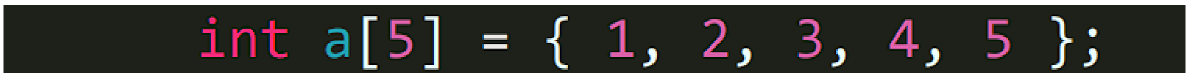
Space Complexity of Array Operation

We already know that space we need for fixed component and variable part more precisely Auxiliary space or extra space we need and input space i.e. the input components.

1. *Initialization*:



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

$for\ a[1] = 1 \rightarrow\ O(1) space\ complexity$.

$for\ a[2] = 2 \rightarrow\ O(1) space\ complexity$.

$for\ a[3] = 3 \rightarrow\ O(1) space\ complexity$.

$for\ a[4] = 4 \rightarrow\ O(1) space\ complexity$.

$for\ a[5] = 5 \rightarrow\ O(1) space\ complexity$.

We see that we not reusing the space rather we are expanding the auxiliary space to enter each element at $O(1)$ constant space complexity.

Hence we can sum up like :

$\boldsymbol{O(1) + O(1) + O(1) + O(1) + O(1) = O(1)}$

$\boldsymbol{or\ we\ can\ say,\ O(5) = O(1)\ as\ 5\ is\ the\ size\ of\ the\ array.}$

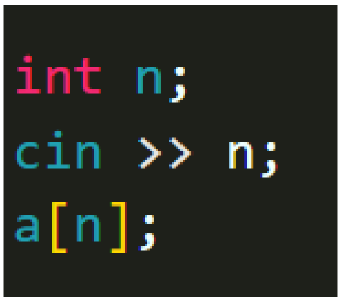
$\boldsymbol{This\ is\ Input\ Space\ Complexity.}$

$\boldsymbol{Also,\ here\ Auxiliary\ Space\ Complexity\ \ is\ also\ O(1), as,}$

$\boldsymbol{no\ extra\ memory\ space\ is\ used, hence,}$

$\boldsymbol{Total\ Space\ Complexity = Input\ Space + Auxiliary\ Space}$

$\boldsymbol{= O(1) + O(1) = O(1).}$



```
int n;
cin >> n;
a[n];
```

$\boldsymbol{Now\ say\ we\ if\ the\ array\ size\ is\ n. Then\ the\ space\ complexity}$

$\boldsymbol{is\ O(n)\ and\ this\ is\ input\ space\ complexity.}$

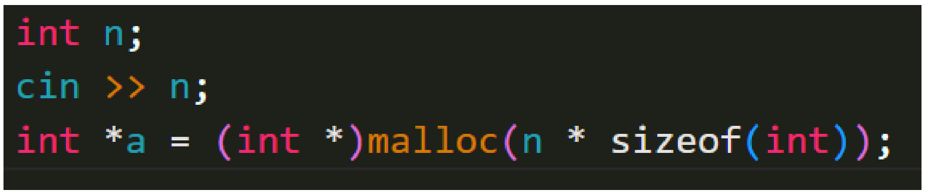
$\boldsymbol{Also,\ here\ Auxiliary\ Space\ Complexity\ \ is\ also\ O(1), as,}$

$\boldsymbol{no\ extra\ memory\ space\ is\ used, hence,}$

$\boldsymbol{Total\ Space\ Complexity = Input\ Space + Auxiliary\ Space}$

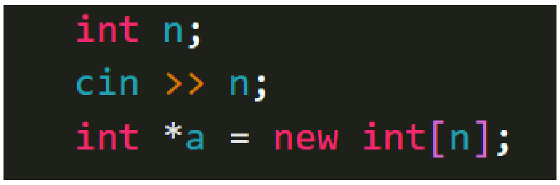
$\boldsymbol{= O(n) + O(1) = O(n).}$

$\boldsymbol{Similarly,}$



```
int n;
cin >> n;
int *a = (int *)malloc(n * sizeof(int));
```

And



```
   int n;
   cin >> n;
   int *a = new int[n];
```

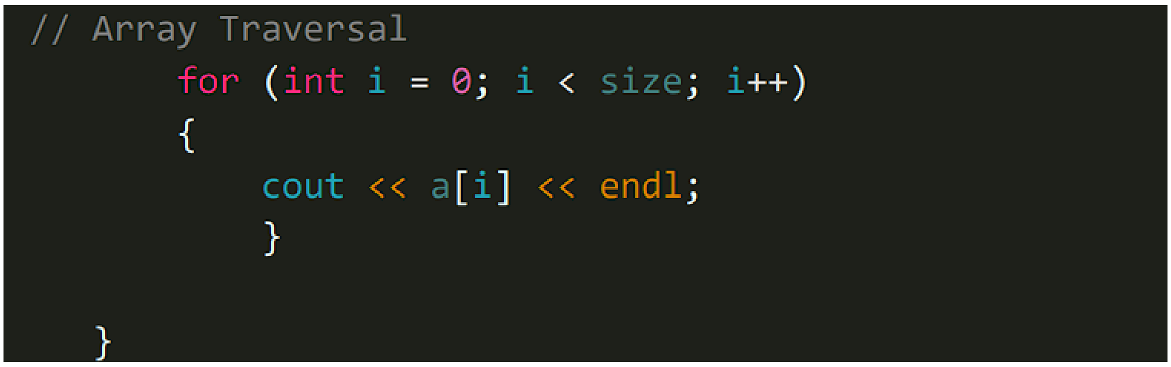
The space complexity of array having dynamic size n is $O(n)$, ***this is input space complexity.***

Also , here Auxiliary Space Complexity is also $O(1)$, ***as, no extra memory space is used, hence ,***

$$\textit{Total Space Complexity} = \textit{Input Space} + \textit{Auxiliary Space}$$
$$= O(n) + O(1) = O(n).$$

2. Traversal of arrays

Array Traversal looks like:



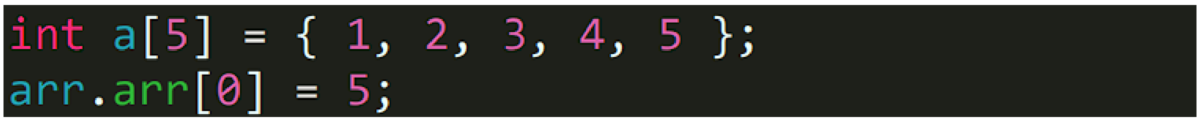
```
// Array Traversal
        for (int i = 0; i < size; i++)
        {
            cout << a[i] << endl;
            }

    }
```

There is no insertion or deletion of any elements in arrays hence it takes no space taken.
Therefore no space complexity generated for traversal of arrays.

Here i is the extra space i.e. Auxiliary Space $= O(1)$ but Input Space Complexity remain \`n\` i.e. $size = n$, hence:

$$\text{Total Space Complexity} = \text{Input Space} + \text{Auxiliary Space}$$
$$= O(n) + O(1) = O(n).$$

3.*Overriding ith element*



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

$a[0] = 1 = 5$ ***gets overriden as only it modifies the 1st element of the array. Hence it doesnot change the fact that the auxiliary space remaining constant i. e.*** $O(1)$.

Here $a[5]$ ***is used , hence fixed size , hence input space complexity also*** $O(1)$.

$\therefore$ ***Total Space Complexity = Input Space + Auxiliary Space***

$$= O(1) + O(1) = O(1).$$

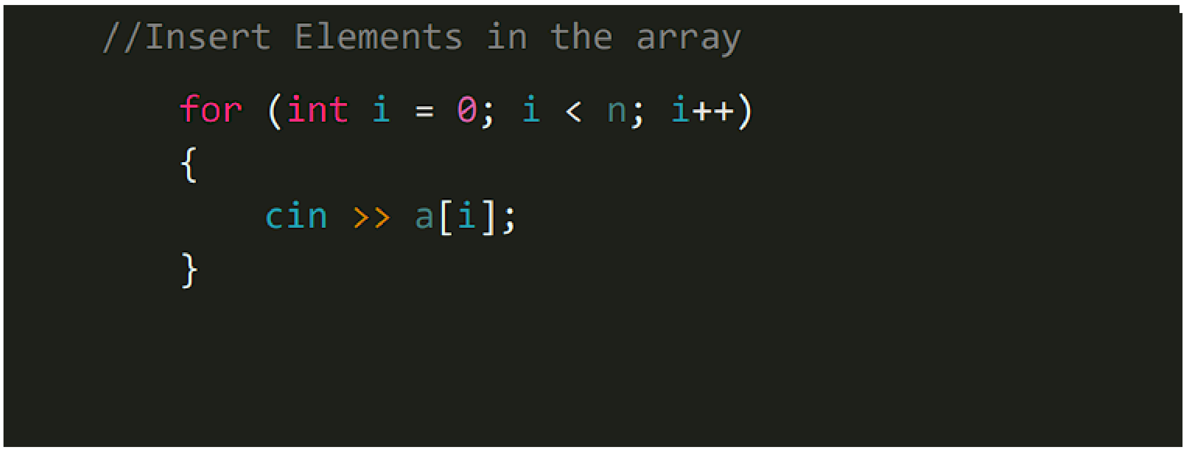
But if it was : $a[n]$ ***, i. e. n size , then ,***

Input Space Complexity becomes $O(n)$ ***and***

$\therefore$ ***Total Space Complexity becomes = Input Space + Auxiliary Space***

$$= O(n) + O(1) = O(n).$$

4. *Insert Element in Array*



```
//Insert Elements in the array
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
```

When we are inserting each element in each input space as arrays are arranged in contiguous memory location.

n	$\rightarrow \boldsymbol{O}(1)$
.	$\rightarrow \boldsymbol{O}(1)$
.	$\rightarrow \boldsymbol{O}(1)$
3	$\rightarrow \boldsymbol{O}(1)$
2	$\rightarrow \boldsymbol{O}(1)$
1	$\rightarrow \boldsymbol{O}(1)$

space remaining constant i.e. input space already created

during creation of array and we input the elements in

that input space which has been already created.

Hence values gets inserted in that input space i.e.,

$\boldsymbol{O(1) + O(1) + O(1) + \cdots n\ times = O(1) \times n = O(n)}.$

That is input space is : $\boldsymbol{O(n)}$.

The loop here helps to input elements in the allocated space

for arrays defines time complexity not space complexity

more over already input space allocated for array is `n`.

Hence loop doesn'take part in space complexity directly but:

The only additional space required is for integer `i` in the

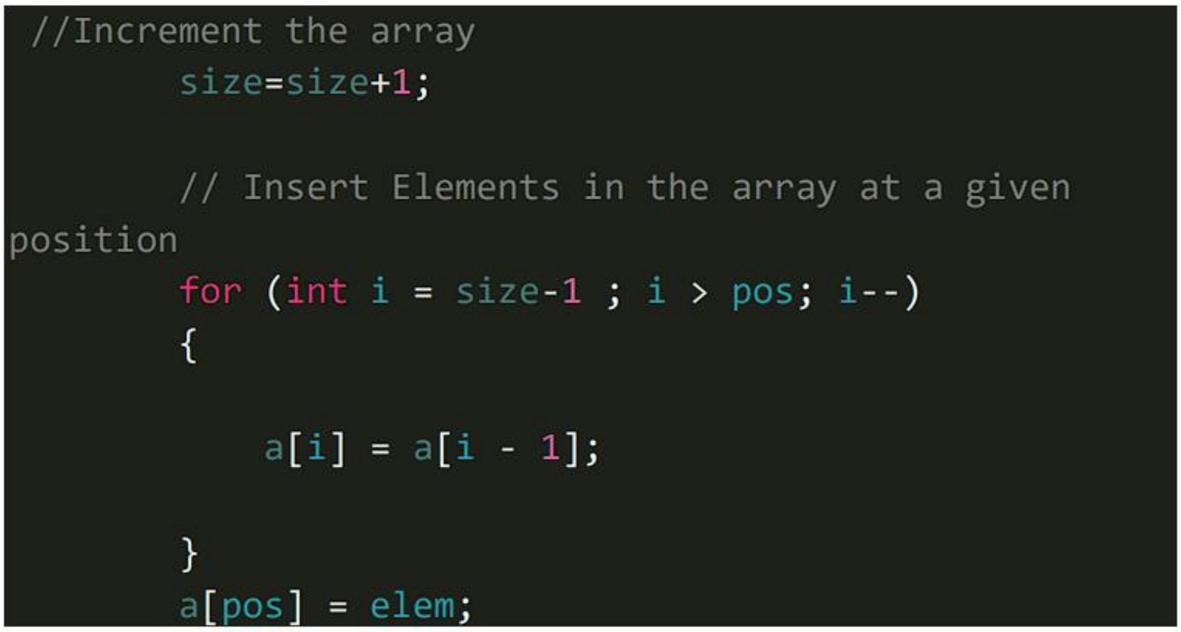
memory , that is the auxiliary space complexity, which is

constant = $\boldsymbol{O(1)}$.

$\therefore$ ***Total Space Complexity becomes = Input Space + Auxiliary Space***

$\boldsymbol{= O(n) + O(1) = O(n)}.$

6. *Insert Element at position In an Array*



```
//Increment the array
        size=size+1;

        // Insert Elements in the array at a given
position
        for (int i = size-1 ; i > pos; i--)
        {

            a[i] = a[i - 1];

        }
        a[pos] = elem;
```

1st We increment the size of the array: size = size + 1.

2nd we shift the element from position to last index:
suppose size = 5 and we have indexes: a[0], a[1], a[2], a[3] and a[4],as size increased , now we have array indexes: a[0], a[1], a[2], a[3], a[4] , a[5] and size = 6.

Now we will put the at index 1. Hence we will shift elements:

- $a[5] = a[4]\ \left[shifted\ to\ right\left[a[4] \rightarrow a[5]\right]\right] \rightarrow 1\ unit\ time$
- $a[4] = a[3]\left[shifted\ to\ right\left[a[3] \rightarrow a[4]\right]\right] \rightarrow 1\ unit\ time$
- $a[3] = a[2]\left[shifted\ to\ right\left[a[2] \rightarrow a[3]\right]\right] \rightarrow 1\ unit\ time$
- $a[2] = a[1]\left[shifted\ to\ right\left[a[1] \rightarrow a[2]\right]\right] \rightarrow 1\ unit\ time$

$Now\ we\ just\ will\ do\ is\ overriding\ the\ ith\ element$:

- $a[1] = elem\ (User\ Input) \rightarrow O(1) space\ complexity$

$\underline{Analysis\ goes\ like}$:

$size = size + 1\ doesnot\ require\ any\ additional\ space.$

$The\ second\ line\ of\ code, for(int\ i = size - 1; i > pos; i - -),$
$iterates\ through\ the\ last\ element\ to\ the\ element\ before\ the$
$specified\ position. For\ each\ iteration\ , the\ value\ of\ the\ current$
$is\ copied\ to\ the\ next\ element. This\ \ does\ not\ require\ any$
$additional\ space, as\ the\ values\ of\ the\ elements\ are$
$simply\ copied\ to\ the\ input\ space\ created\ for\ the\ array.$

$The\ third\ line\ of\ code\ a[pos] = elem, assigns\ the\ value\ of$
$the\ new\ element\ to\ the\ specified\ position\ of\ the\ array.$
$This\ doesnot\ require\ any\ additional\ space\ , as\ the\ value\ of\ the$
$new\ element\ is\ simply\ being\ stored\ in\ the\ input\ space.$

$The\ only\ space\ required\ is\ integer\ \ i\ variable\ \ i.e. the\ extra$
$space\ or\ the\ auxiliary\ space.$

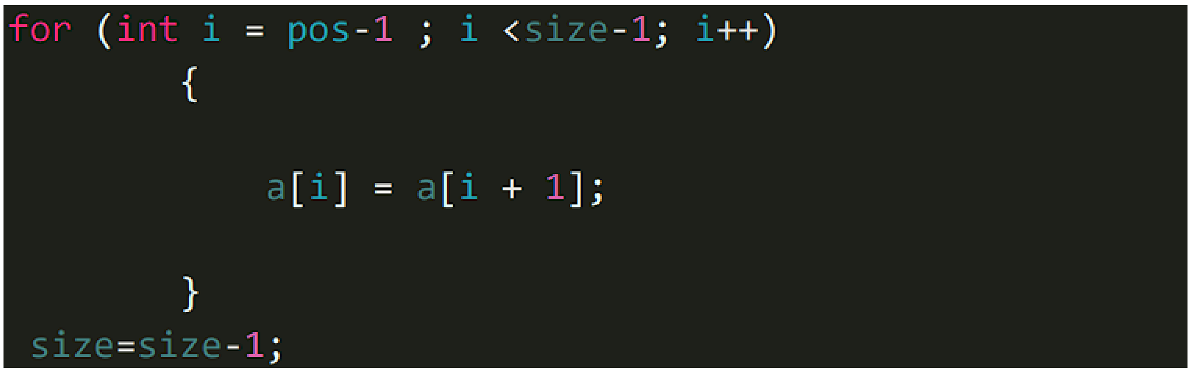
Therefore Input space becomes: $O(n+1) = O(n)$.

The auxiliary space becomes : $O(1)$.

$\therefore$ ***Total Space Complexity becomes*** $=$ ***Input Space*** $+$ ***Auxiliary Space***

$= O(n) + O(1) = O(n)$.

7.Delete Element at position In an Array



```
for (int i = pos-1 ; i <size-1; i++)
        {

            a[i] = a[i + 1];

        }
 size=size-1;
```

Here also we just shift the elements elements which doesnot acquire any additional space and decrease the size .

- ***Suppose we have size*** = **5** ***and And indices***: $a[0]$=1, $a[1]$=2, $a[2]$=3, $a[3]$=4 ***and*** $a[4]$=5.
- ***And we want to delete element at index***: **2** i.e. a[1].

- $\boldsymbol{Then\ we\ go\ through\ a\ process\ of\ Shifting\ to\ left}$:
    - $\boldsymbol{a[2-1=1]=a[2+1=3]} \boldsymbol{--\rightarrow elem: 3 \rightarrow takes\ O(1)\ constant\ space.} [\boldsymbol{a[3] copied\ to\ a[1]\ to\ the\ left}]$

    - $\boldsymbol{a[3-1=2]=a[3+1=4]} \boldsymbol{--\rightarrow elem: 4. \rightarrow takes\ O(1)\ constant\ space.} [\boldsymbol{a[4] copied\ to\ a[2]\ to\ the\ left}]$

    - $\boldsymbol{a[4-1=3]=a[4+1=5]} \boldsymbol{--\rightarrow elem: 5\ . \rightarrow takes\ O(1)\ constant\ space.} [\boldsymbol{a[5] copied\ to\ a[3]\ to\ the\ left}]$

$\boldsymbol{And\ a[0] will\ be\ remain\ untouched.}$

- $\boldsymbol{Then\ we\ decrement\ the\ size}$:
    - $\boldsymbol{size = size - 1}$, $\boldsymbol{now\ size\ is\ 4}$, $\boldsymbol{traversal\ will\ take\ place\ from\ a[0]=1}$, $\boldsymbol{a[1]=3}$, $\boldsymbol{a[2]=4}$, $\boldsymbol{and\ a[3]=5}$.

$\boldsymbol{Hence}$, $\boldsymbol{input\ space\ becomes\ O(n-1)=O(n).}$
$\boldsymbol{The\ only\ additional\ space\ it\ needs\ is\ for\ i\ in\ stack}$
$\boldsymbol{which\ is\ negligible\ i.e. Auxiliary\ space\ is : O(1).}$

$\therefore \boldsymbol{Total\ Space\ Complexity\ becomes = Input\ Space + Auxiliary\ Space}$

$\boldsymbol{= O(n) + O(1) = O(n).}$

$Hence\ the\ space\ complexity\ for\ an\ array$
$can\ be\ summed\ up\ like$:

$SPACE\ COMPLEXITY\ OF\ ARRAY\ OPERATION$
$When\ size\ is\ fixed\ not\ `n`\ .say : a[5]\ etc.$

Operation	**Input Space Complexity**	**Auxiliary Space Complexity**	**Total Space Complexity**
Initialization size	$O(1)$	$O(1)$	$O(1)$
Traversal	$O(1)$	$O(1)$	$O(1)$
Overriding ith element	$O(1)$	$O(1)$	$O(1)$
Insert element in the array	$O(1)$	$O(1)$	$O(1)$
Insert element at a position	$O(1)$	$O(1)$	$O(1)$
Delete element at a position	$O(1)$	$O(1)$	$O(1)$

$SPACE\ COMPLEXITY\ OF\ ARRAY\ OPERATION$

$When\ size\ is\ `n`\ : a[n], not\ fixed.$

Operation	Input Space Complexity	Auxiliary Space Complexity	Total Space Complexity
Initialization size	$O(n)$	$O(1)$	$O(n)$
Traversal	$O(n)$	$O(1)$	$O(n)$
Overriding ith element	$O(n)$	$O(1)$	$O(n)$
Insert element in the array	$O(n)$	$O(1)$	$O(n)$
Insert element at a position	$O(n)$	$O(1)$	$O(n)$
Delete element at a position	$O(n)$	$O(1)$	$O(n)$

Space Complexity of Two Dimensional Array

The space complexity of a two-dimensional array depends on its size and the data type it holds.

Let's assume that the two-dimensional array has dimensions N rows and M columns, and each element in the array occupies a fixed amount of space, regardless of the data type.

In this case, the space complexity of the two-dimensional array would be O(N * M) because you would need to allocate space for N rows and M columns. Each element takes up a fixed amount of space, so the total space required is proportional to the product of N and M.

However, if the size of each element in the array is not fixed, and it can vary depending on the data type or other factors, then you need to consider the space complexity on a per-element basis. For example, if each element in the array is an object that requires K units of space, then the space complexity would be O(N * M * K).

It's important to note that the space complexity only accounts for the storage required by the array itself and does not include any auxiliary data structures or variables used in algorithms or functions that operate on the array.

When $row\ is\ not\ equal\ to\ column\ then\ space\ created\ by$

$\boldsymbol{array\ is\ row * column\ , hence\ it\ is\ O(row * column) here.}$

$\boldsymbol{if\ row = m\ and\ column = n\ , then\ space\ complexity = O(m * n).}$

When $\boldsymbol{row\ is\ \ equal\ to\ column\ then\ space\ created\ by\ array\ is\ row * column = n^2, hence\ it\ is\ O(n^2).}$

Space Complexity of Three Dimensional Array

The space complexity of a three-dimensional array depends on its dimensions and the data type it holds.

Let's assume the three-dimensional array has dimensions N rows, M columns, and K depth, and each element in the array occupies a fixed amount of space, regardless of the data type.

In this case, the space complexity of the three-dimensional array would be O(N * M * K) because you would need to allocate space for N rows, M columns, and K depth. Each element takes up a fixed amount of space, so the total space required is proportional to the product of N, M, and K.

However, if the size of each element in the array is not fixed, and it can vary depending on the data type or other factors, then you need to consider the space complexity on a per-element basis. For example, if each element in the array is an object that requires L units of space, then the space complexity would be O(N * M * K * L).

It's important to note that the space complexity only accounts for the storage required by the array itself and does not include any auxiliary data structures or variables used in algorithms or functions that operate on the array.

When $\boldsymbol{row\ is\ not\ equal\ to\ column\ and\ depth\ is\ not\ equal\ to}$
$\boldsymbol{row\ and\ column\ then\ space\ created\ by}$
$\boldsymbol{array\ is\ depth * row * column\ , hence\ it\ is}$
$\boldsymbol{O(depth * row * column) here}.$
$\boldsymbol{if\ depth = p\ , row = m\ and\ column = n\ ,}$
$\boldsymbol{then\ space\ complexity = O(p * m * n)}.$

When $\boldsymbol{row\ is\ \ equal\ to\ column\ and\ depth\ is\ equal\ to\ row}$
$\boldsymbol{and\ column\ then\ space\ created\ by}$
$\boldsymbol{array\ is\ depth * row * column = n^3, hence\ it\ is\ O(n^3)}.$