U.N.S.I.E.T, VEER BAHADUR SINGH PURVANCHAL UNIVERSITY, JAUNPUR
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
EC/EI/EE- 2nd Sem
Subject- PROGRAMMING FOR PROBLEM SOLVING
RAVI KANT YADAV
(Assistant Professor)

UNIT-III

NOTES For

FUNCTIONS

VEER BAHADUR SINGH PURVANCHAL UNIVERSITY, JAUNPUR

(A UP State University Celebrating 25 Years of Excellence)

What is Function in C Language?

A function in C language is a block of code that performs a specific task. It has a name and it is reusable i.e. it can be executed from as many different parts in a C Program as required. It also optionally returns a value to the calling program

So function in a C program has some properties discussed below.

- Every function has a unique name. This name is used to call function from "main()" function. A function can be called from within another function.
- A function is independent and it can perform its task without intervention from or interfering with other parts of the program.
- A function performs a specific task. A task is a distinct job that your program must perform as a part of its overall operation, such as adding two or more integer, sorting an array into numerical order, or calculating a cube root etc.
- A function returns a value to the calling program. This is optional and depends upon the task your function is going to accomplish. Suppose you want to just show few lines through function then it is not necessary to return a value. But if you are calculating area of rectangle and wanted to use result somewhere in program then you have to send back (return) value to the calling function.

C language is collection of various inbuilt functions. If you have written a program in C then it is evident that you have used C's inbuilt functions. Printf, scanf, clrscr etc. all are C's inbuilt functions. You cannot imagine a C program without function.

Structure of a Function

A general form of a C function looks like this:

```
<return type> FunctionName (Argument1, Argument2, Argument3……)
{
Statement1;
Statement2;
Statement3;
}
```

An example of function.

```
int sum (int x, int y)
{
int result;
result = x + y;
return (result);
}
```

Advantages of using functions:

There are many advantages in using functions in a program they are:

1. It makes possible top down modular programming. In this style of programming, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. The length of the source program can be reduced by using functions at appropriate places.
3. It becomes uncomplicated to locate and separate a faulty function for further study.
4. A function may be used later by many other programs this means that a c programmer can use function written by others, instead of starting over from scratch.
5. A function can be used to keep away from rewriting the same block of codes which we are going use two or more locations in a program. This is especially useful if the code involved is long or complicated.

Types of functions:

A function may belong to any one of the following categories:

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions that return multiple values.
5. Functions with no arguments and return values.

Example of a simple function to add two integers.

```
view plain
1.  #include<stdio.h>
2.  #include<conio.h>
3.  void add(int x,int y)
4.  {
5.  int result;
6.  result = x+y;
7.  printf("Sum of %d and %d is %d.\n\n",x,y,result);
8.  }
9.  void main()
10. {
11. clrscr();
12. add(10,15);
13. add(55,64);
14. add(168,325);
15. getch();
16. }
```

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

void add(int x,int y)
{
        int result;
        result = x+y;
        printf("Sum of %d and %d is %d.\n\n",x,y,result);
}

void main()
{
clrscr();
add(10,15);
add(55,64);
add(168,325);
getch();
}
```

Used Defined Function

Function Calling

Program Output

```
Turbo C++ IDE
Sum of 10 and 15 is 25.
Sum of 55 and 64 is 119.
Sum of 168 and 325 is 493.
```

Types of Function in C Programming Languages:

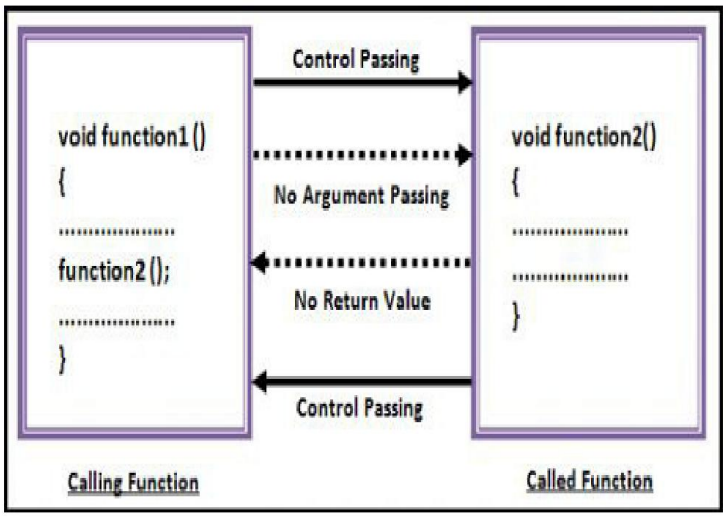
five types of functions and they are:

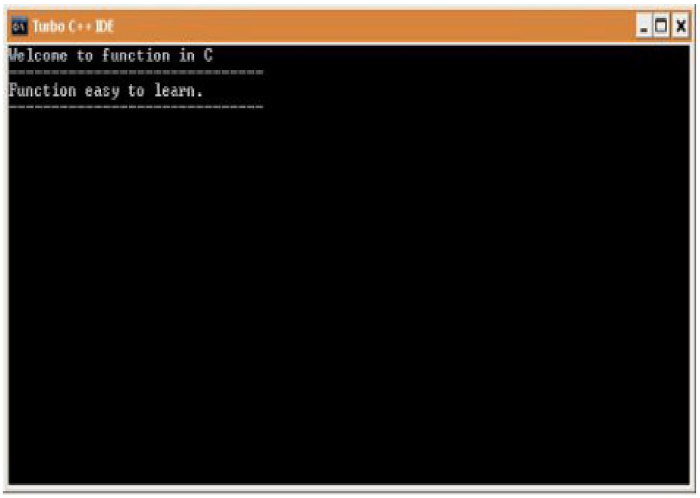
1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions that return multiple values.
5. Functions with no arguments and return values.

1. Functions with no arguments and no return value.

A C function without any arguments means you cannot pass data (values like int, char etc) to the called function. Similarly, function with no return type does not pass back data to the calling function. It is one of the simplest types of function in C. This type of function which does not return any value cannot be used in an expression it can be used only as independent statement. Let's have an example to illustrate this.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  void printline()
4.  {
5.  int i;
6.  printf("\n");
7.  for(i=0;i<30;i++)
8.  {
9.  printf("-");
10. }
11. printf("\n");
12. }
13. void main()
14. {
15. clrscr();
16. printf("Welcome to function in C");
17. printline();
18. printf("Function easy to learn.");
19. printline();
20. getch();
21. }
```





Output of above program.

Source Code Explanation:

The above C program example illustrates that how to declare a function with no argument and no return type. I am going to explain only important lines only because this C program example is for those who are above the beginner level.

Line 3-12: This C code block is a user defined function (UDF) whose task is to print a horizontal line. This is a simple function and a basic programmer can understand this. As you can see in line no. 7 I have declared a "for loop" which loops 30 time and prints "-" symbol continuously.

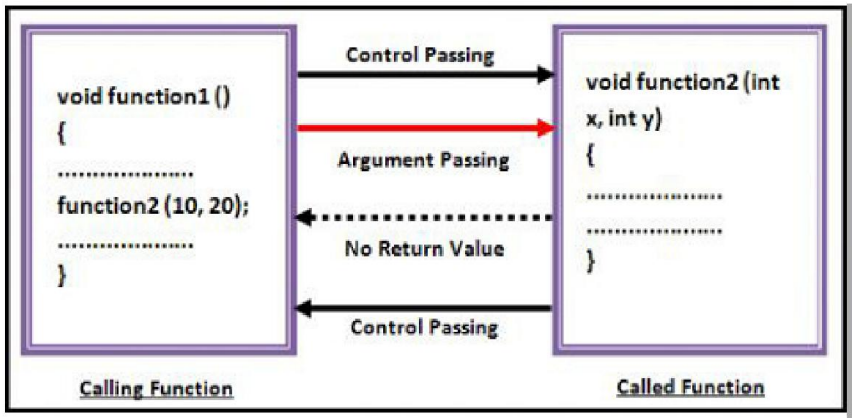
Line 13-21: These line are "main()" function code block. Line no. 16 and 18 simply prints two different messages. And line no. 17 and 18 calls our user defined function "printline()". You can see output this program below

2. Functions with arguments and no return value.

In our previous example what we have noticed that "main()" function has no control over the UDF "printfline()", it cannot control its output. Whenever "main()" calls "printline()", it simply prints line every time. So the result remains the same.

A C function with arguments can perform much better than previous function type. This type of function can accept data from calling function. In other words, you send data to the called function from calling function but you cannot send result data back to the calling function. Rather, it displays the result on the terminal. But we can control the output of function by providing various values as arguments. Let's have an example to get it better.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  void add(int x, int y)
4.  {
5.  int result;
6.  result = x+y;
7.  printf("Sum of %d and %d is %d.\n\n",x,y,result);
8.  }
9.  void main()
10. {
11. clrscr();
12. add(30,15);
13. add(63,49);
14. add(952,321);
15. getch();
16. }
```



Logic of the function with arguments and no return value.

Source Code Explanation:

This program simply sends two integer arguments to the UDF "add()" which, further, calculates its sum and stores in another variable and then prints that value. So simple program to understand.

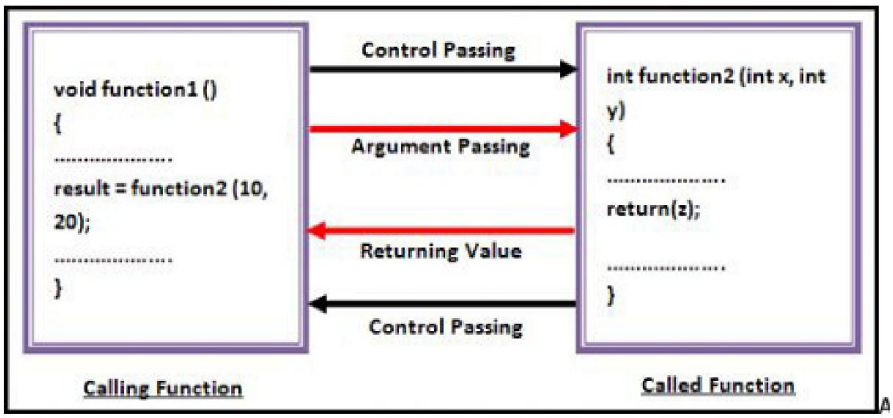
Line 3-8: This C code block is "add()" which accepts two integer type arguments. This UDF also has a integer variable "result" which stores the sum of values passed by calling function (in this example "main()"). And line no. 7 simply prints the result along with argument variable values.

Line 9-16: This code block is a "main()" function but only line no. 12, 13, 14 is important for us now. In these three lines we have called same function "add()" three times but with different values and each function call gives different output. So, you can see, we can control function's output by providing different integer parameters which was not possible in function type 1. This is the difference

3. Functions with arguments and return value.

This type of function can send arguments (data) from the calling function to the called function and wait for the result to be returned back from the called function back to the calling function. And this type of function is mostly used in programming world because it can do two way communications; it can accept data as arguments as well as can send back data as return value. The data returned by the function can be used later in our program for further calculations.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  int add(int x, int y)
4.  {
5.  int result;
6.  result = x+y;
7.  return(result);
8.  }
9.  void main()
10. {
11. int z;
12. clrscr();
13. z = add(952,321);
14. printf("Result %d.\n\n",add(30,55));
15. printf("Result %d.\n\n",z);
16. getch();
17. }
```



Logic of the function with arguments and return value.

Output of the above program.

Source Code Explanation:

This program sends two integer values (x and y) to the UDF "add()", "add()" function adds these two values and sends back the result to the calling function (in this program to "main()" function). Later result is printed on the terminal.

Line No. 3-8: Look line no. 3 carefully, it starts with **int**. This int is the return type of the function, means it can only return integer type data to the calling function. If you want any function to return character values then you must change this to char type. On line no. 7 you can see return statement, return is a keyword and in bracket we can give values which we want to return. You can assign any integer value to experiment with this return which ultimately will change its output. Do experiment with all you program and don't hesitate.

Line No. 9-17: In this code block only line no. 13, 14 and 15 is important. We have declared an integer "z" which we used in line no. 13. Why we are using integer variable "z" here? You know that our UDF "add()" returns an integer value on calling. To store that value we have declared an integer value. We have passed 952, 321 to the "add()" function, which finally return 1273 as result. This value will be stored in "z" integer variable. Now we can use "z" to print its value or to other function.

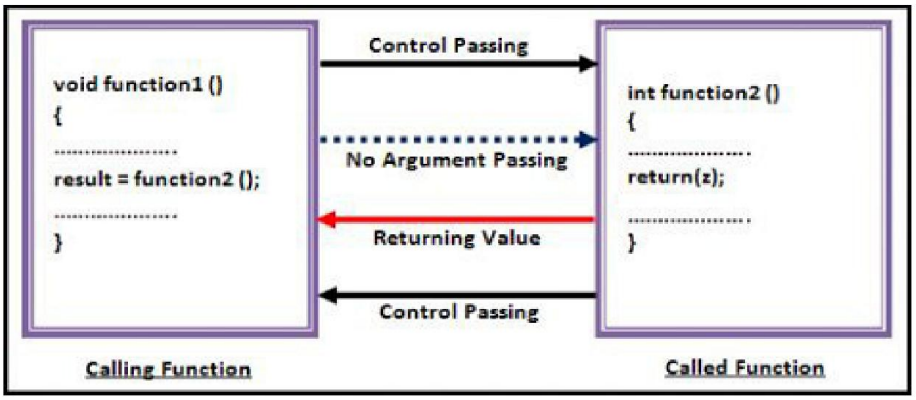
You will also notice some strange statement in line no. 14. Actually line no. 14 and 15 does the same job, in line no. 15 we have used an extra variable whereas on line no. 14 we directly printed the value without using any extra variable. This was simply to show you how we can use function in different ways.

4. Functions with no arguments but returns value.

We may need a function which does not take any argument but only returns values to the calling function then this type of function is useful. The best example of this type of function is "getchar()" library function which is declared in the header file "stdio.h". We can declare a similar library function of own. Take a look.

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  int send()
4.  {
5.  int no1;
6.  printf("Enter a no : ");
7.  scanf("%d",&no1);
8.  return(no1);
9.  }
10. void main()
11. {
12. int z;
13. clrscr();
14. z = send();
15. printf("\nYou entered : %d.", z);
16. getch();
17. }
```

Functions with no arguments and return values.



Source Code Explanation:

In this program we have a UDF which takes one integer as input from keyboard and sends back to the calling function. This is a very easy code to understand if you have followed all above code explanation. So I am not going to explain this code. But if you find difficulty please post your problem and I will solve that.

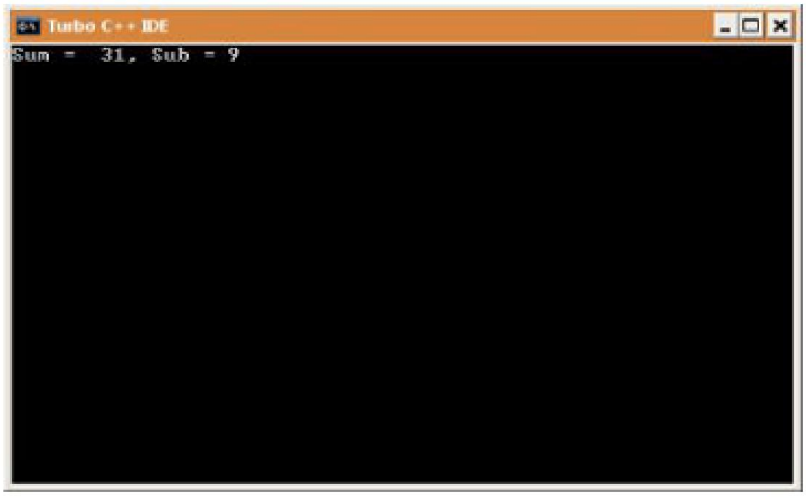
5. Functions that return multiple values.

So far, we have learned and seen that in a function, return statement was able to return only single value. That is because; a return statement can return only one value. But if we want to send back more than one value then how we could do this?

We have used arguments to send values to the called function, in the same way we can also use arguments to send back information to the calling function. The arguments that are used to send back data are called **Output Parameters**.

It is a bit difficult for novice because this type of function uses pointer. Let's see an example:

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.  void calc(int x, int y, int *add, int *sub)
4.  {
5.  *add = x+y;
6.  *sub = x-y;
7.  }
8.  void main()
9.  {
10. int a=20, b=11, p,q;
11. clrscr();
12. calc(a,b,&p,&q);
13. printf("Sum =  %d, Sub = %d",p,q);
14. getch();
15. }
```



Output of the above program.

Source Code Explanation:

Logic of this program is that we call UDF "calc()" and sends argument then it adds and subtract that two values and store that values in their respective pointers. The "*" is known as indirection operator whereas "&" known as address operator. We can get memory address of any variable by simply placing "&" before variable name. In the same way we get value stored at specific memory location by using "*" just before memory address. These things are a bit confusing but when you will understand pointer then these thing will become clearer.

Line no. 3-7: This UDF function is different from all above UDF because it implements pointer. I know line no. 3 looks something strange, let's have a clear idea of it. "Calc()" function has four arguments, first two arguments need no explanation. Last two arguments are integer pointer which works as output parameters (arguments). Pointer can only store address of the value rather than value but when we add * to pointer variable then we can store value at that address.

Line no. 8-15: When we call "calc()" function in the line no. 12 then following assignments occurs. Value of variable "a" is assigned to "x", value of variable "b" is assigned to "y", address of "p" and "q" to "add" and "sub" respectively. In line no. 5 and 6 we are adding and subtracting values and storing the result at their respective memory location. This is how the program works.

HOW TO CALL C FUNCTIONS IN A PROGRAM?

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

1. CALL BY VALUE:

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter can not be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY VALUE):

- In this program, the values of the variables "m" and "n" are passed to the function "swap".
- These values are copied to formal parameters "a" and "b" in swap function and used.

```
1  #include<stdio.h>
2  // function prototype, also called function declaration
3  void swap(int a, int b);
4
5  int main()
6  {
7      int m = 22, n = 44;
8      // calling swap function by value
9      printf(" values before swap  m = %d \nand n = %d", m, n);
10     swap(m, n);
11 }
12
13 void swap(int a, int b)
14 {
15     int tmp;
16     tmp = a;
17     a = b;
18     b = tmp;
19     printf(" \nvalues after swap m = %d\n and n = %d", a, b);
```

```
20 }
```

COMPILE & RUN

OUTPUT:

```
values before swap m = 22
and n = 44
values after swap m = 44
and n = 22
```

2. CALL BY REFERENCE:

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY REFERENCE):

- In this program, the address of the variables “m” and “n” are passed to the function “swap”.
- These values are not copied to formal parameters “a” and “b” in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.

```
1  #include<stdio.h>
2  // function prototype, also called function declaration
3  void swap(int *a, int *b);
4
5  int main()
6  {
7      int m = 22, n = 44;
8      //  calling swap function by reference
9      printf("values before swap m = %d \n and n = %d",m,n);
10     swap(&m, &n);
11 }
12
13 void swap(int *a, int *b)
14 {
15     int tmp;
16     tmp = *a;
17     *a = *b;
18     *b = tmp;
19     printf("\n values after swap a = %d \nand b = %d", *a, *b);
20 }
```

COMPILE & RUN

OUTPUT:

```
values before swap m = 22
and n = 44
values after swap a = 44
and b = 22
```

Difference between Call by Value and Call by Reference

Functions can be invoked in two ways: **Call by Value** or **Call by Reference**. These two ways are generally differentiated by the type of values passed to them as parameters.
The parameters passed to function are called *actual parameters* whereas the parameters received by function are called *formal parameters*.
Call By Value: In this parameter passing method, values of actual parameters are copied to function’s formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

Call by Reference: Both the actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

CALL BY VALUE	CALL BY REFERENCE
While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”.	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call By References.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.
In call by values we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
Values of variables are passes by Simple technique.	Pointer variables are necessary to define to store the address values of variables.

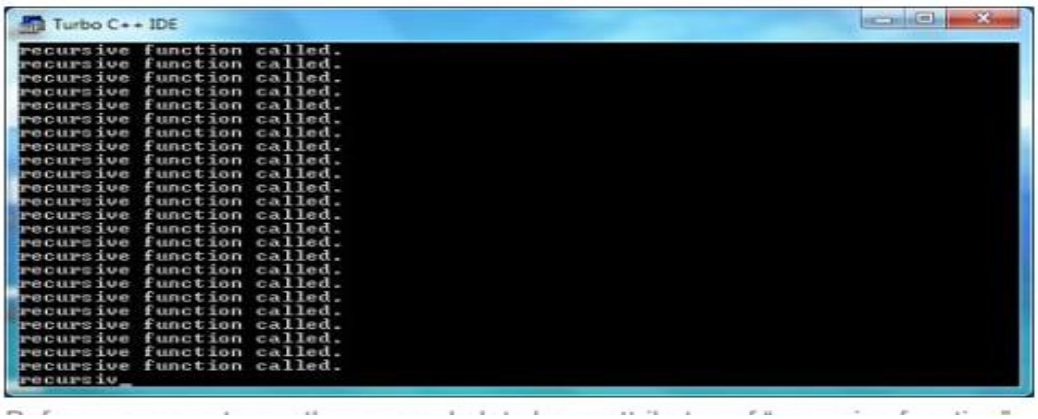
C Programming - Recursive Function

We have learnt different types function C language and now I am going to explain recursive function in C. A function is called “recursive” if a statement within body of that function calls the same function for example look at below code:

```
void main()
{
printf(“recursive function called.\n”);
main();
}
```

When you will run this program it will print message “recursive function called.” indefinitely. If you are using Turbo C/C++ compiler then you need to press Ctrl + Break key to break this in definite loop.

Recursive function example output



Before we move to another example lets have attributes of "recursive function":-

1. A recursive function is a function which calls itself.
2. The speed of a recursive program is slower because of stack overheads. (This attribute is evident if you run above C program.)
3. A recursive function must have recursive conditions, terminating conditions, and recursive expressions.

Calculating factorial value using recursion

To understand how recursion works lets have another popular example of recursion. In this example we will calculate the factorial of n numbers. The factorial of n numbers is expressed as a series of repetitive multiplication as shown below:

Factorial of n = n(n-1)(n-2).....1.

Example :

Factiorial of 5 = 5x4x3x2x1
=120

view plain

```
1.  #include<stdio.h>
2.  #include<conio.h>
3.
4.  int factorial(int);
5.
6.  int factorial (int i)
7.  {
8.      int f;
9.      if(i==1)
10.     return 1;
11.     else
12.     f = i* factorial (i-1);
13. return f;
14. }
15.
16. void main()
17. {
18.     int x;
19.     clrscr();
20.     printf("Enter any number to calculate factorial :");
21.     scanf("%d",&x);
22.     printf("\nFactorial : %d", factorial (x));
23.     getch();
24. }
```

Factorial value using recursive function output

So from **line no. 6 – 14** is a user defined recursive function "**factorial**" that calculates factorial of any given number. This function accepts integer type argument/parameter and return integer value. If you have any problem to understand how function works then you can check my tutorials on C function (click here).

In **line no. 9** we are checking that whether value of i is equal to 1 or not; i is an integer variable which contains value passed from main function i.e. value of integer variable x. If user enters 1 then the factorial of 1 will be 1. If user enters any value greater than 1 like 5 then it will execute statement in **line no. 12** to calculate factorial of 5. This line is extremely important because in this line we implemented recursion logic.

Let's see how **line no. 12** exactly works. Suppose value of i=5, since i is not equal to 1, the statement;

f = i* factorial (i-1);

will be executed with i=5 i.e.

f = 5* factorial (5-1);

will be evaluated. As you can see this statement again calls factorial function with value i-1 which will return value:

4*factorial(4-1);

This recursive calling continues until value of i is equal to 1 and when i is equal to 1 it returns 1 and execution of this function stops. We can review the series of recursive call as follow:

f = 5* factorial (5-1);

f = 5*4* factorial (4-1);

f = 5*4*3* factorial (3-1);

f = 5*4*3*2* factorial (2-1);

f = 5*4*3*2*1;

f = 120;

U.N.S.I.E.T, VEER BAHADUR SINGH PURVANCHAL UNIVERSITY, JAUNPUR
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

EC/EI/EE- 2nd Sem

Subject- PROGRAMMING FOR PROBLEM SOLVING

RAVI KANT YADAV
(Assistant Professor)

UNIT-IV

NOTES For

ARRAYS

VEER BAHADUR SINGH PURVANCHAL UNIVERSITY, JAUNPUR

(A UP State University Celebrating 25 Years of Excellence)

Array in C programming

What is an Array?

An array in C language is a collection of similar data-type, means an array can hold value of a particular data type for which it has been declared. Arrays can be created from any of the C data-types int, float, and char. So an integer array can only hold integer values and cannot hold values other than integer. When we declare array, it allocates contiguous memory location for storing values whereas 2 or 3 variables of same data-type can have random locations. So this is the most important difference between a variable and an array.

Types of Arrays:

1. One dimension array (Also known as 1-D array).
2. Two dimension array (Also known as 2-D array).
3. Multi-dimension array.

Decla

Syntax: data_type array_name[width];

Example: int roll[8];

In our example, int specifies the type if the variable, roll specifies the name of the variable and the value in bracket [8] is new for newbie. The bracket ([ ]) tells compiler that it is an array and number mention in the bracket specifies that how many elements (values in any array is called elements) it can store. This number is called dimension of array.

So, with respect to our example we have declared an array of integer type and named it "roll" which can store roll numbers of 8 students. You can see memory arrangement of above declared array in the following image:

Name	roll[0]	roll[1]	roll[2]	roll[3]	roll[4]	roll[5]	roll[6]	roll[7]
Values	12	45	32	23	17	49	5	11
Address	1000	1002	1004	1006	1008	1010	1012	1014

1-D Array memory arrangement

C Array Assignment and Initialization:

We can initialize and assign values to the arrays in the same way as we do with variable. We can assign value to an array at the time of declaration or during runtime. Let's look at each approach.

Syntax: data_type array_name[size]={list of values};

Example:

```
int arr[5]={1,2,3,4,5};
int arr[]={1,2,3,4,5};
```

In our above array example we have declared an integer array and named it "arr" which can hold 5 elements, we are also initializing arrays in the same time.

Both statements in our example are valid method to declare and initialize single dimension array. In our first example we mention the size (5) of an array and assigned it values in curly brace, separating element's value by comma (,). But in second example we left the size field blank but we provided its element's value. When we only give element values without providing size of an array then C compiler automatically assumes its size from given element values.

There is one more method to initialize array C programming; in this method we can assign values to individual element of an array. For this let's look at example:

Array Initialization Example

view plain

```
1. #include<stdio.h>
2. #include<conio.h>
3.
4. void main()
5. {
6. int arr[5],i;
7. clrscr();
8. arr[0]=10;
9. arr[1]=20;
10. arr[2]=30;
11. arr[3]=40;
12. arr[4]=50;
13.
14. printf("Value in array arr[0] : %d\n",arr[0]);
15. printf("Value in array arr[1] : %d\n",arr[1]);
16. printf("Value in array arr[2] : %d\n",arr[2]);
17. printf("Value in array arr[3] : %d\n",arr[3]);
18. printf("Value in array arr[4] : %d\n",arr[4]);
19. printf("\n");
20.
21. for(i=0;i<5;i++)
22. {
23.     printf("Value in array arr[%d] : %d\n",i,arr[i]);
24. }
25. getch();
26. }
```

In the above c arrays example we have assigned the value of integer array individually like we do with an integer variable. We have called array element's value individually and using for loop so that it would be clear for beginner and semi-beginner C programmers. So, from the above example it is evident that we can assign values to an array element individually and can call them individually whenever we need them.

How to work with Two Dimensional Arrays in C

We know how to work with an array (1D array) having one dimension. In C language it is possible to have more than one dimension in an array. In this tutorial we are going to learn how we can use two dimensional arrays (2D arrays) to store values. Because it is a 2D array so its structure will be different from one dimension array. The 2D array is also known as Matrix or Table, it is an array of array. See the below image, here each row is an array.

Declaration of 2D array:

Syntax: data_type array_name[row_size][column_size];

Example: int arr[3][3];

So the above example declares a 2D array of integer type. This integer array has been named arr and it ca hold up to 9 elements (3 rows x 3 columns).

2D Array

arr	col [0]	col [1]	col [2]
row [0]	10	20	45
row [1]	42	79	81
row [2]	89	9	36

2D Array Arrangement

Memory Map of 2D Array

arr[0][0]	arr[0][1]	arr[0][2]	arr[1][0]	arr[1][1]	arr[1][2]	arr[2][0]	arr[2][1]	arr[2][2]
12	45	63	89	34	73	19	76	49
1000	1002	1004	1006	1008	1010	1012	1014	1016

Memory Map of 2 Dimentional Array

Code for assigning & displaying 2D Array

view plain

```
1. #include<stdio.h>
2. #include<conio.h>
3.
4. void main()
5. {
6. int i, j;
7. int arr[3][3]={
8.          {12, 45, 63},
9.          {89, 34, 73},
10.         {19, 76, 49}
11.         };
12. clrscr();
13. printf(":::2D Array Elements:::\n\n");
14. for(i=0;i<3;i++)
15. {
16.     for(j=0;j<3;j++)
17.     {
18.     printf("%d\t",arr[i][j]);
19.     }
20.     printf("\n");
21. }
22. getch();
23. }
```

So in the above example we have declared a 2D array named arr which can hold 3x3 elements. We have also initialized that array with values, because we told the compiler that this array will contain 3 rows (0 to 2) so we divided elements accordingly. Elements for column have been differentiated by a comma (,). When compiler finds comma in array elements then it assumes comma as beginning of next element value. We can also define the same array in other ways, like.
int arr[3][3]={12, 45, 63, 89, 34, 73, 19, 76, 49}; or,
int arr[ ][3]={12, 45, 63, 89, 34, 73, 19, 76, 49};

But this kind of declaration is not acceptable in C language programming.

int arr[2][ ]={12, 45, 63, 89, 34, 73, 19, 76, 49}; or,
int arr[ ][ ]={12, 45, 63, 89, 34, 73, 19, 76, 49};

To display 2D array elements we have to just point out which element value we want to display. In our example we have a arr[3][3], so the array element reference will be from arr[0][0] to arr[2][2]. We can print display any element from this range. But in our example I have used for loop for my convenience, otherwise I had to write 9 printf statements to display all elements of array. So for loop i handles row of 2D array and for loop j handles column. I have formatted the output display of array so that we can see the elements in tabular form.

How to work with Multidimensional Array in C Programming

C allows array of two or more dimensions and maximum numbers of dimension a C program can have is depend on the compiler we are using. Generally, an array having one dimension is called 1D array, array having two dimensions called 2D array and so on. So in C programming an array can have two or three or four or even ten or more dimensions. More dimensions in an array means more data it can hold and of course more difficulties to manage and understand these arrays. A multidimensional array has following syntax:

Syntax:

type array_name[d1][d2][d3][d4]……[dn];
Where dn is the size of last dimension.

Example:

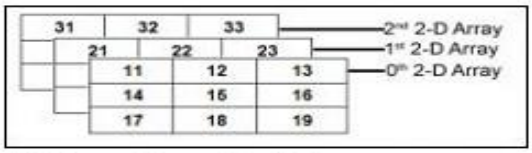
int table[5][5][20];
float arr[5][6][5][6][5];

In our example array “table” is a 3D (A 3D array is an array of arrays of arrays.) array which can hold 500 integer type elements. And array “arr” is a 5D array which can hold 4500 floating-point elements. Can see the power of array over variable? When it comes to hold multiple values in a C programming, we need to declare several variables (for example to store 150 integers) but in case of array, a single array can hold thousands of values (depending on compiler, array type etc).

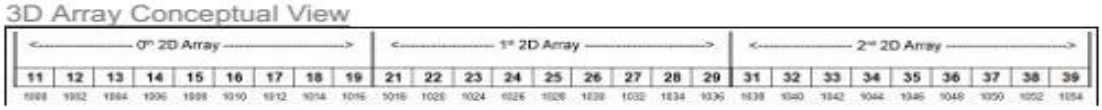
Note: To make this multidimensional array example simple I will discuss 3D array for the sake of simplicity. Once you grab the logic how 3D array works then you can handle 4D array or any multidimensional array easily.

How to Declaration and Initialization 3D Array

Before we move to serious programming let's have a look of 3D array. A 3D array can be assumed as an array of arrays of arrays, it is array (collection) of 2D arrays and as you know 2D array itself is array of 1D array. It sounds a bit confusing but don't worry as you will lead your learning on multidimensional array, you will grasp all logic and concept. A diagram can help you to understand this.



3D Array Conceptual View



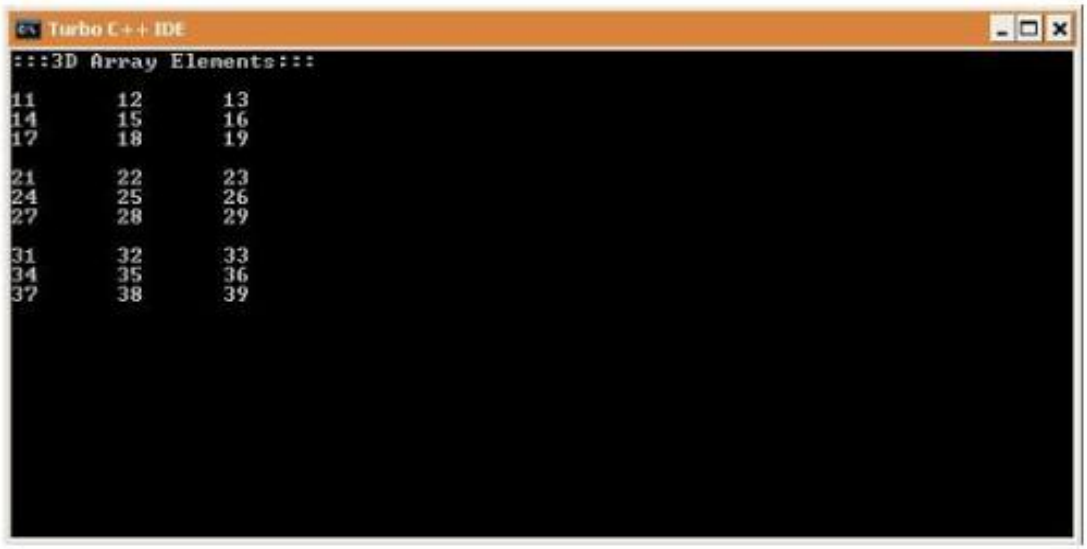
3D array memory map.

We can initialize a 3D array at the compile time as we initialize any other variable or array, by default an un-initialized 3D array contains garbage value. Let's see a complete example on how we can work with a 3D array.

Example of Declaration and Initialization 3D Array

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

void main()
{
int i, j, k;
int arr[3][3][3]=
        {
            {
            {11, 12, 13},
            {14, 15, 16},
            {17, 18, 19}
            },
            {
            {21, 22, 23},
            {24, 25, 26},
            {27, 28, 29}
            },
            {
            {31, 32, 33},
            {34, 35, 36},
            {37, 38, 39}
            },
        };
clrscr();
printf(":::3D Array Elements:::\n\n");
for(i=0;i<3;i++)
{
   for(j=0;j<3;j++)
   {
        for(k=0;k<3;k++)
        {
        printf("%d\t",arr[i][j][k]);
        }
        printf("\n");
   }
   printf("\n");
}
getch();
}
```



So in the above example we have declared multidimensional array and named this integer array as "arr" which can hold 3x3x3 (27 integers) elements. We have also initialized multidimensional array with some integer values.

As I told you earlier that a 3D array is array of 2D array therefore I have divided elements accordingly so that you can get 3D array better and understand it easily. See the C code sample above, line no. 9-13, 14-18 and 19-23, each block is a 2D array and collectively from line no. 2-24 makes a 3D array. You can also assign values to this multidimensional array in other way like this.

int arr[3][3][3] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39};

This kind of C multidimensional array (3D array) declaration is quite confusing for new C programmers; you cannot guess location of array element by just looking at the declaration. But look at the above multidimensional array example where you can get a clear idea about each element location. For example, consider 3D array as a collection of tables, to access or store any element in a 3D array you need to know first table number then row number and lastly column number. For instance you need to access value 25 from above 3D array. So, first check the table (among 3 tables which table has the value), once you find the table number now check which row of that table has the value again if you get the row no then check column number and you will get the value. So applying above logic, 25 located in table no. 1 row no. 1 and column no. 1, hence the address is arr[1][1][1]. Print this address and you will get the output.

So the conceptual syntax for 3D array stands like this.

data_type array_name[table][row][column];

If you want to store values in any 3D array then first point to table number, row number and lastly to column number.

arr[0][1][2] = 32;
arr[1][0][1] = 49;

Above code is for assigning values at particular location of an array but if you want to store value in continuous location of array then you should use loop. Here is an example using for loop.

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

void main()
{
int i, j, k, x=1;
int arr[3][3][3];
clrscr();
printf(":::3D Array Elements:::\n\n");

for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        for(k=0;k<3;k++)
        {
        arr[i][j][k] = x;
        printf("%d\t",arr[i][j][k]);
        x++;
        }
        printf("\n");
    }
    printf("\n");
}
getch();
}
```

CHARACTER ARRAY AND STRING IN C

String and Character Array

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

Declaring and Initializing a string variables

There are different ways to initialize a character array variable.

```
char name[13] = "StudyTonight";       // valid character array initialization

char name[10] = {'L','e','s','s','o','n','s','\0'};     // valid initialization
```

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

Some examples of illegal initialization of character array are,

```
char ch[3] = "hell";    // Illegal


char str[4];

str = "hell";   // Illegal
```

String Input and Output

Input function scanf() can be used with **%s** format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World" using scanf() function, it will only read **Hello** and terminate after encountering white spaces.

However, C supports a format specification known as the **edit set conversion code %[..]** that can be used to read a line containing a variety of characters, including white spaces.

```
#include<stdio.h>

#include<string.h>

void main()

{

   char str[20];

   printf("Enter a string");

   scanf("%[^\n]", &str);  //scanning the whole string, including the white spaces

   printf("%s", str);

}
```

Another method to read character string with white spaces from terminal is by using the gets() function.

```
char text[20];

gets(text);

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

String Handling Functions

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

The following are the most commonly used string handling functions.

Method	Description
strcat()	It is used to concatenate(combine) two strings
strlen()	It is used to show length of a string
strrev()	It is used to show reverse of a string
strcpy()	Copies one string into another
strcmp()	It is used to compare two string

strcat() function

```
strcat("hello", "world");
```

strcat() function will add the string **"world"** to **"hello"** i.e it will ouput helloworld.

strlen() function

strlen() function will return the length of the string passed to it.

```
int j;
j = strlen("studytonight");
printf("%d",j);
```

strcmp() function

strcmp() function will return the ASCII difference between first unmatching character of two strings.

```
int j;
j = strcmp("study", "tonight");
printf("%d",j);
```

strcpy() function

It copies the second string argument to the first string argument.

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

int main()
{
    char s1[50];
    char s2[50];

    strcpy(s1, "StudyTonight");     //copies "studytonight" to string s1
    strcpy(s2, s1);     //copies string s1 to string s2

    printf("%s\n", s2);

    return(0);
}
```

strrev() function

It is used to reverse the given string expression.

```
#include<stdio.h>

int main()
{
    char s1[50];
    printf("Enter your string: ");
    gets(s1);
    printf("\nYour reverse string is: %s",strrev(s1));
    return(0);
}
```

```
Enter your string: studytonight
Your reverse string is: thginotyduts
```

STRUCTURE IN C PROGRAMMING

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

Lets 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 can be anything of your choice. 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

```
#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
```

UNION IN C PROGRAMMING

C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members
- We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.
- Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Using normal variable	Using pointer variable

Syntax:
union tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	union tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

Example:
union student
{
int mark;
char name[10];
float average;
};
	Example:
	union student
	{
	int mark;
	char name[10];
	float average;
	};

Declaring union using normal variable: union student report;	**Declaring union using pointer variable:** union student *report, rep;
Initializing union using normal variable: union student report = {100, “Mani”, 99.5};	**Initializing union using pointer variable:** union student rep = {100, “Mani”, 99.5}; report = &rep;
Accessing union members using normal variable: report.mark; report.name; report.average;	**Accessing union members using pointer variable:** report -> mark; report -> name; report -> average;

EXAMPLE PROGRAM FOR C UNION:

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

union student
{
        char name[20];
        char subject[20];
        float percentage;
};

int main()
{
   union student record1;
   union student record2;

   // assigning values to record1 union variable
      strcpy(record1.name, "Raju");
      strcpy(record1.subject, "Maths");
      record1.percentage = 86.50;

      printf("Union record1 values example\n");
      printf(" Name       : %s \n", record1.name);
      printf(" Subject    : %s \n", record1.subject);
```

```
24      printf(" Percentage : %f \n\n", record1.percentage);
25
26    // assigning values to record2 union variable
27      printf("Union record2 values example\n");
28      strcpy(record2.name, "Mani");
29      printf(" Name       : %s \n", record2.name);
30
31      strcpy(record2.subject, "Physics");
32      printf(" Subject    : %s \n", record2.subject);
33
34      record2.percentage = 99.50;
35      printf(" Percentage : %f \n", record2.percentage);
36      return 0;
37 }
```

OUTPUT:

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

EXPLANATION FOR ABOVE C UNION PROGRAM:

There are 2 union variables declared in this program to understand the difference in accessing values of union members.

Record1 union variable:

- "Raju" is assigned to union member "record1.name" . The memory location name is "record1.name" and the value stored in this location is "Raju".
- Then, "Maths" is assigned to union member "record1.subject". Now, memory location name is changed to "record1.subject" with the value "Maths" (Union can hold only one member at a time).
- Then, "86.50" is assigned to union member "record1.percentage". Now, memory location name is changed to "record1.percentage" with value "86.50".
- Like this, name and value of union member is replaced every time on the common storage space.
- So, we can always access only one union member for which value is assigned at last. We can't access other member values.
- So, only "record1.percentage" value is displayed in output. "record1.name" and "record1.percentage" are empty.

Record2 union variable:

- If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.
- Each union members are accessed in record2 example immediately after assigning values to them.
- If we don't access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.
- We can't access all members in union at same time but structure can do that.

EXAMPLE PROGRAM – ANOTHER WAY OF DECLARING C UNION:

In this program, union variable "record" is declared while declaring union itself as shown in the below program.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  union student
5  {
```

```
6           char name[20];
7           char subject[20];
8           float percentage;
9   }record;
10
11 int main()
12 {
13
14          strcpy(record.name, "Raju");
15          strcpy(record.subject, "Maths");
16          record.percentage = 86.50;
17
18          printf(" Name       : %s \n", record.name);
19          printf(" Subject    : %s \n", record.subject);
20          printf(" Percentage : %f \n", record.percentage);
21          return 0;
22 }
```

OUTPUT:

```
Name :
Subject :
Percentage : 86.500000
```

NOTE:

- We can access only one member of union at a time. We can't access all member values at the same time in union.
- But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.

DIFFERENCE BETWEEN STRUCTURE AND UNION IN C:

C Structure	C Union
Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
Structure occupies higher memory space.	Union occupies lower memory space over structure.
We can access all members of structure at a time.	We can access only one member of union at a time.

Structure example:
struct student
{
int mark;
char name[6];
double average;
};
	Union example:
	union student
	{
	int mark;
	char name[6];
	double average;
	};

For above structure, memory allocation will be like below.	For above union, only 8 bytes of memory will be allocated since double data type will occupy

int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	maximum space of memory over other data types. Total memory allocation=8 bytes

ENUMERATED DATA TYPE IN C

enum in C

Enumeration (enum) is a user-defined datatype (same as structure). It consists of various elements of that type. There is no such specific use of enum, we use it just to make our codes neat and more readable. We can write C programs without using enumerations also.

For example, Summer, Spring, Winter and Autumn are the names of four seasons. Thus, we can say that these are of types season. Therefore, this becomes an enumeration with name season and Summer, Spring, Winter and Autumn as its elements.

So, you are clear with the basic idea of enum. Now let's see how to define it.

Defining an Enum

An enum is defined in the same way as structure with the keyword struct replaced by the keyword enum and the elements separated by 'comma' as follows.

```
enum enum_name
{
  element1,
  element2,
  element3,
  element4,
};
```

Now let's define an enum of the above example of seasons.

```
enum Season{
  Summer,
  Spring,
  Winter,
  Autumn
};
```

Here, we have defined an enum with name 'season' and 'Summer, Spring, Winter and Autumn' as its elements.

Declaration of Enum Variable

We also declare an enum variable in the same way as that of structures. We create an enum variable as follows.

```
enum season{
  Summer,
  Spring,
  Winter,
  Autumn
};
```

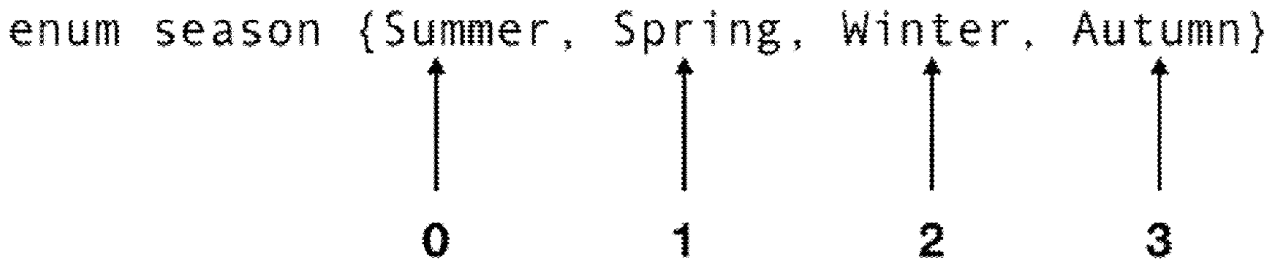
```
main()
{
  enum season s;
}
```

So, here 's' is the variable of the enum named season. This variable will represent a season. We can also declare an enum variable as follows.

```
enum season{
  Summer,
  Spring,
  Winter,
  Autumn
}s;
```

Values of the Members of Enum

All the elements of an enum have a value. By default, the value of the first element is 0, that of the second element is 1 and so on.



Let's see an example.

```c
#include <stdio.h>
enum season{ Summer, Spring, Winter, Autumn};
int main()
{
   enum season s;
   s = Spring;
   printf("%d\n",s);
   return 0;
}
```

Output

Here, first we defined an enum named 'season' and declared its variable 's' in the main function as we have seen before. The values of Summer, Spring, Winter and Autumn are 0, 1, 2 and 3 respectively. So, by writing s = Spring, we assigned a value '1' to the variable 's' since the value of 'Spring' is 1.

We can also change the default value and assign any value of our choice to an element of enum. Once we change the default value of any enum element, then the values of all the elements after it will also be changed accordingly. An example will make this point clearer.

```c
#include <stdio.h>
enum days{ sun, mon, tue = 5, wed, thurs, fri, sat};
int main()
```

```
{
   enum days day;
   day = thurs;
   printf("%d\n",day);
   return 0;
}
```

Output

The default value of 'sun' will be 0, 'mon' will be 1, 'tue' will be 2 and so on. In the above example, we defined the **value of tue** as **5**. So the values of 'wed', 'thurs', 'fri' and 'sat' will become 6, 7, 8 and 9 respectively. There will be no effect on the values of sun and mon which will remain 0 and 1 respectively. Thus the value of thurs i.e. 7 will get printed.

Let's see one more example of enum.

```
#include <stdio.h>
enum days{ sun, mon, tue, wed, thurs, fri, sat};
int main()
{
   enum days day;
   day = thurs;
   printf("%d\n",day+2);
   return 0;
}
```

Output

In this example, the value of 'thurs' i.e. 4 is assigned to the variable day. Since we are printing 'day+2' i.e. 6 (=4+2), so the output will be 6.

ARRAY OF STRUCTURE

Array of Structures in C

In C Programming, Structures are useful to group different data types to organize the data in a structural way. And Arrays are used to group the same data type values. In this article, we will show you the Array of Structures in C concept with one practical example.

For example, we are storing employee details such as name, id, age, address, and salary. We usually group them as employee structure with the members mentioned above. We can create the structure variable to access or modify the structure members. A company may have 10 to 100 employee, how about storing the same for 100 employees?

In C Programming, We can easily solve the problem mentioned above by combining two powerful concepts Arrays of Structures in C. We can create the employee structure. Then instead of creating the structure variable, we create the array of a structure variable.

Declaring C Array of Structures at structure Initialization

Let me declare an Array of Structures in C at the initialization of the structure

```
/* Array of Structures in C Initialization */
struct Employee
{
  int age;
  char name[50];
  int salary;
} Employees[4] = {
                {25, "Suresh", 25000},
                {24, "Tutorial", 28000},
                {22, "Gateway", 35000},
                {27, "Mike", 20000}
                };
```

Here, Employee structure is for storing the employee details such as age, name, and salary. We created the array of structures variable Employees [4] (with size 4) at the declaration time only. We also initialized the values of each structure member for all 4 employees.

From the above,

```
Employees[0] = {25, "Suresh", 25000}

Employees[1] = {24, "Tutorial", 28000}

Employees[2] = {22, "Gateway", 35000}

Employees[3] = {27, "Mike", 20000}
```

Declaring C Array of Structures in main() Function

```
/* Array of Structures in C Initialization */
struct Employee
{
  int age;
  char name[50];
  int salary;
};
```

Within the main() function, Create the Employee Structure Variable

```
struct Employee  Employees[4];
Employees[4] = {
                {25, "Suresh", 25000},
                {24, "Tutorial", 28000},
                {22, "Gateway", 35000},
                {27, "Mike", 20000}
                };
```

Array of Structures in C Example

This program for an Array of Structures in C will declare the student structure and displays the information of N number of students.

```
/* Array of Structures in C example */

#include <stdio.h>

struct Student
{
  char Student_Name[50];
```

```
  int C_Marks;
  int DataBase_Marks;
  int CPlus_Marks;
  int English_Marks;
};

int main()
{
  int i;
  struct Student Students[4] =
             {
              {"Suresh", 80, 70, 60, 70},
              {"Tutorial", 85, 82, 65, 68},
              {"Gateway", 75, 70, 89, 82},
              {"Mike", 70, 65, 69, 92}
              };

  printf(".....Student Details....");
  for(i=0; i<4; i++)
  {
    printf("\n Student Name = %s", Students[i].Student_Name);
    printf("\n First Year Marks = %d", Students[i].C_Marks);
    printf("\n Second Year Marks = %d", Students[i].DataBase_Marks);
    printf("\n First Year Marks = %d", Students[i].CPlus_Marks);
    printf("\n Second Year Marks = %d", Students[i].English_Marks);
  }

 return 0;
}
```

OUTPUT:

```
struct Student
{
  char Student_Name[50];
  int C_Marks;
  int DataBase_Marks;
  int CPlus_Marks;
  int English_Marks;
};
int main()
{
  int i;
  struct Student Students[4] =
  {
              {"Suresh", 80, 70, 60, 70},
              {"Tutorial", 85, 82, 65, 68},
              {"Gateway", 75, 70, 89, 82},
              {"Mike", 70, 65, 69, 92}
```

```
C:\Users\Personal\Documents\Unt
.....Student Details.... ©tutorialgateway.org

Student Name         = Suresh
C Programming Marks  = 80
Data Base Marks      = 70
C++ Marks            = 60
English Marks        = 70

Student Name         = Tutorial
C Programming Marks  = 85
Data Base Marks      = 82
C++ Marks            = 65
English Marks        = 68

Student Name         = Gateway
C Programming Marks  = 75
Data Base Marks      = 70
C++ Marks            = 89
English Marks        = 82

Student Name         = Mike
C Programming Marks  = 70
Data Base Marks      = 65
C++ Marks            = 69
English Marks        = 92
```

ANALYSIS

Within this Array of Structures in C example, We declared the student structure with Student Name, C Marks, DataBase Marks, C++ Marks, and English Marks members of different data types.

Within the main() function, we created the array of structures student variable. Next, we initialized the appropriate values to the structure members

In the Next line, we have For Loop in C Programming Condition inside the for loop. It will control the compiler not to exceed the array limit. The below printf statements will print the values inside the student structure array.

```
printf("\n Student Name        = %s", Students[i].Student_Name);

printf("\n C Programming Marks = %d", Students[i].C_Marks);

printf("\n Data Base Marks     = %d", Students[i].DataBase_Marks);

printf("\n C++ Marks           = %d", Students[i].CPlus_Marks);

printf("\n English Marks       = %d", Students[i].English_Marks);
```

Let us explore the Array of Structures in C program in iteration wise

First Iteration

Student Name = Students[i].Student_Name

C Programming Marks = Students[i].C_Marks

Data Base Marks = Students[i].DataBase_Marks

C++ Marks = Students[i].CPlus_Marks

English Marks = Students[i].English_Marks

i = 0 and the condition 0 < 4 is TRUE so

Student Name = Students[0].Student_Name = Suresh

C Programming Marks = Students[0].C_Marks = 80

Data Base Marks = Students[0].DataBase_Marks= 70

C++ Marks = Students[0].CPlus_Marks = 60

English Marks = Students[0].English_Marks = 70

i value incremented by 1 using i++ Incremental Operator. So i becomes 1

Second Iteration of Array of Structures in C

i = 1 and the condition 1 < 4 is TRUE

Student Name = Students[1].Student_Name = Tutorial

C Programming Marks = Students[1].C_Marks = 85

Data Base Marks = Students[1].DataBase_Marks = 82

C++ Marks = Students[1].CPlus_Marks = 65

English Marks = Students[1].English_Marks = 68

i value incremented by 1. So i becomes 2

Third Iteration

i = 2 and the condition 2 < 4 is TRUE

Student Name = Students[2].Student_Name = Gateway

C Programming Marks = Students[2].C_Marks = 75

Data Base Marks = Students[2].DataBase_Marks = 70

C++ Marks = Students[2].CPlus_Marks = 89

English Marks = Students[2].English_Marks = 82

i value incremented by 1. So i becomes 3

Fourth Iteration of the Array of Structures in C

i = 3, and the condition 3 < 4 is TRUE so,

Student Name = Students[0].Student_Name = Mike

C Programming Marks = Students[0].C_Marks = 70

Data Base Marks = Students[0].DataBase_Marks = 65

C++ Marks = Students[0].CPlus_Marks = 69

English Marks = Students[0].English_Marks = 92

i value incremented by 1 using i++ incremental Operator. So, i becomes 4, and the i<4 condition Fails. So, the compiler will exit from the loop.

PASSING ARRAY TO FUNCTION IN c

How to pass Array to a Function in C

Whenever we need to pass a list of elements as argument to any function in C language, it is prefered to do so using an array. But how can we pass an array as argument to a function? Let's see how its done.

Declaring Function with array as a parameter

There are two possible ways to do so, one by using call by value and other by using call by reference.

1. We can either have an array as a parameter.

```
int sum (int arr[]);
```

2. Or, we can have a pointer in the parameter list, to hold the base address of our array.

```
int sum (int* ptr);
```

   We will study the second way in details later when we will study pointers.

Returning an Array from a function

We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned. But we must, make sure that the array exists after the function ends i.e. the array is not local to the function.

```
int* sum (int x[])
{
   // statements
   return x ;
}
```

We will discuss about this when we will study pointers with arrays.

Passing arrays as parameter to function

Now let's see a few examples where we will pass a single array element as argument to a function, a one dimensional array to a function and a multidimensional array to a function.

Passing a single array element to a function

Let's write a very simple program, where we will declare and define an array of integers in our main() function and pass one of the array element to a function, which will just print the value of the element.

```
#include<stdio.h>

void giveMeArray(int a);

int main()
{
   int myArray[] = { 2, 3, 4 };
```

```
    giveMeArray(myArray[2]);        //Passing array element myArray[2] only.
    return 0;
}


void giveMeArray(int a)
{
    printf("%d", a);
}
```

Passing a complete One-dimensional array to a function

To understand how this is done, let's write a function to find out average of all the elements of the array and print it.

We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

```
#include<stdio.h>


float findAverage(int marks[]);


int main()
{
    float avg;
    int marks[] = {99, 90, 96, 93, 95};
    avg = findAverage(marks);       // name of the array is passed as argument.
    printf("Average marks = %.1f", avg);
    return 0;
}


float findAverage(int marks[])
{
    int i, sum = 0;
    float avg;
    for (i = 0; i <= 4; i++) {
        sum += marks[i];
    }
    avg = (sum / 5);
    return avg;
}
```

Passing a Multi-dimensional array to a function

Here again, we will only pass the name of the array as argument.

```
#include<stdio.h>


void displayArray(int arr[3][3]);


int main()
{
    int arr[3][3], i, j;
    printf("Please enter 9 numbers for the array: \n");
    for (i = 0; i < 3; ++i)
    {
        for (j = 0; j < 3; ++j)
        {
            scanf("%d", &arr[i][j]);
        }
    }
    // passing the array as argument
    displayArray(arr);
    return 0;
}
void displayArray(int arr[3][3])
{
    int i, j;
    printf("The complete array is: \n");
    for (i = 0; i < 3; ++i)
    {
        // getting cursor to new line
        printf("\n");
        for (j = 0; j < 3; ++j)
        {
            // \t is used to provide tab space
            printf("%d\t", arr[i][j]);
        }
    }
}
```

Please enter 9 numbers for the array:

1

2

3

4

5

6

7

8

9

The complete array is:

1 2 3

4 5 6

7 8 9

U.N.S.I.E.T, VEER BAHADUR SINGH PURVANCHAL UNIVERSITY, JAUNPUR
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

EC/EI/EE- 2nd Sem

Subject- PROGRAMMING FOR PROBLEM SOLVING

RAVI KANT YADAV
(Assistant Professor)

UNIT-IV

NOTES For

BASIC ALGORITHM

VEER BAHADUR SINGH PURVANCHAL UNIVERSITY, JAUNPUR

(A UP State University Celebrating 25 Years of Excellence)

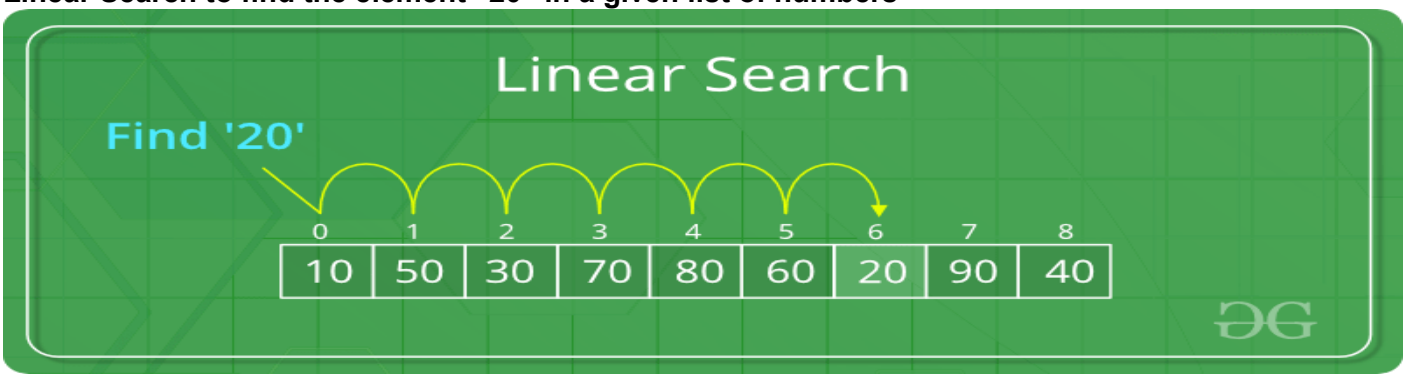
SEARCHING AND BASIC SORTING ALGORITHMS

Searching Algorithms

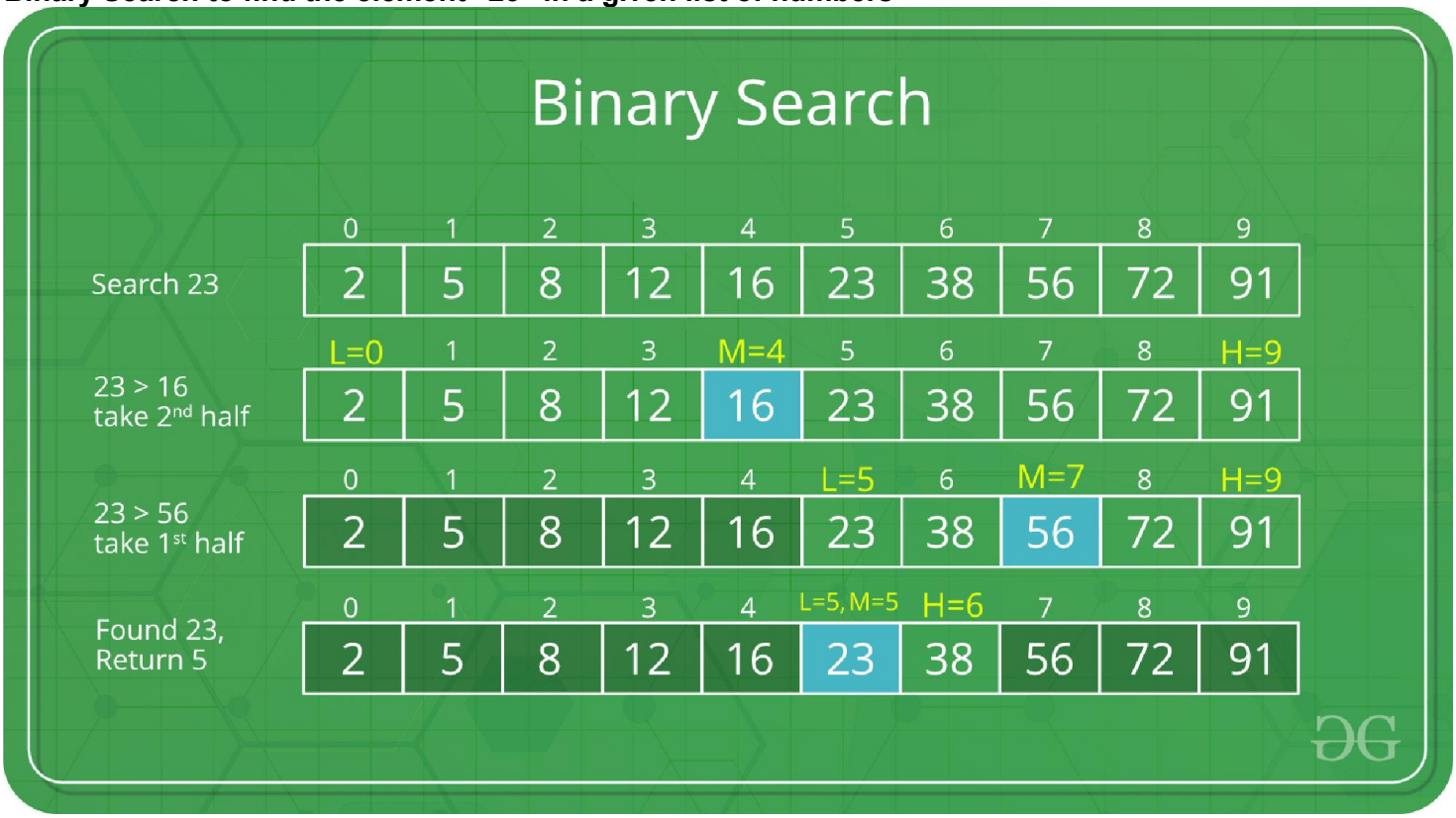
Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Linear Search to find the element "20" in a given list of numbers



Binary Search to find the element "23" in a given list of numbers



Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

Examples :

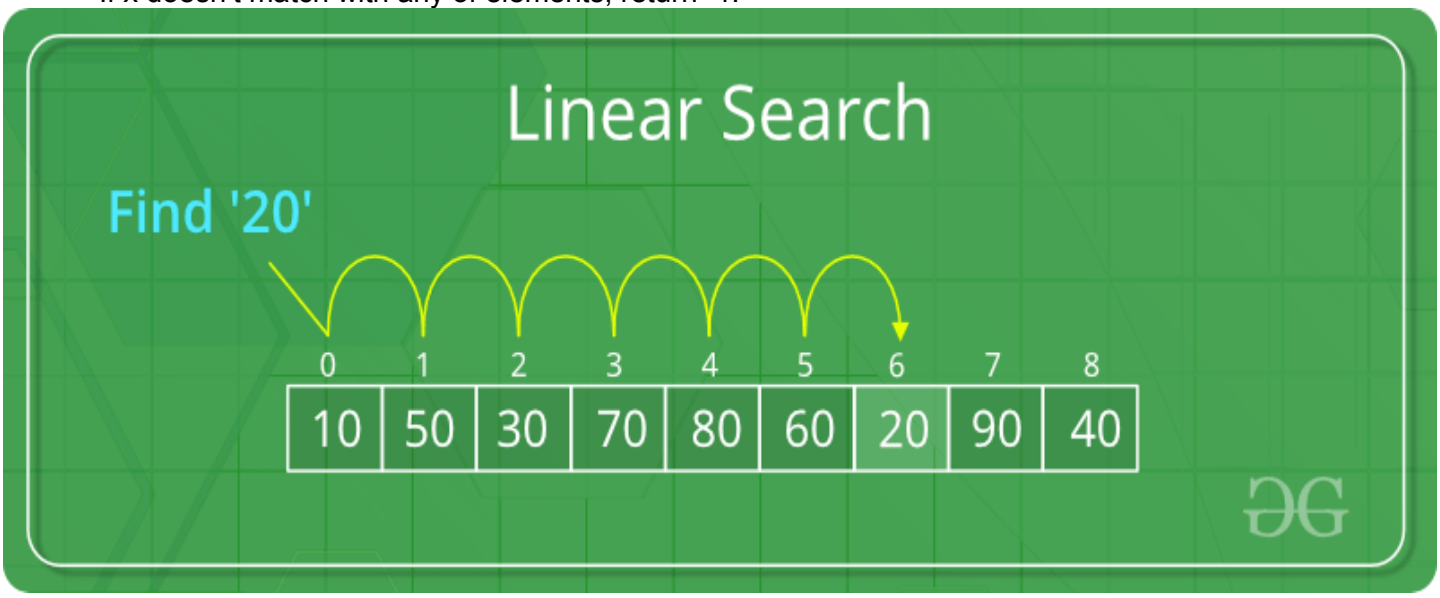
```
Input : arr[] = {10, 20, 80, 30, 60, 50,
                110, 100, 130, 170}
```

```
        x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
                 110, 100, 130, 170}
         x = 175;
Output : -1
Element x is not present in arr[].
```

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

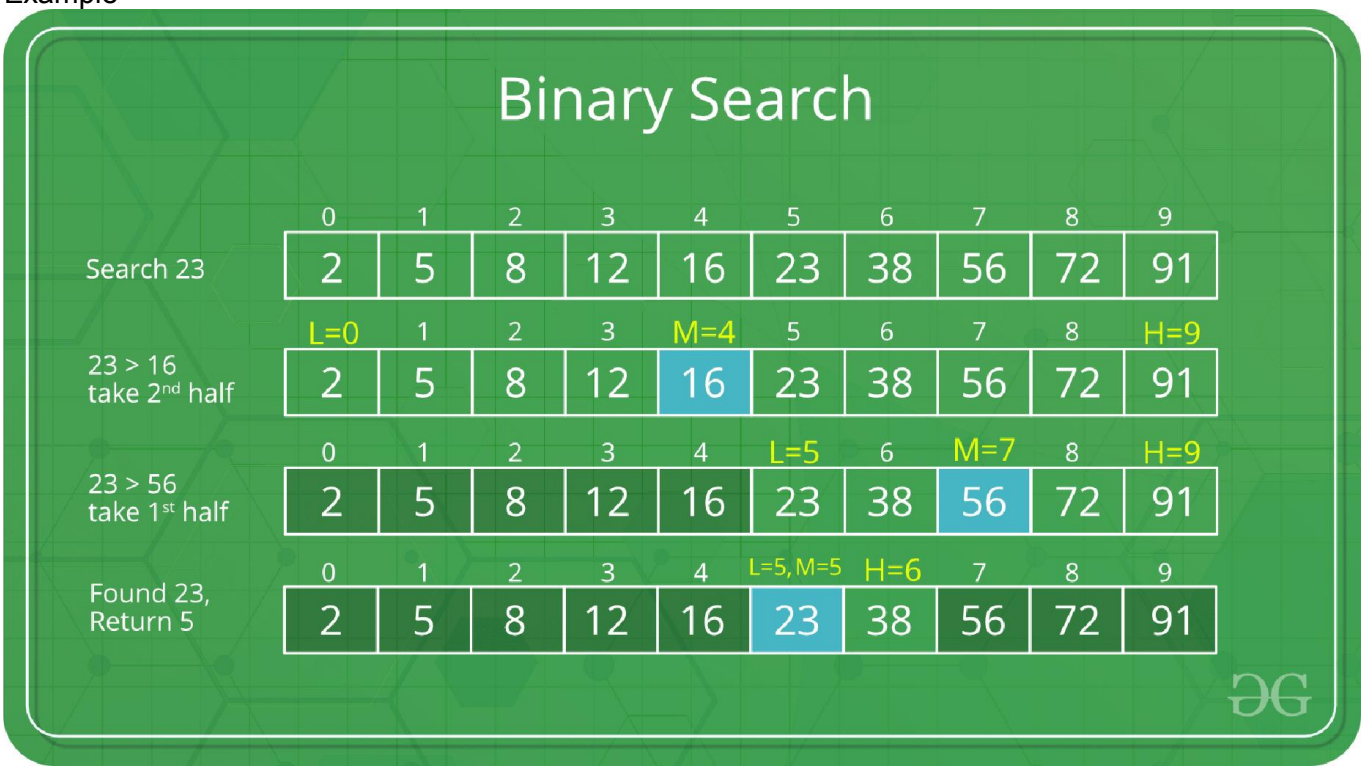
- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.



Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
A simple approach is to do **linear search.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.
Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

For example: The below list of characters is sorted in increasing order of their ASCII values. That is, the character with lesser ASCII value will be placed first than the character with higher ASCII value.

g e e k s f o r g e e k s =====> e e e e f g g k k o r s s

Input Output

SELECTION SORT

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

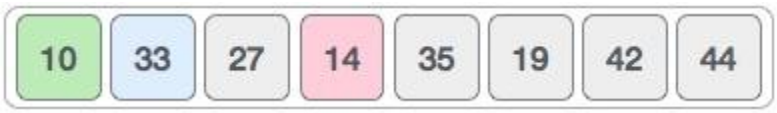
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

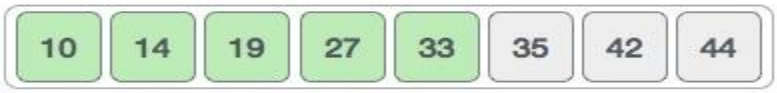
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

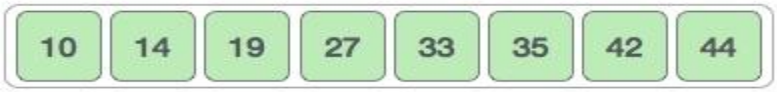
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.
Following is a pictorial depiction of the entire sorting process −





Now, let us learn some programming aspects of selection sort.

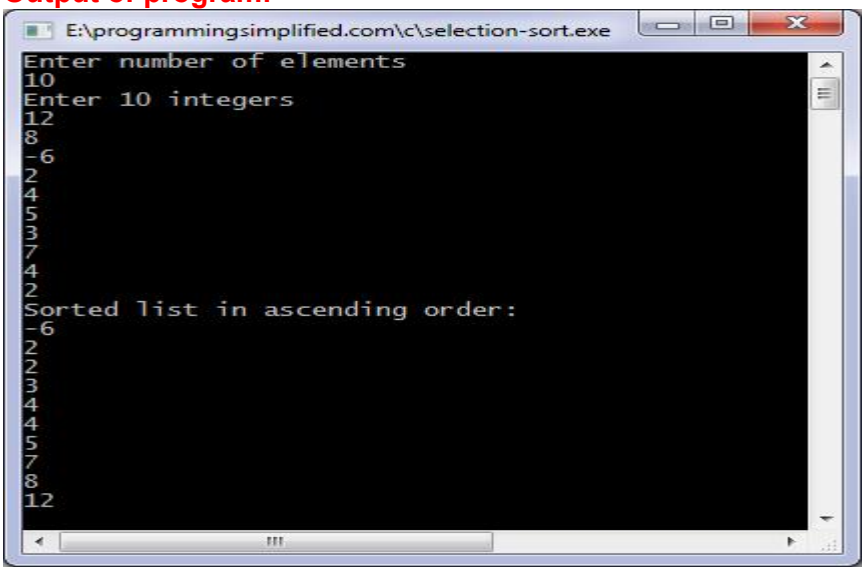
Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Selection sort program in C

```
#include <stdio.h>
int main()
{
  int array[100], n, c, d, position, t;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0; c < (n - 1); c++) // finding minimum element (n-1) times
  {
    position = c;
    for (d = c + 1; d < n; d++)
    {
      if (array[position] > array[d])
        position = d;
    }
    if (position != c)
    {
      t = array[c];
      array[c] = array[position];
      array[position] = t;
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
  return 0;
}
```

Output of program:



Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

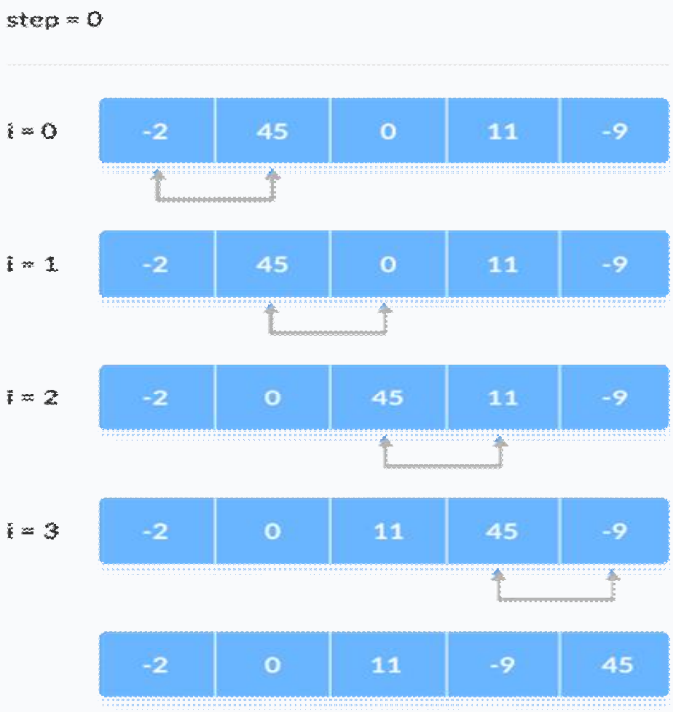
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

How Bubble Sort Works?

1. Starting from the first index, compare the first and the second elements.If the first element is greater than the second element, they are swapped.

   Now, compare the second and the third elements. Swap them if they are not in order.

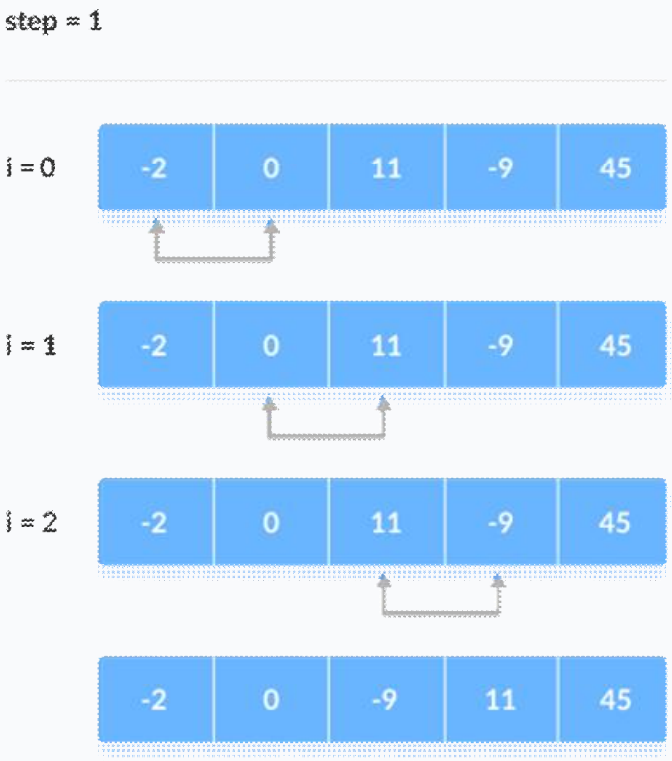
   The above process goes on until the last



   element. Compare the adjacent elements
2. The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.

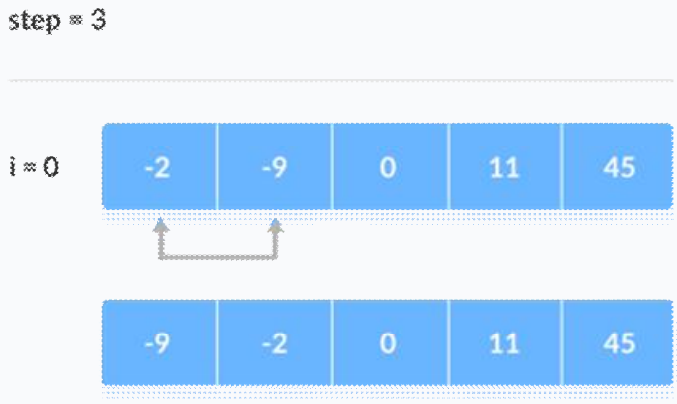
In each iteration, the comparison takes place up to the last unsorted element.

The array is sorted when all the unsorted elements are placed at their correct positions.



Compare the adjacent elements

Compare the adjacent elements



Compare the adjacent elements

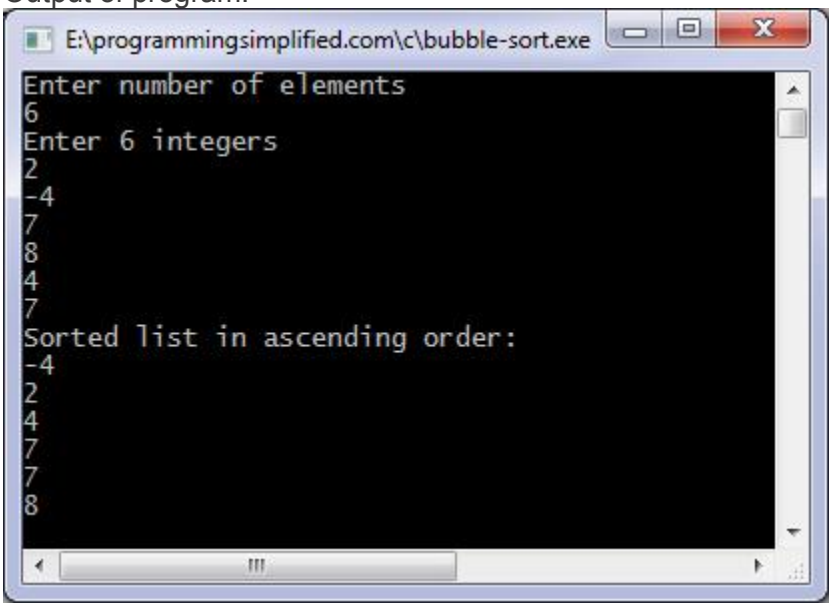
Bubble Sort Algorithm

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Bubble sort program in C

```
/* Bubble sort code */
#include <stdio.h>
int main()
{
  int array[100], n, c, d, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0 ; c < n - 1; c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For decreasing order use < */
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
  return 0;
}
```

Output of program:



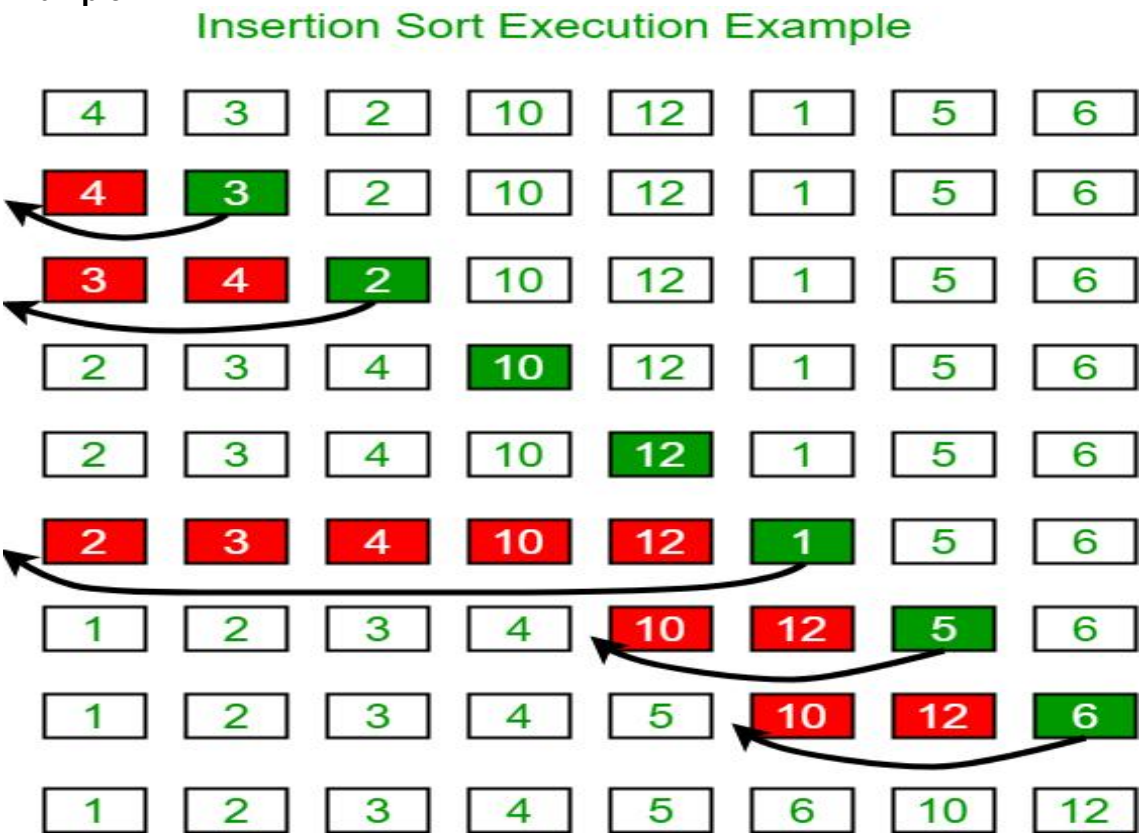
Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm

// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Example:



Another Example:

12, 11, 13, 5, 6
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)
i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6
i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
11, 12, 13, 5, 6
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6
i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

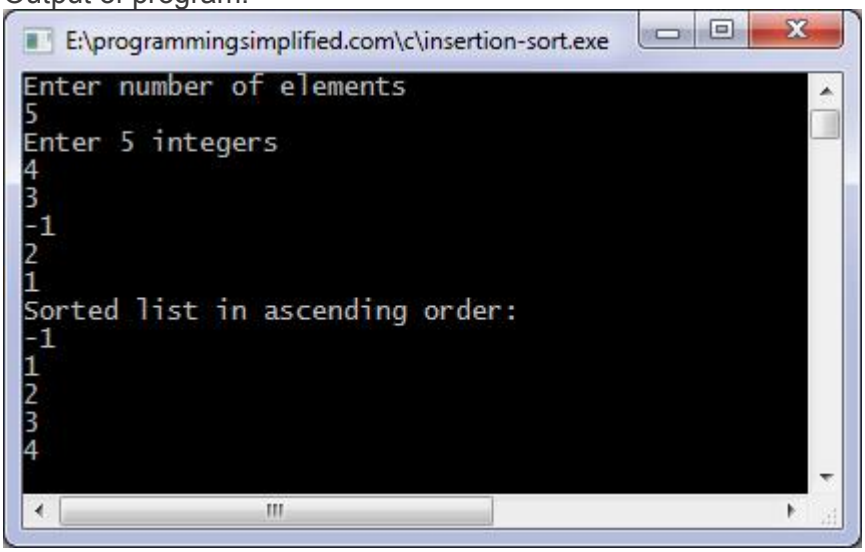
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Insertion sort algorithm implementation in C

```c
/* Insertion sort ascending order */
#include <stdio.h>
int main()
{
  int n, array[1000], c, d, t, flag = 0;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 1 ; c <= n - 1; c++) {
    t = array[c];
    for (d = c - 1 ; d >= 0; d--) {
      if (array[d] > t) {
        array[d+1] = array[d];
        flag = 1;
      }
      else
        break;
    }
    if (flag)
      array[d+1] = t;
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
  }
  return 0;
}
```

Output of program:



Analysis of Algorithms | (NOTATION AND COMPLEXITY)

Complexity of Algorithm

It is very convenient to classify algorithm based on the relative amount of time or relative amount of space they required and specify the growth of time/space requirement as a function of input size.

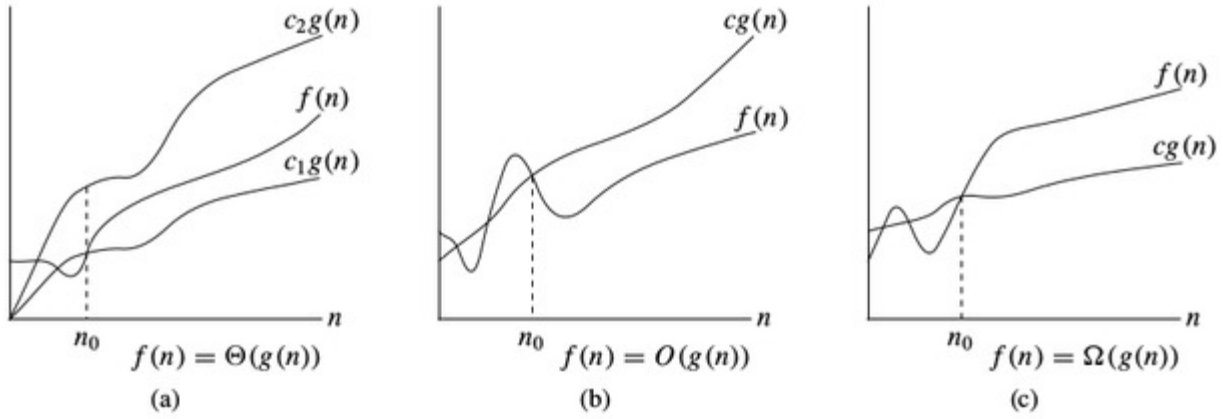
1. **Time Complexity:** Running time of a program as a function of the size of the input.
2. **Space Complexity:** Some forms of analysis could be done based on how much space an algorithm needs to complete its task. This space complexity analysis was critical in the early days of computing when storage space on the computer was limited. When considering this algorithm are divided into those that need extra space to do their work and those that work in place.

But now a day's problem of space rarely occurs because space on the computer (internal or external) is enough.

Broadly, we achieve the following types of analysis -

- **Worst-case: f (n)** defined by the maximum number of steps taken on any instance of size **n**.
- **Best-case: f (n)** defined by the minimum number of steps taken on any instance of size **n**.
- **Average case: f (n)** defined by the average number of steps taken on any instance of size **n**.

- Sometimes, there are more than one way to solve a problem. We need to learn how to compare the performance different algorithms and choose the best one to solve a particular problem. While analyzing an algorithm, we mostly consider time complexity and space complexity. Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.
- Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.
- Lets start with a simple example. Suppose you are given an array A and an integer x and you have to find if x exists in array A.
- **Order of growth** is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.
- O**-notation:**
  To denote asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n)) (pronounced “big-oh of g of n”) the set of functions:
  $O(g(n))= \{ f(n) :$ there exist positive constants c and n0 such that $0 \le f(n) \le c*g(n)$ for all $n \ge n0 \}$
- Ω**-notation:**
  To denote asymptotic lower bound, we use Ω-notation. For a given function g(n), we denote by Ω(g(n)) (pronounced “big-omega of g of n”) the set of functions:
  $\Omega(g(n))= \{ f(n) :$ there exist positive constants c and n0 such that $0 \le c*g(n) \le f(n)$ for all $n \ge n0 \}$
- Θ**-notation:**
  To denote asymptotic tight bound, we use Θ-notation. For a given function g(n), we denote by Θ(g(n)) (pronounced “big-theta of g of n”) the set of functions:
  $\Theta(g(n))= \{ f(n) :$ there exist positive constants c1,c2 and n0 such that $0 \le c1*g(n) \le f(n) \le c2*g(n)$ for all $n>n0 \}$



- While analysing an algorithm, we mostly consider O-notation because it will give us an upper limit of the execution time i.e. the execution time in the worst case.