2.

```
#include <stdio.h>  
#include <string.h>  
int main()  
{  
    char s1[ ] = "123abc";  
    char s2[ ] = "123456790";  
    int i = strspn(s1, s2);  
    printf("%d\n", i);  
    return 0;  
}  
Output
```

```
3
```

---

### **strpbrk**

**strpbrk(s1, s2)** returns the first occurrence of any of the characters of the string s2 in the string s1.

```
#include <stdio.h>  
#include <string.h>  
int main()  
{  
    char s1[ ] = "Hey 123 World";  
    char s2[ ] = "123456790";  
    char *p;  
    p = strpbrk(s1, s2);  
    printf("%s\n", p);  
}
```

```
    return 0;
}
Output
```

123 World

---

## **strtok**

**strtok(s1, s2) finds s2 in s1 and returns a pointer to it and returns NULL if not found.**

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hey,123,World";
    char *p;
    p = strtok(s1, ",");

    while(p != NULL)
    {
        printf("%s\n", p);
        p = strtok(NULL, ",");
    }
    return 0;
}
```

Output

Hey  
123  
World

---

## **Example Programs on Strings**

### **1. To reverse a string and check if a string is palindrome or not**

```
#include <stdio.h>
#include <string.h>
int main()
{
    int n,i,j;
    char s[30],t[30];
    printf("\n Enter the string:");
    scanf("%[^\\n]",s);
    n=strlen(s)-1;
    for(i=0,j=n;j>=0;i++,j--)
        t[i]=s[j];
    t[i]='\\0';
    printf("\nReverse of string is %s\n",t);
    if(strcmp(s,t)==0)
        printf("String is Palindrome");
    else
        printf("String is not Palindrome");
    return 0;
}
```

Output 1:

Enter the string:hello

```
Reverse of string is olleh
String is not Palindrome
```

Output 2:

```
Enter the string:MADAM

Reverse of string is MADAM
String is Palindrome
```

## Multidimensional Arrays

An array of two dimensions can be declared as follows:

```
data_type array_name[size1][size2];
```

Here, data\_type is the name of some type of data, such as int. Also, size1 and size2 are the sizes of the array's first and second dimensions, respectively.

In this sense, every element in the array a is identified by an element name in the form a[i][j], where 'a' is the name of the array, and 'i' and 'j' are the indexes that uniquely identify, or show, each element in 'a'.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
Row 1	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
Row 2	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

## Initializing a multidimensional array

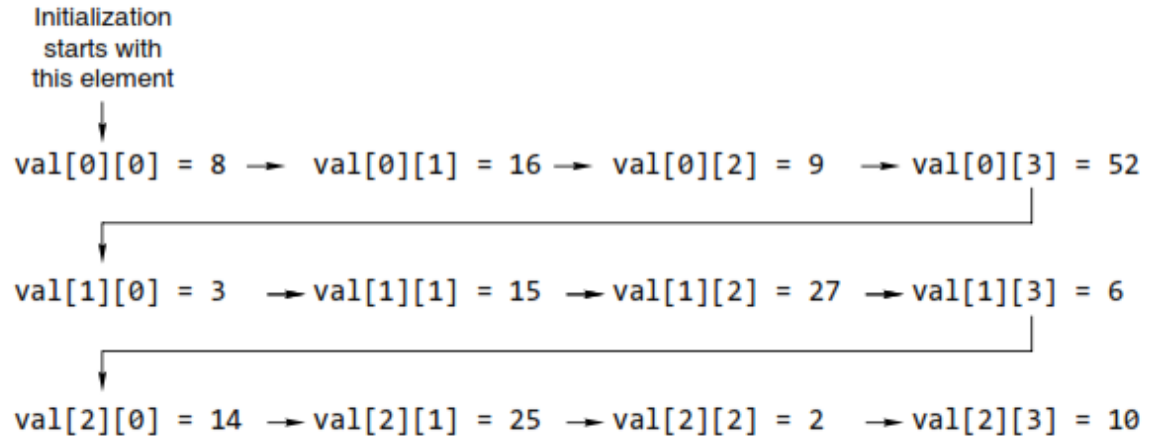
Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int val[3][4] = {
    {8, 16, 9, 52} , /* initializers for row indexed by 0 */
    {3, 15, 27, 9} , /* initializers for row indexed by 1 */
    {14, 25, 2, 10} /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example –

```
int val[3][4] = {8, 16, 9, 52, 3, 15, 27, 6, 14, 25, 2, 10};
```

The values are stored as shown in the figure below



### Unsize Array initializations

If unsize arrays are declared, the C compiler automatically creates an array big enough to hold all the initializers. This is called an unsize array. The following are examples of declarations with initialization.

(a) `char e1[] = "read error\n";`

(b) `char e2[] = "write error\n";`

(c) `int sgrs[][2] =`

```

{
    1,1,
    2,4,
    3,9,
    4,16,
};
  
```

### Accessing Multi-Dimensional Arrays

```

#include <stdio.h>

int main()
{
    int m,n, i, j;
    int a[10][10],b[10][10],c[10][10];
    printf("Enter the size of the matrix \n");
    scanf("%d%d",&m, &n);
    printf("enter the values into the first matrixx \n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    printf("The matrix entered is \n");

    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d  ", a[i][j]);
        }
    }
  
```

```

        printf("\n");
    }
}

```

The below statements used to get data into the matrix using scanf function. The operation is as follows.

The first for loop

```

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        scanf("%d", &a[i][j]);
    }
}

```

1. The outer for loop is at i=0 and inner for loop has j=0; the element at the location [0][0] is filled, next the inner for index j gets incremented to fill locations like [0][1], [0][2] and so on till [0][n-1].
2. Once the (n-1)th location is filled, the outer for loop gets incremented making i=1 and the process repeats filling up locations [1][0], [1][1] and so on till [1][n-1].
3. This process repeats till the last location [m-1][n-1] gets filled.

Similarly two for loops are used for reading the data of matrix as shown below. The way the loops work to access the elements remains same as explained for scanf.

```

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d  ", a[i][j]);
    }
    printf("\n");
}

```

## Example Programs of Matrix Operations:

### Matrix Addition:

```

#include <stdio.h>

int main()
{
    int m,n, i, j;
    int a[10][10],b[10][10],c[10][10];
    printf("Enter the size of the matrix \n");
    scanf("%d%d",&m, &n);
    printf("enter the values into the first matrixx \n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
}

```

```

printf("The matrix entered is \n");

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d  ", a[i][j]);
    }
    printf("\n");
}

printf("enter the values into the second matrixx \n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        scanf("%d", &b[i][j]);
    }
}
printf("The matrix entered is \n");

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d  ", b[i][j]);
    }
    printf("\n");
}

//matrix addition

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        c[i][j] = a[i][j]+b[i][j]; //for matrix subtraction replace + by -
    }
}

printf("The resultant matrix after addition is \n");

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d  ", c[i][j]);
    }
    printf("\n");
}
}

```

## Transpose of a Matrix:

```

#include <stdio.h>

int main(int argc, char **argv)
{
    int m,n, i, j;

```

```

int a[10][10],b[10][10];
printf("Enter the size of the matrix \n");
scanf("%d%d",&m, &n);
printf("enter the values into the first matrixx \n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        scanf("%d", &a[i][j]);
    }
}
printf("The matrix entered is \n");

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d  ", a[i][j]);
    }
    printf("\n");
}

printf("Transpose of the martix is \n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        b[j][i]=a[i][j];
    }
}

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d  ", b[i][j]);
    }
    printf("\n");
}

}

```

### **Matrix Multiplication:**

The below steps show the process of multiplication and how a C program can be written to perform matrix multiplication.

A 3x3 matrix multiplication happens as shown below. If matrix 1 has m rows and n columns, and matrix 2 has p rows and q columns, it is necessary that n should be equal to be for matrix multiplication to be performed. The resultant matrix would be of size mxq.



$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} j & k & l \\ m & n & o \\ p & q & r \end{pmatrix} = \begin{pmatrix} (aj+bm+cp) & (ak+bn+cq) & (al+bo+cr) \\ (dj+em+fp) & (dk+en+fq) & (dl+eo+fr) \\ (gj+hm+ip) & (gk+hn+iq) & (gl+ho+ir) \end{pmatrix}$$

```
#include <stdio.h>

int main()
{
    int m, n, p, q, c, d, k, sum = 0;
    int first[10][10], second[10][10], multiply[10][10];

    printf("Enter number of rows and columns of first matrix\n");
    scanf("%d%d", &m, &n);
    printf("Enter elements of first matrix\n");

    for (c = 0; c < m; c++)
        for (d = 0; d < n; d++)
            scanf("%d", &first[c][d]);

    printf("Enter number of rows and columns of second matrix\n");
    scanf("%d%d", &p, &q);

    if (n != p)
        printf("The matrices can't be multiplied with each other.\n");
    else
    {
        printf("Enter elements of second matrix\n");

        for (c = 0; c < p; c++)
            for (d = 0; d < q; d++)
                scanf("%d", &second[c][d]);

        for (c = 0; c < m; c++)
        {
            for (d = 0; d < q; d++)
            {
                for (k = 0; k < p; k++)
                {
                    sum = sum + first[c][k]*second[k][d];
                }

                multiply[c][d] = sum;
                sum = 0;
            }
        }

        printf("Product of the matrices:\n");

        for (c = 0; c < m; c++) {
            for (d = 0; d < q; d++)
                printf("%d\t", multiply[c][d]);

            printf("\n");
        }
    }
}
```

```
    }  
}  
  
return 0;  
}
```

Output:

Enter number of rows and columns of first matrix

2 3

Enter elements of first matrix

1 2 3

4 5 6

Enter number of rows and columns of second matrix

3 2

Enter elements of second matrix

3 2 1

6 5 4

Product of the matrices:

20        26

47        62