# Notes

|  |
| --- |
|  |
| **#pragma startup** and **#pragma exit**: |
| **get\_temporary\_buffer, return\_temporary\_buffer**   * This function take a size n and return the largest available buffer up to size n which can be fit into physical memory. * This function is used to get a memory of temporary nature mostly used for the operation of an algorithm as some algorithms required extra space to perform correctly .   Once the the memory block which is assigned is not needed anymore it shall be released by calling return\_temporary\_buffer |
| *ptrdiff\_t* |
| * stream\_object.rdbuf(): Returns pointer to the stream buffer of stream\_object * stream\_object.rdbuf(streambuf \* p): Sets the stream buffer to the object pointed by p   // discards the input buffer   * cin.ignore(numeric\_limits<streamsize>::max(),'\n');   OR   * cin.sync() |
| Set a variable without using Arithmetic, Relational or Conditional Operator   * x = ((1 - c) \* a) + (c \* b)   OR   * x = (a + b) - (!c \* b) - (c \* a);   OR   * x = (a \* !c) | (b \* c); |
| calculate size of a type   * #define my\_sizeof(type) ((char \*)(&type+1)-(char\*)(&type)) * int size = \*(&arr + 1) - arr |
| * Swap() : for change 2 element of array or vector or… * strchr() : This function Returns a pointer to the last occurrence of a character in a string * isspace(): In C++, isspace is a predefined function used for string and character handling.cstring is the header file required for string functions and cctype is the headerfile required for character functions * Strstr(): find string in other string * lexicographical\_compare: Comparing strings can be generally used in ****dictionary****, where we need to place words in lexicographical order. Example of this can be to find the word which occurs 1st in dictionary among given set of words. * set\_symmetric\_difference: The symmetric difference of two sets is formed by the elements that are present in one of the sets, but not in the other : First array contains : 5 10 15 20 25, Second array contains : 50 40 30 20 10, Output: 5 15 25 30 40 50 * strpbrk() : first match character: char s1[] = "geeksforgeeks"; char s3[] = "kite"; t = strpbrk(s1, s3);=> t= e; * strcoll() : compare two string => less than zero: str1 is less than str2, zero: str1 is equal to str2, greater than zero: str1 is greater than str2 |
| int nums[2][3] = { {16, 18, 20}, {25, 26, 27} }; => \*(\*(nums + 1) + 2) = nums[1][2] = 27; |
| sqrt(-2) = -nan  float b = sqrt(-2); b==b ? “it’s real” : “it’s NaN”; OR use isnan(b)? “” : “”; |
| Note: Declaring a [friend function](http://quiz.geeksforgeeks.org/friend-class-function-cpp/) is a way to give private access to a non-member function. |
| Local class Notes:   * + 1. A local class type name can only be used in the enclosing function     2. All the methods of Local classes must be defined inside the class only     3. A Local class cannot contain static data members. It may contain static functions though.     4. Member methods of local class can only access static and enum variables of the enclosing func.     5. Local classes can access global types, variables and functions. |
| [Copy elision](http://en.wikipedia.org/wiki/Copy_elision) (or Copy omission) is a compiler optimization technique that avoids unnecessary copying of objects |
| Compiler doesn’t create a default constructor if we write any constructor even if it is copy constructor |
| Operators that can be overloaded:  + - \* / % ^  & | ~ !, =  = ++ --  <> == != && ||  += -= /= %= ^= &=  |= \*= <>= [] ()  -> ->\* [new](https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/) new [] delete delete [] |
| Operators that cannot be overloaded:  1> [Scope Resolution Operator](https://www.geeksforgeeks.org/scope-resolution-operator-in-c/) (::)  2> Pointer-to-member Operator (.\*)  3> Member Access or Dot operator (.)  4> [Ternary or Conditional Operator](https://www.geeksforgeeks.org/cc-ternary-operator-some-interesting-observations/) (?:)  5> Object size Operator ([sizeof](https://www.geeksforgeeks.org/sizeof-operator-c/))  6> Object type Operator (typeid) |
| In C++, like other functions, assignement operator function is inherited in derived class. |
| In C++, a static member function of a class cannot be virtual.  Also, static member function cannot be const and volatile |
| in C++, RTTI (Run-time type information) is a mechanism that exposes information about an object’s data type at runtime and is available only for the classes which have at least one virtual function |
| In C++, virtual functions can be private and can be overridden by the derived class. |
| Inline functions are very useful when small functions are frequently used and called in a program many times |
| Pure Virtual Functions and Abstract Classes:   1. A class is abstract if it has at least one pure virtual function. 2. We can have pointers and references of abstract class type. 3. If we do not override the pure virtual function in derived class, then derived class also becomes abstract class. 4. An abstract class can have constructors. |
| **Can a destructor be pure virtual in C++?** Yes, it is possible to have pure virtual destructor. |
| *make\_heap change a vector to a heap* |
| asinh() : is an inbuilt function in C++ STL which returns the inverse hyperbolic sine of an angle given in radians. |
| atanh(): is an inbuilt function in C++ STL which returns the inverse hyperbolic tangent of an angle given in radians. |
| Inherit a Dummy class with a private copy constructor and a private copy assignment operator. private:      Dummy(const Dummy& temp\_obj)  {   }      Dummy& operator=(const Dummy& temp\_obj)   {   }  Using Deleted copy constructor and copy assignment operator:  Base(const Base& temp\_obj) = delete;      Base& operator=(const Base& temp\_obj) = delete; |
| accumulate(v.begin(), v.end(), initial\_sum); SUM Elements of vector or array or … accumulate(first, last, sum);  accumulate(first, last, sum, myfun); |
| **partial\_sum( )** y0 = x0  y1 = x0 + x1 , y2 = x0 + x1 + x2 , y3 = x0 + x1 + x2 + x3 |
| C qsort() vs C++ sort() . sort is very faster. |
| Can we access global variable if there is a local variable with same name? Yes,with ::X for global variable  Swap two variables in one line (x ^= y), (y ^= x), (x ^= y); b = (a + b) – (a = b);  a += b – (b = a); |
| RAII: Resource Acquisition Is Initialization |

# STL:

## Iterators in C++ STL

### Iterators are used to point at the memory addresses of [STL](http://quiz.geeksforgeeks.org/the-c-standard-template-library-stl/) containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.

### **advance()** : This function is used to **increment the iterator position** till the specified number mentioned in its arguments.

// Using advance() to increment iterator position

// points to 4

advance(ptr, 3);

### **next()** :- This function **returns the new iterator** that the iterator would point after **advancing the positions** mentioned in its arguments.

 // Using next() to return new iterator

 // points to 4

 auto it = next(ptr, 3);

### **prev()** :- This function **returns the new iterator** that the iterator would point **after decrementing the positions** mentioned in its arguments.

// Using prev() to return new iterator

// points to 3

auto it1 = prev(ftr, 3);

### **inserter() :- This function is used to insert the elements at any position in the container. It accepts 2 arguments, the container and iterator to position where the elements have to be inserted.**

 vector<int> ar = { 1, 2, 3, 4, 5 };

 vector<int> ar1 = {10, 20, 30};

 vector<int>::iterator ptr = ar.begin();

 advance(ptr, 3);

copy(ar1.begin(), ar1.end(), inserter(ar,ptr));

## Pair in C++ Standard Template Library (STL)

### The pair container is a simple container defined in **<utility>** header consisting of two data elements or objects.

## Queue in Standard Template Library (STL)

### Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

### empty() – Returns whether the queue is empty

### size() – Returns the size of the queue

### front() – Returns a reference to the first element of the queue

### back() – Returns a reference to the last element of the queue

### push(g) – Adds the element ‘g’ at the end of the queue

### pop() – Deletes the first element of the queue

## Priority Queue in C++ Standard Template Library (STL)

### Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non decreasing order(hence we can see that each element of the queue has a priority{fixed order}).

### empty() – Returns whether the queue is empty

### size() – Returns the size of the queue

### top() – Returns a reference to the top most element of the queue

### push(g) – Adds the element ‘g’ at the end of the queue

### pop() – Deletes the first element of the queue

## Stack in C++ Standard Template Library (STL)

### Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

### empty() – Returns whether the stack is empty

### size() – Returns the size of the stack

### top() – Returns a reference to the top most element of the stack

### push(g) – Adds the element ‘g’ at the top of the stack

### pop() – Deletes the top most element of the stack

## Set in C++ Standard Template Library (STL)

### Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

### begin() – Returns an iterator to the first element in the set

### end() – Returns an iterator to the theoretical element that follows last element in the set

### size() – Returns the number of elements in the set

### max\_size() – Returns the maximum number of elements that the set can hold

### empty() – Returns whether the set is empty

### pair <iterator, bool> insert(const g) – Adds a new element ‘g’ to the set iterator insert (iterator position, const g) – Adds a new element ‘g’ at the position pointed by iterator erase(iterator position) – Removes the element at the position pointed by the iterator

### erase(const g)- Removes the value ‘g’ from the set

### clear() – Removes all the elements from the set

### key\_comp() / value\_comp() – Returns the object that determines how the elements in the set are ordered (‘<‘ by default)

### find(const g) – Returns an iterator to the element ‘g’ in the set if found, else returns the iterator to end

### count(const g) – Returns 1 or 0 based on the element ‘g’ is present in the set or not.

### lower\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the set

### upper\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will go after the element ‘g’ in the set

## Multiset in C++ Standard Template Library (STL)

### Multisets are a type of associative containers similar to set, with an exception that multiple elements can have same values.

### begin() – Returns an iterator to the first element in the multiset

### end() – Returns an iterator to the theoretical element that follows last element in the multiset

### size() – Returns the number of elements in the multiset

### max\_size() – Returns the maximum number of elements that the multiset can hold

### empty() – Returns whether the multiset is empty

### pair insert(const g) – Adds a new element ‘g’ to the multiset

### iterator insert (iterator position,const g) – Adds a new element ‘g’ at the position pointed by iterator

### erase(iterator position) – Removes the element at the position pointed by the iterator

### erase(const g)- Removes the value ‘g’ from the multiset

### clear() – Removes all the elements from the multiset

### key\_comp() / value\_comp() – Returns the object that determines how the elements in the multiset are ordered (‘<' by default)

### find(const g) – Returns an iterator to the element ‘g’ in the multiset if found, else returns the iterator to end

### count(const g) – Returns the number of matches to element ‘g’ in the multiset

### lower\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the multiset

### upper\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will go after the element ‘g’ in the multiset

## Map in C++ Standard Template Library (STL)

### Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

### begin() – Returns an iterator to the first element in the map

### end() – Returns an iterator to the theoretical element that follows last element in the map

### size() – Returns the number of elements in the map

### max\_size() – Returns the maximum number of elements that the map can hold

### empty() – Returns whether the map is empty

### pair insert(keyvalue,mapvalue) – Adds a new element to the map

### erase(iterator position) – Removes the element at the position pointed by the iterator

### erase(const g)- Removes the key value ‘g’ from the map

### clear() – Removes all the elements from the map

### key\_comp() / value\_comp() – Returns the object that determines how the elements in the map are ordered (‘<' by default)

### find(const g) – Returns an iterator to the element with key value ‘g’ in the map if found, else returns the iterator to end

### count(const g) – Returns the number of matches to element with key value ‘g’ in the map

### lower\_bound(const g) – Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the map

### upper\_bound(const g) – Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the map

## Multimap in C++ Standard Template Library (STL)

### Multimap is similar to mapwith an addition that multiple elements can have same keys. Rather than each element being unique, the key value and mapped value pair has to be unique in this case.

### begin() – Returns an iterator to the first element in the multimap

### end() – Returns an iterator to the theoretical element that follows last element in the multimap

### size() – Returns the number of elements in the multimap

### max\_size() – Returns the maximum number of elements that the multimap can hold

### empty() – Returns whether the multimap is empty

### pair<int,int> insert(keyvalue,multimapvalue) – Adds a new element to the multimap

### erase(iterator position) – Removes the element at the position pointed by the iterator

### erase(const g)– Removes the key value ‘g’ from the multimap

### clear() – Removes all the elements from the multimap

### key\_comp() / value\_comp() – Returns the object that determines how the elements in the multimap are ordered (‘<‘ by default)

### find(const g) – Returns an iterator to the element with key value ‘g’ in the multimap if found, else returns the iterator to end

### count(const g) – Returns the number of matches to element with key value ‘g’ in the multimap

### lower\_bound(const g) – Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the multimap

### upper\_bound(const g) – Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the multimap

## Deque in C++ Standard Template Library (STL)

### Double ended queues are sequence containers with the feature of expansion and contraction on both the ends. They are similar to vectors, but are more efficient in case of insertion and deletion of elements at the end, and also the beginning. Unlike vectors, contiguous storage allocation may not be guaranteed.

## Array class in C++

### Array classes knows its own size, whereas C-style arrays lack this property. So when passing to functions, we don’t need to pass size of Array as a separate parameter.

### With C-style array there is more risk of [array being decayed into a pointer](https://www.geeksforgeeks.org/what-is-array-decay-in-c-how-can-it-be-prevented/). Array classes don’t decay into pointers

### Array classes are generally more efficient, light-weight and reliable than C-style arrays

**Operations on array**:

### **at()** :- This function is used to access the elements of array.

### **get()** :- This function is also used to access the elements of array. This function is not the member of array class but overloaded function from class tuple.

### **operator[]** :- This is similar to C-style arrays. This method is also used to access array elements.

### **front()** :- This returns the first element of array.

### **back()** :- This returns the last element of array.

### **size()** :- It returns the number of elements in array. This is a property that C-style arrays lack.

### **max\_size()** :- It returns the maximum number of elements array can hold i.e, the size with which array is declared. The size() and max\_size() return the same value.

### **swap()** : The swap() swaps all elements of one array with other.

### **empty()** :- This function returns true when the array size is zero else returns false.

### **fill()** :- This function is used to fill the entire array with a particular value.

## Forward List in C++ | Set 1 (Introduction and Important Functions)

### Forward list in STL implements singly linked list. Introduced from C++11, forward list are useful than other containers in insertion, removal and moving operations (like sort) and allows time constant insertion and removal of elements.

### Forward List is preferred over list when only forward traversal is required (same as singly linked list is preferred over doubly linked list) as we can save space. Some example cases are, chaining in hashing, adjacency list representation of graph, etc.

### **assign()** :- This function is used to assign values to forward list, its another variant is used to assign repeated elements.

### **push\_front()** :- This function is used to insert the element at the first position on forward list. The value from this function is copied to the space before first element in the container. The size of forward list increases by 1.

### **emplace\_front()** :- This function is similar to the previous function but in this no copying operation occurs, the element is created directly at the memory before the first element of the forward list.

### **pop\_front()** :- This function is used to delete the first element of list.

### **insert\_after()** This function gives us a choice to insert elements at any position in forward list. The arguments in this function are copied at the desired position.

### **emplace\_after()** This function also does the same operation as above function but the elements are directly made without any copy operation.

### **erase\_after()** This function is used to erase elements from a particular position in the forward list.

### **remove()** :- This function removes the particular element from the forward list mentioned in its argument.

### **remove\_if()** :- This function removes according to the condition in its argument.

### **splice\_after()** :- This function transfers elements from one forward list to other.

# **Manipulating Functions**

### **merge()** :- This function is used to merge one forward list with other. If both the lists are sorted then the resulted list returned is also sorted.

### **operator “=”** :- This operator copies one forward list into other. The copy made in this case is deep copy.

### **sort()** :- This function is used to sort the forward list.

### **unique()** :- This function deletes the multiple occurrences of a number and returns a forward list with unique elements. The forward list should be sorted for this function to execute successfully.

### **reverse()** :- This function is used to reverse the forward list.

### **swap()** :- This function swaps the content of one forward list with other.

### **clear()** :- This function clears the contents of forward list. After this function, the forward list becomes empty.

### **empty()** :- This function returns true if the list is empty otherwise false.

## List in C++ Standard Template Library (STL)

### Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.

### front() – Returns the value of the first element in the list

### back() – Returns the value of the last element in the list

### push\_front(g) – Adds a new element ‘g’ at the beginning of the list

### push\_back(g) – Adds a new element ‘g’ at the end of the list

### pop\_front() – Removes the first element of the list, and reduces size of the list by 1

### pop\_back() – Removes the last element of the list, and reduces size of the list by 1

### begin() – Returns an iterator pointing to the first element of the list

### end() – Returns an iterator pointing to the theoretical last element which follows the last element

### empty() – Returns whether the list is empty(1) or not(0)

### insert() – Inserts new elements in the list before the element at a specified position

### erase() – Removes a single element or a range of elements from the list

### assign() – Assigns new elements to list by replacing current elements and resizes the list

### remove() – Removes all the elements from the list, which are equal to given element

### reverse() – Reverses the list

### size() – Returns the number of elements in the list

### sort() – Sorts the list in increasing order

**Some Useful Functions:**

### **emplace(position, value)** :- This function is used to **insert** an element at the **position** specified.

### **emplace\_back(value)** :- This function adds **value at end** of list. It is different from push\_back() by the fact that it directly creates element at position whereas push\_back() first makes a temporary copy and copies from there. emplace\_back() is faster in implementation than push\_back() in most situations.

### **emplace\_front** :- This function adds **value at beginning** of list. It is different from push\_front() by the fact that it directly creates element at position whereas push\_front() first makes a temporary copy and copies from there. emplace\_front() is faster in implementation than push\_front() in most situations.

### **merge(list2)** :- This function is used to **merge list2 with list1**. If both the lists are in sorted order, then the resulting list is also sorted.

### **remove\_if(condition)** :- This function **removes the element** from list on the **basis of condition** given in its argument.

### **unique()** :- This function is used to **delete the repeated occurrences** of the number. List has to be **sorted** for this function to get executed.

### **splice(position, list2)** :- This function is used to **transfer elements** from one list into another.

### **swap(list2)** :- This function is used to **swap one list element with other**.

## Vector in C++ STL

### Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time, because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

## Iterators

### begin() – Returns an iterator pointing to the first element in the vector

### end() – Returns an iterator pointing to the theoretical element that follows last element in the vector

### rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

### rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

## Capacity

### size() – Returns the number of elements in the vector

### max\_size() – Returns the maximum number of elements that the vector can hold

### capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements

### resize(size\_type g) – Resizes the container so that it contains ‘g’ elements

### empty() – Returns whether the container is empty

## **Accessing the elements**

### reference operator [g] – Returns a reference to the element at position ‘g’ in the vector

### at(g) – Returns a reference to the element at position ‘g’ in the vector

### front() – Returns a reference to the first element in the vector

### back() – Returns a reference to the last element in the vector

## Algorithm Library | C++ Magicians STL Algorithm

**Non-Manipulating Algorithms**

### [**sort**](https://www.geeksforgeeks.org/sort-c-stl/)**(first\_iterator, last\_iterator)** – To sort the given vector.

### **reverse(first\_iterator, last\_iterator)** – To reverse a vector.

### **\*max\_element (first\_iterator, last\_iterator)** – To find the maximum element of a vector.

### **\*min\_element (first\_iterator, last\_iterator)** – To find the minimum element of a vector.

### **accumulate(first\_iterator, last\_iterator, initial value of sum)** – Does the summation of vector elements

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

### **find(first\_iterator, last\_iterator, x)** – Points to last address of vector ((name\_of\_vector).end()) if element is not present in vector.

### [**binary\_search**](http://quiz.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’.

**Some Manipulating Algorithms**

### **arr.erase(position to be deleted)** – This erases selected element in vector and shifts and resizes the vector elements accordingly.

### **arr.erase(unique(arr.begin(),arr.end()),arr.end())** – This erases the duplicate occurrences in sorted vector in a single line.

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

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

### **distance(first\_iterator,desired\_position)** – It returns the distance of desired position from the first iterator.This function is very useful while finding the index.

## Array algorithms in C++ STL (all\_of, any\_of, none\_of, copy\_n and iota)

### **all\_of()**

### **any\_of()**

### **none\_of()**

### **copy\_n()**

### **iota()**

## std::partition in C++ STL

### C++ has a class in its STL algorithms library which allows us easy partition algorithms using certain inbuilt functions. Partition refers to act of dividing elements of containers depending upon a given condition.

### **partition(beg, end, condition)** :- This function is used to **partition the elements** on **basis of condition** mentioned in its arguments.

### **is\_partitioned(beg, end, condition)** :- This function returns boolean **true if container is partitioned** else returns false.

### **stable\_partition(beg, end, condition)** :- This function is used to **partition the elements** on **basis of condition** mentiond in its arguments in **sorted order**.

### **4. partition\_point(beg, end, condition)** :- This function **returns an iterator pointing to the partition point** of container i.e. the first element in the partitioned range [beg,end) for which condition is not true. The container should already be partitioned for this function to work.

### **partition\_copy(beg, end, beg1, beg2, condition)** :- This function **copies the partitioned elements** in the differenet containers mentioned in its arguments. It takes 5 arguments. **Beginning and ending position of container, beginning position of new container where elements have to be copied (elements returning true for condition), beginning position of new container where other elements have to be copied (elements returning false for condition) and the condition**. **Resizing** new containers **is necessary** for this function.

## Sort in C++ Standard Template Library (STL)

### [Sorting](https://www.geeksforgeeks.org/sorting-algorithms/) is one of the most basic functions applied to data. It means arranging the data in a particular fashion, which can be increasing or decreasing. There is a builtin function in C++ STL by the name of sort(). Internally this function is implemented as Quick-sort. The complexity of it is O(N\*log(N)).

## Binary Search in C++ Standard Template Library (STL)

### [Binary search](https://www.geeksforgeeks.org/binary-search/) is a widely used searching algorithm that requires the array to be sorted before search is applied. The main idea behind this algorithm is to keep dividing the array in half (divide and conquer) until the element is found, or all the elements are exhausted.

### It works by comparing the middle item of the array with our target, if it matches, it returns true otherwise if the middle term is greater than the target, the search is performed in the left sub-array. If the middle term is less than target, the search is performed in the right sub-array.

## Priority Queue

### Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non decreasing order(hence we can see that each element of the queue has a priority{fixed order}).

## std::istream\_iterator and std::ostream\_iterator

### Iterators are the link between the Containers and the Algorithms. They are the common interface to these classes. An Iterator is an object which can be used to iterator over the elements in a container. Thus iterators are used by Algorithms to modify the containers. The advantage of stream iterators is that they provide a common interface to access elements in I/O stream, file streams and also other stream to external physical devices.

### Once an iterator to the respective stream has been obtained the code that follows is nearly same form all types of streams.

### Thus tasks like reading from an input stream and reading from another external stream become similar.

### Stream Iterators allow us access to all the powerful STL algorithms like for\_each, replace\_if which take an input range to operate on. A particularly useful function is the [copy() function](https://www.geeksforgeeks.org/different-methods-copy-c-stl-stdcopy-copy_n-copy_if-copy_backwards/). This function is used to copy the elements from one container to another.

### Using the copy() function, we can easily transfer data from a stream to a container and vice-versa. Here are a few example programs to demonstrate working with stream iterators

## std::next\_permutation and prev\_permutation

### It is used to rearrange the elements in the range [first, last) into the next lexicographically greater permutation. A permutation is each one of the N! possible arrangements the elements can take (where N is the number of elements in the range). Different permutations can be ordered according to how they compare lexicographically to each other.

## shuffle vs random\_shuffle

### It randomly rearrange elements in range [first, last). The function swaps the value of each element with some other randomly picked element. When provided, the function gen determines which element is picked in every case. Otherwise, the function uses some unspecified source of randomness.

**What is the difference between shuffle and random\_shuffle c++?**

* The only difference is that **random\_shuffle** uses **rand()** function to randomize the items, while the **shuffle** uses **urng** which is a better random generator, though with the particular overload of random\_shuffle, we can get the same behavior (as with the shuffle).
* shuffle is an improvement over random\_shuffle, and we should prefer using the formerfor better results.

## Difference between set, multiset, unordered\_set, unordered\_multiset

### [**Set**](https://www.geeksforgeeks.org/set-in-cpp-stl/)

* Stores the values in sorted order.
* We can erase more than 1 element by giving start iterator and end iterator position.
* Stores only unique values.
* Elements can only be inserted or deleted but cannot be modified.
* Traversal using iterators.
* Sets are implemented as [Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/).

### [**Multiset**](https://www.geeksforgeeks.org/multiset-in-cpp-stl/)

* Stores elements in sorted order.
* We can erase more than 1 element by giving start iterator and end iterator.
* It allows storage of multiple elements.
* Note:- All other properties similar to set.
* [**Unordered\_set**](https://www.geeksforgeeks.org/unorderd_set-stl-uses/)
* Elements can be stored in any order. ( no sorted order )
* **Hash-table** used to store elements.
* Stores only unique values.
* We can erase only the element for which iterator position is given.
* Note:- All other properties similar to set.

### [**Unordered\_multiset**](https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/)

* Elements can be stored in any order.
* **Hash-table** used to store elements.
* Duplicate elements can be stored.
* We can erase only the element for which iterator position is given.
* Note:- All other properties similar to set.

## std::slice (Valarray slice selector)

## std::slice is the selector class that identifies a subset of std::valarray. An object of type std::slice holds three values: the starting index, the stride, and the total number of values in the subset. Objects of type std::slice can be used as indexes with valarray’s operator[].

**slice(size\_t start, size\_t size, size\_t stride );**

**size\_t star :** is the index of the first element in the selection

**size\_t size :** is the number of elements in the selection

**stride :** is the span that separates the elements selected.

* In given Syntax, by default constructor isequivalent to slice(0, 0, 0). This constructor exists only to allow construction of arrays of slices.
* It constructs a new slice with parameters start, size, stride. This slice will refer to size number of elements, each with the position.

**Example:**

* **slice(1, 5, 4)**
* Input : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
* Output : 1 5 9 13 17
* Explanation: starting from index 1 then next index 1 + 1 \* 4 = 5, next index 1 + 2 \* 4 = 9, next index 1 + 3 \* 4 = 13, next index 1 + 4 \* 4 = 17.

## std::memchr

## One of them is memchr() function which will search for the first occurrence of a character in a specified number of characters.

## std::strncmp

### function lexicographically compares not more than count characters from the two null-terminated strings and returns an integer based on the outcome.

* This function takes two strings and a number **num** as arguments and compare at most first **num** bytes of both the strings.
* **num** should be at most equal to the length of the longest string. If **num** is defined greater than the string length than comparison is done till the null-character(‘\0’) of either string.
* This function compares the two strings lexicographically. It starts comparison from the first character of each string. If they are equal to each other, it continues and compare the next character of each string and so on.
* This process of comparison stops until a terminating null-character of either string is reached or **num** characters of both the strings matches.

## std::stable\_sort()

## Like [std::sort()](https://www.geeksforgeeks.org/sort-c-stl/), stable\_sort also sorts an array. The syntax is also same.

### **When to prefer stable\_sort over sort()?**

### Sometime we want to make sure that order of equal elements is same in sorted array as it was in original array. This can be useful if these values have associated other fields. For example, consider sorting students by marks, if two students have same marks, we may want to keep them in same order as they appear input. Please refer [stability in sorting algorithms](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/) for details.

### Consider following example. We have time intervals sorted by ending time and we want to sort by start time. Also, if two start times are same, we want to keep them sorted by end time. This is not guaranteed by sort().

### **Implementation**

### sort() function usually uses [Introsort](https://www.geeksforgeeks.org/know-your-sorting-algorithm-set-2-introsort-cs-sorting-weapon/). Therefore, sort() may preserve the physical order of semantically equivalent values but can’t be guaranteed. stable\_sort() function usually uses [mergesort](https://www.geeksforgeeks.org/merge-sort/). Therefore, stable\_sort() preserve the physical order of semantically equivalent values and its guaranteed.

### **Time Complexity**

### For sort() it is O(n\*log(n)) For stable\_sort() it is O(n\*log^2(n)) if additional memory linearly proportional to length is not available. If its available then O(n\*log(n)).

## std::bucket\_count and std::bucket\_size in unordered\_map

## as we know, internally [unordered\_map](https://www.geeksforgeeks.org/unordered_map-in-stl-and-its-applications/) is implemented using hash table so, a bucket is a slot in the internal hash Table to which elements are assigned based on the hash value of their key. Buckets are numbered from 0 to (bucket\_count-1). Hence this function returns the bucket no. where element with **key** is located in unordered\_map. Time Complexity: O(1).

## std::includes()

## C++ defines a function which can be used to recognize if all the numbers in a container, also exists in other container. This task can be achieved by using “**includes()**”

## std::sort\_heap

### The **sort\_heap( )** is an STL algorithm which sorts a heap within the range specified by start and end. Sorts the elements in the heap range [start, end) into ascending order.

### The second form allows you to specify a comparison function that determines when one element is less than another.

## map vs unordered\_map

### Use std::map when

### You need ordered data.

### You would have to print/access the data (in sorted order).

### You need predecessor/successor of elements.

### See [advantages of BST over Hash Tabl](https://www.geeksforgeeks.org/advantages-of-bst-over-hash-table/)e for more cases.

### Use std::unordered\_map when

### You need to keep count of some data (Example – strings) and no ordering is required.

### You need single element access i.e. no traversal.

|  |  |  |
| --- | --- | --- |
|  | map | unordered\_map |
| Ordering | increasing order | no ordering |
| Implementation | Self balancing BST like [Red-Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/) | Hash Table |
| search time | log(n) | O(1) -> Average  O(n) -> Worst Case |
| Insertion time | log(n) + Rebalance | Same as search |
| Deletion time | log(n) + Rebalance | Same as search |

## Heap in C++ STL

### make\_heap(), push\_heap(), pop\_heap(), sort\_heap(), is\_heap, is\_heap\_until()

### Heap data structure can be implemented in a range using STL which allows faster input into heap and retrieval of a number always results in the largest number i.e. largest number of the remaining numbers is popped out each time. Other numbers of the heap are arranged depending upon the implementation.

### **make\_heap()** :- This function is used to **convert a range** in a container **to a heap.**

### **front()** :- This function displays the **first element** of heap which is the **maximum number**.

### **push\_heap()** :- This function is used to **insert** elements into heap. The size of the heap is increased by 1. New element is placed appropriately in the heap.

### **pop\_heap()** :- This function is used to **delete the maximum element** of the heap. The size of heap is decreased by 1. The heap elements are reorganised accordingly after this operation.

### **sort\_heap()** :- This function is used to **sort** the heap. After this operation, the container is **no longer a heap**.

### **is\_heap()** :- This function is used to **check** whether the container is **heap** or not. Generally, in most implementations, the **reverse sorted container** is considered as heap. Returns true if container is heap else returns false.

### **is\_heap\_until()** :- This function returns the iterator to the position **till the container is the heap.** Generally, in most implementations, the **reverse sorted container** is considered as heap.

## std::transform()

## Consider the problem of adding contents of two arrays into a third array. It is given that all arrays are of same size.

## What all is inherited from parent class in C++?

### Following are the things which a derived class inherits from its parent.

### Every data member defined in the parent class (although such members may not always be accessible in the derived class!)

### Every ordinary member function of the parent class (although such members may not always be accessible in the derived class!)

### The same initial data layout as the base class.

### Following are the things which a derived class doesn’t inherits from its parent :

### The base class’s constructors and destructor.

### The base class’s friends

## Binary Search (binary\_search, lower\_bound and upper\_bound)

### **binary\_search(start\_ptr, end\_ptr, num)** : This function returns boolean **true if the element is present** in the container, else returns false.

### **lower\_bound(start\_ptr, end\_ptr, num)** : Returns pointer to “**position of num**” if container contains **1** occurrence of num. Returns pointer to “**first position of num**” if container contains **multiple occurrence** of num. Returns pointer to “**position of next higher number than num**” if container **does not contain occurrence** of num. Subtracting the pointer to 1st position i.e “**vect.begin()**” returns the actual index.

### **upper\_bound(start\_ptr, end\_ptr, num)** : Returns pointer to “**position of next higher number than num**” if container contains **1** occurrence of num. Returns pointer to “**first position of next higher number than last occurrence of num**” if container contains **multiple occurrence** of num. Returns pointer to “**position of next higher number than num**” if container **does not contain occurrence** of num. Subtracting the pointer to 1st position i.e “**vect.begin()**” returns the actual index.

## Insertion and Deletion in STL Set

### **Insert function** is used to insert the elements in the set. After insertion, the reordering of elements takes place and the the set is sorted. This function is implemented in 3 ways.

### insert(ele) : This function inserts the element in set. The insertion only takes place when the element passed is not already in set. It returns a pointer pair . First element pointing to the element already present or newly inserted. Second element returning the boolean status “true” or “false”.

### insert(hint, ele) : In this implementation, the hint pointer is sent with the element to be inserted. The use of hint pointer is to help insert() know where the actual insertion has to take place. Hence, trying to reduce time to allocate the element. Hint pointer does not force the insertion at specific position. This function returns the pointer to the position where element is inserted.

### insert(beg\_ptr, end\_ptr) : This type of insertion is required to insert the elements of other container into set. The repeated elements are not inserted if they are present in the source container.

### **Emplace** is also used to insert the element into the Set. This function is similar to “insert()” discussed above, the only difference being **that “in-place” construction of element takes place at the position of element insertion** contrary to insert() which copies or movies existing object.

### emplace() : Inserts element using in-place construction strategy. Increases the size of set by 1. returns a pointer pair. 1st element of which is iterator pointing to the position of inserted element. 2nd returns a boolean variable indicating an already present or newly created element.

### emplace\_hint() : Takes a “hint\_iterator” to get a hint of position of insertion to possibly reduce the time required to insert the element inserted. This does not effect the position of insertion. It takes place where it is defined to internally.

### **Erase** is used to **erase** the element in set mentioned in argument, either its position, its value or a range of number.

### **erase(num)** : Erases the **value** mentioned in its argument. reorders the set after deletion.

### **erase(iter)** : Erases the value **at the position pointed by the iterator** mentioned in its argument.

### **erase(strt\_iter,end\_iter)** : Erases the **range of elements** starting from “strt\_iter” to the “end\_iter”.

## User Defined Literals

### **U**ser **D**efined **L**iterals (**UDL**) are added in C++ from C++11. Although, C++ provides literals for a variety of built-in types but these are limited.

### **UDLs** are treated as a call to a literal operator. Only suffix form is supported. The name of the literal operator is **operator “”** followed by the suffix.

## Random

### **Pseudo-random number engines:** They use an algorithm to generate random numbers based on an initial seed.

### **linear\_congruential\_engine**: It is the simplest engine in the STL library that generates random unsigned integer numbers

### **mersenne\_twister\_engine:** It is a random number engine based on Mersenne Twister algorithm. It produces high quality unsigned integer random numbers in the interval [0, (2^w)-1]. where ‘w’ is word size: Number of bits of each word in the state sequence.

### **subtract\_with\_carry\_engine:**It is a pseudo-random number generator engine that produces unsigned integer numbers. The algorithm used is a lagged [fibonacci generator](https://en.wikipedia.org/wiki/Lagged_Fibonacci_generator), with a state sequence of r integer elements, plus one carry value.

### **Random number generator**:It is a random number generator that produces non-deterministic random numbers.

### **Pseudo-random number engines (instantiations)**: These are the particular instantiations of generator engines and adaptors

### **default\_random\_engine**: This is a random number engine class that generates pseudo-random numbers.

### **minstd\_rand:** It generates pseudo random numbers; it is similar to [linear congruential generator](https://en.wikipedia.org/wiki/Linear_congruential_generator)

### **mt19937:** It is Mersenne Twister 19937 generator.It is a pseudo-random generator of 32-bit numbers with a state size of 19937 bits.

### **ranlux24\_base:** It is Ranlux 24 base generator. It’s a subtract-with-carry pseudo-random generator of 24-bit numbers, generally used as the base engine for the ranlux24 generator.

## **Engine Adaptors**

### **discard\_block\_engine:** It is an engine adaptor class template that adapts a **pseudo-random number generator Engine** type by using only ‘r’ elements of each block of ‘p’ elements from the sequence it produces, discarding the rest.

### **independent\_bits\_engine:** It is an engine adaptor class template that adapts a **pseudo-random number generator Engine** type to produce random numbers with a specific number of bits (w).

### **shuffle\_order\_engine:** It is an engine adaptor class template that adapts a **pseudo-random number generator Engine** type so that the numbers are delivered in a different sequence.

## Array Type Manipulation

### **is\_array() :** As the name suggest, the sole purpose of this function is to check if a variable is a array type or not. Notable here is that even an [std::array](https://www.geeksforgeeks.org/array-class-c/) is not considered an array according to this function. The “value” member constant returns true if type is array, else returns a false.

### **is\_same() :** This function is to check **Type relationships** and it returns true if two types have exactly same characteristics. The “value” member constant returns true if types are same, else returns a false.

### **rank() :** This is a **property query function** which returns the rank of the array. Rank means the **dimension of the array**. The value member constant returns the rank of object.

### **extent() :** Both extent and remove extent are **Compound type alterations** that can be applied to arrays in C++. This function returns the size of the particular dimension of array. This function takes two arguments, the array type and the dimension in whose size have to be found. This also has the member constant value for printing value.

### **remove\_extent() :** This function removes the first dimension from left in the matrix/array declared.

### **remove\_all\_extents() :** This function removes all the dimensions of the matrix/array and converts it into base data type.

## std::stod, std::stof, std::stold

### **std::stod() :** It convert string into double.

### **std::stof :** It convert string into float.

### **std::stold :** It convert string into long double.

## **Valarray**

### **apply()** :- This function **applies the manipulation** given in its arguments **to all** the valarray elements at once and **returns a new valarray** with manipulated values.

### **sum()** :- This function **returns the summation** of all the elements of valarrays at once.

### **min()** :- This function returns the **smallest** element of valarray.

### **max()** :- This function returns the **largest** element of valarray.

### **shift()** :- This function returns the new valarray after **shifting elements by** the **number** mentioned in its argument. If the **number is positive**, **left-shift** is applied, if **number is negative**, **right-shift** is applied.

### **cshift()** :- This function returns the new valarray after **circularly shifting(rotating)** elements **by** the **number** mentioned in its argument. If the **number is positive, left-circular** **shift** is applied, if **number is negative, right-circular shift** is applied.

### **swap()** :- This function **swaps** one valarray with other.

## Floating Point Manipulation (fmod(), remainder(), remquo()

### **fmod()** : This function is used to return the **remainder(modulus) of 2 floating point numbers** mentioned in its arguments. The quotient computed is **truncated**.

### **remainder()** : This function is also used to return the **remainder(modulus) of 2 floating point numbers** mentioned in its arguments.The quotient computed is **rounded**.

### **remquo()** : This function returns the remainder and also stored remainder in variable reference passed as argument. **This function takes 3 arguments, numerator, denominator and reference of variable where quotient has to be stored.**

### **copysign()** : This function returns a number with the **magnitude to 1st argument and sign of 2nd argument**.

### **nextafter()** : This function computes the **next representable value of 1st argument in the direction of 2nd argument**.

### **fmin()** : Returns the **smallest** of two arguments.

### **fmax()** : Returns the **largest** of two arguments.

### **fdim()** : Returns the **positive difference** of the numbers passed as arguments.

### **fma()** : This function takes **3 arguments** and returns **multiply-add** “**x\*y+z**” value after computing.

## Character Classification

### **isalpha() :** This function returns **true if character is an alphabet** else returns false. All the characters from **a-z and A-Z** return true according to this function.

### **isalnum() :** This function returns **true if character is an alphabet or a number** else returns false. All the characters from **a-z , A-Z and all numbers** return true according to this function.

### **isdigit() :** This function returns **true if character is a number** else returns false. All **all numbers** return true according to this function.

### **isblank() :** This function returns **true if character is a space or tab** else returns false.

### **isspace() :** This function returns **true if character is a space or tab or whitespace control code ( Eg.\n,\r )** else returns false.

### **iscntrl() :** This function returns **true if character is tab or any control code** else returns false.

### **isprint() :** This function returns **true if character is printable on console i.e. all characters except control code** else returns false.

### **isxdigit() :** This function returns **true if character is hexadecimal i.e 0-9 and a-f** else returns false.

### **ispunct() :** This function returns **true if character is punctuation mark** else returns false.

## std::none\_of

### Returns true if pred returns false for all the elements in the range [first, last] or if the range is empty, and false otherwise.

## rint(), rintf(), rintl()

### rint() is used to round of the floating-point argument to an integer value (in floating-point format). You can also determine the current rounding mode using a function **fesetround()** according to which the rint function returns the rounded integer value.

## std::gslice | Valarray generalized slice selector

### **Valarray generalized slice selector :** This class represents a valarray generalized slice selector (a multidimensional slice). It does not contain nor refers to any element – it only describes a selection of elements to be used as an index in **valarray::operator[]**.

## std::setbase, std::setw , std::setfill

### **std::base** : Set basefield flag; Sets the base-field to one of its possible values: dec, hex or oct according to argument base.

### **std::setw** : Set field width; Sets the field width to be used on output operations. Behaves as if member width were called with n as argument on the stream on which it is inserted/extracted as a manipulator (it can be inserted/extracted on input streams or output streams).

### **std::setfill** : Set fill character; Sets c as the stream’s fill character. Behaves as if member fill were called with c as argument on the stream on which it is inserted as a manipulator (it can be inserted on output streams).

## strxfrm()

### The c/c++ library **strxfrm()** transform the characters of source string into current locale and place them in destination string. For that LC\_COLLATE category is used which is defined in **locale.h** . strxfrm() function performs transformation in such a way that result of strcmp on two strings is the same as result of strcoll on two original strings.

## strstr()

### This function takes two strings **s1** and **s2** as an argument and finds the first occurrence of the sub-string **s2** in the string **s1**. The process of matching does not include the terminating null-characters(‘\0’), but function stops there.

## difftime()

### The C library function **difftime()** returns the difference, in seconds between starting time and ending time.(**ending time-starting time**)

## Std::String

### **Input Functions**

### **getline()** :- This function is used to **store a stream of characters** as entered by the user in the object memory.

### **push\_back()** :- This function is used to **input** a character at the **end** of the string.

### **pop\_back()** :- Introduced from C++11(for strings), this function is used to **delete the last character** from the string.

### **Capacity Functions**

### **3. capacity()** :- This function **returns the capacity** allocated to the string, which can be **equal to or more than the size** of the string. Additional space is allocated so that when the new characters are added to the string, the **operations can be done efficiently**.

### **4. resize()** :- This function **changes the size of string**, the size can be increased or decreased.

### **5.shrink\_to\_fit()** :- This function **decreases the capacity** of the string and makes it equal to its size. This operation is **useful to save additional memory** if we are sure that no further addition of characters have to be made.

### **Iterator Functions**

### **7. begin()** :- This function returns an **iterator** to **beginning** of the string.

### **8. end()** :- This function returns an **iterator** to **end** of the string.

### **9. rbegin()** :- This function returns a **reverse iterator** pointing at the **end** of string.

### **10. rend()** :- This function returns a **reverse iterator** pointing at **beginning** of string.

### **Manipulating Functions**

### **11. copy(“char array”, len, pos)** :- This function **copies the substring in target character array** mentioned in its arguments. It takes 3 arguments, **target char array, length to be copied and starting position in string to start copying.**

### **12. swap()** :- This function **swaps** one string with other.

## Merge operations

#### merge(), includes(), set\_union(), set\_intersection(), set\_difference(), ., inplace\_merge,

### **merge(beg1, end1, beg2, end2, beg3)** :- This function merges two sorted containers and stores in new container in sorted order (merge sort). It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container.

### **includes(beg1, end1, beg2, end2)** :- This function is used to check whether one sorted container elements are including other sorted container elements or not. Returns true if 1st container includes 2nd container else returns false.

### **inplace\_merge(beg1, beg2, end) :-** This function is used to sort two consecutively placed sorted ranges in a single container. It takes 3 arguments, iterator to beginning of 1st sorted range, iterator to beginning of 2nd sorted range, and iterator to last position.

### **set\_union(beg1, end1, beg2, end2, beg3)** :- This function computes the set union of two containers and stores in new container .It returns the iterator to the last element of resultant container. It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container . The containers should be sorted and it is necessary that new container is resized to suitable size.

### **set\_intersection(beg1, end1, beg2, end2, beg3)** :- This function computes the set intersection of two containers and stores in new container .It returns the iterator to the last element of resultant container. It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container . The containers should be sorted and it is necessary that new container is resized to suitable size.

### **set\_difference(beg1, end1, beg2, end2, beg3)** :- This function computes the set difference of two containers and stores in new container .It returns the iterator to the last element of resultant container. It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container . The containers should be sorted and it is necessary that new container is resized to suitable size.

### s**et\_symmetric\_difference(beg1, end1, beg2, end2, beg3)** :- This function computes the set symmetric difference of two containers and stores in new container .It returns the iterator to the last element of resultant container. It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container . The containers should be sorted and it is necessary that new container is resized to suitable size.

## adjacent\_difference(), inner\_product() and iota()

### **adjacent\_difference :** This functions assigns the difference between the corresponding elements of an array to another array. It returns the **adjacent difference of all the set of values lying between [ First, last )**.

b[0] = a[1]

b[1] = a[1] – a[0]

b[2] = a[2] – a[1]

b[3] = a[3] – a[2]

b[4] = a[4] – a[3]

### **inner\_product:** This function returns the result of addition of var with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

### **Iota :** This function assigns a value to the elements in the range [first,last ) of the array which is incremented at each step by val++

## Bind function and placeholders

### **How does bind() work?**

### Bind function with the help of placeholders, helps to manipulate the position and number of values to be used by the function and modifies the function according to the desired output.

### **What are placeholders?**

### Placeholders are namespace which direct the position of a value in a function. They are represented by **\_1, \_2, \_3**…

## Tuples

### A tuple is an object that can hold a number of elements. The elements can be of different data types. The elements of tuples are initialized as arguments in order in which they will be accessed.

### **get()** :- get() is used to access the tuple values and modify them, it accepts the index and tuple name as arguments to access a particular tuple element.

### **make\_tuple()** :- make\_tuple() is used to assign tuple with values. The values passed should be in order with the values declared in tuple.

### **tuple\_size** :- It returns the number of elements present in the tuple.

### **swap()** :- The swap(), swaps the elements of the two different tuples.

### **tie()** :- The work of tie() is to unpack the tuple values into seperate variables. There are two variants of tie(), with and without “ignore” , the “ignore” ignores a particular tuple element and stops it from getting unpacked.

### **tuple\_cat()** :- This function concatenates two tuples and returns a new tuple.

## std::regex\_match, std::regex\_replace() | Regex

### Regex is the short form for “**Regular expression**”, which is often used in this way in programming languages and many different libraries. It is supported in C++11 onward compilers.

### **regex\_match()** -This function return true if the regular expression is a match against the given string otherwise it returns false.

### **regex\_search()** – This function is used to search for a pattern matching the regular expression

### **regex\_replace()** This function is used to replace the pattern matching to the regular expression with a string.

## constexpr specifier

### In C++ 11, a constexpr function should contain only one return statement. C++ 14 allows more than one statements.

### constexpr function should refer only constant global variables.

### constexpr function can call only other constexpr function not simple function.

### Function should not be of void type and some operator like prefix increment (++v) are not allowed in consexpr function.

### They serve different purposes. constexpr is mainly for optimization while const is for practically const objects like value of Pi.

### Both of them can be applied to member methods. Member methods are made const to make sure that there are no accidental changes by the method. On the other hand, the idea of using constexpr is to compute expressions at compile time so that time can be saved when code is run. const can be only be used with non-static member function whereas constexpr can be used with with member and non-member functions, even with constructors but with condition that argument and return type must be of literal types.

### This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## std::fill() and fill\_n()

### **fill()**

### The ‘fill’ function assigns the value ‘val’ to all the elements in the range [begin, end), where ‘begin’ is the initial position and ‘end’ is the last position.

### **fill\_n()**

### In fill\_n(), we specify beginning position, number of elements to be filled and values to be filled. The following code demonstrates the use of fill\_n.

## Bitset

### A bitset is an array of bool but each Boolean value is not stored separately instead bitset optimizes the space such that each bool takes 1 bit space only, so **space taken by bitset bs is less than that of bool bs[N] and vector bs(N)**. However, a limitation of bitset is, **N must be known at compile time, i.e., a constant** (this limitation is not there with vector and dynamic array)

## Placement new operator

### Placement new is a variation [new](https://www.geeksforgeeks.org/malloc-vs-new/) operator in C++. Normal new operator does two things : (1) Allocates memory (2) Constructs an object in allocated memory.

### Placement new allows us to separate above two things. In placement new, we can pass a preallocated memory and construct an object in the passed memory.

### **new vs placement new**

### Normal new allocates memory in heap and constructs objects tehre whereas using **placement new**, object construction can be done at **known address**.

### With normal new, it is not known that, at what address or memory location it’s pointing to, whereas the address or memory location that it’s pointing is known while **using placement new.**

### The deallocation is done using [delete](https://www.geeksforgeeks.org/g-fact-30/) operation when allocation is done by new but there is no placement delete, but if it is needed one can write it with the help of [destructor](https://www.geeksforgeeks.org/playing-with-destructors-in-c/)

### http://cdncontribute.geeksforgeeks.org/wp-content/uploads/final1.jpg

### Advantages of placement new operator over new operator

### The address of memory allocation is known before hand.

### Useful when building a memory pool, a garbage collector or simply when performance and exception safety are paramount.

### There’s no danger of allocation failure since the memory has already been allocated, and constructing an object on a pre-allocated buffer takes less time.

### This feature becomes useful while working in an environment with limited resources.

## find\_end

### **std::find\_end** is used to find the last occurrence of a sub-sequence inside a container. It searches the range [first1,last1) for the last occurrence of the sequence defined by [first2,last2), and returns an iterator to its first element, or last1 if no occurrences are found.

### It is similar to [**std::search**](https://www.geeksforgeeks.org/stdsearch-in-c/) in such a way that in std::search , we look for the **first** occurrence of a sub-sequence inside another container, whereas in **std::find\_end**, we look for the **last** occurrence of such sub-sequence, and returns an iterator to the first element if such sub-sequence is found.

## find\_first\_of

### **std::find\_first\_of** is used to compare elements between two containers. It compares all the elements in a range [first1,last1) with the elements in the range [first2,last2), and if any of the elements present in the second range is found in the first one , then it returns an iterator to that element.

## swap\_ranges

### **td::swap** is used for swapping of elements between two containers. One of its variation is **std::swap\_ranges,** which as the name suggests is used for swapping the elements within a range.

## std::nth\_element

### **std::nth\_element()** is an STL algorithm which rearranges the list in such a way such that the element at the nth position is the one which should be at that position if we sort the list.

### It does not sort the list, just that all the elements, which precede the nth element are not greater than it, and all the elements which succeed it are not less than it.

# Containers Notes:

|  |
| --- |
| Vector: |
| Initialize a vector:  * vect.push\_back(10); * vector<int> vect(n, 10); * vector<int> vect{ 10, 20, 30 }; * vector<int> vect(arr, arr +( sizeof(arr) / sizeof(arr[0]))); * vector<int> vect2(vect1.begin(), vect.end()); |
| Ways to copy a vector:  * for (int i=0; i<vect1.size(); i++)   vect2.push\_back(vect1[i]);   * vect2 = vect1; * vector<int> vect2(vect1); * copy(vect1.begin(), vect1.end(), back\_inserter(vect2)); * vect2.assign(vect1.begin(), vect1.end()); |
| Can we use function on left side of an expression in C and C++? int &fun()  {     static int x;     return x;  }  int main()  {     fun() = 10;     printf(" %d ", fun());  } |
| Check two numbers are equal without using arithmetic operators or comparison operators. int x = 10;     int y = 10;     if ( !(x ^ y) )        printf(" x is equal to y ");     else        printf(" x is not equal to y "); |
| Print 1..N withot Semicolon int main()  {    static int x = 1;    if (cout << x << " " && x++ < N && main())    { }    return 0;  } |
| Program to find Maximum and minimum of two numbers without using any loop or condition.  int a = 15, b = 20;     printf("max = %d\n", ((a + b) + abs(a - b)) / 2);     printf("min = %d", ((a + b) - abs(a - b)) / 2);     return 0; |
| Sum the digits of a given number  * int getSum(int n)   {     int sum = 0;     while (n != 0)     {         sum = sum + n % 10;         n = n/10;     }     return sum;  }   * for (sum = 0; n > 0; sum += n % 10, n /= 10); * int sumDigits(int no)   {     return no == 0 ? 0 : no%10 + sumDigits(no/10) ;  } |

# Practices:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |