**C++ – LEARNINGS**

**(Very important from interview perspective as you need to know STL concepts to code)**

(<https://www.topcoder.com/community/competitive-programming/tutorials/power-up-c-with-the-standard-template-library-part-1/>)

These are topics (mainly of **STL**) due to which programmers use c++ more in CP contests. The things about each of them is as follows -

1. **Vectors (**<https://www.codesdope.com/cpp-stdvector/> - Also saved Offline)

Vector is just an **array (dynamic), in form of a template (**functions and classes to operate with generic types**)**, with extended functionality of **resize itself automatically when an element is inserted or deleted**, with their storage being handled automatically by the container. Some of the important features you learned are –

1. **Declaration –**

Just like normal arrays with datatype being vector<datatype> -

vector< *datatype* > *vector\_name*

1. **Initialization (insertion at the time of declaration) –**

<https://www.geeksforgeeks.org/initialize-a-vector-in-cpp-different-ways/>

1. **(Method 2 in above) Specifying size and initializing all values to same type (using constructor)**

vector< *datatype* > *vector\_name* (*size, default value*)

**NOTE –** Using something like this – **vector<int> vec()** will give an error for its reason you need to know about calling default constructor in c++ –

class base {

int i;

public:

base()

{

i = 10;

cout << "in the constructor" << endl;

}

};

base a declares a variable a of type base and calls its default constructor (assuming it's not a builtin type).

base a(); declares a function a that takes no parameters and returns type base.

The reason for this is because the language basically specifies that in cases of ambiguity like this anything that can be parsed as a function declaration should be so parsed. You can search for "C++ most vexing parse" for an even more complicated situation.

So **vector<int> vec()** is also making a function with name vec and return type **vector<int>.** To call the default constructor just use **vector<int> vec** (this is just the way to create an empty vector (declaration) i.e. allocate some memory to vec) **or vector<int> ()** (this will allocate space without any (object) name to the vector).

1. **(Method 3) Initializing just like array**

vector< *datatype* > *vector\_name* = { *values* }

1. **Accessing -**

<https://thispointer.com/c-how-to-get-element-by-index-in-vector-at-vs-operator/>

There are 2 methods –

1. **Using [] operator (like in c)**

As, operator [] returns a reference to the element in vector, so we can change the content of vector too using operator [].

**NOTE** - While accessing any element through operator [] we need to **make sure that given index is in range** i.e. less than the size of vector, **otherwise it will result in undefined behaviour** and can also crash application.

1. **Using at() function**

This also returns the reference of element at index n in vector but the difference is - **If index n is out of range** i.e. greater then size of vector then **it will throw out\_of\_range exception.**

**Which method to use?**

Always use [] only because it **is faster than at() as as() does bounds checking it makes it slower.**

**But then when to use at()?**

They are mainly **useful for catching bugs** in your code. Design your code with at() and if bounds are exceeded your program aborts meaning there is a bug.

1. **Insertion / assigning values –**
2. **Insert data one by one at the end on demand (main advantage over C array) –**

In vectors, data can be inserted at the end (and also in the middle with O(n) time) **without the knowing the end index (so you don’t have to allocate memory at compile time i.e. we can insert items as and when they arrive at run time) using push\_back()** demonstrated by the following -

**Example –** look at A3 – UNITGCD in codefiles. It perfectly demonstrates the advantage of vectors over c array in terms of space

**Disadvantage -**

Inserting at the end takes differential time, as sometimes there may be a need of extending the array.

**When to use –**

In most cases **use c style array only because it gives access to an element in O(1). But in case space is an issue (and time isn’t much of an issue)** then using this is better. Like in problem UNITGCD (space constraint was too high and using this did work better as this was giving answers till 104 as against 103)

1. **Using ‘=’ with accessing methods defined above**

As both return **a reference to the element** in vector and maybe because ‘=’ is overloaded such that if reference is left operand then the right operand value will be inserted at that reference.

1. **Vector Functions –**

A list of functions is given here <https://www.geeksforgeeks.org/vector-in-cpp-stl/>

1. **2-D Vectors**

<https://www.geeksforgeeks.org/2d-vector-in-cpp-with-user-defined-size/>

**Example –** A1 - UNITGCD in code files

* 1. **Allocating empty vector**

This is a very common patter when you need to allocate an empty vector to the 2-d vector and allocate them as and when need. This pattern is shown in the below code **Example** – BFS Tree Traversal & UNITGCD

1. **Unordered\_sets**

Unordered set are hash maps without value i.e. that is there are only key’s and they can be accessed in **O(1)** time just like with unordered\_maps.

<https://www.geeksforgeeks.org/unordered_set-insert-function-in-c-stl/>

<https://www.geeksforgeeks.org/unordered_set-in-cpp-stl/>

**Example –** A1 - UNITGCD

1. **Unordered\_maps (hash maps – chef and star value)**
2. **Iterators**

An *iterator* is used to move thru the elements an STL container (vector, list, set, map, ...) in a similar way to array indexes or pointers. The \* operator dereferences an iterator (ie, is used to access the element an iterator points to) , and ++ (and -- for most iterators) increments to the next element.

**Iterating over a vector with subscripts**

Passing over all the elements of a vector is simple, and is usually done using subscripts instead of an iterator. The main advantage of an iterator is that it can be used by many of the functions in <algorithms>. Subscripts provide an easy, familiar way to access vector elements in much the way that array elements would be accessed.

//--- Iterating over vector with subscript.

vector<int> v;

. . .

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

cout << v[i] << endl;

}

**Iterators are similar to pointers**

In fact, vectors iterators are usually implemented as pointers. If you feel comfortable using pointers to iterate over an array, you will feel comfortable using an iterator. For review, here is an example of one way to use a pointer to iterate over an array. Note that &a[n] is the address of the element *after* the last value currently in the array.

//--- Iterating over array with pointer.

int a[100];

int n = ...; // current number of elements

We could loop over this array like this.

for (int\* p = &a[0]; p != &a[n]; p++) {

cout << \*p << endl;

}

or

for (int\* p = a; p != a+n; p++) {

cout << \*p << endl;

}

**Iterating over a vector with an iterator**

//--- Iterating over vector with iterator.

vector<int> v;

. . .

for (vector<int>::iterator it = v.begin(); it!=v.end(); ++it) {

cout << \*it << endl;

}

**Why use iterators when subscripts work so well**

There are several reasons to use iterators.

* **Not always possible**. Subscripts cann**ot** be used on most of the containers (eg, list and map), so you must use iterators in many cases.
* **Flexible**. It is easily to change underlying container types. For example, you might decide later that the number of insertions and deletions is so high that a list would be more efficient than a vector.
* **Member functions**. Many of the member functions for vector use iterators, for example, assign, insert, or erase.
* **Algorithms**. The <algorithm> functions use iterators.

**Random Access**

Vector iterators are random-access.

1. **getline() & stringstream to read chars/numbers form a string**

These 2 methods are a really good way to read a string and manipulate it easily like – by reading chars or numbers form the string. The use of these functions is demonstrated in below code -

**Example code – Finding orders (UVA 124 – Stored offline)**

1. **getline()**

AS the name tells, getline() is used **to read a line from stream object (like cin or stringstream obj) and store it in a string.**

Everything is explained here - <https://www.geeksforgeeks.org/getline-string-c/> (Saved offline)

1. **Stringstream class**

* The stringstream class in C++ **allows a string object to be treated as a stream**.
* By treating the strings as streams we can perform extraction and insertion operation from/to string just like cin and cout streams.

These types of operations are mostly useful to convert string to numerical data types and vice versa. The stringstream class also proves to be helpful in different types of parsing.

<https://www.geeksforgeeks.org/stringstream-c-applications/>

<https://www.tutorialspoint.com/stringstream-in-cplusplus>

1. **Sorting with comparator functions**

This “comparator” function returns a value; convertible to bool, which basically tells us whether the passed “first” argument should be placed before the passed “second” argument or not.

To better understand this point, watch the video (sorting using comparator - codingblocks) given in folder.

Example Code – Finding orders (kahn’s algo approach) / <https://www.geeksforgeeks.org/sort-c-stl/>

**Passing comparator function in parameter**

1. **Through Callback -**

As given in the example code, Comparator function is **passed as a pointer to function**. This is because sort() wants the whole comparator function to be passed so that sort() can call the function whenever it wants. This is a **technique known as Callback (this can also be done in other ways as shown below but function pointer method is mostly used in CP).**

**Note** – sort() doesn’t want value of function, which we will get if we pass like - sortcol() as parameter.

1. **By using operator<() in a class –**

A really great article explaining this is - <http://fusharblog.com/3-ways-to-define-comparison-functions-in-cpp/> (also available offline). Read article carefully to understand the below -

This method is just **overloading operator < in a class (in article Class ‘Edge’) so that when we call sort(), we won’t be needing any comparator function.** To understand consider the below code from the article –

|  |
| --- |
| vector<Edge> v; |

|  |
| --- |
| sort(v.begin(), v.end()); |
|  |

* As no comparator function is passed while calling sort() above, **sort() will use operator < inside its definition to compare the individual Edge objects.**
* Now as we have overloaded operator < for objects of Edge**, when sort uses < on objects of Edge – the definition of our overloaded function is executed which is just like a comparator function used for sorting in descending order** and thus our edges will be sorted in descending order.

(If you don’t understand “STL Container” Vs “STL Function” read both 1st and 2nd way in article)

1. **Function pointers**

This article explains most (available OFFLINE ASLO) <https://www.geeksforgeeks.org/function-pointer-in-c/>

1. **Callbacks in C++**

A callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time (like in case of comparators functions for sorting). As you saw above, callback is a very useful feature so there are actually 3 ways in c++ to make use of callback –

1. **Using function pointer (ONLY WAY for callback in C. Read ‘function pointers’ first) –**

Using callback through function pointer, we just have to define the comparator function and just **pass** **the address of function i.e. passing just function name without parenthesis () to pass the whole function code of comparator function.**

Example code **-** Finding orders (kahn’s algo) / <https://www.geeksforgeeks.org/callbacks-in-c/>

**Problems with this approach (when we can’t use this) -**

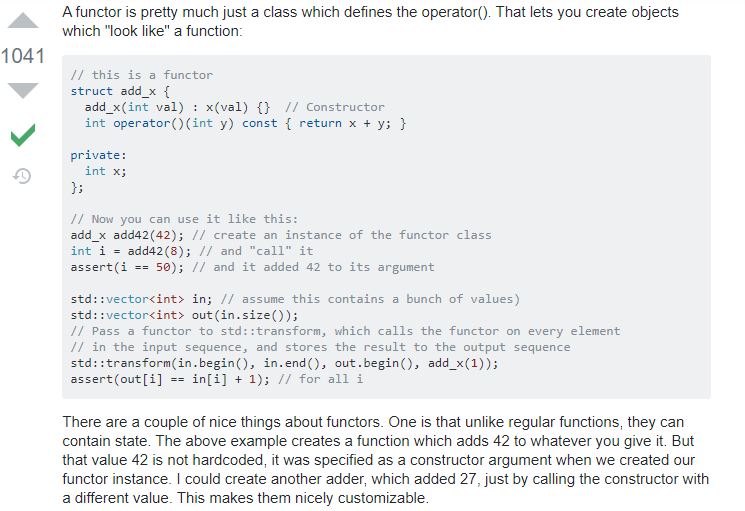
* But this method works only when the comparator is strictly a function (**not class method. For more see the note).**
* Also it can’t state which a class method can hold. Example case - <https://stackoverflow.com/questions/356950/what-are-c-functors-and-their-uses>

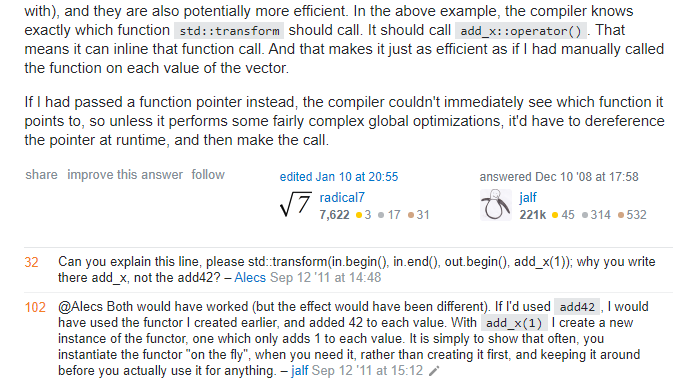
1. **Functors (not usable in c because no method can be defined inside struct)**

Functors (Function object) is a feature which **allows to use an object like a function** (by overloading **operator () inside the class definition).**

This useful over function pointer as **functors can store states (As they are objects).** To understand this better read the below tut & then the answers from stack overflow –

* <https://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html> (saved offline).
* Answer from -<https://stackoverflow.com/questions/356950/what-are-c-functors-and-their-uses>





**NOTE – Remember greater<int>()** **to sort in descending order, it is a BUILT IN functor as described in part of an article (given below) –**

|  |
| --- |
| STL also has some built-in functor classes (class templates) such as less<T>, the default comparison class, and greater<T>, the negation of less<T>. A functor can be used as an ordinary function by instantiating the class. For this purpose, the simplest way is to append **()** after the class name. So, for example, if you want to sort a vector<int> **data** in descending order (that’s the negation of ascending order), you may use:  sort(data.begin(), data.end(), greater<int>()); |

**Advantages over function pointer –**

1. From the above example, functors can save states whereas function pointers can’t
2. See the **NOTE at the end of this topic** to know one more case when function pointer can’t be used.
3. **Using lambda / closure**

A lambda is a functor - just defined with a shorter syntax (So it is better for CP as it saves time).

<https://www.cprogramming.com/c++11/c++11-lambda-closures.html>

**When to use lambda over functor –**

<https://stackoverflow.com/questions/4686507/lambda-expression-vs-functor-in-c>

(read accepted answer (1st answer), & **answer by martin York,** & answer at last explaining how to use lambda more than one)

**When to use functor over lambda –**

<https://stackoverflow.com/questions/27256062/when-to-use-functors-over-lambdas/27256160>

**(Read accepted answer & answer by** [5gon12eder](https://stackoverflow.com/users/1392132/5gon12eder))

**NOTE –**

The comparator in sort() **needs to be a callable object.** In other words, it can be: a function pointer or a lambda / closure, or a class with a suitable operator() (a functor).

**TRICK CASE – FUNCTION INSIDE A CLASS IS NOT A FUNCTION BUT A CLASS METHOD.**

A function inside a class is not a callable object. It's a class method. It is not a callable object for the simple reason that you cannot call it directly. A class method can only be called on an instance of the class. You must have an instance of this class somewhere, and you call its compare() method. It looks like a function, but it really isn't. It's a class method. **So function pointer can’t be used (because it’s not a function only).** TOounderstand better take a look at this problem from –

<https://stackoverflow.com/questions/41129973/c-how-to-pass-a-comparator-to-stl-functions-correctly>

So to solve the above kind of a problem (when comparator is part of the the same class as the calling function), we can use 2 ways (function pointer method can’t be used) –

1. **Using Lambda expression**

One easy solution would be to capture this via a lambda, something like (C++14):

nth\_element(points\_.begin(), points\_.begin() + k - 1, points\_.end(),

[this](const auto &a, const auto &b)

{

return this->compare(a, b);

});

The lambda captures this, and compare() can be invoked on this, just like it can be called directly from the parent method.

1. **Using Functors –**

The functor solution is elegant as well, to complete the picture.

You can add this functor inside (or outside) the PointCollection class:

struct compare {

Point center\_;

bool operator()(const Point &a, const Point &b) const {

return std::abs(a.val\_ - center\_.val\_) < std::abs(b.val\_ - center\_.val\_);

}

}

Then:

std::nth\_element(points\_.begin(), points\_.begin() + k - 1, points\_.end(), compare{center} )

1. **Solution to problems wanting you to write a function but you want to or have to write it ina different way –**

Now this a problem faced in GFG inorder traversal question when you wanted to make a function which wouldn’t return a an inorder array but the function definition given to you made it mandatory to return a vector with inorder. You can see the question here <https://practice.geeksforgeeks.org/problems/inorder-traversal/1>.

Now you actually can’t make such a function because with each function call the array data will be lost or if you make a global or static vector then the problem will come when ther will be multiple TC’s.

**Solution –** is to make the function you **want a s a helper function.** This is demonstrated below –

//This is the definition you want to have

void inOrderHelper(Node\* root, vector<int> &inorder){

if(root==NULL) return;

inOrderHelper(root->left, inorder);// Your code here

inorder.push\_back(root->data);

inOrderHelper(root->right, inorder);

}

//This is the definition system wants to have.

vector<int> inOrder(Node\* root)

{

vector<int> inorder;

inOrderHelper(root, inorder);

return inorder;

}

1. **Queue using STL**

[Read this article](OFFLINE%20SAVED%20Code%20Files,%20Docs,%20&%20Videos%20(used%20as%20reference%20in%20the%20Notes)/Articles/Queue%20in%20Standard%20Template%20Library%20(STL)%20-%20GeeksforGeeks%20-%20Copy.docx)

1. **Stack using STL**

[Read this article](OFFLINE%20SAVED%20Code%20Files,%20Docs,%20&%20Videos%20(used%20as%20reference%20in%20the%20Notes)/Articles/Stack%20in%20Standard%20Template%20Library%20(STL)%20-%20GeeksforGeeks.docx)

1. **Priority\_queue in STL**