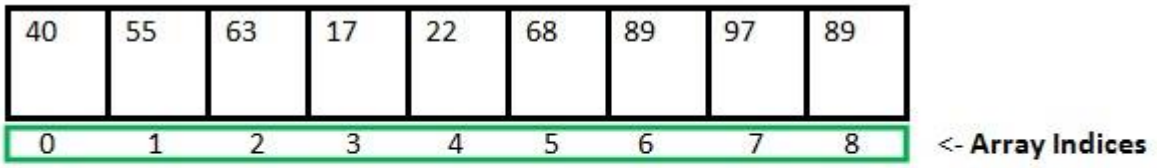
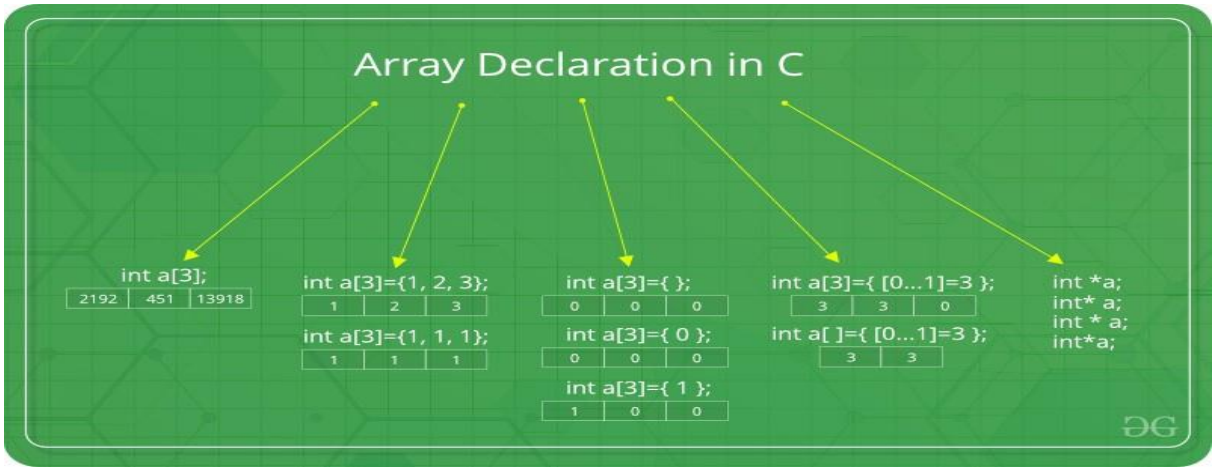
- An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.
- They can be used to store collection of primitive data types such as int, float, double, char, etc. of any particular type.
- To add to it, an array in C/C++ can store derived data types such as the structures, pointers etc. Given below is the picture representation of an array.



Array Length = 9
First Index = 0
Last Index = 8

- The idea of an array is to represent many instances in one variable.



- With recent C/C++ versions, we can also declare an array of user specified size int n = 10; int arr2[n];

Advantages of an Array in C/C++:

1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

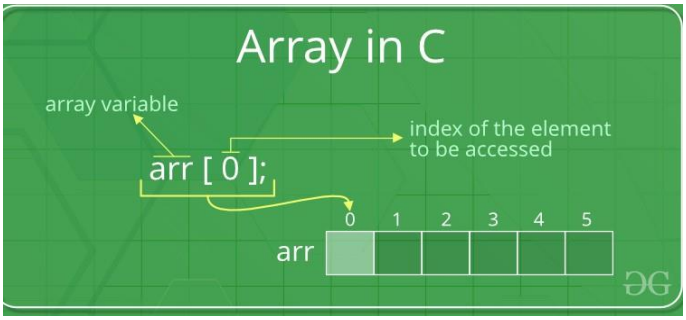
Disadvantages of an Array in C/C++:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.

2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation

Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.

- Name of the array is also a pointer to the first element of array.



- **No Index Out of bound Checking:**
  There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

```
/ This C++ program compiles fine

// as index out of bound

// is not checked in C.



#include <iostream>

using namespace std;

int main ()

{

    int arr [2];

    cout << arr[3] << " ";

    cout << arr[-2] << " ";

 return 0;
```

-

}

Output

```
-449684907 4195777
```

- In C, it is not a compiler error to initialize an array with more elements than the specified size. int arr [2] = {1,2,3,4,5,6};
- The program won't compile in C++. If we save the above program as a .cpp, the program generates compiler error *"error: too many initializers for 'int [2]'"*.

- Arr[i]=i[Arr]

Vectors in c++

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted
- Vector is a container object which means that **it holds things of certain type**
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- Vectors can change their size during run time
- In vectors, data is inserted at the end, Removing the last element takes only constant time O (1) because no resizing happens.
- Inserting and erasing at the beginning or in the middle is linear in time O(n) .

Vector in c++ is of LIFO type Functions of vector vector::begin ()

- begin () function is used to return an iterator pointing to the first element of the vector container.
- Bidirectional iterators are **iterators that can be used to access the sequence of elements in a range in both directions** (towards the end and towards the beginning). They are similar to forward iterators, except that they can move in the backward direction also, unlike the forward iterators, which can move only in the forward direction.
- begin () function **returns a bidirectional iterator** to the first element of the container.

***vectorname*. begin ()**

Parameters:

-

No parameters are passed.

Returns:

This function returns a bidirectional iterator

pointing to the first element.

Examples:

Input: my vector {1, 2, 3, 4, 5};

myvector.begin();

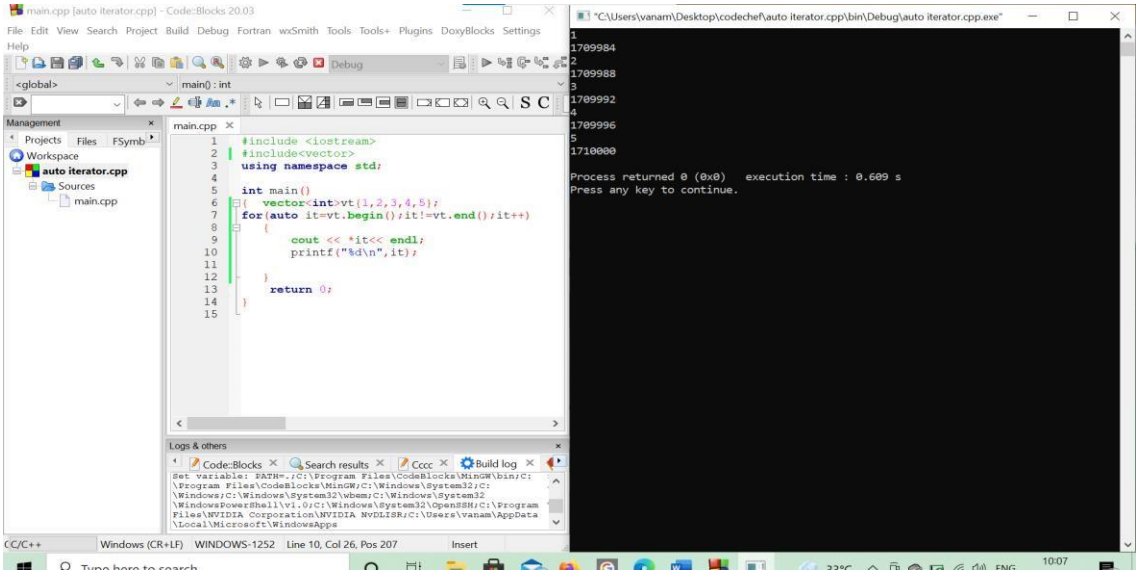
Output : *returns an iterator to the element 1*

Input : myvector{"This", "is", "Geeksforgeeks"};

myvector.begin();

Output : *returns an iterator to the element This*

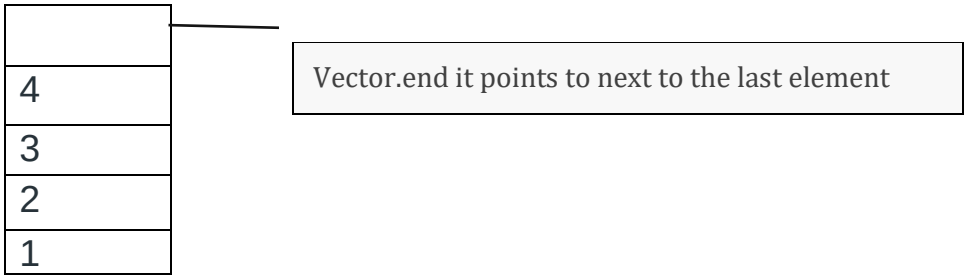
Shows error when a parameter is passed.



- **Time Complexity:** O(1)

vector::end()

- end() function is used to return an iterator pointing to next to last element of the vector container. end() function **returns a bidirectional iterator**.



Syntax :

vectorname.end()
Parameters :
No parameters are passed.
Returns :
This function returns a bidirectional iterator
pointing to next to last element.

Examples:
Input : myvector{1, 2, 3, 4, 5};
myvector.end();
Output : returns an iterator after 5
Input : myvector{"computer", "science", "portal"};
myvector.end();
Output : returns an iterator after portal
Errors and Exceptions
1. It has a no exception throw guarantee.
2. Shows error when a parameter is passed.

```
// INTEGER VECTOR EXAMPLE
// CPP program to illustrate
// Implementation of end() function
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   // declaration of vector container
   vector<int> myvector{ 1, 2, 3, 4, 5 };
```

```
   // using end() to print vector
for (auto it = myvector.begin();
it != myvector.end(); ++it)
cout << ' ' << *it;     return 0;
}
```

Output:

1 2 3 4 5

```
// STRING VECTOR EXAMPLE
// CPP program to illustrate
// Implementation of end() function
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
   // declaration of vector container
vector<string> myvector{ "computer",
                         "science", "portal" };

   // using end() to print vector
for (auto it = myvector.begin();
it != myvector.end(); ++it)
       cout << ' ' << *it;
   return 0;
}
```

Output: computer
science portal

Time Complexity: O(1)

vector::push_back()

push_back() function is used to push elements into a vector from the back. The new value is inserted into the vector at the end, after the current last element and the container size is increased by 1.

Syntax :

vectorname.push_back(value) Parameters
:
The value to be added in the back is
passed as the parameter **Result**
:
Adds the value mentioned as the parameter to
the back of the vector named as vectorname
Examples:

Input : myvector = {1, 2, 3, 4, 5};
myvector.push_back(6);
Output :1, 2, 3, 4, 5, 6

Input : myvector = {5, 4, 3, 2, 1};
myvector.push_back(0);
Output :5, 4, 3, 2, 1, 0

vector::rbegin() is a built-in function in C++ STL which returns a reverse iterator pointing to the last element in the container.
Syntax: vector_name.rbegin()

- **Parameters:** The function does not accept any parameter.
- **Return value**: The function returns a reverse iterator pointing to the last element in the container.
- **vector::rend()** is a built-in function in C++ STL which returns a reverse iterator pointing to the theoretical element right before the first element in the array container.
- **Syntax:** vector_name.rend()
- **Parameters:** The function does not take any parameter.
- **Return value:** The function returns a reverse iterator pointing to the theoretical element right before the first element in the array container.

```
// CPP program to illustrate
// the vector::rend() function #include
<bits/stdc++.h>
using namespace std;
```

```
int main()
{
    vector<int> v;
    v.push_back(11);
    v.push_back(12);
    v.push_back(13);
    v.push_back(14);
    v.push_back(15);

    cout << "The last element is: " << *v.rbegin();

    // prints all the elements
    cout << "\nThe vector elements in reverse order are:\n";
    for (auto it = v.rbegin(); it != v.rend(); it++)
    cout << *it << " ";
    return 0;
}
```

Output:

The vector elements in reverse order are:

```
15 14 13 12 11
```

vector::cbegin()

The function returns a constant iterator which is used to iterate container.

- The iterator points to the beginning of the vector. •

  Iterator cannot modify the contents of the vector.

Main difference between vector.begin() and vector.cbegin is that vector.begins allows to modify the value in the container

Syntax: *vectorname*.cbegin()

Parameters: *There is no parameter*

Return value:
Constant random access iterator points to the beginning of the vector.

Exception: *No exception*

Syntax:

***vectorname*.cend()**
Parameters: *There is no parameter*

Return value:
Constant random access iterator points to ***past-the-end*** *element of the vector.*

Exception:
No exception

```
// CPP program to illustrate

// use of cbegin()

#include <iostream>

#include <string>

#include <vector>


using namespace std;


int main()

{

        vector<string> vec;
```

```
        // 5 string are inserted
vec.push_back("first");
vec.push_back("second");    vec.push_back("third");
vec.push_back("fourth");    vec.push_back("fifth");

        // displaying the contents    cout <<
"Contents of the vector:" << endl;          for
(auto itr = vec.cbegin();
                itr != vec.end();
                ++itr)
                cout << *itr << endl;

        return 0;
}
```

Output:

```
Contents of the vector:
first
second
third fourth
fifth
// C++ program to illustrate the
// iterators in vector #include
<iostream>
```

```
#include <vector>

using namespace std;

int main()
{
	vector<int> g1;

	for (int i = 1; i <= 5; i++)
		g1.push_back(i);

	cout << "Output of begin and end: ";
	for (auto i = g1.begin(); i != g1.end(); ++i)
cout << *i << " ";

	cout << "\output of cbegin and cend: ";
	for (auto i = g1.cbegin(); i != g1.cend(); ++i)
cout << *i << " ";

	cout << "\nOutput of rbegin and rend: ";
	for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
cout << *ir << " ";

	cout << "\nOutput of crbegin and crend : ";
	for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
cout << *ir << " ";

	return 0;
}
```

Output:

Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1 **vector::empty()**

empty() function is used to check if the vector container is empty or not. **Syntax :**

***vectorname*.empty()**

Parameters :

No parameters are passed.

Returns : True, if vector is empty

False, Otherwise

```
// CPP program to illustrate
// Implementation of empty() function
#include <iostream> #include
<vector>
using namespace std;

int main()
{
        vector<int> myvector{};
        if (myvector.empty())
        {
                cout << "True";
        }
        else {
                cout << "False";
        }
        return 0;
}
```

Output: True **vector::size()** size() function is used to return the size of the vector container or the number of elements in the vector container.

Syntax :

***vectorname*.size()**

Parameters :

No parameters are passed.

Returns :

Number of elements in the container.

```
// CPP program to illustrate
```

```
// Implementation of size() function
#include <iostream> #include
<vector>
using namespace std;

int main()
{
        vector<int> myvector{ 1, 2, 3, 4, 5 };
cout << myvector.size();
        return 0;
}

// CPP program to illustrate
// Implementation of size() function
#include <iostream> #include
<vector>
using namespace std;

int main()
{
        vector<int> myvector{ 1, 2, 3, 4, 5 };
        cout << myvector.size();
        return 0;
}
```

Why is empty() preferred over size()

empty() function is often said to be preferred over the size() function due to some of these points-

1. empty() function **does not use any comparison operators**, thus it is more convenient to use
2. empty() function is **implemented in constant time**, regardless of container type, whereas some implementations of size() function require O(n) time complexity such as list::size().

The **vector::max_size()** is a built-in function in C++ STL which returns the maximum number of elements that can be held by the vector container.

vector_name.max_size()

Parameters: The function does not accept any parameters.

Return value: The function returns the maximum numbers that can fit into the vector container.

```
// C++ program to illustrate the
// vector::max_size() function #include
<bits/stdc++.h>
using namespace std;

int main()
{
        // initialize a vector
        vector<int> vec;

        // returns the max_size of vector
        cout << "max_size of vector 1 = " << vec.max_size() << endl;
```

```
        vector<int> vec1;

        // returns the max_size of vector
        cout << "max_size of vector 2 = " << vec1.max_size() << endl;
return 0;
}
```

Output: max_size of vector 1 =
4611686018427387903 max_size of vector 2 =
4611686018427387903

vector::resize()

The function alters the container's content in actual by inserting or deleting the elements from it. It happens so,

- If the given value of n is less than the size at present then extra elements are demolished.
- If n is more than current size of container then upcoming elements are appended at the end of the vector.

Syntax: vectorname.resize(int n,
int val) Parameters:

- **n** – it is new container size, expressed in number of elements.
- **val** – if this parameter is specified then new elements are initialized with this value.

Return value:

- This function do not returns anything.

Exception:

- The only exception if it so happens is Bad_alloc thrown, if reallocation fails.

Below programs illustrate the working of the function

Size of the vector container is lowered.

```
// resizing of the vector
#include <iostream>
#include <vector>

using namespace std;

int main()


   // in the vector
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);
```

```
    vec.push_back(5);




{
    vector<int> vec;

    // 5 elements are inserted
    cout << "Contents of vector before resizing:"
         << endl;

    // displaying the contents of the
// vector before resizing     for (int
i = 0; i < vec.size(); i++)
        cout << vec[i] << " ";

    cout << endl;

    // vector is resized
    vec.resize(4);

    cout << "Contents of vector after resizing:"
         << endl;

    // displaying the contents of the
// vector after resizing     for (int i
= 0; i < vec.size(); i++)
        cout << vec[i] << " ";

    return 0;
}
```

Output:

```
Contents of the vector before resizing:
1 2 3 4 5
Contents of the vector after resizing:
1 2 3 4
```

Size of the vector container is increased.

```
// resizing of the vector


    // in the vector
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);
```

```
   vec.push_back(5);




#include <iostream>
#include <vector>

using namespace std;

int main()
{    vector<int>
vec;

   // 5 elements are inserted
   cout << "Contents of vector before resizing:"
       << endl;

   // displaying the contents of the
// vector before resizing     for (int
i = 0; i < vec.size(); i++)        cout
<< vec[i] << " ";

   cout << endl;

   // vector is resized
   vec.resize(8);

   cout << "Contents of vector after resizing:"
       << endl;

   // displaying the contents of
// the vector after resizing     for
(int i = 0; i < vec.size(); i++)
cout << vec[i] << " ";

   return 0;
}
Output:

Contents of the vector before resizing:
1 2 3 4 5


   // in the vector
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);
```

```
   vec.push_back(5);
```

Contents of the vector after resizing:
1 2 3 4 5 0 0 0

Size of the vector container is increased and new elements are initialized with specified value.

```
// resizing of the vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> vec;

   // 5 elements are inserted
```

```
   // in the vector
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);
```

```
    cout << "Contents of vector before resizing:"
         << endl;

    // displaying the contents of
// the vector before resizing     for
(int i = 0; i < vec.size(); i++)
        cout << vec[i] << " ";

    cout << endl;

    // vector is resized
    vec.resize(12, 9);

    cout << "Contents of vector after resizing:"
         << endl;

    // displaying the contents
// of the vector after resizing
for (int i = 0; i < vec.size(); i++)
        cout << vec[i] << " ";

    return 0;
}
```

Output:

```
Contents of the vector before resizing:
1 2 3 4 5
Contents of the vector after resizing:
1 2 3 4 5 9 9 9 9 9 9 9
```

```
vec.push_back(5);
```