

Arrays in C

An array is defined as the collection of similar type of data items stored at contiguous memory locations. 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.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

Properties of Array

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- 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.

Advantages of C Array

- 1) Code Optimization: Less code to 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.

Disadvantages of C Array

- 1) Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaration of C Array

We can declare an array in the C language in the following way.

```
data_type array_name[array_size];
```

Now, let us see the example to declare the array.

```
int marks[5];
```

Here, int is the data_type, marks are the array_name, and 5 is the array_size.

Initialization of C Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

```
marks[0]=80;//initialization of array
```

```
marks[1]=60;
```

```
marks[2]=70;
```

```
marks[3]=85;
```

```
marks[4]=75;
```

Initialization of array in language C array example

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

```
int main(){
```

```
int i=0;
```

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

```
marks[0]=80;//initialization of array
```

```
marks[1]=60;
```

```
marks[2]=70;
```

```
marks[3]=85;
```

```
marks[4]=75;
```

```
//traversal of array
```

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

```
printf("%d \n",marks[i]);

} //end of for loop

return 0;

}
```

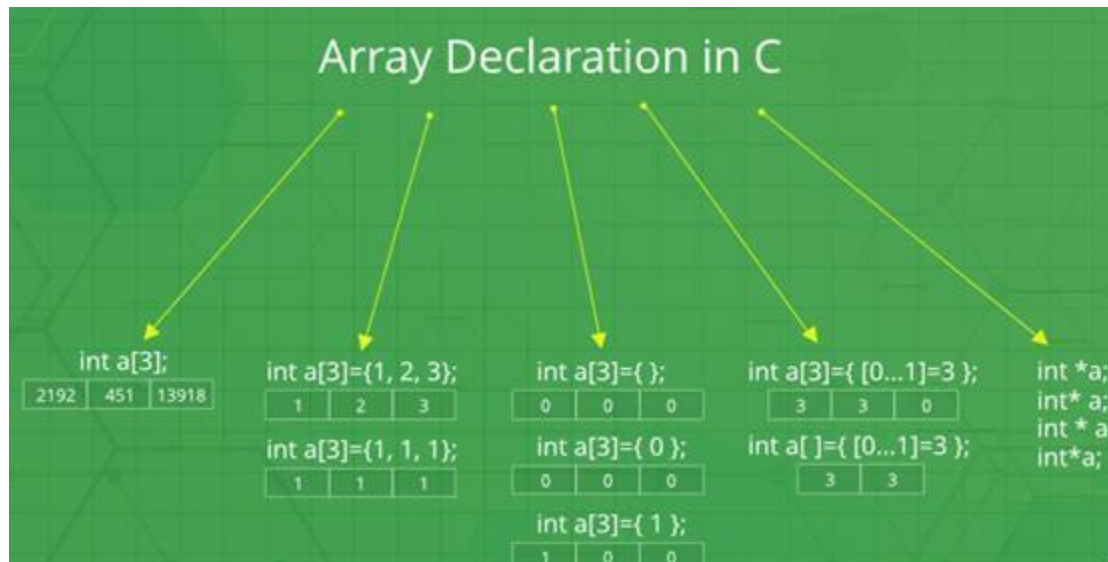
Output

```
80
60
70
85
75
```

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

Array Length = 9
First Index = 0
Last Index = 8



There are various ways in which we can declare an array. It can be done by specifying its type and size, by initializing it or both.

1. Array declaration by specifying size

```
// Array declaration by specifying size  
int arr1[10];
```

```
// With recent C/C++ versions, we can also  
// declare an array of user specified size  
int n = 10;  
int arr2[n];
```

2. Array declaration by initializing elements

```
// Array declaration by initializing elements  
int arr[] = { 10, 20, 30, 40 };
```

```
// Compiler creates an array of size 4.  
// above is same as "int arr[4] = {10, 20, 30, 40};"
```

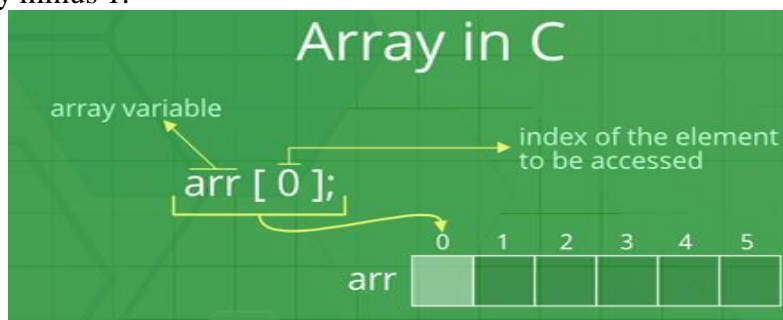
3. Array declaration by specifying size and initializing elements

```
// Array declaration by specifying size and initializing  
// elements  
int arr[6] = { 10, 20, 30, 40 }
```

```
// Compiler creates an array of size 6, initializes first  
// 4 elements as specified by user and rest two elements as 0.  
// above is same as "int arr[] = {10, 20, 30, 40, 0, 0};"
```

Accessing Array Elements

Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.



Example:

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

```

int main()
{
    int arr[5];
    arr[0] = 5;
    arr[1] = -10;
    arr[2] = 2;
    arr[3] = arr[0];

    printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);

    return 0;
}

```

Output:

```
5 -10 2 5
```

The elements are stored at contiguous memory locations

Example:

```

// C program to demonstrate that array elements are stored
// contiguous locations

#include <stdio.h>
int main()
{
    // an array of 10 integers. If arr[0] is stored at
    // address x, then arr[1] is stored at x + sizeof(int)
    // arr[2] is stored at x + sizeof(int) + sizeof(int)
    // and so on.
    int arr[5], i;

    printf("Size of integer in this compiler is %lu\n", sizeof(int));

    for (i = 0; i < 5; i++)
        // The use of '&' before a variable name, yields
        // address of variable.
        printf("Address arr[%d] is %p\n", i, &arr[i]);

    return 0;
}

```

Output:

```

Size of integer in this compiler is 4
Address arr[0] is 0x7ffd636b4260
Address arr[1] is 0x7ffd636b4264
Address arr[2] is 0x7ffd636b4268

```

Address arr[3] is 0x7ffd636b426c

Address arr[4] is 0x7ffd636b4270

Some Programs on Arrays

C program to declare and initialize the array.

```
#include<stdio.h>
int main(){
    int i=0;

    int marks[5]={20,30,40,50,60}; //declaration and initialization of array
    //traversal of array
    for(i=0;i<5;i++){
        printf("%d \n",marks[i]);
    }
    return 0;
}
```

Output

```
20
30
40
50
60
```

C Programs for Sorting an array

```
#include<stdio.h>
void main ()
{
    int i, j, temp;

    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] > a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

```

        }}
    }
    printf("Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n",a[i]);
    }
}

```

C Program to print the largest and second largest element of the array.

```

#include<stdio.h>
void main ()
{
    int arr[100],i,n,largest,sec_largest;
    printf("Enter the size of the array?");
    scanf("%d",&n);
    printf("Enter the elements of the array?");
    for(i = 0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0;i<n;i++)
    {
        if(arr[i]>largest)
        {
            sec_largest = largest;
            largest = arr[i];
        }
        else if (arr[i]>sec_largest && arr[i]!=largest)
        {
            sec_largest=arr[i];
        }
    }
    printf("largest = %d, second largest = %d",largest,sec_largest);
}

```

Program to take 5 values from the user and store them in an array
 // Print the elements stored in the array

```
#include <stdio.h>

int main() {
    int values[5];

    printf("Enter 5 integers: ");

    // taking input and storing it in an array
    for(int i = 0; i < 5; ++i) {
        scanf("%d", &values[i]);
    }

    printf("Displaying integers: ");

    // printing elements of an array
    for(int i = 0; i < 5; ++i) {
        printf("%d\n", values[i]);
    }
    return 0;
}
```

Output

```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```

// Program to find the average of n numbers using arrays

```
#include <stdio.h>
int main()
{
    int marks[10], i, n, sum = 0, average;

    printf("Enter number of elements: ");
```



```

scanf("%d", &n);

for(i=0; i<n; ++i)
{
    printf("Enter number%d: ",i+1);
    scanf("%d", &marks[i]);

    // adding integers entered by the user to the sum variable
    sum += marks[i];
}

average = sum/n;
printf("Average = %d", average);

return 0;
}

```

Output

```

Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39

```

There are seven numbers stored in an array. The following program prints all the numbers of that array as well as prints the numbers in backward.

```

#include<stdio.h>

int main()
{
    int M[10];
    int j;

    /* store seven numbers in array M */

    M[0] = 2;
    M[1] = 4;

```

```

M[2] = 6;

M[3] = 8;

M[4] = 10;

M[5] = 12;

M[6] = 14;

/* print numbers in M */

printf("Print all the Numbers : \n");

for (j = 0; j < 7; ++j)

printf("M[%d] = %d\n",j,M[j]);

/* print numbers in M backwards */

printf("\nFrom End to Beginning : \n");

for (j = 6; j >= 0; --j)

{

printf("M[%d] = %d\n",j,M[j]);

}

}

```

Output:

```

C:\shared\PRINTARRAY1.exe
Print all the Numbers :
M[0] = 2
M[1] = 4
M[2] = 6
M[3] = 8
M[4] = 10
M[5] = 12
M[6] = 14

From End to Beginning :
M[6] = 14
M[5] = 12
M[4] = 10
M[3] = 8
M[2] = 6
M[1] = 4
M[0] = 2

```

Copying arrays:

We have two arrays list1 and list2

```
int list1[6] = {2, 4, 6, 8, 10, 12};
```

```
int list2[6];
```

and we want to copy the contents of list1 to list2. For general variables (e.g. int x=3, y=5) we use simple assignment statement (x=y or y=x). But for arrays the following statement is wrong.

```
list2 = list1;
```

We must copy between arrays element by element and the two arrays must have the same size. In the following example, we use a for loop which makes this easy.

```
#include <stdio.h>

main()

int list1[6] = {2, 4, 6, 8, 10, 12};

int list2[6];

for (int ctr = 0; ctr<6; ctr++)

{

    list2[ctr] = list1[ctr];

}

printf("Elements of list2 :\n");

for (int ctr = 0; ctr<6; ctr++)

{

    printf("%d ",list2[ctr]);

}

}
```

Copy

Output:

```
Elements of list2 :
2 4 6 8 10 12
```

C Multidimensional Arrays

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

```
float x[3][4];
```

Here, `x` is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array

	Column 1	Column 2	Column 3	Column 4
Row 1	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>	<code>x[0][3]</code>
Row 2	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>	<code>x[1][3]</code>
Row 3	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>	<code>x[2][3]</code>

Similarly, you can declare a three-dimensional (3d) array. For example,

```
float y[2][4][3];
```

Here, the array `y` can hold 24 elements.

Initializing a multidimensional array

Initialization of a 2d array

```
// Different ways to initialize two-dimensional array
```

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
```

```
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
```

```
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Initialization of a 3d array

You can initialize a three-dimensional array in a similar way like a two-dimensional array. Here's an example,

```
int test[2][3][4] = {  
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},  
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

Example 1: Two-dimensional array to store and print values

```
// C program to store temperature of two cities of a week and display it.
```

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

```
const int CITY = 2;
```

```
const int WEEK = 7;
```

```
int main()
```

```
{
```

```
    int temperature[CITY][WEEK];
```

```
    // Using nested loop to store values in a 2d array
```

```
    for (int i = 0; i < CITY; ++i)
```

```
    {
```

```
        for (int j = 0; j < WEEK; ++j)
```

```
        {
```

```
            printf("City %d, Day %d: ", i + 1, j + 1);
```

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

```
        }
```

```
    }
```

```
    printf("\nDisplaying values: \n\n");
```

```
    // Using nested loop to display vlues of a 2d array
```

```
    for (int i = 0; i < CITY; ++i)
```

```
    {
```

```
        for (int j = 0; j < WEEK; ++j)
```

```
        {
```

```
        printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
}
return 0;
}
```

Output

City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
City 2, Day 7 = 26

Example 2: Sum of two matrices

// C program to find the sum of two matrices of order 2*2

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

```
int main()
```

```
{
```

```
    float a[2][2], b[2][2], result[2][2];
```

```
    // Taking input using nested for loop
```

```
    printf("Enter elements of 1st matrix\n");
```

```
    for (int i = 0; i < 2; ++i)
```

```
        for (int j = 0; j < 2; ++j)
```

```
        {
```

```
            printf("Enter a%d%d: ", i + 1, j + 1);
```

```
            scanf("%f", &a[i][j]);
```

```
        }
```

```
    // Taking input using nested for loop
```

```
    printf("Enter elements of 2nd matrix\n");
```

```
    for (int i = 0; i < 2; ++i)
```

```
        for (int j = 0; j < 2; ++j)
```

```
        {
```

```
            printf("Enter b%d%d: ", i + 1, j + 1);
```

```
            scanf("%f", &b[i][j]);
```

```
        }
```

```
    // adding corresponding elements of two arrays
```

```
    for (int i = 0; i < 2; ++i)
```

```
        for (int j = 0; j < 2; ++j)
```

```
        {
```

```
            result[i][j] = a[i][j] + b[i][j];
```

```
        }
```

```
    // Displaying the sum
```

```
    printf("\nSum Of Matrix:");
```

```
    for (int i = 0; i < 2; ++i)
```

```
        for (int j = 0; j < 2; ++j)
```

```
        {
```

```
            printf("%.1f\t", result[i][j]);
```

```
            if (j == 1)
```

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

```
    }  
    return 0;  
}
```

Output

```
Enter elements of 1st matrix  
Enter a11: 2;  
Enter a12: 0.5;  
Enter a21: -1.1;  
Enter a22: 2;  
Enter elements of 2nd matrix  
Enter b11: 0.2;  
Enter b12: 0;  
Enter b21: 0.23;  
Enter b22: 23;
```

```
Sum Of Matrix:  
2.2   0.5  
-0.9  25.0
```

Example 3: Three-dimensional array

// C Program to store and print 12 values entered by the user

```
#include <stdio.h>  
int main()  
{  
    int test[2][3][2];  
  
    printf("Enter 12 values: \n");  
  
    for (int i = 0; i < 2; ++i)  
    {  
        for (int j = 0; j < 3; ++j)  
        {  
            for (int k = 0; k < 2; ++k)  
            {  
                scanf("%d", &test[i][j][k]);  
            }  
        }  
    }  
}
```



```
// Printing values with proper index.

printf("\nDisplaying values:\n");
for (int i = 0; i < 2; ++i)
{
    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 2; ++k)
        {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
        }
    }
}

return 0;
}
```

Output

Enter 12 values:

1
2
3
4
5
6
7
8
9
10
11
12

Displaying Values:

test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10

```
test[1][2][0] = 11
test[1][2][1] = 12
```

C program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

```
#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
```

Things that you must consider while initializing a 2D array

We already know, when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration. Let's understand this with the help of few examples –

```
/* Valid declaration*/
int abc[2][2] = { 1, 2, 3 ,4 }
/* Valid declaration*/
int abc[][2] = { 1, 2, 3 ,4 }
/* Invalid declaration – you must specify second dimension*/
int abc[][] = { 1, 2, 3 ,4 }
/* Invalid because of the same reason mentioned above*/
int abc[2][] = { 1, 2, 3 ,4 }
```

How to store user input data into 2D array

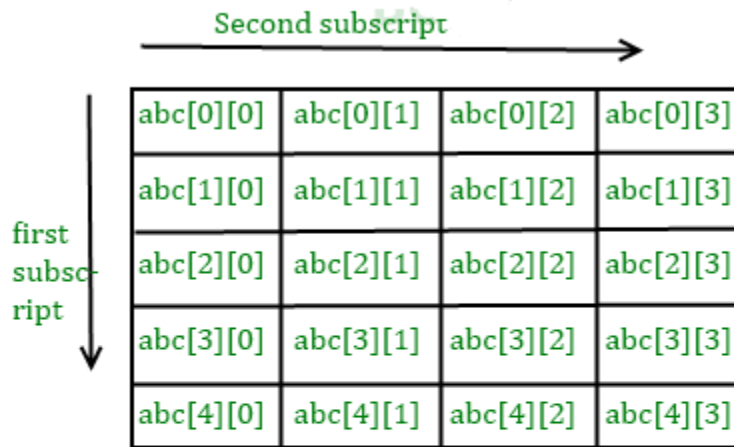
We can calculate how many elements a two dimensional array can have by using this formula: The array `arr[n1][n2]` can have $n1 * n2$ elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has **firstsubscript** value as 5 and **second subscript** value as 4. So the array `abc[5][4]` can have $5 * 4 = 20$ elements.

To store the elements entered by user we are using two for loops, one of them is a nested loop. The outer loop runs from 0 to the (first subscript -1) and the inner for loops runs from 0 to the (second subscript -1). This way the the order in which user enters the elements would be `abc[0][0]`, `abc[0][1]`, `abc[0][2]`...so on.

```
#include<stdio.h>
int main(){
    /* 2D array declaration*/
    int abc[5][4];
    /*Counter variables for the loop*/
    int i, j;
    for(i=0; i<5; i++) {
        for(j=0;j<4;j++) {
            printf("Enter value for abc[%d][%d]:", i, j);
            scanf("%d", &abc[i][j]);
        }
    }
    return 0;
}
```

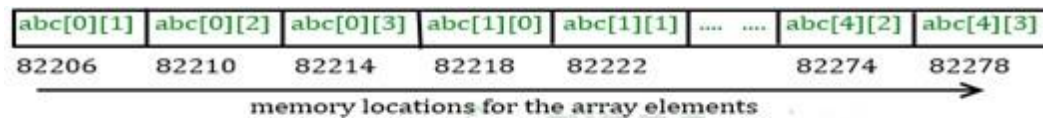
In above example, We have a 2D array `abc` of integer type. Conceptually we can visualize the above array like this:

2D array conceptual memory representation



Here my array is `abc[5][4]`, which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means `abc[0][0]` would be the first element of the array.

However the actual representation of this array in memory would be something like this:



Array is of integer type so each element would use 4 bytes that's the reason there is a difference of 4 in element's addresses.

The addresses are generally represented in hex. This diagram shows them in integer just to show you that the elements are stored in contiguous locations, so that you can understand that the address difference between each element is equal to the size of one element (int size 4). For better understanding see the program below.

Actual memory representation of a 2D array

C Character Array and Strings

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null. The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If We follow the rule of array initialization then we can write the above statement as follows:

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

Following is the memory presentation of above-defined string in C.

H	e	l	l	o	'\0'
---	---	---	---	---	------

String Declaration

String Declaration

1) `char str1[]={'A', 'B', 'C', 'D', '\0'};`

2) `char str1[]="ABCD";`



**'\0' would automatically
insterted at the end in this
type of declaration**

Method 1:

```
char address[]={ 'T', 'E', 'X', 'A', 'S', '\0'};
```

Method 2: The above string can also be defined as –

```
char address[]="TEXAS";
```

In the above declaration NULL character (\0) will automatically be inserted at the end of the string.

What is NULL Char “\0”?

'\0' represents the end of the string. It is also referred as String terminator & Null Character.

String I/O

1) printf and scanf

2) puts and gets

Syntax of above functions - Assume string as str1

printf("%s", str1);

puts(str1); --%s not require here.

scanf("%s", &str1);

gets(str1); --%s not require

BeginnersBook.com

Read & write Strings in C using Printf() and Scanf() functions

```
#include <stdio.h>
#include <string.h>
int main()
{
    /* String Declaration*/
    char nickname[20];

    printf("Enter your Nick name:");

    /* I am reading the input string and storing it in nickname
    * Array name alone works as a base address of array so
    * we can use nickname instead of &nickname here
    */
    scanf("%s", nickname);

    /*Displaying String*/
    printf("%s",nickname);

    return 0;
}
```

Output:

Enter your Nick name:Negan

Negan

Note: %s format specifier is used for strings input/output

Read & Write Strings in C using gets() and puts() functions

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

```

int main()
{
    /* String Declaration*/
    char nickname[20];

    /* Console display using puts */
    puts("Enter your Nick name:");

    /*Input using gets*/
    gets(nickname);

    puts(nickname);

    return 0;
}

```

C – String functions

strlen - Finds out the length of a string

strlwr - It converts a string to lowercase

strupr - It converts a string to uppercase

strcat - It appends one string at the end of another

strncat - It appends first n characters of a string at the end of another.

strcpy - Use it for Copying a string into another

strncpy - It copies first n characters of one string into another

strcmp - It compares two strings

strncmp - It compares first n characters of two strings

strcmpi - It compares two strings without regard to case ("i" denotes that this function ignores case)

stricmp - It compares two strings without regard to case (identical to strcmpi)

strnicmp - It compares first n characters of two strings, Its not case sensitive

strdup - Used for Duplicating a string

strchr - Finds out first occurrence of a given character in a string

strrchr - Finds out last occurrence of a given character in a string

strstr - Finds first occurrence of a given string in another string

strset - It sets all characters of string to a given character

strnset - It sets first n characters of a string to a given character

strrev - It Reverses a string

C String function – strlen

Syntax:

```
size_t strlen(const char *str)
```

size_t represents unsigned short

It returns the length of the string without including end character (**terminating char**

'\0'). **Example of strlen:**

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[20] = "BeginnersBook";
    printf("Length of string str1: %d", strlen(str1));
    return 0;
}
```

Output:

```
Length of string str1: 13
```

strlen vs sizeof

strlen returns you the length of the string stored in array, however sizeof returns the total allocated size assigned to the array. So if I consider the above example again then the following statements would return the below values.

strlen(str1) returned value 13.

sizeof(str1) would return value 20 as the array size is 20 (see the first statement in main function).

C String function – strnlen

Syntax:

```
size_t strnlen(const char *str, size_t maxlen)
```

size_t represents unsigned short

It returns length of the string if it is less than the value specified for maxlen (maximum length) otherwise it returns maxlen value.

Example of strnlen:

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



```

char str1[20] = "BeginnersBook";
printf("Length of string str1 when maxlen is 30: %d", strlen(str1, 30));
printf("Length of string str1 when maxlen is 10: %d", strlen(str1, 10));
return 0;
}

```

Output:

Length of string str1 when maxlen is 30: 13

Length of string str1 when maxlen is 10: 10

Have you noticed the output of second printf statement, even though the string length was 13 it returned only 10 because the maxlen was 10.

C String function – strcmp

```
int strcmp(const char *str1, const char *str2)
```

It compares the two strings and returns an integer value. If both the strings are same (equal) then this function would return 0 otherwise it may return a negative or positive value based on the comparison.

If string1 < string2 OR string1 is a substring of string2 then it would result in a negative value. If string1 > string2 then it would return positive value.

If string1 == string2 then you would get 0(zero) when you use this function for compare strings.

Example of strcmp:

```

#include <stdio.h>
#include <string.h>
int main()
{
    char s1[20] = "BeginnersBook";
    char s2[20] = "BeginnersBook.COM";
    if (strcmp(s1, s2) == 0)
    {
        printf("string 1 and string 2 are equal");
    } else
    {
        printf("string 1 and 2 are different");
    }
    return 0;
}

```

Output:

string 1 and 2 are different

C String function – strcmp

```
int strcmp(const char *str1, const char *str2, size_t n)
```

size_t is for unsigned short

It compares both the string till n characters or in other words it compares first n characters of both the strings.

Example of strcmp:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[20] = "BeginnersBook";
    char s2[20] = "BeginnersBook.COM";
    /* below it is comparing first 8 characters of s1 and s2*/
    if (strcmp(s1, s2, 8) == 0)
    {
        printf("string 1 and string 2 are equal");
    } else
    {
        printf("string 1 and 2 are different");
    }
    return 0;
}
```

Output:

```
string1 and string 2 are equal
```

C String function – strcat

```
char *strcat(char *str1, char *str2)
```

It concatenates two strings and returns the concatenated string.

Example of strcat:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[10] = "Hello";
    char s2[10] = "World";
    strcat(s1, s2);
    printf("Output string after concatenation: %s", s1);
}
```

```
    return 0;  
}
```

Output:

Output string after concatenation: HelloWorld

C String function – strncat

```
char *strncat(char *str1, char *str2, int n)
```

It concatenates n characters of str2 to string str1. A terminator char ('\0') will always be appended at the end of the concatenated string.

Example of strncat:

```
#include <stdio.h>  
#include <string.h>  
int main()  
{  
    char s1[10] = "Hello";  
    char s2[10] = "World";  
    strncat(s1,s2, 3);  
    printf("Concatenation using strncat: %s", s1);  
    return 0;  
}
```

Output:

Concatenation using strncat: HelloWor

C String function – strcpy

```
char *strcpy( char *str1, char *str2)
```

It copies the string str2 into string str1, including the end character (terminator char '\0').

Example of strcpy:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[30] = "string 1";
    char s2[30] = "string 2 : I'm gonna copied into s1";
    /* this function has copied s2 into s1*/
    strcpy(s1,s2);
    printf("String s1 is: %s", s1);
    return 0;
```

```
}
```

Output:

```
String s1 is: string 2: I'm gonna copied into s1
```

C String function – strncpy

`char *strncpy(char *str1, char *str2, size_t n)`
size_t is unsigned short and n is a number.

Case1: If length of str2 > n then it just copies first n characters of str2 into str1.

Case2: If length of str2 < n then it copies all the characters of str2 into str1 and appends several terminator chars('\0') to accumulate the length of str1 to make it n.

Example of strncpy:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char first[30] = "string 1";
    char second[30] = "string 2: I'm using strncpy now";
    /* this function has copied first 10 chars of s2 into s1 */
    strncpy(s1, s2, 12);
    printf("String s1 is: %s", s1);
    return 0;
}
```

Output:

```
String s1 is: string 2: I'm
```

C String function – strchr

`char *strchr(char *str, int ch)`

It searches string str for character ch (you may be wondering that in above definition I have given data type of ch as int, don't worry I didn't make any mistake it should be int only. The thing is when we give any character while using strchr then it internally gets converted into integer for better searching.

Example of strchr:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char mystr[30] = "I'm an example of function strchr";
```

```
    printf ("%s", strchr(mystr, 'f'));
    return 0;
}
```

Output:

```
f function strchr
```

C String function – Strchr

```
char *strchr(char *str, int ch)
```

It is similar to the function strchr, the only difference is that it searches the string in reverse order, now you would have understood why we have extra r in strrchr, yes you guessed it correct, it is for reverse only.

Now let's take the same above example:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char mystr[30] = "I'm an example of function strchr";
    printf ("%s", strrchr(mystr, 'f'));
    return 0;
}
```

Output:

```
function strchr
```

Why output is different than strchr? It is because it started searching from the end of the string and found the first 'f' in function instead of 'of'.

C String function – strstr

```
char *strstr(char *str, char *srch_term)
```

It is similar to strchr, except that it searches for string srch_term instead of a single char.

Example of strstr:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char inputstr[70] = "String Function in C at BeginnersBook.COM";
    printf ("Output string is: %s", strstr(inputstr, 'Begi'));
    return 0;
}
```

Output:

Output string is: BeginnersBook.COM

Some C Programs on String

1. Write a program in C to find the length of a string without using library

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

void main()
{
    char str[100]; /* Declares a string of size 100 */
    int l= 0;

    printf("\n\nFind the length of a string :\n");
    printf(".....\n");
    printf("Input the string : ");
    fgets(str, sizeof str, stdin);
    while(str[l]!='\0')
    {
        l++;
    }
    printf("Length of the string is : %d\n\n", l-1);
}
```

Output:

Find the length of a string :

Input the string : w3resource.com
Length of the string is : 15
Flowchart:

2. Write a program in C to separate the individual characters from a string.
#include <stdio.h>

```
#include <stdlib.h>
```

```
void main()
```

```
{
```

```
    char str[100]; /* Declares a string of size 100 */  
    int l= 0;
```

```
        printf("\n\nSeparate the individual characters from a string :\n");
```

```
        printf("\n");printf("Input the string : ");
```

```
        fgets(str, sizeof str, stdin);
```

```
            printf("The characters of the string are : \n");  
while(str[l]!='\0')
```

```
{
```

```
    printf("%c ", str[l]);  
    l++;
```

```
}
```

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

```
}
```

Sample Output:

Separate the individual characters from a string :

Input the string : w3resource.com

The characters of the string are :

w 3 r e s o u r c e . c o m

3. Write a program in C to print individual characters of string in reverse order.

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

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

```
#include <stdlib.h>
```

```
void main()
```

```
{
```

```
    char str[100]; /* Declares a string of size 100 */  
    int l,i;
```



```

printf("\n\nPrint individual characters of string in reverse order :\n");
printf(".....\n");

printf("Input the string : ");
fgets(str, sizeof str, stdin);

    l=strlen(str);

    printf("The characters of the string in reverse are : \n");
    for(i=l;i>=0;i--)

    {
        printf("%c ", str[i]);
    }

printf("\n");
}

```

The program can also be written as below:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void main()
{
    char str[100]; /* Declares a string of size 100 */
    int l=0;

    printf("\n\nPrint individual characters of string in reverse order :\n");
    printf(".....\n");

    printf("Input the string : ");
    fgets(str, sizeof str, stdin);

        l=strlen(str);

        printf("The characters of the string in reverse are : \n");
        for(str[l]='\0';l>=0;l--)

        {
            printf("%c ", str[l]);
        }

printf("\n");
}

```

Sample Output:

Print individual characters of string in reverse order :

Input the string : w3resource.com

The characters of the string in reverse are :

m o c . e c r u o s e r 3 w

4. Write a program in C to count the total number of words in a string.

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

```
#define str_size 100 //Declare the maximum size of the string
```

```
void main()
```

```
{
```

```
    char str[str_size];
```

```
    int i, wrd;
```

```
        printf("\n\nCount the total number of words in a string :\n");
```

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

```
        printf("Input the string : ");
```

```
        fgets(str, sizeof str, stdin);
```

```
    i = 0;
```

```
    wrd = 1;
```

```
    /* loop till end of string */
```

```
    while(str[i]!='\0')
```

```
    {
```

```
        /* check whether the current character is white space or new line or tab character*/
```

```
        if(str[i]==' ' || str[i]=='\n' || str[i]=='\t')
```

```
        {
```

```
            wrd++;
```

```
        }
```

```

        i++;
    }

    printf("Total number of words in the string is : %d\n", wrd-1);
}

```

Sample Output:

Count the total number of words in a string :

```

-----
Input the string : This is w3resource.com
Total number of words in the string is : 3

```

5. Write a program in C to convert a string to lowercase.

```

#include<stdio.h>
#include<ctype.h>

int main()
{
    int ctr=0;
    char str_char;
    char str[100];

    printf("\n Convert a string to lowercase :\n");
    printf(".....");

    printf("\n Input a string in UPPERCASE : ");
    fgets(str, sizeof str, stdin);

    printf(" Here is the above string in lowercase :\n ");
    while (str[ctr])

```

```

    {
        str_char=str[ctr];

        putchar
        (tolower(str_char))
        ;ctr++;
    }
    return 0;
}

```

Sample Output:

Convert a string to lowercase :

Input a string in UPPERCASE : THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.

Here is the above string in lowercase :

the quick brown fox jumps over the lazy dog.

A function is a block of statements that performs a specific task. Let's say you are writing a C program and you need to perform a same task in that program more than once. In such case you have two options:

- a) Use the same set of statements every time you want to perform the task
- b) Create a function to perform that task, and just call it every time you need to perform that task.

Using option (b) is a good practice and a good programmer always uses functions while writing code in C.

Why we need functions in C

Functions are used because of following reasons –

- a) To improve the readability of code.
- b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
- c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.

d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Types of functions

1) Predefined standard library functions

Standard library functions are also known as built-in functions. Functions such as `puts()`, `gets()`, `printf()`, `scanf()` etc are standard library functions. These functions are already defined in header files (files with .h extensions are called header files such as `stdio.h`), so we just call them whenever there is a need to use them.

For example, `printf()` function is defined in `<stdio.h>` header file so in order to use the `printf()` function, we need to include the `<stdio.h>` header file in our program using `#include <stdio.h>`.

2) User Defined functions

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.

Now we will learn how to create user defined functions and how to use them in C Programming

Syntax of a function

```
return_type function_name (argument list)
{
    Set of statements - Block of code
}
```

return_type: Return type can be of any data type such as `int`, `double`, `char`, `void`, `short` etc. Don't worry you will understand these terms better once you go through the examples below.

function_name: It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing its name.

argument list: Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument.

Block of code: Set of C statements, which will be executed whenever a call will be made to the function.

Do you find above terms confusing? – Do not worry I'm not gonna end this guide until you learn all of them :)
Let's take an example – Suppose you want to create a function to add two integer variables.

Let's split the problem so that it would be easy to understand –
Function will add the two numbers so it should have some meaningful name like sum, addition, etc. For example let's take the name addition for this function.

```
return_type addition(argument list)
```

This function addition adds two integer variables, which means I need two integer variable as input, let's provide two integer parameters in the function signature. The function signature would be –

```
return_type addition(int num1, int num2)
```

The result of the sum of two integers would be integer only. Hence function should return an integer value – I got my return type – It would be integer –

```
int addition(int num1, int num2);
```

So you got your function prototype or signature. Now you can implement the logic in C program like this:

How to call a function in C?

Consider the following C program

Example1: Creating a user defined function addition()

```
#include <stdio.h>
int addition(int num1, int num2)
{
    int sum;
    /* Arguments are used here*/
    sum = num1+num2;
```

```

    /* Function return type is integer so we are returning
    * an integer value, the sum of the passed numbers.
    */
    return sum;
}

int main()
{
    int var1, var2;
    printf("Enter number 1: ");
    scanf("%d",&var1);
    printf("Enter number 2: ");
    scanf("%d",&var2);

    /* Calling the function here, the function return type
    * is integer so we need an integer variable to hold the
    * returned value of this function.
    */
    int res = addition(var1, var2);
    printf ("Output: %d", res);

    return 0;
}

```

Output:

```

Enter number 1: 100
Enter number 2: 120
Output: 220

```

Example2: Creating a void user defined function that doesn't return anything

```

#include <stdio.h>
/* function return type is void and it doesn't have parameters*/
void introduction()
{
    printf("Hi\n");
    printf("My name is Chaitanya\n");
    printf("How are you?");
    /* There is no return statement inside this function, since its
    * return type is void
    */
}

int main()
{
    /*calling function*/
    introduction();
    return 0;
}

```

```
}
```

Output:

```
Hi  
My name is Chaitanya  
How are you?
```

Few Points to Note regarding functions in C:

- 1) `main()` in C program is also a function.
- 2) Each C program must have at least one function, which is `main()`.
- 3) There is no limit on number of functions; A C program can have any number of functions.
- 4) A function can call itself and it is known as “Recursion”. I have written a separate guide for it.

C Functions Terminologies that you must remember

return type: Data type of returned value. It can be void also, in such case function doesn't return any value.

Note: for example, if function return type is char, then function should return a value of char type and while calling this function the `main()` function should have a variable of char data type to store the returned value.

Structure would look like –

```
char abc(char ch1, char ch2)
{
    char ch3;
    ...
    return ch3;
}

int main()
{
    ...
    char c1 = abc('a', 'x');
    ...
}
```

More Topics on Functions in C

- 1) Function – Call by value method – In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

2) Function – Call by reference method – Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

Function call by value is the default way of calling a function in C programming. Before we discuss function call by value, let's understand the terminologies that we will use while explaining this:

Actual parameters: The parameters that appear in function calls.

Formal parameters: The parameters that appear in function declarations.

For example:

```
#include <stdio.h>
int sum(int a, int b)
{
    int c=a+b;
    return c;
}

int main(
{
    int var1 =10;
    int var2 = 20;
    int var3 = sum(var1, var2);
    printf("%d", var3);

    return 0;
}
```

In the above example variable a and b are the formal parameters (or formal arguments). Variable var1 and var2 are the actual arguments (or actual parameters). The actual parameters can also be the values. Like sum(10, 20), here 10 and 20 are actual parameters.

In this guide, we will discuss function call by value. If you want to read call by reference method then refer this guide: [function call by reference](#).

Let's get back to the point.

What is Function Call By value?

When we pass the actual parameters while calling a function then this is known as function call by value. In this case the values of actual parameters are copied

to the formal parameters. Thus operations performed on the formal parameters don't reflect in the actual parameters.

Example of Function call by Value

As mentioned above, in the call by value the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters. Lets take an example to understand this:

```
#include <stdio.h>
int increment(int var)
{
    var = var+1;
    return var;
}

int main()
{
    int num1=20;
    int num2 = increment(num1);
    printf("num1 value is: %d", num1);
    printf("\nnum2 value is: %d", num2);

    return 0;
}
```

Output:

```
num1 value is: 20
num2 value is: 21
```

Explanation

We passed the variable num1 while calling the method, but since we are calling the function using call by value method, only the value of num1 is copied to the formal parameter var. Thus change made to the var doesn't reflect in the num1.

Example 2: Swapping numbers using Function Call by Value

```
#include <stdio.h>
void swapnum( int var1, int var2 )
{
    int tempnum ;
    /*Copying var1 value into temporary variable */
    tempnum = var1 ;
```

```

/* Copying var2 value into var1*/
var1 = var2 ;

/*Copying temporary variable value into var2 */
var2 = tempnum ;

}
int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);

    /*calling swap function*/
    swapnum(num1, num2);
    printf("\nAfter swapping: %d, %d", num1, num2);
}

```

Output:

```

Before swapping: 35, 45
After swapping: 35, 45

```

Why variables remain unchanged even after the swap?

The reason is same – function is called by value for num1 & num2. So actually var1 and var2 gets swapped (not num1 & num2). As in call by value actual parameters are just copied into the formal parameters.

Just like variables, array can also be passed to a function as an argument . In this guide, we will learn how to pass the array to a function using call by value and call by reference methods.

To understand this guide, you should have the knowledge of following C Programming topics:

1. C – Array
2. Function call by value in C
3. Function call by reference in C

Passing array to function using call by value method

As we already know in this type of function call, the actual parameter is copied to the formal parameters.

```

#include <stdio.h>
void disp( char ch)

```

```

{
    printf("%c ", ch);
}
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
    for (int x=0; x<10; x++)
    {
        /* I'm passing each element one by one using subscript*/
        disp (arr[x]);
    }

    return 0;
}

```

Output:

```
a b c d e f g h i j
```

Passing array to function using call by reference

When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

```

#include <stdio.h>
void disp( int *num)
{
    printf("%d ", *num);
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    for (int i=0; i<10; i++)
    {
        /* Passing addresses of array elements*/
        disp (&arr[i]);
    }

    return 0;
}

```

Output:

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

How to pass an entire array to a function as an argument?

In the above example, we have passed the address of each array element one by one using a for loop in C. However you can also pass an entire array to a function like this:

Note: The array name itself is the address of first element of that array. For example if array name is arr then you can say that arr is equivalent to the &arr[0].

```
#include <stdio.h>
void myfuncn( int *var1, int var2)
{
    /* The pointer var1 is pointing to the first element of
     * the array and the var2 is the size of the array. In the
     * loop we are incrementing pointer so that it points to
     * the next element of the array on each increment.
     */
    for(int x=0; x<var2; x++)
    {
        printf("Value of var_arr[%d] is: %d \n", x, *var1);
        /*increment pointer for next element fetch*/
        var1++;
    }
}

int main()
{
    int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
    myfuncn(var_arr, 7);
    return 0;
}
```

Output:

```
Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77
```

Structure

Structure is a group of variables of different data types represented by a single name. Let's take an example to understand the need of a structure in C programming.

Why we need structure in C ?

Let's say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student. This may sound confusing, do not worry we will understand this with the help of example.

How to create a structure in C Programming

We use struct keyword to create a structure in C. The struct keyword is a short form of structured data type.

```
struct struct_name {  
    DataType member1_name;  
    DataType member2_name;  
    DataType member3_name;  
    ...  
};
```

Here `struct_name` is the name of the structure, this is chosen by the programmer and can be anything. However, always choose meaningful, short and easy to read names and avoid using keywords and reserved words as structure names as those are not allowed. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

First we will see the syntax of creating struct variable, accessing struct members etc and then we will see a complete example.

How to declare variable of a structure?

```
struct struct_name var_name;
```

or

```
struct struct_name {  
    DataType member1_name;
```

```
    DataType member2_name;  
    DataType member3_name;  
    ...  
} var_name;
```

How to access data members of a structure using a struct variable?

```
var_name.member1_name;  
var_name.member2_name;  
...
```

How to assign values to structure members?

There are three ways to do this.

1) Using Dot(.) operator

```
var_name.memeber_name = value;
```

2) All members assigned in one statement

```
struct struct_name var_name =  
{value for memeber1, value for memeber2 ...so on for all the members}
```

3) Designated initializers – We will discuss this later at the end of this post.

Example of Structure in C

In this example, we have created a structure `StudentData` with three data members `stu_name`, `stu_id` and `stu_age`. In this program, we are storing the student name, id and age into the structure and accessing structure data members to display these values as an output.

```
#include <stdio.h>  
/* Created a structure here. The name of the structure is  
 * StudentData.  
 */  
struct StudentData{  
    char *stu_name;  
    int stu_id;  
    int stu_age;  
};  
int main()  
{  
    /* student is the variable of structure StudentData*/  
    struct StudentData student;  
  
    /*Assigning the values of each struct member here*/  
}
```

```

    student.stu_name = "Steve";
    student.stu_id = 1234;
    student.stu_age = 30;

    /* Displaying the values of struct members */
    printf("Student Name is: %s", student.stu_name);
    printf("\nStudent Id is: %d", student.stu_id);
    printf("\nStudent Age is: %d", student.stu_age);
    return 0;
}

```

Output:

```

Student Name is: Steve
Student Id is: 1234
Student Age is: 30

```

Nested Structure in C: Struct inside another struct

You can use a structure inside another structure, this is called nesting of structures. As I explained above, once a structure is declared, the struct `stu_name` acts as a new data type so you can include it in another struct just like the data type of other data members. Sounds confusing? Don't worry. The following example will clear your doubt.

Example of Nested Structure in C Programming

Let's say we have two structure like this: The second structure `stu_data` has `stu_address` as a data member. Here, `stu_data` is called outer structure or parent structure and `stu_address` is called inner structure or child structure.

Structure 1: `stu_address`

```

struct stu_address
{
    int street;
    char *state;
    char *city;
    char *country;
};

```

Structure 2: `stu_data`

```

struct stu_data
{
    int stu_id;
    int stu_age;
}

```



```
    char *stu_name;
    struct stu_address stuAddress;
};
```

As you can see here that I have nested a structure inside another structure.

Assignment for struct inside struct (Nested struct)

Let's take the example of the two structure that we seen above to understand the logic. We are using the . (dot) operator twice to access the data member of inner structure.

```
struct stu_data mydata;
mydata.stu_id = 1001;
mydata.stu_age = 30;
mydata.stuAddress.state = "UP"; //Nested struct assignment
..
```

Similarly other data members of inner structure can be accessed like this:

```
mydata.stuAddress.city = "Delhi";
mydata.stuAddress.country = "India";
```

How to access nested structure members?

Using chain of "." operator. Let's say you want to display the city alone from nested struct:

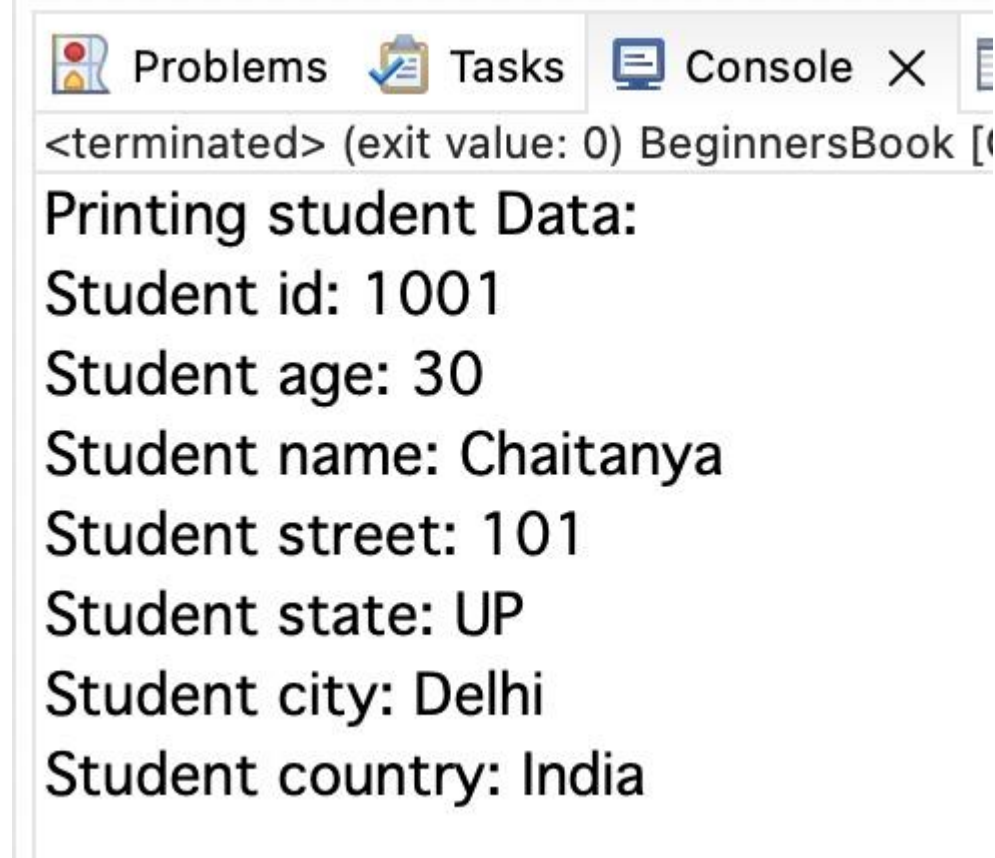
```
printf("%s", mydata.stuAddress.city);
```

Let's see the complete program:

```
#include <stdio.h>
struct stu_address
{
    int street;
    char *state;
    char *city;
    char *country;
};
struct stu_data
{
    int stu_id;
    int stu_age;
    char *stu_name;
    struct stu_address stuAddress;
};
int main(){
```

```
struct stu_data mydata;  
mydata.stu_id = 1001;  
mydata.stu_age = 30;  
mydata.stu_name = "Chaitanya";  
mydata.stuAddress.state = "UP";  
mydata.stuAddress.street = 101;  
mydata.stuAddress.city = "Delhi";  
mydata.stuAddress.country = "India";  
printf("Printing student Data: ");  
printf("\nStudent id: %d",mydata.stu_id);  
printf("\nStudent age: %d",mydata.stu_age);  
printf("\nStudent name: %s",mydata.stu_name);  
printf("\nStudent street: %d",mydata.stuAddress.street);  
printf("\nStudent state: %s",mydata.stuAddress.state);  
printf("\nStudent city: %s",mydata.stuAddress.city);  
printf("\nStudent country: %s",mydata.stuAddress.country);  
  
return 0;  
}
```

Output:

A screenshot of a code editor's console window. The window has a title bar with icons for 'Problems', 'Tasks', and 'Console', followed by a close button 'X'. The console text shows the program's output: '<terminated> (exit value: 0) BeginnersBook [0', followed by a blank line, then 'Printing student Data:', and then seven lines of student data: 'Student id: 1001', 'Student age: 30', 'Student name: Chaitanya', 'Student street: 101', 'Student state: UP', 'Student city: Delhi', and 'Student country: India'.

```
<terminated> (exit value: 0) BeginnersBook [0  
  
Printing student Data:  
Student id: 1001  
Student age: 30  
Student name: Chaitanya  
Student street: 101  
Student state: UP  
Student city: Delhi  
Student country: India
```

Use of typedef in Structure

typedef makes the code short and improves readability. In the above discussion we have seen that while using structs every time we have to use the lengthy syntax, which makes the code confusing, lengthy, complex and less readable. The simple solution to this issue is use of typedef. It is like an alias of struct.

Code without typedef

```
struct home_address {  
    int local_street;  
    char *town;  
    char *my_city;  
    char *my_country;  
};  
...  
struct home_address var;  
var.town = "Agra";
```

Code using typedef

```
typedef struct home_address{  
    int local_street;  
    char *town;  
    char *my_city;  
    char *my_country;  
}addr;  
..  
..  
addr var1;  
var.town = "Agra";
```

Instead of using the struct home_address every time you need to declare struct variable, you can simply use addr, the typedef that we have defined. You can read the [typedef in detail here](#).

Array of Structures in C

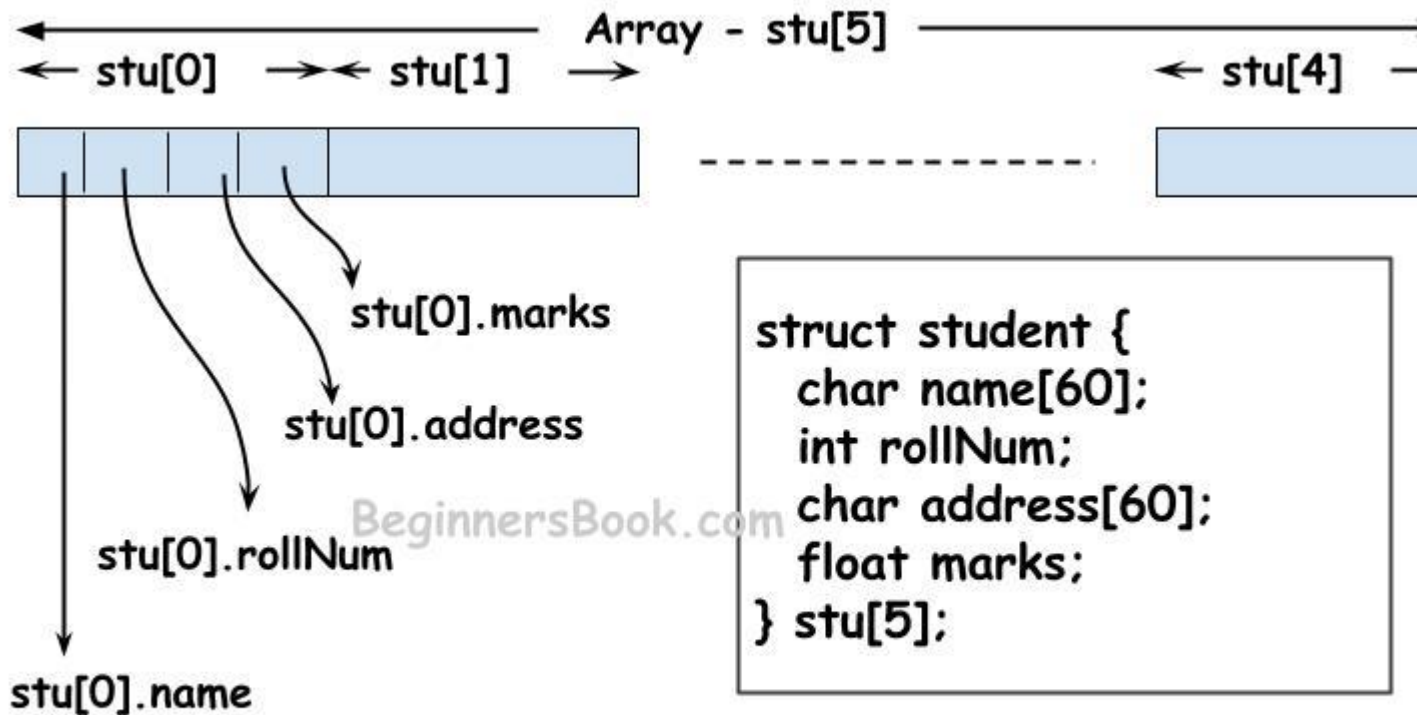
An array of structures is an array with structure as elements.

For example:

Here, stu[5] is an array of structures. This array has 5 elements and these elements are structures of the same type "student". The element s[0] will store the values such as name, rollNum, address & marks of a student, similarly element s[1] will store these details for another student and so on.

```
struct student {  
    char name[60];
```

```
int rollNum;
char address[60];
float marks;
} stu[5];
```



To learn more about structures in C with complete example: [Refer this guide](#).

Designated initializers to set values of Structure members

We have already learned two ways to set the values of a struct member, there is another way to do the same using designated initializers. This is useful when we are doing assignment of only few members of the structure. In the following example the structure variable s2 has only one member assignment.

```
#include <stdio.h>
struct numbers
{
    int num1, num2;
};
int main()
{
```

```
// Assignment using using designated initialization
struct numbers s1 = {.num2 = 22, .num1 = 11};
struct numbers s2 = {.num2 = 30};

printf ("num1: %d, num2: %d\n", s1.num1, s1.num2);
printf ("num1: %d", s2.num2);
return 0;
}
```

Output:

```
num1: 11, num2: 22
num1: 30
```

Pointers

A pointer is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of a integer variable. In this guide, we will discuss pointers in [C programming](#) with the help of examples.

Before we discuss about pointers in C, lets take a simple example to understand what do we mean by the address of a variable.

A simple example to understand how to access the address of a variable without pointers?

In this program, we have a variable num of int type. The value of num is 10 and this value must be stored somewhere in the memory, right? A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. For example we live in a house and our house has an address, which helps other people to find our house. The same way the value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

So let's say the address assigned to variable num is `0x7fff5694dc58`, which means whatever value we would be assigning to num should be stored at the location: `0x7fff5694dc58`. See the diagram below.

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

```

{
    int num = 10;
    printf("Value of variable num is: %d", num);
    /* To print the address of a variable we use %p
     * format specifier and ampersand (&) sign just
     * before the variable name like &num.
     */
    printf("\nAddress of variable num is: %p", &num);
    return 0;
}

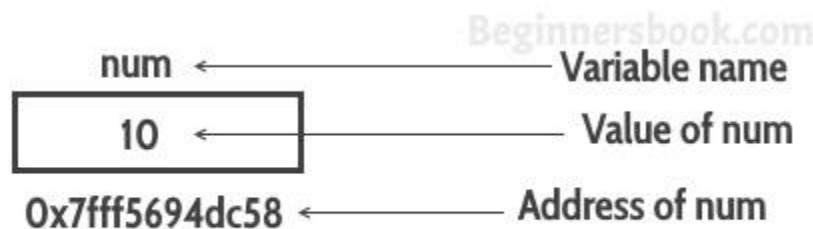
```

Output:

```

Value of variable num is: 10
Address of variable num is: 0x7fff5694dc58

```



A Simple Example of Pointers in C

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we have discussed them later in this guide. For now, we just need to know how to link a pointer to the address of a variable.

Important point to note is: The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable. In the example below, the pointer and the variable both are of int type.

```

#include <stdio.h>
int main()
{
    //Variable declaration
    int num = 10;

    //Pointer declaration
    int *p;

    //Assigning address of num to the pointer p

```

```
p = #  
  
printf("Address of variable num is: %p", p);  
return 0;  
}
```

Output:

```
Address of variable num is: 0x7fff5694dc58
```

C Pointers – Operators that are used with Pointers

Lets discuss the operators & and * that are used with Pointers in C.

“Address of”(&) Operator

We have already seen in the first example that we can display the address of a variable using ampersand sign. I have used &num to access the address of variable num. The & operator is also known as “Address of” Operator.

```
printf("Address of var is: %p", &num);
```

Point to note: %p is a format specifier which is used for displaying the address in hex format.

Now that you know how to get the address of a variable but how to store that address in some other variable? That's where pointers comes into picture. As mentioned in the beginning of this guide, pointers in C programming are used for holding the address of another variables.

Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.

“Value at Address”(*) Operator

The * Operator is also known as Value at address operator.

How to declare a pointer?

```
int *p1 /*Pointer to an integer variable*/  
double *p2 /*Pointer to a variable of data type double*/  
char *p3 /*Pointer to a character variable*/  
float *p4 /*pointer to a float variable*/
```

The above are the few examples of pointer declarations. If you need a pointer to store the address of integer variable then the data type of the pointer should be int. Same case is with the other data types.

By using * operator we can access the value of a variable through a pointer. For example:

```
double a = 10;  
double *p;  
p = &a;
```

*p would give us the value of the variable a. The following statement would display 10 as output.

```
printf("%d", *p);
```

Similarly if we assign a value to *pointer like this:

```
*p = 200;
```

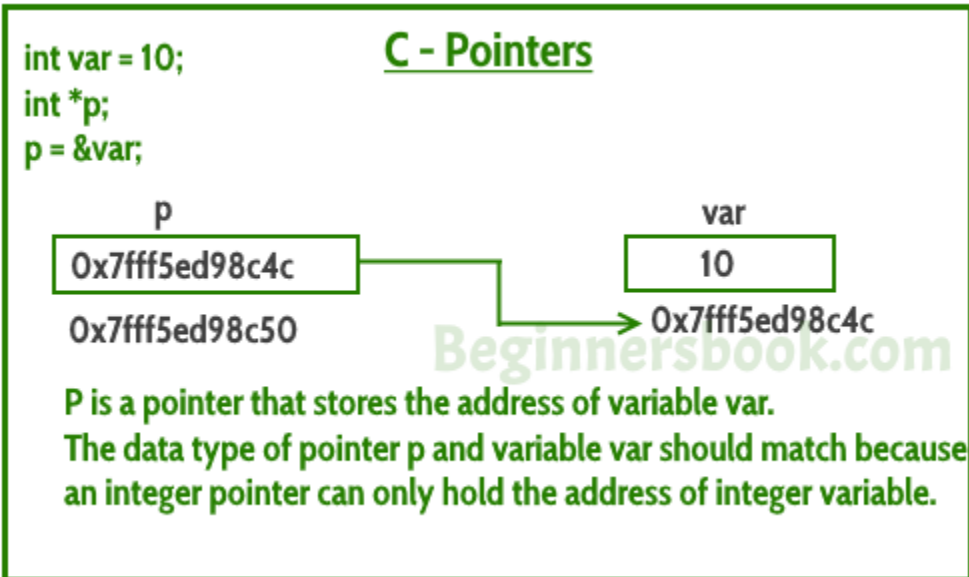
It would change the value of variable a. The statement above will change the value of a from 10 to 200.

Example of Pointer demonstrating the use of & and *

```
#include <stdio.h>  
int main()  
{  
    /* Pointer of integer type, this can hold the  
     * address of a integer type variable.  
     */  
    int *p;  
  
    int var = 10;  
  
    /* Assigning the address of variable var to the pointer  
     * p. The p can hold the address of var because var is  
     * an integer type variable.  
     */  
    p = &var;  
  
    printf("Value of variable var is: %d", var);  
    printf("\nValue of variable var is: %d", *p);  
    printf("\nAddress of variable var is: %p", &var);  
    printf("\nAddress of variable var is: %p", p);  
    printf("\nAddress of pointer p is: %p", &p);  
    return 0;  
}
```


Output:

```
Value of variable var is: 10
Value of variable var is: 10
Address of variable var is: 0x7fff5ed98c4c
Address of variable var is: 0x7fff5ed98c4c
Address of pointer p is: 0x7fff5ed98c50
```



Lets take few more examples to understand it better –

Lets say we have a char variable ch and a pointer ptr that holds the address of ch.

```
char ch='a';
char *ptr;
```

Read the value of ch

```
printf("Value of ch: %c", ch);
or
printf("Value of ch: %c", *ptr);
```

Change the value of ch

```
ch = 'b';
or
*ptr = 'b';
```

The above code would replace the value 'a' with 'b'.

Can you guess the output of following C program?

```

#include <stdio.h>
int main()
{
    int var =10;
    int *p;
    p= &var;

    printf ( "Address of var is: %p", &var);
    printf ( "\nAddress of var is: %p", p);

    printf ( "\nValue of var is: %d", var);
    printf ( "\nValue of var is: %d", *p);
    printf ( "\nValue of var is: %d", *( &var));

    /* Note I have used %p for p's value as it represents an address*/
    printf( "\nValue of pointer p is: %p", p);
    printf ( "\nAddress of pointer p is: %p", &p);

    return 0;
}

```

Output:

```

Address of var is: 0x7fff5d027c58
Address of var is: 0x7fff5d027c58
Value of var is: 10
Value of var is: 10
Value of var is: 10
Value of pointer p is: 0x7fff5d027c58
Address of pointer p is: 0x7fff5d027c50

```

More Topics on Pointers

1) Pointer to Pointer – A pointer can point to another pointer (which means it can store the address of another pointer), such pointers are known as double pointer OR pointer to pointer.

2) Passing pointers to function – Pointers can also be passed as an argument to a function, using this feature a function can be called by reference as well as an array can be passed to a function while calling.

3) Function pointers – A function pointer is just like another pointer, it is used for storing the address of a function. Function pointer can also be used for calling a function in C program.

Pointer to Pointer

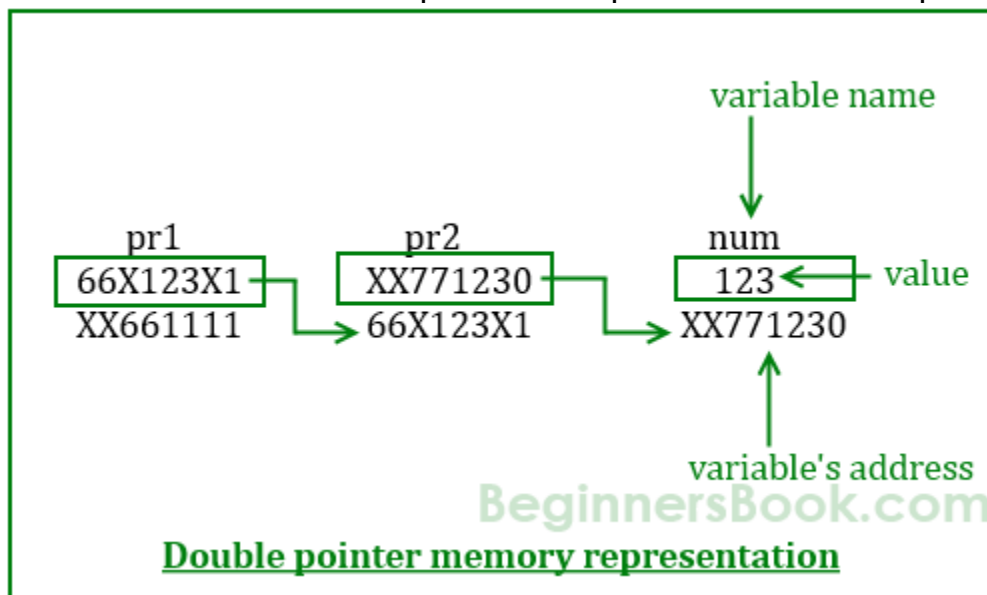
We already know that a pointer holds the address of another variable of same type. When a pointer holds the address of another pointer then such type of pointer is known as pointer-to-pointer or double pointer. In this guide, we will learn what is a double pointer, how to declare them and how to use them in C programming. To understand this concept, you should know the basics of pointers.

How to declare a Pointer to Pointer (Double Pointer) in C?

```
int **pr;
```

Here pr is a double pointer. There must be two *'s in the declaration of double pointer.

Let's understand the concept of double pointers with the help of a diagram:



As per the diagram, pr2 is a normal pointer that holds the address of an integer variable num. There is another pointer pr1 in the diagram that holds the address of another pointer pr2, the pointer pr1 here is a pointer-to-pointer (or double pointer).

Values from above diagram:

```
Variable num has address: XX771230
Address of Pointer pr1 is: XX661111
```

Address of Pointer pr2 is: 66X123X1

Example of double Pointer

Lets write a C program based on the diagram that we have seen above.

```
#include <stdio.h>
int main()
{
    int num=123;

    //A normal pointer pr2
    int *pr2;

    //This pointer pr2 is a double pointer
    int **pr1;

    /* Assigning the address of variable num to the
     * pointer pr2
     */
    pr2 = #

    /* Assigning the address of pointer pr2 to the
     * pointer-to-pointer pr1
     */
    pr1 = &pr2;

    /* Possible ways to find value of variable num*/
    printf("\n Value of num is: %d", num);
    printf("\n Value of num using pr2 is: %d", *pr2);
    printf("\n Value of num using pr1 is: %d", **pr1);

    /*Possible ways to find address of num*/
    printf("\n Address of num is: %p", &num);
    printf("\n Address of num using pr2 is: %p", pr2);
    printf("\n Address of num using pr1 is: %p", *pr1);

    /*Find value of pointer*/
    printf("\n Value of Pointer pr2 is: %p", pr2);
    printf("\n Value of Pointer pr2 using pr1 is: %p", *pr1);

    /*Ways to find address of pointer*/
    printf("\n Address of Pointer pr2 is:%p",&pr2);
    printf("\n Address of Pointer pr2 using pr1 is:%p",pr1);

    /*Double pointer value and address*/
    printf("\n Value of Pointer pr1 is:%p",pr1);
    printf("\n Address of Pointer pr1 is:%p",&pr1);

    return 0;
}
```

Output:

```
Value of num is: 123
Value of num using pr2 is: 123
Value of num using pr1 is: 123
Address of num is: XX771230
Address of num using pr2 is: XX771230
Address of num using pr1 is: XX771230
Value of Pointer pr2 is: XX771230
Value of Pointer pr2 using pr1 is: XX771230
Address of Pointer pr2 is: 66X123X1
Address of Pointer pr2 using pr1 is: 66X123X1
Value of Pointer pr1 is: 66X123X1
Address of Pointer pr1 is: XX661111
```

There are some confusions regarding the output of this program, when you run this program you would see the address similar to this: 0x7fff54da7c58. The reason I have given the address in different format is because I want you to relate this program with the diagram above. I have used the exact address values in the above diagram so that it would be easy for you to relate the output of this program with the above diagram.

You can also understand the program logic with these simple equations:

```
num == *pr2 == **pr1
&num == pr2 == *pr1
&pr2 == pr1
```

Pointer to Array

Pointer to Array

Use a pointer to an array, and then use that pointer to access the array elements. For example,

```
#include<stdio.h>

void main()
```

```
{  
  
    int a[3] = {1, 2, 3};  
  
    int *p = a;  
  
    for (int i = 0; i < 3; i++)  
  
    {  
  
        printf("%d", *p);  
  
        p++;  
  
    }  
  
    return 0;  
}
```

Copy

1 2 3

Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); → **prints the array, by incrementing index**

printf("%d", i[a]); → **this will also print elements of array**

printf("%d", a+i); → **This will print address of all the array elements**

printf("%d", *(a+i)); → **Will print value of array element.**

printf("%d", *a); → **will print value of a[0] only**

a++; → **Compile time error, we cannot change base address of the array.**

Syntax:

```
*(a+i) //pointer with an array
```

Copy

is same as:

```
a[i]
```

Copy

Pointer to Multidimensional Array

Let's see how to make a pointer point to a multidimensional array.

In `a[i][j]`, `a` will give the base address of this array, even `a + 0 + 0` will also give the base address, that is the address of `a[0][0]` element.

Syntax:

```
*(*(a + i) + j)
```

Copy

Pointer and Character strings

Pointer is used to create strings. Pointer variables of `char` type are treated as string.

```
char *str = "Hello";
```

Copy

The above code creates a string and stores its address in the pointer variable `str`. The pointer `str` now points to the first character of the string "Hello".

- The string created using `char` pointer can be assigned a value at runtime.

```
char *str;  
  
str = "hello";
```

Copy

- The content of the string can be printed using `printf()` and `puts()`.

```
printf("%s", str);  
  
puts(str);
```


Copy

- `str` is a pointer to the string and also name of the string. Therefore we do not need to use indirection operator `*`.

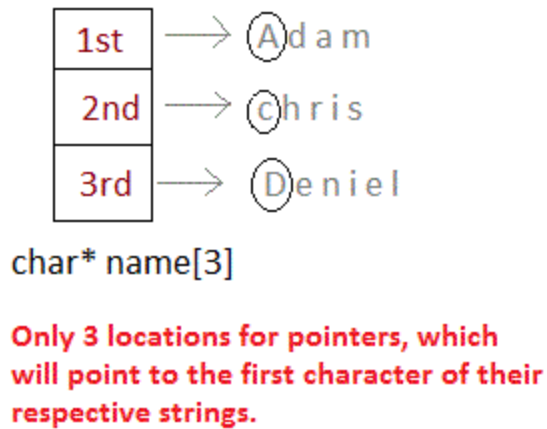
Array of Pointers

Pointers are very helpful in handling character arrays with rows of varying lengths.

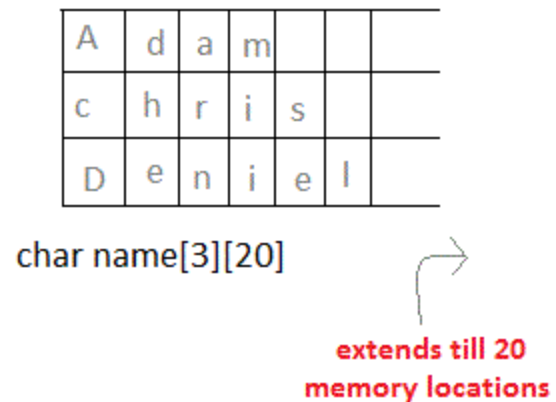
```
char *name[3] = {  
    "Adam",  
    "chris",  
    "Deniel"  
};  
  
//without pointer  
char name[3][20] = {  
    "Adam",  
    "chris",  
    "Deniel"  
};
```

Copy

Using Pointer



Without Pointer



In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

Pointers to Function

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will effect the original variable.

Example Time: Swapping two numbers using Pointer

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

```
void swap(int *a, int *b);

int main()
{
    int m = 10, n = 20;

    printf("m = %d\n", m);

    printf("n = %d\n\n", n);


    swap(&m, &n);    //passing address of m and n
to the swap function

    printf("After Swapping:\n\n");

    printf("m = %d\n", m);

    printf("n = %d", n);

    return 0;
}

/*

pointer 'a' and 'b' holds and
points to the address of 'm' and 'n'
```

```
*/  
  
void swap(int *a, int *b)  
{  
  
    int temp;  
  
    temp = *a;  
  
    *a = *b;  
  
    *b = temp;  
  
}
```

Copy

```
m = 10
```

```
n = 20
```

After Swapping:

```
m = 20
```

```
n = 10
```

Functions returning Pointer variables

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence

if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>

int* larger(int*, int*);

void main()
{
    int a = 15;
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger", *p);
}

int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
```

```
else  
    return y;  
}
```

Copy

```
92 is larger
```

Safe ways to return a valid Pointer.

1. Either use argument with functions. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.
2. Or, use `static` local variables inside the function and return them. As static variables have a lifetime until the `main()` function exits, therefore they will be available throughout the program.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

```
type (*pointer-name) (parameter);
```

Copy

Here is an example :

```
int (*sum)();    //legal declaration of pointer to
function
```

```
int *sum();      //This is not a declaration of
pointer to function
```

Copy

A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);

int (*s)(int, int);

s = sum;
```

Copy

Here `s` is a pointer to a function `sum`. Now `sum` can be called using function pointer `s` along with providing the required argument values.

```
s (10, 20);
```

Copy

Example of Pointer to Function

```
#include <stdio.h>

int sum(int x, int y)
```

```
{  
  
    return x+y;  
  
}  
  
int main( )  
{  
  
    int (*fp)(int, int);  
  
    fp = sum;  
  
    int s = fp(10, 15);  
  
    printf("Sum is %d", s);  
  
    return 0;  
  
}
```

Copy

25

Complicated Function Pointer example

You will find a lot of complex function pointer examples around, lets see one such example and try to understand it.

```
void *(*foo) (int*);
```

Copy

It appears complex but it is very simple. In this case (*foo) is a pointer to the function, whose argument is of int* type and return type is void*.

Introduction to Data Structures

Introduction

We know that in the programming world, data is the center and everything revolves around data. We need to do all data operations including storing, searching, sorting, organizing, and accessing data efficiently and only then our program can succeed.

We need to find the most efficient way of storing data that can help us to build dynamic solutions. Data structure helps us in building such solutions.

A data structure is a special way of organizing and storing data in a computer so that it can be used efficiently.

Array, Linked List, Stack, Queue, Tree, Graph etc are all data structures that stores the data in a special way so that we can access and use the data efficiently.

Each of these mentioned data structures has a different special way of organizing data so we choose the data structure based on the requirement.

The idea is to reduce the space and time complexities of different tasks.

Terms that we use while dealing with data structures:

- **Data:** It is the elementary value. In the above figure, student Roll No. can be data.

- **Group item:** This is the data item that has more than one sub-items. In the above figure, name has First Name and Last Name.
- **Record:** It is a collection of data items. In the above example, data items like student Roll No., Name, Class, Age, Grade, etc. form a record together.
- **Entity:** It is a class of records. In the above diagram, the student is an entity.
- **Attribute or field:** Properties of an entity are called attributes and each field represents an attribute.
- **File:** A file is a collection of records. In the above example, a student entity can have thousands of records. Thus a file will contain all these records.

Need For Data Structure In Programming

- **Searching Large amounts of Data:** With a large amount of data being processed and stored, at any given time our program may be required to search a particular data. If the data is too large and not organized properly, it will take a lot of time to get the required data.
- When we use data structures to store and organize data, the retrieval of data becomes faster and easier.
- **Speed of Processing:** Disorganized data may result in slow processing speed as a lot of time will be wasted in retrieving and accessing data.
- If we organize the data properly in a data structure while storing, then we will not waste time in activities like retrieving, organizing it every time. Instead, we can concentrate on the processing of data to produce the desired output.
- **Multiple Simultaneous Requests:** Many applications these days need to make a simultaneous request to data. These requests should be processed efficiently for applications to run smoothly.
- If our data is stored just randomly, then we will not be able to process all the concurrent requests simultaneously. So it's a wise decision to arrange data in a proper data structure so as to minimize the concurrent requests turnaround time.

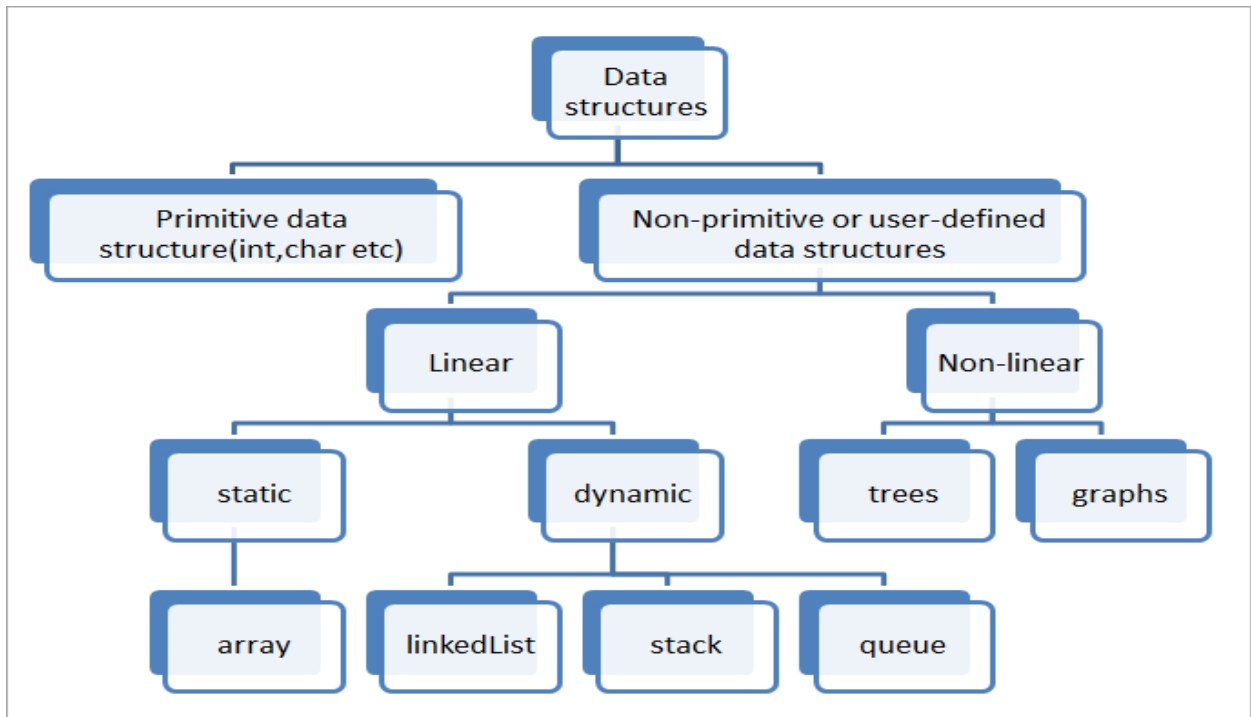
Abstract or User-Defined Data Types:

Abstract or User-Defined Data Types: These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

- **Class**
- **Structure**

- Union
- Enumeration
- Typedef defined DataType

Data Structure Classification



What is a Static Data structure?

In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

Array Length = 9

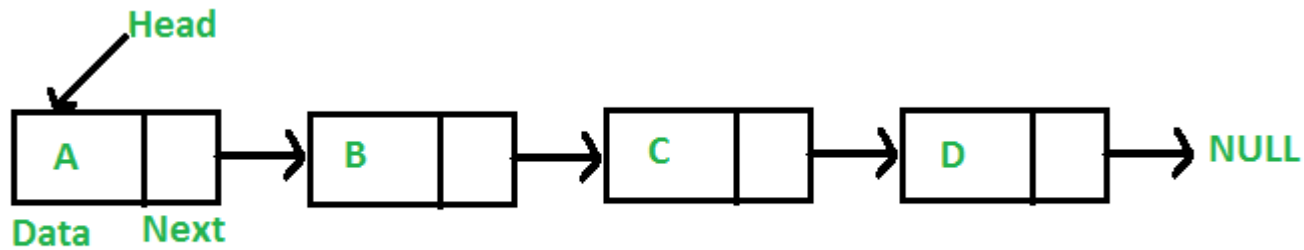
First Index = 0

Last Index = 8

Example of Static Data Structures: Array

What is a Dynamic Data Structure?

In Dynamic data structure the size of the structure is not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.



Example of Dynamic Data Structures: Linked List

Static Data Structure vs Dynamic Data Structure

- Static data structures, such as arrays, have a fixed size and are allocated at compile-time. This means that their memory size cannot be changed during program execution. Index-based access to elements is fast and efficient since the address of the element is known.
- Dynamic data structures, on the other hand, have a variable size and are allocated at run-time. This means that their memory size can be changed during program execution. Memory can be dynamically allocated or de-allocated during program execution. Due to this dynamic nature, accessing elements based on index may be slower as it may require memory allocation and de-allocation.

Aspect	Static Data Structure	Dynamic Data Structure
Memory allocation	Memory is allocated at compile-time	Memory is allocated at run-time
Size	Size is fixed and cannot be modified	Size can be modified during runtime
Memory utilization	Memory utilization may be inefficient	Memory utilization is efficient as memory can be reused
Access	Access time is faster as it is fixed	Access time may be slower due to indexing and pointer usage
Examples	Arrays, Stacks, Queues, Trees (with fixed size)	Lists, Trees (with variable size), Hash tables

Advantage of Static data structure :

- **Fast access time:** Static data structures offer fast access time because memory is allocated at compile-time and the size is fixed, which makes accessing elements a simple indexing operation.
- **Predictable memory usage:** Since the memory allocation is fixed at compile-time, the programmer can precisely predict how much memory will be used by the program, which is an important factor in memory-constrained environments.
- **Ease of implementation and optimization:** Static data structures may be easier to implement and optimize since the structure and size are fixed, and algorithms can be optimized for the specific data structure, which reduces cache misses and can increase the overall performance of the program.
- **Efficient memory management:** Static data structures allow for efficient memory allocation and management. Since the size of the data structure is fixed at compile-time, memory can be allocated and released efficiently, without the need for frequent reallocations or memory copies.
- **Simplified code:** Since static data structures have a fixed size, they can simplify code by removing the need for dynamic memory allocation and associated error checking.
- **Reduced overhead:** Static data structures typically have lower overhead than dynamic data structures, since they do not require extra bookkeeping to manage memory allocation and deallocation.

Advantage Of Dynamic Data Structure :

- **Flexibility:** Dynamic data structures can grow or shrink at runtime as needed, allowing them to adapt to changing data requirements. This flexibility makes them well-suited for situations where the size of the data is not known in advance or is likely to change over time.
- **Reduced memory waste:** Since dynamic data structures can resize themselves, they can help reduce memory waste. For example, if a dynamic array needs to grow, it can allocate additional memory on the heap rather than reserving a large fixed amount of memory that might not be used.
- **Improved performance for some operations:** Dynamic data structures can be more efficient than static data structures for certain operations. For example, inserting or deleting elements in the middle of a dynamic list can be faster than with a static array, since the remaining elements can be shifted over more efficiently.
- **Simplified code:** Dynamic data structures can simplify code by removing the need for manual memory management. Dynamic data structures can also reduce the complexity of code for data structures that need to be resized frequently.
- **Scalability:** Dynamic data structures can be more scalable than static data structures, as they can adapt to changing data requirements as the data grows.

Linear data structures:

- Linear data structures have all their elements arranged in a linear or sequential fashion.
- Each element in a linear data structure has a predecessor (previous element) and a successor (next element)
- Linear data structures are further divided into static and dynamic data structures.

- Static data structures usually have a fixed size and once their size is declared at compile time it cannot be changed.
- Dynamic data structures can change their size dynamically and accommodate themselves.

Non-linear data structures:

- Elements of non-linear data structures are stored and accessed in non-linear order.
- Examples of non-linear data structure are: Tree and Graph

Operations On Data Structure

- **Searching:** This operation is performed to search for a particular element or a key. The most common searching algorithms are sequential/linear search and binary search.
- **Sorting:** Sorting operation involves arranging the elements in a data structure in a particular order either ascending or descending. There are various sorting algorithms that are available for data structures. Most popular among them are Quicksort, Selection sort, Merge sort, etc.
- **Insertion:** Insertion operation deals with adding an element to the data structure. This is the most important operation and as a result of the addition of an element, the arrangement changes and we need to take care that the data structure remains intact.
- **Deletion:** Deletion operation removes an element from the data structure. The same conditions that are to be considered for insertion are to be fulfilled in case of the deletion operation as well.
- **Traversing:** We say that we traverse a data structure when we visit each and every element in the structure. Traversing is required to carry out certain specific operations on the data structure.

Advantages Of Data Structure

- **Abstraction:** Data structures are often implemented as abstract data types. The users only access its outer interface without worrying about the underlying implementation. Thus data structure provides a layer of abstraction.
- **Efficiency:** Proper organization of data results in efficient access of data thereby making programs more efficient. Secondly, we can select the proper data structure depending on our requirements.
- **Reusability:** We can reuse the data structures that we have designed. They can be compiled into a library as well and distributed to the client.

Persistent Data Structure is a data structure which stores the state of the data structure at every timestamp whenever an operation (like insert or delete) is done which modifies the structure of the Data Structure.

Operations on a Persistent Data Structure can be done on any of the saved versions of the Data Structure.

Advantages of Persistent Data Structure:

- Keeps track of all changes made on the Data Structure
- Easy to roll back to a previous state

Examples of Persistent Data Structures are:

- Persistent Segment Tree

Ephemeral Data Structure is a Data Structure which is neither persistent nor partially persistent. It does not store previous versions of the Data Structure. Only the current state of Data Structure is maintained.

Advantages of Ephemeral Data Structure:

- Does not keep track of previous operations / state and makes the overall structure simpler.

Examples of Ephemeral Data Structures:

- Linked List
- Array
- Hash Map
- Self-Balancing Binary Search Tree