Array

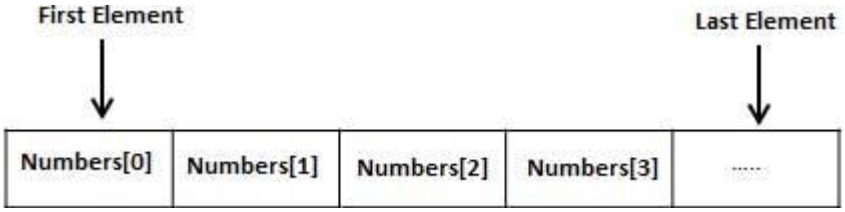
An Array is collection of similar type of data items stored at contiguous memory locations. It is collection of elements of the same data type stored in contiguous memory locations. Arrays allow to store and manipulate a collection of values under a single name. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

An array is a collective name given to a group of 'similar quantities. Array is a collection of similar elements. These similar elements could be all int, or all float, or all char, etc. Usually, the array of characters is called a 'string', whereas an array of int or float is called simply an array.

All elements of any given array must be of the same type, i.e., we cannot have an array of 10 numbers, of which 5 are int and 5 are float.

Properties of Array

- An array is a collection of items stored at contiguous memory locations.
- It is used to store multiple items of the same type together.
- All arrays consist of contiguous memory locations.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.
- Each element in the array is identified by an index, which is a numerical value that represents the position of the element in the array. The first element in the array has an index of 0, the second element has an index of 1, and so on.
- Each element is identified by an index starting with "0". The lowest address corresponds to the first element and the highest address to the last element.



Why to Use Arrays in C?

Arrays are used to store and manipulate the similar type of data. Suppose we want to store the marks of 10 students and find the average. We declare 10 different variables to store 10 different values as follows −

```
int a = 50, b = 55, c = 67, . . . ;
float avg = (float)(a + b + c +. . . ) / 10;
```

These variables will be scattered in the memory with no relation between them. Importantly, if we want to extend the problem of finding the average of 100 (or more) students, then it becomes impractical to declare so many individual variables.

Arrays offer a compact and memory-efficient solution. Since the elements in an array are stored in adjacent locations, we can easily access any element in relation to the current element. As each element has an index, it can be directly manipulated.

Advantage of C Array

1) Code Optimization: Less code to the access the data.
2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.
3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.
4) Random Access: We can access any element randomly using the array.

Array Declaration

In C, it is must to declare the array like any other variable before using it. To declare an array, it is required to specify the data type of the elements, the name of the array, and the size of the array (in brackets). The size must be a positive integer.

Syntax	**data_type Array_name [Array_size];**	Where, A**rray_size** must be an integer constant greater than zero and **data_type** can be any valid C data type.

Example:	**int marks [4];**	In given example, int is the data_type, marks are the array_name, and 4 is the array_size.

When an array is declared in C, the compiler allocates the memory block of the specified size to the array name.

Initialization of C Array

The array is initialized by using the index of each element. We can initialize each element of the array by using the index.

Example.

```
int marks [5]; //declaration of array

marks[0]=80; //initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```

C Array Examples

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

main( )
{
int i;
int marks[5]; //declaration of array
marks[0]=80; //initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
for(i=0; i <5; i++)
  {
     printf("%d \n", marks[i]);
   }
getch();
}
```

Output

```
80
60
70
85
75
```

C Array Declaration with Initialization

At the time of declaring an array, you can initialize it by providing the set of comma-separated values enclosed within the curly braces {}.

Syntax	data_type array_name [size] = {value1, value2, value3, ...};

Example to Initialize an Array

int marks[5]={20,30,40,50,60};
In such case, there is no requirement to define the size. So it may also be written as the following code.

Examples

int num[6] = { 2, 4, 12, 5, 45, 5 } ;

int n[ ] = { 2, 4, 12, 5, 45, 5 } ;

float press[ ] = { 12.3, 34.2, -23.4, -11.3 } ;

EXAMPLE 1	#include<stdio.h> #include<conio.h> main( ) { int i=0; int marks[5] = {20,30,40,50,60}; **//declaration and initialization of array** for(i=0;i<5;i++) { printf("%d \n", marks[i]); } getch( ); }

Output

```
20
30
40
50
60
```

EXAMPLE 2	#include<stdio.h> #include<conio.h> int main( ) { int avg, sum = 0; int i ; int marks[30] ; **/* array declaration */** for (i = 0 ; i <= 29 ; i++) { printf ("Enter marks ") ; scanf ("%d", &marks[i]) ; **/* store data in array */** } for (i = 0 ; i <= 29 ; i++) { sum = sum + marks[i] ; **/* read data from an array*/** }

	avg = sum / 30 ; printf ("Average marks = %d\n", avg) ; getch( ) ; }

Imp points for Array

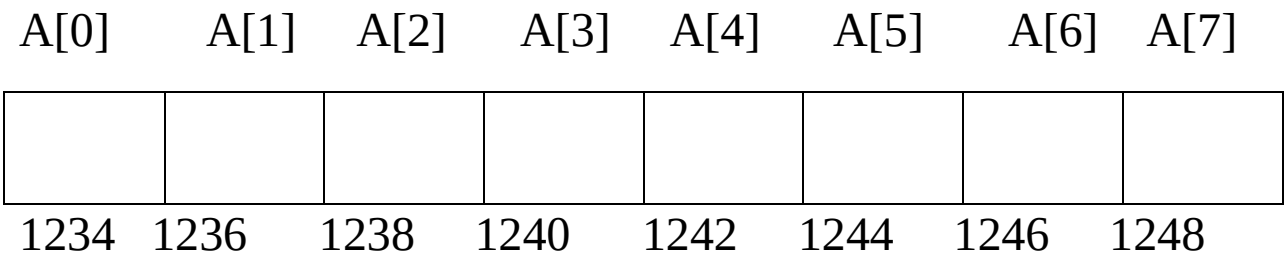
(a) An array is a collection of similar elements.
(b) The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
(c) An array is also known as a subscripted variable.
(d) Before using an array, its type and dimension must be declared.
(e) However big an array, its elements are always stored in contiguous memory locations.

Array Elements in Memory

Consider the following array declaration:

int A[8] ;

After declaration, 16 bytes or 32 bytes get immediately reserved in memory, 2 bytes or 4 bytes each for the 8 integers (depending on compiler). And since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be auto. If the storage class is declared to be static, then all the array elements would have a default initial value as zero. Whatever be the initial values, all the array elements would always be present in contiguous memory locations.



Passing Array Elements to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In the call by value, we pass values of array elements to the function, whereas in the call by reference, we pass addresses of array elements to the function.

/* Demonstration of call by value */

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

void display ( int ) ;

int main( )
 {
   int i ;
   int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

   for ( i = 0 ; i <= 6 ; i++ )
  {
   display ( marks[ i ] ) ;
    }
   getch( );
   }

void display ( int m )
 {
   printf ( "%d ", m ) ;
 }
```

OUTPUT : 55 65 75 56 78 78 90

/* Demonstration of call by reference */

```
 # include<stdio.h>
#include<conio.h>
 void disp ( int * ) ;
int main( )
 {
  int i ;
  int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
  for ( i = 0 ; i <= 6 ; i++ )
  {
   disp ( &marks[ i ] ) ;
  }
   getch( );
  }
void disp ( int *n )
 {
  printf ( "%d ", *n ) ;
 }
```

OUTPUT : 55 65 75 56 78 78 90

Pointers and Arrays

```
# include <stdio.h>
int main( )
{
    int  i = 3, *x ;
    float  j = 1.5, *y ;
    char  k = 'c', *z ;
    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of j = %f\n", j ) ;
    printf ( "Value of k = %c\n", k ) ;
    x = &i ;
    y = &j ;
    z = &k ;
    printf ( "Original address in x = %u\n", x ) ;
    printf ( "Original address in y = %u\n", y ) ;
    printf ( "Original address in z = %u\n", z ) ;
    x++ ;
    y++ ;
    z++ ;
    printf ( "New address in x = %u\n", x ) ;
    printf ( "New address in y = %u\n", y ) ;
    printf ( "New address in z = %u\n", z ) ;
    return 0 ;
}
```

Here is the output of the program.

```
Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65528
New address in y = 65524
New address in z = 65520
```

Multidimensional Arrays

A multi-dimensional array can be defined as an array that has more than one dimension. Having more than one dimension means that it can grow in multiple directions. Some popular multidimensional arrays are 2D arrays and 3D arrays.

Syntax	type arr_name[size1][size2]….[sizeN];

Where type: Type of data to be stored in the array, arr_name: Name assigned to the array and size1, size2,…, sizeN: Size of each dimension.

2D Arrays

The two-dimensional array is also called a matrix. The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

A 2D array, also known as a two-dimensional array or matrix, is a data structure that consists of a collection of elements arranged in rows and columns. It can be visualized as a table with rows and columns, where each element can be accessed using two indices: one for the row and one for the column.

Declaration of Two-Dimensional Array in C

syntax	**data_type array_name[number_of_Rows][number_of_Columns];**

Here, dataType can be any valid C data type, array_name is your chosen identifier for the array, and number_of_rows and number_of_columns define the array's size.

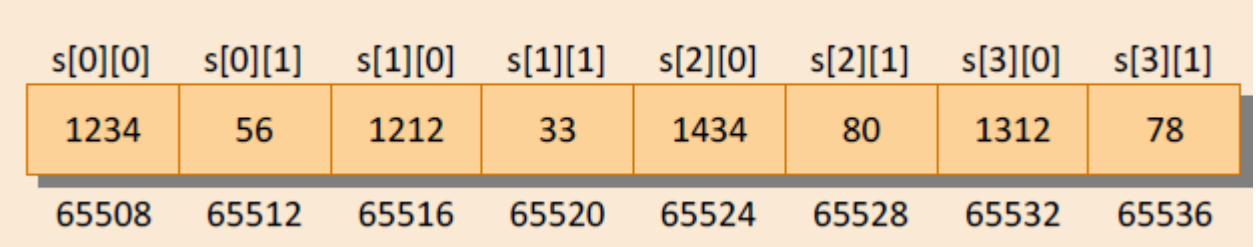
EXAMPLE	int stud [4][2]; char ABC [2][3]; float A [3][4];

Initializing a Two-Dimensional Array

Method 1	**Method 2**
int stud[4][2] = { { 1234, 56 }, { 1212, 33 }, { 1434, 80 }, { 1312, 78 } } ;	int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
While initializing a 2-D array, it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.	int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ; int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;

Memory Map of a Two-Dimensional Array

int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;



Accessing Elements of 2D Array

Accessing elements within a two-dimensional array in C will be accessed by using its row and column.

To reach the desired data point, below format is format:

arrayName[rowIndex][columnIndex];

For example, consider an array Matrix[5][10] and you want to access the data at the 6th row and 3rd column. Keeping in mind C's zero-based indexing, which starts counting rows and columns from 0, so, we have to access the element at Matrix[5][2].

Write a program in c to create a matrix and print it.

```
#include<stdio.h>
#include<conio.h>
main()
{
int A[10][10];
int r,c,i,j;
clrscr();
printf("enter the row and col\n");
scanf("%d%d", &r, &c);

printf("Enter matrix elements\n");

for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d", &A[i][j]);
}
}



printf("Entered matrix is :\n");

for(i=0;i<r;i++)
```

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

Write a program to multiply two given matrices

```
#include<stdio.h>
#include<conio.h>
int main()
{
int r1,r2,c1,c2;
int m1[10][10],m2[10][10], mul[10][10];
int i, j, k;

printf("Enter number of rows for First Matrix:\n");
scanf("%d",&r1);
printf("Enter number of columns for First Matrix:\n");
scanf("%d",&c1);
printf("Enter number of rows for Second Matrix:\n");
scanf("%d",&r2);
printf("Enter number of columns for Second Matrix:\n");
scanf("%d",&c2);

if(c1!=r2)
   {
        printf("Matrices Can't be multiplied together");
   }

   else
   {
        printf("Enter first matrix elements \n");
        for(i=0;i<r1;i++)
        {
           for(j=0;j<c1;j++)
           {
                scanf("%d",&m1[i][j]);
           }
        }

  printf("Enter Second matrix elements\n");
        for(i=0;i<r2;i++)
        {
           for(j=0;j<c2;j++)
           {
                scanf("%d",&m2[i][j]);
           }
        }
```

```
        for(i=0;i<r1;i++)
        {
           for(j=0;j<c2;j++)
           {
                mul[i][j]=0;

                // Multiplying i'th row with j'th column
                for(k=0;k<c1;k++)
                {
                   mul[i][j] = mul[i][j] + m1[i][k]*m2[k][j];
                }
           }
        }
        printf("Multiplied matrix\n");
        for(i=0;i<r1;i++)
        {
           for(j=0;j<c2;j++)
           {
                printf("%d\t",mul[i][j]);
           }
           printf("\n");
        }
   }
   getch();
}
```

Write a program to find the transpose of any given matrix.

```
#include<stdio.h>
#include<conio.h>
main()
{
int A[10][10], T[10][10];
int r,c,i,j;
clrscr();
printf("enter the row and col\n");
scanf("%d%d", &r, &c);

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

printf("Entered matrix is :\n");
for(i=0;i<r;i++)
{
```

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

//finding the transpose matrix elements

for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
T[j][i]= A[i][j];
}
}

printf("Transpose of entered matrix is :\n");

for(i=0;i<c;i++)
{
for(j=0;j<r;j++)
{
printf("%d\t", T[i][j]);
}
printf("\n");
}
getch();
}
```

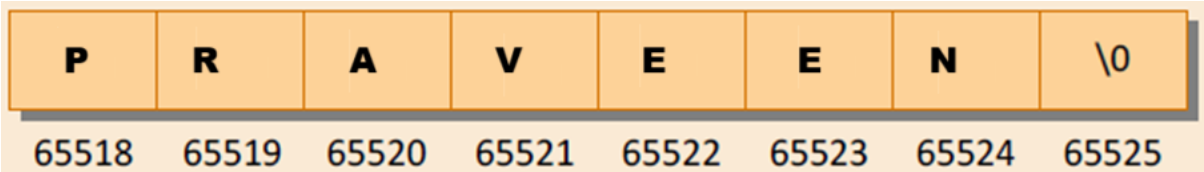
Character Arrays and Strings

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is stored as an array of characters. The difference between a character array and a C string is that the string in C is terminated with a unique character '\0'.

A string constant is a one-dimensional array of characters terminated by a null ('\0'). in C, strings are null-terminated, meaning they have a '\0' character at the end.

For example,

char name [ ] = { 'P', 'R', 'A', 'V', 'E', 'E', 'N', '\0' } ;



Each character in the array occupies 1 byte of memory and the last character is always'\0'. '\0' is called null character. '\0' and '0' are not same. ASCII value of'\0' is 0, whereas ASCII value of '0' is 48.

char name [ ] = "PRAVEEN" ;

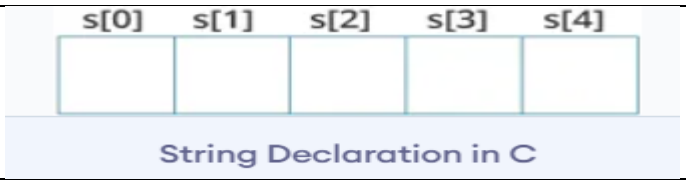
In this declaration'\0' is not necessary. C inserts the null character automatically.

C String Declaration

Syntax:	char string_name[size];

In the above syntax **string_name** is any name given to the string variable and size is used to define the length of the string, i.e the number of characters strings will store.

Example

char s[5];	

C String Initialization

We can initialize a C string in 4 different ways which are as follows:

1. Assigning a String Literal without Size

String literals can be assigned without size. Here, the name of the string str acts as a pointer because it is an array.

char str[] = "PRAVEEN";

2. Assigning a String Literal with a Predefined Size

String literals can be assigned with a predefined size. But we should always account for one extra space which will be assigned to the null character. If we want to store a string of size n then we should always declare a string with a size equal to or greater than n+1.

char str[50] = "PRAVEEN";

3. Assigning Character by Character with Size

We can also assign a string character by character. But we must set the end character as '\0' which is a null character.

char str[14] = { 'P','R','A','V','E','E','N','\0'};

4. Assigning Character by Character without Size

We can assign character by character without size with the NULL character at the end. The size of the string is determined by the compiler automatically.

char str[] = { 'P','R','A','V','E','E','N','\0'};

When a Sequence of characters enclosed in the double quotation marks is encountered by the compiler, a null character '\0' is appended at the end of the string by default.

```
#include<stdio.h>
#include<conio.h>
main()
{
char name []="Praveen";
int i=0;
for(i=0;i<7;i++)
{
printf("%c", name[i]);
}
getch();
}
```

```
#include<stdio.h>
#include<conio.h>
main()
{
char name []="Praveen";
int i=0;
while (i<7)
{
printf("%c", name[i]);
i++;
}
getch();
}
```

```
#include<stdio.h>
#include<conio.h>
main()
{
char name [] ="Praveen Tripathi";
int i=0;
```

```
#include<stdio.h>
#include<conio.h>
main()
{
char name [] ="Praveen Tripathi";
int i=0;
```

```
while (name[i]  != '\0')
{
printf("%c", name[i]);
i++;
}
getch();
}
```

```
char *ptr; // store base address of string
ptr= name;

while (*ptr != '\0')
{
printf("%c", *ptr);
ptr++;
}
getch();
}
```

```
#include<stdio.h>
#include<conio.h>
main()
{
char name []="Praveen Tripathi";

printf("%s", name);

getch();

}
```

```
#include<stdio.h>
#include<conio.h>
main()
{
char name [10];

printf("enter your name");
scanf("%s", name);

printf("Hello %s", name);

getch();
}
```

Standard String Library Function

1. strlen() Function

The strlen() function in C calculates the length of a given string. It returns the number of characters in the string, excluding the null character (\0). strlen() function in C returns the length of the string.

Syntax : **int strlen (string name);**

Example :
```
char a[30] = “Praveen ”;
int l;
l = strlen (a);
```

Program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char name1[]="Praveen";
int len1, len2;
```

```
len1=strlen(name1);
len2=strlen("Praveen Tripathi");

printf("String : %s and Its lenght : %d", name1, len1);
printf("String :%s and its lenght : %d", "Praveen Tripathi", len2);
getch();
}
```

2. strcpy() Function

The strcpy() is a standard library function in C and is used to copy one string to another. In C, it is present in **<string.h>** header file. The strcpy() function in C copies a source string to a destination string.

Syntax	**strcpy (Destination string, Source String);**

Program 1	**Program 2**
#include<stdio.h> #include<conio.h> #include<string.h> main() { char name1[]="Praveen"; char name2[1]; clrscr(); **strcpy(name2, name1);** printf("name1 String : %s", name1); printf("name2 String : %s", name2); getch(); }	#include<stdio.h> #include<conio.h> #include <string.h> main () { char a[50], b[50]; printf ("enter a source string"); scanf("%s", a); printf("enter destination string"); scanf("%s",b); **strcpy (b,a);** printf ("copied string = %s",b); getch (); }

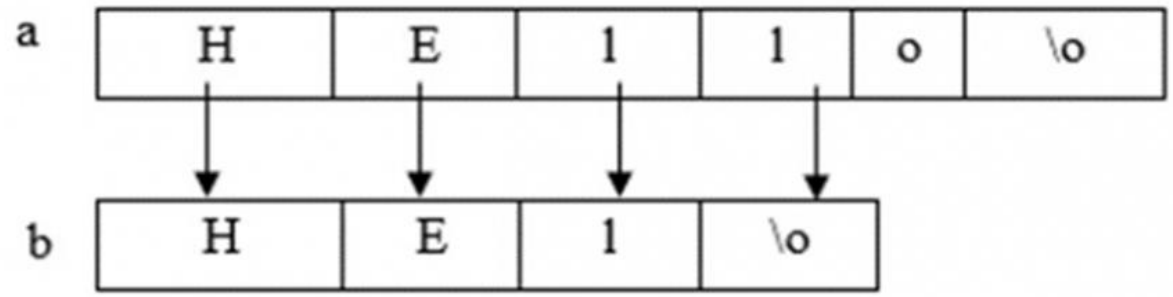
3. The strncpy () function

It copy's 'n' characters of source string into destination string. The length of the destination string must >= that of the source string.
It's not required for the length of the destination string to be greater than or equal to the source string, but the destination string must have enough space to hold at least n characters.

Also, remember to always include a null terminator (\0) at the end of the destination string after using strncpy(), especially if the source string is shorter than n.

Syntax: **strncpy (Destination string, Source String, n);**

Example:



#include<stdio.h> #include<conio.h> #include<string.h> main () { char a[50], b[50]; printf ("enter a string"); gets (a); strncpy (b,a,3);// copy first 3 char from a string b[3] = '\0'; printf ("copied string = %s",b); getch (); }	Output Enter a string : Praveen Copied string = Pra

4. The strcat() Function

The strcat() function in C is used for string concatenation. It will append a copy of the source string to the end of the destination string.

The strcat() function does not perform any bounds checking, so it can lead to buffer overflow errors if the destination string is not large enough to hold the concatenated strings. **The destination string must have enough space to hold the concatenated strings.** The strcat() function appends the entire source string to the end of the destination string.

Syntax : strcat (Destination String, Source string);

#include<stdio.h> #include<conio.h> #include<string.h> main() { char name1[15]="Praveen"; char name2[15]="Tripathi"; clrscr(); **strcat(name2,name1);** printf("name1 String :%s",name1); printf("name2 String :%s",name2); getch();	#include<stdio.h> #include<conio.h> #include <string.h> main() { char a[50] = "Hello"; char b[20] = "Good Morning"; clrscr (); **strcat (a,b);** printf("concatenated string = %s", a); getch (); }

} **OUTPUT:** Name1 String: Praveen Name2 String : TripathiPraveen	**OUTPUT:** Concatenated string = Hello Good Morning

5. The strncat() Function

The strncat() function in C is used to concatenate two strings by appending a specified number of characters from the source string to the end of the destination string.

This is used for combining or concatenating n characters of one string into another. The length of the destination string must be greater than the source string. The resultant concatenated string will be in the destination string.

The strncat() function does not perform any bounds checking on the destination string, so it can lead to buffer overflow errors if the destination string is not large enough to hold the concatenated strings.

Syntax :	**strncat (Destination String, Source string, n);**

#include <stdio.h> #include<conio.h> #include <string.h> main () { char a [30] = "Praveen"; char b [20] = "Good Morning"; clrscr (); **strncat (a,b,4);** printf("concatenated string = %s", a); getch (); }	OUTPUT : PraveenGood

6. The strcmp() Function

- The strcmp() is a built-in library function in C. This function takes two strings as arguments and compares these two strings lexicographically.
- The strcmp() function compares strings character by character, using the ASCII values of the characters.
- The comparison is case-sensitive, meaning that uppercase letters are considered less than lowercase letters.
- The strcmp() function does not perform any bounds checking on the input strings, so it can lead to undefined behavior if the strings are not null-terminated.
- **It returns the ASCII difference of the first two non – matching characters in both the strings.**

Syntax :	int strcmp (string1, string2);

//If the difference is equal to zero, then string1 = string2

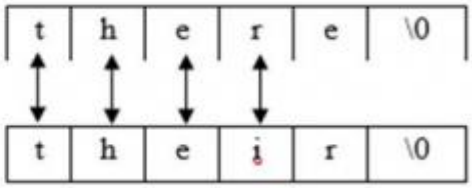
//If the difference is positive, then string1 > string2

//If the difference is negative, then string1 < string2

eg:

1) char a[10]= "there"
   char b[10] = "their"
   strcmp (a,b);

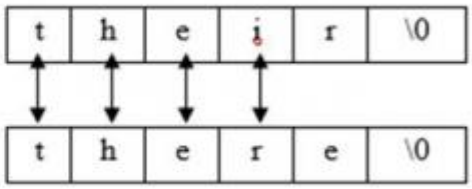
   Output: string1 >string2



'r' > 'i'

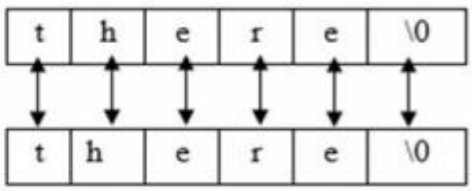
2) char a[10]= "their"
   char b[10] = "there"
   strcmp (a,b);

   Output: string1 <string2



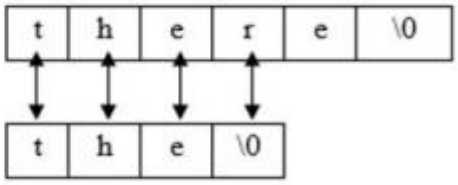
'i'< 'r'

3) char a[10]= "there"
   char b[10] = "there"
   strcmp (a,b);
   Output: string1 =string2



4) char a[10]= "there"
char b[10] = "the"
strcmp (a,b)

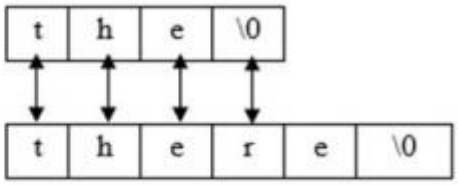
Output: string1 >string2



'r' > '\0'

5) char a[10]= "the"
char b[10] = "there"
strcmp (a,b);

Output: string1 <string2



'\0' < 'r'

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main ()
{
  char a[50], b [50];
  int d;
  printf ("Enter 2 strings:");
  scanf ("%s %s", a,b);

 d = strcmp(a,b);

if (d==0)
{
printf("%s is (alphabetically) equal to %s", a,b);
  }
else if (d>0)
{
    printf("%s is (alphabetically) greater than
%s",a,b);
  }
else if (d<0)
{
 printf("%s is (alphabetically) less than %s", a,b);
  }
getch();
}
```

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

main()
{

   char str1[] = "Praveen";

   char str2[] = "Prbveen";

   char str3[] = "Prvin";


   int result1 = strcmp(str1, str2);

   int result2 = strcmp(str2, str3);

   int result3 = strcmp(str1, str1);


   printf("Comparison of str1 and str2:
%d\n", result1);

   printf("Comparison of str2 and str3:
%d\n", result2);

   printf("Comparison of str1 and str1:
%d\n", result3);
   getch();
}
```

7. The strlwr() function

The strlwr() function is used in C to convert a string to lowercase. It takes a string as input and modifies it in place by changing all uppercase letters to their lowercase equivalents.

Syntax	**strlwr(string_name);**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char d[]="PRAVEEN";
char s[10]="ABC";
clrscr();
strlwr(d);
strlwr(s);
printf("source string:%s\n", s);
printf("destination string:%s", d);
getch();
}
```

8. The strupr () function

The strupr() function is used in C to convert all characters of a given string to uppercase. It takes a string as an argument and modifies it in place, turning all lowercase letters into their uppercase equivalents.

Syntax	**strupr(string_name);**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char d[]="praveen";
char s[10]="abc";
clrscr();
strupr(d);
strupr(s);
printf("source string:%s\n", s);
printf("destination string:%s", d);
getch();
}
```

9. The strrev () function

The strrev() function is to reverse a string. It takes a single parameter, the string you want to reverse, and returns the reversed string. For example, strrev("Hello") would return "olleH".

Syntax	**strrev(string_name);**

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

```
#include<string.h>
main()
{
char d[]="PRAVEEN";
char s[10]="ABCefg";
clrscr();
strrev(d);
strrev(s);
printf("First reversed string:%s\n", s);
printf("Second reversed string:%s", d);
getch();
}
```

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings by ignoring the case
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

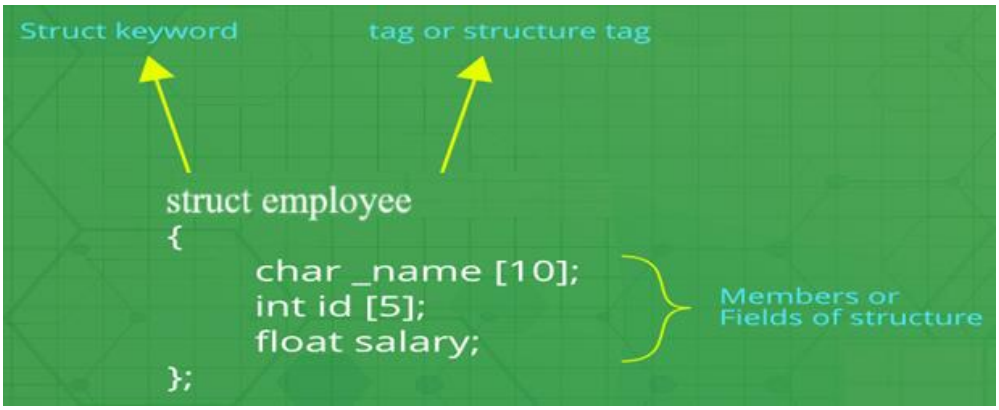
IMP POINTS about STRING

1. A string is nothing but an array of characters terminated by '\0'.
2. Being an array, all the characters of a string are stored in contiguous memory locations.
3. Though **scanf( )** can be used to receive multi-word strings, **gets( )** can do the same job in a cleaner way.
4. Both **printf( )** and **puts( )** can handle multi-word strings.
5. Strings can be operated upon using standard library functions like **strlen( )**, **strcpy( )**, **strcat( )** and **strcmp( )** which can manipulate strings.

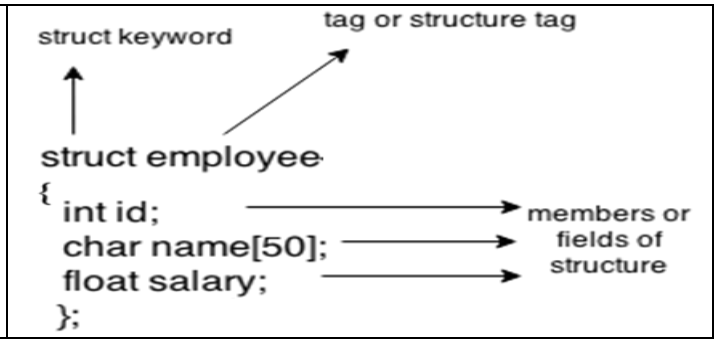
Structure in C

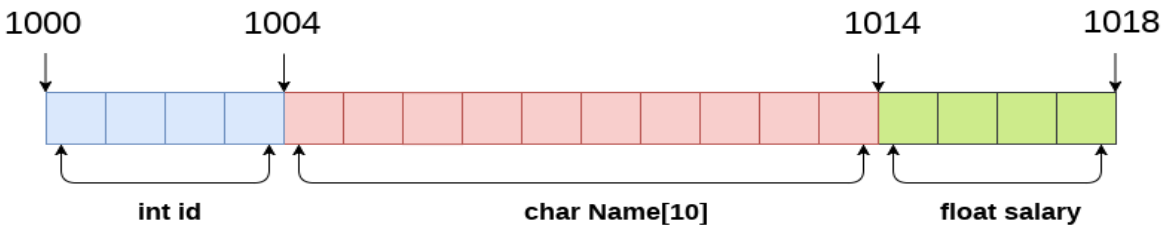
Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. A structure contains a number of data types grouped together. These data types may or may not be of the same type.

The structure can be used to group items of possibly different types into a single type. The struct keyword is used to define the structure in the C programming language. The items in the structure or members of structure can be of any valid data type. The values of a structure are stored in contiguous memory locations.




```
struct employee
{
int id;
char name [20];
float salary;
};
```

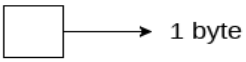





```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte



Declaring a Structure

Syntax:	**struct structure_name** **{** **structure element 1 ;** **structure element 2 ;** **structure element 3 ;** **.....** **.....** **} ;**

EXAMPLE

struct book { char name [10]; float price ; int pages ; } ;	This statement defines a new data type called **struct book**. Each variable of this data type will consist of a character variable called **name**, a float variable called **price** and an integer variable called **pages**.

Declaring structure variable

When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

To access and manipulate the members of the structure, it is needed to declare its variable first. To declare a structure variable, write the structure name along with the "**struct"** keyword followed by the name of the structure variable. This structure variable will be used to access and manipulate the structure members.

Example to declare structure variable

struct book { char name [10]; float price ; int pages ; }**b1, b2, b3 ;**	struct book { char name [10]; float price ; int pages ; } ; **struct book b1, b2, b3 ;**

struct book b1, b2, b3; This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, 16 bytes—ten for **name**, four for **price** and two for **pages**. These bytes are always in adjacent memory locations.

Initialization of structure variable

The initialization of a struct variable is done by placing the value of each element inside curly brackets.

Example 1	struct book { char name[10] ;

	float price ; int pages ; } ; struct book b1 = { "Computer", 130.00, 550 } ; struct book b2 = { "Physics", 150.80, 800 } ; struct book b3 = { “Chemistry”, 130.00, 550 };

Accessing the Structure Members

Accessing structure elements in C involves using either the **dot operator (.)** or the **arrow operator (->)** depending on whether you are working with a regular structure variable or a pointer to a structure.

Accessing Structure Elements Using the Dot Operator

	Syntax:	**structure_variable. member_name**

Example	**b1.pages** **b1.price**

Example	struct Student { char name[50]; int age; }; struct Student s1; **s1.age = 20; // Accessing and assigning a value**

Imp points about Structure

1. The closing brace (}) in structure type declaration must be followed by a semicolon(;)
2. Structure type declaration does not tell the compiler to reserve any space in memory. It only defines the ‘form’ of the structure.
3. Usually, structure type declaration appears at the top of the source code file, before any variables or functions are defined.
4. If a structure variable is initiated to a value { 0 }, then all its elements are set to value 0,

```
#include<stdio.h>
#include<conio.h>
int main( )
{
struct book
{
char name[5] ;
float price ;
int pages ;
};
struct book b1 = { "abc", 130.00, 550 } ;
struct book b2 = { "xyz", 100.50, 150 } ;
```

```
clrscr();

printf ( "Name of book  :%s \t Address of name = %u\n",  b1.name,  &b1.name ) ;
printf ( "Price of book :%f \t Address of price = %u\n", b1.price, &b1.price ) ;
printf ( "Pages of book :%d \t Address of pages = %u\n", b1.pages, &b1.pages ) ;
printf ( "Name of book  :%s \t Address of name = %u\n",  b2.name,  &b2.name ) ;
printf ( "Price of book :%f \t Address of price = %u\n", b2.price, &b2.price ) ;
printf ( "Pages of book :%d \t Address of pages = %u\n", b2.pages, &b2.pages ) ;

getch();
}
```

How Structure Elements are Stored?

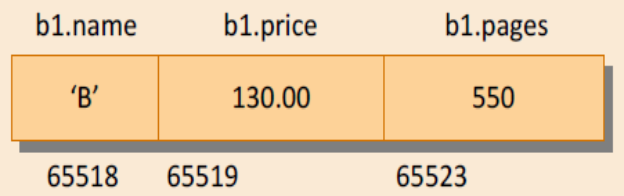
Structure elements in are typically stored in memory as a contiguous block of data. They are always stored in contiguous memory locations.

```
# include <stdio.h>
int main( )
{
struct book
{
char name ;
float price ;
int pages ;
} ;
struct book b1 = { 'B', 130.00, 550 } ;

printf ( "Address of name = %u\n", &b1.name ) ;
printf ( "Address of price = %u\n", &b1.price ) ;
printf ( "Address of pages = %u\n", &b1.pages )
;
return 0 ;
}
```

Output

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```



Array of Structures

An array of structures is a data structure in programming that consists of an array where each element is a structure. A structure is a composite data type that groups different data types together under a single name. By using an array of structures, you can store multiple records of the same type, with each record containing multiple fields. This is particularly useful when dealing with lists of items that share common characteristics.

```
# include <stdio.h>
#include<conio.h>
int main( )
{
struct book
{
```

```
char name ;
float price ;
int pages ;
} ;
struct book b1, b2, b3 ;
printf ( "Enter names, prices & no. of pages of 3 books\n" ) ;

scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;

printf ( "And this is what you entered\n" ) ;

printf ( "%c %f %d\n", b1.name, b1.price, b1.pages ) ;
printf ( "%c %f %d\n", b2.name, b2.price, b2.pages ) ;
printf ( "%c %f %d\n", b3.name, b3.price, b3.pages ) ;
getch();
}
```

Enumerated Data types

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constant. The enum data type in C is a way to define a set of named values. It allows to create a new data type that consists of a fixed number of distinct values.

The enumerated data type in C allows to define a set of named values. These values are constants that have underlying numeric types (typically int) but are given meaningful names to improve code readability and maintainability.

Syntax for declaring an enum is:	**Example**	Where
enum enum_name { value1, value2, value3, ... ….. …… };	enum week { mon, tue, wed, thu, fri, sat, sun };	Identifier values are unsigned integers and start from 0. mon refers 0, tue refers 1 and so on.

We can also assign specific integer values to the constants:

```
enum enum_name
{
   value1 = 1,
   value2 = 5,
   value3 = 10,
   ...
};
```

If we don't assign values, they will be assigned incrementally starting from 0.

Example

```
#include<stdio.h>
#include<conio.h>
enum week {Mon, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{
   enum week day;
   day = Wed;
   printf("%d",day);
   getch();
}
```

Output:

2

In the above example, “day” is as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.

Example

```
#include<stdio.h>
#include<conio.h>
enum year {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
int main()
{
  int i;
  for (i=Jan; i<=Dec; i++)
     printf("%d ", i);

  getch();
}
```

Output:

0 1 2 3 4 5 6 7 8 9 10 11

In this example, the for loop will run from i = 0 to i = 11, as initially the value of i is Jan which is 0 and the value of Dec is 11.

Union

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

In C programming, a union is a user-defined data type similar to a structure. However, while a struct allocates memory for all its members, a union allocates memory equal to the size of its largest member. This means that at any given time, a union can store only one of its members. All members of a union share the same memory space. The size of a union is equal to the size of its largest member. Only one member of a union can be used at a time.

C Union Declaration

At the time of declaration, only the template of the union is declared, i.e., we only declare the members' names and data types along with the name of the union. No memory is allocated to the union in the declaration. A union is declared using the union keyword followed by the union name and its members.

```
union name
{
type1 member1;
type2 member2;
.
.
};
```

Create a Union Variable

It is needed to define a variable of the union type to start using union members. There are two methods using which we can define a union variable:

Creating Union Variable with Declaration	**Creating Union Variable after Declaration**
union name { type member1; type member2; … } **var1, var2, …;**	**union name var1, var2, var3…;** where name is the name of an already declared union.

Access Union Members

Union members can be accessed by using the (.) dot operator just like structures.

Syntax:	var1.member1;

where ***var1*** is the **union variable** and ***member1*** is the **member of the union**.

Initialize Union

The initialization of a union is the initialization of its members by simply assigning the value to it.

Syntax	**var1.member1 = val;**

One important thing to note here is that **only one member can contain some value at a given instance of time.**

Structure	Union
The size of the structure is equal to or greater than the total size of all of its members.	The size of the union is the size of its largest member.
The structure can contain data in multiple members at the same time.	Only one member can contain data at the same time.
It is declared using the struct keyword.	It is declared using the union keyword.

Structure vs Union

Example 1: WAP to implement Union.

```
#include <stdio.h>
#include <string.h>
#include<conio.h>
union ABC
{
  int i;
  float f;
  char str [20];
};

int main ()
{
  union ABC data;

  data.i = 10;
  data.f = 220.5;
  strcpy(data.str, " C Programming ");

  printf("data.i: %d \n", data.i);
  printf("data.f: %f \n", data.f);
  printf("data.str: %s \n", data.str);
  getch();
}
```

Output

When the above code is compiled and executed, it produces the following result −

```
data.i: 1917853763

data.f: 4122360580327794860452759994368.000000

data.str: C Programming
```

Here, values of **i** and **f** (members of the union) show **garbage values** because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed.

Example 2: WAP to implement Union.

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

union ABC
{
  int i;
  float f;
```

Output

When the above code is compiled and executed, it produces the following result −

```
data.i: 10

data.f: 220.500000
```

char str[20]; }; main() { union ABC data; data.i = 10; printf("data.i: %d \n", data.i); data.f = 220.5; printf("data.f: %f \n", data.f); strcpy(data.str, "C Programming"); printf("data.str: %s \n", data.str); getch(); }	data.str: C Programming Here, the values of all the Union members are getting printed because one member is being used at a time.

Size of a Union

The size of a union is the size of its largest member. For example, if a union contains two members of **char** and **int** types. In that case, the size of the union will be the size of **int** because **int** is the largest member. The sizeof() operator can be used to get the size of a union.

Example 3: WAP to explain Size of a Union	
#include <stdio.h> **// Define a union** union Data { int a; float b; char c [20]; }; int main () { union Data data; **// Printing the size of each member of union** printf("Size of a: %lu bytes\n", sizeof(data.a)); printf("Size of b: %lu bytes\n", sizeof(data.b)); printf("Size of c: %lu bytes\n", sizeof(data.c)); **// Printing the size of the union** printf("Size of union: %lu bytes\n", sizeof(data)); return 0; }	**Output** When you compile and execute the code, it will produce the following output − Size of a: 4 bytes Size of b: 4 bytes Size of c: 20 bytes Size of union: 20 bytes