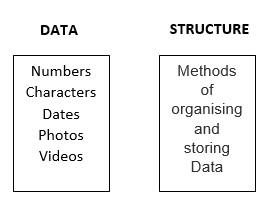
**What is a Data Structure ?**

**Data Structure = Data + Structure**

****

**Data : It is information that must be processed and stored in a computer. Eg: 23, 23.00, KMIT, Large Texts , Photos , Music Files , etc.**

**Structure : It specifies the logical relationship between various data elements and how they are kept in memory.**

**The structure has an impact on the efficiency of data operations.**

***Searching ,Sorting ,Min Max Avg Mean ,Neighbors , Inserting , Deleting , etc***

**It is a framework organizing and manipulating data, making information management and analysis easier.**

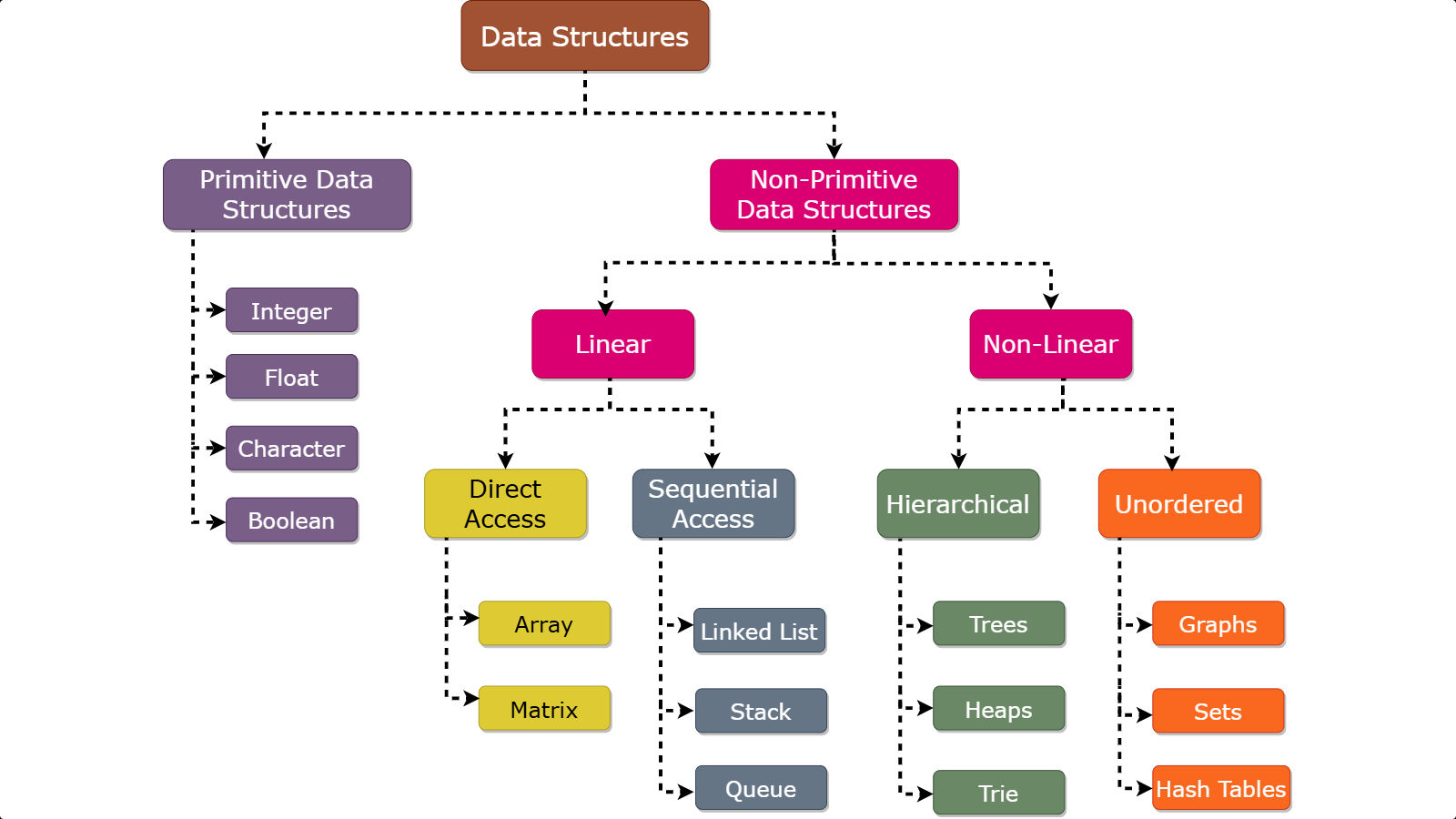
**They also enable the development of efficient data-processing algorithms, allowing difficult issues to be solved in a reasonable amount of time.**

* **What is the need of Data Structure? Improved Time Complexity: Using appropriate data structures can lead to better time complexity, making it possible to solve problems more quickly. For example, searching for an element in a sorted array is faster than searching for it in an unsorted array.**
* **Better Space Complexity: Data structures can help to reduce the amount of memory needed to store data. For example, using a linked list instead of an array can reduce the amount of memory needed to store the same data.**
* **Efficient Data Retrieval: Data structures make it easier to retrieve specific data efficiently. For example, a hash table can retrieve data in constant time, while searching through an unsorted array takes linear time.**
* **Better Data Management: Data structures make it easier to manage and manipulate data. For example, a stack can be used to implement an undo functionality in an application.**
* **Solving Complex Problems: Data structures can provide the foundation for efficient algorithms, making it possible to solve complex problems in a reasonable amount of time. For example, graph algorithms can be used to find the shortest path between two points or to find the minimum spanning tree of a graph.**

**Increased Data Security: Data structures can be designed to provide additional security features, such as data encryption and protection against unauthorized access**

**Characteristics of Data Structure**

* **Representation of Data: a way of representing data in a [computer](https://techskillguru.com/ds/introduction-to-ds)'s memory, making it possible to store, manipulate, and access data efficiently.**
* **Access Techniques: Different data structures provide different techniques for accessing data stored within them, such as random access or sequential access.**
* **Storage Organization: Data structures define the organization of data in memory, such as linear or hierarchical organization.**
* **Insertion and Deletion Operations: Different data structures support different methods for adding and removing elements, such as insertion at the end or deletion from the front.**
* **Time and Space Complexity: Data structures can have different time and space complexities, depending on the operations they support and the way they organize data.**
* **Adaptability: Some data structures are more adaptable to certain types of data and operations than others. For example, a stack is more suitable for problems that require Last-In-First-Out (LIFO) behavior, while a queue is better suited for problems that require First-In-First-Out (FIFO) behavior.**
* **Flexibility: Different data structures have different degrees of flexibility, such as the ability to dynamically grow or shrink in size, or the ability to efficiently insert or delete elements in the middle.**



**Primitive**

* **Primitive Data Structures:These are the basic data types provided by a programming language1**

**Examples include integers, floats, characters, and booleans**

* **They store single values directly in memory**
* **They have a fixed size, which is predefined by the language, making them memory-efficient**
* **Operations are basic, such as arithmetic and logical operations.**
* **They are allocated on the stack, allowing for fast and straightforward access**
* **Non- Primitives**

**Non-primitive data structures are complex and built using primitive data types and/or other non-primitive data types. Examples include arrays, lists, stacks, queues, trees, and graphs**

* **They can store collections of data elements.**
* **Their size can be dynamic and can vary during runtime. However, static arrays are an exception.**
* **Support a wide range of complex operations specific to the data structure, such as insertion, searching, and sorting**
* **Typically allocated on the heap, with a reference to the memory stored in the stack. This allows them to handle dynamic data sizes and more complex data organization**

**RECURSION**

**recursion**is a technique in which a function calls itself repeatedly until a given condition is satisfied. It is used for solving a problem by breaking it down into smaller, simpler sub-problems. Then finding the solution of it and combining this solution to find the global solution.

**Basic Example**:

#include *<iostream>*

**using** **namespace** **std**;

void printHello(int n) {

**if** (n == 0) **return**;

cout << "Hello" << endl;

printHello(n - 1);

}

int main() {

printHello(5);

**return** 0;

}

**Output**

Hello

Hello

Hello

Hello

Hello

## ****Recursive Function****

A function that calls itself is called a **recursive function**. When a recursive function is called, it executes a set of instructions and then calls itself to execute the same set of instructions with a smaller input. A recursive function should contain,

* **Recursive Case**: Recursive case is the way in which the recursive call is present in the function.
* **Base Condition:** The base condition is the condition that is used to terminate the recursion.
* **Base Condition**
* *if (n == 0) return;*
* **Recursive Case**
* *printHello(n – 1*

**Types of [Recursions](https://www.geeksforgeeks.org/recursion/):**   
Recursion are mainly of**two types** depending on whether **a function calls itself from within itself** or **more than one function call one another mutually.** The first one is called **direct recursion** and another one is called **indirect recursion**. Thus, the two types of recursion are:

**1. Direct Recursion**: These can be further categorized into **four types**:

* **Tail Recursion**: If a recursive function calling itself and that recursive call is the last statement in the function then it’s known as **Tail Recursion.** After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.
* **Example:**
* C++
* C
* Java
* Python3
* C#
* Javascript

|  |
| --- |
| // Code Showing Tail Recursion  #include <iostream>  using namespace std;    // Recursion function  void fun(int n)  {      if (n > 0) {          cout << n << " ";            // Last statement in the function          fun(n - 1);      }  } |

OUTPUTë

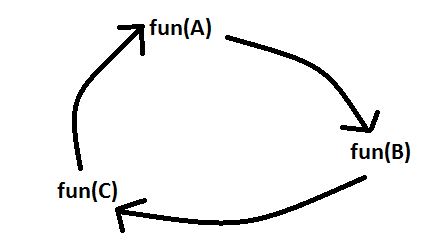
3

2

1

**Time Complexity For Tail Recursion : O(n)**   
**Space Complexity For Tail Recursion : O(n)**

**Indirect Recursion**: In this recursion, there may be more than one functions and they are calling one another in a circular manner.



#include <iostream>

using namespace std;

void funB(int n);

void funA(int n)

{

    if (n > 0) {

        cout <<" "<< n;

        // fun(A) is calling fun(B)

        funB(n - 1);

    }

}

void funB(int n)

{

    if (n > 1) {

        cout <<" "<< n;

 funA(n / 2);

    }

}

// Driver code

int main()

{

    funA(20);

    return 0;

}

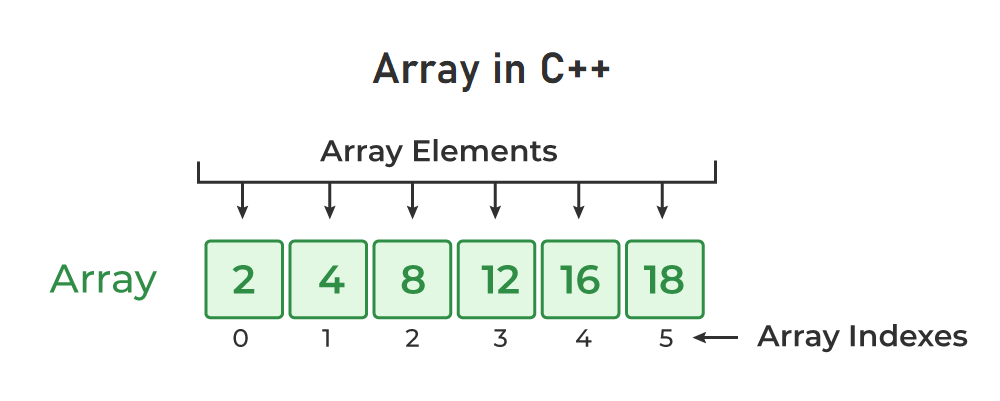
OUTPUT

**20 19 9 8 4 3 1**

**Arrays**

In C++, an array is a data structure that is used to store multiple values of similar data types in a contiguous memory location.

**For example**, if we have to store the marks of 4 or 5 students then we can easily store them by creating 5 different variables but what if we want to store marks of 100 students or say 500 students then it becomes very challenging to create that numbers of variable and manage them. Now, arrays come into the picture that can do it easily by just creating an array of the required size.



## Properties of Arrays in C++

* Array Declaration in C++An Array is a collection of data of the same data type, stored at a contiguous memory location.
* Indexing of an array starts from **0.**It means the first element is stored at the 0th index, the second at 1st, and so on.
* Elements of an array can be accessed using their indices.
* Once an array is declared its size remains constant throughout the program.
* An array can have multiple dimensions.
* The size of the array in bytes can be determined by the sizeof operator using which we can also find the number of elements in the array.
* We can find the size of the type of elements stored in an array by subtracting adjacent addresses.

## Array Declaration in C++

In C++, we can declare an array by simply specifying the data type first and then the name of an array with its size.

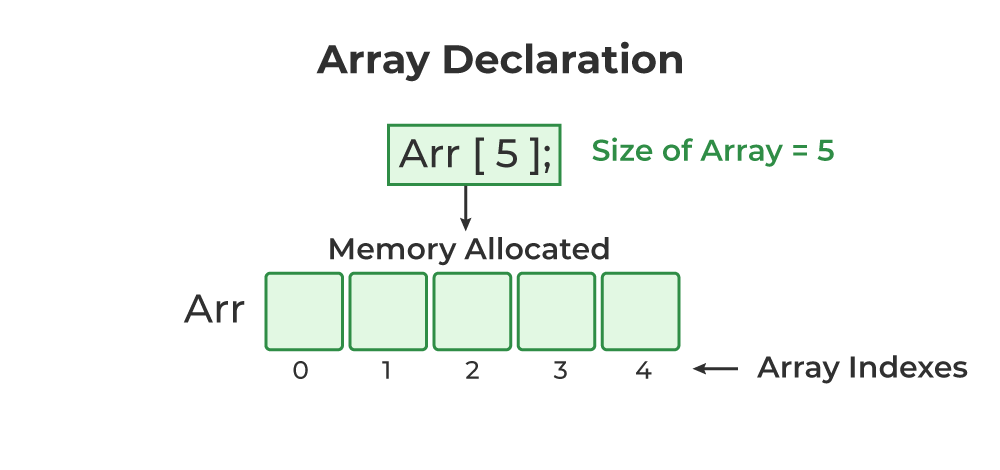
data\_type array\_name[Size\_of\_array];

**Example**

int arr[5];

Here,

* **int:**It is the type of data to be stored in the array. We can also use other data types such as char, float, and double.
* **arr:**It is the name of the array.
* **5:** It is the size of the array which means only 5 elements can be stored in the array.



## Initialization of Array in C++

In C++, we can initialize an array in many ways but we will discuss some most common ways to initialize an array. We can initialize an array at the time of declaration or after declaration.

### 1. Initialize Array with Values in C++

We have initialized the array with values. The values enclosed in curly braces ‘{}’ are assigned to the array. Here, 1 is stored in arr[0], 2 in arr[1], and so on. Here the size of the array is 5.

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

### 2. Initialize Array with Values and w

We have initialized the array with values but we have not declared the length of the array, therefore, the length of an array is equal to the number of elements inside curly braces.

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

### 3. Initialize Array after Declaration (Using Loops)

We have initialized the array using a loop after declaring the array. This method is generally used when we want to take input from the user or we cant to assign elements one by one to each index of the array. We can modify the loop conditions or change the initialization values according to requirements.

for (int i = 0; i < N; i++) {  
 arr[i] = value;  
}

### 4. Initialize an array partially in C++

Here, we have declared an array ‘partialArray’ with size ‘5’ and with values ‘1’ and ‘2’ only. So, these values are stored at the first two indices, and at the rest of the indices ‘0’ is storeint partialArray[5] = {1, 2};

### 5. Initialize the array with zero in C++

We can initialize the array with all elements as ‘0’ by specifying ‘0’ inside the curly braces. This will happen in case of zero only if we try to initialize the array with a different value say ‘2’ using this method then ‘2’ is stored at the 0th index only.

int zero\_array[5] = {0};

## Accessing an Element of an Array in C++

Elements of an array can be accessed by specifying the name of the array, then the index of the element enclosed in the array subscript operator []. For example, arr[i].

### Example 1: The C++ Program to Illustrate How to Access Array Elements

*// C++ Program to Illustrate How to Access Array Elements*

#include *<iostream>*

**using** **namespace** **std**;

int main()

{

int arr[3];

*// Inserting elements in an array*

arr[0] = 10;

arr[1] = 20;

arr[2] = 30;

*// Accessing and printing elements of the array*

cout << "arr[0]: " << arr[0] << endl;

cout << "arr[1]: " << arr[1] << endl;

cout << "arr[2]: " << arr[2] << endl;

**return** 0;

}

**Output**

arr[0]: 10

arr[1]: 20

arr[2]: 30

## Update Array Element

To update an element in an array, we can use the index which we want to update enclosed within the array subscript operator and assign the new value.

arr[i] = new\_value;

## Recursion

Recursion in C++ is a technique in which a function calls itself repeatedly until a given condition is satisfied. In other words, recursion is the process of solving a problem by breaking it down into smaller, simpler sub-problems.

### Syntax Structure of Recursion

return\_type ***recursive\_func*** {  
 ....  
 // Base Condition  
 // Recursive Case  
 ....  
}

### Recursive Function

A function that calls itself is called a **recursive function**. When a recursive function is called, it executes a set of instructions and then calls itself to execute the same set of instructions with a smaller input. This process continues until a base case is reached, which is a condition that stops the recursion and returns a value.

### Base Condition

The base condition is the condition that is used to terminate the recursion. The recursive function will keep calling itself till the base condition is satisfied.

### Recursive Case

Recursive case is the way in which the recursive call is present in the function. Recursive case can contain multiple recursive calls, or different parameters such that at the end, the base condition is satisfied and the recursion is terminated.

## Example of C++ Recursion

The following C++ program illustrates how to perform recursion.

*/ C++ Program to calculate the sum of first N natural*

*// numbers using recursion*

#include *<iostream>*

**using** **namespace** **std**;

int nSum(int n)

{

*// base condition to terminate the recursion when N = 0*

**if** (n == 0) {

**return** 0;

}

*// recursive case / recursive call*

int res = n + nSum(n - 1);

**return** res;

}

int main()

int n = 5;

*// calling the function*

int sum = nSum(n);

cout << "Sum = " << sum;

**return** 0;

}

**Output**

Sum = 15

In the above example,

* **Recursive Function:**nSum() is the Recursive Function
* **Recursive Case:** The expression, **int res = n + nSum(n – 1)**is the Recursive Case.
* **Base Condition:** The base condition is **if (n == 0) { return 0;}**

## Working of Recursion in C++

To understand how C recursion works, we will again refer to the example above and trace the flow of the program.

**1.**In the nSum() function, **Recursive Case**is

int res = n + nSum(n - 1);

**2.** In the example, n = 5, so as **nSum(5)’s** recursive case, we get

int res = 5 + nSum(4);

**3.**In **nSum(4)**, the recursion case and everything else will be the same, but n = 4. Let’s evaluate the recursive case for n = 4,

int res = 4 + nSum(3);

**4.**Similarly, for **nSum(3), nSum(2) and nSum(1)**

* int res = 3 + nSum(2); // nSum(3)  
  int res = 2 + nSum(1); // nSum(2)  
  int res = 1 + nSum(0); // nSum(1)
* Let’s not evaluate nSum(0) and further for now.

**5.** Now recall that the**return value**of the nSum() function in this same integer named **res**. So, instead of the function, we can put the value returned by these functions. As such, for nSum(5), we get

* int res = 5 + 4 + nSum(3);

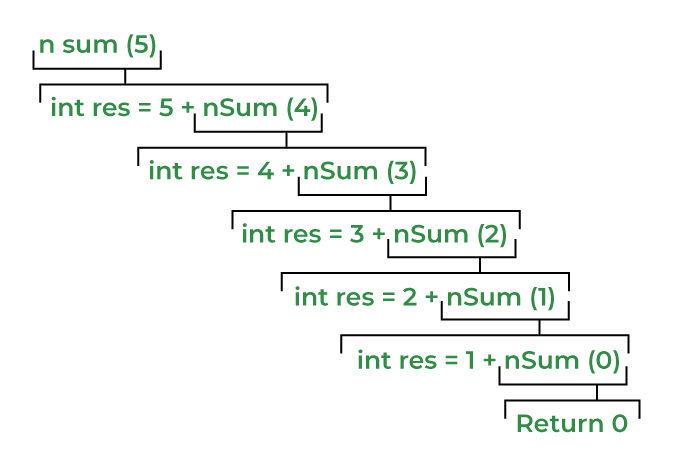
**6.**Similarly, putting return values of nSum() for every n, we get

* int res = 5 + 4 + 3 + 2 + 1 + nSum(0);

**7.**In nSum() function, the**base condition** is

* if (n == 0) {  
   return 0;  
  }
* which means that when nSum(0) will return 0. Putting this value in nSum(5)’s recursive case, we get
* int res = 5 + 4 + 3 + 2 + 1 + 0;  
   = 15

**8.**At this point, we can see that there are no function calls left in the recursive case. So the recursion will stop here and the final value returned by the function will be **15**which is the sum of the first 5 natural numbers.



**Direct Recursion**: These can be further categorized into **four types**:

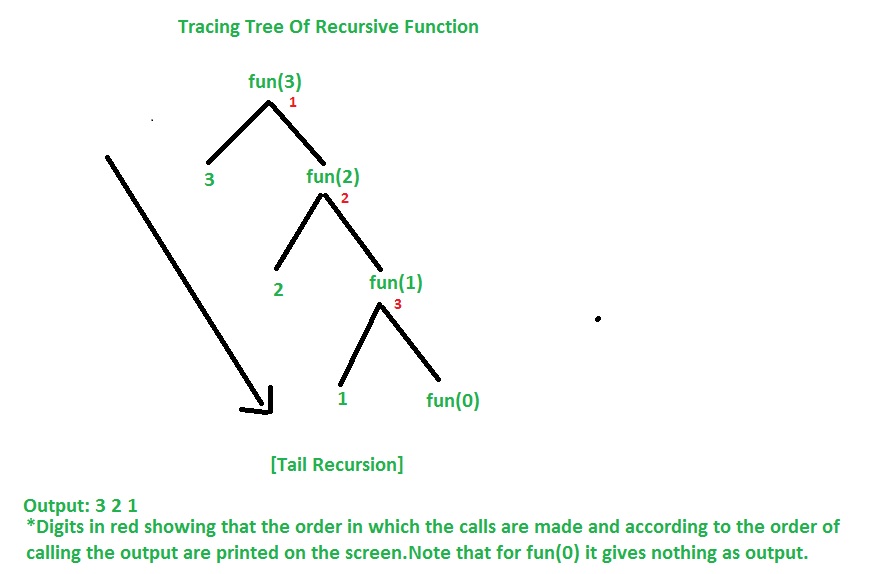
* **Tail Recursion**: If a recursive function calling itself and that recursive call is the last statement in the function then it’s known as **Tail Recursion.** After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

|  |
| --- |
| // Code Showing Tail Recursion  #include <iostream>  using namespace std;    // Recursion function  void fun(int n)  {      if (n > 0) {          cout << n << " ";            // Last statement in the function          fun(n - 1); |
| }  }    // Driver Code  int main()  {      int x = 3;      fun(x);      return 0;  }    // This code is contributed by shubhamsingh10 |

**Output**

3 2 1

Let’s understand the example by **tracing tree of recursive function.** That is how the calls are made and how the outputs are produced.



**Time Complexity For Tail Recursion : O(n)**   
**Space Complexity For Tail Recursion : O(n)**  
**Note:** Time & Space Complexity is given for this specific example. It may vary for another example.

Let’s now converting Tail Recursion into Loop and compare each other in terms of Time & Space Complexity and decide which is more efficient.

* C++
* C
* Java
* Python3
* C#
* Javascript

|  |
| --- |
| // Converting Tail Recursion into Loop  #include <iostream>  using namespace std;    void fun(int y)  {      while (y > 0) {          cout << y << " ";          y--;      }  } |

Driver code

int main()

{

    int x = 3;

    fun(x);

    return 0;

}

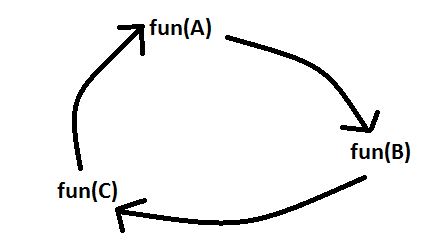
**Output**

3 2 1

**Time Complexity: O(n)**   
**Space Complexity: O(1)**

**Note:** Time & Space Complexity is given for this specific example. It may vary for another example.  
So it was seen that in case of loop the Space Complexity is O(1) so it was better to write code in loop instead of tail recursion in terms of Space Complexity which is more efficient than tail recursion.

**2. Indirect Recursion**: In this recursion, there may be more than one functions and they are calling one another in a circular manner

. 

From the above diagram fun(A) is calling for fun(B), fun(B) is calling for fun(C) and fun(C) is calling for fun(A) and thus it makes a cycle.

**Example:**

* C++
* C
* Java
* C#
* Python3
* Javascript

|  |
| --- |
| // C++ program to show Indirect Recursion  #include <iostream>  using namespace std;    void funB(int n);    void funA(int n)  {      if (n > 0) {          cout <<" "<< n;            // fun(A) is calling fun(B)          funB(n - 1);      } |

}

void funB(int n)

{

    if (n > 1) {

        cout <<" "<< n;

        // fun(B) is calling fun(A)

        funA(n / 2);

    }

}

|  |
| --- |
| / Driver code  int main()  {      funA(20);      return 0;  }    // this code is contributed by shivanisinghss2110 |

**Output**

20 19 9 8 4 3 1

Data Abstraction in C++ means providing only the essential details to the outside world and hiding the internal details, i.e., hiding the background details or implementation. Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.

**One Dimensional Arrays in C++**

One-dimensional arrays are like a row of boxes where you can store things where each box can hold one item, such as a number or a word. For example, in an array of numbers, the first box might hold 5, the second 10, and so on. You can easily find or change what's in each box by referring to its position, called an index. Arrays are handy because they let you store lots of related data in one place and access it quickly.

The one dimensional array can be declared as shown below:

## Syntax of 1D Array in C++

The one dimensional array can be declared as shown below:

element\_type array\_name [size]

## Properties of 1D Array in C++

One-dimensional arrays are organized as a sequence of contiguous memory locations, each capable of holding a single data element of the same type

## How to Use 1d Array in C++

Usage of one-dimensional arrays in C++ involves declaring the array, initializing it with values (optional), accessing and modifying elements, and performing various operations on the array.

### ****Declaring an Array****

To declare an array in C++, you specify the data type of the elements followed by the array name and the size of the array in square brackets. For example, to declare an array of integers with five elements:

*int arr[5];*

This creates an array named arr capable of holding five integers.

### ****Initializing the Array****

You can optionally initialize the array with values at the time of declaration or later using a loop or by assigning values individually. For example:

*int arr[5] = {10, 20, 30, 40, 50}; // Initializing with values*

This initializes the array arr with five integers: 10, 20, 30, 40, and 50.

### ****Accessing and Modifying Elements****

Elements in the array are accessed using indices, starting from 0 for the first element. You can access and modify elements using square brackets []. For example:

int value = arr[2]; // Accessing the element at index 2 (30)

*arr[3] = 60; // Modifying the element at index 3 to 60*

### ****Performing Basic Operations****

You can perform various operations on arrays, such as searching for an element, sorting the elements, or calculating the sum of all elements. These operations typically involve iterating through the array using loops and applying the necessary logic. For example, to find the sum of all elements in the array:

*int sum = 0;*

*for (int i = 0; i < 5; i++) {*

*sum += arr[i];*

*}*

This loop iterates through each element of the array and adds it to the sum variable.

## Complexity Analysis of Basic Array Operations

Several basic operations can be performed on one-dimensional arrays in C++, each with its associated time and space complexity:

### ****Accessing an Element by Index****

*Time Complexity: O(1). Since arrays provide constant-time access to elements based on their indices, accessing an element by index takes constant time regardless of the size of the array.*

*Space Complexity: O(1). This operation requires no additional space beyond the memory used by the array itself.*

### ****Modifying an Element by Index****

*Time Complexity: O(1). Similar to accessing an element, modifying an element by index takes constant time regardless of the size of the array.*

*Space Complexity: O(1). No additional space is required beyond the memory used by the array.*

## Example of One Dimensional Array in C++

#include *<iostream>*

**using** **namespace** **std**;

int main() {

*// Declaration and initialization of an array*

int arr[5] = {10, 20, 30, 40, 50};

*// Accessing elements of the array*

cout << "Element at index 2: " << arr[2] << endl;

*// Modifying elements of the array*

arr[3] = 60;

cout << "Modified element at index 3: " << arr[3] << endl;

*// Calculating the sum of all elements*

int sum = 0;

**for** (int i = 0; i < 5; i++) {

sum += arr[i];

}

cout << "Sum of all elements: " << sum << endl;

**return** 0;

}

**Output**

Element at index 2: 30

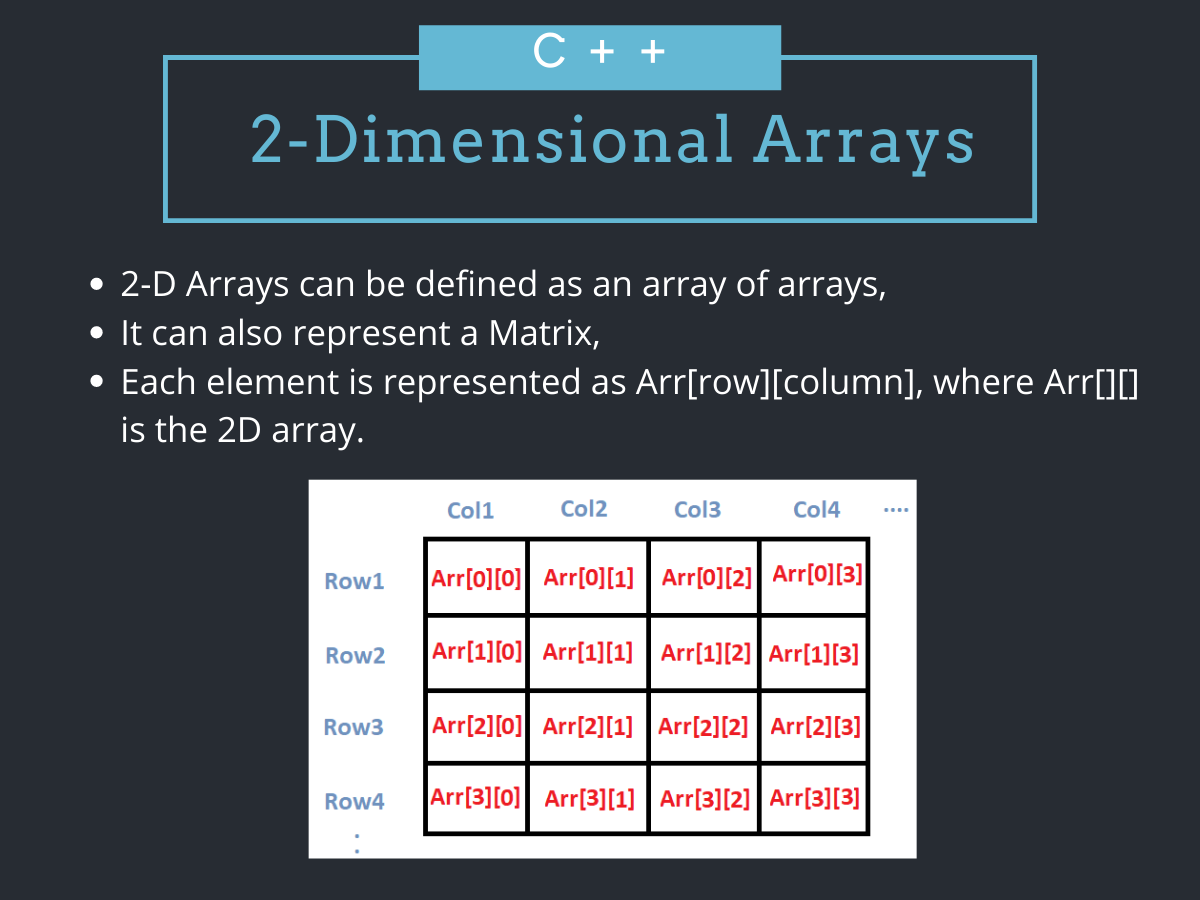
Modified element at index 3: 60

Sum of all elements: 170

## Applications of One Dimensional Array

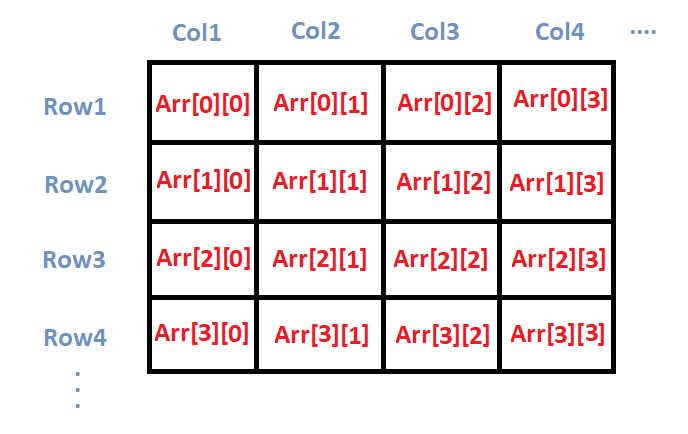
One-dimensional arrays in C++ find wide-ranging applications across various domains due to their simplicity, efficiency, and versatility. Here are some common applications of one-dimensional arrays:

1. Data Storage and Retrieval
2. Numerical Computations
3. Algorithm Design and Implementation
4. Input and Output Operations
5. Memory Management'.



## [Introduction](https://www.digitalocean.com/community/tutorials/two-dimensional-array-in-c-plus-plus" \l "introduction)

A **two-dimensional array** in C++ is the simplest form of a multi-dimensional array. It can be visualized as an array of arrays. The image below depicts a two-dimensional array.



A two-dimensional array is also called a **matrix**. It can be of any type like integer, character, float, etc. depending on the initialization. In the next section, we are going to discuss how we can initialize 2D arrays.

## [Initializing a 2D array in C++](https://www.digitalocean.com/community/tutorials/two-dimensional-array-in-c-plus-plus" \l "initializing-a-2d-array-in-c)

So, how do we initialize a two-dimensional array in C++? As simple as this:

int arr[4][2] = {

{1234, 56},

{1212, 33},

{1434, 80},

{1312, 78}

} ;

So, as you can see, we initialize a 2D array arr, with **4** rows and **2** columns as an array of arrays. Each element of the array is yet again an array of integers.

We can also initialize a **2D** array in the following way.

int arr[4][2] = {1234, 56, 1212, 33, 1434, 80, 1312, 78};

## [Printing a 2D Array in C++](https://www.digitalocean.com/community/tutorials/two-dimensional-array-in-c-plus-plus" \l "printing-a-2d-array-in-c)

We are done initializing a 2D array, now without actually printing the same, we cannot confirm that it was done correctly.

Also, in many cases, we may need to print a resultant 2D array after performing some operations on it. So how do we do that?

The code below shows us how we can do that.

#include<iostream>

using namespace std;

main( )

{

int arr[4][2] = {

{ 10, 11 },

{ 20, 21 },

{ 30, 31 },

{ 40, 41 }

} ;

int i,j;

cout<<"Printing a 2D Array:\n";

for(i=0;i<4;i++)

{

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

{

cout<<"\t"<<arr[i][j];

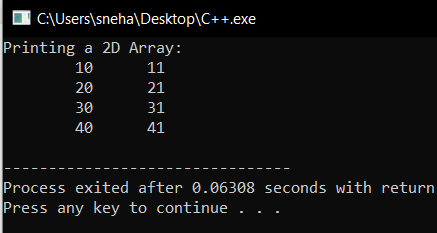
}

cout<<endl;

}

}

OUTPUT



In the above code,

* We firstly initialize a 2D array, arr[4][2] with certain values,
* After that, we try to print the respective array using two for loops,
* the outer for loop iterates over the rows, while the inner one iterates over the columns of the 2D array,
* So, for each iteration of the outer loop, i increases and takes us to the next 1D array. Also, the inner loop traverses over the whole 1D array at a time,
* And accordingly, we print the individual element arr[ i ][ j ].

## [Matrix Addition using Two Dimensional Arrays in C++](https://www.digitalocean.com/community/tutorials/two-dimensional-array-in-c-plus-plus" \l "matrix-addition-using-two-dimensional-arrays-in-c)

#include<iostream>

using namespace std;

main()

{

int m1[5][5], m2[5][5], m3[5][5];

int i, j, r, c;

cout<<"Enter the no.of rows of the matrices to be added(max 5):";

cin>>r;

cout<<"Enter the no.of columns of the matrices to be added(max 5):";

cin>>c;

cout<<"\n1st Matrix Input:\n";

for(i=0;i<r;i++)

{

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

{

cout<<"\nmatrix1["<<i<<"]["<<j<<"]= ";

cin>>m1[i][j];

}

}

cout<<"\n2nd Matrix Input:\n";

for(i=0;i<r;i++)

{

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

{

cout<<"\nmatrix2["<<i<<"]["<<j<<"]= ";

cin>>m2[i][j];

}

}

cout<<"\nAdding Matrices...\n";

for(i=0;i<r;i++)

{

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

{

m3[i][j]=m1[i][j]+m2[i][j];

}

}

cout<<"\nThe resultant Matrix is:\n";

for(i=0;i<r;i++)

{

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

{

cout<<"\t"<<m3[i][j];

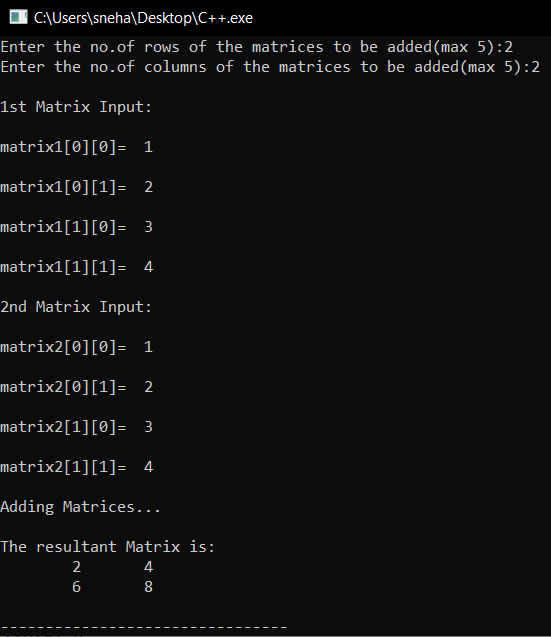
}

cout<<endl;

}

}

**Output**:



Here,

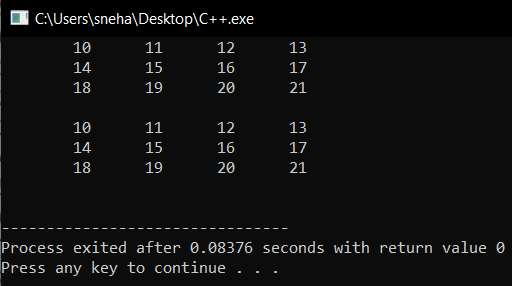
* We take two matrices m1 and m2 with a maximum of 5 rows and 5 columns. And another matrix m3 in which we are going to store the result,
* As user inputs, we took the number of rows and columns for both the matrices. Since we are performing matrix addition, the number of rows and columns should be the same for both the matrices,
* After that, we take both the matrices as user inputs, again using nested for loops,
* At this point, we have both the matrices **m1** and **m2**,

then we traverse through the **m3** matrix, using two for loops and update the respective elements m3[ i ][ j ] by the value of m1[i][j]+m2[i][j]. In this way, by the end of the outer for loop, we get our desired matrix,

* At last, we print out the resultant matrix m3.

## [Passing 2-D Array to a Function](https://www.digitalocean.com/community/tutorials/two-dimensional-array-in-c-plus-plus" \l "passing-2-d-array-to-a-function)

* we are going to learn how to pass a **2D** array to any **function** and access the corresponding elements. In the code below, we pass the array **a**, to two functions show() and print() which prints out the passed 2D array.
* #include<iostream>
* using namespace std;
* void show(int (\*q)[4], int row, int col)
* {
* int i, j ;
* for(i=0;i<row;i++)
* {
* for(j=0;j<col;j++)
* cout<<"\t"<<\*(\*(q + i)+j);
* cout<<"\n";
* }
* cout<<"\n";
* }
* void print(int q[][4], int row, int col)
* {
* int i, j;
* for(i=0;i<row;i++)
* {
* for(j=0;j<col;j++)
* cout<<"\t"<<q[i][j];
* cout<<"\n";
* }
* cout<<"\n";
* }
* int main()
* {
* int a[3][4] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21} ;
* show (a, 3, 4);
* print (a, 3, 4);
* return 0;
* }
* OUTPUT



Here,

* In the show( ) function we have defined q to be a pointer to an array of **4** integers through the declaration int (\*q)[4],
* **q** holds the base address of the zeroth 1-D array
* This address is then assigned to **q**, an int pointer, and then using this pointer all elements of the zeroth 1D array are accessed.
* Next time through the loop when i takes a value **1**, the expression **q+i** fetches the address of the first 1-D array. This is because q is a pointer to the zeroth 1-D array and adding 1 to it would give us the address of the next 1-D array. This address is once again assigned to q and using it all elements of the next 1-D array are accessed
* In the second function print(), the declaration of **q** looks like this: int q[][4] ,

This is same as **int (\*q )[4]**, where q is a pointer to an array of 4 integers. The only advantage is that we can now use the more familiar expression q[i][j] to access array elements. We could have used the same expression in show() as well but for better understanding of the use of pointers, we use pointers to access each element.

# Row Major Order and Column Major Order

Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.  
To find the address of the element using row-major order uses the following formula:

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

*= 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.*

## ****How to find address using Row Major Order?****

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****

If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it’s in column-major order. To find the address of the element using column-major order use the following formula:

***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.*

## ****How to find address using Column Major Order?****

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 Rows 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*

## ****Row Major Order vs Column Major Order****

| **Aspect** | **Row Major Order** | **Column Major Order** |
| --- | --- | --- |
| **Memory Organization** | Elements are stored row by row in contiguous locations. | Elements are stored column by column in contiguous locations. |
| **Memory Layout Example** | For a 2D array A[m][n]: [A[0][0], A[0][1], ..., A[m-1][n-1]] | For the same array: [A[0][0], A[1][0], ..., A[m-1][n-1]] |
| **Traversal Direction** | Moves through the entire row before progressing to the next row. | Moves through the entire column before progressing to the next column. |

|  |  |  |
| --- | --- | --- |
| **Access Efficiency** | Efficient for row-wise access, less efficient for column-wise access. | Efficient for column-wise access, less efficient for row-wise access. |
| **Common Use Cases** | Commonly used in languages like C and C++. | Commonly used in languages like Fortran. |
| **Applications** | Suitable for row-wise operations, e.g., image processing. | Suitable for column-wise operations, e.g., matrix multiplication. |

|  |  |  |
| --- | --- | --- |
| **DYNAMIC ARRAY IN C++** |  |  |

A simple dynamic array can be constructed by allocating an array of fixed-size, typically larger than the number of elements immediately required. The elements of the dynamic array are stored contiguously at the start of the underlying array, and the remaining positions towards the end of the underlying array are reserved, or unused. Elements can be added at the end of a dynamic array in constant time by using the reserved space until this space is completely consumed. When all space is consumed, and an additional element is to be added, the underlying fixed-sized array needs to be increased in size. Typically resizing is expensive because you have to allocate a bigger array and copy over all of the elements from the array you have overgrow before we can finally append our item.

Polynomial is a heavily templated C++ class for polynomials. It provides functionality for polynomial algebra (addition, subtraction, and multiplication) as well as root-finding using either the numerically-accurate companion matrix method or the much faster Sturm sequences method.

# Sparse Matrix and its representations

A [matrix](https://www.geeksforgeeks.org/data-structures/" \l "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
* Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

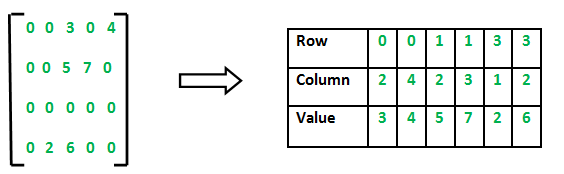
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

**Method 1: Using Arrays:**

2D array is used to represent a sparse matrix in which there are three rows named as

* **Row:**Index of row, where non-zero element is located
* **Column:**Index of column, where non-zero element is located
* **Value:**Value of the non zero element located at index – (row,column)



C++ program for Sparse Matrix Representation

// using Array

#include <iostream>

using namespace std;

int main()

{

    // Assume 4x5 sparse matrix

    int sparseMatrix[4][5] =

    {

        {0 , 0 , 3 , 0 , 4 },

        {0 , 0 , 5 , 7 , 0 },

        {0 , 0 , 0 , 0 , 0 },

        {0 , 2 , 6 , 0 , 0 }

    };

    int size = 0;

    for (int i = 0; i < 4; i++)

        for (int j = 0; j < 5; j++)

            if (sparseMatrix[i][j] != 0)

                size++;

int compactMatrix[3][size];

    // Making of new matrix

    int k = 0;

    for (int i = 0; i < 4; i++)

        for (int j = 0; j < 5; j++)

            if (sparseMatrix[i][j] != 0)

            {

                compactMatrix[0][k] = i;

                compactMatrix[1][k] = j;

                compactMatrix[2][k] = sparseMatrix[i][j];

                k++;

            }

    for (int i=0; i<3; i++)

    {

        for (int j=0; j<size; j++)

            cout <<" "<< compactMatrix[i][j];

        cout <<"\n";

    }

    return 0;

}

**Output**

0 0 1 1 3 3

2 4 2 3 1 2

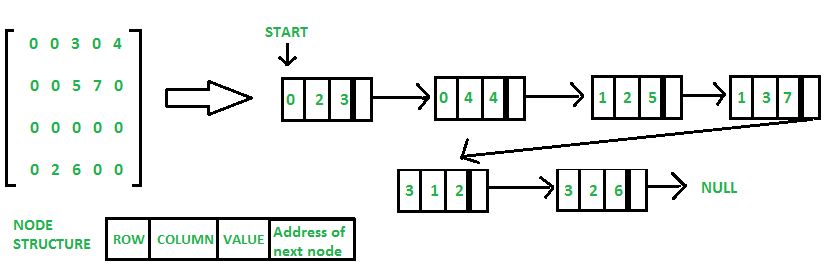
3 4 5 7 2 6

**Time Complexity:**O(NM), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.

**Auxiliary Space:** O(NM), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.

**Method 2: Using Linked Lists**  
In linked list, each node has four fields. These four fields are defined as:

* **Row:**Index of row, where non-zero element is located
* **Column:**Index of column, where non-zero element is located
* **Value:**Value of the non zero element located at index – (row,column)
* **Next node:**Address of the next node



#include <iostream>

using namespace std;

class Node

{

public:

int row;

int col;

int val;

Node \*next;

Node(int r,int c, int v)

{

// newnode=(struct \*Node)malloc(sizeof(struct Node));

row=r;

col=c;

val=v;

next=NULL;

}

};

class SparseList

{

private:

Node \*head;

public:

SparseList()

{

head=NULL;

}

void createSparse(int row, int col,int val)

{

Node \*newnode=new Node(row,col,val);

if(head==NULL)

head=newnode;

else

{

Node \*temp=head;

while(temp->next!=NULL)

{

temp=temp->next;

}

temp->next=newnode;

}

}

void showSparse()

{

Node \*temp=head;

while(temp!=NULL)

{

cout <<temp->row<<" "<<temp->col<< " "<<temp->val;

cout<<endl;

temp=temp->next;

}

}

};

int main() {

SparseList sp;

int m,n;

cout << "enter dimensions of matrix"<<endl;

cin>>m>>n;

int A[m][n];

cout << "enter matrix"<<endl;

for(int i = 0; i < m; i++)

{

for(int j = 0; j < n; j++)

{

cin>>A[i][j];

}

}

for(int i = 0; i < m; i++)

{

for(int j = 0; j < n; j++)

{

if(A[i][j]!=0)

sp.createSparse(i,j,A[i][j]);

}

}

cout << "Sparse matrix using linked list"<<endl;

sp.showSparse();

return 0;

}**Output**

# enter dimensions of matrix

# 4

# 4

# enter matrix

# 0 0 3 0

# 0 0 0 4

# 0 0 0 0

# 0 6 8 0

# Sparse matrix using linked list

# 0 2 3

# 1 3 4

# 3 1 6

# 3 2 8

# Time Complexity:  O(N\*M), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix. Auxiliary Space: O(K), where K is the number of non-zero elements in the array.

# Array of Linked Lists in C/C++

An [array](https://www.geeksforgeeks.org/introduction-to-arrays/) in [C/C++](https://www.geeksforgeeks.org/c-plus-plus/) or be it in any [programming language](https://www.geeksforgeeks.org/top-10-programming-languages-of-the-world-2019-to-begin-with/) is a collection of similar data items stored at contiguous memory locations and elements that can be accessed randomly using indices of an array.  They can be used to store the collection of [primitive data types](https://www.geeksforgeeks.org/c-data-types/) such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store [derived data types](https://www.geeksforgeeks.org/derived-data-types-in-c/) such as structures, pointers, etc. Given below is the picture representation of an array.

an array is a container that can hold a fixed number of elements and these elements should be of the same type. Most of the data structures make use of arrays to implement their algorithms.



 A [linked list](https://www.geeksforgeeks.org/data-structures/linked-list/) is a linear data structure consisting of nodes where each node contains a reference to the next node. To create a link list we need a [pointer](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/) that points to the first node of the list.

**Approach:** To create an array of linked lists below are the main requirements:

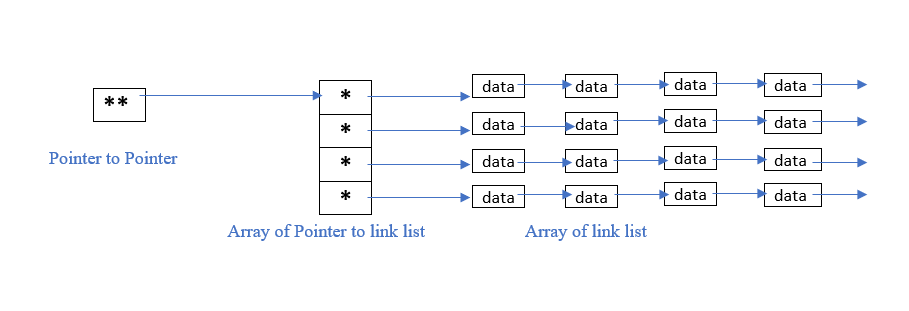
* [An array of pointers](https://www.geeksforgeeks.org/creating-array-of-pointers-in-cpp/).
* For keeping the track of the above-created array of pointers then another pointer is needed that points to the first pointer of the array. This pointer is called [pointer to pointer](https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/).

To create a link list we need a [pointer](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/) that points to the first node of the list.

**Approach:** To create an array of linked lists below are the main requirements:

* [An array of pointers](https://www.geeksforgeeks.org/creating-array-of-pointers-in-cpp/).
* For keeping the track of the above-created array of pointers then another pointer is needed that points to the first pointer of the array. This pointer is called [pointer to pointer](https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/).

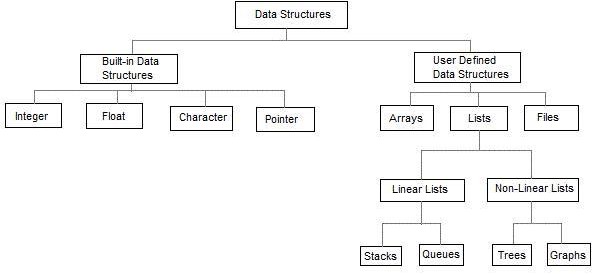
Below is the pictorial representation of the array of linked lists:



**Data structure** A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways

Abstract Data Type

In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is usually a n object of that class. In ADT all the implementation details are hidden



Linear data structures are the data structures in which data is arranged in a list or in a sequence.

Non linear data structures are the data structures in which data may be arranged in a hierarchic al manner

**LIST ADT**

List is basically the collection of elements arrange d in a sequential manner. In memory we can store the list in two ways: one way is we can store the elements in sequential memory locations. That means we can store the list in arrays.

The other way is we can use pointers or links to associate elements sequentially. This is known as linked list.

**LINKED LISTS**

The linked list is very different type of collection from an array. Using such lists, we can store collections of information limited only by the total amount of memory that the OS will allow us to use.Further more, there is no need to specify our needs in advance. The linked list is very flexible dynamic data structure : items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate in advance. This allows us to write robust programs which require much less maintenance.

**The linked allocation has the following draw backs:**

1. No direct access to a particular element.
2. Additional memory required for pointers.

**Linked list are of 3 types:**

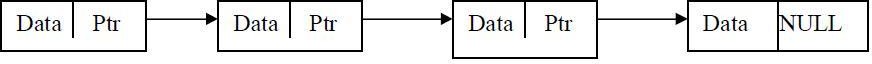
1. Singly Linked List
2. Doubly Linked List
3. Circularly Linked List

SINGLY LINKED LIST

A singly linked list, or simply a linked list, is a linear collection of data items. The linear order is given by means of POINTERS. These types of lists are often referred to as **linear linked list**.

* + Each item in the list is called a node.
  + Each node of the list has two fields:

1. Information- contains the item being stored in the list.
2. Next address- contains the address of the next item in the list.

\* The last node in the list contains NULL pointer to indicate that it is the end of the list. Conceptual view of Singly Linked List

Operations on Singly linked list:

Insertion of a node Deletions of a node Traversing the list



**Structure of a node: Method -1:**

struct node

link

Data

{

int data;

struct node \*link;

};

**Method -2:**

class node

{

public:

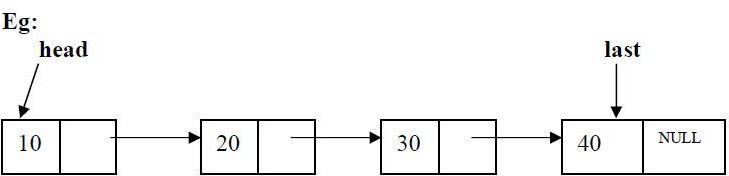
int data; node \*link;

};

**Insertions:** To place an elements in the list there are 3 cases :

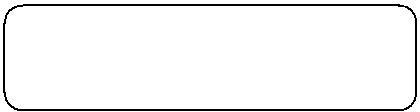
1. At the beginning
2. End of the list
3. At a given position

**case 1:Insert at the beginning**



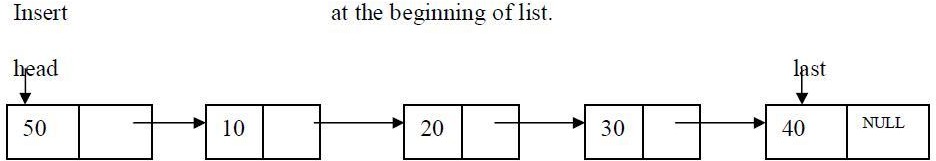
temp

**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is



temp->link=head; head=temp;

**After insertion:**

****

**Code for insert front:-**

template <class T>

void list<T>::insert\_front()

{

struct node <T>\*t,\*temp; cout<<"Enter data into node:"; cin>>item; temp=create\_node(item); if(head==NULL)

head=temp;

else

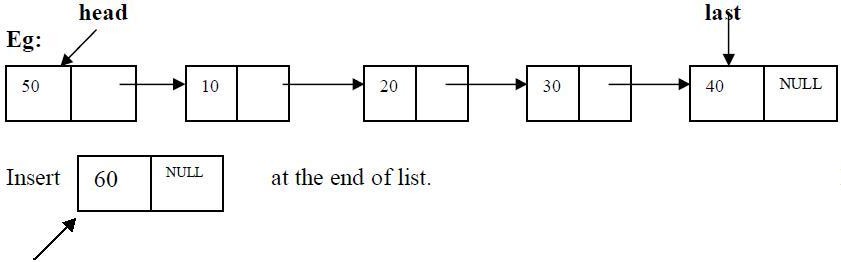
{temp->link=head;

head=temp;

}

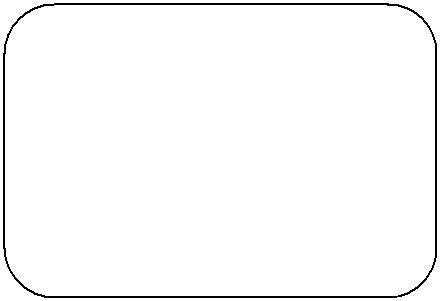
}

**case 2:**Inserting end of the list



temp

**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is



t=head;

while(t->link!=NULL)

{

t=t->link;

}

t->link=temp;

After insertion the linked list is

**code for insert at End:-**

template <class T>

void list<T>::insert\_end()

{

struct node<T> \*t,\*temp; int n;

cout<<"Enter data into node:"; cin>>n;

temp=create\_node(n); if(head==NULL)

head=temp;

else

{t=head; while(t-

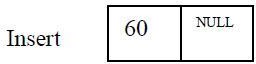
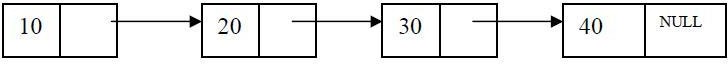
>link!=NULL)

t=t->link; t->link=temp;

}

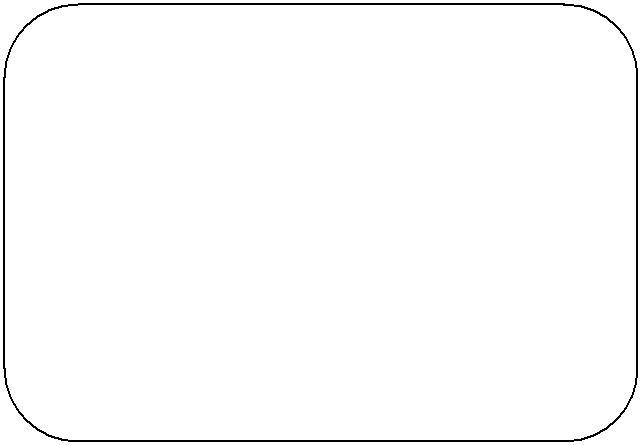
}

**case 3: Insert at a position**

****

insert node at position 3

**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is



c=1;

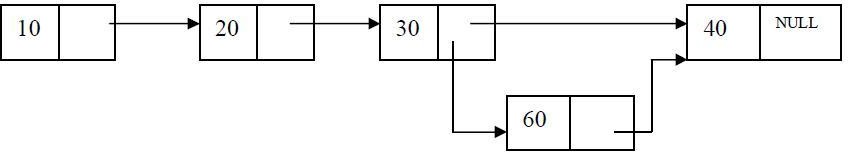
while(c<pos)

{

prev=cur; cur=cur->link; c++;

}

prev->link=temp; temp->link=cur;



**Code for inserting a node at a given position:-**

template <class T>

void list<T>::Insert\_at\_pos(int pos)

{struct node<T>\*cur,\*prev,\*temp; int c=1;

cout<<"Enter data into node:"; cin>>item temp=create\_node(item); if(head==NULL)

head=temp;

else

{

prev=cur=head; if(pos==1)

{

}

else

{

temp->link=head; head=temp;

while(c<pos)

{c++;

prev=cu r;

cur=cur->link;

}

prev->link=temp; temp->link=cur;

}

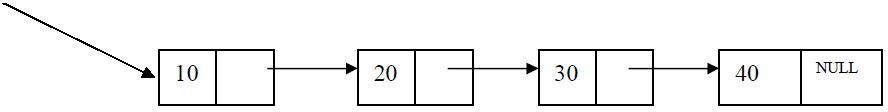
}

}

**Deletions:** Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

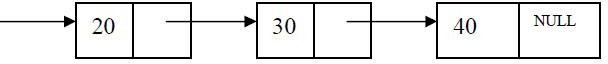
**Case 1**: Delete a node at beginning of the list head

**head** is the pointer variable which contains address of the first node

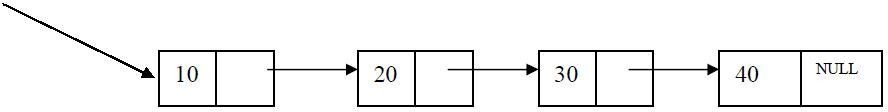
sample code is

t=head; head=head->link;

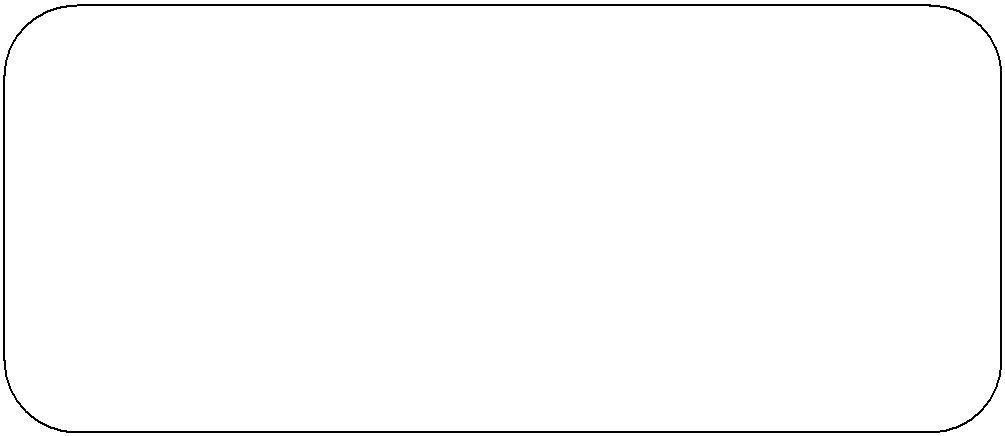
cout<<"node "<<t->data<<" Deletion is sucess"; delete(t);

head

**Case 2. Delete a node at end of the list**

head

To delete last node , find the node using following code



struct node<T>\*cur,\*prev; cur=prev=head;

while(cur->link!=NULL)

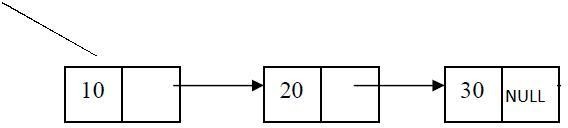
{prev=cur; cur=cur-

>link;

}

prev->link=NULL;

cout<<"node "<<cur->data<<" Deletion is sucess"; free(cur);

head

**Code for deleting a node at end of the list**

template <class T>

void list<T>::delete\_end()

{

struct node<T>\*cur,\*prev; cur=prev=head; if(head==NULL)

cout<<"List is Empty\n";

else

{cur=prev=head; if(head-

>link==NULL)

{

else

cout<<"node "<<cur->data<<" Deletion is sucess"; free(cur);

head=NULL;

}

{while(cur->link!=NULL)

{prev=cur; cur=cur-

>link;

}

prev->link=NULL;

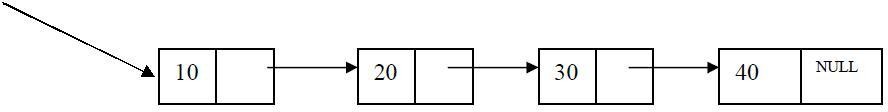
cout<<"node "<<cur->data<<" Deletion is sucess"; free(cur);

}

}

}

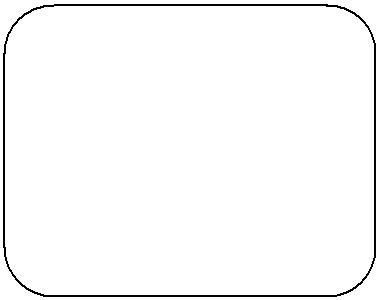
**CASE 3. Delete a node at a given position**

head

Delete node at position 3

**head** is the pointer variable which contains address of the first node. Node to be deleted is node

containing value 30. Finding node at position 3



c=1;

while(c<pos)

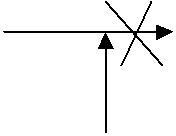
{c++;

prev=cu r;

cur=cur->link;

}

prev cur



30

10

20

NULL

40

cur is the node to be deleted . before deleting update links code to update links

prev->link=cur->link;

cout<<cur->data <<"is deleted successfully"; delete cur;

prev

10

20

30

NULL

40

**Traversing the list:** Assuming we are given the pointer to the head of the list, how do we get the end of the list.

template <class T> void list<T>:: display()

{

struct node<T>\*t;

if(head==NULL)

{

cout<<"List is Empty\n";

}

else

{t=head;

while(t!=NUL L)

{cout<<t->data<<"->"; t=t->link;

}

}

Example for SLL

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* next;

Node(int value) {

data = value;

next = nullptr;

}

};

// Singly Linked List Class

class LinkedList {

private:

Node\* head;

public:

// Constructor

LinkedList() {

head = nullptr;

}

// Insert at the end

void insert(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

}

// Insert at the beginning

void insertAtBeginning(int value) {

Node\* newNode = new Node(value);

newNode->next = head;

head = newNode;

}

// Delete a node by value

void deleteNode(int value) {

if (!head) return;

if (head->data == value) {

Node\* temp = head;

head = head->next;

delete temp;

return;

}

Node\* temp = head;

while (temp->next && temp->next->data != value) {

temp = temp->next;

}

if (temp->next) {

Node\* toDelete = temp->next;

temp->next = temp->next->next;

delete toDelete;

}

}

// Search for a value

bool search(int value) {

Node\* temp = head;

while (temp) {

if (temp->data == value) return true;

temp = temp->next;

}

return false;

}

// Display the linked list

void display() {

Node\* temp = head;

while (temp) {

cout << temp->data << " -> ";

temp = temp->next;

}

cout << "NULL" << endl;

}

// Destructor to free memory

~LinkedList() {

Node\* temp;

while (head) {

temp = head;

head = head->next;

delete temp;

}

}

};

// Main Function

int main() {

LinkedList list;

// Inserting elements at the end

list.insert(10);

list.insert(20);

list.insert(30);

list.insert(40);

// Displaying the list

cout << "Linked List: ";

list.display();

// Insert at the beginning

list.insertAtBeginning(5);

cout << "After inserting 5 at beginning: ";

list.display();

// Searching for an element

int searchVal = 20;

if (list.search(searchVal))

cout << searchVal << " found in the list." << endl;

else

cout << searchVal << " not found in the list." << endl;

// Deleting an element

list.deleteNode(20);

cout << "After deleting 20: ";

list.display();

return 0;

}

DOUBLY LINKED LIST

A singly linked list has the disadvantage that we can only traverse it in one direction. Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list. The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element in the list. The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields.

1. One to store data
2. Remaining are self referential pointers which points to previous and next nodes in the list

|  |  |  |
| --- | --- | --- |
| prev | data | next |

**Implementation of node using structure Method -1:**

struct node

{

int data;

struct node \*prev; struct node \* next;

};

**Implementation of node using class Method -2:**

class node

{

public:

int data; node \*prev; node \* next;

};



30

20

10

NULL

NUL L

Operations on Doubly linked list: Insertion of a node Deletions of a node Traversing the list



**Doubly linked list ADT:**

template <class T> class dlist

{

public:

int data;

struct dnode<T>\*head;

dlist()

{

head=NULL;

}

void display();

struct dnode<T>\*create\_dnode(int n); void insert\_end();

void insert\_front(); void delete\_end(); void delete\_front(); void dnode\_count();

void Insert\_at\_pos(int pos); void Delete\_at\_pos(int pos);

};

**Insertions:** To place an elements in the list there are 3 cases

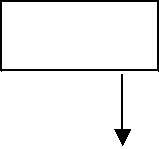
* 1. At the beginning



* 1. End of the list
  2. At a given position

**case 1:Insert at the beginning**

**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample cod



10

40

20

head

30

temp->next=head; head->prev=temp; head=temp;

NUL L

**Code for insert front:-**

template <class T>

void DLL<T>::insert\_front()

{

struct dnode <T>\*t,\*temp; cout<<"Enter data into node:"; cin>>data; temp=create\_dnode(data); if(head==NULL)

head=temp;

else

{

}

}

temp->head=temp;

head ->prev=temp;

head=temp;

**Code to insert a node at End:-**

template <class T>

void DLL<T>::insert\_end()

{

struct dnode<T> \*t,\*temp; int n;

cout<<"Enter data into dnode:"; cin>>n;

temp=create\_dnode(n); if(head==NULL)

head=temp;

else

{t=head; while(t-

>next!=NULL)

t=t->next; t->next=temp;

temp->prev=t;

}

}

**Code to insert a node at a position**

template <class T>

void dlist<T>::Insert\_at\_pos(int pos)

{

struct dnode<T>\*cr,\*pr,\*temp; int count=1;

cout<<"Enter data into dnode:"; cin>>data; temp=create\_dnode(data); display();

if(head==NULL)

{//when list is empty

head=temp;

}

else

{pr=cr=head;

if(pos==1)

{//inserting at pos=1 temp-

>next=head; head=temp;

}

else

{

while(count<pos)

{count++;

pr=cr;

cr=cr->next;

}

pr->next=temp; temp->prev=pr; temp->next=cr; cr->prev=temp;

}

}

}

**Deletions:** Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

Case 1: Delete a node at beginning of the list

head



30

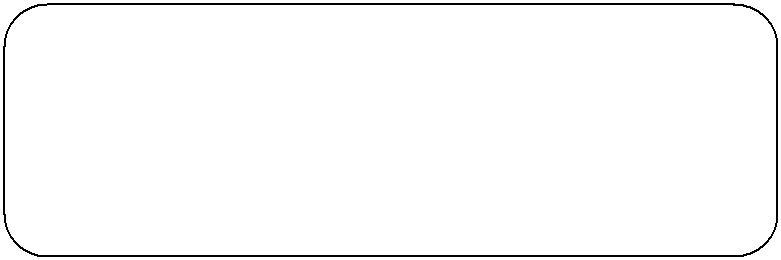
20

10

NUL L

NUL L

**head** is the pointer variable which contains address of the first node sample code is



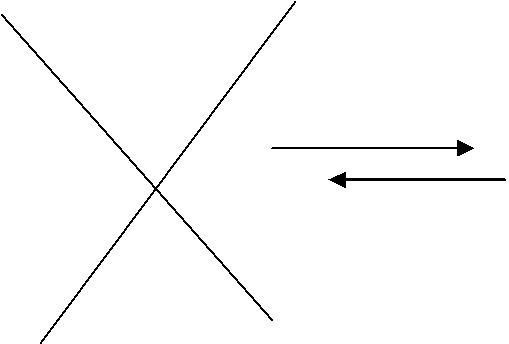
t=head; head=head->next;

head->prev=NULL;

cout<<"dnode "<<t->data<<" Deletion is sucess"; delete(t);

20

**code for deleting a node at front**



head

10

NULL

NULL



NUL L

30

template <class T>

void dlist<T>:: delete\_front()

{struct dnode<T>\*t;

if(head==NULL)

cout<<"List is Empty\n";

else

{t=head;

head=head->next; head->prev=NULL;

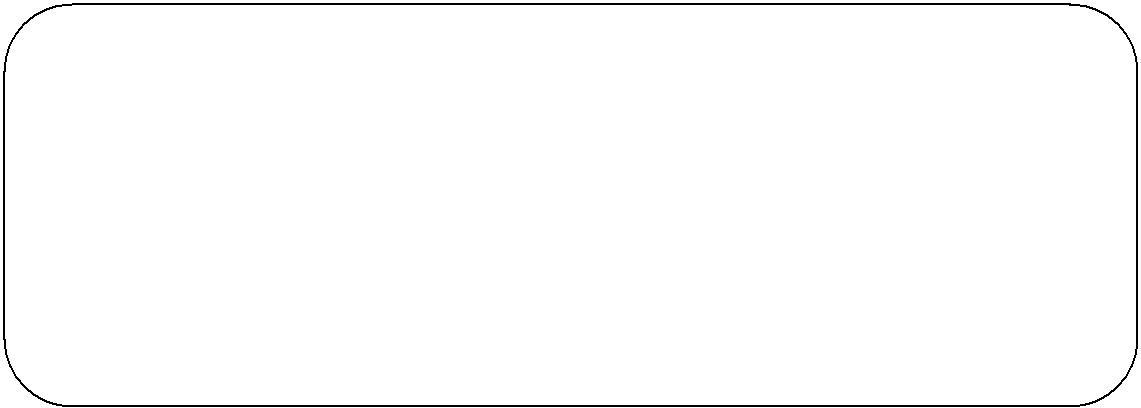
cout<<"dnode "<<t->data<<" Deletion is sucess"; delete(t);

}

}

**Case 2. Delete a node at end of the list**

To deleted the last node find the last node. find the node using following code



struct dnode<T>\*pr,\*cr; pr=cr=head;

while(cr->next!=NULL)

{pr=cr; cr=cr-

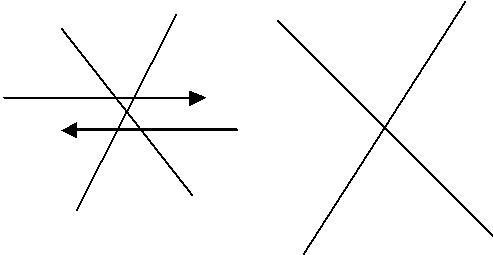
>next;

}

pr->next=NULL;

cout<<"dnode "<<cr->data<<" Deletion is sucess"; delete(cr);

head



20

NULL

30

NULL

cr

10

NULL

pr

**code for deleting a node at end of the list**

template <class T>

void dlist<T>::delete\_end()

{

struct dnode<T>\*pr,\*cr; pr=cr=head; if(head==NULL)

cout<<"List is Empty\n";

else

{cr=pr=head; if(head-

>next==NULL)

{

}

else

cout<<"dnode "<<cr->data<<" Deletion is sucess"; delete(cr);

head=NULL;

{while(cr->next!=NULL)

{pr=cr;

cr=cr->next;

}

pr->next=NULL;

cout<<"dnode "<<cr->data<<" Deletion is sucess"; delete(cr);

}

}

}

**CASE 3. Delete a node at a given position**

head



|  |  |  |
| --- | --- | --- |
| NULL | 10 |  |

|  |  |  |
| --- | --- | --- |
|  | 30 |  |

|  |  |  |
| --- | --- | --- |
|  | 20 | NULL |

Delete node at position 2

**head** is the pointer variable which contains address of the first node. Node to be deleted is node

containing value 30. Finding node at position 2.

while(count<pos)

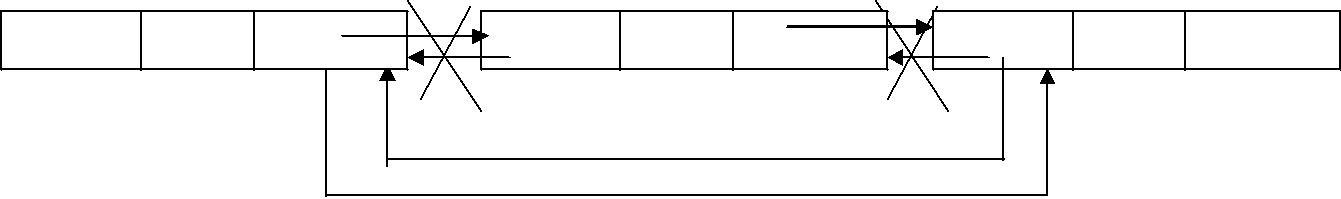
{pr=cr; cr=cr-

>next; count++;

}

pr->next=cr->next; cr->next->prev=pr;

head



NULL

10

30

20

NUL L

cr

pr

Example for DLL

#include <iostream>

using namespace std;

// Node structure for Doubly Linked List

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int value) {

data = value;

prev = nullptr;

next = nullptr;

}

};

// Doubly Linked List Class

class DoublyLinkedList {

private:

Node\* head;

Node\* tail;

public:

// Constructor

DoublyLinkedList() {

head = nullptr;

tail = nullptr;

}

// Insert at the beginning

void insertAtBeginning(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = tail = newNode;

return;

}

newNode->next = head;

head->prev = newNode;

head = newNode;

}

// Insert at the end

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = tail = newNode;

return;

}

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

// Delete a node by value

void deleteNode(int value) {

if (!head) return; // Empty list

Node\* temp = head;

// If head node is to be deleted

if (head->data == value) {

head = head->next;

if (head) head->prev = nullptr;

delete temp;

return;

}

// Searching for the node to delete

while (temp && temp->data != value) {

temp = temp->next;

}

if (!temp) return; // Value not found

// If node is tail

if (temp == tail) {

tail = tail->prev;

tail->next = nullptr;

delete temp;

return;

}

// If node is in the middle

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

delete temp;

}

// Display list forward

void displayForward() {

Node\* temp = head;

while (temp) {

cout << temp->data << " <-> ";

temp = temp->next;

}

cout << "NULL" << endl;

}

// Display list backward

void displayBackward() {

Node\* temp = tail;

while (temp) {

cout << temp->data << " <-> ";

temp = temp->prev;

}

cout << "NULL" << endl;

}

// Destructor to free memory

~DoublyLinkedList() {

Node\* temp;

while (head) {

temp = head;

head = head->next;

delete temp;

}

}

};

// Main Function

int main() {

DoublyLinkedList dll;

// Insert elements

dll.insertAtBeginning(10);

dll.insertAtBeginning(5);

dll.insertAtEnd(20);

dll.insertAtEnd(30);

// Display list forward and backward

cout << "Doubly Linked List (Forward): ";

dll.displayForward();

cout << "Doubly Linked List (Backward): ";

dll.displayBackward();

// Delete a node

dll.deleteNode(20);

cout << "After deleting 20: ";

dll.displayForward();

return 0;

}

Linked lists in C++ are used in various real-time applications where dynamic memory allocation, efficient insertions and deletions, and flexible data structures are required. Here are some real-time applications:

### 1. ****Memory Management****

* Operating systems use linked lists to manage free and allocated memory blocks.
* The heap memory allocator (e.g., malloc and free) often utilizes linked lists to track memory blocks.

### 2. ****Undo/Redo Functionality****

* Applications like text editors (Notepad, Word, etc.) use linked lists to implement undo/redo operations by maintaining a history of changes.

### 3. ****Browser's Forward and Backward Navigation****

* Doubly linked lists help implement the forward and backward navigation system in web browsers.

### 4. ****Music and Video Playlists****

* Circular linked lists are used to maintain playlists where users can move forward and backward seamlessly.

### 5. ****Graph Implementations (Adjacency List)****

* Many graph algorithms use linked lists to store adjacent nodes, making traversal and operations more efficient.

### 6. ****Hash Tables (Chaining for Collision Handling)****

* Linked lists are used in hash tables to handle collisions using separate chaining.

### 7. ****Implementation of Stack and Queue****

* Linked lists are used to implement dynamic stacks and queues without the need for resizing, unlike arrays.

### 8. ****File System Directory Management****

* Operating systems use linked lists to organize file directory structures (e.g., hierarchical file systems).

### 9. ****Social Media Feeds****

* Platforms like Facebook, Twitter, and Instagram use linked lists to maintain dynamic news feeds.

### 10. ****Polynomial Arithmetic****

* Polynomial representation and operations (addition, multiplication) use linked lists to store and manipulate coefficients and exponents efficiently.