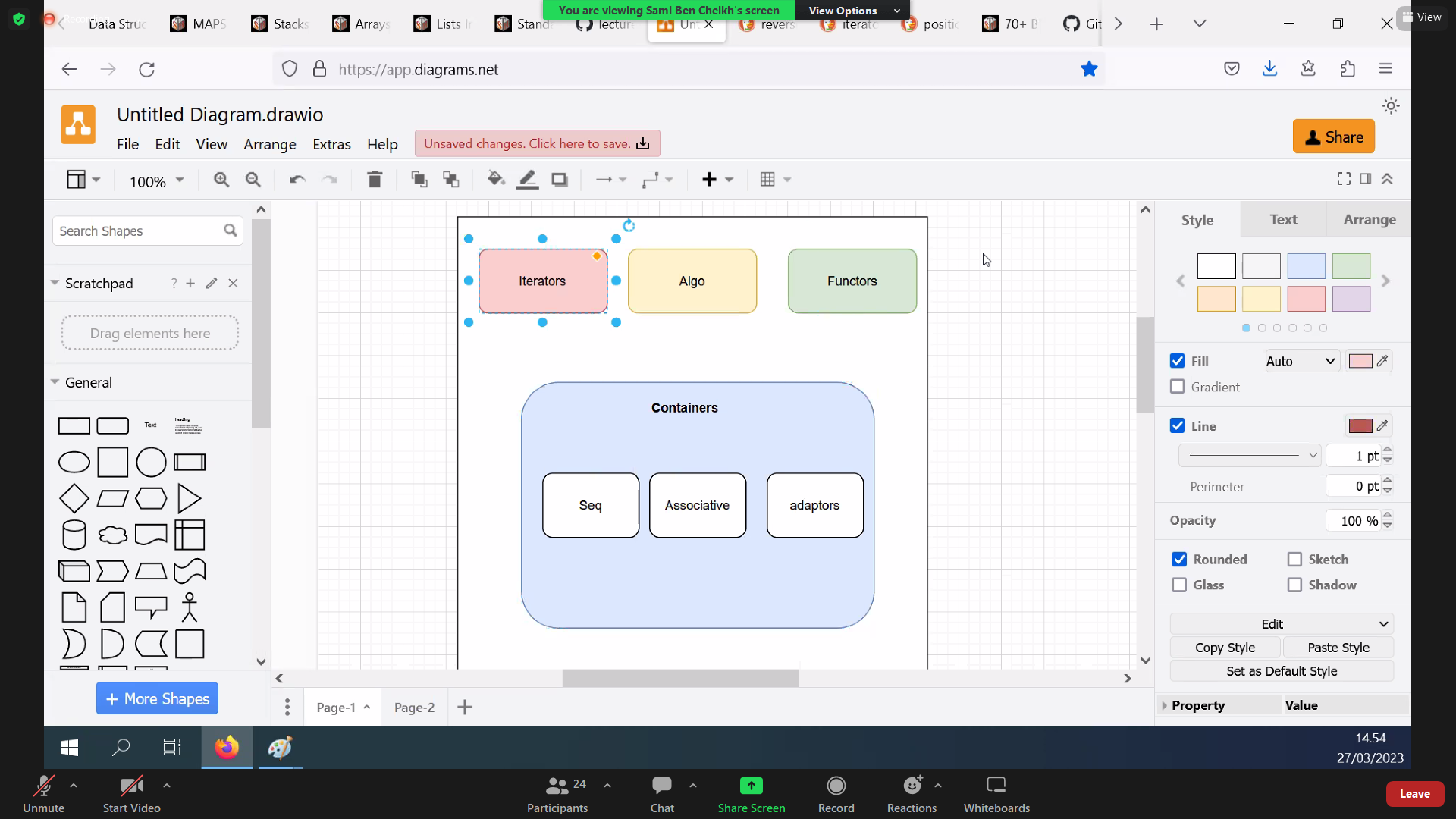
<https://github.com/TT00FE39-3001/lecture8>



<https://ojp.metropolia.fi/lomakkeet/1/lomake.html?code=VFQwMEZFMzktMzAwMQ==>

# Outline

## Topics

* [Review](https://github.com/TT00FE39-3001/lecture8/blob/main/review.md)
* Standard Template Library (STL)
* Asymptotic Notations
* Course Feedback

## This Week in Points

* Group Activities (Max 15 points)
* Peer reviews (Max 10 points)

## Part 1

* Function templates
* std::vector
* [Activity 1](https://github.com/TT00FE39-3001/lecture8/blob/main/activity1)

## Part 2

* Containers
* Iterators
* C++ STL vs Java Collections Framework
* [Activity 2](https://github.com/TT00FE39-3001/lecture8/blob/main/activity2)

## Part 3

* STL algorithms & [Functors](https://www.go4expert.com/articles/cpp-stl-functors-t34696/)
* �, �, Ω, �, �
* Where to go from here?
* [Activity 3](https://github.com/TT00FE39-3001/lecture8/blob/main/activity3)

## Links

* [#](https://github.com/TT00FE39-3001/lecture8/blob/main/links.md)

STL includes a set of template classes that overload the function call operator (operator ()). Instances of those classes are called functors or function objects. Many algorithm functions in STL take those functors as input and apply them on the elements they operate. STL has two kinds of function objects:

1. Unary Functor: Functor that can be called with one arguments
2. Binary Functor: Functor that can be called with two arguments

Among STL functors there is a group of function objects called predicate which take one or two arguments and return boolean value or object convertible to boolean value. The predicates which take one argument are called unary predicates and those who take two arguments are called binary predicates. STL functors are declared in the header <functional> and are part of namespace std. They are divided in following groups according their functionality:

1. [Functors for Arithmetic Operations:](https://www.go4expert.com/articles/cpp-stl-functors-t34696/#arith) They are called for arithmetic operations like addition and subtraction
2. [Functors for Comparison Operations:](https://www.go4expert.com/articles/cpp-stl-functors-t34696/#comparison) They are called for comparing two values like equality or inequality.
3. [Functors for Logical Operations:](https://www.go4expert.com/articles/cpp-stl-functors-t34696/#logical) They are called for logical operation like logical AND
4. [Functors for Bitwise Operations:](https://www.go4expert.com/articles/cpp-stl-functors-t34696/#bitwise) They are called to perform bitwise operations like bitwise AND and bitwise OR.

links

# Links

* MIT 2011: Introduction to Algorithms <https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/pages/syllabus/> <https://www.youtube.com/watch?v=HtSuA80QTyo&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb>
* MIT 2020: Introduction to Algorithms <https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/video_galleries/lecture-videos/> <https://www.youtube.com/playlist?list=PLUl4u3cNGP63EdVPNLG3ToM6LaEUuStEY>
* Standford: [Algorithms Specialization](https://www.coursera.org/specializations/algorithms)
* Princeton: [Algorithms, Part I](https://www.coursera.org/learn/algorithms-part1), [Algorithms, Part II](https://www.coursera.org/learn/algorithms-part2)
* Misc
  + <https://www.enjoyalgorithms.com/data-structures-and-algorithms-course/>
  + <https://kalkicode.com/>

**Review**

**Classification of Algorithm techniques**

* Brute Force
  + Linear search
  + Bubble sort
  + Selection sort
* Decrease and conquer
  + Binary Search
  + Insertion sort
* Divide-and-Conquer
  + Quick Sort
  + Merge Sort
* Dynamic Programming
  + Bottom Up: Tabulation
  + Top Down: Memoization
* Greedy Method

**Data Structures & Abstract data types (ADT)**

* Data Structures (Physical)
  + Arrays
  + Linked Lists
* ADT (Logical)
  + Linear
    - Queues
    - Stacks
    - Hash Tables (linear/non-linear)
  + Non Linear
    - Binary Tree
    - Binary Search Tree
    - Heaps
    - Graphs

**Analysis of Algorithm Efficiency**

* Big O Complexity
* Average case vs worst case
* Space vs Time
* Little-O, Theta, Little Omega, Big Omega

**Standard Template Library (STL)**

* Containers
* Algorithms
* Iterators
* Functors

**Misc**

* FIFO vs LIFO
* Recursion vs Iteration
  + The Top-Down Thought Process
* Logarithms vs Exponential

ACTIVITY 1

**# Activities**

---

> [Course Feedback](https://ojp.metropolia.fi/lomakkeet/1/lomake.html?code=VFQwMEZFMzktMzAwMQ==)

---

**## Task 1**

- What is the difference between function overloading and function templates. You can refer to the programs in the `./src/` folder as well as [Links 2 and 3](#links) below.

- Rewrite the following program using templates

```cpp

#include <iostream>

int add(int x, int y)

{

    return x + y;

}

double add(double x, double y)

{

    return x + y;

}

int main()

{

    std::cout << add(1, 2); // calls add(int, int)

    std::cout << '\n';

    std::cout << add(1.2, 3.4); // calls add(double, double)

    return 0;

}

```

**## Task 2**

Refer to the following link:

<https://www.softwaretestinghelp.com/vectors-in-stl/>

Discuss the difference between

- Size() and capacity()

- begin() and cbegin()

- end() and cend()

**## Links**

1. <https://cpp.sh/>

2. <https://www.learncpp.com/cpp-tutorial/function-templates/>

3. <https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/>

4. <https://www.geeksforgeeks.org/design-and-analysis-of-algorithms/>

ANSWERS:

**## Task 1**

- What is the difference between function overloading and function templates. You can refer to the programs in the `./src/` folder as well as [Links 2 and 3](#links) below.

- Rewrite the following program using templates

Both function overloading and templates are examples of polymorphism.Function overloading is used when multiple functions do similar operations; templates are used when multiple functions do identical operations.Templates provide an advantage when you want to perform the same action on types that can be different.

#include <iostream>

int add(int x, int y)

{

    return x + y;

}

double add(double x, double y)

{

    return x + y;

}

int main()

{

    std::cout << add(1, 2); // calls add(int, int)

    std::cout << '\n';

    std::cout << add(1.2, 3.4); // calls add(double, double)

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity1\src> .\overloading2

3

4.6

#include <iostream>

template <typename T> // this is the template parameter declaration

T add(T x, T y)       // this is the function template definition for add<T>

{

    return x + y;

}

int main()

{

    std::cout << add(1, 2); // calls add(int, int)

    std::cout << '\n';

    std::cout << add(1.2, 3.4); // calls add(double, double)

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity1\src> .\template2

3

4.6

**## Task 2**

Refer to the following link:

<https://www.softwaretestinghelp.com/vectors-in-stl/>

Discuss the difference between

- Size() and capacity()

- begin() and cbegin()

- end() and cend()

Dynamic array will expand on its own as the need arises. STL provides this dynamic array in the form of a vector container.

**Declaring A Vector In C++ With std:: Vector Class**

In STL vector class ‘**std::vector**’ is defined under the header **<vector>**. Thus, in order to use a vector container, we should include this header in our program as shown below:

**#include <vector>**

**We can declare an empty vector as shown below:**

**std::vector<int> myvec;**

The above line of code will create a vector with the elements of the type integer. In memory, this will be laid out as myvec.

**Initialize Vector**

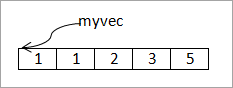
We can initialize a vector with a value at the time of declaring it.

**This is done as follows:**

|  |
| --- |
| #include<vector>  **int** main()  {                   std::vector<**int**> myvec = {1, 1, 2, 3, 5};  } |

In the above code, we declare a vector of type int named myvec containing the first five elements in the Fibonacci sequence.

**The memory layout of this vector will be as below:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/memory-layout.png)

Discuss the difference between

- Size() and capacity()

- begin() and cbegin()

- end() and cend()

**(i) Size of the vector**

The function size() returns the number of elements in the vector container. This is the in-built function of std::vector class and can be used directly to find the size of the vector.

**capacity():** Returns the size of storage space currently allocated. This is returned in terms of the number of elements

#include <iostream>

#include <vector>

using namespace std;

int main()

{

   vector<int> myvec = {1, 1, 2, 3, 5, 8};

   cout << "Vector Size : " << myvec.size();

   cout << " Vector Capacity : " << myvec.capacity();

   return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity1\src> .\vector\_size\_capacity

Vector Size : 6 Vector Capacity : 6

**std::**[**vector**](https://cplusplus.com/reference/vector/vector/)**::capacity**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

size\_type capacity() const;

**Return size of allocated storage capacity**

Returns the size of the storage space currently allocated for the [vector](https://cplusplus.com/vector), expressed in terms of elements.  
  
This *capacity* is not necessarily equal to the [vector size](https://cplusplus.com/vector::size). It can be equal or greater, with the extra space allowing to accommodate for growth without the need to reallocate on each insertion.  
  
Notice that this *capacity* does not suppose a limit on the size of the [vector](https://cplusplus.com/vector). When this *capacity* is exhausted and more is needed, it is automatically expanded by the container (reallocating it storage space). The theoretical limit on the [size](https://cplusplus.com/vector::size) of a [vector](https://cplusplus.com/vector) is given by member [max\_size](https://cplusplus.com/vector::max_size).  
  
The *capacity* of a [vector](https://cplusplus.com/vector) can be explicitly altered by calling member [vector::reserve](https://cplusplus.com/vector::reserve).

* **begin():**Returns iterator pointed to the first element of the vector container.
* **cbegin():**Returns a constant iterator pointing to the first element in the vector container.

**end():**Returns an iterator pointing to the element that follows the last element in the vector

* **cend():**Returns a constant iterator pointing to the element following the last element of the vector container.

#include <iostream>

#include <vector>

using namespace std;

int main()

   {

      vector<int> v1;

      for (int i = 1; i <= 5; i++)

         v1.push\_back(i+1);

         cout << "Output of Vector with begin and end: ";

      for (auto i = v1.begin(); i != v1.end(); ++i)

         cout << \*i << " ";

         cout << "\nOutput of Vector with rbegin and rend: ";

      for (auto itr = v1.rbegin(); itr != v1.rend(); ++itr)

         cout << \*itr << " ";

         cout << "\nOutput Vector of with cbegin and cend: ";

      for (auto itc = v1.cbegin(); itc != v1.cend(); ++itc)

         cout << \*itc << " ";

         cout << "\nOutput Vector of with crbegin and crend : ";

      for (auto icr = v1.crbegin(); icr != v1.crend(); ++icr)

         cout << \*icr << " ";

         return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity1\src> .\beginnings

Output of Vector with begin and end: 2 3 4 5 6

Output of Vector with rbegin and rend: 6 5 4 3 2

Output Vector of with cbegin and cend: 2 3 4 5 6

Output Vector of with crbegin and crend : 6 5 4 3 2

# std::[vector](https://cplusplus.com/reference/vector/vector/)::cbegin

const\_iterator cbegin() const noexcept;

**Return const\_iterator to beginning**

Returns a const\_iterator pointing to the first element in the container.  
  
A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by [vector::begin](https://cplusplus.com/vector::begin), but it cannot be used to modify the contents it points to, even if the [vector](https://cplusplus.com/vector) object is not itself const.  
  
If the container is [empty](https://cplusplus.com/vector::empty), the returned iterator value shall not be dereferenced.

ACTIVITY 2

**# Activities**

**## Task 1**

> Refer to the following links while discussing the answer.

- What is the difference between `array` and `std::array`

<https://stackoverflow.com/questions/30263303/stdarray-vs-array-performance>

- What is the difference between `std::array` and `std::vector`

<https://www.softwaretestinghelp.com/arrays-in-stl/>

- What is the difference between `std::list` and `std::vector`

<https://www.softwaretestinghelp.com/lists-in-stl/>

**## Task 2**

- Run the Stack and Queue examples in the following link

<https://www.softwaretestinghelp.com/stacks-and-queues-in-stl/>

> make sure you correct the syntax e.g `&lt;int&gt; becomes  <int>`

**## Task 3**

- Discuss the different types of iterators present in C++. You can refer to the following link

<https://www.geeksforgeeks.org/introduction-iterators-c/>

- What are the Benefits of Iterators

**## Links**

- <https://cpp.sh/>

ANSWERS:

- What is the difference between `array` and `std::array`

<https://stackoverflow.com/questions/30263303/stdarray-vs-array-performance>

std::array has value semantics while raw arrays do not. This means you can copy std::array and treat it like a primitive value. You can receive them by value or reference as function arguments and you can return them by value.

If you never copy a std::array, then there is no performance difference than a raw array. If you do need to make copies then std::array will do the right thing and should still give equal performance

If you can use std::array you should use it.

- What is the difference between `std::array` and `std::vector`

<https://www.softwaretestinghelp.com/arrays-in-stl/>

# Difference between std::vector and std::array in C++

[C++](https://www.tutorialspoint.com/articles/category/Cplusplus)[Server Side Programming](https://www.tutorialspoint.com/articles/category/Server-Side-Programming)[Programming](https://www.tutorialspoint.com/articles/category/Programming)

The following are the differences between vector and array −

* Vector is a sequential container to store elements and not index based.
* Array stores a fixed-size sequential collection of elements of the same type and it is index based.
* Vector is dynamic in nature so, size increases with insertion of elements.
* As array is fixed size, once initialized can’t be resized.
* Vector occupies more memory.
* Array is memory efficient data structure.
* Vector takes more time in accessing elements.
* Array access elements in constant time irrespective of their location as elements are arranged in a contiguous memory allocation.

Vectors and arrays can be declared with the following syntax −

Vector declaration:vector<datatype>array name;

Array declaration:type array\_name[array\_size];

Vector initialization:vector<datatype>array name={values};

Array initialization:datatype arrayname[arraysize] = {values};

#include <algorithm>

#include <array>

#include <iostream>

#include <iterator>

using namespace std;

int main() {

  array<int, 5> myarray = {1, 1, 2, 3, 5};

  cout << "Size of array: " << endl;

  cout << myarray.size() << endl;

  cout << "\nmyarray contents: ";

  for (auto i : myarray)

    cout << i << ' ';

  // sort operation

 sort(myarray.begin(), myarray.end());

   cout << "\nsorted myarray : ";

   for (auto i : myarray)

      cout << i << ' ';

   cout<<"\nFirst element of myarray "<<myarray.at(0);

   cout<<endl;

   cout<<"FRONT myarray: "<<myarray.front();

   cout<<endl;

   cout<<"BACK myarray: "<<myarray.back();

   cout<<endl;

   // Filling ar2 with 10

   myarray.fill(8);

  cout << "\nFilled myarray : ";

  for (auto i : myarray)

     cout << i << ' ';

return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity2\src> .\array

Size of array:

5

myarray contents: 1 1 2 3 5

sorted myarray : 1 1 2 3 5

First element of myarray 1

FRONT myarray: 1

BACK myarray: 5

Filled myarray : 8 8 8 8 8

## std::vector:

## Example Code

#include <iostream>

#include <vector>

using namespace std;

int main() {

   vector<vector<int>>v{ { 4, 5, 3 }, { 2, 7, 6 }, { 3, 2, 1 ,10 } };

   cout<<"the 2D vector is:"<<endl;

   for (int i = 0; i < v.size(); i++) {

      for (int j = 0; j < v[i].size(); j++)

         cout << v[i][j] << " ";

         cout << endl;

   }

   return 0;

}

## Output

the 2D vector is:

4 5 3

2 7 6

3 2 1 10

## std:: array:

## Example Code

#include<iostream>

#include<array>

using namespace std;

int main() {

   array<int,4>a = {10, 20, 30, 40};

   cout << "The size of array is : ";

   //size of the array using size()

   cout << a.size() << endl;

   //maximum no of elements of the array

   cout << "Maximum number of elements array can hold is : ";

   cout << a.max\_size() << endl;

   // Printing array elements using at()

   cout << "The array elements are (using at()) : ";

   for ( int i=0; i<4; i++)

      cout << a.at(i) << " ";

      cout << endl;

      // Filling array with 1

      a.fill(1);

      // Displaying array after filling

      cout << "Array after filling operation is : ";

   for ( int i=0; i<4; i++)

      cout << a[i] << " ";

      return 0;

}

## Output

The size of array is : 4

Maximum number of elements array can hold is : 4

The array elements are (using at()) : 10 20 30 40

Array after filling operation is : 1 1 1 1

Arrays are contiguous memory locations. Array container is a sequential homogenous container and is of a fixed size.

Actually, in programming, we rarely use such a static container as in real-time scenarios we need containers that can expand or shrink dynamically. Nevertheless, as an array is one of the basic containers, we will begin our discussions about STL containers with arrays.

**The general syntax of declaring an array container is:**

**array<object\_type, size> array\_name;**

The above declaration creates an array container ‘array\_name’ with size ‘size’ and with objects of type ‘object\_type’.

**We can also initialize this array container as shown below,**

**Array<int,5> myarray = {1,1,2,3,5};**

Array container supports various operations that can be carried out to facilitate efficient traversing and manipulation of array container elements.

**Some of the functions that are supported by the array container include:**

* **At:** Returns value in the array container at a given position. ‘Out\_of\_range’ exception is thrown if the position specified is beyond the array limits.
* **Front:** Returns the first element in the array container.
* **Back:** Returns the last element in the array container if the container is completely filled the other returns the rightmost element in the container.
* **Fill:** Assigns a given value to every element in the array container.
* **Swap:** Swaps contents of two arrays with the same type and same size index wise.
* **Empty:** Boolean function to check if an array container is empty or not.
* **Size:** Returns the number of elements in the array container.
* **Max\_size:** Returns the maximum size of the array container.
* **Begin:** Returns the iterator pointing to the beginning of the array container i.e. first element of the array.
* **End:** Returns the iterator pointing to the location next to the last element in the array container.

What is the difference between `std::list` and `std::vector`

<https://www.softwaretestinghelp.com/lists-in-stl/>

Lists are sequential containers. Lists contain elements in non-contiguous locations. We have discussed arrays and vectors in our previous tutorials.

In case of the array and vector containers, as these containers store data in contiguous memory, insert operation in the middle of these containers proves to be very costly as we have to shift the existing elements accordingly to make space for the new element.

### Overview

The list is a container that overcomes this drawback of the array and vector containers. It allows us to insert elements anywhere in the list without causing much of an overhead. But lists are slower than vectors as far as traversal is concerned.

In this tutorial, we will see the implementation of lists in STL along with the various operations of traversal, manipulations and accessing list with examples.

Note that a majority of list operations are similar to those of vectors and hence readers who have already read our tutorial on vectors will not have problems in interpreting list concepts.

### Declaration And Initialization

For implementing list container and using all its benefits, we need to include a header file <list> in our program.

**#include <list>**

**The general declaration for list container is**

**std::list<objectType> listName;**

**For Example, we can declare a list named ‘mylist’ of type int as follows:**

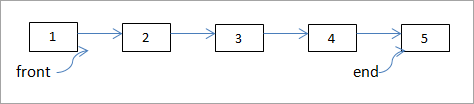
**std::list<int> mylist;**

We can also initialize list at the time of declaration or add elements to it using one of the operations it supports.

**Let’s see how we can initialize the list we created above.**

**std::list<int> mylist = {1, 1, 2, 3, 5};**

**The above initialization will be laid out in memory as shown below:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/Declaration-and-initialization1.png)

Once we initialize the list, we can access the elements of a list using an iterator. The Iterator functions ‘begin’ and ‘end’ help us to traverse through the list elements.

**Note:** Iterator for the list also supports other iterators like reverse iterators (rbegin, rend), constant iterators (cbegin, cend) and constant reverse iterators (crbegin, crend) and can be used in a similar way like vectors.

#include <iostream>

#include <list> // for list operations

using namespace std;

int main()

{

   list<int> mylist = {1,1,2};

   list<int>::iterator it = mylist.begin();

   // iterator to point to 4th position

   advance(it, 3);

   // inserts 3 at 4th position

   mylist.insert(it, 3);

   cout << "The list after inserting"

   << " 1 element using insert() is : ";

   for (list<int>::iterator i = mylist.begin();i != mylist.end();i++)

      cout << \*i << " ";

      cout << endl;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity2\src> .\list

The list after inserting 1 element using insert() is : 1 1 2 3

**## Task 2**

- Run the Stack and Queue examples in the following link

<https://www.softwaretestinghelp.com/stacks-and-queues-in-stl/>

stack:

#include <iostream>

#include <stack>

using namespace std;

void printStack(stack <int> stk)

{

   while (!stk.empty())

      {

         cout << "\t" << stk.top();

         stk.pop();

      }

   cout << "\n";

}

int main ()

{

   stack <int> oddstk;

   oddstk.push(1);

   oddstk.push(3);

   oddstk.push(5);

   oddstk.push(7);

   oddstk.push(9);

   cout << "The stack is : ";

   printStack(oddstk);

   cout << "\nSize of stack: " << oddstk.size();

   cout << "\nTop of stack: " << oddstk.top();

   cout << "\noddstk.pop() : ";

   oddstk.pop();

   printStack(oddstk);

   cout << "\nAnother pop(): ";

   oddstk.pop();

   printStack(oddstk);

   return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity2\src> .\stack

The stack is : 9 7 5 3 1

Size of stack: 5

Top of stack: 9

oddstk.pop() : 7 5 3 1

Another pop(): 5 3 1

queue:

#include <iostream>

#include <queue>

using namespace std;

void printQueue(queue <int> myqueue)

{

   queue <int> secqueue = myqueue;

   while (!secqueue.empty())

   {

      cout << '\t' << secqueue.front();

      secqueue.pop();

   }

   cout << '\n';

}

int main()

   {

      queue <int> myqueue;

      myqueue.push(2);

      myqueue.push(4);

      myqueue.push(6);

      myqueue.push(8);

      cout << "The queue myqueue is : ";

      printQueue(myqueue);

      cout << "\nmyqueue.size() : " << myqueue.size();

      cout << "\nmyqueue.front() : " << myqueue.front();

      cout << "\nmyqueue.back() : " << myqueue.back();

      cout << "\nmyqueue.pop() : ";

      myqueue.pop();

      printQueue(myqueue);

      return 0;

   }

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity2\src> .\queue

The queue myqueue is : 2 4 6 8

myqueue.size() : 4

myqueue.front() : 2

myqueue.back() : 8

myqueue.pop() : 4 6 8

**## Task 3**

- Discuss the different types of iterators present in C++. You can refer to the following link

<https://www.geeksforgeeks.org/introduction-iterators-c/>

- What are the Benefits of Iterators

An **iterator**is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of an iterator is a pointer. A pointer can point to elements in an array and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers. Depending upon the functionality of iterators they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality.

Graphical user interface, text, application, chat or text message

Description automatically generated

Now each one of these iterators are not supported by all the containers in STL, different containers support different iterators, like vectors support [Random-access iterators](https://www.geeksforgeeks.org/random-access-iterators-in-cpp/), while lists support [bidirectional iterators.](https://www.geeksforgeeks.org/bidirectional-iterators-in-cpp/) The whole list is as given below: Table

Description automatically generated **Types of iterators**: Based upon the functionality of the iterators, they can be classified into five major categories:

1. [**Input Iterators**](https://www.geeksforgeeks.org/input-iterators-in-cpp/): They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.
2. [**Output Iterators**](https://www.geeksforgeeks.org/output-iterators-cpp/): Just like [input iterators](https://www.geeksforgeeks.org/input-iterators-in-cpp/), they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.
3. [**Forward Iterator**](https://www.geeksforgeeks.org/forward-iterators-in-cpp/): They are higher in the hierarchy than[input](https://www.geeksforgeeks.org/input-iterators-in-cpp/) and [output iterators](https://www.geeksforgeeks.org/output-iterators-cpp/), and contain all the features present in these two iterators. But, as the name suggests, they also can only move in a forward direction and that too one step at a time.
4. [**Bidirectional Iterators**](https://www.geeksforgeeks.org/bidirectional-iterators-in-cpp/): They have all the features of[forward iterators](https://www.geeksforgeeks.org/forward-iterators-in-cpp/) along with the fact that they overcome the drawback of [forward iterators](https://www.geeksforgeeks.org/forward-iterators-in-cpp/), as they can move in both the directions, that is why their name is bidirectional.
5. [**Random-Access Iterators**](https://www.geeksforgeeks.org/random-access-iterators-in-cpp/): They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers.

Table

Description automatically generated

**Benefits of Iterators**

There are certainly quite a few ways which show that iterators are extremely useful to us and encourage us to use it profoundly. Some of the benefits of using iterators are as listed below:

* **Convenience in programming:**It is better to use iterators to iterate through the contents of containers as if we will not use an iterator and access elements using [ ] operator, then we need to be always worried about the size of the container, whereas with iterators we can simply use member function end() and iterate through the contents without having to keep anything in mind.

// C++ program to demonstrate iterators

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    // Declaring a vector

    vector<int> v = { 1, 2, 3 };

    // Declaring an iterator

    vector<int>::iterator i;

    int j;

    cout << "Without iterators = ";

    // Accessing the elements without using iterators

    for (j = 0; j < 3; ++j)

    {

        cout << v[j] << " ";

    }

    cout << "\nWith iterators = ";

    // Accessing the elements using iterators

    for (i = v.begin(); i != v.end(); ++i)

    {

        cout << \*i << " ";

    }

    // Adding one more element to vector

    v.push\_back(4);

    cout << "\nWithout iterators = ";

    // Accessing the elements without using iterators

    for (j = 0; j < 4; ++j)

    {

        cout << v[j] << " ";

    }

    cout << "\nWith iterators = ";

    // Accessing the elements using iterators

    for (i = v.begin(); i != v.end(); ++i)

    {

        cout << \*i << " ";

    }

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity2\src> .\iterator1

Without iterators = 1 2 3

With iterators = 1 2 3

Without iterators = 1 2 3 4

With iterators = 1 2 3 4

**Explanation:**As can be seen in the above code that without using iterators we need to keep track of the total elements in the container. In the beginning there were only three elements, but after one more element was inserted into it, accordingly the for loop also had to be amended, but using iterators, both the time the for loop remained the same. So, iterator eased our task.

* **Code reusability:**Now consider if we make **v** a list in place of vector in the above program and if we were not using iterators to access the elements and only using [ ] operator, then in that case this way of accessing was of no use for list (as they don’t support [random-access iterators](https://www.geeksforgeeks.org/random-access-iterators-in-cpp/)). However, if we were using iterators for vectors to access the elements, then just changing the vector to list in the declaration of the iterator would have served the purpose, without doing anything else So, iterators support reusability of code, as they can be used to access elements of any container.
* **Dynamic processing of the container:**Iterators provide us the ability to dynamically add or remove elements from the container as and when we want with ease.
* CPP

|  |
| --- |
| // C++ program to demonstrate iterators |

// C++ program to demonstrate iterators

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    // Declaring a vector

    vector<int> v = { 1, 2, 3 };

    // Declaring an iterator

    vector<int>::iterator i;

    int j;

    // Inserting element using iterators

    for (i = v.begin(); i != v.end(); ++i) {

        if (i == v.begin()) {

            i = v.insert(i, 5);

            // inserting 5 at the beginning of v

        }

    }

    // v contains 5 1 2 3

    // Deleting a element using iterators

    for (i = v.begin(); i != v.end(); ++i) {

        if (i == v.begin() + 1) {

            i = v.erase(i);

            // i now points to the element after the

            // deleted element

        }

    }

    // v contains 5 2 3

    // Accessing the elements using iterators

    for (i = v.begin(); i != v.end(); ++i) {

        cout << \*i << " ";

    }

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity2\src> .\iterator2

5 2 3

**Explanation:**As seen in the above code, we can easily and dynamically add and remove elements from the container using iterator, however, doing the same without using them would have been very tedious as it would require shifting the elements every time before insertion and after deletion.

ACTIVITY 3

**# Activities**

**## Task 1**

- What are the advantages of the C++ Standard Template Library (STL):

- What are the disadvantages of STL?

- What are the main components of STL?

> You can refer to the following link: <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>

**## Task 2**

- Refer to `./src/non-manipulative.cpp`

  - Discuss how the program works

  - What are the Non-Manipulating Algorithms used in the program?

- Refer to `./src/manipulative.cpp`

  - Discuss how the program works

  - What are the Manipulating Algorithms used in the program?

> You can refer to the following link: <https://www.geeksforgeeks.org/c-magicians-stl-algorithms/>

**## Task 3**

- Refer to the following article. Reflect on the difference between Big Oh and little oh

  https://www.baeldung.com/cs/big-o-vs-little-o-notation

**## Task 4**

Refer to one of the following articles. Reflect on the differences between Big Oh, Big Omega and Theta.

- <https://jarednielsen.com/big-o-omega-theta/>

- <https://www.codeandgadgets.com/big-oh-big-omega-and-theta-definitions/>

- <https://www.geeksforgeeks.org/difference-between-big-oh-big-omega-and-big-theta/>

ANSWERS:

**## Task 1**

- What are the advantages of the C++ Standard Template Library (STL):

- What are the disadvantages of STL?

- What are the main components of STL?

> You can refer to the following link: <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of [template classes](https://www.geeksforgeeks.org/templates-cpp/) is a prerequisite for working with STL.

The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.

One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type.

The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.

### Some of the key components of the STL include:

1. Containers: The STL provides a range of containers, such as vector, list, map, set, and stack, which can be used to store and manipulate data.
2. Algorithms: The STL provides a range of algorithms, such as sort, find, and binary\_search, which can be used to manipulate data stored in containers.
3. Iterators: Iterators are objects that provide a way to traverse the elements of a container. The STL provides a range of iterators, such as forward\_iterator, bidirectional\_iterator, and random\_access\_iterator, that can be used with different types of containers.
4. Function Objects: Function objects, also known as functors, are objects that can be used as function arguments to algorithms. They provide a way to pass a function to an algorithm, allowing you to customize its behavior.
5. Adapters: Adapters are components that modify the behavior of other components in the STL. For example, the reverse\_iterator adapter can be used to reverse the order of elements in a container.

By using the STL, you can simplify your code, reduce the likelihood of errors, and improve the performance of your programs.

**STL has 4 components:**

* **Algorithms**
* **Containers**
* **Functors**
* **Iterators**

### 1. Algorithms

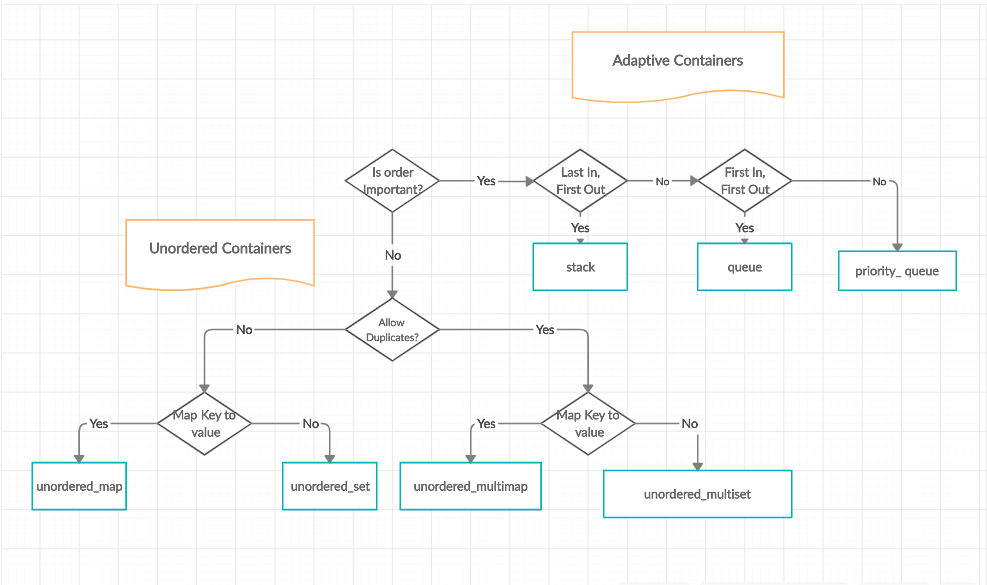
The header algorithm defines a collection of functions specially designed to be used on a range of elements. They act on containers and provide means for various operations for the contents of the containers.

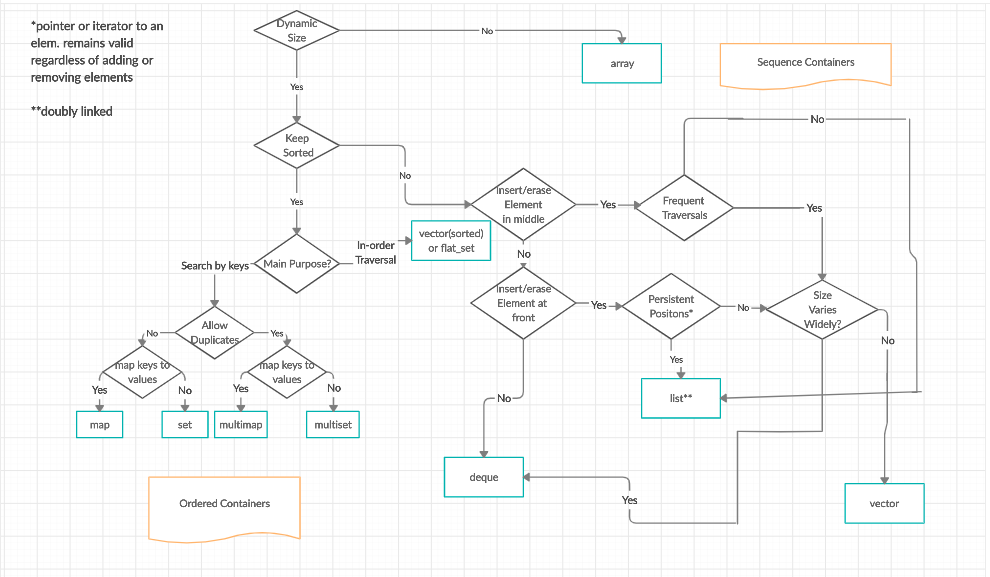
* Algorithm
  + [Sorting](https://www.geeksforgeeks.org/sort-algorithms-the-c-standard-template-library-stl/)
  + [Searching](https://www.geeksforgeeks.org/binary-search-algorithms-the-c-standard-template-library-stl/)
  + [Important STL Algorithms](https://www.geeksforgeeks.org/c-magicians-stl-algorithms/)
  + [Useful Array algorithms](https://www.geeksforgeeks.org/useful-array-algorithms-in-c-stl/)
  + [Partition Operations](https://www.geeksforgeeks.org/stdpartition-in-c-stl/)
* Numeric
  + [valarray class](https://www.geeksforgeeks.org/std-valarray-class-c/)

### 2. Containers

[Containers or container classes](https://www.geeksforgeeks.org/containers-cpp-stl/) store objects and data. There are in total seven standards “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

* Sequence Containers: implement data structures that can be accessed in a sequential manner.
  + [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/)
  + [list](https://www.geeksforgeeks.org/list-cpp-stl/)
  + [deque](https://www.geeksforgeeks.org/deque-cpp-stl/)
  + [arrays](https://www.geeksforgeeks.org/array-class-c/)
  + [forward\_list](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)( Introduced in C++11)
* Container Adaptors: provide a different interface for sequential containers.
  + [queue](https://www.geeksforgeeks.org/queue-cpp-stl/)
  + [priority\_queue](https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/)
  + [stack](https://www.geeksforgeeks.org/stack-in-cpp-stl/)
* Associative Containers: implement sorted data structures that can be quickly searched (O(log n) complexity).
  + [set](https://www.geeksforgeeks.org/set-in-cpp-stl/)
  + [multiset](https://www.geeksforgeeks.org/multiset-in-cpp-stl/)
  + [map](https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)
  + [multimap](https://www.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)
* Unordered Associative Containers: implement unordered data structures that can be quickly searched
  + [unordered\_set](https://www.geeksforgeeks.org/unordered_set-in-cpp-stl/) (Introduced in C++11)
  + [unordered\_multiset](https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/) (Introduced in C++11)
  + [unordered\_map](https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/) (Introduced in C++11)
  + [unordered\_multimap](https://www.geeksforgeeks.org/unordered_multimap-and-its-application/) (Introduced in C++11)





### **3. Functors**

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed. **Must Read** – [Functors](https://www.geeksforgeeks.org/functors-in-cpp/)

### 4. **Iterators**

As the name suggests, iterators are used for working on a sequence of values. They are the major feature that allows generality in STL. **Must Read**– [Iterators](https://www.geeksforgeeks.org/iterators-c-stl/)

## ****Utility Library****

Defined in header <utility>. **Must Read** – [Pair in C++ STL](https://www.geeksforgeeks.org/pair-in-cpp-stl/)

**For more details, refer to the**[**Recent Articles on STL!**](https://www.geeksforgeeks.org/tag/stl/)

### Advantages of the C++ Standard Template Library (STL):

1. Reusability: One of the key advantages of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This can lead to more efficient and maintainable code.
2. Efficient algorithms: Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.
3. Improved code readability: The STL provides a consistent and well-documented way of working with data, which can make your code easier to understand and maintain.
4. Large community of users: The STL is widely used, which means that there is a large community of developers who can provide support and resources, such as tutorials and forums.

### Disadvantages of the C++ Standard Template Library (STL):

1. Learning curve: The STL can be difficult to learn, especially for beginners, due to its complex syntax and use of advanced features like iterators and function objects.
2. Lack of control: When using the STL, you have to rely on the implementation provided by the library, which can limit your control over certain aspects of your code.
3. Performance: In some cases, using the STL can result in slower execution times compared to custom code, especially when dealing with small amounts of data.

**## Task 2**

- Refer to `./src/non-manipulative.cpp`

  - Discuss how the program works

  - What are the Non-Manipulating Algorithms used in the program?

- Refer to `./src/manipulative.cpp`

  - Discuss how the program works

  - What are the Manipulating Algorithms used in the program?

> You can refer to the following link: <https://www.geeksforgeeks.org/c-magicians-stl-algorithms/>

ANSWERS:

NON-MANIPULATIVE.CPP

// A C++ program to demonstrate working of sort(),

// reverse()

#include <algorithm>

#include <iostream>

#include <vector>

#include <numeric> //For accumulate operation

using namespace std;

int main()

{

    // Initializing vector with array values

    int arr[] = {10, 20, 5, 23, 42, 15};

    int n = sizeof(arr) / sizeof(arr[0]);

    vector<int> vect(arr, arr + n);

    cout << "Vector is: ";

    for (int i = 0; i < n; i++)

        cout << vect[i] << " ";

    // Sorting the Vector in Ascending order

    sort(vect.begin(), vect.end());

    cout << "\nVector after sorting is: ";

    for (int i = 0; i < n; i++)

        cout << vect[i] << " ";

    // Sorting the Vector in Descending order

    sort(vect.begin(), vect.end(), greater<int>());

    cout << "\nVector after sorting in Descending order is: ";

    for (int i = 0; i < n; i++)

        cout << vect[i] << " ";

    // Reversing the Vector (descending to ascending , ascending to descending)

    reverse(vect.begin(), vect.end());

    cout << "\nVector after reversing is: ";

    for (int i = 0; i < n; i++)

        cout << vect[i] << " ";

    cout << "\nMaximum element of vector is: ";

    cout << \*max\_element(vect.begin(), vect.end());

    cout << "\nMinimum element of vector is: ";

    cout << \*min\_element(vect.begin(), vect.end());

    // Starting the summation from 0

    cout << "\nThe summation of vector elements is: ";

    cout << accumulate(vect.begin(), vect.end(), 0);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity3\src> .\non-manipulative

Vector is: 10 20 5 23 42 15

Vector after sorting is: 5 10 15 20 23 42

Vector after sorting in Descending order is: 42 23 20 15 10 5

Vector after reversing is: 5 10 15 20 23 42

Maximum element of vector is: 42

Minimum element of vector is: 5

The summation of vector elements is: 115

Here

    // Sorting the Vector in Ascending order

    sort(vect.begin(), vect.end());

    // Sorting the Vector in Descending order

    sort(vect.begin(), vect.end(), greater<int>());

reverse(vect.begin(), vect.end());

\*max\_element(vect.begin(), vect.end());

\*min\_element(vect.begin(), vect.end());

accumulate(vect.begin(), vect.end(), 0);

1. [**sort**](https://www.geeksforgeeks.org/sort-c-stl/)**(first\_iterator, last\_iterator)** – To sort the given vector.
2. **sort(first\_iterator, last\_iterator, greater<int>())** – To sort the given container/vector in descending order
3. **reverse(first\_iterator, last\_iterator)** – To reverse a vector. ( if ascending -> descending  OR  if descending -> ascending)
4. **\*max\_element (first\_iterator, last\_iterator)** – To find the maximum element of a vector.
5. **\*min\_element (first\_iterator, last\_iterator)**– To find the minimum element of a vector.
6. **accumulate(first\_iterator, last\_iterator, initial value of sum)** – Does the summation of vector element

There are also other non-manipulative algorithms:

**count(first\_iterator, last\_iterator,x)** – To count the occurrences of x in vector

**find(first\_iterator, last\_iterator, x)** – Returns an iterator to the first occurrence of x in vector and points to last address of vector ((name\_of\_vector).end()) if element is not present in vector.

**[binary\_search](https://www.geeksforgeeks.org/binary-search-algorithms-the-c-standard-template-library-stl/)(first\_iterator, last\_iterator, x)** – Tests whether x exists in sorted vector or not.

**lower\_bound(first\_iterator, last\_iterator, x)** – returns an iterator pointing to the first element in the range [first,last) which         has a value not less than ‘x’.

**upper\_bound(first\_iterator, last\_iterator, x)** – returns an iterator pointing to the first element in the range [first,last)                  which has a value greater than ‘x

MANIPULATIVE.CPP

// C++ program to demonstrate working

// of erase

#include <algorithm>

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    // Initializing vector with array values

    int arr[] = {5, 10, 15, 20, 20, 23, 42, 45};

    int n = sizeof(arr) / sizeof(arr[0]);

    vector<int> vect(arr, arr + n);

    cout << "Given Vector is:\n";

    for (int i = 0; i < n; i++)

        cout << vect[i] << " ";

    vect.erase(find(vect.begin(), vect.end(), 10));

    cout << "\nVector after erasing element:\n";

    for (int i = 0; i < vect.size(); i++)

        cout << vect[i] << " ";

    vect.erase(unique(vect.begin(), vect.end()),

               vect.end());

    cout << "\nVector after removing duplicates:\n";

    for (int i = 0; i < vect.size(); i++)

        cout << vect[i] << " ";

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture8-main\activity3\src> .\manipulative

Given Vector is:

5 10 15 20 20 23 42 45

Vector after erasing element:

5 15 20 20 23 42 45

Vector after removing duplicates:

5 15 20 23 42 45

1. **arr.erase(position to be deleted)** – This erases selected element in vector and shifts and resizes the vector elements accordingly.
2. **arr.erase(unique(arr.begin(),arr.end()),arr.end())** – This erases the duplicate occurrences in sorted vector in a single line

**3. next\_permutation(first\_iterator, last\_iterator)**– This modified the vector to its next permutation.

**4. prev\_permutation(first\_iterator, last\_iterator)**– This modified the vector to its previous permutation.

**## Task 3**

- Refer to the following article. Reflect on the difference between Big Oh and little oh

<https://www.baeldung.com/cs/big-o-vs-little-o-notation>

In this brief tutorial, we’ll learn about **how big-O and little-o notations differ**. In short, they are both asymptotic notations that specify upper-bounds for functions and running times of algorithms.

However, the difference is that **big-O may be asymptotically tight** while **little-o makes sure that the upper bound isn’t asymptotically tight**.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, Word

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**## Task 4**

Refer to one of the following articles. Reflect on the differences between Big Oh, Big Omega and Theta.

- <https://jarednielsen.com/big-o-omega-theta/>

- <https://www.codeandgadgets.com/big-oh-big-omega-and-theta-definitions/>

- <https://www.geeksforgeeks.org/difference-between-big-oh-big-omega-and-big-theta/>

# Difference between Big Oh, Big Omega and Big Theta

**1. Big oh notation (O):**

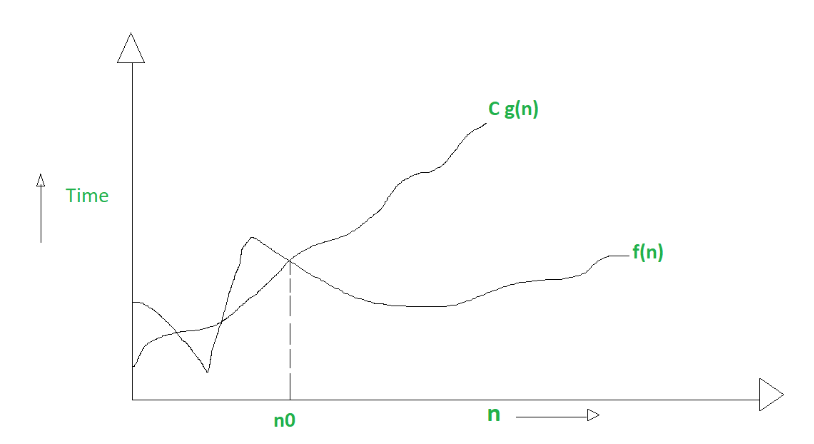
It is define as upper bound and upper bound on an algorithm is the most amount of time required ( the worst case performance).  
**Big oh notation** is used to describe **asymptotic upper bound**.

Mathematically, if**f(n)** describes the running time of an algorithm; **f(n)** is **O(g(n))** if there exist positive constant **C** and **n0** such that,

*0 <= f(n) <= Cg(n) for all n >= n0*

**n** = used to give upper bound an a function.   
If a function is **O(n)**, it is automatically **O(n-square)** as well.

Graphic example for**Big oh (O) :**

**

**2. Big Omega notation (Ω) :**

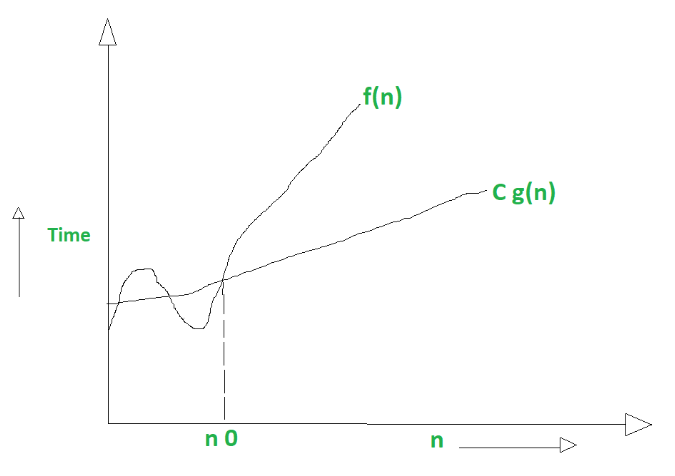
It is define as lower bound and lower bound on an algorithm is the least amount of time required ( the most efficient way possible, in other words best case).  
Just like **O** **notation** provide an **asymptotic upper bound**, **Ω** **notation** provides **asymptotic lower bound**. 

Let **f(n)** define running time of an algorithm;  
**f(n)** is said to be**Ω(g (n))** if there exists positive constant **C** and **(n0)** such that

*0 <= Cg(n) <= f(n) for all n >= n0*

**n** = used to given lower bound on a function   
If a function is **Ω(n-square)** it is automatically **Ω(n)** as well.

Graphical example for **Big Omega (Ω):**

**

**3. Big Theta notation (Θ) :**

It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.

Let **f(n)** define running time of an algorithm.   
**f(n)** is said to be **Θ(g(n))** if **f(n)** is **O(g(n))** and**f(n)** is **Ω(g(n)).**

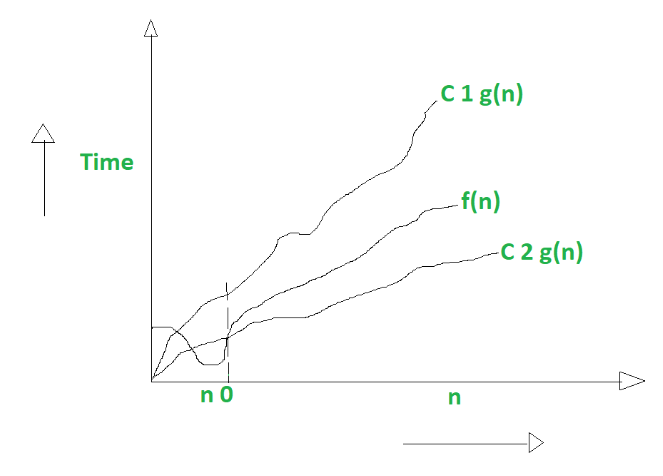
***Mathematically,***

*0 <= f(n) <= C1g(n) for n >= n0  
0 <= C2g(n) <= f(n) for n >= n0*

***Merging both the equation, we get :***

*0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0*

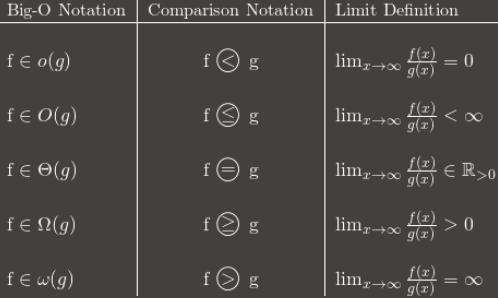
**The equation simply means there exist positive constants C1 and C2 such that f(n) is sandwich between C2 g(n) and C1g(n).**  
   
 Graphic example of **Big Theta (Θ)**:

**

**Difference Between Big oh, Big Omega and Big Theta :** 

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Big Oh (O)** | **Big Omega (Ω)** | **Big Theta (Θ)** |
| **1.** | **It is like (<=)  rate of growth of an algorithm is less than or equal to a specific value.** | **It is like (>=)  rate of growth is greater than or equal to a specified value.** | **It is like (==)  meaning the rate of growth is equal to a specified value.** |
| **2.** | **The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. Asymptotic upper bound is  given by Big O notation.** | **The algorithm’s lower bound is represented by Omega notation. The asymptotic lower bound is given by Omega notation.** | **The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.** |
| **3.** | **Big oh (O) – Upper Bound** | **Big Omega (Ω) – Lower Bound** | **Big Theta (Θ) – Tight Bound** |
| **4.** | **It is define as upper bound and upper bound on an algorithm is the most amount of time required ( the worst case performance).** | **It is define as lower bound and lower bound on an algorithm is the least amount of time required ( the most efficient way possible, in other words best case).** | **It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.** |
| **5.** | **Mathematically: Big Oh is 0 <= f(n) <= Cg(n) for all n >= n0** | **Mathematically: Big Omega is 0 <= Cg(n) <= f(n) for all n >= n0** | **Mathematically – Big Theta is 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0** |

For more details, please refer: [Design and Analysis of Algorithms](https://www.geeksforgeeks.org/design-and-analysis-of-algorithms/).



Diagram, venn diagram

Description automatically generated