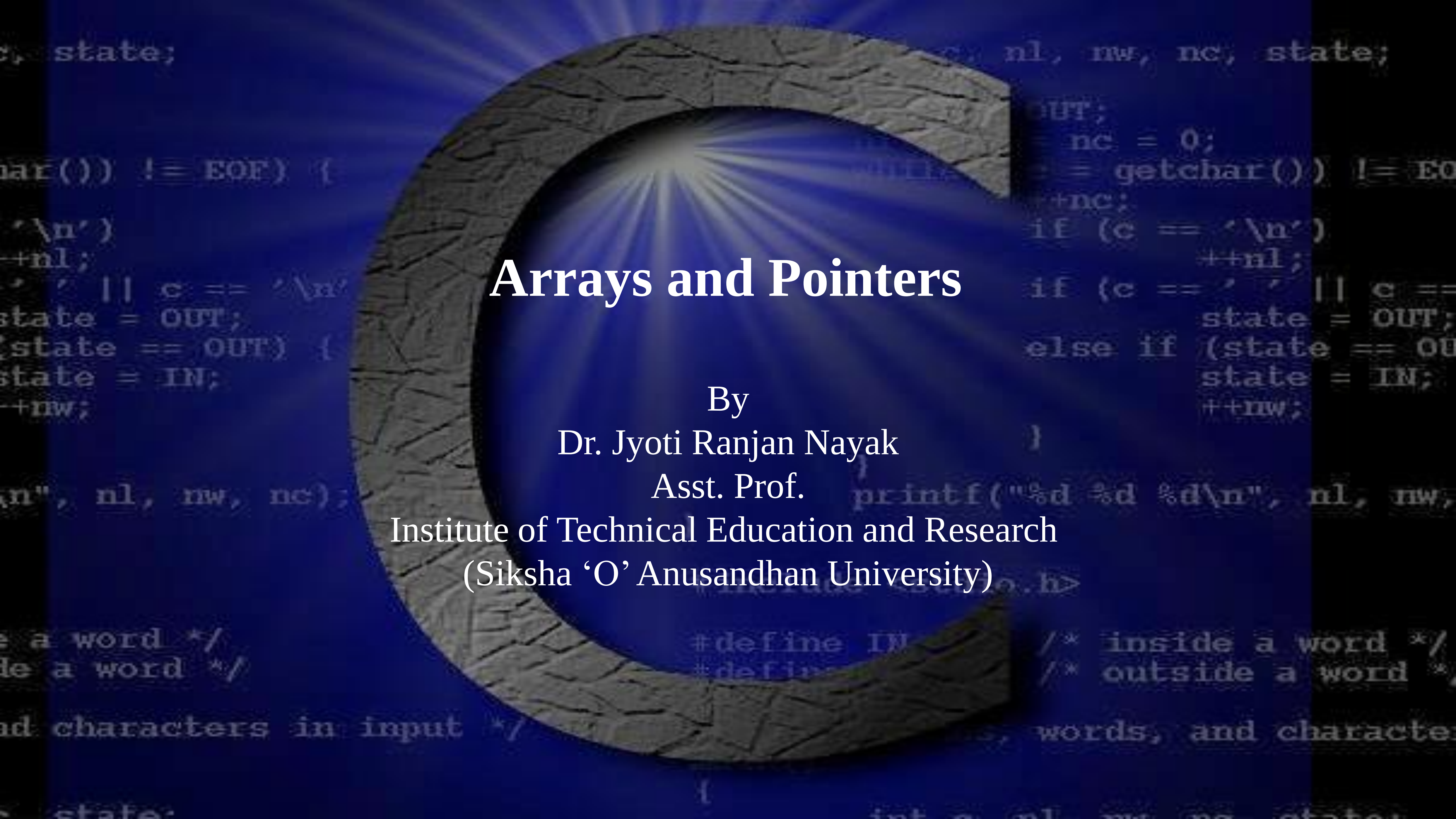


Arrays and Pointers

By
Dr. Jyoti Ranjan Nayak
Asst. Prof.
Institute of Technical Education and Research
(Siksha ‘O’ Anusandhan University)

Array

- An array is defined as the collection of similar type of data items stored at contiguous memory locations.
- These entities or elements can be of int, float, char, or double data type or can be of user-defined data types too like structures.
- However, in order to be stored together in a single array, all the elements should be of the same data type.
- The elements are stored from left to right with the left-most index being the 0th index and the rightmost index being the (n-1) index.

Properties:

i. Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.

ii. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.

iii. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantages of C++ Array

1) ***Code Optimization***: Less code to the access the data.

2) ***Ease of traversing***: By using the for loop, we can retrieve the elements of an array easily.

3) ***Ease of sorting***: To sort the elements of the array, we need a few lines of code only.

4) ***Random Access***: We can access any element randomly using the array.

Disadvantage of C++ Array

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

Types of Array in C++

i. One Dimensional Arrays (1D Array)

ii. Multidimensional Arrays

Declaring Arrays:

- Array is a type consisting of a contiguously allocated nonempty sequence of objects with a particular element type. The number of those objects (the array size) never changes during the array lifetime.
- Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [ ] brackets for each dimension of the array.
- Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets.
- Dimensions used when declaring arrays in C++ must be positive integral constants or constant expressions.

Syntax:

Data_type array_name [array_size];

Where,

Data_type can be int, float, double, char etc.

array_name should be a valid variable name.

array_size is an integer value.

Initializing Arrays:

- ➢ The elements of an array can be initialized by using an initialization list. An initialization list is a comma-separated list of initializers enclosed within braces.
- ➢ An initializer is an expression that determines the initial value of an element of the array.
- ➢ If the type of initializers is not the same as the element type of an array, implicit type casting will be done, if the types are compatible. If types are not compatible, there will be a compilation error. this fact.

Syntax:

Data_type array_name [array_size]={value1, value2, ... valueN};

Where,

Values are of same data type or implicit data type.

- There are different ways through which we can initialize the array as follows.

Method 1: Initialize an array using an Initializer List

- An initializer list initializes elements of an array in the order of the list.

Example-1:

```
int arr[5] = {1, 2, 3, 4, 5};          //a[0]=1, a[1]=2….
```

Example-2:

```
int arr[5];

a[0]=1; a[1]=2; a[2]=3; a[3]=4; a[4]=5;
```

Example-2:

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

1	2	3	0	0

Method 2: Initialize an array in C++ using a for loop

```
#include<iostream>
using namespace std;
int main()
{
   int n;
   cout<<"Enter the number of elements: ";
   cin>>n;
   int a[n];
   for(int i=0;i<n;i++)
   {
      cout<<"Enter the "<< i<<"th element of a: ";
      cin>>a[i];
   }
   return(0);
}
```

Input:

```
Enter the number of elements: 4

Enter the 0th element of a: 14

Enter the 1th element of a: 15

Enter the 2th element of a: 22

Enter the 3th element of a: 24
```

Accessing Array:

- We can access any element of an array in C++ using the array subscript operator [ ] and the index value 'i' of the element.
- One thing to note is that the indexing in the array always starts with 0, i.e., the first element is at index 0 and the last element is at N – 1 where N is the number of elements in the array.

```
array_name [index];
```

```
#include<iostream>
using namespace std;
int main()
{
   //Array declaration and initialization
   int arr[5]={14, 21,11,5,47};
   //Accessing array elements.
   cout<<"Element at arr[0] is "<<arr[0]<<endl;
   cout<<"Element at arr[2] is "<<arr[2]<<endl;
   cout<<"Element at arr[4] is "<<arr[4]<<endl;
   return(0);
}
```

Output:

```
Element at arr[0] is 14
Element at arr[2] is 11
Element at arr[4] is 47
```

- We can access all elements of an array in C++ using loop.

```
#include<iostream>
using namespace std;
int main()
{
   //Array declaration and initialization
   int arr[5]={14, 21,11,5,47};
   //Accessing array elements
   for(int i=0;i<5;i++)
   {
      cout<<arr[i]<<'\t';
   }
   return(0);
}
```

Output:

```
14      21      11      5      47
```

- Array of string.

Syntax:

string array_name[string_numbers] = {string1,string2,. . . };

```
#include<iostream>
using namespace std;
int main()
{
   //Array declaration and initialization
   string arr[5]={"ITER","SOA","Training","C++","Class"};
   //Accessing array elements
   cout<<arr[0]<<endl;
   cout<<arr[2]<<endl;
   cout<<arr[4]<<endl;
   arr[0]="CSE";
   cout<<arr[0]<<endl;
   return(0);
}
```

Output:

```
ITER
Training
Class
CSE
```

Operation on Arrays in C++

There are a number of operations that can be performed on an array which are:

1. Traversal
2. Copying
3. Sorting
4. Searching
5. Reversing
6. Insertion
7. Deletion
8. Merging

1. Traversal

- Traversing basically means the accessing the each and every element of the array at least once. Traversing is usually done to be aware of the data elements which are present in the array.

Algorithm:

```
Step 01: Start
Step 02: [Initialize counter variable. ] Set i = 0 (Indexing starts from 0 to n-1).
Step 03: Repeat for i = 0 to n-1.
Step 04: Apply process to arr[i].
Step 05: [End of loop. ]
Step 06: Stop
```

```
#include<iostream>
using namespace std;
int main()
{
   //Array declaration and initialization
   int arr[5]={14, 21,11,5,47};
   //Accessing array elements
   for(int i=0;i<5;i++)
   {
      cout<<arr[i]<<'\t';
   }
   return(0);
}
```

Output:

```
14      21      11      5       47
```

2. Copying

- Copying an array involves index-by-index copying.
- For this to work we shall know the length of array in advance, which we shall use in iteration.

Algorithm:

```
*Step 01:* Start
*Step 02:* Take two arrays, A and B.
*Step 03:* Store values in A
*Step 04:* Loop for each value of A
*Step 05:* Copy each index value to B array at the same index location
*Step 06:* Stop
```

Copying all elements of A to B.

```
using namespace std;
int main()
{
    int A[5]={5,14,24,63,7};
    int B[5]={15,4,34,13,27};
    for (int i=0;i<5;i++)
    {
        B[i]=A[i];
        cout<<B[i]<<'\t';
    }
    return(0);
}
```

Output:

```
5      14      24      63      7
```

Copying first 3 elements of A to B.

```
#include<iostream>
using namespace std;
int main()
{
    int A[5]={5,14,24,63,7};
    int B[5]={15,4,34,13,27};
    for (int i=0;i<3;i++)
    {
        B[i]=A[i];
    }
    for (int i=0;i<5;i++)
    {
        cout<<B[i]<<'\t';
    }
    return(0);
}
```

Output:

```
5      14      24      13      27
```

3. Sorting

- Sorting is the process of arranging elements in a list or array in a specific order, typically in ascending or descending order.

   i. Selection Sort

   ii. Bubble Sort

   iii. Insertion Sort

i. Selection Sort

- Selection sort is a sorting algorithm that selects the smallest or largest element from an unsorted list in each iteration and places that element at the beginning or end position of the unsorted list.

Algorithm:

Step 1: Set Min to location 0 in Step 1.

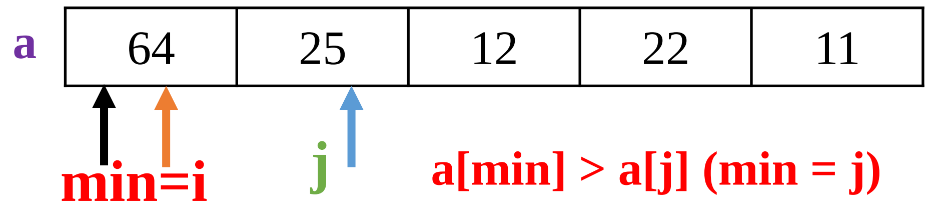
Step 2: Look for the smallest element on the list.

Step 3: Replace the value at location Min with a different value.

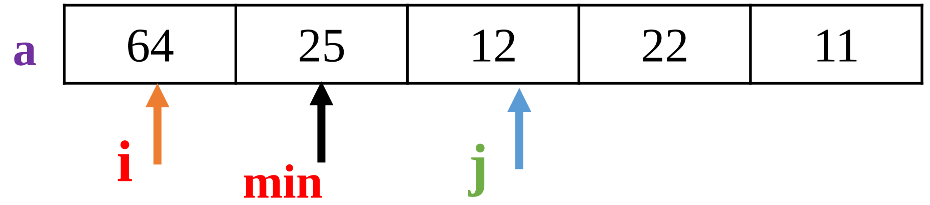
Step 4: Increase Min to point to the next element

Step 5: Continue until the list is sorted.

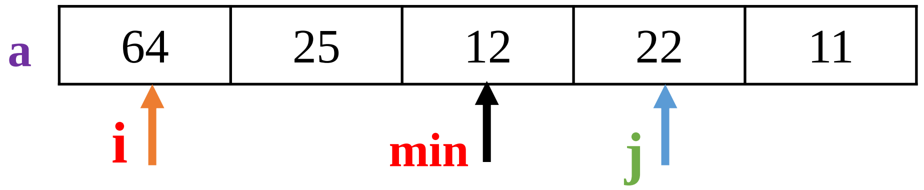
Example: Sorting of an array in ascending order using selection sort



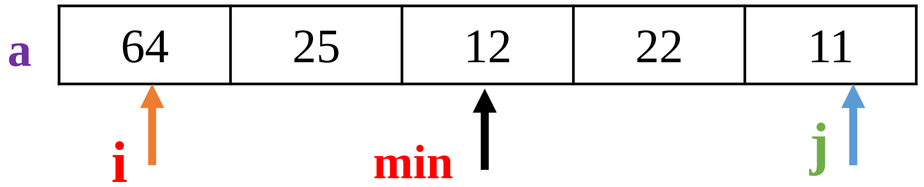
a[min] > a[j] (min = j)



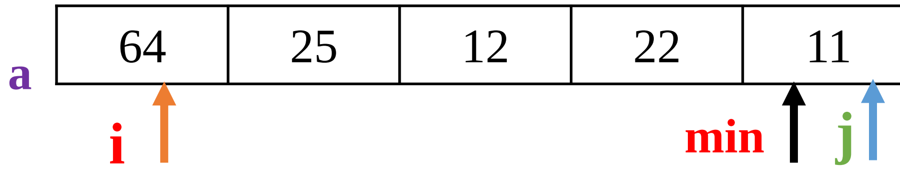
a[min] > a[j] (min = j)



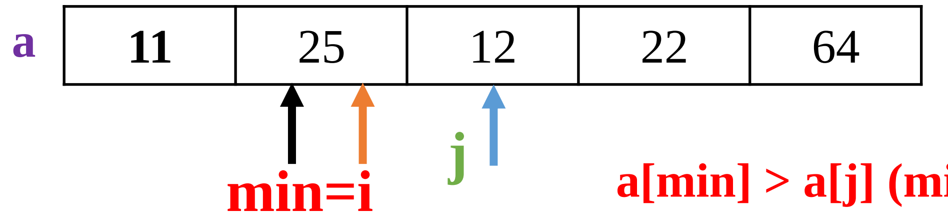
a[min] < a[j] (No change of min)



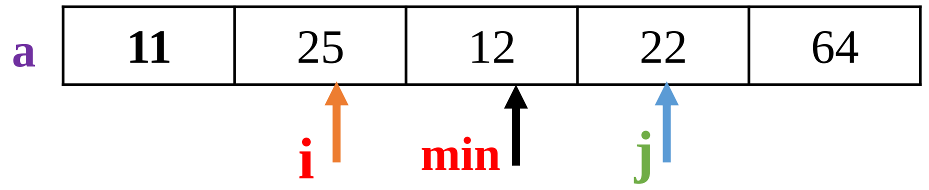
a[min] > a[j] (min=j)



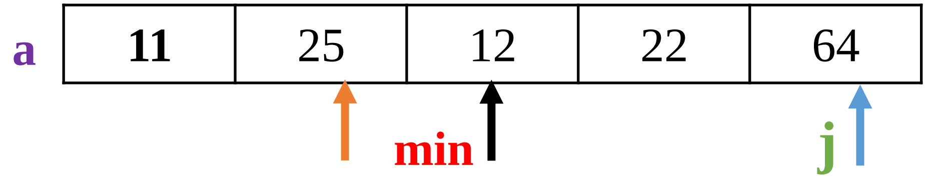
Swap a[i] and a[min]



a[min] > a[j] (min = j)

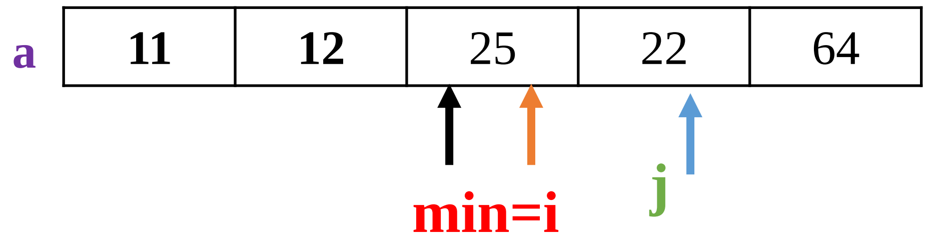


a[min] < a[j] (No change of min)

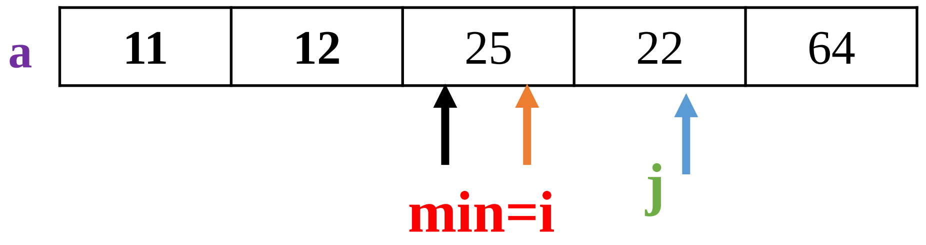


a[min] < a[j] (No change of min)

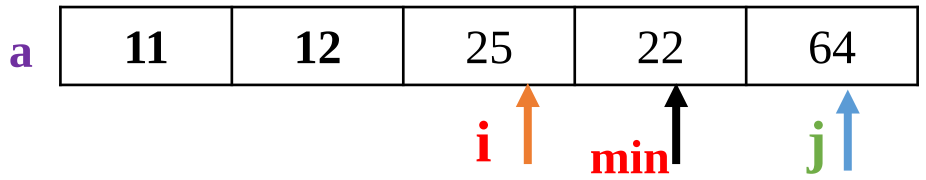
Swap a[i] and a[min]



Repeat the steps upto i < 5

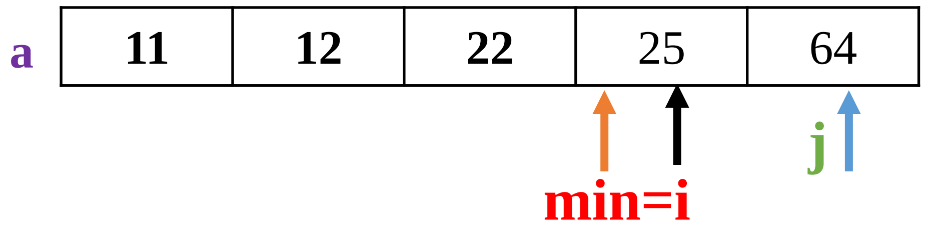


a[min] > a[j] (min=j)



a[min] > a[j] (No change)

Swap a[i] and a[min]



a[min] > a[j] (No change)

11	**12**	**22**	**25**	**64**

```
#include<iostream>
using namespace std;
int main()
{
    int arr[5]={5,14,24,63,7};
    int i,j,min,temp;
    for(i=0;i<4;i++)
    {
        min=i;
   for(j=i+1;j<=4;j++)
        {
            if (arr[min]>arr[j])
            {
                min=j;
            }
        }
            if (arr[min]<arr[i])
        {
            temp=arr[min];
            arr[min]=arr[i];
            arr[i]=temp;
        }
    }
    for (i=0;i<5;i++)
    {
        cout<<arr[i]<<'\t';
    }
    return(0);
}
```

Output:

```
5       7       14      24      63
```

ii. Bubble Sort

- Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

In this algorithm,

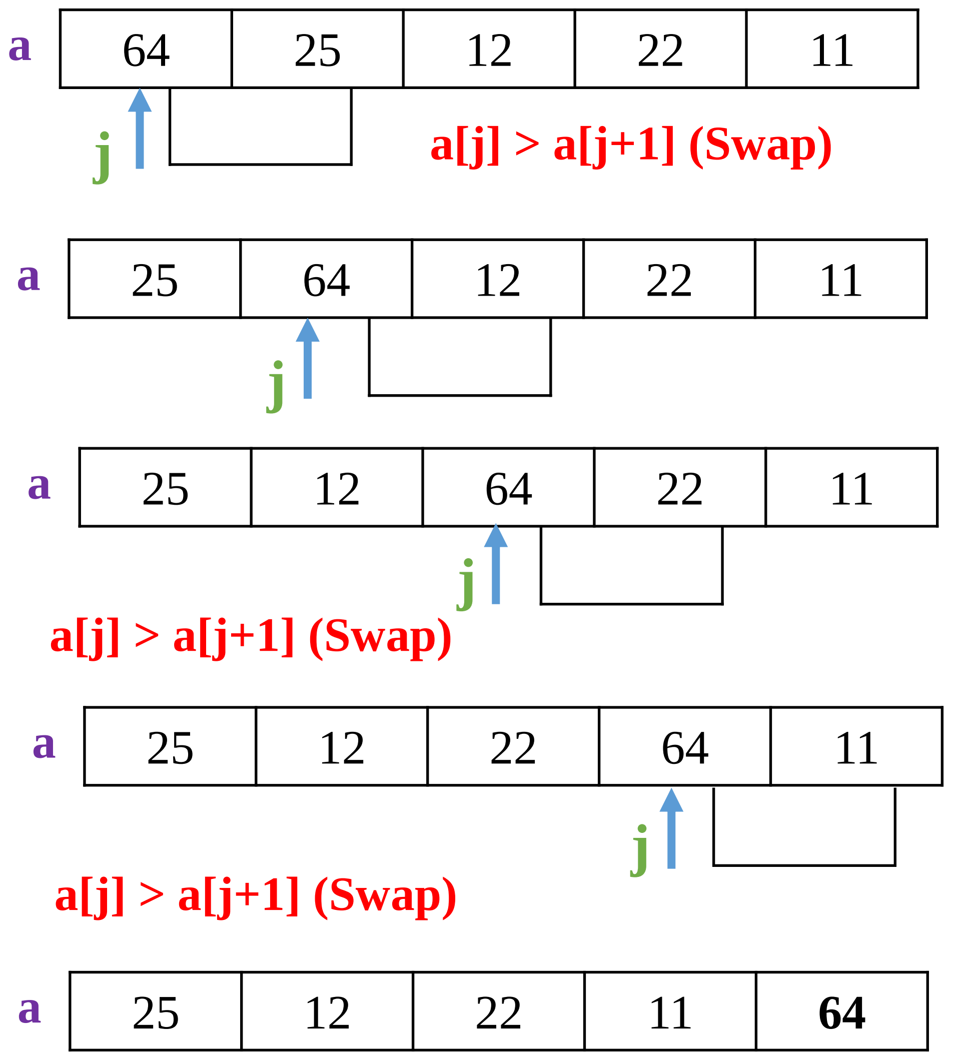
- Traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

Algorithm:

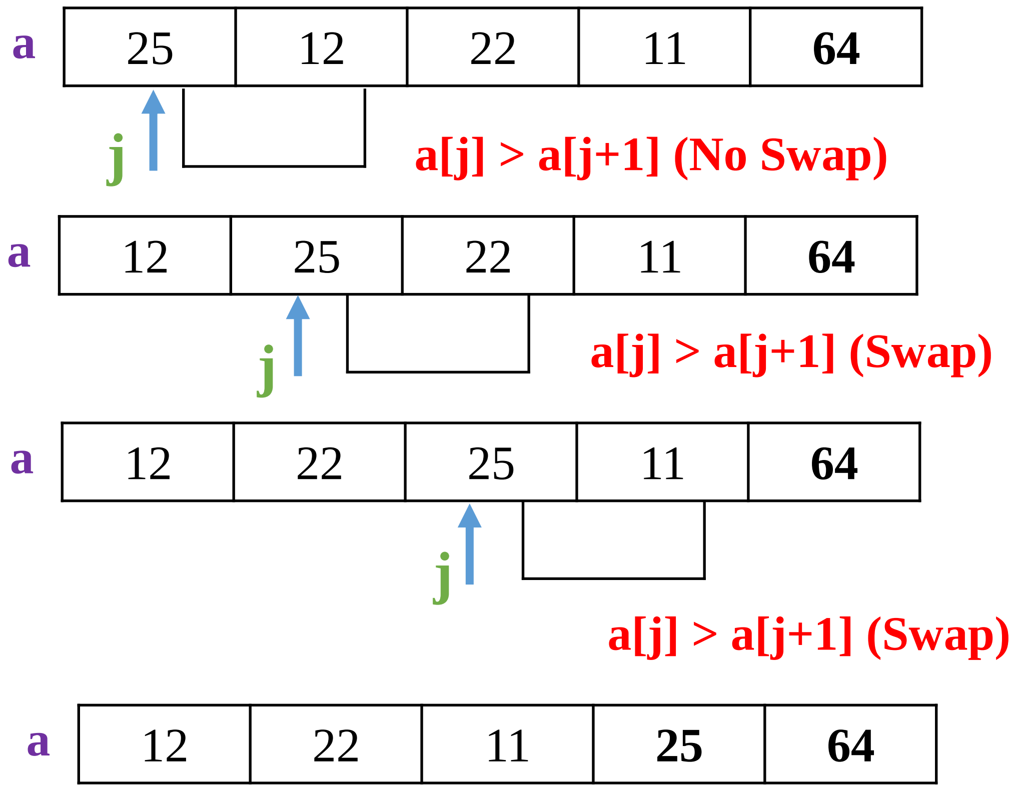
```
for all elements of list
   if list[i] > list[i+1]
      swap(list[i], list[i+1])
   end if
end for
return list
```

Example: Sorting of an array in ascending order using bubble sort

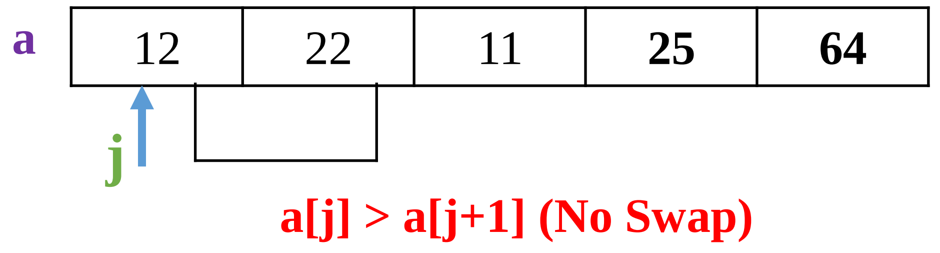
For i = 1

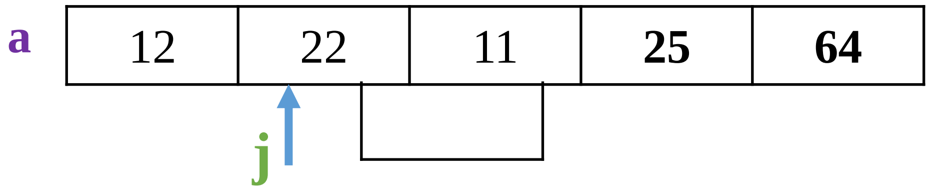


For i = 2

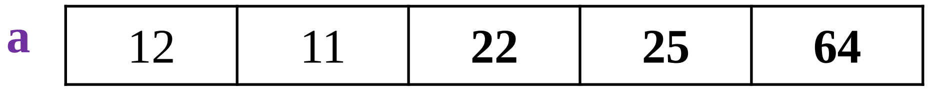


For i = 3

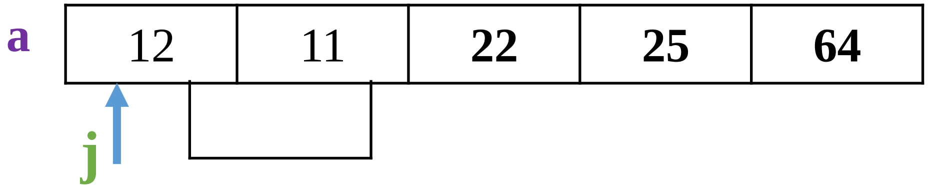




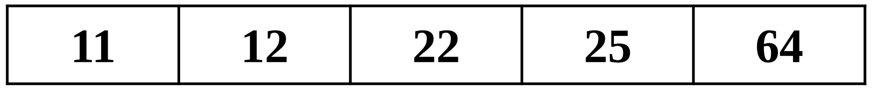
a[j] > a[j+1] (Swap)



For i = 4



a[j] > a[j+1] (Swap)



```
#include<iostream>
using namespace std;
int main()
{
    int arr[5]={5,14,24,63,7};
    int i,j,temp;
    for(i=0;i<4;i++)
    {
        for(j=0;j<=5-2-i;j++)
        {
            if(arr[j]>arr[j+1])
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
    for (i=0;i<5;i++)
    {
        cout<<arr[i]<<'\t';
    }
    return(0);
}
```

Output:

```
5	7	14	24	63
```

iii. Insertion Sort

- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Algorithm:

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step 2 - Pick the next element, and store it separately in a key.

Step 3 - Now, compare the key with all elements in the sorted array.

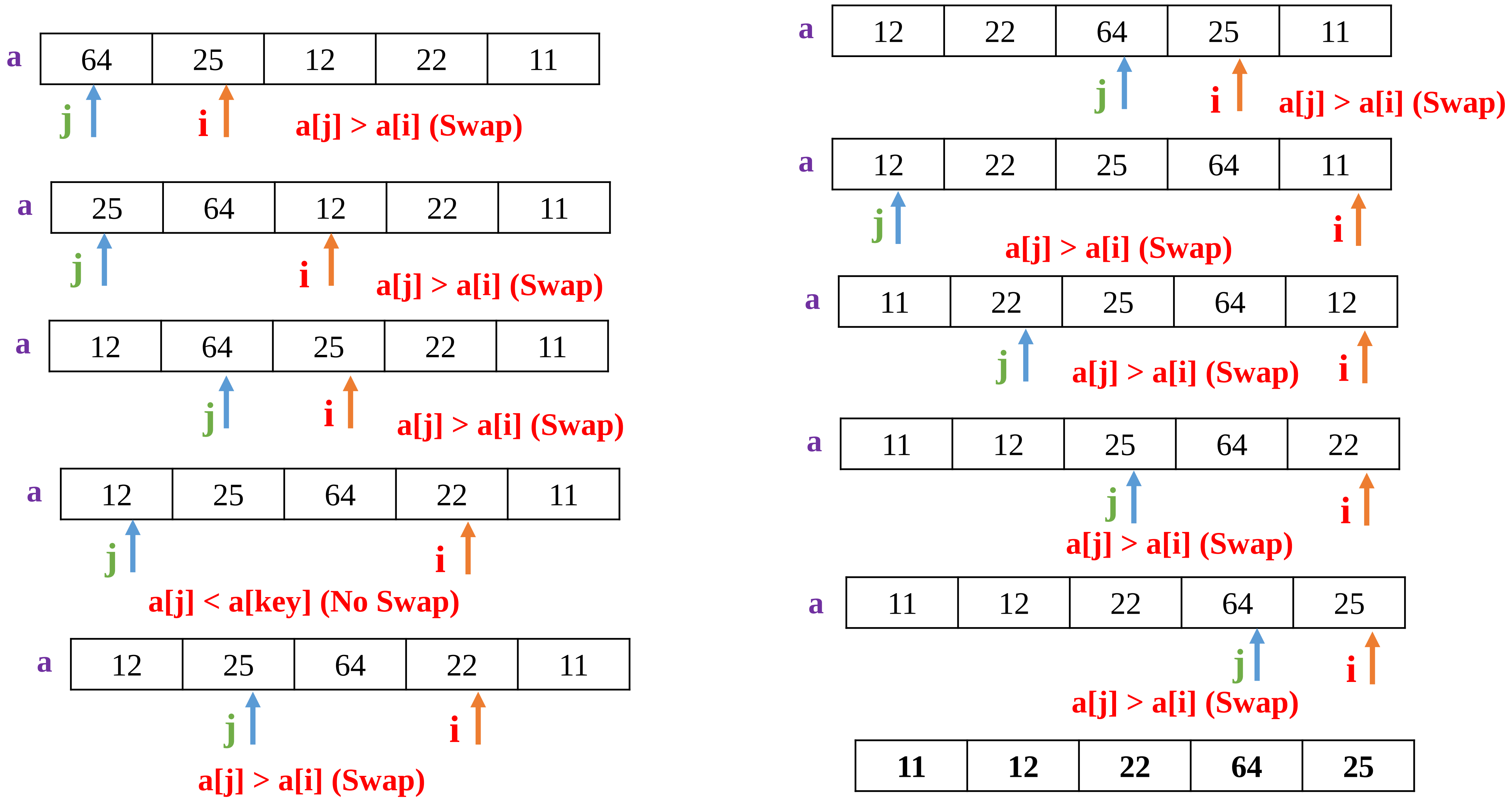
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element.

Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Example: Sorting of an array in ascending order using insertion sort


a | 64 | 25 | 12 | 22 | 11
j
i
a[j] > a[i] (Swap)
a | 25 | 64 | 12 | 22 | 11
j
i
a[j] > a[i] (Swap)
a | 12 | 64 | 25 | 22 | 11
j
i
a[j] > a[i] (Swap)
a | 12 | 25 | 64 | 22 | 11
j
i
a[j] < a[key] (No Swap)
a | 12 | 25 | 64 | 22 | 11
j
i
a[j] > a[i] (Swap)
a | 12 | 22 | 64 | 25 | 11
j
i
a[j] > a[i] (Swap)
a | 12 | 22 | 25 | 64 | 11
j
i
a[j] > a[i] (Swap)
a | 11 | 22 | 25 | 64 | 12
j
a[j] > a[i] (Swap)
i
a | 11 | 12 | 25 | 64 | 22
j
i
a[j] > a[i] (Swap)
a | 11 | 12 | 22 | 64 | 25
j
i
a[j] > a[i] (Swap)
11 | 12 | 22 | 64 | 25

```
#include<iostream>
using namespace std;
int main()
{
    int arr[5]={5,14,24,63,7};
    int i,j,temp;
    for(i=1;i<5;i++)
    {
        for(j=0;j<=i;j++)
        {
            if(arr[j]>arr[i])
            {
                temp=arr[j];
                arr[j]=arr[i];
                arr[i]=temp;
            }
        }
    }
    for (i=0;i<5;i++)
    {
        cout<<arr[i]<<'\t';
    }
    return(0);
}
```

Output:

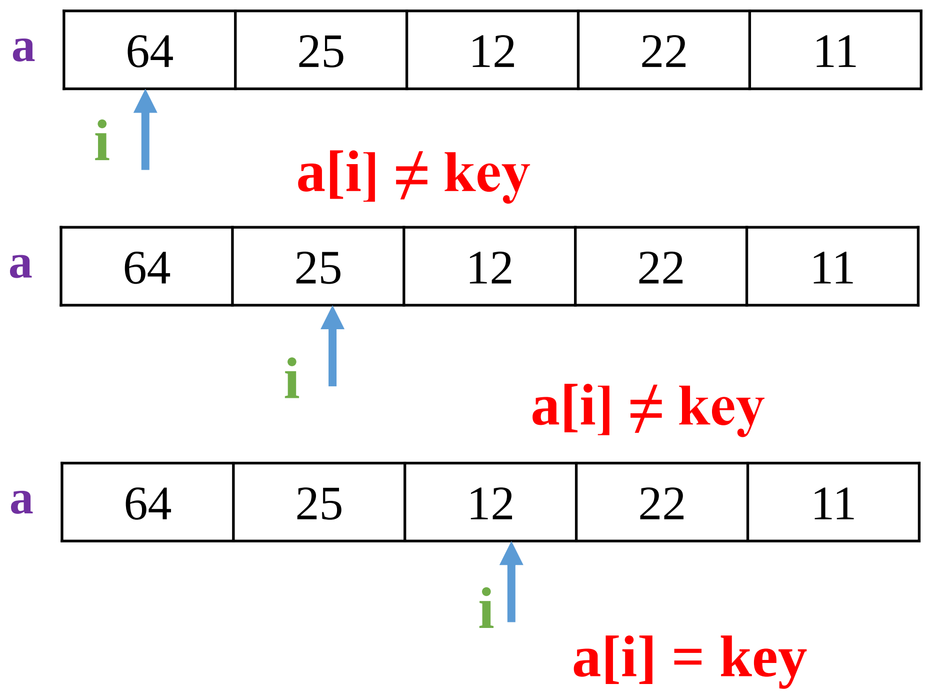
```
5	7	14	24	63
```

3. Searching

- Array searching can be defined as the operation of finding a particular element or a group of elements in the array.
- There are several searching algorithms. The most commonly used among them are:
    i. Linear Search
    ii. Binary Search

i. Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.

Linear Search algorithm to search 12 (key=12)



Now, the element to be searched is found. So algorithm will return the index of the element matched.

```
#include<iostream>
using namespace std;
int main()
{
    int arr[5]={5,14,24,63,7};
    int i,j,key; key=24;j=0;
    for(i=0;i<5;i++)
    {
        if(arr[i]==key)
        {
            cout<<"The index is "<<i;
            j=1;
        }
    }
    if (j==0)
    {
        cout<<"The index is not found";
    }
    return(0);
}
```

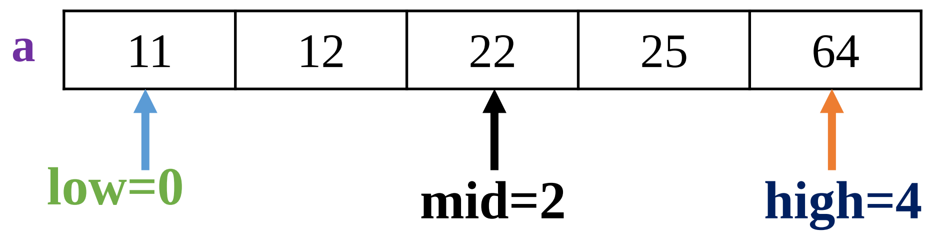
Output: *The index is 2*

ii. Binary Search Binary Search is defined as a searching algorithm used in a **sorted array** by repeatedly dividing the search interval in half.

Algorithm:

- Initialize ‘low’ as minimum index and ‘high’ as maximum index number (n-1).
- Divide the search space into two halves by finding the middle index “mid”.
- Compare the middle element of the search space with the “key”.
- If the ‘key’ is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
    - If the key is smaller than the middle element, then the left side is used for next search i.e. high=mid-1.
    - If the key is larger than the middle element, then the right side is used for next search i.e. low=mid+1.
- This process is continued until the key is found or the total search space is exhausted.

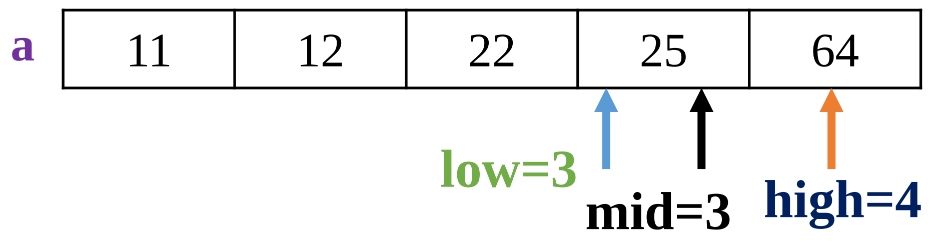
Binary Search algorithm to search 25 (key=25)



key > a[mid]

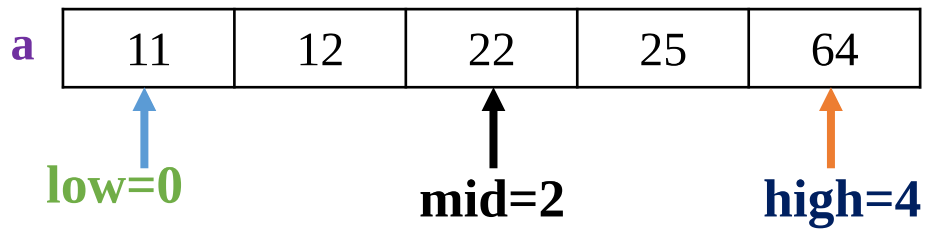
So low = mid+1;

mid = (low + high)/2



key = a[mid] = 3

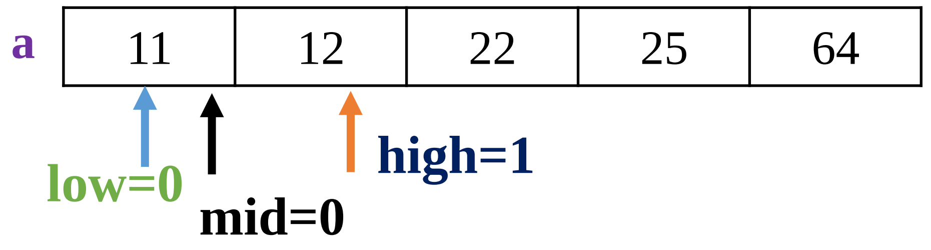
Binary Search algorithm to search 11 (key = 11)



key < a[mid]

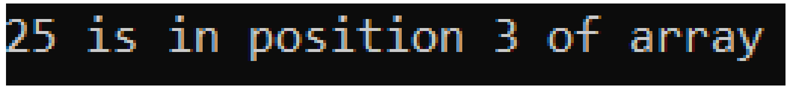
So high = mid-1;

mid=(low + high)/2



key = a[mid] = 0

```
#include<iostream>
using namespace std;
int main()
{
    int arr[5]={11,12,22,25,64};
    int i,key,low,high,mid; key=25;i=-1;
    low=0; high=4;
    mid=(low+high)/2;
    while(low!=high)
    {
        if(key<arr[mid])
        {
            high=mid-1;
        }
        if(key>arr[mid])
        {
            low=mid+1;
        }
        mid=(low+high)/2;
if(arr[mid]==key)
        {
            i=mid;
            cout<<"Position of key is:"<<i;
            break;
        }
    }
    if(i==-1)
    {
        cout<<"key is not in the array";
    }
    return(0);
}
```



```
25 is in position 3 of array
```

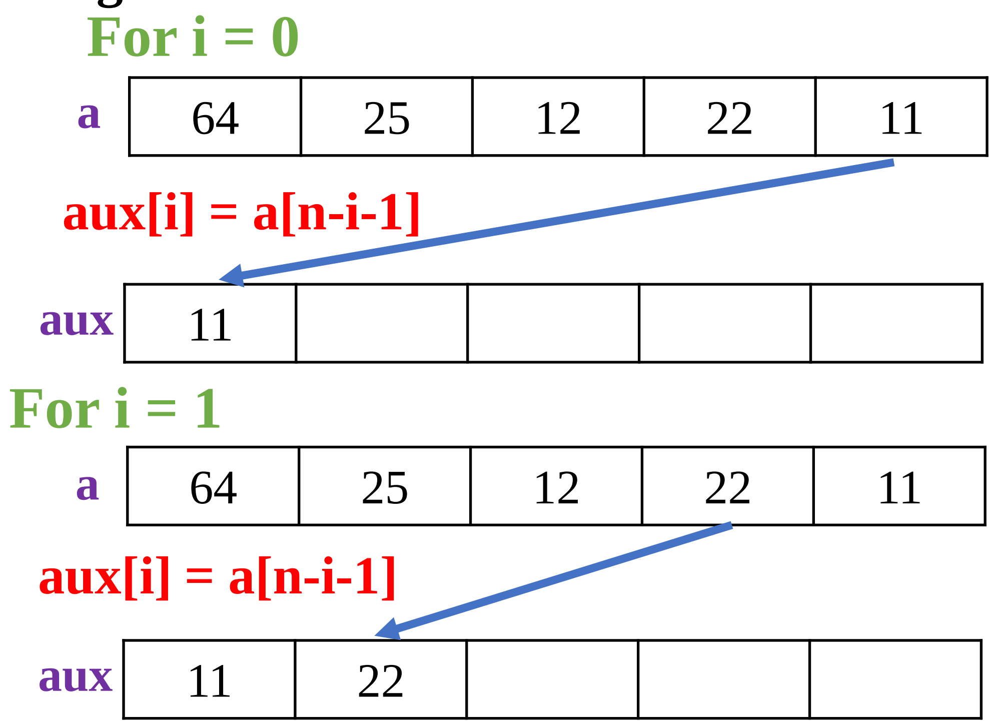
5. Reversing

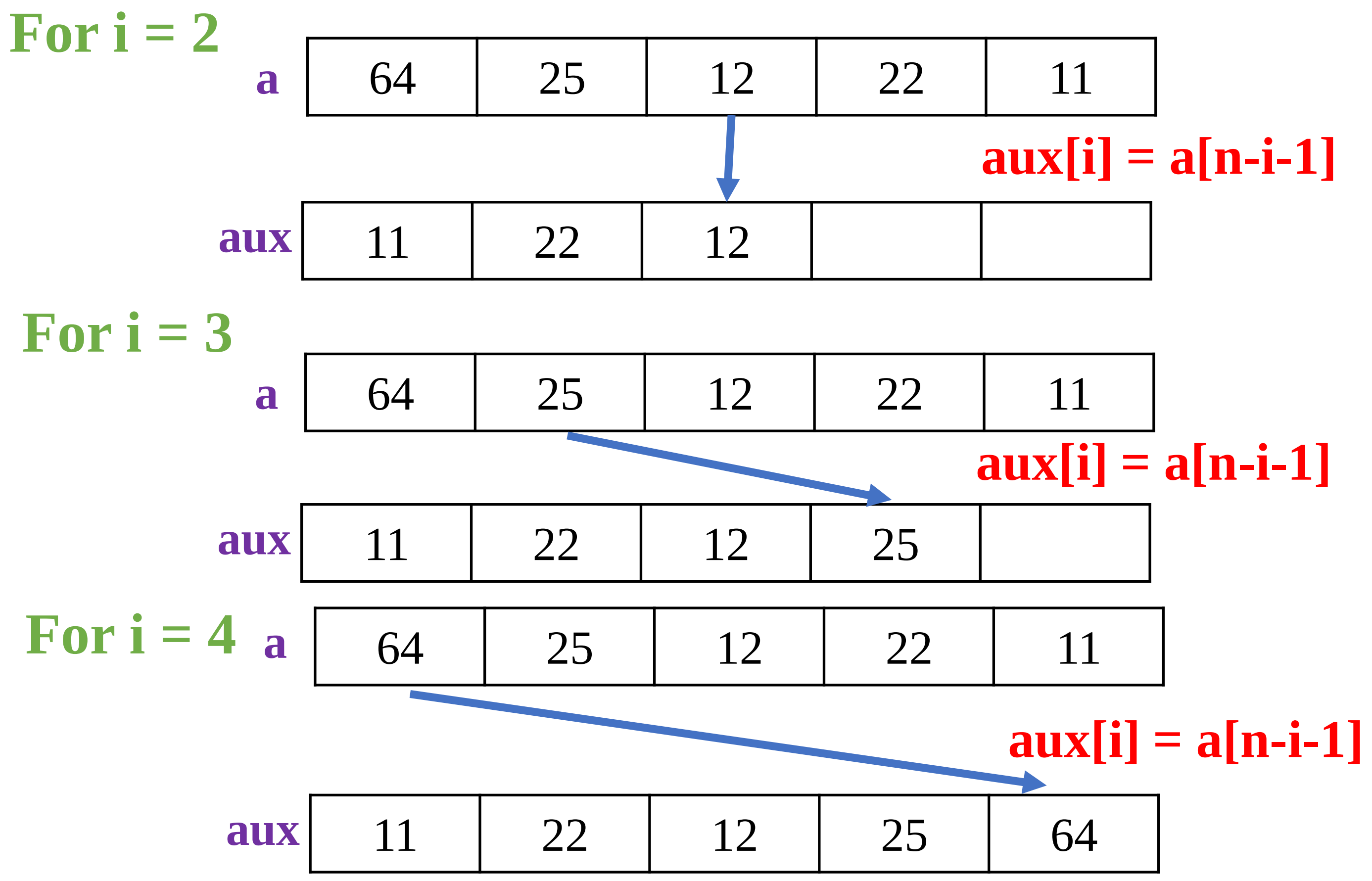
- Reversing an array is one of the easiest problems, while there are several methods to reverse an array in C, two of them being the most efficient solutions; Using an **auxiliary array** and **In-place Swapping**.

Declaring auxiliary array

- We can declare another array, iterate the original array from backward and send them into the new array from the beginning.

Algorithm:





Write a program to reverse an array

```
#include<iostream>
using namespace std;
int main()
{
    int arr[5]={64,25,12,22,11};
    int i; int aux[5]; int n=5;
    for (i=0;i<5;i++)
    {
        aux[i]=arr[n-1-i];
    }
    for (i=0;i<5;i++)
    {
        cout<<aux[i]<<'\t';
    }
    return(0);
}
```

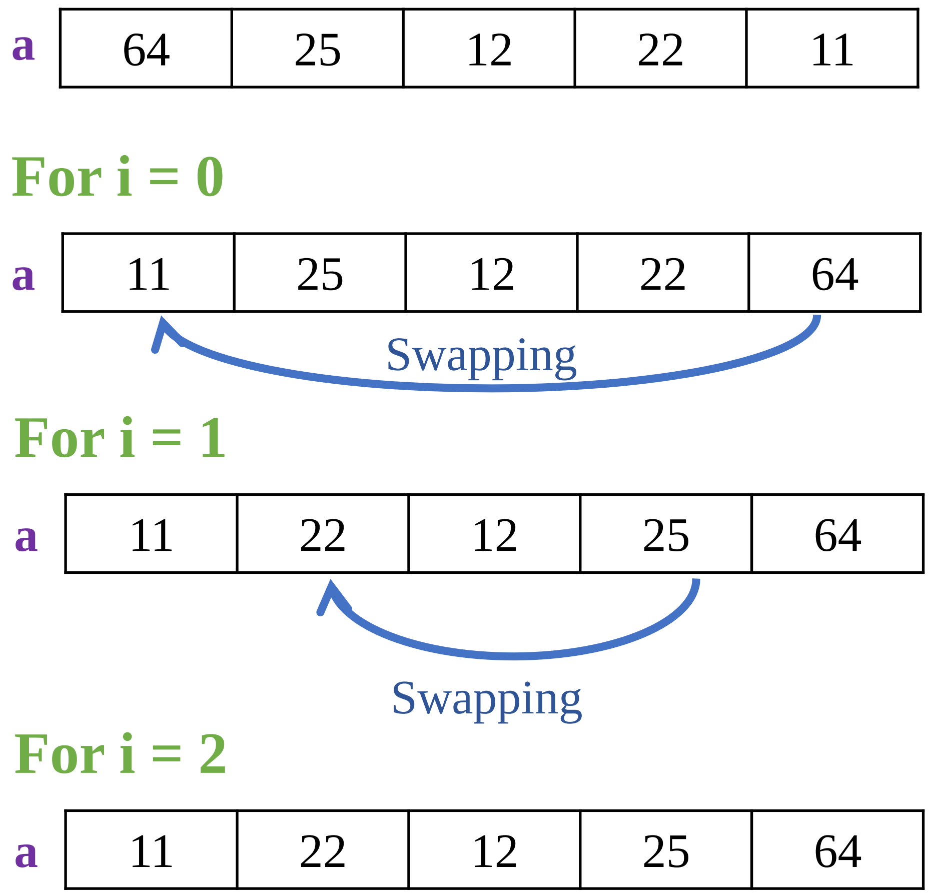
Output:

```
11      22      12      25      64
```

In-place Swapping

- We iterate the array till size/2, and for an element in index i, we swap it with the element at index (size - i - 1).
- We can keep iterating and swapping elements from both sides of the array till the middle element.

Algorithm: n is the number of elements of array



```
#include<iostream>
using namespace std;
int main()
{
    int a[5]={64,25,12,22,11};
    int i, temp; int n=5;
    for (i=0;i<n/2;i++)
    {
        temp=a[i];
        a[i]=a[n-1-i];
        a[n-1-i]=temp;
    }
    for (i=0;i<5;i++)
    {
        cout<<a[i]<<'\t';
    }
    return(0);
}
```

Output:

```
11      22      12      25      64
```

6. Insertion

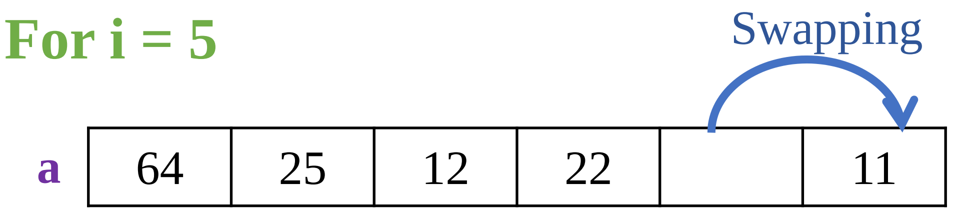
- ➢ We can insert the elements wherever we want, which means we can insert either at starting position or at the middle or at last or anywhere in the array.
- ➢ After inserting the element in the array, the positions or index location is increased but it does not mean the size of the array is increasing.

The logic used to insert element is −

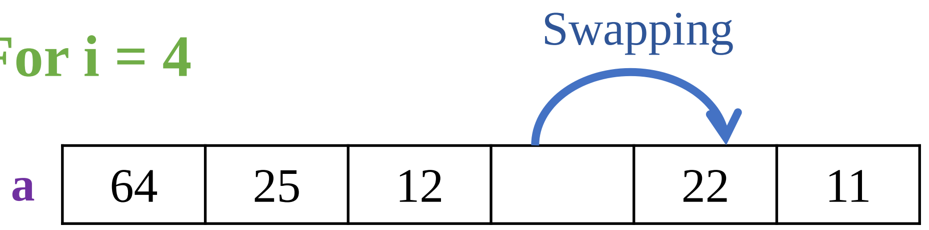
- ✓ Enter the size of the array
- ✓ Enter the position where you want to insert the element
- ✓ Next enter the number that you want to insert in that position

Insertion algorithm to insert 30 at position 2

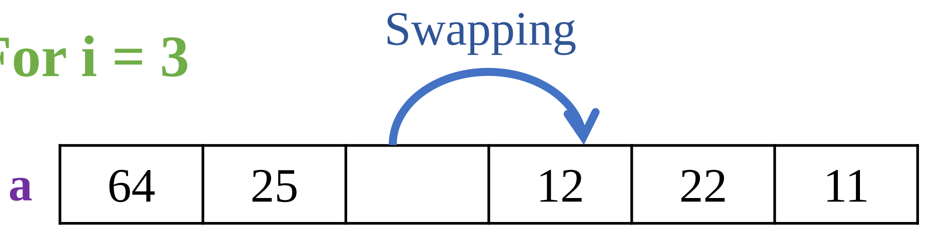
For i = 5



For i = 4

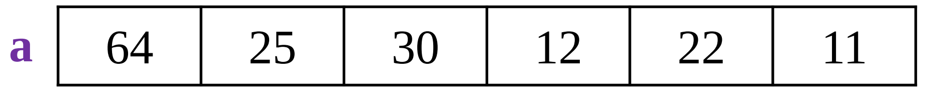


For i = 3

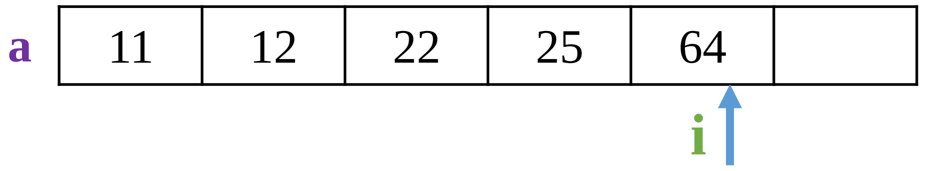


For i = 2

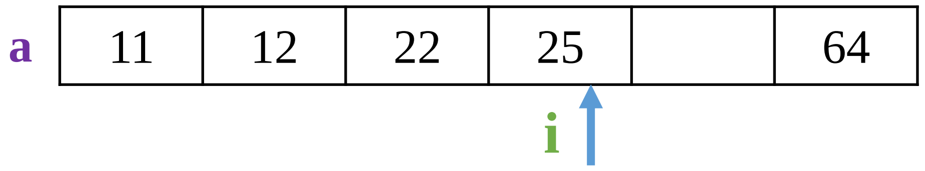
pos = i; So a[pos] = 30

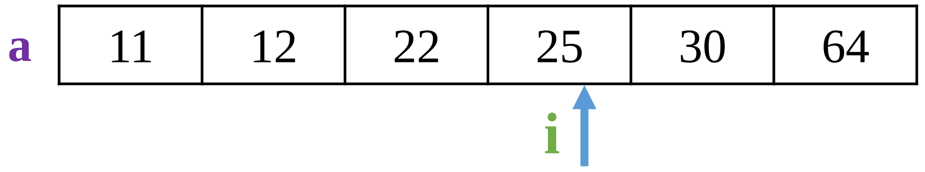


Insertion algorithm to insert 30 (key) for shorted array



a[i] <= key, then a[i+1]=a[i]





a[i] > key, then a[i+1]=key;
break;

Insertion algorithm to insert 30 at position 2

```
#include<iostream>
using namespace std;
int main()
{
    int a[5]={64,25,12,22,11};
    int i, key; int n=5;
    key=30;
    for (i=5;i>2;i--)
    {
        a[i]=a[i-1];
    }
    a[2]=key;
    for (i=0;i<=5;i++)
    {
        cout<<a[i]<<'\t';
    }
    return(0);
}
```

Output:

```
64   25     30     12     22     11
```

Insertion algorithm to insert 18 (key) for shorted array

```
#include<iostream>
using namespace std;
int main()
{
    int a[5]={11,12,22,25,64};
    int i; int key=18;
    for (i=4;i>=0;i--)
    {
        if(a[i]>key)
        {
            a[i+1]=a[i];
        }
        else
        {
            a[i+1]=key;
            break;
        }
    }
    for (i=0;i<=5;i++)
    {
        cout<<a[i]<<'\t';
    }
    return(0);
}
```

7. Deletion

- To delete a specific element from an array, a user must define the position from which the array's element should be removed.

Steps

Step 1: Input the size of the array a[] using num, and then declare the pos variable to define the position, and i represent the counter value.

Step 2: Use a loop to insert the elements in an array until (i < num) is satisfied.

Step 3: Now, input the position of the particular element that the user or programmer wants to delete from an array.

Step 4: Compare the position of an element (pos) from the total no. of elements (num+1). If the pos is greater than the num+1, the deletion of the element is not possible and jump to step 7.

Step 5: Else removes the particular element and shift the rest elements' position to the left side in an array.

Step 6: Display the resultant array after deletion or removal of the element from an array.

```
#include<iostream>
using namespace std;
int main()
{
   int a[5]={64,25,12,22,11};
   int pos=2;
   for (int i=pos;i<5;i++)
   {
      a[i]=a[i+1];
   }
   for (int i=0;i<4;i++)
   {
      cout<<a[i]<<'\t';
   }
   return(0);
}
```

Output:

```
64     25     22     11
```

Another method

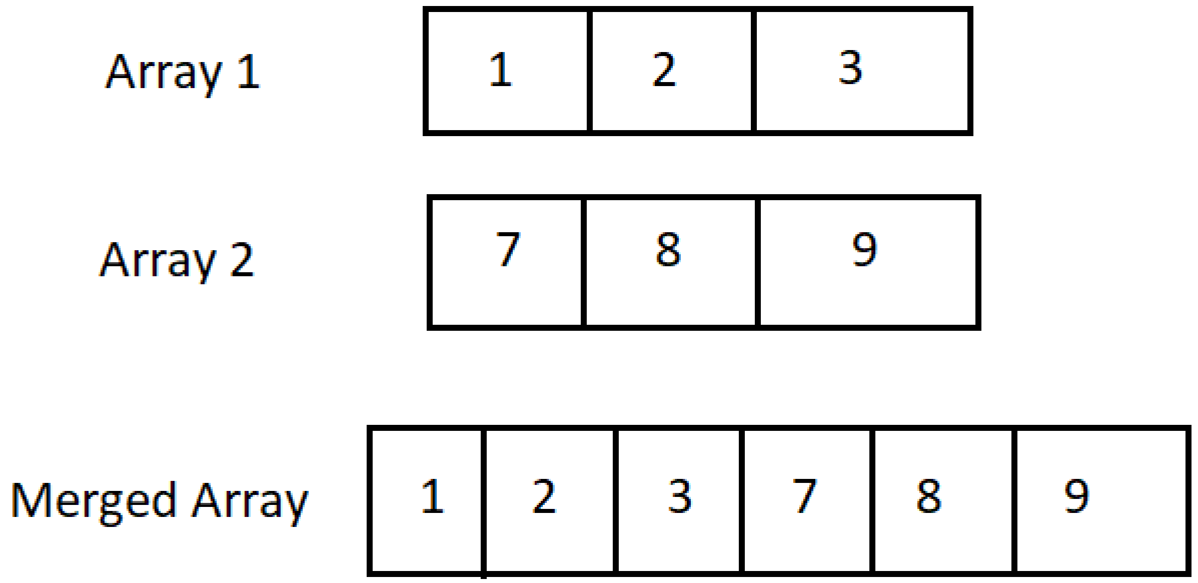
```
#include<iostream>
using namespace std;
int main()
{
    int a[5]={64,25,12,22,11}; int b[4];
    int i,j; j=0;
    int pos=2;
    for (i=0;i<5;i++)
    {
        if (i==pos)
        {
            j=i;
            continue;
        }
        b[j]=a[i];
        j++;
    }
    for (int i=0;i<4;i++)
    {
        cout<<b[i]<<'\t';
    }
    return(0);
}
```

Output:

```
64      25      22      11
```

8. Merging

- Combining two arrays into a single array is known as merging two arrays. For example, if the first array has 5 elements and the second array has 4, the resultant array will have 9 elements.
- To merge any two arrays in C programming, start adding each and every element of the first array to the third array (the target array). Then start appending each and every element of the second array to the third array (the target array), as shown in the program given below.



```
#include<iostream>
using namespace std;
int main()
{
    int a[5]={64,25,12,22,11};
    int b[4]={6,8,3,12};
    int i,j; j=0;
    int n=5; int k=4;
    int c[n+k];
    for (i=0;i<n+k;i++)
    {
        if (i<n)
        {
            c[i]=a[i];
        }
        if (i>=n)
        {
            c[i]=b[j];
            j++;
        }
    }
    for (int i=0;i<n+k;i++)
    {
        cout<<c[i]<<'\t';
    }
    return(0);
}
```

Output:

```
64      25      12      22      11      6       8       3       12
```

Arrays as Function Arguments

- We need to pass the array to a function to make it accessible within the function. If we pass an entire array to a function, all the elements of the array can be accessed within the function.
- Single array elements can also be passed as arguments. This can be done in exactly the same way as we pass variables to a function.

Syntax for passing single element of an array:

return_type function_name(type array_name[index]);

Example:

```
#include<iostream>
using namespace std;
int disp(int x)
{
    cout<<x<<'\t';
}
int main()
{
    int a[5]={5,7,15,54,23};
    for(int i=0;i<5;i++)
    {
        disp(a[i]);
    }
    return(0);
}
```

Output:
```
5       7       15      54      23
```

Syntax for passing an array:

return_type function_name(type array_name[]);

or

return_type function_name(type array_name[]) ;

```
#include<iostream>

using namespace std;

int sum(int marks[]);

int main()

{

    int marks[5]={55,71,15,54,23};

    int result;

    result=sum(marks);

    cout<<"Result is "<<result;

    return(0);

}
```

```
int sum(int marks[])

{

    int sum=0;

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

    {

        sum+=marks[i];

    }

    return sum;

}
```

Output: *Result is 218*

```
#include<iostream>
using namespace std;
int avg(int marks[], int size);
int main()
{
    int marks[]={55,71,15,54,23};
    int result; int size=sizeof(marks)/sizeof(int);
    int average=avg(marks,size);
    cout<<"Result is "<<average;
    return(0);
}
```

```
int avg(int marks[], int size)
{
    int sum=0; int aver;
    for(int i=0;i<size;++i)
    {
        sum+=marks[i];
    }
    aver=sum/size;
    return aver;
}
```

Output: Result is 43

Two Dimensional Array

- So far we have explored arrays with only one dimension. It is also possible for arrays to have two or more dimensions. The two dimensional array is also called a matrix.

Syntax:

Declaration:

data_type array_name[rows][columns];

Example: int a[2][3]; //This is a matrix of size (2×3).

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initialization:

data_type array_name[rows][columns]={{row_1 elements}, {row_2 elements}, ….};

or

*data_type array_name[rows][columns]={1st Element, 2nd Element,…, rows*columns Element};*

Example: int a[2][3]={{1,4,7},{2,5,9}}; or int a[2][3]={1, 4, 7, 2, 5, 9};

$$//a = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 9 \end{bmatrix}$$

```
#include<iostream>
using namespace std;
int main()
{
    int a[2][3]={{1,4,6},{2,6,8}};
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            cout<<a[i][j]<<'\t';
        }
        cout<<endl;
    }
    return(0);
}
```

Output:

```
1       4       6
2       6       8
```

```
#include<iostream>
using namespace std;
int main()
{
    int a[2][3]={1,4,6,2,6,8};
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            cout<<a[i][j]<<'\t';
        }
        cout<<endl;
    }
    return(0);
}
```

Output:

```
1       4       6
2       6       8
```

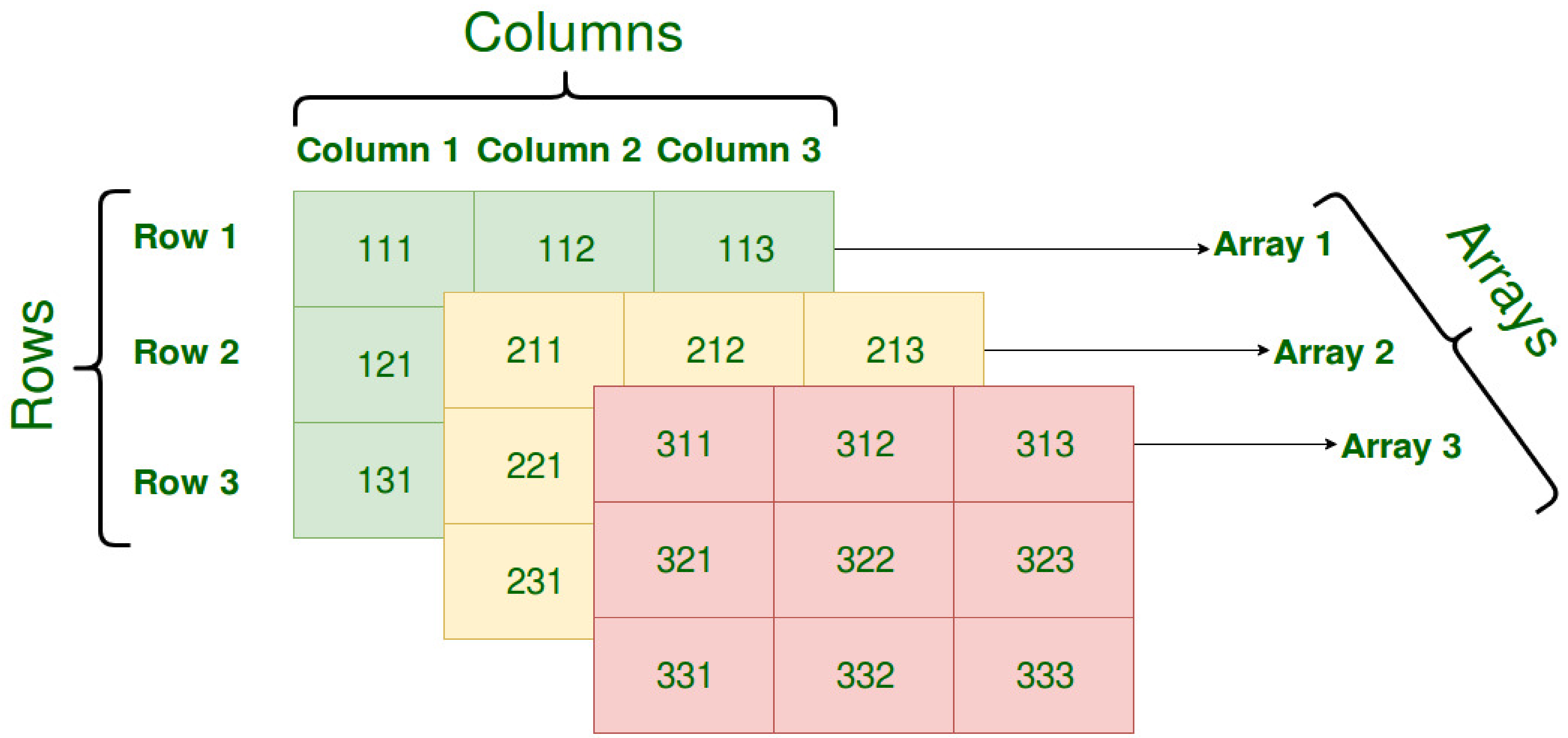
```
#include<iostream>

using namespace std;

int main()

{

    int a[2][3];

    int i,j;

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

    {

        for(j=0;j<3;j++)

        {

            cout<<"Enter a["<<i+1<<"]["<<j+1<<"] ";

            cin>>a[i][j];

        }

    }
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            cout<<a[i][j]<<'\t';
        }
        cout<<endl;
    }
    return(0);
}
```

Output:

```
Enter a[1][1] 1
Enter a[1][2] 4
Enter a[1][3] 6
Enter a[2][1] 2
Enter a[2][2] 6
Enter a[2][3] 8
1       4       6
2       6       8
```

Multi Dimensional Array

- A Three Dimensional Array or 3D array in C++ is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.



Syntax:

Declaration:

data_type array_name[No of 2-D arrays][rows][columns];

Example: int a[3][2][3]; //This will be a three matrices of size (2×3).

Initialization:

data_type array_name [No of 2-D arrays][rows][columns]={{Elements of 1st matrix}, {Elements of 2nd matrix}, …., {}};

or

*data_type array_name [No of 2-D arrays][rows][columns]={1st Element, 2nd Element,…, No of 2-D arrays *rows*columns Element};*

Example: int a[3][2][3]={{{1,4,7},{2,5,9}}, {{7,5,2},{3,5,5}}, {{1,3,4},{4,6,8}}};

or

int a[3][2][3]={1, 4, 7, 2, 5, 9,7,5,2,3,5,5,1,3,4,4,6,8};

```
#include<iostream>
using namespace std;
int main()
{
    int a[3][2][3]={{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}},{{13,14,15},{16,17,18}}};
    int i,j,k;
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<3;k++)
            {
                cout<<a[i][j][k]<<'\t';
            }
            cout<<endl;
        }
        cout<<endl<<endl;
    }
    return(0);
}
```

Output:

```
1       2       3
4       5       6


7       8       9
10      11      12


13      14      15
16      17      18
```

```
#include<iostream>
using namespace std;
int main()
{
    int a[3][2][3]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
    int i,j,k;
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<3;k++)
            {
                cout<<a[i][j][k]<<'\t';
            }
            cout<<endl;
        }
        cout<<endl<<endl;
    }
    return(0);
}
```

Output:

```
1       2       3
4       5       6


7       8       9
10      11      12


13      14      15
16      17      18
```

```
#include<iostream>
using namespace std;
int main()
{
    int a[3][2][3]; int i,j,k;
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<3;k++)
            {
                cout<<"Enter a["<<i+1<<"]["<<j+1<<"]["<<k+1<<"] ";
                cin>>a[i][j][k];
            }
        }
    }
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<3;k++)
            {
                cout<<a[i][j][k]<<'\t';
            }
            cout<<endl;
        }
        cout<<endl<<endl;
    }
    return(0);
}
```

Output:

```
Enter a[1][1][1] 1
Enter a[1][1][2] 2
Enter a[1][1][3] 3
Enter a[1][2][1] 4
Enter a[1][2][2] 5
Enter a[1][2][3] 6
Enter a[2][1][1] 7
Enter a[2][1][2] 8
Enter a[2][1][3] 9
Enter a[2][2][1] 10
Enter a[2][2][2] 11
Enter a[2][2][3] 12
Enter a[3][1][1] 13
Enter a[3][1][2] 14
Enter a[3][1][3] 15
Enter a[3][2][1] 16
Enter a[3][2][2] 17
Enter a[3][2][3] 18
1       2       3
4       5       6

7       8       9
10      11      12

13      14      15
16      17      18
```

Pointer

- A pointer is a variable that holds the address of a variable or a function. A pointer is a powerful feature that adds enormous power and flexibility to C++ language.
- If you have a variable *var* in your program, *&var* will give you its address in the memory.
- As the pointers in C++ store the memory addresses, their size is independent of the type of data they are pointing to.

Pointer Declaration:

- To declare a pointer, we use the (*) dereference operator before its name.

Syntax: *data_type *pointer_variable_name;*

Example

*int *ptr; //ptr is the pointer variable and *ptr holds the values stored in the variable.*

- The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called *wild pointers*.

Pointer Initialization:

- Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the (&) addressof operator to get the memory address of a variable and then store it in the pointer variable.

Syntax: *data_type variable_name = value;*

*data_type *pointer_variable_name;*

pointer_variable_name=&variable_name

or

*data_type *pointer_variable_name=&variable_name;*

Example

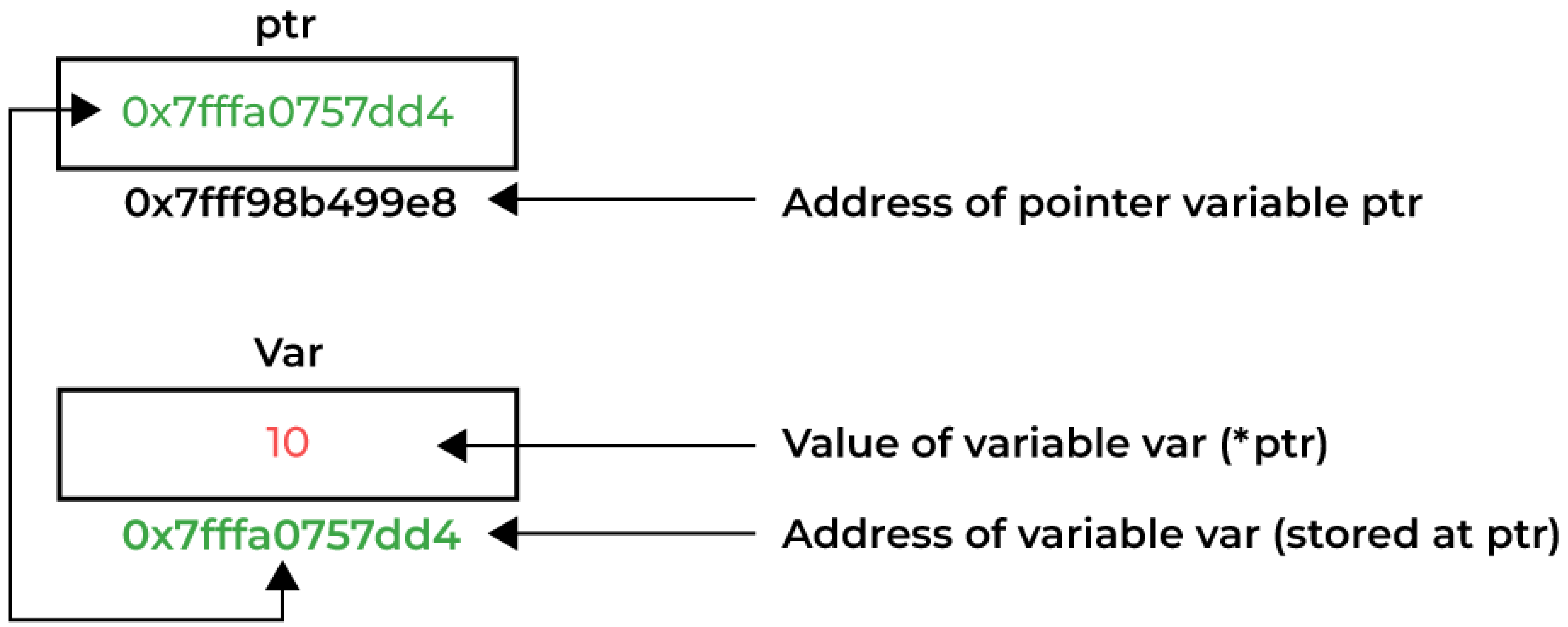
```
int var = 10;
int * ptr;
ptr = &var;
```

or

```
int var = 10;
int *ptr = &var;
```

Pointer Dereferencing

- Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer.
- We use the same **(*) dereferencing operator** that we used in the pointer declaration.


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (stored at ptr)

```
#include<iostream>
using namespace std;
int main()
{
    int a=10;
    int *p;
    p=&a;
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<p<<endl;
    cout<<*p<<endl;
    return(0);
}
```

Output:

```
10
0x61ff08
0x61ff08
10
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
    int a=10;
    int *p;
    p=&a;
    // p=a;  //Error
    // *p=&a;//Error
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<p<<endl;
    cout<<*p<<endl;
    return(0);
}
```

Output:

```
10
0x61ff08
0x61ff08
10
```

Example

```
#include<iostream>
using namespace std;
int main()
{
   int a=10;
   int *p=&a;
   cout<<a<<endl;
   cout<<p<<endl;
   cout<<*p<<endl;
   *p=22;
   cout<<a<<endl;
   cout<<p<<endl;
   cout<<*p<<endl;
   return(0);
}
```

Output:

```
10
0x61ff08
10

22
0x61ff08
22
```

Advantages of pointers:

i. Pointers are more efficient in handling arrays and data tables.

ii. Pointers permit references to functions and there by facilitating passing of function as arguments to other functions.

iii. The use of pointer array to character strings results in saving of data storage space in memory.

iv. Pointers allows us to support dynamic memory management.

v. They increase the execution speed of the program.

Disadvantages

i. If an implicit value is provided to the pointer, memory corruption can occur.

ii. Pointers are slightly slower than normal variables.

iii. There is a possibility of memory leakage.

iv. Working with pointer may be a bit difficult for the programmer but it is the programmer’s responsibility to use the pointer properly.

v. Using a pointer can cause many problems, so it is necessary to use the pointer correctly.

Pointer Expression and Arithmetic:

- Pointer expression is the combination of pointer variables, operators and constants arranged as per the syntax of the language.

Examples:

**p1+*p2*

**p1+10*

10(*p1/ *p2)*

- There are some pointer expressions as follows:

    i. Arithmetic Operators

    ii. Relational Operators

    iii. Assignment Operators

    iv. Conditional Operators

    v. Unary Operators

    vi. Bitwise Operators

i. Arithmetic Operators

- We can perform arithmetic operations (+, -, *, / and %) to pointer variables using arithmetic operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   cout<<*p1+*p2<<endl;
   cout<<*p1-*p2<<endl;
   cout<<*p1**p2<<endl;
   cout<<*p1/ *p2<<endl;
   cout<<*p1%*p2<<endl;
   cout<<*p1+*p2+2<<endl;
   return(0);
}
```

Output:

```
35
5
300
1
5
37
```

- A space should be there between / and *. /* is used for comment line.

ii. Relational Operators

- We can perform relational operations (<, >, <=, <=, == and !=) to pointer variables using relational operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   if(*p1<*p2)
   {
      cout<<"b is greater than
a"<<endl;
   }
   if(*p1>*p2)
   {
      cout<<"a is greater than
b"<<endl;
   }
   if(*p1==*p2)
   {
      cout<<"a is equal to b"<<endl;
   }
   if(*p1!=*p2)
   {
      cout<<"a is not equal to b"<<endl;
   }
   return(0);
}
```

Output:

```
a is greater than b
a is not equal to b
```

iii. Assignment Operators

- We can perform assignment operations (=, +=, -=, *=, /= and %=) to pointer variables using assignment operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   cout<<(*p1=11)<<endl;
   cout<<(*p1+=*p2)<<endl;
   cout<<(*p1-=*p2)<<endl;
   cout<<(*p1*=*p2)<<endl;
   cout<<(*p1/=*p2)<<endl;
   cout<<(*p1%=*p2)<<endl;
   return(0);
}
```

Output:

```
11
26
11
165
11
11
```

iv. Conditional Operators

➢ We can perform conditional operation to pointer variables using conditional operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   (*p1>*p2)? cout<<"a is greater" :cout<<"b is greater";
   return(0);
}
```

Output: *a is greater*

v. Unary (Increment/Decrement) Operators

- We can perform Unary operations (++ and --) to pointer variables using unary operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   cout<<p1<<'\t'<<p2<<endl;
   cout<<*p1<<'\t'<<*p2<<endl;
   (*p1)++; p2++;
   cout<<p1<<'\t'<<p2<<endl;
   cout<<*p1<<'\t'<<*p2<<endl;
   return(0);
}
```

Output:

```
0x61ff04      0x61ff00
20      15
0x61ff04      0x61ff04
21      21
```

- ✓ p2++ increase the pointer address by 4 bytes which is equal to address of a or p1.
- ✓ *p2 is the value stored in the address of p1.

vi. Bitwise Operators

- We can perform bitwise operations (&, |, ^, <<, >> and ~) to pointer variables using bitwise operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=22; int b=18;
   int *p1=&a; int *p2=&b;
   cout<<(*p1&*p2)<<endl
;
   cout<<(*p1|*p2)<<endl;
   cout<<(*p1^*p2)<<endl;
   cout<<(*p1<<2)<<endl;
   cout<<(*p1>>2)<<endl;
   return(0);
}
```

Output:

```
18
22
4
88
5
```

Binary of

22=	10110	
18=	10010	
22&18=	10010	= 18
22|18=	10110	= 22
22^18=	00100	= 4
22<<2=	1011000	= 88
22>>2=	00101	= 5

Types of Pointer:

There are eight different types of pointers which are as follows –

1. Null pointer
2. Void pointer
3. Wild pointer
4. Dangling pointer
5. Smart pointer
6. Complex pointer
7. Near pointer
8. Far pointer
9. Huge pointer

1. Null pointer:

- In the C++ programming language, a null pointer is a pointer that does not point to any memory location and hence does not hold the address of any variables.
- That is, the null pointer in C++ holds the value Null, but the type of the pointer is void.
- An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant.
- Here, Null means that the pointer is referring to the 0th memory location.

Syntax-1: *type pointer_name = NULL;*

Example-1:

```
int *ptr=NULL;
float *ptr=NULL;
char *ptr=NULL;
```

Syntax-2: *type pointer_name = (data_type *)NULL;*

Example-2:

```
int *ptr=(int*)NULL;
float *ptr=(float*)NULL;
char *ptr=(char*)NULL;
```

Syntax-3: *type pointer_name = (data_type *)0;*

Example-3:

```
int *ptr=(int*)0;
float *ptr=(float*)0;
char *ptr=(char*)0;
```

Applications of Null Pointer:

- It is used to initialize 0 pointer variable when the pointer does not point to a valid memory address.
- It is used to perform error handling with pointers before dereferencing the pointers.
- It is passed as a function argument and to return from a function when we do not want to pass the actual memory address.

Example-1

```
#include<iostream>
using namespace std;
int main()
{
   int *p=NULL;
   cout<<p;
   return(0);
}
```

Output: 0

2. Void pointer:

- A void pointer is a pointer that has no associated data type with it. A void pointer can hold an address of any type and can be typecasted to any type.

Syntax-1: *void *pointer_name;* *// void keyword followed by name of the pointer*

Note:

i. Void pointers cannot be dereferenced.

Example:

```
#include<iostream>
using namespace std;
int main()
{
   int a=10;
   void *p=&a;
   cout<<p<<endl;
   cout<<*p;
   return(0);                    error: 'void*' is not a pointer-to-object type
}
```

ii. We typecast the void pointer to the type of data stored in pointer address by using the statement given below:

Syntax:

*(data_type *) pointer_name;* *//To access the address holds by the pointer*

**(data_type *) pointer_name;* *//To access the value stored at that address*

Example:

```
#include<iostream>
using namespace std;
int main()
{
   int a=10;float b=2.54;
   void *p, *q;
   p=&a; q=&b;
   cout<<"Adress stored in p pointer "<<(int*)p<<endl;
   cout<<"Value at p pointer "<<*(int*)p<<endl;
   cout<<"Adress stored in q pointer "<<(int*)q<<endl;
   cout<<"Value at q pointer "<<*(float*)q<<endl;
   return(0);
}
```

Output:

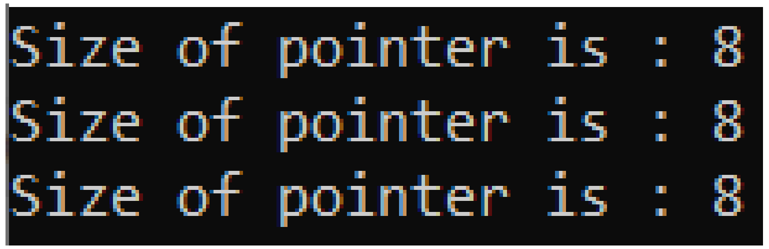
```
Adress stored in p pointer 0x61ff04
Value at p pointer 10
Adress stored in q pointer 0x61ff00
Value at q pointer 2.54
```

iii. The size of the Void Pointer in C++ depends on the type of Platform you are using. It does not depend upon the data type of the variable.

For 32-Bit Architecture: 4 Bytes

For 64-Bit Architecture: 8 Bytes

```
#include<iostream>
using namespace std;
int main()
{
   int a=10;float b=2.54; char c='k';
   void *p;
   p=&a;
   cout<<"Size of pointer is "<<sizeof(p)<<endl;
   p=&b;
   cout<<"Size of pointer is "<<sizeof(p)<<endl;
   p=&c;
   cout<<"Size of pointer is "<<sizeof(p)<<endl;
   return(0);
}
```



```
Size of pointer is : 8
Size of pointer is : 8
Size of pointer is : 8
```

iv. We cannot apply the arithmetic operations on void pointers in C++ directly. We need to apply the proper typecasting so that we can perform the arithmetic operations on the void pointers.

```
#include<iostream>
using namespace std;
int main()
{
   float a[]={1.2,2.6,2.1,5.8};
   void *p;
   p=&a;
   for(int i=0;i<4;i++)
   {
      cout<<*((float *)p+i)<<'\t';
   }
   return(0);
}
```

Output: 1.2   2.6   2.1   5.8

Advantages:

i. Void Pointer in C++ is used for implementing the generic function. These functions take a void pointer as an argument and typecast them as per the requirement.

ii. Generic Data Structures are implemented using the Void Pointer in C++ because they can be typecast according to the data type.

iii. Void Pointer in C++ is used for Dynamic Memory Allocation as functions such as calloc and malloc return a void pointer which can be typecast into different types of pointers.

3. Wild pointer:

- Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location while they are not assigned to any other memory location. This may even cause a program to crash or behave unpredictably at times.

Example: 1

```
#include<iostream>
using namespace std;
int main()
{
   int *p,*q,*r;
   cout<<p<<'\t'<<q<<'\t'<<r;
   return(0);
}
```

Output:

```
0       0       0
```

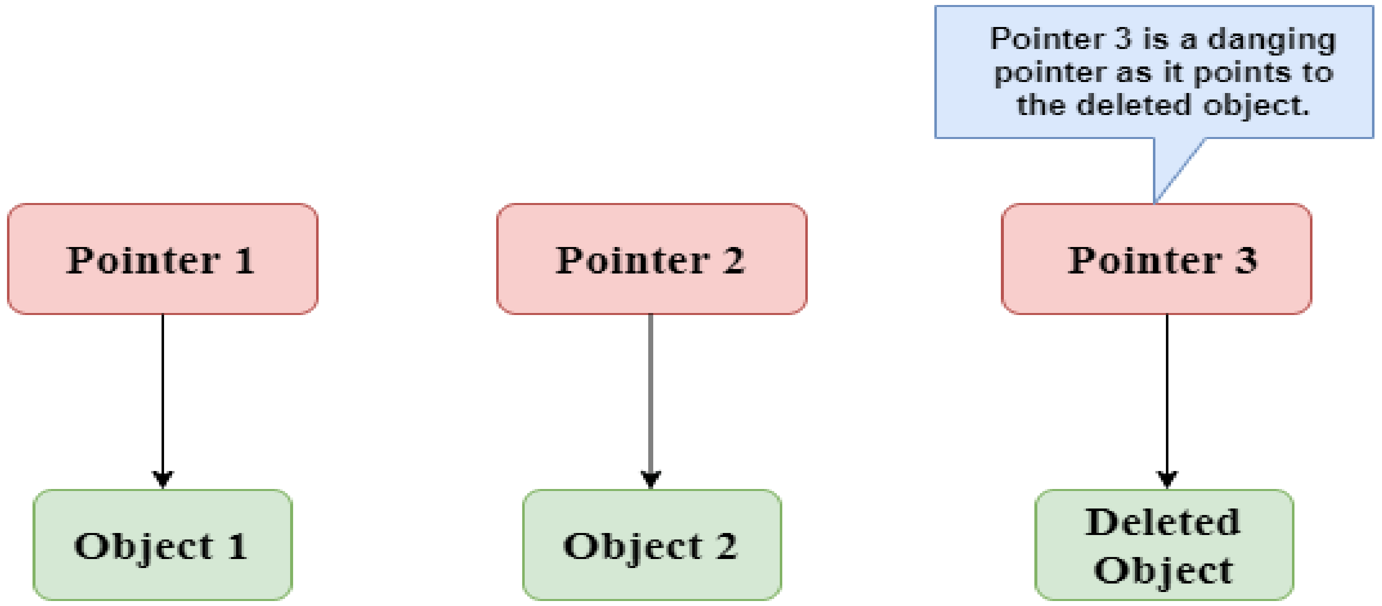
Example: 2 (p is not a wild pointer. q and r are the wild pointers)

```
#include<iostream>
using namespace std;
int main()
{
   int *p,*q,*r,a;
   p=&a;a=10;
   cout<<p<<'\t'<<q<<'\t'<<r;
   return(0);
}
```

Output: 0x7fff875e974c 0 0

4. Dangling pointer:

➢ A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

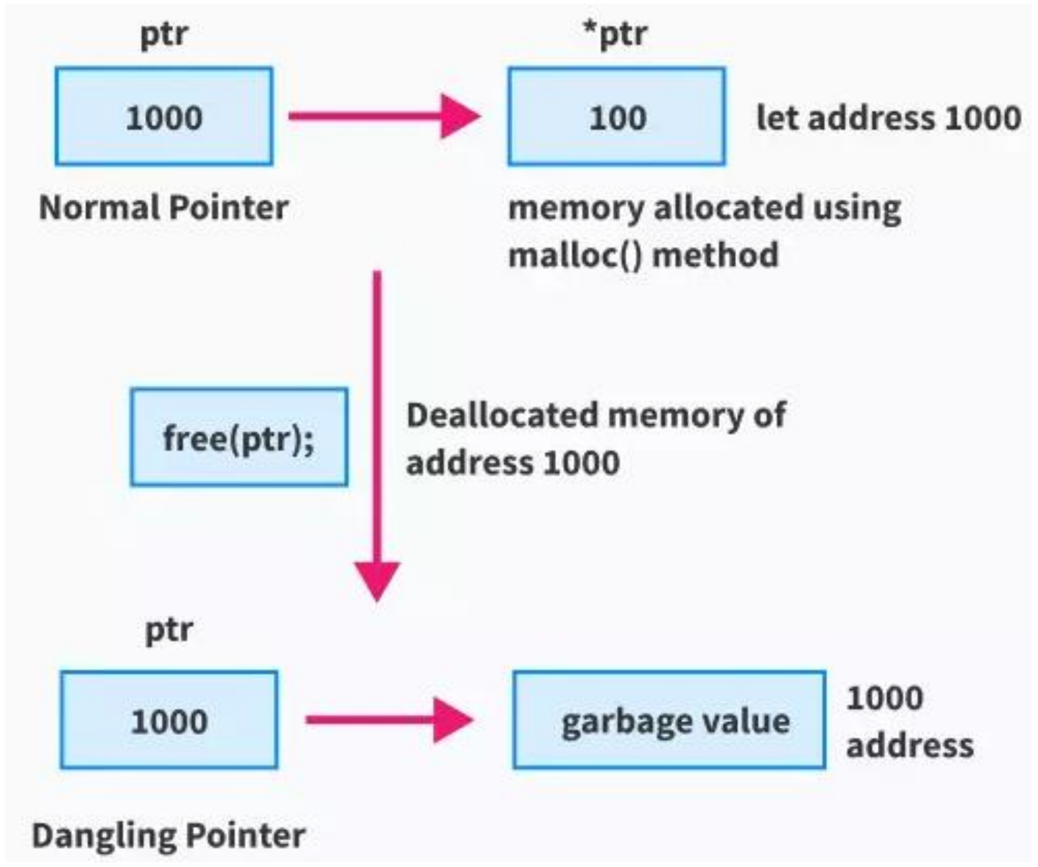

Pointer 3 is a danging pointer as it points to the deleted object.
Pointer 1
Pointer 2
Pointer 3
Object 1
Object 2
Deleted Object

There are three different ways in which a *pointer* can act as a dangling pointer in C++ :

1. Deallocation of memory
2. Function Call
3. Variable goes out of scope

1. Deallocation of memory:

- Allocation and deallocation of memory blocks are performed using library functions, like malloc(), calloc() functions are used for allocating a memory block, while free() function is used to deallocate a memory block.
- So, When we deallocate a memory block using free() function and do not modify the pointer value, it will cause the pointer to act as a Dangling Pointer.
- The free() function takes a single parameter, i.e. a pointer pointing to the memory to be deallocated.



Example:

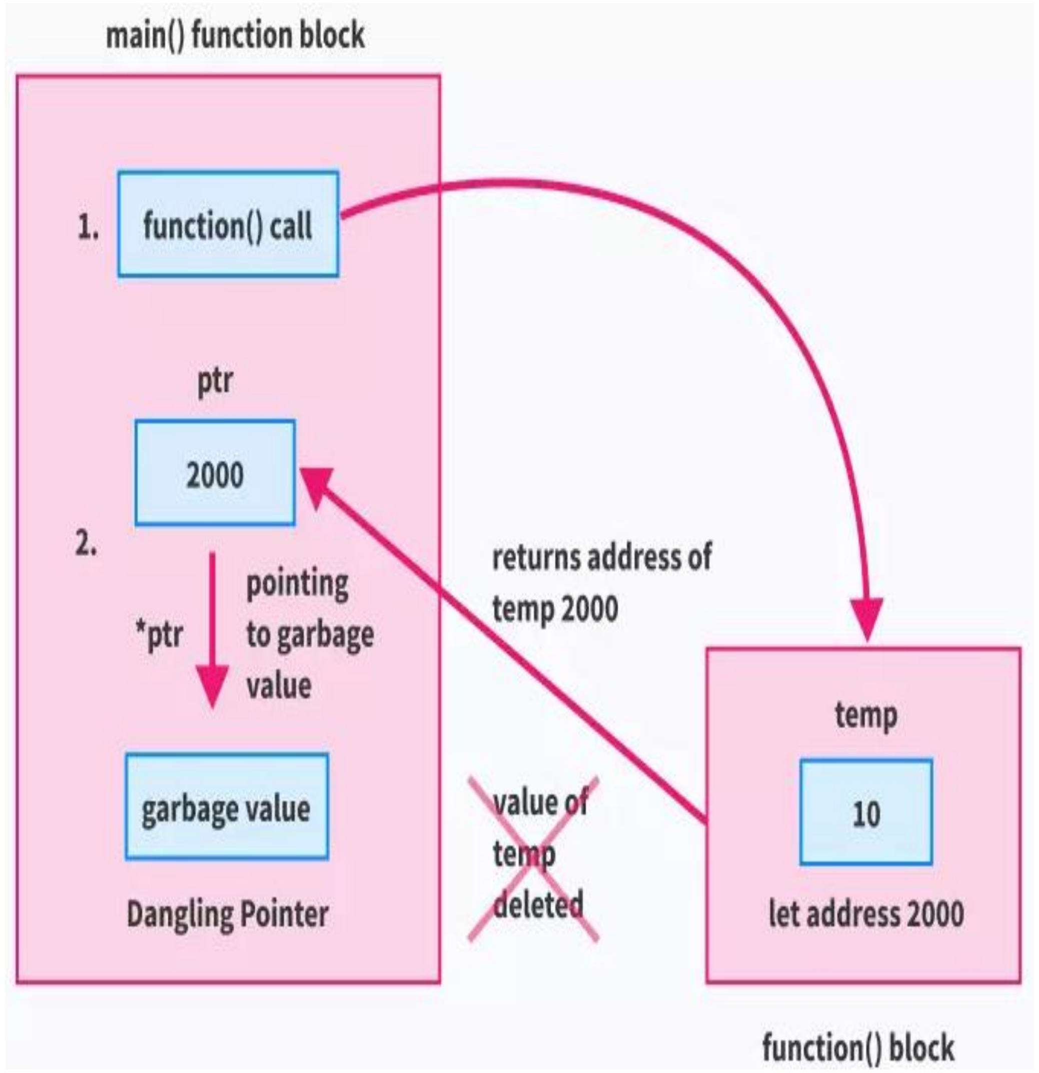
```
#include<iostream>
using namespace std;
int main()
{
    int *p=(int *)malloc(sizeof(int));
    *p=10;
    cout<<p<<'\t'<<*p<<endl;
    free(p);
    cout<<p<<'\t'<<*p<<endl;
    return(0);
}
```

Output:

```
0x2746d98      10
0x2746d98      41185144
```

2. Function Call:

- If we declare a variable inside a function, then that variable will be local to that function execution and cannot be accessed outside that function's scope.
- Now, suppose main() function's pointer stores the address of that local variable inside the function, this way we can access address of that local variable as long as the function is executing.
- But once the function execution gets over, all internal variables goes to garbage collection and are not in memory anymore, but main() function's pointer is still pointing to that particular address which is now not available in memory, hence it would be called as a Dangling Pointer.



```
#include<iostream>
using namespace std;
int *fun()
{
    int y=10;
    return &y;
}
int main()
{
    int *p=fun();
    cout<<*p<<endl;
    return 0;
}
```

```
In function 'int* fun()':
warning: address of local variable 'y' returned
[-Wreturn-local-addr]
```

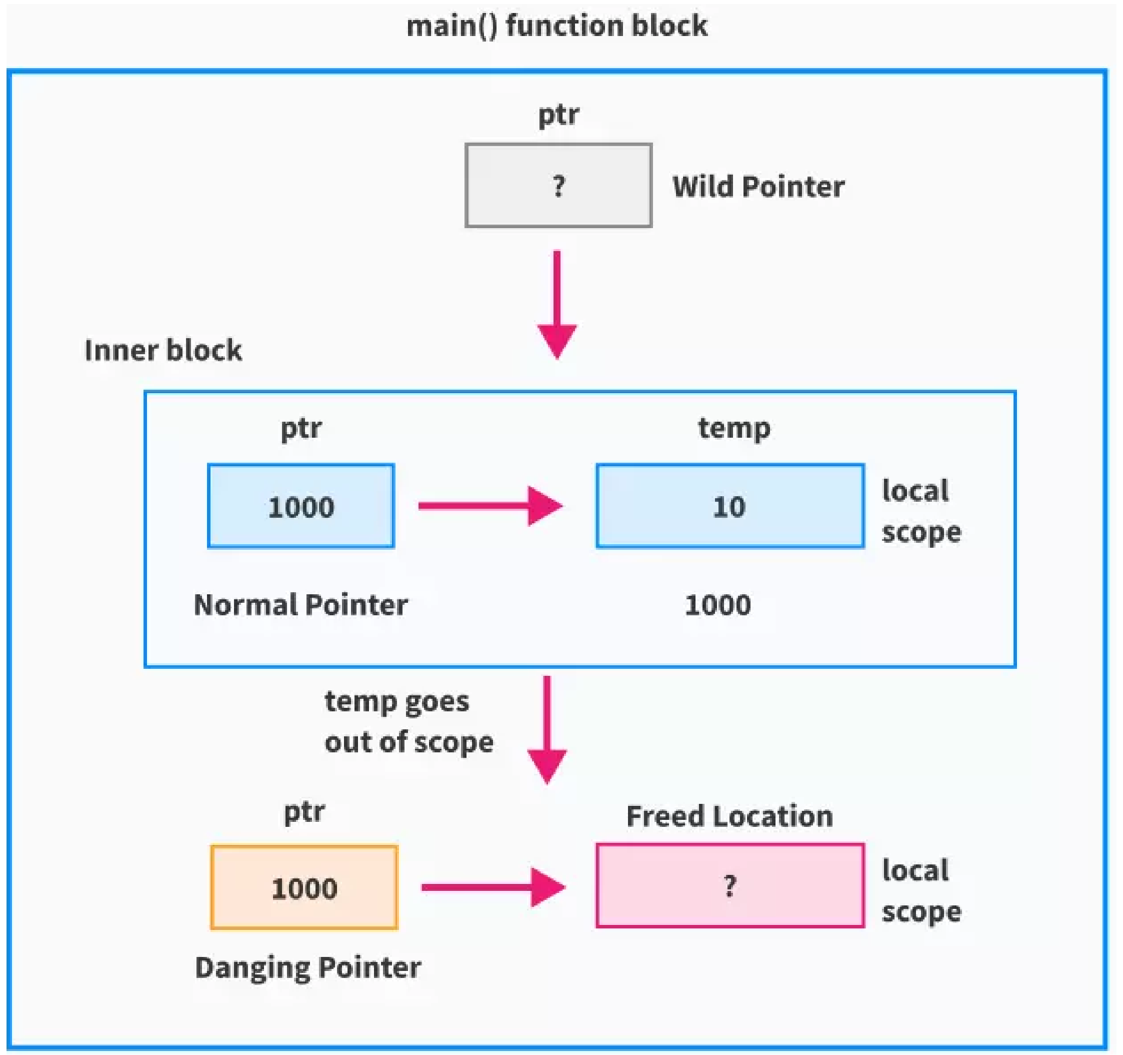
```
#include<iostream>
using namespace std;
int *fun()
{
    static int y=10;
    return &y;
}
int main()
{
    int *p=fun();
    cout<<*p<<endl;
    return 0;
}
```

Output: *10*

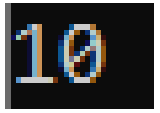
So, you need to remember that the pointer pointing to the local variable doesn’t become a dangling pointer when the local variable is static.

3. Variable goes out of scope:

- If a variable is declared inside some inner block of code, then the variable will have a local scope, and it will be deleted once the execution of the inner block ends.
- If the address of this local variable is assigned to a pointer declared outside the scope, then it will act as a Dangling Pointer outside the inner block of the code.



```
#include<iostream>
using namespace std;
int main()
{
    int *ptr;
    {
        int temp = 10;
        ptr=&temp;
    }
    cout<<*ptr<<endl;
    return 0;
}
```



```
10
```

5. Smart pointer:

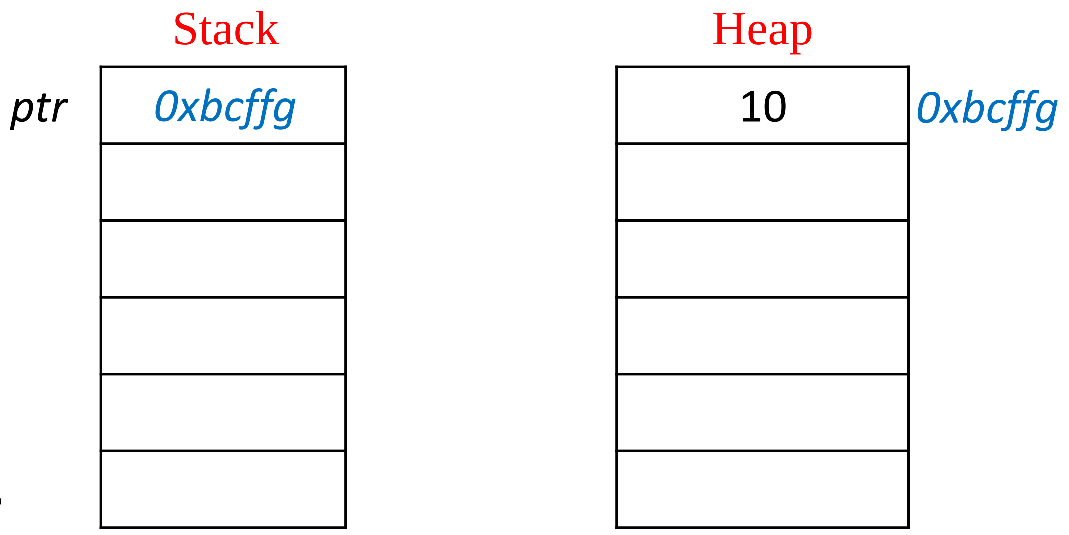
- A Smart Pointer is a wrapper class over a pointer with an operator like * and -> overloaded. The objects of the smart pointer class look like normal pointers. But, unlike Normal Pointers, it can deallocate and free destroyed object memory.

Problems with normal pointer

- To use a pointer you must allocate the memory for that pointer variable. The problem with them is that if you forget to free the memory held by the pointer will be occupied, for the entire duration the program runs. This is called a **memory leak**.
- **Example:**

```
int main()
{
   int *ptr=new  int (10)
   return 0;
}
```

After executing the program stack memory will be deallocated, But the heap memory will not be cleared unless untill you free the memory. This is called as memory leak.



Types of Smart Pointers

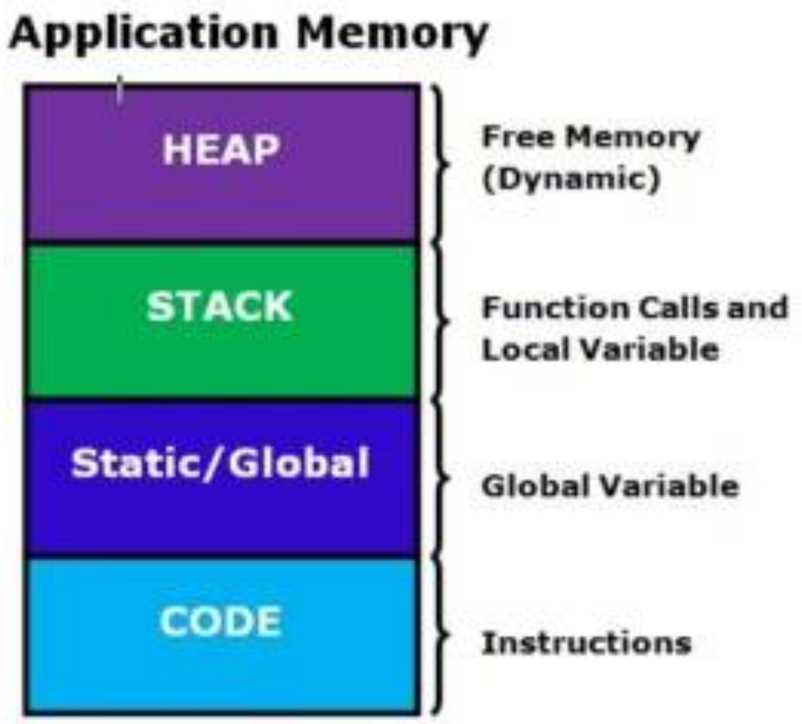
i. Unique Pointer

ii. Shared Pointer

iii. Weak Pointer

Dynamic Memory Allocation

- Dynamic memory allocation is the process of assigning the memory space during the execution time or the run time.
- In C++, dynamic memory is allocated using some standard library functions in *stdlib.h* header file.
- Static memory allocation can only be done on stack whereas dynamic memory allocation can be done on both stack and heap.
- A heap is an area of pre-reserved computer main memory that a program process can use to store data in some variable amount that won't be known until the program is running.
- A stack is similar to a heap except that the blocks are taken out of storage in a certain order and returned in the same way.
- While allocating memory on heap we need to delete the memory manually as memory is not freed (deallocated) by the compiler itself even if the scope of allocated memory finishes (as in case of stack).



Dynamic memory allocation in c language is possible by 6 functions of stdlib.h header file.

a. malloc()

b. calloc()

c. realloc()

d. free()

e. new()

f. delete()

Requirement:

- ➢ Let us consider, int a [9] = {4,5,8,7,9,15,23,2,17} is an array of size 9.
  - ✓ If there is a situation where only 6 elements are needed to be entered in this array. In this case, the remaining 2 indices are just wasting memory in this array. So there is a requirement to lessen the size of the array from 9 to 6.
  - ✓ Take another situation. In this, there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the size of the array needs to be changed from 9 to 12.

a. malloc(): (memory allocation)

- ➢ The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.
- ➢ It doesn't initialize memory at execution time, so it has garbage value initially.
- ➢ It returns NULL if memory is not sufficient.

Syntax:

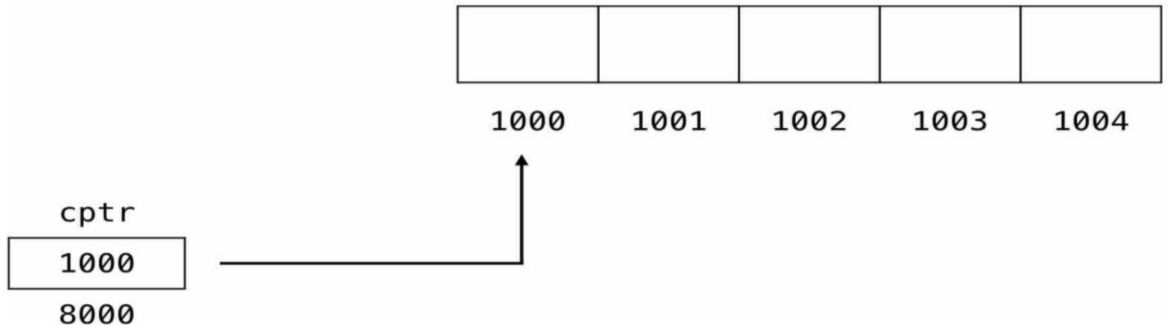
pointer_name = (cast_Type) malloc(byte_size);*

- Pointer_name is a pointer of cast_Type.
- The malloc function returns a pointer to the allocated memory of byte_size.

Example:

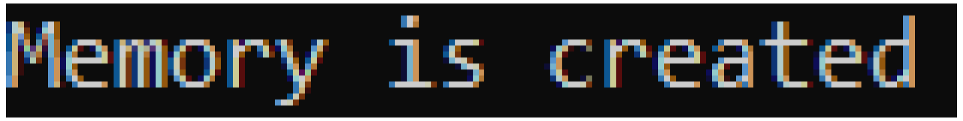
char *ptr = (char*) malloc(5 * sizeof(char));

- Since the size of char is 1 byte, this statement will allocate 5 byte of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



Program to check memory is created using the malloc() function

```
#include <iostream>
using namespace std;
int main()
{
        int *p;
        p=(int*)malloc(sizeof(int));
        if(p!=NULL)
        {
                cout<<"Memory is created";
        }
        else
        {
                cout<<"Memory is not created";
        }
}
```



```
Memory is created
```

Program to create a dynamic memory using the malloc() function

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
        int n,i,*p, sum=0;
        cout<<"Enter no of elements:"<<endl;
        cin>>n;
        p=(int*)malloc(n*sizeof(int));
        cout<<p<<'\t'<<*p<<endl;
        cout<<"Enter elements of array:"<<endl;
        for(i=0;i<n;i++)
        {
                cin>>*(p+i);
                sum+=*(p+i);
        }
        cout<<"Sum is "<<sum;
        free(p);
        return(0);
}
```

```
Enter no of elements:
4
0x2766e00     41316312
Enter elements of array:
10
8
21
4
Sum is 43
```

b. calloc(): (Contiguous memory allocation)

- A calloc() function is used to create multiple blocks at the run time of a program having the same size in the memory.
- It has two parameters, no. of blocks and the size of each block. When the dynamic memory is allocated using the calloc() function, it returns the base address of the first block, and each block is initialized with 0. And if memory is not created, it returns a NULL pointer.

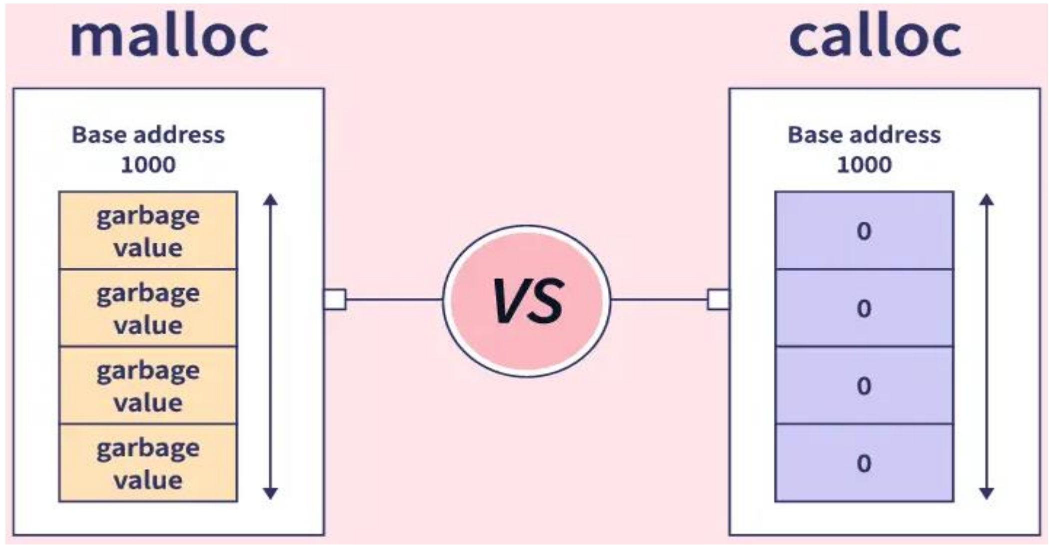
Syntax:

pointer_name = (cast_Type) calloc(number, byte_size);*

Example:

int *ptr = (int*) calloc(5, sizeof(int));

- Since the size of int is 4 bytes, this statement will allocate 20 bytes of memory.



```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
        int n,i,*p, sum=0;
        cout<<"Enter no of elements:"<<endl;
        cin>>n;
        p=(int*)calloc(n,sizeof(int));
        cout<<p<<'\t'<<*p<<endl;
        cout<<"Enter elements of array:"<<endl;
        for(i=0;i<n;i++)
        {
                cin>>*(p+i);
                sum+=*(p+i);
        }
        cout<<"Sum is "<<sum;
        free(p);
        return(0);
}
```

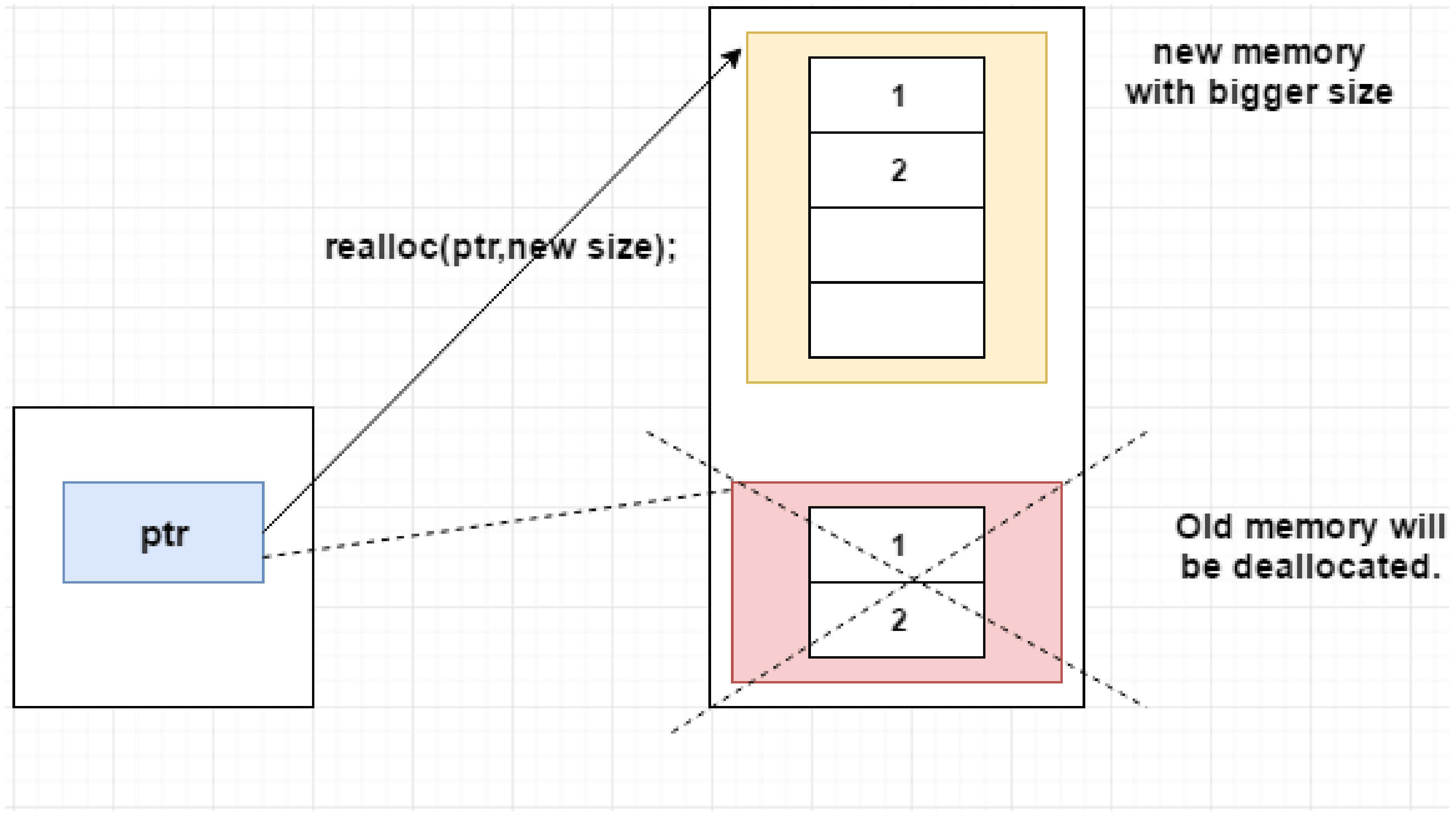
```
Enter no of elements:
4
0x2736e00      0
Enter elements of array:
10
8
21
4
Sum is 43
```

c. realloc(): (Re-allocation)

- The realloc() function in the C++ programming language is used to resize the memory block pointed to a pointer that was previously allocated to the variable by the malloc() or calloc() function.
- If the pointer passed to realloc is NULL, it behaves like malloc and allocates a new memory block of the given size. If the size passed to realloc is zero, it behaves like free and deallocates the memory block pointed to by the pointer.
- It is important to note that realloc may move the memory block to a new location if it cannot extend or shrink the existing block in place. It means that any pointers or references to the old block become invalid after the call to realloc.

Syntax:

pointer_name = (cast_Type) realloc (pointer_name ,new_size);*


new memory
with bigger size
1
2
realloc(ptr,new size);
ptr
1
2
Old memory will
be deallocated.

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
        int *p=(int*)malloc(2*sizeof(int));
        int i;
        int *ptr;
        *p=10; *(p+1)=14;
        ptr=(int*)realloc(p,3*sizeof(int));
        *(ptr+2)=20;
        for(i=0;i<3;i++)
        {
                cout<<*(ptr+i)<<endl;
        }
        free(p);
        cout<<p<<'\t'<<*p<<endl;
        cout<<*(p+1)<<endl;
        cout<<*(p+2);
        return(0);
}
```

Output:

```
10
14
20
0x27b6e00       41643992
41025728
20
```

d. free(): (Free-allocation)

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.
- The free() function takes a single parameter. If the pointer passed is a null pointer nothing is happened.
- free in C++ doesn't return any value. It has a return type of void.

Syntax:

free(pointer_name);

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
        int *p=(int*)malloc(1*sizeof(int));
        int *ptr=(int*)calloc(1,sizeof(int));
        *p=10; *ptr=14;
        cout<<*p<<'\t'<<*ptr<<endl;
        free(p);
        free(ptr);
        cout<<*p<<'\t'<<*ptr<<endl;
        return(0);
}
```

Output:

```
10      14
188253 771165677
```

e. new:

- The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.
- When you create an object of class using new keyword(normal new).
  - The memory for the object is allocated using operator new from heap.
  - The constructor of the class is invoked to properly initialize this memory.
- **Syntax:** *pointer-variable = new data-type; or pointer-variable = new data-type(value);*

  Example:// Pointer initialized with NULL. Then request memory for the variable

  *int *p = NULL;*

  p = new int;

  OR

  // Combine declaration of pointer and their assignment

  *int *p = new int;*

```
#include <iostream>
using namespace std;
int main()
{
   // pointer to store the address returned by the new
   int* ptr;
   // allocating memory for integer
   ptr = new int;
   // assigning value using dereference operator
   *ptr = 50;
   // printing value and address
   cout << "Address: " << ptr << endl;
   cout << "Value: " << *ptr;
   return 0;
}
```

Output:

```
Address: 0x2796d98
Value: 50
```

e. delete:

- It is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.
- ***Syntax:***

```
// Release memory pointed by pointer-variable
delete pointer-variable;
                    Or
// Release block of memory pointed by pointer-variable
delete[] pointer-variable;
```

```
#include <iostream>
using namespace std;
int main ()
{
   int *ptr1, *ptr2;
   // allocated memory space using new operator
   ptr1 = new int;
   ptr2 = new int;
   *ptr1=5; *ptr2=15;
   cout<<ptr1<<'\t'<<*ptr1<<endl;
   cout<<ptr2<<'\t'<<*ptr2<<endl;
   // delete pointer variable
   delete ptr1;
   cout<<ptr1<<'\t'<<*ptr1<<endl;
   cout<<ptr2<<'\t'<<*ptr2<<endl;
   return 0;
}
```

Output:

```
0x1f6d98      5
0x1f6da8      15
0x1f6d98      2059704
0x1f6da8      15
```

```
#include <iostream>
using namespace std;
int main ()
{
   // declaration of variables
   int *arr, n, i;
   n=4;
   // use new operator to declare array memory at run
time
   arr = new int [n];
   cout << " Enter the numbers: \n";
   for (i = 0; i< n; i++)  // input array from user
   {
      cout << " Number " << i+1 << " is ";
      cin >> arr[i];
   }
   cout <<" Numbers are : ";
   for (i = 0; i < n; i++)
   {
      cout << arr[i] << "\t";
   }
   cout<<endl;
   delete [2] arr;
    for (i = 0; i < n; i++)
      {
         cout << arr[i] << "\t";
      }
       return 0;
   }
```

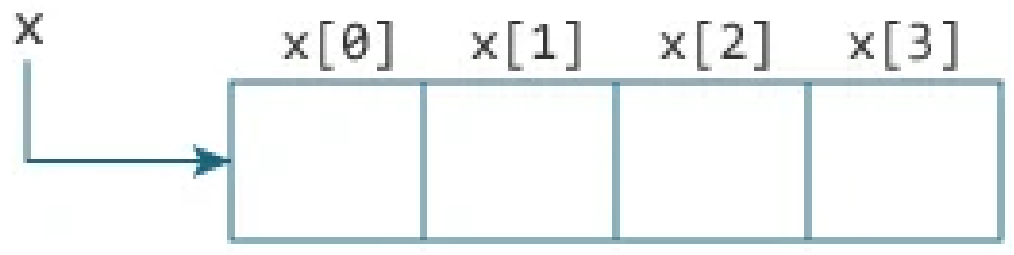
```
Enter the numbers:
 Number 1 is 5
 Number 2 is 12
 Number 3 is 7
 Number 4 is 9
 Numbers are : 5       12      7       9
8023928     7995584    7       9
```

malloc()	**new**
It is a function from the C standard library.	It is a C++ operator.
It only allocates memory.	It allocates memory and calls the constructor of the object being created.
It returns a pointer to the first byte of the allocated memory.	It returns a pointer to the newly constructed object.
Memory allocated with malloc() must be explicitly freed using the free() function.	Memory allocated with new is automatically freed when the program exits.
It returns void *.	It returns exact data type.
It does not perform type checking.	It checks for type correctness and throws an exception if the allocation fails.
It does not call constructors.	It calls constructors.
malloc() cannot be overloaded.	Operator new can be overloaded.
It can be used to allocate aligned memory.	It cannot be used to allocate aligned memory.

delete	**free()**
It is an operator.	It is a library function.
It de-allocates the memory dynamically.	It destroys the memory at runtime.
It should only be used for deallocating the memory allocated either using the new operator or for a NULL pointer.	It should only be used for deallocating the memory allocated either using malloc(), calloc(), realloc() or for a NULL pointer.
This operator calls the destructor before it destroys the allocated memory.	This function only frees the memory from the heap. It does not call the destructor.
It is comparatively slower because it invokes the destructor for the object being deleted before deallocating the memory.	It is faster than delete operator.

Pointers and Arrays:

- Array and Pointers in C++ Language hold a very strong relationship. Generally, pointers with arrays stores the starting address of the array.
- Array name itself acts as a pointer to the first element of the array and also if a pointer variable stores the base address of an array then we can manipulate all the array elements using the pointer variable only.
- Pointers can be associated with the multidimensional arrays (2-D and 3-D arrays) as well. Also, We can create an array of pointers to store multiple addresses of different variables.
- There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).
- Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.



- From the above example, it is clear that &x[0] is equivalent to x. And, x[0] is equivalent to *x.

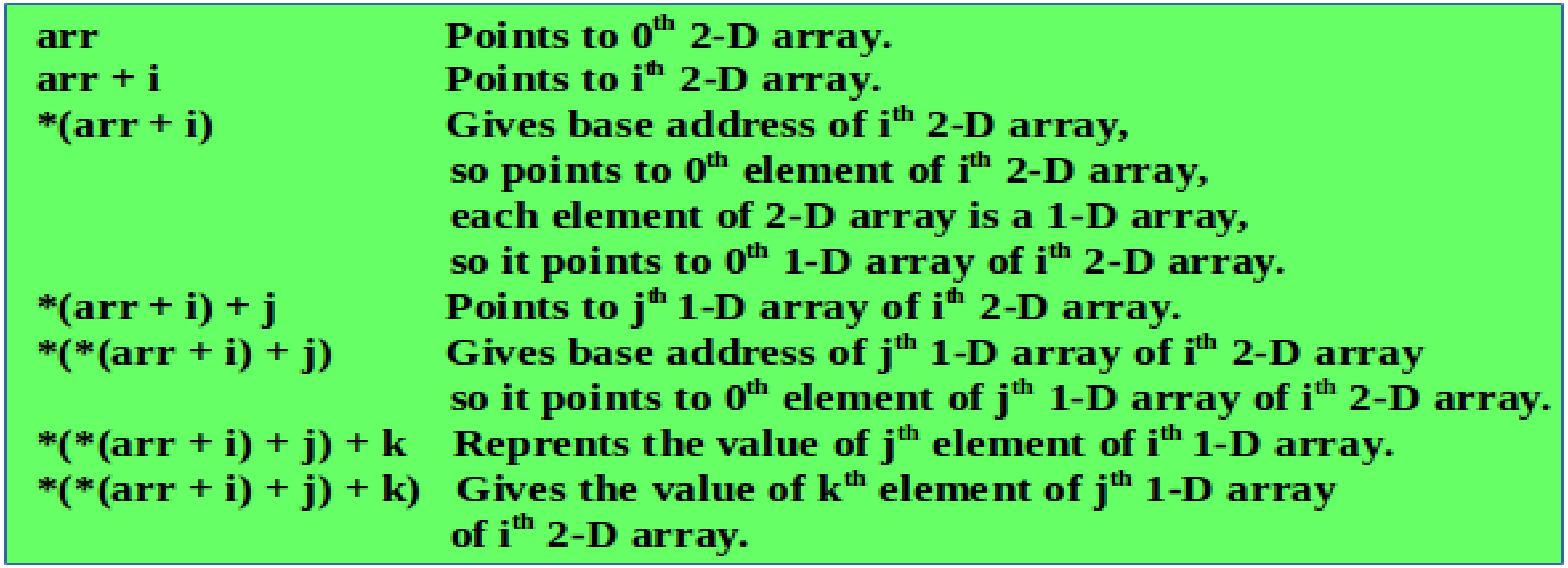
  Similarly,

  &x[1] is equivalent to x+1 and x[1] is equivalent to *(x+1).

  &x[2] is equivalent to x+2 and x[2] is equivalent to *(x+2).

  ...

  Basically, &x[i] is equivalent to x+i and x[i] is equivalent to *(x+i).



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

Example-1

```
#include <iostream>
using namespace std;
int main ()
{
    int i;
    int arr[5]={5,7,8,9,11};
    int *ptr=arr;
    for (i=0; i<5; i++)
    {
        cout << (ptr+i) << "\t";
        cout << *(ptr+i) << "\t";
    }
     return 0;
}
```

Output:

```
0x61fef4        5       0x61fef8        7       0x61fefc        8
0x61ff00  9       0x61ff04        11
```

Pointers as Function Arguments:

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

Syntax:

*return_type pointer_name (*arg_1, *arg_2, ...);*

Examples:

float add (*int a, *int b);   // function declaration

```
#include <iostream>
using namespace std;
void swap (int *a, int *b);
int main ()
{
   int m,n;
   m=10; n=20;
   swap(&m,&n);
   cout<<m<<'\t'<<n<<endl;
   return 0;
}
void swap(int *a, int *b)
{
   int temp;
   temp=*a;
   *a=*b;
   *b=temp;
}
```

Output:

```
20     10
```

Functions returning Pointer variables:

- A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

Syntax:

*type (*pointer-name)(Argument);*

Examples:

```
float add(int, int);                    // declaration of a function
float (*s)(int, int);        // declaration of a pointer to a function
s=add;                                  // assigning address of add() to 'a' pointer
```

- Here s is a pointer to a function add. Now add can be called using function pointer s along with providing the required argument value

```
#include<iostream>
using namespace std;
int *larger(int*,int*);
int main()
{
    int a,b, *ptr;
    a=15; b=92;
    ptr=larger(&a,&b);
    cout<<"The larger value is "<<*ptr;
    return 0;
}
```

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

Output:

```
The larger value is 92
```

Pointer to Pointer (Double Pointer)

- A pointer-to-pointer (double pointer) is used to store the address of another pointer.
- The first pointer stores the address of a variable, and the second pointer stores the address of the first pointer.
- This makes them useful for changing the values of normal pointers or creating variable-sized 2-D arrays. Importantly, a double pointer occupies the same amount of memory as a regular pointer on the stack.

Syntax:

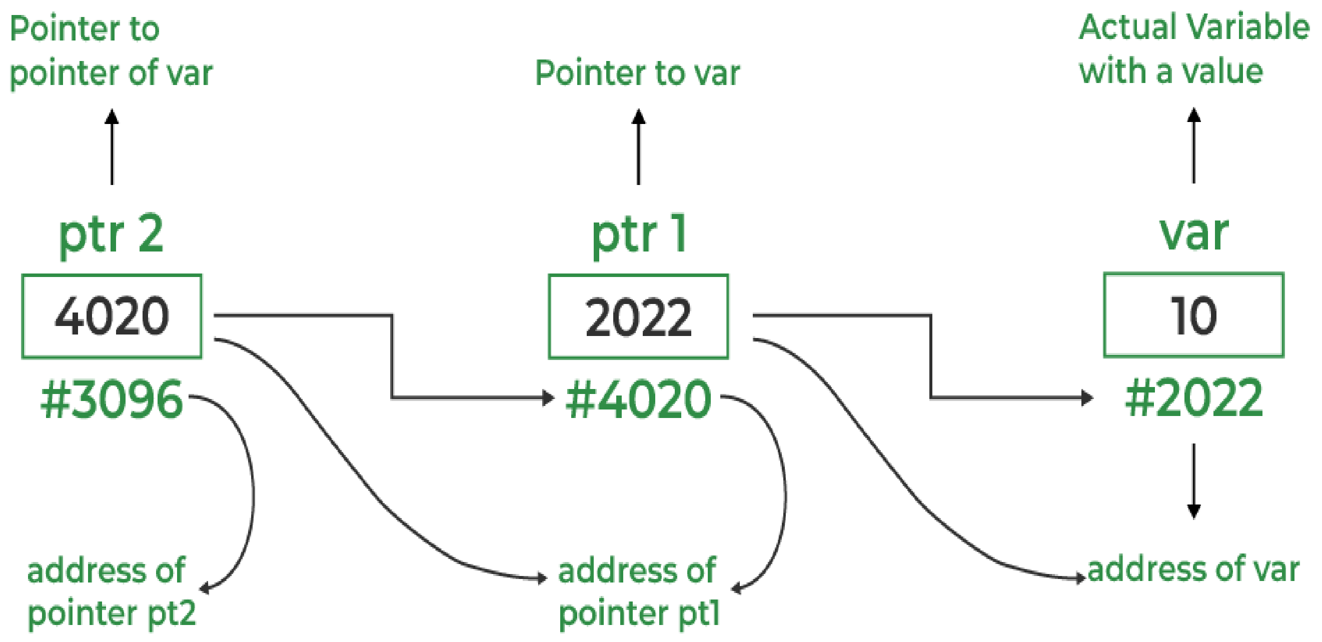
*data_type_of_pointer **double_pointer_name= & first_pointer_name;*

Examples:

int val = 5;

int *ptr = &val; // storing address of val to pointer ptr.

int **d_ptr = &ptr; // pointer to a pointer declared



```
#include<iostream>
using namespace std;
int main()
{
    int var=20;
    int* ptr1, **ptr2;
    ptr1=&var; ptr2=&ptr1;
    cout<<"The value of var is "<<var<<endl;
    cout<<"Address of single pointer "<<ptr1<<endl;
    cout<<"Value of single pointer "<<*ptr1<<endl;
    cout<<"Address of double pointer "<<ptr2<<endl;
    cout<<"Value of double pointer "<<*ptr2<<endl;
    return 0;
}
```

Output:

```
The value of var is 20
Address of single pointer 0x61ff08
Value of single pointer 20
Address of double pointer 0x61ff04
Value of double pointer 0x61ff08
```

Application of Double Pointers in C++

Following are the main uses of pointer to pointers in C++:

- They are used in the dynamic memory allocation of multidimensional arrays.
- They can be used to store multilevel data such as the text document paragraph, sentences, and word semantics.
- They are used in data structures to directly manipulate the address of the nodes without copying.
- They can be used as function arguments to manipulate the address stored in the local pointer.

Command Line Arguments

- The command line is a text interface for your computer that allows you to enter commands for immediate execution.
- C++ allows passing values from the command line at execution time in the program.
- A command-line can perform almost everything that a graphical user interface can.
- Many tasks can be performed more rapidly and are easier to automate.
- Command line arguments are the arguments specified after the program name in the operating system's command line. These argument values are passed to your program during execution from your operating system.
- They are used when we need to control our program from outside instead of hard-coding it.
- They make installation of programs easier.

Syntax:

*int main(int argc, char *argv[])*

argc denotes the number of arguments given which is a non-negative integer

argv[] is a pointer array pointing to each parameter passed to the program which is a character datatype.

- ➢ The name of the program is stored in *argv[0]*, the first command-line parameter in *argv[1]*, and the last argument in *argv[n]*.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main(int argc, char *argv[])
{
    cout<<"Program name is ";
    puts (argv[0]);
    if(argc<2)
    {
        cout<<"No argument passed through command line.";
    }
    else
    {
        cout<<"First argument is: ";
        puts(argv[1]);
    }
    return 0;
}
```

Output:

```
Program name is C:\Users\drjrn\OneDrive\Desktop\Coding\c++\arg_pass.exe
No argument passed through command line.
```

```
#include<iostream>
#include<conio.h>
using namespace std;
int main(int argc, char *argv[])
{
   cout<<"No of command line argument: "<<argc<<endl;
   for (int i=0;i<argc;i++)
   {
      cout<<"argv["<<i<<"]:";
      puts(argv[i]);
   }
   return 0;
}
```

Output: *No of command line argument: 1*

```
argv[0]:C:\Users\drjrn\OneDrive\Desktop\Coding\c++\arg_pass.exe
```

argv[0] will be the program/file name. Then after file name we can enter characters followed by spaces

Output:

```
C:\Users\drjrn\OneDrive\Desktop\Coding\c++\arg_pass.exe Training class of ITER
No of command line argument: 5
argv[0]:C:\Users\drjrn\OneDrive\Desktop\Coding\c++\arg_pass.exe
argv[1]:Training
argv[2]:class
argv[3]:of
argv[4]:ITER
```

Advantages of Command-Line Arguments in C++

- A command-line argument allows us to provide an unlimited number of arguments.
- The data is passed as strings as arguments, so we can easily convert it to numeric or other formats.
- It is useful for configuration information while launching our application.
- Command-line arguments are useful when you want to control your program from outside rather than hard coding the values inside the code.

Thank You