

DATA STRUCTURE

UNIT – 1

Introduction:

Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations, Review of Arrays, Structures, Self-Referential Structures, and Unions. Pointers and Dynamic Memory Allocation Functions. Representation of Linear Arrays in Memory, dynamically allocated arrays. Performance analysis of an algorithm and space and time complexities

Data Structures

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

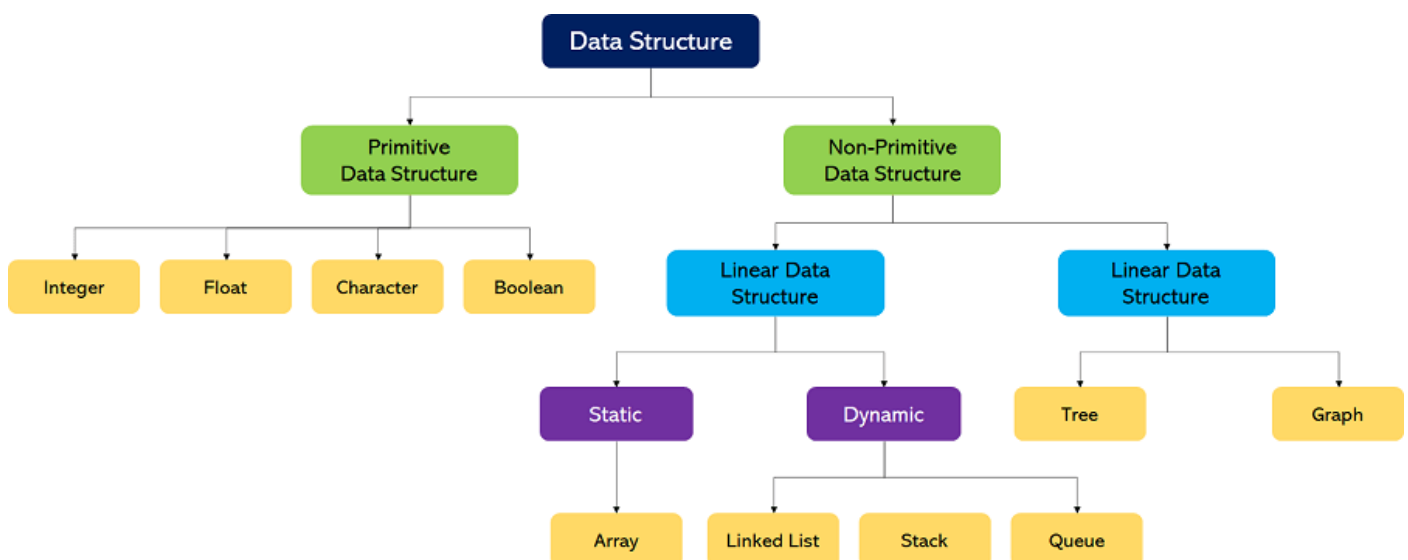
A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data.

Classification of Data Structures

A Data Structure delivers a structured set of variables related to each other in various ways. It forms the basis of a programming tool that signifies the relationship between the data elements and allows programmers to process the data efficiently.

We can classify Data Structures into two categories:

1. Primitive Data Structure
2. Non-Primitive Data Structure



Primitive Data Structures

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.

3. Basic data types like **Integer**, **Float**, **Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -

a. Linear Data Structures

b. Non-Linear Data Structures

What is the Linear data structure?

A linear **data structure** is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

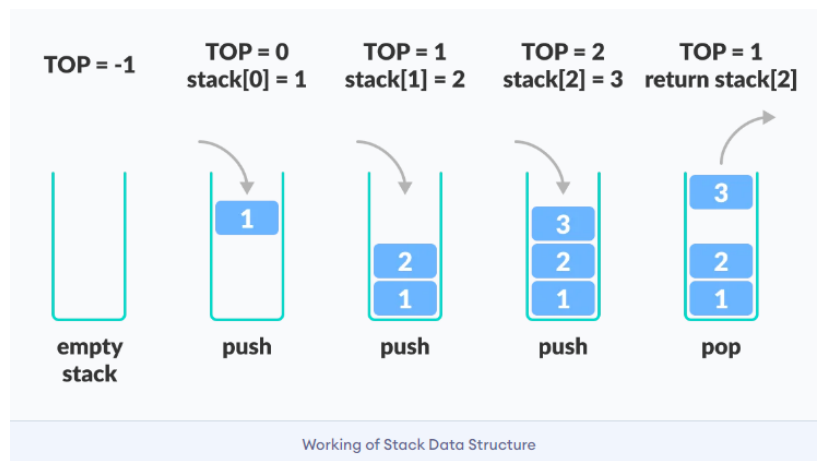
The types of linear data structures are Array, Queue, Stack, Linked List.

Let's discuss each linear data structure in detail.

Array: An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index.

For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size 10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.

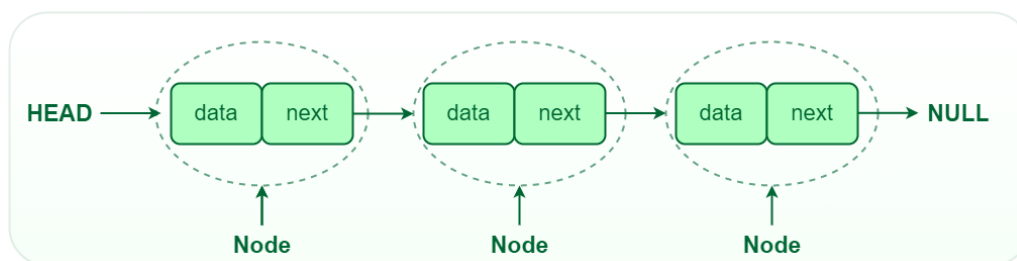
Stack: It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a **push operation**, and the deletion of data element from the list is known as **pop operation**.



Queue: It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue **front** end and **rear**. The insertion operation performed at the back end is known as **enqueue**, and the deletion operation performed at the front end is known as **dequeue**.



Linked list: It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.



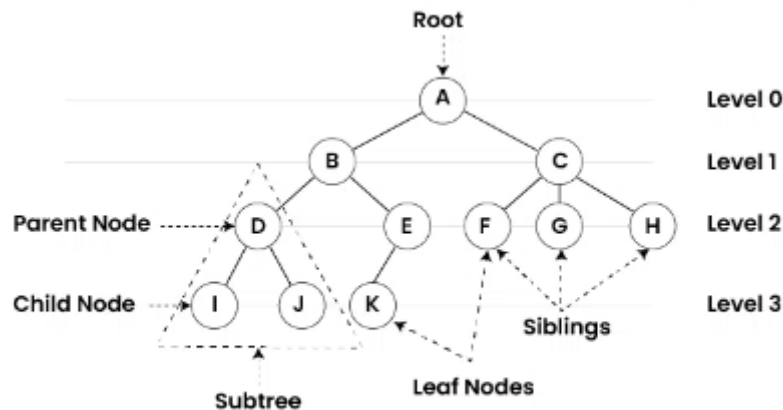
Non-linear data structure

A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and **Graphs** are the types of non-linear data structure.

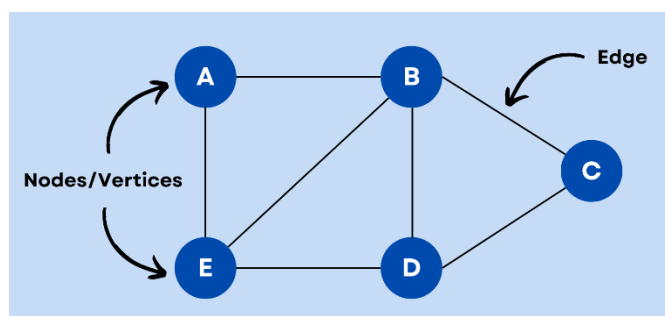
Trees:

A tree is a multilevel data structure defined as a set of nodes. The topmost node is named root node while the bottom most nodes are called leaf nodes. Each node has only one parent but can have multiple children.



Graph:

A graph is a nonlinear data structure and a graph is a set of vertexes and edges. Each vertex connected with path; each path is called edges. A graph is a pictorial representation of a set of objects connected by links known as edges. The interconnected nodes are represented by points named vertices, and the links that connect the vertices are called edges.



OPERATIONS ON VARIOUS DATA STRUCTURES

Data Structure is the way of storing data in computer's memory so that it can be used easily and efficiently. There are different data-structures used for the storage of data. The representation of particular data structure in the main memory of a computer is called as storage structure.

For Examples: Array, Stack, Queue, Tree, Graph

Operations on different Data Structure:

There are different types of operations that can be performed for the manipulation of data in every data structure. Some operations are explained and illustrated below:

- a) Traversing
- b) Searching
- c) Insertion
- d) Deletion

- a) **Traversing:** Traversing a Data Structure means to visit the element stored in it. It visits data in a systematic manner. This can be done with any type of DS.

Below is the program to illustrate traversal in an array

```
#include <stdio.h>

int main() {
    // Initialise array
    int arr[] = { 1, 2, 3, 4};

    // size of array
    int N = sizeof(arr) / sizeof(arr[0]);

    // Traverse the element of arr[]
    for (int i = 0; i < N; i++) {
        // Print the element
        printf("%d ", arr[i]);
    }
    return 0;
}
```

b) Searching: Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found. Searching is the operation which we can performed on data-structures like array, linked-list, tree, graph, etc.

Below is the program to illustrate searching an element in an array

```
#include <stdio.h>

int main() {
    // Initialise array
    int arr[] = { 1, 2, 3, 4};
```

```

// Element to be found
int K = 3;

// Size of array
int N = sizeof(arr) / sizeof(arr[0]);

// Traverse the element of arr[] to find element K
for (int i = 0; i < N; i++)
{
    // If Element is present then print the index and return
    if (arr[i] == K)
    {
        printf("Element found!\n");
        return 0;
    }
}

printf("Element Not found!\n");

return 0;
}

```

- c) **Insertion:** Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure. It is unsuccessful in some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element. The insertion has the same name as an insertion in the data-structure as an array, linked-list, graph, tree.

```

#include <stdio.h>

int main() {
    // Initialize array
    int arr[10] = { 1, 2, 3, 4}; // Initial elements
    int N = 4; // Current number of elements

    // Print original array
    printf("Original array: ");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Insert element at a specific position
    int pos = 2;

```

```

int element = 99;
if (pos >= 0 && pos <= N) {
    for (int i = N; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = element;
    N++;
} else {
    printf("Invalid position\n");
}
printf("After inserting 99 at position 2: ");
for (int i = 0; i < N; i++) {
    printf("%d ", arr[i]);
}
printf("\n");

// Insert element at the front
element = 77;
for (int i = N; i > 0; i--) {
    arr[i] = arr[i - 1];
}
arr[0] = element;
N++;
printf("After inserting 77 at the front: ");
for (int i = 0; i < N; i++) {
    printf("%d ", arr[i]);
}
printf("\n");

// Insert element at the end
element = 88;
arr[N] = element;
N++;
printf("After inserting 88 at the end: ");
for (int i = 0; i < N; i++) {
    printf("%d ", arr[i]);
}
printf("\n");

return 0;
}

```

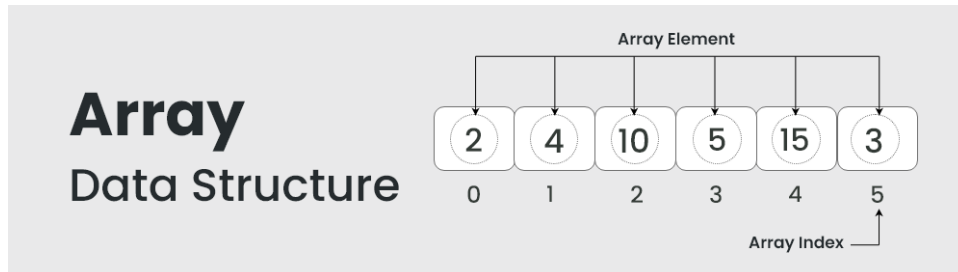
- d) Deletion: It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure. The deletion has the same name as a deletion in the data-structure as an array, linked-list, graph, tree, etc.

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

```
int main() {  
    // Initialize array with a large enough size  
    int arr[10] = { 1, 2, 3, 4}; // Initial elements  
    int N = 4; // Current number of elements  
  
    // Print original array  
    printf("Original array: ");  
    for (int i = 0; i < N; i++) {  
        printf("%d ", arr[i]);  
    }  
    printf("\n");  
  
    // Remove element at a specific position  
    pos = 3;  
    if (pos >= 0 && pos < N) {  
        for (int i = pos; i < N - 1; i++) {  
            arr[i] = arr[i + 1];  
        }  
        N--;  
    } else {  
        printf("Invalid position\n");  
    }  
    printf("After removing element at position 3: ");  
    for (int i = 0; i < N; i++) {  
        printf("%d ", arr[i]);  
    }  
    printf("\n");  
  
    return 0;  
}
```

Array Data Structure

- An array is a collection of items of the same variable type that are stored at contiguous memory locations.
- It's one of the most popular and simple data structures and is often used to implement other data structures.
- Each item in an array is indexed starting with 0 .
- Each element in an array is accessed through its index.



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

1.Declare an array of integers:

```
int arr[5]; // Array of 5 integers
```

This declares an array named **arr** that can hold 5 integers.

2.Declare an array of characters (string):

```
char str[10]; // Array of 10 characters (string)
```

This declares an array named **str** that can hold 10 characters, which is typically used for storing strings in C.

3.Initialize an array with values:

```
int numbers[3] = {1, 2, 3}; // Array of 3 integers initialized with values
```

This declares an array named **numbers** of size 3 and initializes it with the values {1, 2, 3}.

4.Declare an array without initializing it:

```
float data[100]; // Array of 100 floats, initially uninitialized
```

This declares an array named **data** that can hold 100 floating-point numbers. The initial values of the array elements are undefined.

Array Operations

Common operations performed on arrays include:

Traversal : Visiting each element of an array in a specific order (e.g., sequential, reverse).

Insertion : Adding a new element to an array at a specific index.

Deletion : Removing an element from an array at a specific index.

Searching : Finding the index of an element in an array.

Applications of Array

Arrays are used in a wide variety of applications, including:

Storing data for processing

Implementing data structures such as stacks and queues

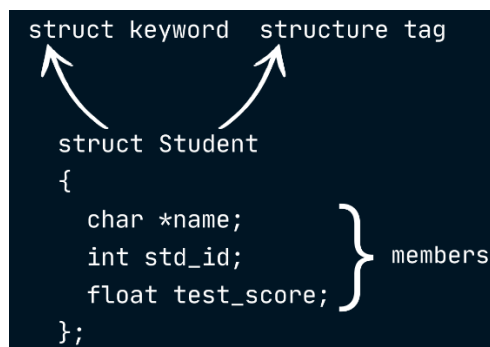
Representing data in tables and matrices

Creating dynamic data structures such as linked lists and trees

C Structures

The structure in C is a user-defined data type that can be used to **store several variable and data type into a single place**.

The **struct** keyword is used to define the structure in the C programming language. The items in the structure are called its **member**.



```
struct keyword  structure tag
  ^             ^
struct Student
{
    char *name;
    int std_id;
    float test_score;
};
```

The diagram shows the C structure syntax with annotations. Two arrows point from the text 'struct keyword' and 'structure tag' to the 'struct' and 'Student' parts of the code. A bracket on the right side of the member list is labeled 'members'.

C Structure Declaration

In structure declaration, we specify its member variables along with their datatype. We can use the **struct keyword** to declare the structure in C using the following syntax:

Syntax

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
};
```

Access Structure Members

We can access structure members by using the (.) dot operator and (->) pointer.

Syntax for (.) dot operator:

```
structure_name.member1;  
structure_name.member2;
```

Example program:

```
#include <stdio.h>  
// Define the Person structure  
struct Person {  
    char name[50];  
    int age;  
    float height;  
};  
  
int main() {  
    // Declare a Person instance  
    struct Person person1;  
  
    // Assign values to person1's members  
    printf("Enter name: ");  
    scanf("%s", person1.name);  
  
    printf("Enter age: ");  
    scanf("%d", &person1.age);  
  
    printf("Enter height (in meters): ");  
    scanf("%f", &person1.height);  
    // Print the details of person1  
    printf("\nPerson Details:\n");  
    printf("Name: %s\n", person1.name);  
    printf("Age: %d\n", person1.age);  
    printf("Height: %.2f meters\n", person1.height);  
    return 0;  
}
```

```
Enter name: JohnDoe  
Enter age: 25  
Enter height (in meters): 1.75  
  
Person Details:  
Name: JohnDoe  
Age: 25  
Height: 1.75 meters
```

Example program for -> pointer operator:

```
#include <stdio.h>

// Define the Person structure
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Declare a Person instance
    struct Person person1;
    // Declare a pointer to the Person instance
    struct Person *ptr = &person1;

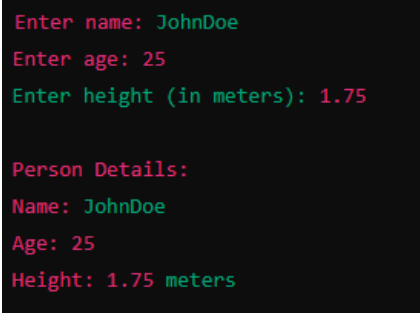
    // Assign values to person1's members using the pointer
    printf("Enter name: ");
    scanf("%s", ptr->name);

    printf("Enter age: ");
    scanf("%d", &ptr->age);

    printf("Enter height (in meters): ");
    scanf("%f", &ptr->height);

    // Print the details of person1 using the pointer
    printf("\nPerson Details:\n");
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    printf("Height: %.2f meters\n", ptr->height);

    return 0;
}
```

A screenshot of a terminal window with a black background and green text. It shows the output of the C program, including prompts for name, age, and height, followed by a section titled 'Person Details:' that displays the entered values: Name: JohnDoe, Age: 25, and Height: 1.75 meters.

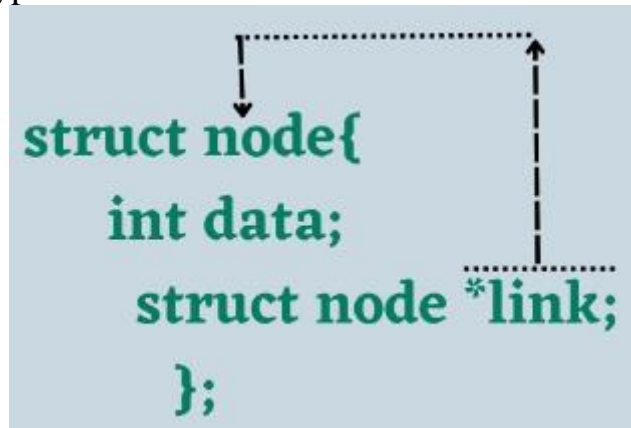
```
Enter name: JohnDoe
Enter age: 25
Enter height (in meters): 1.75

Person Details:
Name: JohnDoe
Age: 25
Height: 1.75 meters
```

Self-Referential Structures:

Self-referential structures are those structures that contain a reference to the data of its same type.

That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure.



Here, the structure node will contain two types of data: an integer value and a pointer next. You must be wondering why we need such a structure. Actually, self-referential structure is the foundation of other data structures.

This is commonly used in data structures such as linked lists, trees, and other recursive data structures.

Example program:

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

```
// Define the Node structure
struct Node {
    int data;
    struct Node* next; // Pointer to the next node
};
```

```
int main() {
    // Create three nodes
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // Allocate memory for nodes in the heap
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    // Assign data and link the nodes
    head->data = 1; // Assign data to head node
    head->next = second; // Link head node to second node
```

```
second->data = 2; // Assign data to second node
second->next = third; // Link second node to third node
```

```
third->data = 3; // Assign data to third node
third->next = NULL; // End of the list
```

```
// Print the linked list
struct Node* current = head;
while (current != NULL) {
    printf("Data: %d\n", current->data);
    current = current->next;
}
```

```
// Free allocated memory
free(head);
free(second);
free(third);
```

```
return 0;
```

```
}
```

Output:

```
Data: 1
Data: 2
Data: 3
```

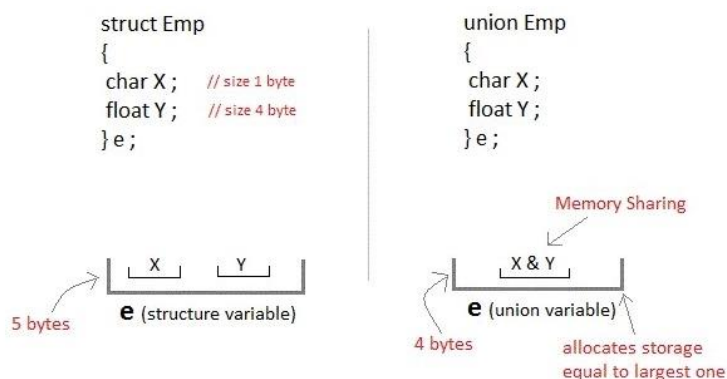
Unions:

Union is a collection of variables of different data types, that you can only store information in one field at any one time. unions are used to save memory.

They are useful for applications that involve multiple members, where values need not be assigned to all the members at any one time.

Size of Union

The size of the union will always be equal to the size of the largest member of the array. All the less-sized elements can store the data in the same space without any overflow.



Program:

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

```
// Define a union
```

```
union Info {
    int number;
    float decimal;
    char text[20];
};
```

```
int main() {
```

```
    // Declare a union variable
    union Info info;
```

```
    // Assign and print integer value
    info.number = 42;
```

```
    // Assign and print float value
    info.decimal = 3.14;
```

```
    // Assign and print string value
    strcpy(info.text, "Hello, World!");
    printf("info.number : %d\n", info.number);
    printf("info.decimal : %f\n", info.decimal);
    printf("info.text : %s\n", info.text);
```

```
    return 0;
```

```
}
```

Output:

```
info.number : 1111638594
info.decimal : 0.000000
info.text : Hello, World!
```

Difference between unions and structures

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

Pointers:

- A **pointer** is a variable that **stores** the **memory address** of another variable as its value.
- A **pointer variable points** to a **data type**(like int) of the same type, and is created with the*operator.
- The address of the variable you are working with is assigned to the pointer:

Syntax for pointer :

```
datatype * ptr;
```

- ptr is the name of the pointer.
- datatype is the type of data it is pointing to.

Program:

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

```
int main() {  
    int num = 10;  
    int *ptr;  
  
    // Assign the address of num to ptr  
    ptr = &num;  
  
    printf("Address of num variable: %p\n", &num);  
    printf("Value of ptr variable: %p\n", ptr);  
    printf("Value of num variable: %d\n", num);  
    printf("Value of *ptr variable: %d\n", *ptr);  
  
    // Modify the value using pointer  
    *ptr = 20;  
  
    printf("\nAfter modification:\n");  
    printf("Value of num variable: %d\n", num);  
    printf("Value of *ptr variable: %d\n", *ptr);  
  
    return 0;  
}
```

```
Address of num variable: 0x7ffee0b2b9ac  
Value of ptr variable: 0x7ffee0b2b9ac  
Value of num variable: 10  
Value of *ptr variable: 10  
  
After modification:  
Value of num variable: 20  
Value of *ptr variable: 20
```


Types of Pointers in C

Pointers in C can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

- Integer Pointers
- Array Pointer
- Structure Pointer
- Function Pointers
- Double Pointers
- NULL Pointer
- Wild Pointers
- Constant Pointers

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

Syntax : `int *ptr;`

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays. We can create a pointer to an array using the given syntax.

Syntax : `char *ptr = &array_name;`

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

```
int main() {
    int arr[] = { 10, 20, 30, 40, 50};
    int *ptr;

    // Assign the base address of arr to ptr
    ptr = arr;

    printf("Elements of the array using array indices:\n");
    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    printf("\nElements of the array using pointer arithmetic:\n");
    for (int i = 0; i < 5; i++) {
        printf("*(ptr + %d) = %d\n", i, *(ptr + i));
    }

    return 0;
}
```

```
Elements of the array using array indices:
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50

Elements of the array using pointer arithmetic:
*(ptr + 0) = 10
*(ptr + 1) = 20
*(ptr + 2) = 30
*(ptr + 3) = 40
*(ptr + 4) = 50
```

3. Structure Pointer

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

Syntax : struct struct_name *ptr;

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – int func (int, char), the function pointer for this function will be

Syntax: int (*ptr)(int, char);

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.

Syntax: datatype ** pointer_name;

6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax: data_type *pointer_name = NULL;
or

```
pointer_name = NULL;
```

7. Wild Pointers

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values is updated using wild pointers, they could cause data abort or data corruption.

Example

```
int *ptr;  
char *str;
```

8. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Syntax: `data_type * const pointer_name;`

Dynamic memory allocation in C:

C 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. They are:

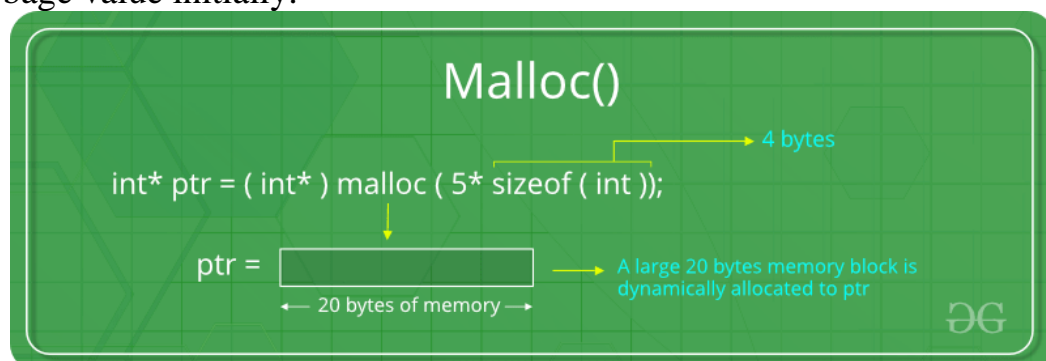
1. `malloc()`
2. `calloc()`
3. `free()`
4. `realloc()`

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:

```
graph TD
    A[cast-type] --- B["(int*)"]
    C[pointer_name] --- D["ptr"]
    E[size] --- F["sizeof(int)"]
```

`int* ptr = (int*) malloc(sizeof(int));`

Program:

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

```
int main() {
    int* array_ptr; // Pointer to hold the base address of the dynamically allocated array
    int num_elements, i;

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

    // Dynamically allocate memory using malloc()
    array_ptr = (int*)malloc(num_elements * sizeof(int));

    // Check if the memory allocation was successful
    if (array_ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit the program if allocation fails
    } else {
        // Memory allocation was successful
        printf("Memory successfully allocated using malloc.\n");

        // Initialize the elements of the array
        for (i = 0; i < num_elements; ++i) {
            array_ptr[i] = i + 1; // Store values in the array (e.g., 1, 2, 3, ...)
        }

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

    // Free dynamically allocated memory
    free(array_ptr);

    return 0;
}
```

```
Enter number of elements: 5
Entered number of elements: 5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

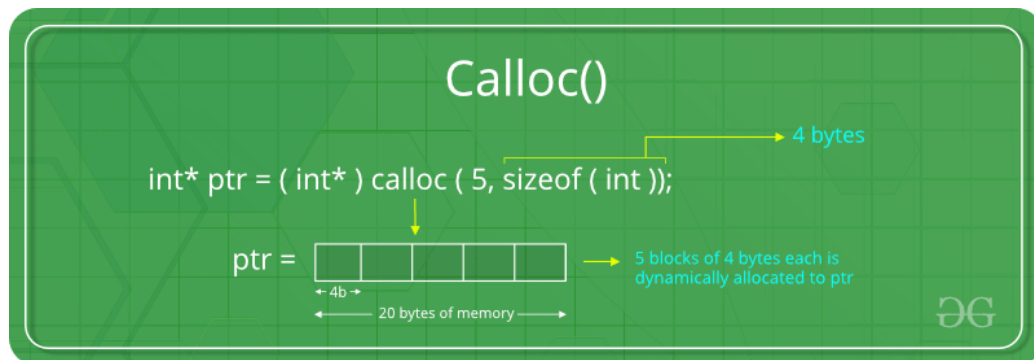
C calloc() method

- “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:
- It initializes each block with a default value ‘0’.
- It has two parameters or arguments as compare to malloc().

Syntax of calloc() in C

ptr = (cast-type*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.



Program:

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

```
int main() {
    int* array_ptr; // Pointer to hold the base address of the dynamically allocated array
    int num_elements = 5; // Number of elements for the array
    int i;

    printf("Enter number of elements: %d\n", num_elements);

    // Dynamically allocate memory using calloc()
    array_ptr = (int*)calloc(num_elements, sizeof(int));

    // Check if the memory allocation was successful
    if (array_ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit the program if allocation fails
    } else {
        // Memory allocation was successful
    }
}
```

```

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

// Initialize the elements of the array
for (i = 0; i < num_elements; ++i) {
    array_ptr[i] = i + 1; // Store values in the array (e.g., 1, 2, 3, ...)
}

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

// Free dynamically allocated memory
free(array_ptr);

return 0;
}

```

```

Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

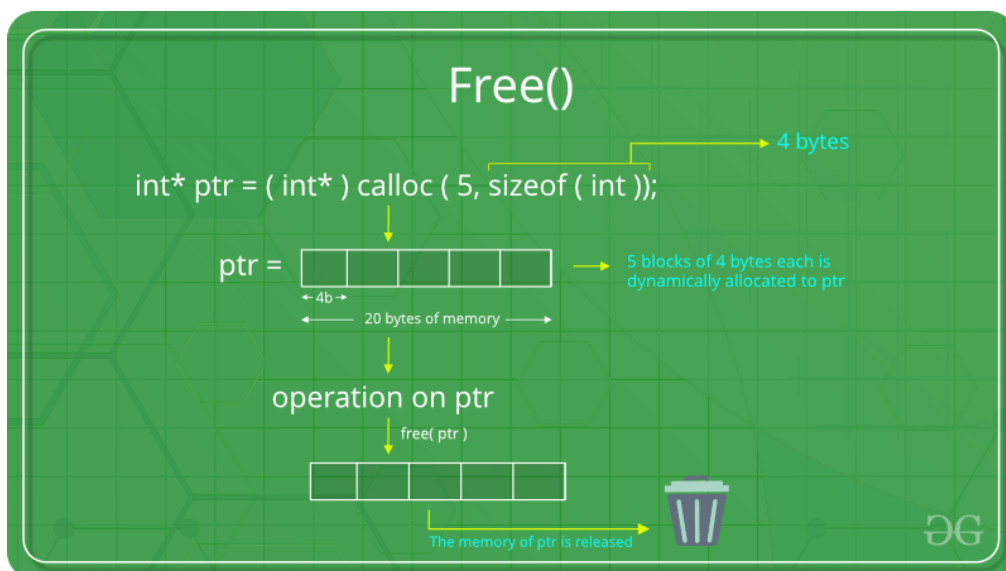
```

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 of free() in C : **free(ptr);**



Program:

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

```
int main() {
    int *ptr;

    // Allocate memory dynamically
    ptr = (int*)malloc(5 * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Use allocated memory
    for (int i = 0; i < 5; ++i) {
        ptr[i] = i + 1;
    }

    // Free allocated memory
    free(ptr);
    ptr = NULL; // Optional: Set pointer to NULL after freeing

    return 0;
}
```

Expected Output:

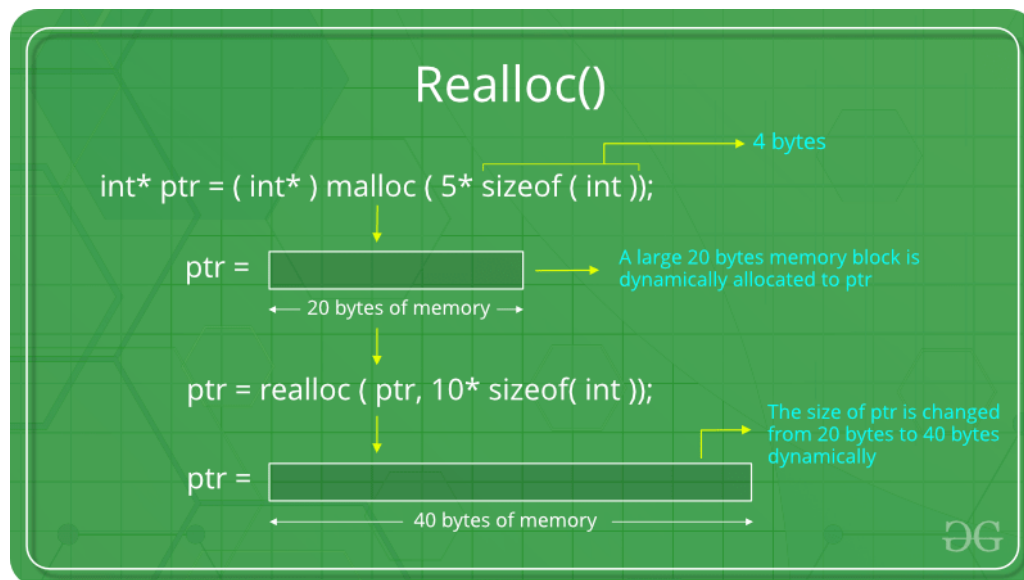
This program does not produce any visible output to the console because `free()` itself does not print anything.

The purpose of `free()` is to manage memory behind the scenes. It ensures that memory allocated dynamically is released properly when no longer needed, preventing memory leaks and improving overall program performance.

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 of `realloc()` in C : **ptr = realloc(ptr, newSize);**



Program:

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

```
int main() {
    int *ptr; // Declare a pointer to int
    int x = 20;

    // Allocate memory using malloc
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Assign value to the allocated memory
    ptr[0] = x;
    printf("Value of ptr before memory reallocation: %d\n", ptr[0]);

    // Reallocate memory using realloc
    ptr = (int *)realloc(ptr, 5 * sizeof(int));
    if (ptr == NULL) {
        printf("Memory reallocation failed.\n");
        return 1;
    }

    // Print the value after reallocation
    printf("Value of ptr after memory reallocation: %d\n", ptr[0]);

    // Free allocated memory
    free(ptr);
    return 0;
}
```



```
Value of ptr before memory reallocation: 20
Value of ptr after memory reallocation: 20
```

Memory Representation of Arrays

- In memory, arrays are stored in contiguous locations.
- Each element is stored in adjacent memory locations.
- The memory representation of an array is like a long tape of bytes, with each element taking up a certain number of bytes.

Variable Declaration and Memory Storage

- When a variable is declared, memory is allocated to store its value.
- For example, `int a = 5;` means a variable `a` of type `int` is declared and assigned the value 5.
- In memory, the value 5 will be stored in the allocated memory space for variable `a`. #

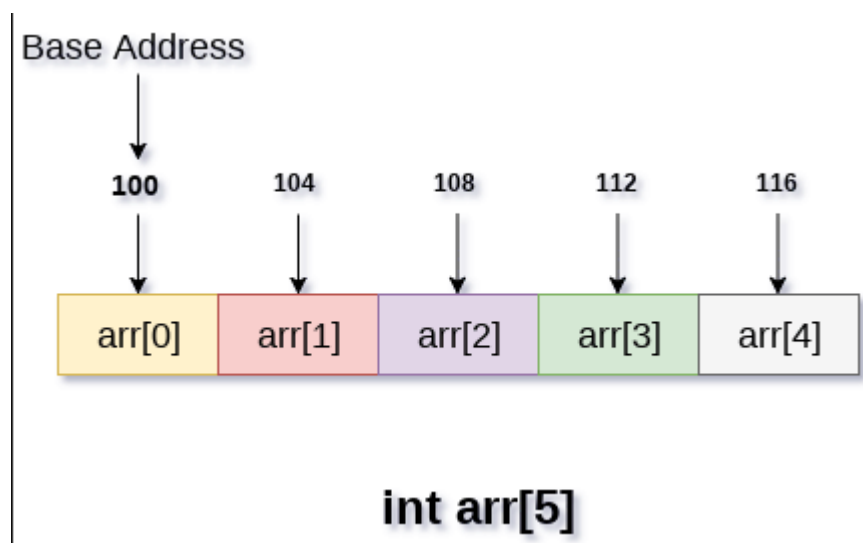
Memory Allocation for Integer Variable

Memory Allocation

The memory manager allocates 4 bytes of memory for storing the integer value.

Allocation of Bytes

To store the binary number “101”, we allocate 4 bytes in memory. Each byte consists of 8 bits, and the total number of bits required to store the number is 32 bits.



We can define the indexing of an array in the below ways:

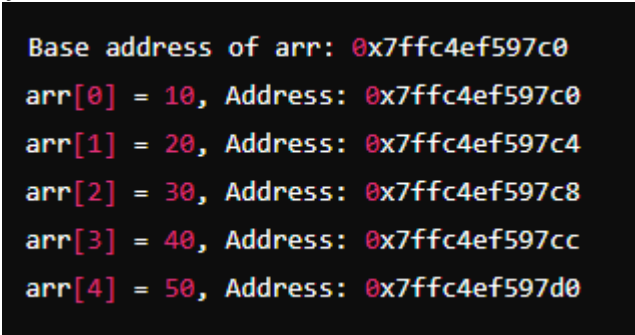
1.0 (zero-based indexing): The first element of the array will be arr[0].

2.1 (one-based indexing): The first element of the array will be arr[1].

3.n (n -based indexing): The first element of the array can reside at any random index number.

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

```
int main() {  
    // Define and initialize an array of integers  
    int arr[5] = {10, 20, 30, 40, 50};  
  
    // Print the base address of the array  
    printf("Base address of arr: %p\n", (void *)arr);  
  
    // Accessing and printing elements along with their addresses  
    for (int i = 0; i < 5; ++i) {  
        printf("arr[%d] = %d, Address: %p\n", i, arr[i], (void *)&arr[i]);  
    }  
  
    return 0;  
}
```



```
Base address of arr: 0x7ffc4ef597c0  
arr[0] = 10, Address: 0x7ffc4ef597c0  
arr[1] = 20, Address: 0x7ffc4ef597c4  
arr[2] = 30, Address: 0x7ffc4ef597c8  
arr[3] = 40, Address: 0x7ffc4ef597cc  
arr[4] = 50, Address: 0x7ffc4ef597d0
```

Dynamic allocate array:

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

```
#include <stdlib.h> // Required for malloc, free, and NULL
```

```
int main() {  
    int *arr; // Declare a pointer to int  
  
    int n; // Variable to store the size of the array  
    printf("Enter the size of the array: ");  
    scanf("%d", &n); // Input the size of the array  
  
    // Dynamically allocate memory for the array  
    arr = (int *)malloc(n * sizeof(int));
```

```

// Check if memory allocation is successful
if (arr == NULL) {
    printf("Memory allocation failed.\n");
    return 1; // Exit the program if allocation fails
}

// Input elements into the dynamically allocated array
printf("Enter %d elements:\n", n);
for (int i = 0; i < n; ++i) {
    scanf("%d", &arr[i]);
}

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

return 0;
}

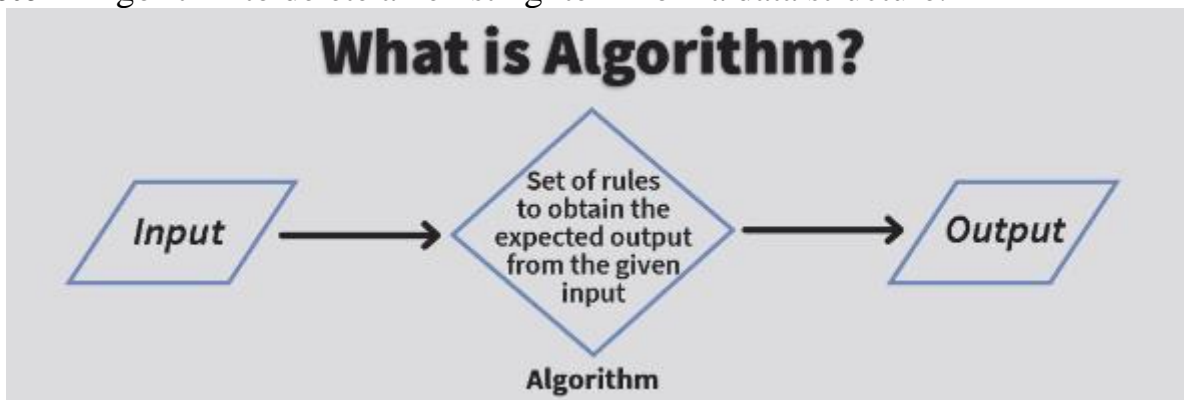
```

Analysis of an algorithm

Algorithm

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

- ☐ **Search** – Algorithm to search an item in a data structure.
- ☐ **Sort** – Algorithm to sort items in a certain order.
- ☐ **Insert** – Algorithm to insert item in a data structure.
- ☐ **Update** – Algorithm to update an existing item in a data structure.
- ☐ **Delete** – Algorithm to delete an existing item from a data structure.



Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem – Design an algorithm to add two numbers and display the result. **public class** SumOfNumbers1

```
{  
public static void main(String args[])  
{  
int a = 225, n = 115, c;  
c = a + b;  
System.out.println("The sum of numbers is: "+sum);  
}  
}
```

Step 1 – START

Step 2 – declare three integers **a**, **b** & **c**

Step 3 – define values of **a** & **b**

Step 4 – add values of **a** & **b**

Step 5 – store output of step 4 to **c**

Step 6 – print **c**

Step 7 – STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

Step 1 – START ADD

Step 2 – get values of **a** & **b**

Step 3 – $c \leftarrow a + b$

Step 4 – display **c**

Step 5 – STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.

EFFICIENCY OF AN ALGORITHM:

The efficiency of an algorithm in data structures refers to how well the algorithm utilizes computational resources, such as time and memory, to solve a problem.

There are two main measures for the efficiency of an algorithm. The complexity of an algorithm is divided into two types:

1. Time complexity
2. Space complexity

Time complexity

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.