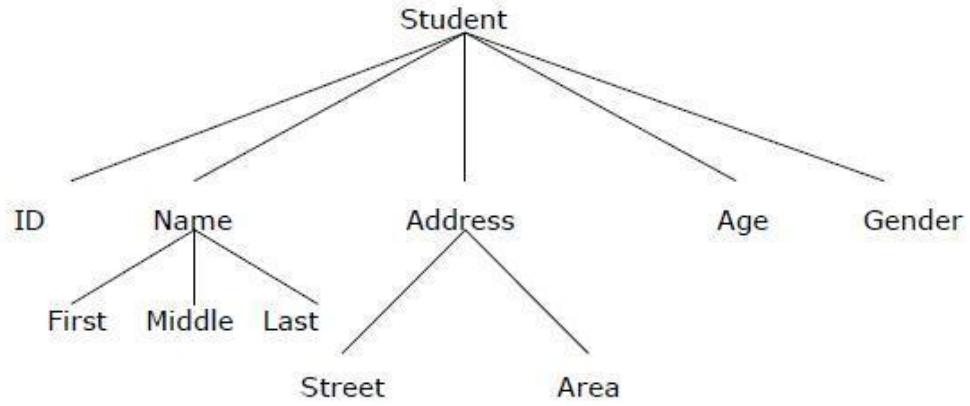
Unit-1

1.1 BASIC TERMINOLOGY: ELEMENTARY DATA

ORGANIZATION 1.1.1 Data and Data Item

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of values. Data items that are divided into sub items are group items; those that are not are called elementary items. For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item.



In the above example (ID, Age, Gender, First, Middle, Last, Street, Area) are elementary data items, whereas (Name, Address) are group data items.

1.1.2 Data Type

Data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. It is of two types: Primitive and non-primitive data type. Primitive data type is the basic data type that is provided by the programming language with built -in support. This data type is native to the language and is supported by machine directly while non-primitive data type is derived from primitive data type. For example- array, structure etc.

1.1.3 Variable

It is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents and these may change during the course of program execution.

1.1.4 Record

Collection of related data items is known as record. The elements of records are usually called fields or members. Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

1.1.5 Program

A sequence of instructions that a computer can interpret and execute is termed as program.

1.1.6 Entity

An entity is something that has certain attributes or properties which may be assigned some values. The values themselves may be either numeric or non-numeric.
Example:

Attributes:	**Name**	**Age**	**Gender**	**Social Society number**
Values:	Hamza	20	M	134-24-5533
	Ali Rizwan	23	M	234-9988775
	Fatima	20	F	345-7766443

1.1.7 Entity Set

An entity set is a group of or set of similar entities. For example, employees of an organization, students of a class etc. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute. The term *"information"* is sometimes used for data with given attributes, of, in other words meaningful or processed data.

1.1.8 Field

A field is a single elementary unit of information representing an attribute of an entity, a record is the collection of field values of a given entity and a file is the collection of records of the entities in a given entity set.

1.1.9 File

File is a collection of records of the entities in a given entity set. For example, file containing records of students of a particular class.

1.1.10 Key

A key is one or more field(s) in a record that take(s) unique values and can be used to distinguish one record from the others.

1.2 ALGORITHM

A well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output. It can also be defined as sequence of computational steps that transform the input into the output.

An algorithm can be expressed in three ways:-
(i) in any natural language such as English, called pseudo code.
(ii) in a programming language or
(iii) in the form of a flowchart.

1.3 EFFICIENCY OF AN ALGORITHM

In computer science, algorithmic efficiency are the properties of an algorithm which relate to the amount of resources used by the algorithm. An algorithm must be analyzed to determine its resource usage. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process.

For maximum efficiency we wish to minimize resource usage. However, the various resources (e.g. time, space) can not be compared directly, so which of two algorithms is considered to be more efficient often depends on which measure of efficiency is being considered as the most important, e.g. is the requirement for high speed, or for minimum memory usage, or for some other measure. It can be of various types:

- **Worst case efficiency**: It is the maximum number of steps that an algorithm can take for any collection of data values.
- **Best case efficiency:** It is the minimum number of steps that an algorithm can take any collection of data values.
- **Average case efficiency**: It can be defined as
  - The efficiency averaged on all possible inputs
  - Must assume a distribution of the input
  - We normally assume uniform distribution (all keys are equally probable) If the input has size *n*, efficiency will be a function *of n*

1.4 POINTER AND DYNAMIC MEMORY ALLOCATION

1.4.1 Pointer

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.
Consider the following example to define a pointer which stores the address of an integer.

```
int n = 10;
int* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type inte
ger.
```

Declaring a pointer

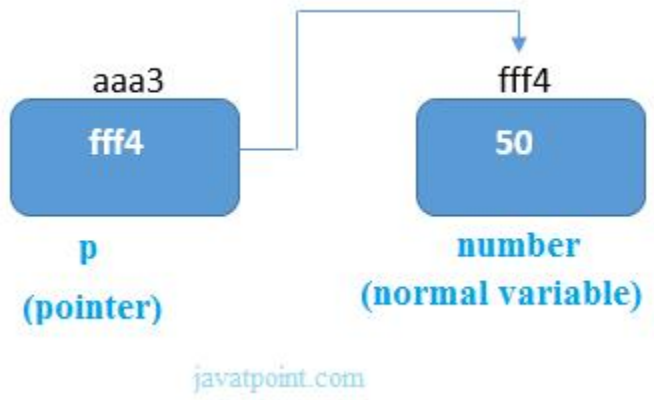
The general form of a pointer variable declaration is-

type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of

the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication.It is also known as indirection pointer used to dereference a pointer.

```
int *a;//pointer to int
char *c;//pointer to char
```



In the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (indirection operator), we can print the value of pointer variable p.

```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore pri
nting p gives the address of number.
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer t
herefore if we print *p, we will get the value stored at the address contained by p.
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

“To access the value stored in the address we use the unary operator (*) that returns the value of the variable located at the address specified by its operand. This is also called Dereferencing.”

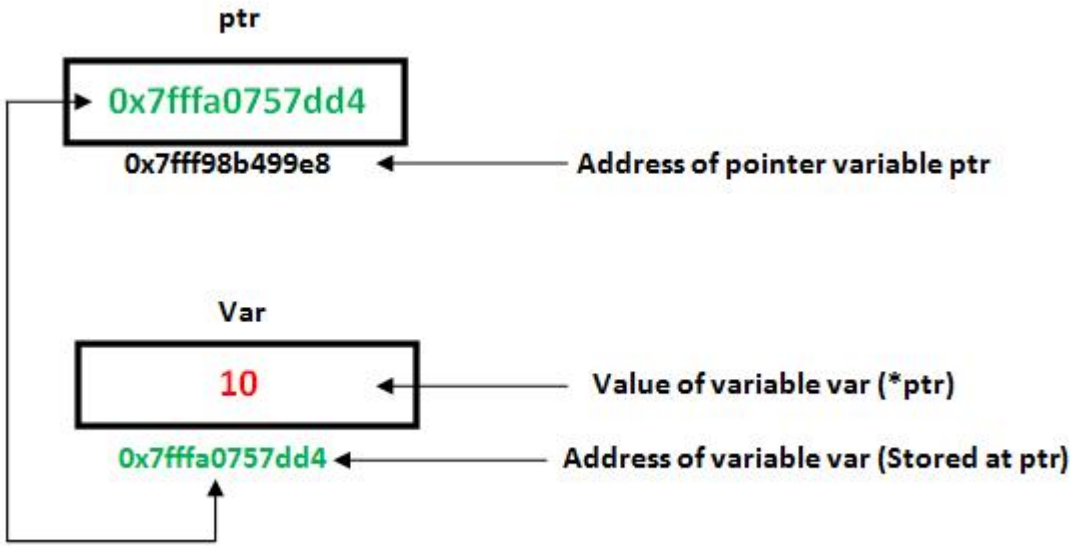
```
// C program to demonstrate use of * for pointers in C
#include <stdio.h>

int main()
{
   // A normal integer variable
   int Var = 10;

   // A pointer variable that holds address of var.
   int *ptr = &Var;

   // This line prints value at address stored in ptr.
   // Value stored is value of variable "var"
   printf("Value of Var = %d\n", *ptr);

   // The output of this line may be different in different
   // runs even on same machine.
   printf("Address of Var = %p\n", ptr);

   // We can also use ptr as lvalue (Left hand
   // side of assignment)
   *ptr = 20; // Value at address is now 20

   // This prints 20
   printf("After doing *ptr = 20, *ptr is %d\n", *ptr);

   return 0;
}
```

Output :
Value of Var = 10

Address of Var = 0x7fffa057dd4

After doing *ptr = 20, *ptr is 20

Below is pictorial representation of above program:



NULL Pointers

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

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

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

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

The value of ptr is 0

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

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and - .

Incrementing a Pointer

Using a pointer in program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

```
#include <stdio.h>
const int MAX = 3;
int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = var;

  for ( i = 0; i < MAX; i++) {

     printf("Address of var[%d] = %x\n", i, ptr );
     printf("Value of var[%d] = %d\n", i, *ptr );

     /* move to the next location */
     ptr++;
  }

  return 0;}
```

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

```
Address of var[0] = 23fe30
```

```
Value of var[0] = 10
Address of var[1] = 23fe34
Value of var[1] = 100
Address of var[2] = 23fe38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

```
#include <stdio.h>
const int MAX = 3;
int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = &var[MAX-1];

  for ( i = MAX; i > 0; i--) {

     printf("Address of var[%d] = %x\n", i-1, ptr );
     printf("Value of var[%d] = %d\n", i-1, *ptr );

     /* move to the previous location */
     ptr--;
  }
 return 0;}
```

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

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
```

Value of var[0] = 10

Pointer with Array

```
#include<stdio.h>

int main()
{
  int arr[5] = { 1, 2, 3, 4, 5 };
  int *ptr = arr;

  printf("%p\n", ptr);
  return 0;
}
```

a pointer ptr that points to the 0th element of the array. Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array. This pointer is useful when talking about multidimensional arrays.

data_type (*var_name)[size_of_array];

Example:

int (*ptr)[10];

Here ptr is pointer that can point to an array of 10 integers. Since subscript have higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is 'pointer to an array of 10 integers'.

Note : The pointer that points to the 0th element of array and the pointer that points to the whole array are totally different. The following program shows this:

```
// C program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include<stdio.h>
int main()
{
    // Pointer to an integer
    int *p;

    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];
```

```
    // Points to 0th element of the arr.
    p = arr;

    // Points to the whole array arr.
    ptr = &arr;

    printf("p = %p, ptr = %p\n", p, ptr);

    p++;
    ptr++;

    printf("p = %p, ptr = %p\n", p, ptr);

    return 0;
}
```

Output:

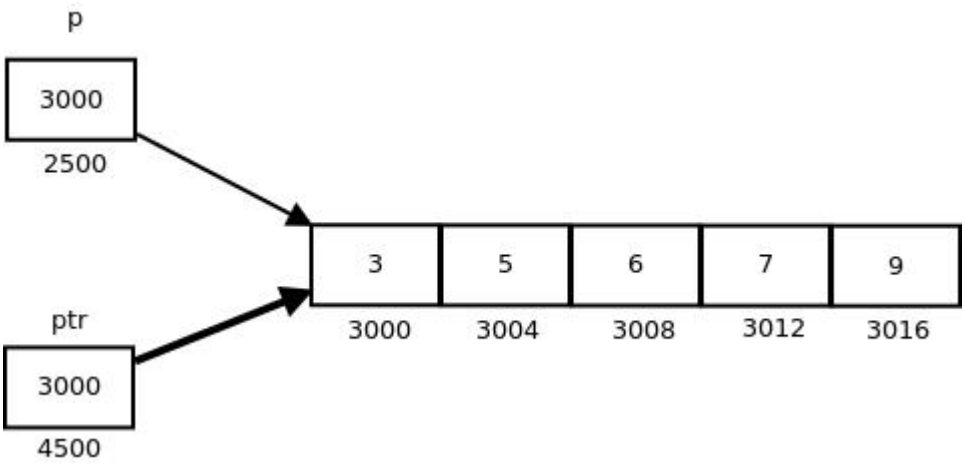
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50

p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64

p: is pointer to 0th element of the array arr, while ptr is a pointer that points to the whole array arr.

- The base type of p is int while base type of ptr is 'an array of 5 integers'.
- We know that the pointer arithmetic is performed relative to the base size, so if we write ptr++, then the pointer ptr will be shifted forward by 20 bytes.

The following figure shows the pointer p and ptr. Darker arrow denotes pointer to an array.



- On dereferencing a pointer expression we get a value pointed to by that pointer expression. Pointer to an array points to an array, so on dereferencing it, we should get the array, and the name of array denotes the base address. So whenever a pointer to an array is dereferenced, we get the base address of the array to which it points.

```c
// C program to illustrate sizes of
// pointer of array
#include<stdio.h>

int main()
{
    int arr[] = { 3, 5, 6, 7, 9 };
    int *p = arr;
    int (*ptr)[5] = &arr;

    printf("p = %p, ptr = %p\n", p, ptr);
    printf("*p = %d, *ptr = %p\n", *p, *ptr);

    printf("sizeof(p) = %lu, sizeof(*p) = %lu\n",
                          sizeof(p), sizeof(*p));
    printf("sizeof(ptr) = %lu, sizeof(*ptr) = %lu\n",
                         sizeof(ptr), sizeof(*ptr));
    return 0;
}
```

Output:

p = 0x7ffde1ee5010, ptr = 0x7ffde1ee5010

*p = 3, *ptr = 0x7ffde1ee5010

sizeof(p) = 8, sizeof(*p) = 4

sizeof(ptr) = 8, sizeof(*ptr) = 20

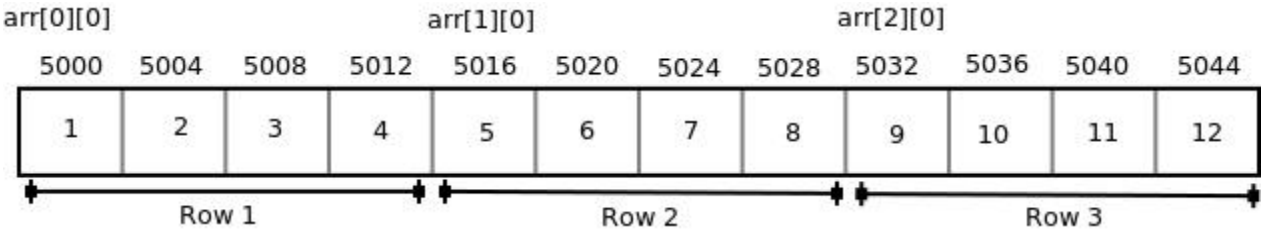
Pointers and two dimensional Arrays: In a two dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element arr[i][j] of the array using the pointer expression *(*(arr + i) + j). Now we'll see how this expression can be derived.

Let us take a two dimensional array arr[3][4]:

int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };

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

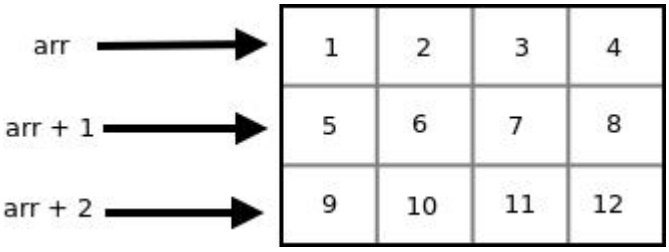
Since memory in a computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words, we can say that 2-D dimensional arrays that are placed one after another. So here arr is an array of 3 elements where each element is a 1-D array of 4 integers.

We know that the name of an array is a constant pointer that points to 0th 1-D array and contains address 5000. Since arr is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.

So we can say that arr points to the 0th 1-D array, arr + 1 points to the 1st 1-D array and arr + 2 points to the 2nd 1-D array.



arr	-	Points to 0th element of arr	-	Points to 0th 1-D array - 5000
arr + 1	-	Points to 1th element of arr	-	Points to 1nd 1-D array - 5016
arr + 2	-	Points to 2th element of arr	-	Points to 2nd 1-D array - 5032

In general we can write:

arr + i Points to ith element of arr ->Points to ith 1-D array

- Since arr + i points to ith element of arr, on dereferencing it will get ith element of arr which is of course a 1-D array. Thus the expression *(arr + i) gives us the base address of ith 1-D array.
- We know, the pointer expression *(arr + i) is equivalent to the subscript expression arr[i]. So *(arr + i) which is same as arr[i] gives us the base address of ith 1-D array.
- To access an individual element of our 2-D array, we should be able to access any jth element of ith 1-D array.
- Since the base type of *(arr + i) is int and it contains the address of 0th element of ith 1-D array, we can get the addresses of subsequent elements in the ith 1-D array by adding integer values to *(arr + i).
- For example *(arr + i) + 1 will represent the address of 1st element of 1stelement of ith 1-D array and *(arr+i)+2 will represent the address of 2nd element of ith 1-D array.
- Similarly *(arr + i) + j will represent the address of jth element of ith 1-D array. On dereferencing this expression we can get the jth element of the ith 1-D array.

Pointers and Three Dimensional Arrays

In a three dimensional array we can access each element by using three subscripts. Let us take a 3-D array-

int arr[2][3][2] = { {{5, 10}, {6, 11}, {7, 12}}, {{20, 30}, {21, 31}, {22, 32}} };

We can consider a three dimensional array to be an array of 2-D array i.e each element of a 3-D array is considered to be a 2-D array. The 3-D array arr can be considered as an array consisting of two elements where each element is a 2-D array. The name of the array arr is a pointer to the 0th 2-D array.

arr	**Points to 0^{th} 2-D array.**
arr + i	**Points to i^{th} 2-D array.**
***(arr + i)**	**Gives base address of i^{th} 2-D array, so points to 0^{th} element of i^{th} 2-D array, each element of 2-D array is a 1-D array, so it points to 0^{th} 1-D array of i^{th} 2-D array.**
***(arr + i) + j**	**Points to j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j)**	**Gives base address of j^{th} 1-D array of i^{th} 2-D array so it points to 0^{th} element of j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j) + k**	**Reprents the value of j^{th} element of i^{th} 1-D array.**
***(*(arr + i) + j) + k)**	**Gives the value of k^{th} element of j^{th} 1-D array of i^{th} 2-D array.**

Thus the pointer expression *(*(*(arr + i) + j) + k) is equivalent to the subscript expression arr[i][j][k].

We know the expression *(arr + i) is equivalent to arr[i] and the expression *(*(arr + i) + j) is equivalent arr[i][j]. So we can say that arr[i] represents the base address of ith 2-D array and arr[i][j] represents the base address of the jth 1-D array.

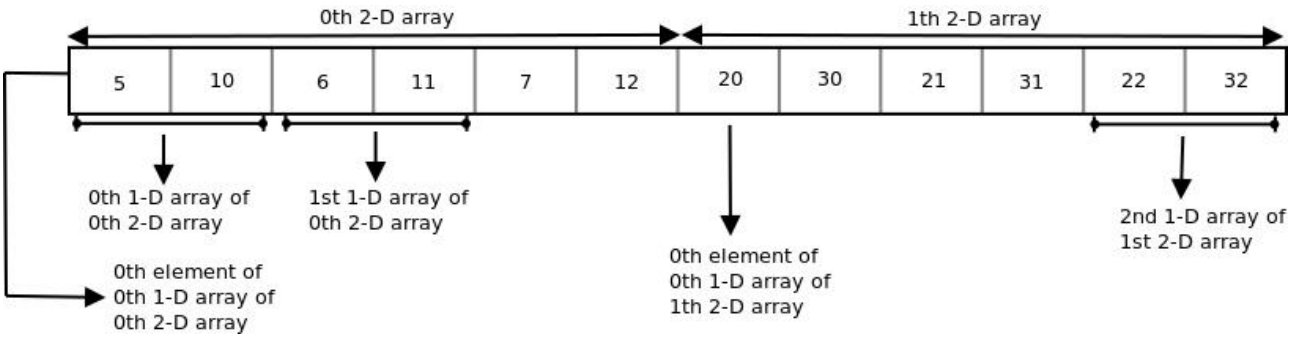
```
// C program to print the elements of 3-D
// array using pointer notation
#include<stdio.h>
int main()
{
  int arr[2][3][2] = {
                  {
                    {5, 10},
                    {6, 11},
                    {7, 12},
                  },
                  {
                    {20, 30},
                    {21, 31},
                    {22, 32},
                  }
                 };
  int i, j, k;
```

```
 for (i = 0; i < 2; i++)
 {
  for (j = 0; j < 3; j++)
  {
    for (k = 0; k < 2; k++)
      printf("%d\t", *(*(*(arr + i) + j) +k));
    printf("\n");
  }
 }

 return 0;
}
```

Output:

5 10
6 11
7 12
20 30
21 31
22 32

The following figure shows how the 3-D array used in the above program is stored in memory.

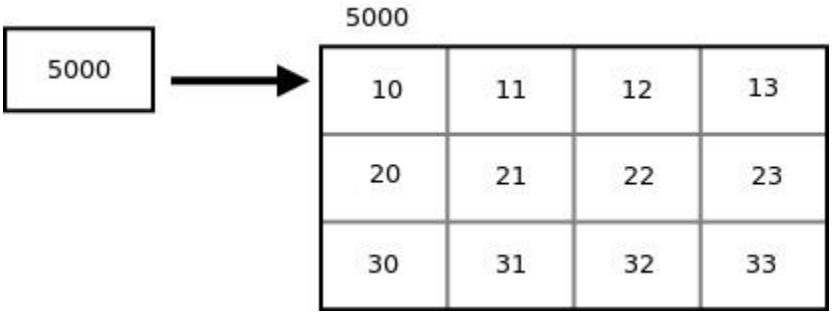


Subscripting Pointer to an Array

Suppose arr is a 2-D array with 3 rows and 4 columns and ptr is a pointer to an array of 4 integers, and ptr contains the base address of array arr.

```
int arr[3][4] = {{10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33}};
int (*ptr)[4];
```

ptr = arr;



Since ptr is a pointer to an array of 4 integers, ptr + i will point to ith row. On dereferencing ptr + i, we get base address of ith row. To access the address of jth element of ith row we can add j to the pointer expression *(ptr + i). So the pointer expression *(ptr + i) + j gives the address of jth element of ith row and the pointer expression *(*(ptr + i)+j) gives the value of the jth element of ith row.

We know that the pointer expression *(*(ptr + i) + j) is equivalent to subscript expression ptr[i][j]. So if we have a pointer variable containing the base address of 2-D array, then we can access the elements of array by double subscripting that pointer variable.

```
// C program to print elements of a 2-D array
// by scripting a pointer to an array
#include<stdio.h>

int main()
{
  int arr[3][4] = {
                    {10, 11, 12, 13},
                    {20, 21, 22, 23},
                    {30, 31, 32, 33}
                  };
  int (*ptr)[4];
  ptr = arr;
  printf("%p %p %p\n", ptr, ptr + 1, ptr + 2);
  printf("%p %p %p\n", *ptr, *(ptr + 1), *(ptr + 2));
  printf("%d %d %d\n", **ptr, *(*(ptr + 1) + 2), *(*(ptr + 2) + 3));
  printf("%d %d %d\n", ptr[0][0], ptr[1][2], ptr[2][3]);
  return 0;
}
```

Output:

0x7ffead967560 0x7ffead967570 0x7ffead967580

0x7ffead967560 0x7ffead967570 0x7ffead967580

10 22 33

10 22 33

Function Pointer in C

like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Output:

```
Value of a is 10
```

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```c
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    void (*fun_ptr)(int) = fun;  // & removed

    fun_ptr(10);  // * removed

    return 0;
}
```

Output:
Value of a is 10

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

```c
#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
```

```
            "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150
```

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.
For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
```

1.4.2 Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming.

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

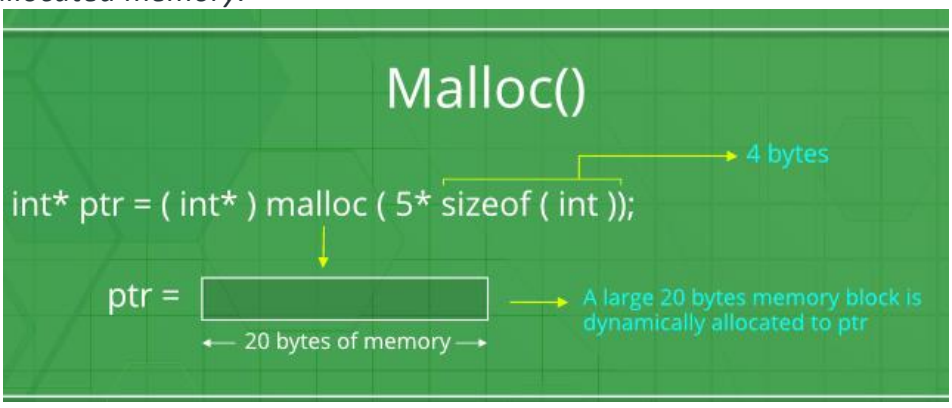
C malloc() method

The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax:

ptr = (cast-type*) malloc(byte-size)**For Example:**

ptr = (int*) malloc(100 * sizeof(int));
Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



If space is insufficient, allocation fails and returns a NULL pointer.

```c
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

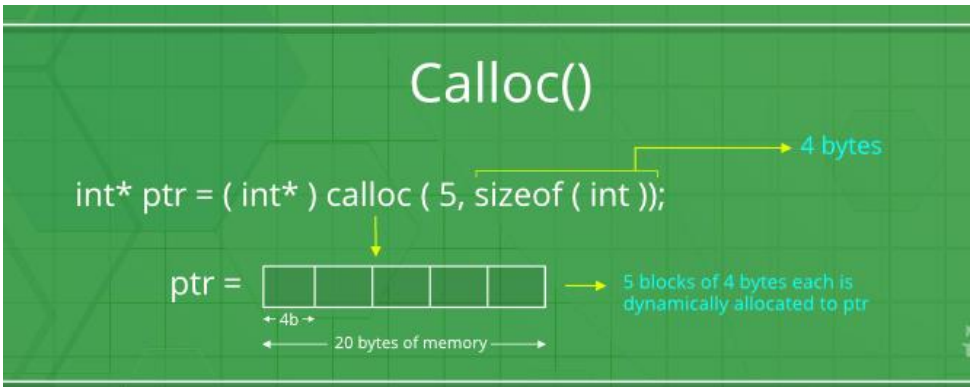
Output:

```
Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

C calloc() method

1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

Syntax:-
ptr = (cast-type*)calloc(n, element-size);
here, n is the no. of elements and element-size is the size of each element.



If space is insufficient, allocation fails and returns a NULL pointer.

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
```

```
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Output:

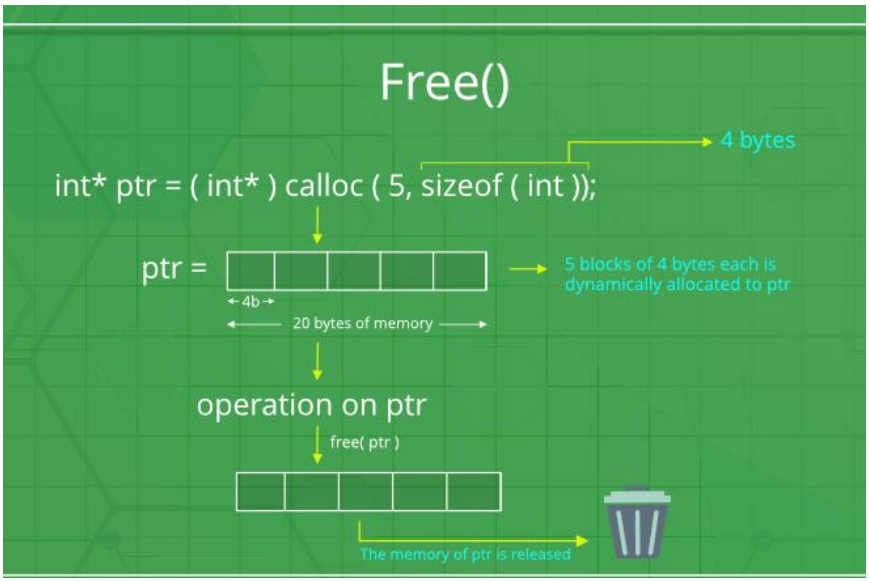
```
Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

C free() method

“free” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```



Example:

C

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
```

```
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");

        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```

Output:

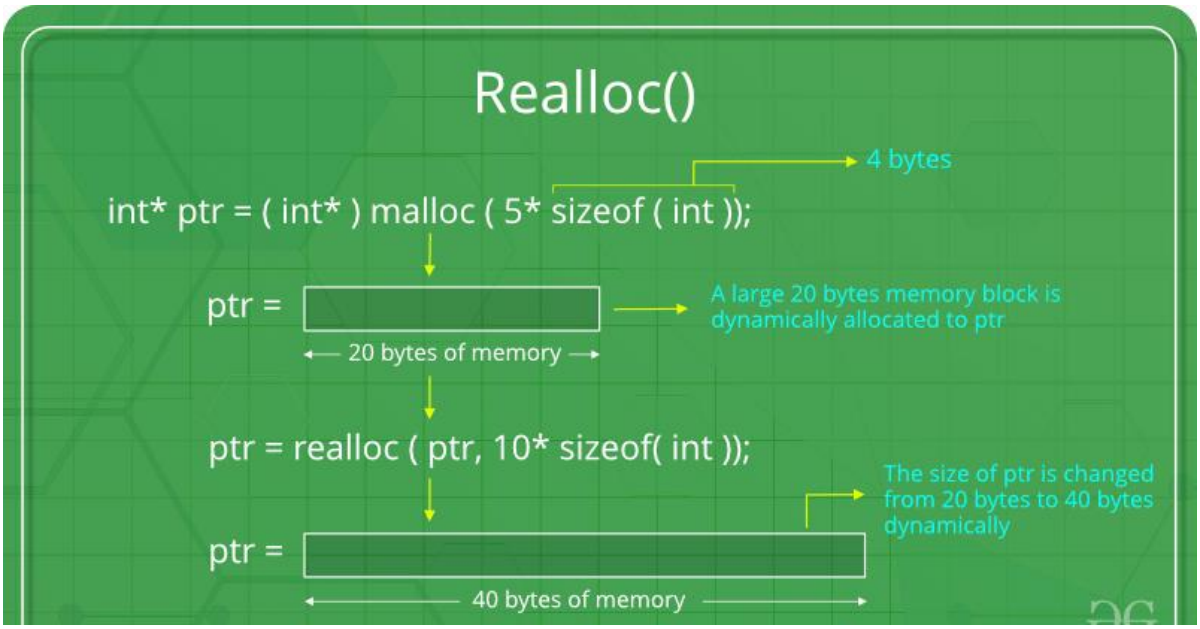
```
Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.
```

C realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax:

```
ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.
```



If space is insufficient, allocation fails and returns a NULL pointer.

Example:

C

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
```

```
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

Output:

```
Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

```
Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

1.5 DATA STRUCTURE

In computer science, a data structure is a particular way of storing and organizing data in a computer's memory so that it can be used efficiently. Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.

Need of data structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

1.5.1 Selecting a data structure

Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported. Quantify resource constraint for each operation
- Select the data structure that best meets these requirements.
- Each data structure has cost and benefits. Rarely is one data structure better than other in all situations. A data structure require :
    - Space for each item it stores
    - Time to perform each basic operation
    - Programming effort.

Each problem has constraints on available time and space. Best data structure for the task requires careful analysis of problem characteristics.

1.5.2 Type of data structure

1.5.2.1 Static data structure

A data structure whose organizational characteristics are invariant throughout its lifetime. Such structures are well supported by high-level languages and familiar examples are arrays and records. The prime features of static structures are

(a) None of the structural information need be stored explicitly within the elements – it is often

held in a distinct logical/physical header;
(b) The elements of an allocated structure are physically contiguous, held in a single segment of memory;
(c) All descriptive information, other than the physical location of the allocated structure, is determined by the structure definition;
(d) Relationships between elements do not change during the lifetime of the structure.

1.5.2.2 Dynamic data structure

A data structure whose organizational characteristics may change during its lifetime. The adaptability afforded by such structures, e.g. linked lists, is often at the expense of decreased efficiency in accessing elements of the structure. Two main features distinguish dynamic structures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating it logically to other elements of the structure. Secondly, using a single block of contiguous storage is often not appropriate, and hence it is necessary to provide some storage management scheme at run-time.

1.5.2.3 Linear Data Structure

A data structure is said to be linear if its elements form any sequence. There are basically two ways of representing such linear structure in memory.
a) One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays**.**
b) The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists**.**
The common examples of linear data structure are arrays, queues, stacks and linked lists.

1.5.2.4 Non-linear Data Structure

This structure is mainly used to represent data containing a hierarchical relationship between elements. E.g. graphs, family trees and table of contents.

1.6 A BRIEF DESCRIPTION OF DATA STRUCTURES

1.6.1 Array

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2, 3 . . . . . *n*. if we choose the name **A** for the array, then the elements of **A** are denoted by subscript notation

A 1,**A**2, **A** 3 . . . .**A**n

or by the parenthesis notation

A (1), A (2), A (3) . . . . . A (n)

or by the bracket notation

A [1], A [2], A [3] . . . . . A [n]

Example:
A linear array **A[8]** consisting of numbers is pictured in following figure.

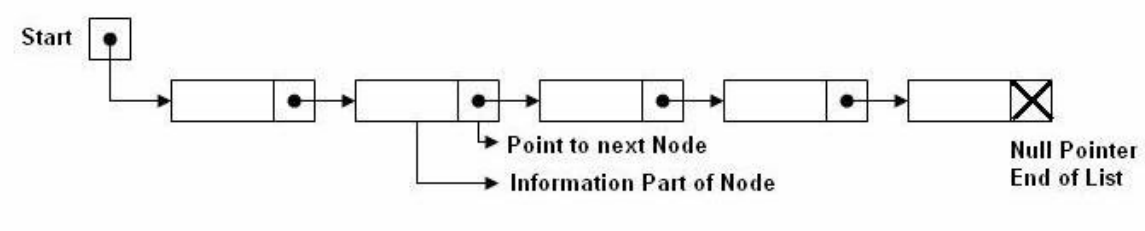
1.6.2 Linked List

A linked list or one way list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. Each node is divided into two parts:

- The first part contains the information of the element/node
- The second part contains the address of the next node (link /next pointer field) in the list.

There is a special pointer Start/List contains the address of first node in the list. If this special pointer contains null, means that List is empty.

Example:

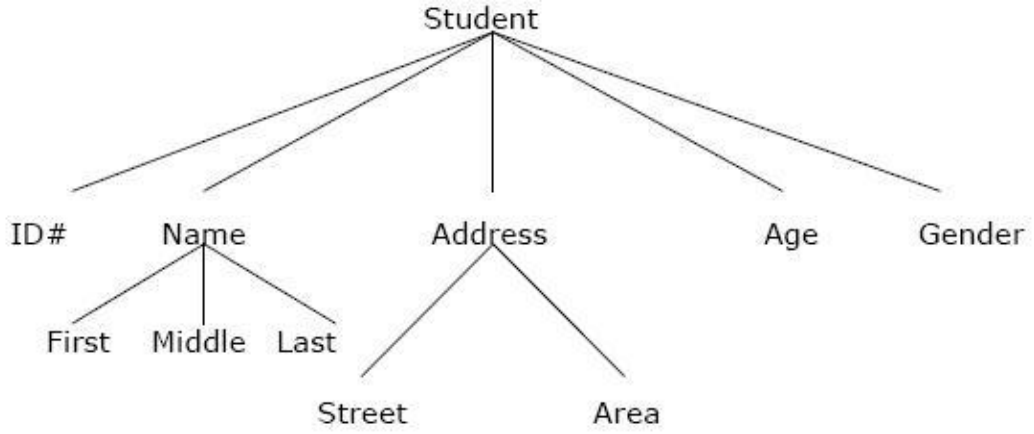


Note:-

- According to Access strategies Linked list is a linear one. According to Storage Linked List is a Non-linear one.
- It actually depends on where you intend to apply linked lists. If you based it on storage, a linked list is considered non-linear. On the other hand, if you based it on access strategies, then a linked list is considered linear.

1.6.3 Tree

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or, simply, a tree.



1.6.4 Graph

Data sometimes contains a relationship between pairs of elements which is not necessarily hierarchical in nature, e.g. an airline flights only between the cities connected by lines. This data structure is called Graph.

1.6.5 Queue

A queue, also called FIFO system, is a linear list in which deletions can take place only at one end of the list, the Font of the list and insertion can take place only at the other end Rear.

1.6.6 Stack

It is an ordered group of homogeneous items of elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).

1.7 DATA STRUCTURES OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely in the frequency with which specific operations are performed.
The following four operations play a major role in this text:

- **Traversing:** accessing each record/node exactly once so that certain items in

the recordmay be processed. (This accessing and processing is sometimes called “visiting” the record.)

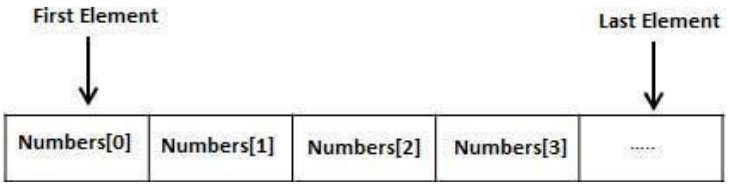
- **Searching:** Finding the location of the desired node with a given key value, or findingthe locations of all such nodes which satisfy one or more conditions.
- **Inserting:** Adding a new node/record to the structure.
- **Deleting:** Removing a node/record from the structure.

1.8 ARRAYS: DEFINITION

C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



The array may be categorized into –

- One dimensional array
- Two dimensional array
- Multidimensional array

1.8.1 Representation of One-Dimensional Array

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
Datatype arrayName [ arraySize ];
```

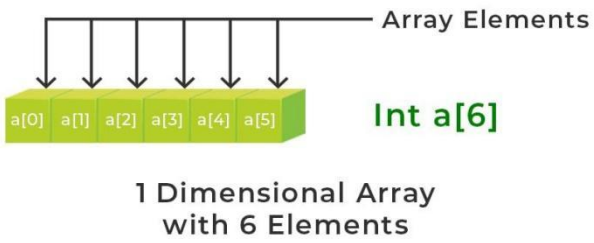
This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Calculating the address of any element In the 1-D array:

A 1-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript that can either represent a row or column index.



1 Dimensional Array with 6 Elements

Address of A[I] = B + W * (I – LB)

I = Subset of element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LB = Lower Limit/Lower Bound of subscript(If not specified assume zero).

Example: Given the base address of an array **A[1300 …… 1900]** as **1020** and the size of each element is 2 bytes in the memory, find the address of **A[1700].**

Solution:

Given:
Base address B = 1020
Lower Limit/Lower Bound of subscript LB = 1300
Storage size of one element store in any array W = 2 Byte
Subset of element whose address to be found I = 1700

Formula used:
*Address of A[I] = B + W * (I – LB)*

Solution:
*Address of A[1700] = 1020 + 2 * (1700 – 1300)*
*= 1020 + 2 * (400)*
= 1020 + 800
Address of A[1700] = 1820

1.8.2 Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
DataType arrayName [ x ][ y ];
```

Where type can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be think as a table which will have x number of rows and y

number of columns. A 2-dimensional array a, which contains three rows and four columns can be shown as below:

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

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Representation of two dimensional arrays in memory

A two dimensional 'm x n' Array A is the collection of m X n elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways-

- Row Major Order: First row of the array occupies the first set of memory locations reserved for the array; Second row occupies the next set, and so forth.

Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

Example: Given an array, **arr[1……10][1……15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order.

Solution:

Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix N = Upper Bound – Lower Bound + 1

= 15 – 1 + 1
= 15

Formula:
*Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))*

Solution:
*Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))*
*= 100 + 1 * ((7) * 15 + (5))*
*= 100 + 1 * (110)*
Address of A[I][J] = 210

- Column Major Order**:**Order elements of first column stored linearly and then comes elements of next column.

Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
M = Number of rows given in the matrix.

Example: Given an array **arr[1……10][1……15]** with a base value of **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.

Solution:

Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix M = Upper Bound – Lower Bound + 1
= 10 – 1 + 1
= 10

Formula: used
*Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))*
*Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))*
*= 100 + 1 * ((5) * 10 + (7))*

*= 100 + 1 * (57)*

Address of A[I][J] = 157

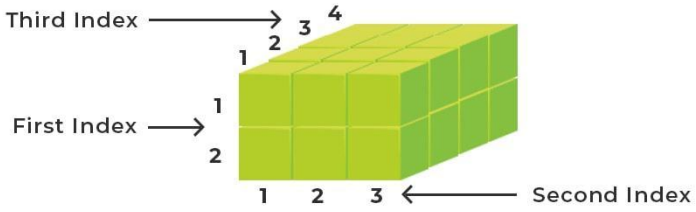
1.8.3 Calculate the address of any element in the 3-D Array:

A **3-Dimensional** array is a collection of 2-Dimensional arrays. It is specified by using three subscripts:

1. Block size
2. Row size
3. Column size

More dimensions in an array mean more data can be stored in that array.

Example:



Three-Dimensional Array
with 24 Elements

3-D array

To find the address of any element in 3-Dimensional arrays there are the following two ways-

- Row Major Order
- Column Major Order

1. Row Major Order:

To find the address of the element using row-major order, use the following formula:

Address of A[i][j][k] = B + W (M * N(i-x) + N *(j-y) + (k-z))

Here:

B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row

y = Lower Bound of Column
z = Lower Bound of Width

Example: Given an array, **arr[1:9, -4:1, 5:10]** with a base value of **400** and the size of each element is **2 Bytes** in memory find the address of element **arr[5][-1][8]** with the help of row-major order?

Solution:

Given:
Row Subset of an element whose address to be found I = 5
Column Subset of an element whose address to be found J = -1
Block Subset of an element whose address to be found K = 8
Base address B = 400
Storage size of one element store in any array(in Byte) W = 2
Lower Limit of row/start row index of matrix x = 1
Lower Limit of column/start column index of matrix y = -4
Lower Limit of blocks in matrix z = 5
M = Upper Bound – Lower Bound + 1 = 1 – (-4) + 1 = 6
N = Upper Bound – Lower Bound + 1 = 10 – 5 + 1 = 6

Formula used:
*Address of[I][J][K] =B + W (M * N(i-x) + N *(j-y) + (k-z))*

Solution:
*Address of[][][] = 400 + 2 * {[6 * 6 * (5 – 1)] + 6 * [(-1 + 4)]} + [8 – 5]*
*= 400 + 2 * ((4 * 6 + 3) * 6 + 3)*
*= 400 + 2 * (165)*
= 730

2. Column Major Order:

To find the address of the element using column-major order, use the following formula:1

Address of A[i][j][k = B + W(M * N(i – x) + M *(k – z) + (j – y))

Here:

B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Example: Given an array **arr[1:8, -5:5, -10:5]** with a base value of **400** and the size of each element is **4 Bytes** in memory find the address of element **arr[3][3][3]** with the help of column-major order?

Solution:

Given:
Row Subset of an element whose address to be found I = 3
Column Subset of an element whose address to be found J = 3
Block Subset of an element whose address to be found K = 3
Base address B = 400
Storage size of one element store in any array(in Byte) W = 4
Lower Limit of row/start row index of matrix x = 1
Lower Limit of column/start column index of matrix y = -5
Lower Limit of blocks in matrix z = -10
M = Upper Bound – Lower Bound + 1 = 5 + 5 + 1 = 11
N = Upper Bound – Lower Bound + 1 = 5 + 10 + 1 = 16

Formula used:
*Address of[i][j][k] = B + W(M * N(i – x) + M * (k – z) + (j – y))*

Solution:
*Address of[3][3][3] = 400 + 4 * {[(3 – 1)] * 16 + [3 + 10]]} * 11 + [3 + 5]*

*= 400 + 4 * ((32 + 13) * 11 + 8)*
*= 400 + 4 * (503)*
= 400 + 2012
= 2412

Applications of Arrays

1) Array stores data elements of the same data type.

2) Maintains multiple variable names using a single name. Arrays help to maintain large data under a single variable name. This avoid the confusion of using multiple variables.

3) Arrays can be used for sorting data elements. Different sorting techniques like Bubble sort, Insertion sort, Selection sort etc use arrays to store and sort elements easily.

4) Arrays can be used for performing matrix operations. Many databases, small and large, consist of one-dimensional and two-dimensional arrays whose elements are records.

5) Arrays can be used for CPU scheduling.

6) Lastly, arrays are also used to implement other data structures like Stacks, Queues, Heaps, Hash tables etc.

Array as Parameter

Example 1: Pass Individual Array Elements

```
#include <stdio.h>
void display(int age1, int age2) {
  printf("%d\n", age1);
  printf("%d\n", age2);
}
int main() {
  int ageArray[] = {2, 8, 4, 12};
  // pass second and third elements to display()
  display(ageArray[1], ageArray[2]);
  return 0;
}
```

Output:

```
8
4
```

Example 2: Pass Arrays to Functions

```
// Program to calculate the sum of array elements by passing to a function
#include <stdio.h>
float calculateSum(float num[]);
int main() {
  float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};

  // num array is passed to calculateSum()
  result = calculateSum(num);
  printf("Result = %.2f", result);
  return 0;
}
float calculateSum(float num[]) {
  float sum = 0.0;

  for (int i = 0; i < 6; ++i) {
    sum += num[i];
  }
  return sum;
}
```

Output

```
Result = 162.50
```

Pass Multidimensional Arrays to a Function

To pass multidimensional arrays to a function, only the name of the array is passed to the function (similar to one-dimensional arrays).

Example 3: Pass two-dimensional arrays

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main() {
  int num[2][2];
  printf("Enter 4 numbers:\n");
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
      scanf("%d", &num[i][j]);
    }
  }

  // pass multi-dimensional array to a function
  displayNumbers(num);
  return 0;
}
void displayNumbers(int num[2][2]) {
  printf("Displaying:\n");
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
      printf("%d\n", num[i][j]);
    }
  }
}
```

Output

```
Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5
```

Basic Operations Following are the basic operations supported by an array.

Traverse − print all the array elements one by one.
Insertion − Adds an element at the given index.
Deletion − Deletes an element at the given index.
Search − Searches an element using the given index or by the value.
Update − Updates an element at the given index

INSERTION
(Inserting into a Linear Array) INSERT (LA, N. K. ITEM) Here LA is a linear array with N elements and K is a positive integer such that KSN. This algorithm inserts an element ITEM into the Kth position in LA.

1. Initialize counter.] Set J := N.
2. Repeat Steps 3 and 4 while J > K.
3. [Move Jth element downward.] Set LA[J + 1] := LA[J].
4. [Decrease counter.] Set J := J-1.
   [End of Step 2 loop.]
5. [Insert element.] Set LA[K]:= ITEM.
6. [Reset N.J Set N = N + 1.
7. Exit.

DELETION
(Deleting from a Linear Array) DELETE(LA, N. K. ITEM) Here LA is a linear array with N elements and K is a positive integer such that K<=N. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].
2. Repeat for J= K to N-1:
[Move J + 1st element upward.] Set LA[J]:= LAIJ + 1].
[End of loop.]
3. [Reset the number N of elements in LA.] Set N:= N - 1.
4. Exit.

SEARCH
1. (Linear Search) LINEAR(DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC:= 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA[N+ 1] := ITEM.
2. [Initialize counter.] Set LOC := 1.
3. [Search for ITEM.]
Repeat while DATA[LOC]!= ITEM:
    Set LOC:= LOC +1.
[End of loop.]

4. [Successful?] If LOC = N + 1, then: Set LOC := 0.
5. Exit.

```
procedure LINEAR_SEARCH (array, key)

  for each item in the array
    if match element == key
      return element's index
    end if
  end for

end procedure
```

2. (Binary Search) BINARY (DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]
   Set BEG LB, END:= UB and MID= INT((BEG+ END)/2).
2. Repeat Steps 3 and 4 while BEG <= END and DATA[MID]!= ITEM.
3. If ITEM <DATA[MID], then:
   Set END:= MID - 1.
   Else:
   Set BEG:= MID + 1.
   [End of If structure.]
4. Set MID:= INT((BEG+ END)/2).
   [End of Step 2 loop.]
5.If DATA[MID] = ITEM, then:
   Set LOC:= MID.
   Else:
   Set LOC: =NULL

[End of If structure.]
6. Exit.

Binary Search Algorithm can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

1. Iteration Method

```
binarySearch(arr, x, low, high)
   repeat till low = high
        mid = (low + high)/2
          if (x == arr[mid])
          return mid

          else if (x > arr[mid]) // x is on the right side
            low = mid + 1

          else              // x is on the left side
            high = mid - 1
```

2. Recursive Method (The recursive method follows the divide and conquers approach)

```
binarySearch(arr, x, low, high)
     if low > high
        return False

     else
        mid = (low + high) / 2
          if x == arr[mid]
          return mid

        else if x > arr[mid]       // x is on the right side
          return binarySearch(arr, x, mid + 1, high)

        else                    // x is on the right side
          return binarySearch(arr, x, low, mid - 1)
```

Sparse Matrix and its representations

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

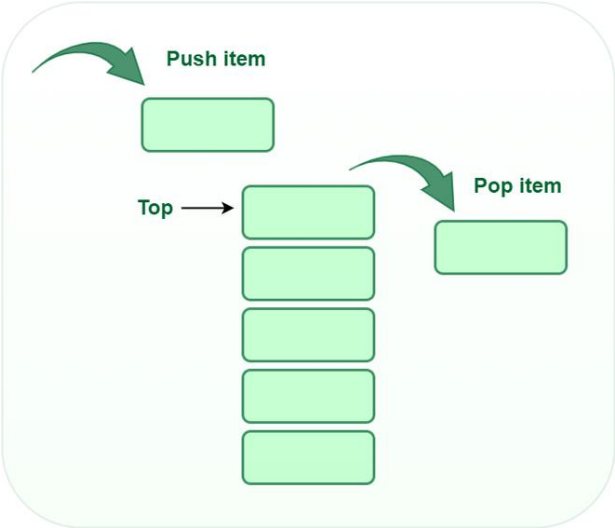
- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

STACK

- In C, a Stack is a linear data structure that follows the LIFO (Last In First Out) approach to perform a series of basic operations like push, pop, peek, and traverse. A Stack can be implemented using an Array or Linked List.
- Element may be inserted or deleted only at one end, called the top of the stack.
- Element are removed in reverse order of that in which they inserted into stack.
- A Stack is an abstract linear data structure serving as a collection of elements that are inserted (push operation) and removed (pop operation) according to the Last in First Out (LIFO) approach. Insertion and deletion happen on the same end (top) in a Stack.



Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations provided to us.

- push() to insert an element into the stack
- pop() to remove an element from the stack
- top() Returns the top element of the stack.
- isEmpty() returns true is stack is empty else false
- size() returns the size of stack

Push:

Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.
1. [Stack already filled?]
If TOP= MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP:= TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP]:=ITEM. [Inserts ITEM in new TOP position.]

Algorithm for push:

```
begin
 if stack is full
    return
 endif
else
```

increment top

stack[top] assign value

end else

end procedure

Pop:

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]

    If TOP= 0, then: Print: UNDERFLOW, and Return.

2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]

3. Set TOP:= TOP 1. [Decreases TOP by 1.]

4. Return.

Algorithm for pop:

begin

if stack is empty

return

endif

else

store value of stack[top]

decrement top

return value

end else

end procedure

Top:

Returns the top element of the stack.

Algorithm for Top:

begin

return stack[top]

end procedure

isEmpty:

Returns true if the stack is empty, else false.

Algorithm for isEmpty:

```
begin
 if top < 1
   return true
 else
   return false
end procedure
```

Infix to Postfix conversion:-

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C*D,D/A** etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)*C(D/(J+D))**.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

Infix Expression

It follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>. Such an expression is termed infix expression. E.g., **A+B**

Postfix Expression

It follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>. E.g., **AB+**

Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push "(" onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
    [End of If]
6. If a right parenthesis is encountered ,then:

    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
    [End of If]
    [End of If]

7. END.

Let's take an example to better understand the algorithm

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Evaluation of Postfix Expressions Using Stack [with C program]

As discussed in , the compiler finds it convenient to evaluate an expression in its postfix form. The virtues of postfix form include elimination of parentheses which signify priority of evaluation and the elimination of the need to observe rules of hierarchy, precedence and associativity during evaluation of the expression.

As **Postfix expression** is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

Evaluation rule of a Postfix Expression states:

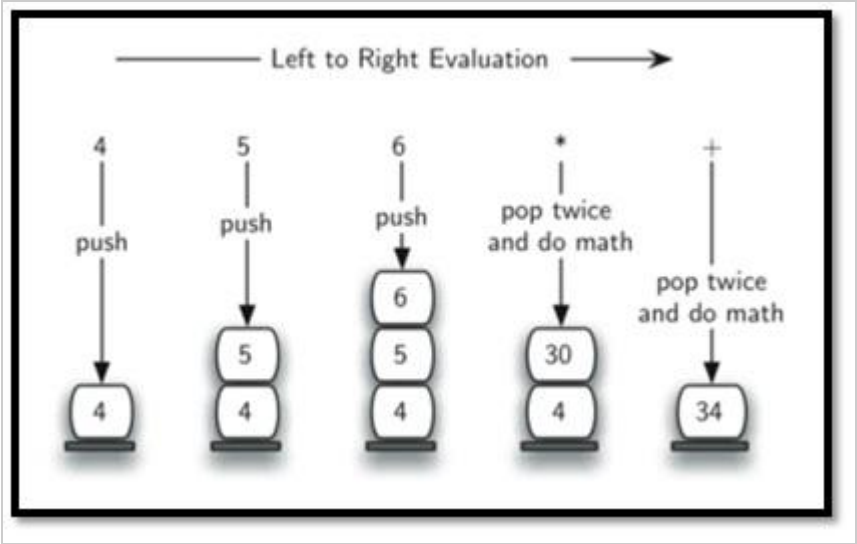
1. While reading the expression from left to right, push the element in the stack if it is an operand.
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

1) Add `)` to postfix expression.
2) Read postfix expression Left to Right until `)` encountered
3) If operand is encountered, push it onto Stack
[End If]
4) If operator is encountered, Pop two elements
i) `A -> Top` element
ii) `B-> Next to Top` element
iii) Evaluate `B` operator `A`
push `B` operator `A` onto Stack
5) Set `result = pop`
6) END

Let's see an example to better understand the algorithm:

Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Result: 34

Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
- Using a recursive algorithm, certain problems can be solved quite easily.
- Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.
- A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.
- Many more recursive calls can be generated as and when required.
- It is essential to know that we should provide a certain case in order to terminate this recursion process.

Need of Recursion

- Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write.
- It has certain advantages over the iteration technique which will be discussed later.
- A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

A Mathematical Interpretation

Let us consider a problem that a programmer has to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 1 to n. So the function simply looks like this,

approach(1) – Simply adding one by one

f(n) = 1 + 2 + 3 +……..+ n

but there is another mathematical approach of representing this,

approach(2) – Recursive adding

f(n) = 1 n=1

f(n) = n + f(n-1) n>1

There is a simple difference between the approach (1) and approach(2) and that is in approach(2) the function " f( ) " itself is being called inside the function, so this phenomenon is named recursion, and the function containing recursion is called recursive function, at the end, this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way.

How are recursive functions stored in memory?

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

What is the base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

In the above example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know the factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

Why Stack Overflow error occurs in recursion?

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
    // wrong base case (it may cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

What is the difference between direct and indirect recursion?

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly.

```
// An example of direct recursion
void directRecFun()
{
    // Some code....

    directRecFun();

    // Some code...
}
// An example of indirect recursion
void indirectRecFun1()
{
    // Some code...
```

```
    indirectRecFun2();

    // Some code...
}
void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

What is the difference between tailed and non-tailed recursion?

A recursive function is tail recursive when a recursive call is the last thing executed by the function.

How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Recursion VS Iteration

SR No.	Recursion	Iteration
1)	Terminates when the base case becomes true.	Terminates when the condition becomes false.
2)	Used with functions.	Used with loops.
3)	Every recursive call needs extra space in the stack	Every iteration does not require any extra space.

memory.

4)	Smaller code size.	Larger code size.

What are the disadvantages of recursive programming over iterative programming?

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than the iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

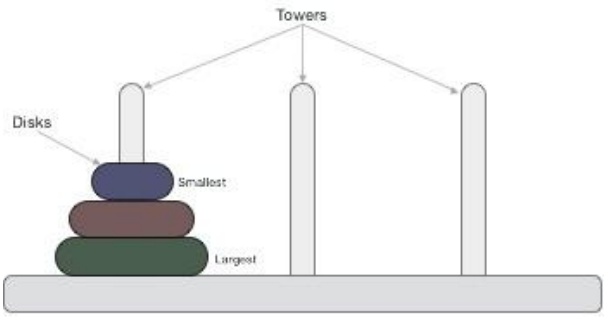
Moreover, due to the smaller length of code, the codes are difficult to understand and hence extra care has to be practiced while writing the code. The computer may run out of memory if the recursive calls are not properly checked.

What are the advantages of recursive programming over iterative programming?

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi.

Tower of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the

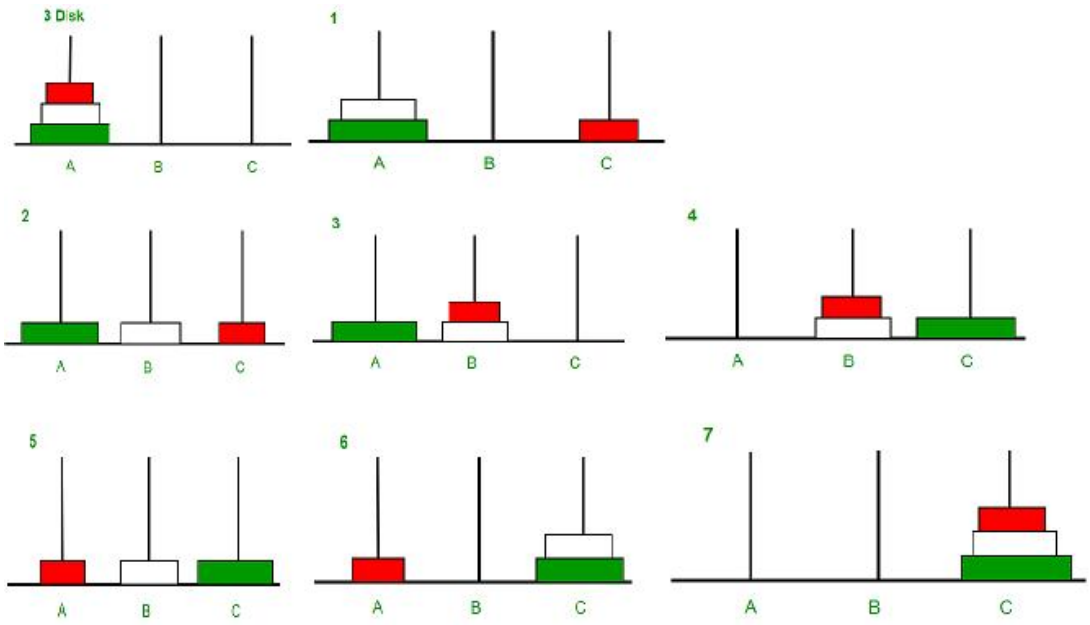
puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Tower of Hanoi puzzle with n disks can be solved in minimum 2^n-1 steps. This presentation shows that a puzzle with 3 disks has taken 2^3 - 1 = 7 steps.



TOWER(N, BEG, AUX, END)
This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If N = 1, then:
   (a) Write: BEG → END.
   (b) Return.
   [End of If structure.]
2. [Move N – 1 disks from peg BEG to peg AUX.]
   Call TOWER(N – 1, BEG, END, AUX).
3. Write: BEG → END.
4. [Move N – 1 disks from peg AUX to peg END.]
   Call TOWER(N – 1, AUX, BEG, END).
5. Return.