

# B.C.A study

## Unit-1: Arrays

### what is array

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

### Declaration and initialization of array

An array is a variable that can store multiple values. For example, if you want to store 100 integers, you can create an array for it.

```
dataTypearrayName[arraySize];
```

```
int data[100];
```

It is possible to initialize an array during declaration. For example,

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

You can also initialize an array like this.

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

### Accessing array

Array can be accessed using array-name and subscript variable written inside pair of square brackets [].

arr[3] = Third Element of Array

arr[5] = Fifth Element of Array

arr[8] = Eighth Element of Array

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

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

```
void main()
```

```
{
```

```
intarr[] = {51,32,43,24,5,26};
```

```
int i;
```

```
for(i=0; i<=5; i++) {
```

```
printf("\nElement at arr[%d] is %d",i,arr[i]);
```

```
    }
```

```
getch();
```

```
}
```

## Displaying array

The implementation of the above derived pseudocode is as follows –

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

```
int main() {
```

```
int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
```

```
int loop;
```

```
for(loop = 0; loop < 10; loop++)
```

```
printf("%d ", array[loop]);
```

```
return 0;
```

```
}
```

The output should look like this –

1 2 3 4 5 6 7 8 9 0

# Sorting arrays and functions

Sorting is the method of arranging a given array in accending or decending order.

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

```
#define ARRAY_SIZE 5
```

```
intmain()
```

```
{
```

```
int numbers[ARRAY_SIZE], i ,j ,temp;
```

```
    // Read Input
```

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

```
{
```

```
    printf("Enter the Number : %d : ", (i+1));
```

```
    scanf("%d", &numbers[i]);
```

```
}
```

```
    // Array Sorting – Ascending Order
```

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

```
{
```

```
for (j = i + 1; j < ARRAY_SIZE; ++j)
```

```
{
```

```
if (numbers[i] > numbers[j])
```

```
{
```

```
temp = numbers[i];
```

```
numbers[i] = numbers[j];
```

```
numbers[j] = temp;
```

```
}
```

```
}
```

```
}  
  
printf("Sorting Order Array: \n");  
  
for (i = 0; i < ARRAY_SIZE; ++i)  
  
printf("%d\n", numbers[i]);  
  
return 0;  
  
}
```

Enter the elements you want to sort:

45, 50, 36, 3, 87,

Sorted array: 3, 36, 45, 50, 87.

## Two dimensional array:

An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns.

### declaration

The syntax declaration of 2-D array is not much different from 1-D array. In 2-D array, to declare and access elements of a 2-D array we use 2 subscripts instead of 1.

**Syntax:** datatype array\_name[ROW][COL];

The total number of elements in a 2-D array is ROW\*COL. Let's take an example.

```
int arr[2][3];
```

This array can store  $2*3=6$  elements. You can visualize this 2-D array as a matrix of 2 rows and 3 columns.

### initialization

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {  
    {10, 11, 12, 13},  
    {14, 15, 16, 17}  
};
```

OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Although both the above declarations are valid, I recommend you to use the first method as it is more readable, because you can visualize the rows and columns of 2d array in this method.

## accessing of array element

	Column 0	Column 1	Column 2
Row 0	<b>x[0][0]</b>	<b>x[0][1]</b>	<b>x[0][2]</b>
Row 1	<b>x[1][0]</b>	<b>x[1][1]</b>	<b>x[1][2]</b>
Row 2	<b>x[2][0]</b>	<b>x[2][1]</b>	<b>x[2][2]</b>

The individual elements of the above array can be accessed by using two subscript instead of one. The first subscript denotes row number and second denotes column number. As we can see in the above image both rows and columns are indexed from 0. So the first element of this array is at arr[0][0] and the last element is at arr[1][2]. Here are how you can access all the other elements:

arr[0][0] – refers to the first element

arr[0][1] – refers to the second element

arr[0][2] – refers to the third element

arr[1][0] – refers to the fourth element

arr[1][1] – refers to the fifth element

arr[1][2] – refers to the sixth element

### Simple Two dimensional(2D) Array Example

```
#include<stdio.h>
int main(){
    /* 2D array declaration*/
    int disp[2][3];
    /*Counter variables for the loop*/
    int i, j;
    for(i=0; i<2; i++) {
        for(j=0; j<3; j++) {
            printf("Enter value for disp[%d][%d]:", i, j);
            scanf("%d", &disp[i][j]);
        }
    }
    //Displaying array elements
    printf("Two Dimensional array elements:\n");
    for(i=0; i<2; i++) {
        for(j=0; j<3; j++) {
            printf("%d ", disp[i][j]);
            if(j==2){
                printf("\n");
            }
        }
    }
    return 0;
}
```

Output:

```
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6
```

# Memory representations of array

A 2D array is stored in the computer's memory one row following another. The address of the first byte of memory is considered as the memory location of the entire 2D array. ... Again one should not think of a 2D array as just an array with two indexes. You should think of it as an array of arrays.

## Row Major System:

The address of a location in Row Major System is calculated using the following formula:

$$\text{Address of } A[I][J] = B + W * [N * (I - L_r) + (J - L_c)]$$

## Column Major System:

The address of a location in Column Major System is calculated using the following formula:

$$\text{Address of } A[I][J] \text{ Column Major Wise} = B + W * [(I - L_r) + M * (J - L_c)]$$

Where,

B = Base address

I = Row subscript of element whose address is to be found

J = Column subscript of element whose address is to be found

W = Storage Size of one element stored in the array (in byte)

L<sub>r</sub> = Lower limit of row/start row index of matrix, if not given assume 0 (zero)

L<sub>c</sub> = Lower limit of column/start column index of matrix, if not given assume 0 (zero)

M = Number of row of the given matrix

N = Number of column of the given matrix

# Multidimensional array

A multidimensional array is an array containing one or more arrays. PHP understands multidimensional arrays that are two, three, four, five, or more levels deep. However, arrays more than three levels deep are hard to manage for most people.

It is an array of arrays; an array that has multiple levels. The simplest multi-dimensional array is the 2D array, or two-dimensional array. It's technically an array of arrays, as you will see in the code. A 2D array is also called a matrix, or a table of rows and columns.

Declaring a multi-dimensional array is similar to the one-dimensional arrays. For a 2D array, we need to tell C that we have 2 dimensions.

...

**[A WordPress.com Website.](#)**



## B.C.A study

# UNIT -2: Pointers

## Pointers

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is `–type *var-name;`

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk `*` used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –`int *ip;` /\* pointer to an integer \*/ `double *dp;` /\* pointer to a double \*/ `float *fp;` /\* pointer to a float \*/ `char *ch` /\* pointer to a character \*/

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator `*` that returns the value of the variable located at the address specified by its operand.

## NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h> int main () { int *ptr = NULL; printf("The value of ptr is : %x\n", ptr ); return 0; }
```

When the above code is compiled and executed, it produces the following result –The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –if(ptr) /\* succeeds if p is not null \*/ if(!ptr) /\* succeeds if p is null \*/

## Indirection operator

An indirection operator, in the context of C#, is an operator used to obtain the value of a variable to which a pointer points. While a pointer pointing to a variable provides an indirect access to the value of the variable stored in its memory address, the indirection operator dereferences the pointer and returns the value of the variable at that memory location. The indirection operator is a unary operator represented by the symbol (\*).

The indirection operator can be used in a pointer to a pointer to an integer, a single-dimensional array of pointers to integers, a pointer to a char, and a pointer to an unknown type.

The indirection operator is also known as the dereference operator.

The (\*) symbol is used in declaring pointer types and in performing pointer indirection, while the 'address-of' operator (&) returns the address of a variable. Hence, the indirection operator and the address-of operator are inverses of each other.

C# allows using pointers only in an unsafe region, which implies that the safety of the code within that region is not verified by the common language runtime (CLR). In the unsafe region, the indirection operator is allowed to read and write to a pointer. The following C# statements illustrate the usage of the indirection operator:

- o int a = 1, b; // line 1
- o int \*pInt = &a; // line 2
- o b = \*pInt; // line 3

In the first line above, a and b are integer variables and a is assigned a value of 1. In line 2, the address of a is stored in the integer pointer pInt (line 2). The dereference operator is used in line 3 to assign the value at the address pointed to by pInt to the integer variable b.

The indirection operator should be used to dereference a valid pointer with an address aligned to the type it points to, so as to avoid undefined behavior at runtime. It should not be applied to a void pointer or to an expression that is not of a pointer type, to avoid compiler errors. However, after casting a void pointer to the right pointer type, the indirection operator can be used.

When declaring multiple pointers in a single statement, the indirection operator should be written only once with the underlying type and not repeated for each pointer name. The indirection operator is distributive in C#, unlike C and C++. When the indirection operator is applied to a null pointer, it results in an implementation-defined behavior. Since this operator is used in an unsafe context, the keyword unsafe should be used before it along with the /unsafe option during compilation.

## Address operators

The "Address Of" Operator denoted by the **ampersand character** (&), & is a unary operator, which returns the address of a variable.

After declaration of a pointer variable, we need to initialize the pointer with the valid memory address; to get the memory address of a variable **Address Of** (&) Operator is used.

**Consider the program** `#include <stdio.h>. int main() {. int x=10; //integer variable *ptrX; //integer pointer declaration ptrX=&x; //pointer initialization with the address of x printf("Value of x is %d\n",*ptrX); return 0; }`

OutputValue of x: 10

## Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
  - Decrement
  - Addition
  - Subtraction
  - Comparison
- 

## Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

1.  $\text{new\_address} = \text{current\_address} + i * \text{size\_of}(\text{data type})$

Where  $i$  is the number by which the pointer get increased.

### 32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

### 64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```

1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+1;
8. printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by 4 bytes.
9. return 0;
10. }

```

## Output

Address of p variable is 3214864300 After increment: Address of p variable is 3214864304

---

## Traversing an array by using pointer

```

1. #include<stdio.h>
2. void main ()
3. {
4.   int arr[5] = {1, 2, 3, 4, 5};
5.   int *p = arr;
6.   int i;
7.   printf("printing array elements... \n");
8.   for(i = 0; i < 5; i++)
9.   {
10.    printf("%d ",*(p+i));
11.   }
12. }

```

## Output

printing array elements... 1 2 3 4 5

---

## Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

```
1. new_address= current_address - i * size_of(data type)
```

## 32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

## 64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```
1. #include <stdio.h>
2. void main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-1;
8. printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immediate previous location.
9. }
```

## Output

Address of p variable is 3214864300 After decrement: Address of p variable is 3214864296

---

## C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

```
1. new_address= current_address + (number * size_of(data type))
```

## 32-bit

For 32-bit int variable, it will add 2 \* number.

## 64-bit

For 64-bit int variable, it will add 4 \* number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+3; //adding 3 to pointer variable
8. printf("After adding 3: Address of p variable is %u \n",p);
```

```

9. return 0;
10. }

```

## Output

Address of p variable is 3214864300 After adding 3: Address of p variable is 3214864312

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e.,  $4 \times 3 = 12$  increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e.,  $2 \times 3 = 6$ . As integer value occupies 2-byte memory in 32-bit OS.

---

## C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

1.  $\text{new\_address} = \text{current\_address} - (\text{number} * \text{size\_of}(\text{data type}))$

### 32-bit

For 32-bit int variable, it will subtract  $2 * \text{number}$ .

### 64-bit

For 64-bit int variable, it will subtract  $4 * \text{number}$ .

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```

1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-3; //subtracting 3 from pointer variable
8. printf("After subtracting 3: Address of p variable is %u \n",p);
9. return 0;
10. }

```

## Output

Address of p variable is 3214864300 After subtracting 3: Address of p variable is 3214864288

You can see after subtracting 3 from the pointer variable, it is 12 ( $4 \times 3$ ) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1.  $\text{Address2} - \text{Address1} = (\text{Subtraction of two addresses}) / \text{size of data type which pointer points}$

Consider the following example to subtract one pointer from another.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i = 100;
5.     int *p = &i;
6.     int *temp;
7.     temp = p;
8.     p = p + 3;
9.     printf("Pointer Subtraction: %d - %d = %d", p, temp, p-temp);
10. }
```

## Output

Pointer Subtraction: 1030585080 – 1030585068 = 3

## Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
  - Address \* Address = illegal
  - Address % Address = illegal
  - Address / Address = illegal
  - Address & Address = illegal
  - Address ^ Address = illegal
  - Address | Address = illegal
  - ~Address = illegal
- 

## Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

```
1. #include<stdio.h>
2. int addition ();
3. int main ()
```

```

4. {
5.   int result;
6.   int (*ptr)();
7.   ptr = &addition;
8.   result = (*ptr)();
9.   printf("The sum is %d",result);
10. }
11. int addition()
12. {
13.   int a, b;
14.   printf("Enter two numbers?");
15.   scanf("%d %d",&a,&b);
16.   return a+b;
17. }

```

## Output

Enter two numbers?10 15 The sum is 25

## Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example.

```

1. #include<stdio.h>
2. int show();
3. int showadd(int);
4. int (*arr[3])();
5. int (*(*ptr)[3])();
6.
7. int main ()
8. {
9.   int result1;
10.  arr[0] = show;
11.  arr[1] = showadd;
12.  ptr = &arr;
13.  result1 = (**ptr)();
14.  printf("printing the value returned by show : %d",result1);
15.  ((*ptr+1))(result1);
16. }
17. int show()
18. {
19.   int a = 65;
20.   return a++;
21. }
22. int showadd(int b)
23. {
24.   printf("\nAdding 90 to the value returned by show: %d",b+90);
25. }

```



## Output

printing the value returned by show : 65 Adding 90 to the value returned by show: 155

## Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation. static memory allocation memory is allocated at compile time. dynamic memory allocation memory is allocated at run time. static memory can't be increased while executing program. dynamic memory can be increased while executing program. static memory is used in array. dynamic memory is used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation. **malloc()** allocates single block of requested memory. **calloc()** allocates multiple block of requested memory. **realloc()** reallocates the memory occupied by malloc() or calloc() functions. **free()** frees the dynamically allocated memory.

---

## malloc() function in C

The malloc() function allocates single block of requested memory. It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

1. ptr=(cast-type\*)malloc(byte-size)

Let's see the example of malloc() function.

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4.   int n,i,*ptr,sum=0;
5.   printf("Enter number of elements: ");
6.   scanf("%d",&n);
7.   ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
8.   if(ptr==NULL)
9.   {
10.    printf("Sorry! unable to allocate memory");
11.    exit(0);
12.  }
13.  printf("Enter elements of array: ");

```

```

14.  for(i=0;i<n;++i)
15.  {
16.      scanf("%d",&ptr[i]);
17.      sum+=*(ptr+i);
18.  }
19.  printf("Sum=%d",sum);
20.  free(ptr);
21. return 0;
22. }

```

Output:Enter elements of array: 3 Enter elements of array: 10 10 10 Sum=30

---

## calloc() function in C

The calloc() function allocates multiple block of requested memory. It initially initialize all bytes to zero. It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

```
1. ptr=(cast-type*)calloc(number, byte-size)
```

Let's see the example of calloc() function.

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4.  int n,i,*ptr,sum=0;
5.  printf("Enter number of elements: ");
6.  scanf("%d",&n);
7.  ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
8.  if(ptr==NULL)
9.  {
10.   printf("Sorry! unable to allocate memory");
11.   exit(0);
12.  }
13.  printf("Enter elements of array: ");
14.  for(i=0;i<n;++i)
15.  {
16.      scanf("%d",&ptr[i]);
17.      sum+=*(ptr+i);
18.  }
19.  printf("Sum=%d",sum);
20.  free(ptr);
21. return 0;
22. }

```

Output:Enter elements of array: 3 Enter elements of array: 10 10 10 Sum=30

---

## realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, new-size)

---

## free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. free(ptr)



**[A WordPress.com Website.](#)**

# B.C.A study

## Unit-3 : Strings

### definition

The string can be defined as the one-dimensional array of characters terminated by a null ('\\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0 .

### declaration

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

1. **char** ch[10]={'j','a','v','a','t','p','o','i','n','t','\\0'};

As we know, array index starts from 0, so it will be represented as in the figure given below.

0	1	2	3	4	5	6	7	8	9	10
j	a	v	a	t	p	o	i	n	t	\\0

While declaring string, size is not mandatory. So we can write the above code as given below:

1. **char** ch[]={ 'j','a','v','a','t','p','o','i','n','t','\\0'};

We can also define the **string by the string literal** in C language. For example:

1. **char** ch[]="javatpoint";

In such case, '\0' will be appended at the end of the string by the compiler.

## Initializing String [Character Array] :

Whenever we declare a String then it will contain garbage values inside it. We have to initialize String or Character array before using it. Process of Assigning some legal default data to String is Called **Initialization of String**. There are different ways of initializing String in C Programming –

1. Initializing Unsized Array of Character
2. Initializing String Directly
3. Initializing String Using Character Pointer

### Way 1 : Unsized Array and Character

1. **Unsized Array** : Array Length is not specified while initializing character array using this approach
2. Array length is Automatically calculated by Compiler
3. Individual Characters are written inside Single Quotes , Separated by comma to form a list of characters. Complete list is wrapped inside **Pair of Curly braces**
4. **Please Note** : NULL Character should be written in the list because it is ending or terminating character in the String/Character Array

```
char name [] = {'P','R','I','T','E','S','H','\0'};
```

### Way 2 : Directly initialize String Variable

1. In this method we are directly assigning String to variable by writing text in double quotes.
2. In this type of initialization , we don't need to put NULL or **Ending / Terminating character** at the end of string. It is appended automatically by the compiler.

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

### Way 3 : Character Pointer Variable

1. Declare Character variable of pointer type so that it can hold the **base address of "String"**
2. Base address means address of first array element i.e (**address of name[0]** )
3. NULL Character is appended **Automatically**

```
char *name = "PRITESH";
```

## standard library function

C supports a large number of string handling functions in the standard library "string.h".

Few commonly used string handling functions are discussed below:

FUNCTION	WORK OF FUNCTION
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings

Strings handling functions are defined under "string.h" header file.

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

## gets() and puts()

Functions gets() and puts() are two string functions to take string input from the user and display it respectively as mentioned in the **previous chapter** (<https://www.programiz.com/c-programming/c-strings>).

```
#include<stdio.h>int main(){    char name[30];    printf("Enter name: ");  
gets(name);        //Function to read string from user.    printf("Name: ");  
puts(name);        //Function to display string.    return 0;}
```

**Note:** Though, `gets()` and `puts()` function handle strings, both these functions are defined in "`stdio.h`" header file.

## 1. `strlen()` Function

This function counts the number of characters present in a string. Its usage is illustrated in the following program.

### Program to demonstrate `strlen()` function

```
char arr[ ] = "Generalnote" ;
int len1, len2 ;
len1 = strlen ( arr ) ;
len2 = strlen ( "C Tutorial" ) ;
printf(" \nstring = %s length = %d ", arr, len1) ;
printf(" \nstring = %s length = %d ", "C Tutorial", len2) ;
return ( 0 ) ;
}
```

### Output :

```
string = Generalnote length = 11
string = C Tutorial length = 10
```

In the first call to the function `strlen()`, we are passing the base address of the string, and the function in turn returns the length of the string. While calculating the length it doesn't count '`\0`'.

## 2. `strcpy()` Function

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function.

### Program to demonstrate `strcpy()` function

```
\* C Program to demonstrate strcpy() function *\
```

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

```

char source[ ] = "General note" ;
char target[25];
strcpy( target, source) ;
printf(" \nsource string = %s ", source) ;
printf(" \ntarget string = %s ", target) ;
return ( 0 ) ;
}

```

## Output :

```

source string = Generalnote
target string = Generalnote

```

On supplying the base addresses, `strcpy( )` goes on copying the characters in source string into the target string till it doesn't encounter the end of source string (`'\0'`). It is our responsibility to see to it that the target string's dimension is big enough to hold the string being copied into it. Thus, a string gets copied into another, piece-meal, character by character.

## 3.strcat() function

This function concatenates the source string at the end of the target string.

## Program to demonstrate strcpy() function

```

\* C Program to demonstrate strcpy() function *\

```

```

# include <stdio.h>
int main( )
{

char source[ ] = "General" ;
char target[30] = "Note" ;
strcat( target, source) ;
printf(" \nsource string = %s ", source) ;
printf(" \ntarget string = %s ", target) ;
return ( 0 ) ;
}

```

## Output :

```

source string = General
target string = GeneralNote

```



## 4.strcmp( ) Function

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, strcmp( ) returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters.

### Program to demonstrate strcmp() function

```
\* C Program to demonstrate strcmp() function *\
```

```
# include <stdio.h>
int main( )
{
    char string1[ ] = "General" ;
    char string2[ ] = "Note" ;
    int i, j, k ;
    i = strcmp( string1, "General") ;
    j = strcmp( string1, string2) ;
    k = strcmp( string1, "General Note") ;
    printf(" \n %d %d %d ", i, j, k) ;
    return ( 0 ) ;
}
```

Output :

```
0 6 -32
```

## implementation without using standard library functions

### 1.Calculate Length of String without Using strlen() Function

```
#include <stdio.h>

int stringLength(char*);

int main()
{
    char str[100]={0};
    int length;

    printf("Enter any string: ");
    scanf("%s",str);

    length=stringLength(str);

    printf("String length is : %d\n",length);

    return 0;
}

int stringLength(char* txt)
{
    int i=0,count=0;

    while(txt[i++]!='\0'){
        count+=1;
    }

    return count;
}
```

## Output

```
Enter any string: IncludeHelp
String length is : 11
```

## 2.C Program to Copy One String into Other Without Using Library Function.

1. #include <stdio.h>
- 2.
3. int main()

```
4. {  
5.  int c = 0;  
6.  char s[1000], d[1000] = "What can I say about my programming skills?";  
7.  
8.  printf("Before copying, the string: %s\n", d);  
9.  
10. printf("Input a string to copy\n");  
11. gets(s);  
12.  
13. while (s[c] != '\0') {  
14.     d[c] = s[c];  
15.     c++;  
16. }  
17.  
18. d[c] = '\0';  
19.  
20. printf("After copying, the string: %s\n", d);  
21.  
22. return 0;  
23. }
```

Output of the program:

1. Before copying, the string: What can I say about my programming skills?
2. Input a string to copy
3. My programming skills are improving.
4. After copying, the string: My programming skills are improving.

...

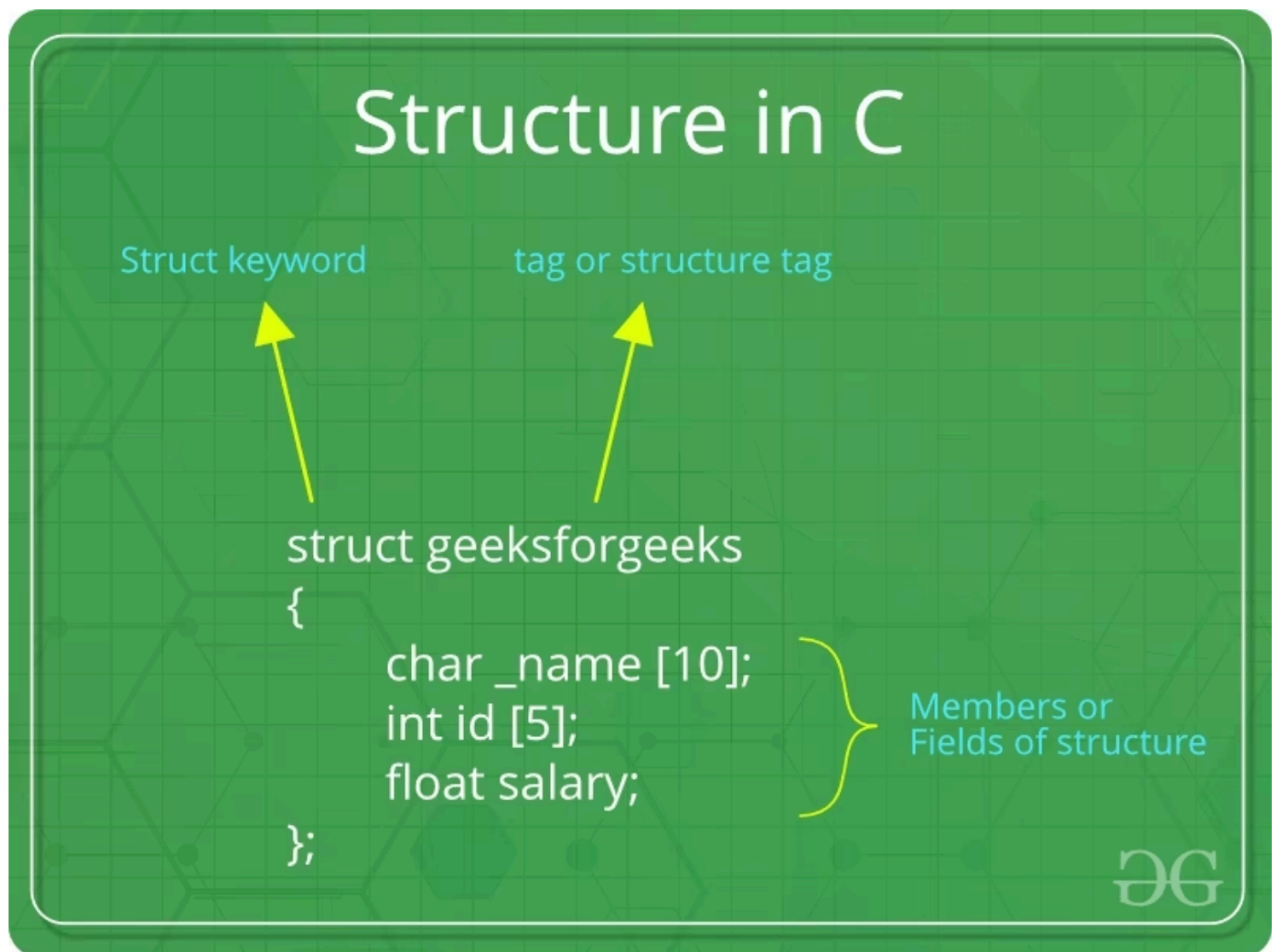
**[A WordPress.com Website.](https://bcastudyguide.com/unit-3-strings/)**

# B.C.A study

## Unit-4: Structure

### *What is a structure?*

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.



***How to create a structure?***

'struct' keyword is used to create a structure. Following is an example. struct address { char name[50]; char street[100]; char city[50]; char state[20]; int pin;};

***How to declare structure variables?***

A structure variable can either be declared with structure declaration or as a separate declaration like basic types. // A variable declaration with structure declaration. struct Point { int x, y; } p1; // The variable p1 is declared with 'Point'. // A variable declaration like basic data types struct Point { int x, y; }; int main() { struct Point p1; // The variable p1 is declared like a normal variable }

***How to initialize structure members?***

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation. struct Point { int x = 0; // COMPILER ERROR: cannot initialize members here int y = 0; // COMPILER ERROR: cannot initialize members here};

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces '{}'. For example, following is a valid initialization.

struct Point { int x, y; }; int main() { // A valid initialization. member x gets value 0 and y. // gets value 1. The order of declaration is followed.

struct Point p1 = {0, 1}; }

***How to access structure elements?***

Structure members are accessed using dot (.) operator.

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

```
struct Point
```

```
{
```

```
int x, y;};
```

```
int main()
```

```
{ struct Point p1 = {0, 1}; // Accessing members of point p1
```

```
p1.x = 20;
```

```
printf("x = %d, y = %d", p1.x, p1.y);
```

```
return 0;}
```

**Output:** x = 20, y = 1

***What is a structure pointer?***

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow ( -> ) operator.

```
#include<stdio.h>

struct Point{intx, y;};

int main()

{struct Point p1 = {1, 2}; // p2 is a pointer to structure p1

struct Point *p2 = &p1; // Accessing structure members using structure pointer

printf("%d %d", p2->x, p2->y);

return0;

}
```

**Output:**1 2

## Nested Structures in C

A structure can be nested inside another structure. In other words, the members of a structure can be of any other type including structure. Here is the syntax to create nested structures.

**Syntax:**

```
structure tagname_1

{ member1;

member2;

member3; ...

membern;

structure tagname_2

{ member_1;

member_2;

member_3;

... member_n;
```

```
}, var1
```

```
} var2;
```

**Note:** Nesting of structures can be extended to any level.

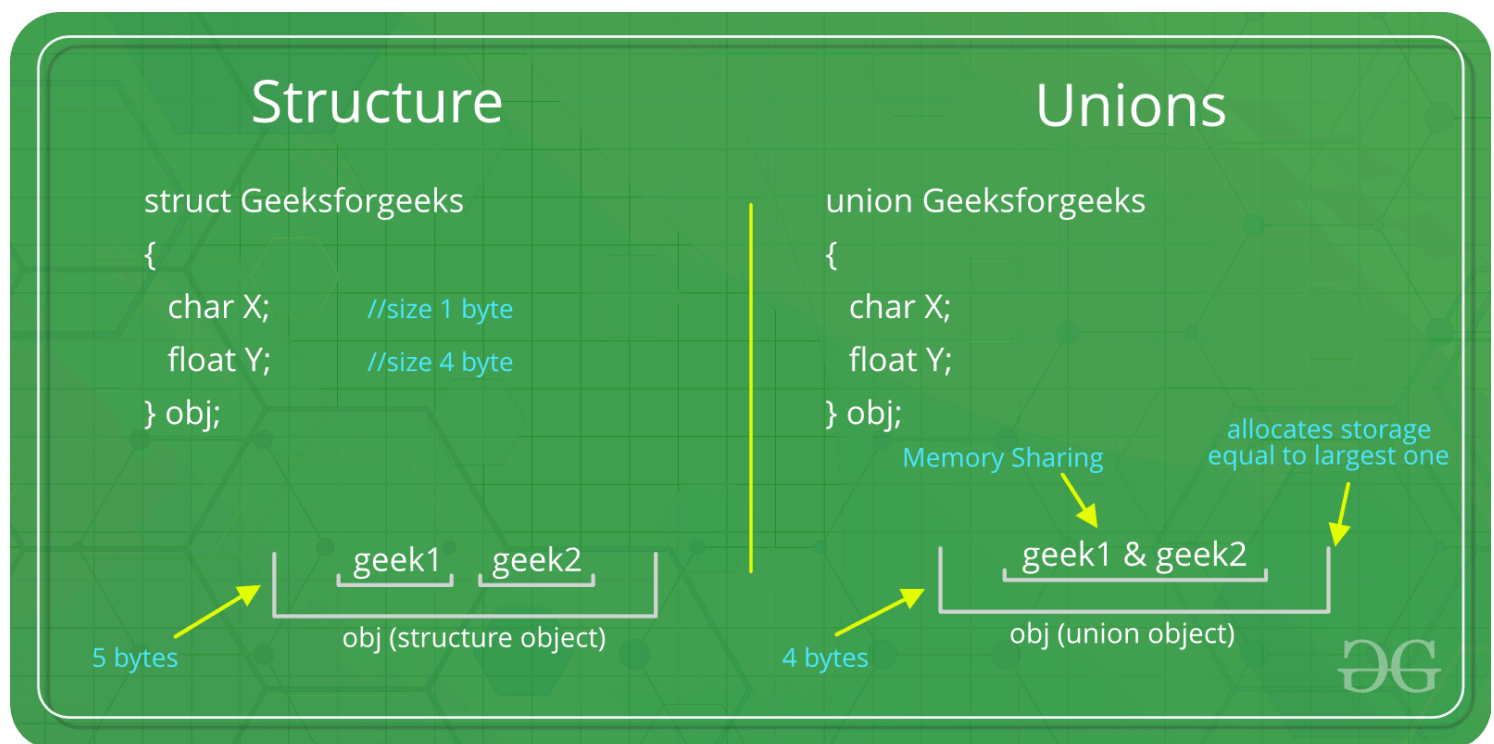
To access the members of the inner structure, we write a variable name of the outer structure, followed by a dot(.) operator, followed by the variable of the inner structure, followed by a dot(.) operator, which is then followed by the name of the member we want to access.

var2.var1.member\_1 – refers to the member\_1 of structure tagname\_2

var2.var1.member\_2 – refers to the member\_2 of structure tagname\_2

## Union in C

Like Structure union is a user defined data type. In union, all members share the same memory location.



For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

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

```
// Declaration of union is same as structures
```

```
union test
```

```

{ intx, y; };

int main() {

// A union variable t

union test t;

t.x = 2;

// t.y also gets value 2

printf("After making x = 2:\n x = %d, y = %d\n\n", t.x, t.y);

t.y = 10; // t.x is also updated to 10

printf("After making y = 10:\n x = %d, y = %d\n\n", t.x, t.y);

return 0;

}

```

**Output:**After making x = 2:

x = 2, y = 2

After making y = 10:

x = 10, y = 10

### Pointers to unions

Like structures, we can have pointers to unions and can access members using the arrow operator (->). The following example demonstrates the same.

```

#include <stdio.h>

union test

{ intx; chary; };

int main()

{

union test p1;

p1.x = 65; // p2 is a pointer to union p1

union test* p2 = &p1; // Accessing union members using pointer

printf("%d %c", p2->x, p2->y);

return 0;

```



}

**Output:**65 A**Difference between STRUCTURES UNION**

Struct keyword is used to declare the structure Union keyword is used to declare the Union

Structure variable will allocate memory for all the structure members separately. Union variable will allocate common memory for all the union members.

Example:struct Employee{ int age; char name[50]; float salary; };

**Example:**union Employee{ int age; char name[50]; float salary; };

Structures will occupy more memory space.Memory\_Size = addition of all the structure members sizes.


Memory\_Size = int + char array [50] + float

Memory\_Size = 2 + 50 + 4 Bytes

Memory\_Size = 56 Byte

Union will occupy less memory space compared to structures.Memory\_Size = Size of the largest Union member. From the above example, the Largest Union member is char array. So, Memory\_Size = 50 Bytes

Structure allows us to access any or all the members at any time.

Union allows us to access only one union member  at a time.

**[A WordPress.com Website.](#)**

## B.C.A study.

# Unit-5: Introduction to C preprocessor

## what is c preprocessor?

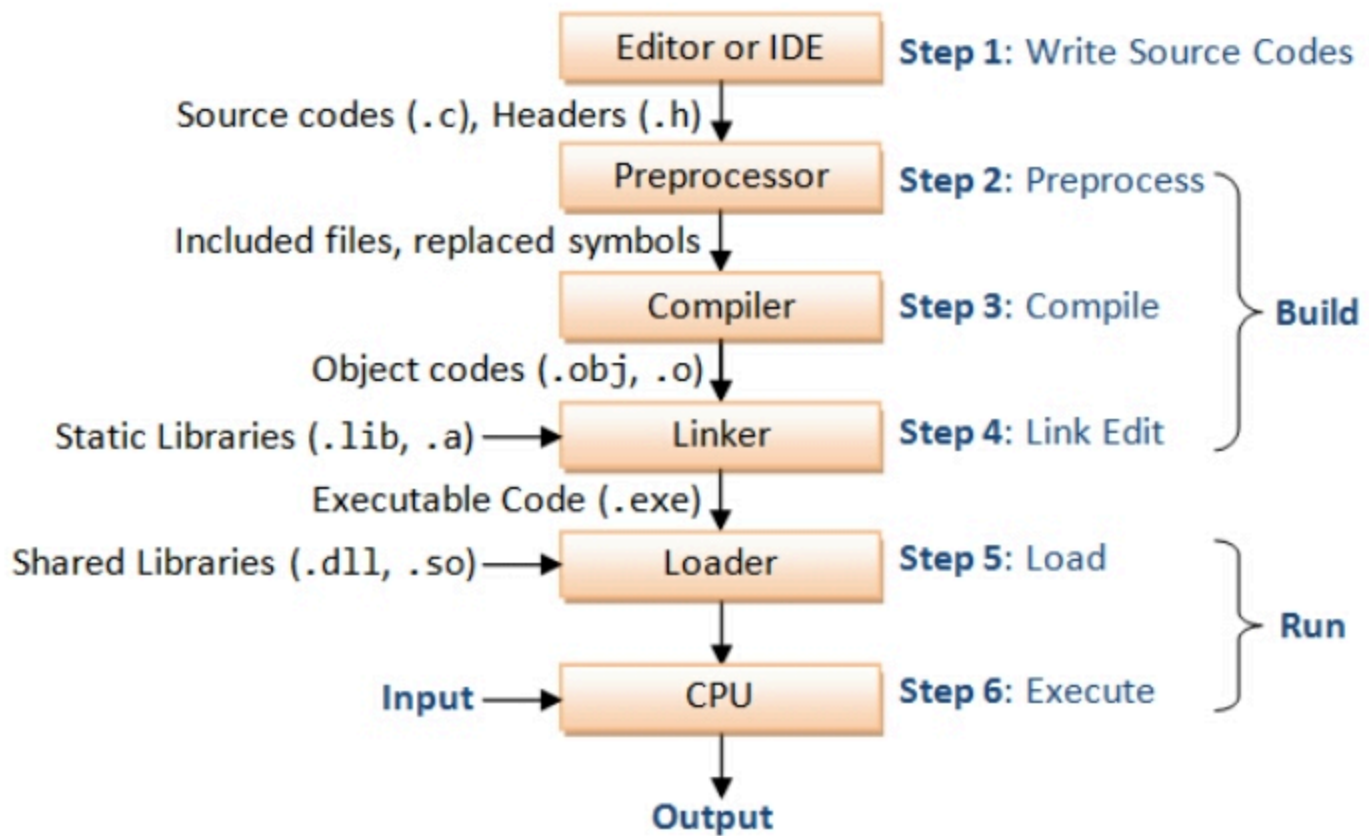
The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

The **C preprocessor** is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor provides four separate facilities that you can use as you see fit:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

## how c preprocessor works



## preprocessor directives

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Directive	Function
<b>#define</b>	Defines a <b>Macro</b> Substitution
<b>#undef</b>	Undefines a <b>Macro</b>
<b>#include</b>	Includes a File in the Source Program
<b>#ifdef</b>	Tests for a <b>Macro</b> Definition
<b>#endif</b>	Specifies the end of #if
<b>#ifndef</b>	Checks whether a <b>Macro</b> is defined or not
<b>#if</b>	Checks a Compile Time Condition
<b>#else</b>	Specifies alternatives when #if Test Fails

[bcastudyguide.com \(https://bcastudyguide.wordpress.com\)](https://bcastudyguide.wordpress.com)

## Macro sustitution directives

## Macro substitution

Macro substitution has a name and replacement text, defined with #define directive. The preprocessor simply replaces the name of macro with replacement text from the place where the macro is defined in the source code.



Now we will see the fact through an example.

```
1.#include <stdio.h>
2.#define PI 3.1415
3.
4.int main()
5.{
6.float radius, area;
7.printf("Enter the radius: ");
8.scanf("%f", &radius);
9.
10.// Notice, the use of PI
11.area = PI*radius*radius;
12.
13.printf("Area=%.2f", area);
14.return 0;
```

15. }

## Example 2: Using #define preprocessor

```
1.#include <stdio.h>
2.#define PI 3.1415
3.#define circleArea(r) (PI*r*r)
4.
5.int main() {
6.float radius, area;
7.
8.printf("Enter the radius: ");
9 scanf("%f", &radius);
10.area = circleArea(radius);
11.printf("Area = %.2f", area);
12.
13.return 0;
14. }
```

## file inclusion directives

1. File inclusive Directories are **used to include user define header file** inside C Program.
2. File inclusive directory checks included header file inside same directory (if path is not mentioned).
3. File inclusive directives begins with **#include**
4. If Path is mentioned then it will include that **header file into current scope**.
5. Instead of using triangular brackets we use **"Double Quote"** for inclusion of user defined header file.
6. It instructs the compiler to include **all specified files**.

## Ways of including header file

### way 1 : Including Standard Header Files

```
#include<filename>
```

- Search File in Standard Library

### way 2 :User Defined Header Files are written in this way

```
#include"FILENAME"
```

- Search File in Current Library
- If not found , Search File in Standard Library
- User defined header files are written in this format

## Live Example :

```
#include<stdio.h>    // Standard Header File
#include<conio.h>     // Standard Header File
#include"myfunc.h"    // User Defined Header File
```

### Explanation :

1. In order to include user defined header file inside C Program , we must have to **create one user defined header file**.
2. Using double quotes include user defined header file inside Current C Program.
3. "myfunc.h" is user defined header file .
4. **We can combine all our user defined functions** inside header file and can include header file whenever require.

## conditional compilation

**Conditional Compilation:** Conditional Compilation directives help to compile a specific portion of the program or let us skip compilation of some specific part of the program based on some condition.

1. **#ifdef:** This directive is the simplest conditional directive. This block is called a conditional group. The controlled text will get included in the preprocessor output iff the macroname is defined. The controlled text inside a conditional will embrace preprocessing directives. They are executed only if the conditional succeeds. You can nest these in multiple layers, but they must be completely nested. In other words, '#endif' always matches the nearest '#ifdef' (or '#ifndef', or '#if'). Also, you can't begin a conditional group in one file and finish it in another. **Syntax:**

```
#ifdef MACRO
    controlled text
#endif /* macroname */
```

**2. #ifndef:** We know that the in #ifdef directive if the macroname is defined, then the block of statements following the #ifdef directive will execute normally but if it is not defined, the compiler will simply skip this block of statements. The #ifndef directive is simply opposite to that of the #ifdef directive. In case of #ifndef , the block of statements between #ifndef and #endif will be executed only if the macro or the identifier with #ifndef is not defined.

**Syntax :**

```
#ifndef macro_name
    statement1;
    statement2;
    statement3;
    .
    .
    .
    statementN;
#endif
```

If the macro with name as 'macroname' is not defined using the #define directive then only the block of statements will execute.

**3.#if, #else and #elif:** These directives works together and control compilation of portions of the program using some conditions. If the condition with the #if directive evaluates to a non zero value, then the group of line immediately after the #if directive will be executed otherwise if the condition with the #elif directive evaluates to a non zero value, then the group of line immediately after the #elif directive will be executed else the lines after #else directive will be executed.

**Syntax:**

```
#if macro_condition
    statements
#elif macro_condition
    statements
#else
    statements
#endif
```

...

[A WordPress.com Website.](https://bcastudyguide.com/unit-4-introduction-to-c-preprocessor/)





# B.C.A study.

## Unit- 6 :File handling

### File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
  - Opening an existing file
  - Reading from the file
  - Writing to the file
  - Deleting the file
- 

### Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
-----	----------	-------------

1	fopen()	opens new or existing file
---	---------	----------------------------

2	fprintf()	write data into the file
---	-----------	--------------------------

3	fscanf()	reads data from the file
---	----------	--------------------------

4 fputc()-writes a character into the file

5 fgetc()-reads a character from file

6 fclose()-closes the file

7 fseek()-sets the file pointer to given position

8 fputw() writes an integer to file

9 fgetw()reads an integer from file

10 ftell()-returns current position

11 rewind()-sets the file pointer to the beginning of the file

---

## Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

1. FILE \*fopen( const char \* filename, const char \* mode );

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like “c://some\_folder/some\_file.ext”.
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
r	opens a binary file in read mode
w	opens a binary file in write mode
a	opens a binary file in append mode
r+	opens a binary file in read and write mode
w+	opens a binary file in read and write mode
a+	opens a binary file in read and write mode

---

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
1. #include<stdio.h>
2. void main( )
3. {
4. FILE *fp ;
5. char ch ;
6. fp = fopen("file_handle.c","r") ;
7. while ( 1 )
8. {
9. ch = fgetc ( fp ) ;
10. if ( ch == EOF )
11. break ;
12. printf("%c",ch) ;
13. }
14. fclose (fp ) ;
15. }
```

## Output

The content of the file will be printed. #include;. void main( ) { FILE \*fp; // file pointer. char ch; fp = fopen("file\_handle.c","r"); while ( 1 ) { ch = fgetc ( fp ); //Each character of the file is read and stored in the character file. if ( ch == EOF ) break; printf("%c",ch); }. fclose (fp); }

## Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

```
1. int fclose( FILE *fp );
```

## feof()

## Description

The C library function **int feof(FILE \*stream)** tests the end-of-file indicator for the given stream.

## Declaration

Following is the declaration for feof() function.

```
int feof(FILE *stream)
```

## Parameters

- **stream** – This is the pointer to a FILE object that identifies the stream.

## Return Value

This function returns a non-zero value when End-of-File indicator associated with the stream is set, else zero is returned.

## Example

The following example shows the usage of feof() function.

```
#include <stdio.h>

int main () {
    FILE *fp;
    int c;

    fp = fopen("file.txt","r");
    if(fp == NULL) {
        perror("Error in opening file");
        return(-1);
    }

    while(1) {
        c = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
        printf("%c", c);
    }
    fclose(fp);

    return(0);
}
```

## rewind( )

### Description

The C library function **void rewind(FILE \*stream)** sets the file position to the beginning of the file of the given **stream**.

### Declaration

Following is the declaration for rewind() function.

```
void rewind(FILE *stream)
```

## Parameters

- **stream** – This is the pointer to a FILE object that identifies the stream.

## Return Value

This function does not return any value.

## Example

The following example shows the usage of `rewind()` function.

```
#include <stdio.h>

int main () {
    char str[] = "This is tutorialspoint.com";
    FILE *fp;
    int ch;

    /* First let's write some content in the file */
    fp = fopen( "file.txt" , "w" );
    fwrite(str , 1 , sizeof(str) , fp );
    fclose(fp);

    fp = fopen( "file.txt" , "r" );
    while(1) {
        ch = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
        printf("%c", ch);
    }
    rewind(fp);
    printf("\n");
    while(1) {
        ch = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
        printf("%c", ch);
    }
    fclose(fp);

    return(0);
}
```

## fputc() and fgetc()

---

### Writing File : fputc() function

The `fputc()` function is used to write a single character into file. It outputs a character to a stream.

**Syntax:**

```
1. int fputc(int c, FILE *stream)
```

**Example:**

```
1. #include <stdio.h>
2. main(){
3.  FILE *fp;
4.  fp = fopen("file1.txt", "w");//opening file
5.  fputc('a',fp);//writing single character into file
6.  fclose(fp);//closing file
7. }
```

**file1.txt**

---

## Reading File : `fgetc()` function

The `fgetc()` function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

**Syntax:**

```
1. int fgetc(FILE *stream)
```

**Example:**

```
1. #include<stdio.h>
2. #include<conio.h>
3. void main(){
4.  FILE *fp;
5.  char c;
6.  clrscr();
7.  fp=fopen("myfile.txt","r");
8.
9.  while((c=fgetc(fp))!=EOF){
10. printf("%c",c);
11. }
12. fclose(fp);
13. getch();
14. }
```

**myfile.txt**this is simple text message



## fseek() function

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

### Syntax:

```
1. int fseek(FILE *stream, long int offset, int whence)
```

There are 3 constants used in the fseek() function for whence: SEEK\_SET, SEEK\_CUR and SEEK\_END.

### Example:

```
1. #include <stdio.h>
2. void main(){
3.   FILE *fp;
4.
5.   fp = fopen("myfile.txt","w+");
6.   fputs("This is javatpoint", fp);
7.
8.   fseek( fp, 7, SEEK_SET );
9.   fputs("sonoo jaiswal", fp);
10.  fclose(fp);
11. }
```

**myfile.txt**This is sonoo jaiswal

## fscanf( )

fscanf function reads formatted input from a file. This function is implemented in file related programs for reading formatted data from any file that is specified in the program

### syntax

```
1 | int fscanf(FILE *stream, const char *format, ...)
```

Its return the number of variables that are assigned values, or EOF if no assignments could be made.

**[A WordPress.com Website.](https://bcastudyguide.com/unit-6-file-handling/)**