***Vectors***

1. **why should I use vector?**

Vector is a template-based container that behaves just like a Dynamic Array. It can expand its memory at run time and always store elements in contiguous memory location just like Array. We can store any type of element in vector by specifying the type as template argument.

**Ordered Collection:** In vector all elements will remain in same order in which they are inserted.

**Provides random access:** Indexing is very fast in vector using opeartor [], just like arrays.

**Performance:** It Performs better if insertion and deletion is in end only and gives worst performance if insertion/deletion is at middle or at starting of vector.

**Contains Copy:** It always stores copy of the object not the same reference. So, if you are adding objects of user defined classes the you should define copy constructor and assignment operator in you class.

# How does vector work internally ?

vector allocates a memory on heap and store all its elements in contiguous memory location. But what if memory it allocated initially is filled?

a.) It will allocate a bigger chunk of memory on heap i.e. almost double the size of previously allocated.  
b.) Then it copies all the elements from old memory location to new one. Yes it copies them, so in case our elements are user defined objects then their copy constructor will be called. Which makes this step quite heavy in terms of speed.  
c.) Then after successful copying it deletes the old memory.

We can check the current capacity of vector i.e. how much elements it can store in current allocated memory using capacity() member function. To check the count of currently stored elements in vector one can use size() member function.

# 3. How to fill a vector with random numbers in C++

We can fill a std::vector with random numbers using std::generate.

template<typename \_FIter, typename \_Generator>

void generate(\_FIter start, \_FIter end, \_Generator gen);

std::generate() copies the elements i.e. it does not push elements. So, it expects that vector is already has the capacity i.e. vector’s size is already n.

We can fill random number by using lambda function or using functors.

// Initialize a vector with 10 ints of value 0

std::vector<int> vecOfRandomNums(10);

// Generate 10 random numbers by lambda func and fill it in vector

std::generate(vecOfRandomNums.begin(), vecOfRandomNums.end(), []() {

return rand() % 100;

});

struct RandomGenerator {

int maxValue;

RandomGenerator(int max) :

maxValue(max) {

}

int operator()() {

return rand() % maxValue;

}

};

struct RandomGenerator {

int maxValue;

RandomGenerator(int max) :

maxValue(max) {

}

int operator()() {

return rand() % maxValue;

}

};

// Generate 10 random numbers by a Functor and fill it in vector

std::generate(vecOfRandomNums.begin(), vecOfRandomNums.end(), RandomGenerator(500));

Here, std::generate iterates the vector from begin to end. During each iteration calls the lambda function or functors and assigns each returned value to corresponding entry in vector.

# Importance of Constructors while using User Defined Objects with std::vector

For User Defined classes if Copy Constructor and Assignment Operator are public then only one can insert it’s object in std::vector.

This is because of two reasons,

* All STL contains always stores the copy of inserted objects not the actual one. So, whenever we insert any element or object in container then it’s copy constructor is called to create a copy and then this copy is inserted in the container.
* While insertion in std::vector it might be possible that storage relocation takes place internally due to insufficient space. In such cases assignment operator will be called on objects inside the container to copy them from one location to another.

std::vector<Sample> vecOfSample(2); //Size is 2

Here 2 objects of Sample will be created using default constructor. But if default constructor is not available then it will give compile error.

std::vector<Sample> vecOfSample;

vecOfSample.reserve(10); //Size is 0 Capacity is 10.

Note: reserve() just increases the capacity of vector not the size.

# How to use vector efficiently in C++?

**1.) Vector will be more efficient if elements are inserted or removed from the backend only: V**ector internally stores all the elements in consecutive memory location. Therefore, if an element is added/erased in middle, then vector right and left respectively shifts all the right-side elements of that location by 1. Also, if elements were user defined objects then copy constructors for all those elements are called.

# ****2.)  Set the storage of vector initially using reserve() member function:****

# reserve() function requests the vector capacity to be at least enough to contain n elements. It only increases the vector’s capacity, size remains same.

**3.)  Instead of adding single element in multiple calls, large set of elements is added in single call:**

Adding single element can cause

* Shifting of some elements in vector.
* Allocation of new memory and movement of all elements on new location.

If we add a single element multiple times than all the above things can happen multiple times. Whereas, if we insert elements in together i.e. in a set than this shifting and copying can happen only once. vector can check if it has the capacity to store **n** elements or not or it needs to shift some elements by **n** location.

# vector and Iterator Invalidation

An Iterator becomes invalidate when the container it points to changes its shape internally i.e. move elements from one location to another and the initial iterator still points to old invalid location.

**Iterator invalidation in vector happens when,**

* An element is inserted to vector at any location
* An element is deleted from vector.

std::vector<int> vecArr;

for (int i = 1; i <= 10; i++)

vecArr.push\_back(i);

// Erase and element with value 5.

auto it = std::find(vecArr.begin(), vecArr.end(), 5);

if (it != vecArr.end())

vecArr.erase(it);

// Now iterator 'it' is invalidated because it still points to

// old location, which has been deleted. So, if you will try to

// do the use the same iterator then it can show undefined

// behavior.

for (; it != vecArr.end(); it++)   // Unpredicted Behavior

std::cout << (\*it) << "  ";          // Unpredicted Behavior

erase() function returns an iterator pointing to the new location of the element that followed the last element erased by the same function. Also, if the element deleted was the last element of the container then it returns the end of the container.

// Erase and element with value 5.

auto it = std::find(vecArr.begin(), vecArr.end(), 5);

if(it != vecArr.end())

   it = vecArr.erase(it);

**Iterator Invalidation Example on Element Insertion in vector:**

When a new element is inserted in vector then it internally shifts its elements and hence the old iterators become invalidated. Reasons for element shift are as follows,

* If element is inserted in between then it shifts all the right elements by 1.
* If the new size of vector is more than its current capacity, then it relocates a bigger chunk of memory and copies all the elements there.

it = vecArr.begin();

    vecArr.insert ( it + 2 , 1 , 200 ); // Insert an element in position 2,

    // Now old iterator it has become invalidated

    // SO, using it as it is can result in undefined behavior

    for(; it != vecArr.end(); it++)   // Undefined Behavior

        std::cout<<(\*it)<<"  ";          // Undefined Behavior

Solution: After calling the insert function update the value of iterator **‘it’**i.e. by re-assigning it.

// Insert an element in position 2,

vecArr.insert ( it + 2, 1 , 200 );

// Reinitialize the invalidated iterator to the begining.

it = vecArr.begin();

# *Use Erase-Remove idiom for removing elements from vector.*

# Let’s say vector contain following numbers 1,2,5,4,5,1,5,7,8,9. Now we want to delete all the occurrences of 5 from it, so that vector contents should become – 1 2 4 1 7 8 9 .

std::remove transforms the given range into a range with all the elements that compare not equal to given element shifted to the start of the container. So, don’t remove the matched elements. It just shifted the non-matched to starting and gives an iterator to new valid end. It just requires O(n) complexity.

Output of remove algorithm will be : 1 2 4 1 7 8 9 ? ? ?

Now use vector’s erase function to delete elements from new end to old end of vector. It requires O(1) time.

vec.erase(std::remove(vec.begin(), vec.end(), elem), vec.end());

vec.erase(std::remove\_if(vec.begin(), vec.end(), predicate), vec.end());

# Be careful with hidden cost of std::vector for user defined objects.

class Item {

public:

static int m\_ConstructorCalledCount;

static int m\_DestCalledCount;

static int m\_CopyConstructorCalledCount;

Item() {

m\_ConstructorCalledCount++;

}

~Item() {

m\_DestCalledCount++;

}

Item(const Item& obj) {

m\_CopyConstructorCalledCount++;

}

};

int Item::m\_ConstructorCalledCount = 0;

int Item::m\_CopyConstructorCalledCount = 0;

int Item::m\_DestCalledCount = 0;

And we want to create a vector of 10000 Item objects. So, let’s create a factory class for it:

class ItemFactory{

public:

static std::vector<Item> getItemObjects(int count){

std::vector<Item> vecOfItems;

vecOfItems.reserve(count);

for (int var = 0; var < count; ++var) {

vecOfItems.push\_back(Item());

}

return vecOfItems;

}

};

int count = 10000;

std::vector<Item> vecOfItems;

vecOfItems = ItemFactory::getItemObjects(count);

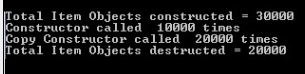
std::cout << "Total Item Objects constructed = " << (Item::m\_ConstructorCalledCount + Item::m\_CopyConstructorCalledCount);

std::cout << "Constructor called " << Item::m\_ConstructorCalledCount << "times";

std::cout << "Copy Constructor called " << Item::m\_CopyConstructorCalledCount;

std::cout << "Total Item Objects destructed = " << Item::m\_DestCalledCount;

Above code seems fine, we created 10000 objects of class Item. But while creating these 10000 objects we wasted  20000 objects, that’s double of what we needed.



# Culprit in this case is the getItemObjects function from ItemFactory class:

* Inside the for loop we created 10000 objects, so constructor is called 10000 times.
* Then after creating every object we inserted the newly created object in vector 10000 times, therefore copy constructor is called 10000 times and destructor of old 10000 Item object is called.
* In last line of function, we returned the vector and all its content was copied to vecOfItems vector, so again 10000 times copy constructor is called and destructor of old 10000 Item object is called.

Now with a small change we can reduce the wasted object count to 10000 from 20000.

# std::vector<Item> vecOfItems = ItemFactory::getItemObjects(count);

# With this, instead of copying 10000 objects, all objects will be moved to new vector. So, now output will be:

# 

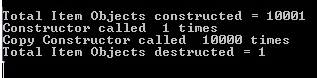
# But still we are wasting 10000 objects. How to fix that?

# Instead of returning the whole new vector from factory function, just passing the new vector as a reference to factory function

static void getItemObjects(std::vector<Item> & vecItems, int count){

vecItems.assign(count, Item());

}



1. Just use the vector’s assign function to create the 10000 copies of 1 object i.e.

static std::vector<Item> getItemObjects( int count){

std::vector<Item> vecOfItems;

vecOfItems.assign(count, Item());

return vecOfItems;

}



# Map Tutorial Part 1: Usage Detail with examples

Map is an associative container that store elements in key-value pair.

## **Benefits of using std::map :**

* It stores only unique keys and that too in sorted order based on its assigned sorting criteria.
* As keys are in sorted order therefore searching element in map through key is very fast i.e. it takes logarithmic time.
* In std::map there will be only one value attached with the every key.
* std::map can be used as associative arrays.
* It might be implemented using balanced binary trees.

## **Different between operator [] and insert function:**

If specified key already existed in map, then operator [] will silently change its value whereas insert will not replace already added key instead it returns the information i.e. if element is added or not. e.g.

# Using User defined class objects as keys in std::map

We have two options:

* **Overloading operator < for sorting of keys :** Default sorting criteria i.e. operator < defined for our Class.

class User

{

std::string m\_id;

std::string m\_name;

public:

User(std::string name, std::string id)

:m\_id(id), m\_name(name)

{}

const std::string& getId() const {

return m\_id;

}

const std::string& getName() const {

return m\_name;

}

bool operator< (const User& userObj) const { //Overloaded Operator.

if (userObj.m\_id < this->m\_id)

return true;

}

## };

std::map<User, int> m\_UserInfoMap;

    m\_UserInfoMap.insert(std::make\_pair<User, int>(User("Mr.X", "3"), 100) );

    m\_UserInfoMap.insert(std::make\_pair<User, int>(User("Mr.X", "1"), 120) );

    m\_UserInfoMap.insert(std::make\_pair<User, int>(User("Mr.Z", "2"), 300) );

*Note: I usually prefer custom comparator instead of that.*

* **Using Comparator for sorting of keys: M**ap should be assigned with an external sorting criterion i.e. comparator that can compare two objects of your user defined class.

struct UserNameComparator: public binary\_function< User, User,bool>{

bool operator()(const User & left, const User & right) const {

return (left.getName() > right.getName());

}

};

std::map<User, int, UserNameComparator> m\_UserInfoMap;

m\_UserInfoMap.insert(std::make\_pair<User, int>(User("Mr.X", "3"), 100));

m\_UserInfoMap.insert(std::make\_pair<User, int>(User("Mr.X", "1"), 120));

m\_UserInfoMap.insert(std::make\_pair<User, int>(User("Mr.Z", "2"), 300));

# Set vs Map : How to choose a right associative container ?

**Set :**

* Set is an associative container which we need to store unique elements.
* It always keeps the elements in sorted order.
* Internally it maintains a balanced binary search tree of elements. Therefore, when we search an element inside the set then it takes only log(n) complexity to search it.

**Important Point about set – Once added then cannot change i.e.**

Each element added inside the set is const i.e. you cannot modify that element, because if you could, then it would hamper set’s internal data structure i.e. it will lose its internal balanced binary search tree property and results in undefined behavior.

*But what if want to associate some Value with this key and want to change that associated value at run time. Then we need something other than SET and that’s it MAP.*

**Map:**

* Map is an associative container that is used to store key-value pair of elements with unique keys.
* It always keeps the inserted pairs in sorted order based on the key.
* Internally it maintains a balanced binary search tree to store keys. Therefore, when searching key inside the map takes only log(n) complexity.
* We cannot modify the key of any inserted pair in map.
* We can modify the value associated with a key in any inserted pair in map.

**When to choose SET and when MAP?**

So, if you want to maintain a data structure of unique keys only without any associated value that plan to modify in future then use set. If you want to modify any element in set, then erase it and then insert the new one. Whereas, use map if you want to maintain a data structure of unique keys and some associated value with each key that you want to change in future.

Unordered\_map

Unordered\_map provides a functionality of map i.e. it stores the elements in key value pair and with unique key only.

Unordered\_map internally uses the hashing to achieve this. It internally uses a hash table to implement this hashing feature. In an unordered\_map elements are stored in a key value pair combination. But elements are stored in arbitrary order unlike associative containers where elements were stored in sorted order of keys.

string str = "Rajeev Kumar Sharma";

map<char, int> mp;

unordered\_map<char, int> um;

for (char c : str) mp[c]++; //Count number of char in str.

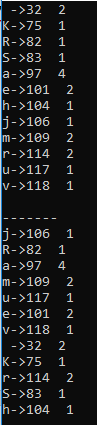
for (char c : str) um[c]++; //Count number of char in str.

for (auto &iter : mp) cout << iter.first <<"->" << charToShort(iter.first)

<< " " << iter.second << endl;

for (auto &iter : um) cout << iter.first <<"->" <<charToShort(iter.first)

<< " " << iter.second << endl;



Here we can clearly see the oput of Map in sorted order while

Unordered\_map in random order.

| map | unordered\_map

---------------------------------------------------------

Ordering | increasing order | no ordering

| (by default) |

Implementation | Self balancing BST | Hash Table

| like R & B tree |

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

## How unordered\_map store elements?

Whenever we try to insert an element in a unordered\_map, it internally does the following steps:

* First hash of key is calculated using Hasher function and then based on that hash an appropriate bucket is choose.
* Once bucket is identified then it compares the key with key of each element inside the bucket using Comparator function to identify if given element is a duplicate or not.
* If it’s not a duplicate, then only it stores the element in that bucket.

Insert, find and delete into unordered\_map

//Diffrence between operator[] and insert method.

//operator[] perform insert & update operation based on key

//While MAP::insert() perform only insert. Refuse to insert or

//update when key available in container.

typedef map<char, int> CharMapType;

map<char, int> charMap;

pair<map<char, int>::iterator, bool> isInsert;

charMap['A'] = 65; //Key Not found, insert new node

charMap['B'] = 66; //Key Not found, insert new node

charMap['C'] = 67; //Key Not found, insert new node

//charMap['A'] = 100; //Key found, Update the value only.

isInsert = charMap.insert({ 'A',100 }); //Refuse to update value.

//OR, isInsert = charMap.insert(CharMapType::value\_type({ 'A',100 }));

**Like map, unordered\_map has same behavior to insert, find and erase.**

typedef std::unordered\_map<std::string, int>::iterator UOMIterator;

// Pair of Map Iterator and bool value

std::pair< UOMIterator, bool> result;

// Inserting an element through pair

result = wordMap.insert(std::make\_pair<std::string, int>("Second", 6));

//result=wordMap.insert({“Second”,6});

if (result.second == false)

cout<<”Element not inserted\n”

unordered\_map<char,int>::iterator it=um.find('R');

if (it != um.end()) {

cout << "Element found\n";

}

int &x = charMap['A']; //If Key ‘A’ is available in map, return its

x = 100; //value reference, we can also change its value.

//If key not found then create

**Erasing via key and iterator.**

*// Initialize an unordered\_map through initializer\_list*

std::unordered\_map<std::**string**, **int**> wordMap( { { "First", 1 }, { "Second",

2 }, { "Third", 3 }, { "Fourth", 4 }, { "Fifth", 5 } });

*//Erase element by key*

**if** (wordMap.erase("Second") == 1)

std::**cout** << "Element Deleted" << std::endl;

*// Iterator pointing to first element of unordered\_map*

std::unordered\_map<std::**string**, **int**>::iterator it = wordMap.find("Fourth");

*// Erase the element pointed by iterator it*

**if** (it != wordMap.**end**())

wordMap.erase(it);

**Erase while iterating**

// Erase all element whose key starts with letter 'F' in an iteration

while (it != wordMap.end()) {

// Check if key's first character is F

if (it->first[0] == 'F') {

// erase() function returns the iterator of the next

// to last deleted element.

it=wordMap.erase(it); // Or wordMap.erase(it++)

}

else

it++;

}

**Map vs unordered\_map | When to choose one over another ?**

Both std::map & std::unordered\_map store elements in key value pair & provide member functions to efficiently insert, search & delete key value pairs, but they are different in following areas:

* **Internal Implementation:**
* **Memory Usage:** Memory usage is more in unordered\_map as compared to map because unordered\_map need space for storing hash table too.
* **Time Complexity:** Time complexity for searching elements in **std::map** is

O(log n). Even in worst case it will be O(log n) because elements are stored internally as Balanced Binary Search tree (BST).

Whereas, in **std::unordered\_map** best case time complexity for searching is O(1).

If hash code function is not good then, worst case complexity can be O(n) (In case

all keys are in same bucket).

* **Using user defined objects as keys:**

For std::map to use user defined object as keys, we need to override either < operator or pass external comparator i.e. a functors or function pointer that can be used by map for comparing keys. Whereas, For **std::unordered\_map** we need to provide definition of function std::hash<K> for our key type K. Also, we need to override == operator.

**std::map** Internally store elements in a balanced BST. Therefore, elements will be stored in sorted order of keys.

**std::unordered\_map** store elements using hash table. Therefore, elements will not be stored in any sorted order. They will be stored in arbitrary order.

**When to choose map instead of unordered\_map**

* **When we need Low Memory:** Unordered\_map consumes extra memory for internal hashing, so if you are keeping millions and billions of data inside the map and want to consume less memory then choose std::map instead of std::unordered\_map.
* **When we are interested in Ordering too:**
* **When you need guaranteed Performance:** For searching an element, unordered\_map gives the complexity O(1) in best case but O(n) in worst case (if hash implementation is not perfect).

**When to choose unordered\_map instead of map**

* **When we have good hasher and no memory limitation:** Unordered\_map consumes extra memory for internal hashing. But to due to this searching complexity is O(1), if hasher function is good.

unordered\_set

**std::unordered\_set** is an STL container and its introduced in C++11. It provides a functionality of a Set i.e. it can contain the unique elements only. unordered\_set stores the elements internally using a [Hash Table](https://thispointer.com/what-is-hashing-and-hash-table/).

std::unordered\_set<T> setOfTypeT;

Elements of Type with which an unoredered\_set is declared can only be added in it. Also, all elements of Type T should be:

* Copiable or Movable
* Comparable

Why because of the element added in unordered\_set will act as both key and value for internal Hash Table. As it’s a key therefore it should be comparable and copy able.

Also, all elements in the **unordered\_set** are **immutable** i.e. they cannot be modified once inserted. They can only be deleted after the insertion.

std::unordered\_set<std::string> setOfStrs; // Creating an Unoredered\_set of type string

setOfStrs.insert("First"); // Insert strings to the set

setOfStrs.insert("second");

setOfStrs.insert("third");

  setOfStrs.insert("second"); // Try to Insert a duplicate string in set

// Iterate Over the Unordered Set and display it

for (std::string s : setOfStrs)

std::cout << s << std::endl;

Output:

third

second

First

Initialization of unoredered\_set:

int arr[] = { 2, 4, 6, 1, 3, 6, 8, 3, 2 };

// Create an unoredered set and initialize it with the array. Set will

// contain only unique elements

std::unordered\_set<int> setOfNum(arr, arr + sizeof(arr) / sizeof(int));

// Create an unoredered set and initialize it initializer\_list, Set will // contain only unique elements

std::unordered\_set<int> setOfNum2( { 1, 2, 3, 1, 3, 4, 2 });

std::vector<int> vec( { 14, 5, 6, 7 });

// Create an unoredered set and initialize it with vector

std::unordered\_set<int> setOfNum3(vec.begin(), vec.end());

## Inserting element in unordered\_set by value

std::unordered\_set provides the different overloaded versions of insert() member function to insert elements:

pair<iterator,bool> insert ( const value\_type& val );

void insert ( InputIterator first, InputIterator last );

void insert ( initializer\_list<value\_type> il );

**Insert via pair object. What does this pair contain?**

If an element equivalent to passed **element is not present in the set,** then new element will be inserted successfully and it will return the pair with following values,

* Iterator will point to newly inserted element
* Bool flag will be true.

If an element equivalent to**passed element is already present in set**, then new element will be not be inserted in set and it will return the pair with following values.

* Iterator will point to the position of first matched equivalent element.
* Bool flag will be false.

// Declare a pair of iterator and bool flag

std::pair<std::unordered\_set<int>::iterator, bool > result;

// Try to insert a Duplicate Element, it will return a pair of iterators

// and bool

result = setOfNum.insert(6);

if(result.second == false) // Check if element inserted or not

std::cout<<"Failed to Insert 6"<<std::endl;

**How to search, if an element is present in an unordered\_set or not?**

std::unordered\_set provides a member function find () that accepts an element of same type as set and search for that element in set. If element is found, then it returns the iterator pointing to that element else it returns an iterator pointing to end of set.

std::unordered\_set<int> setOfNum( { 1, 2, 3, 1, 3, 4, 2 });

std::unordered\_set<int>::const\_iterator it = setOfNum.find(4);

if (it != setOfNum.end())

std::cout << "4 exist in the set" << std::endl;

it = setOfNum.find(9);

if (it == setOfNum.end())

std::cout << "9 dont exist in the set" << std::endl;

Using unordered\_set with custom hasher and comparison function

Unordered\_set uses the Hash table implementation to provide the set functionality. Whenever we insert an element in unordered\_set two things happen:

* It calls the hasher function on passed element and compute the hash code. Now based on this hash code it selects the appropriate bucket for the element.
* Then compares the given element with all the elements in the bucket to check if any similar element is already present. If not, then it stores the element in that bucket.

Therefore, if two elements are equal then there hash code should always be same. Otherwise it will impossible to search for correct element in unordered\_set.

While declaring an unordered\_set we provides the type of element that can be stored in set. Along with that we can also provide the type of custom hasher and comparison functions.

Now, suppose our unordered\_set is of type T and didn’t supplied any default custom hasher function and comparison function. Then in that case default hasher and comparison function will be used i.e.

* std::hash<T>()
* std::equal\_to<T>

For example, if create a unordered\_set of std::string:

std::unordered\_set<std::string> setOfStrs;

Above unordered\_set uses the default hasher and comparison function and it is equivalent to:

unordered\_set<string, std::hash<string>,std::equal\_to<string> > setOfStrs;

std::hash calculates the hash of primitive data types and std::equals\_to internally calls == function on the passed elements for comparison. Let’s see an example of  custom hasher and comparison function.

Custom Hasher and Comparator for unordered\_set of unique length strings

Suppose, we want to create an std::unordered\_set of std::string and want to store strings of unique lengths only. As, the default equals\_to<>() function uses the == operator to compare the elements. But in our scenario, we want to compare elements based on length i.e. two elements should be considered equal if they have the same length. For that we need to create our custom Comparison Function i.e.

// Custom comparator that compares the string objects by length

struct StringEqualBySize {

public:

bool operator()(const std::string & str1, const std::string & str2) const{

if (str1.length() == str2.length())

return true;

else return false;

}};

For example, according to above compare function “abc” and “def” are equal because their length is same. But their hash code can be different if computed with default std::hash<std::string>. Therefore, we need custom hasher function, so that elements which are equal based on above compare function should have same hash code.

// Custom Hash Functor that will compute the hash on the passed string

// objects length

struct StringHashBySize {

public:

size\_t operator()(const std::string & str) const {

int size = str.length();

return std::hash<int>()(size);

}

};

Declaring unordered\_set with custom hasher and compare function:

// Declaring unordered\_set with Custom Hash Function and comparator

unordered\_set<std::string, StringHashBySize, StringEqualBySize> setOfStrs;

# How to use Unordered\_set with User defined classes

std::unordered\_set uses the hasher and comparison function. For primitive data types like int, string etc. default hasher and comparison functions will work i.e.

* std::hash<T>()
* std::equal\_to<T>

But for User defined Objects, these default implementations will not work. We need to provide some extra to make this default functions work.

There are 2 ways to make an unordered\_set of User Define Types / Classes i.e.

* Create special functions to make default std::hash<> & std::equals\_to<> functions to work with User Defined classes
* Creating Custom Hasher and Comparison Functors and pass it to unordered\_set.

Let’s see them one by one. But, first create a Student class i.e.

class Student{

int mId;

std::string mName;

public:

Student(int id, std::string name) :mId(id), mName(name){}

void displayInfo(){

std::cout << mId << " :: " << mName << std::endl;

}

bool operator ==(const Student & obj) const {

if (mId == obj.mId) return true;

else return false;

}

Int getId() const {return mId;}

std::string getName() const {

return mName;

}

# };

## Unordered\_set of User Defined Class with default Hasher & Comparison Function

Let’s create an unordered\_set of type Student that should store unique Student objects based on ID.

// Declaring unordered\_set of Student

std::unordered\_set<Student> setOfStudents;

Here, we have not provided any Custom Hasher or Comparison function hence default hasher and comparison function will be used.

**Default Hasher Function for class Student**

std::hash<Student>(const Student & obj);

**Default Comparison Function for class Student**

std::equals\_to<Student>(const Student & obj1,const Student & obj2 );

This will internally class the == operator() function. So, to make this unordered\_set<Student> work, we need to implement the above hasher and == operator for class Student.

**Implementing std::hash<Student>**

namespace std{

template<>

struct hash<Student>{

size\_t operator()(const Student & obj) const {

return hash<int>()(obj.getId());

}

};

}

It will compute the hash on only member variable ID for an object of class Student.

**Implementing == operator for class Student,**

This will compare the ID only to check if two objects of class Student are equal or not i.e.

bool operator ==(const Student & obj) const{

if (mId == obj.mId) return true;

else return false;

}

Now we can use the std::unordered\_set of type Student with default hasher and comparator i.e.

std::unordered\_set<Student> setOfStudents;

// Inserting elements

setOfStudents.insert(Student(11, "John"));

setOfStudents.insert(Student(12, "Harry"));

setOfStudents.insert(Student(13, "Ritz"));

setOfStudents.insert(Student(14, "John"));

// Trying to insert with duplicate ID

// It will not be inserted

auto res = setOfStudents.insert(Student(12, "Varun"));

if (res.second == false)

std::cout << "Failed to insert with ID 12" << std::endl;

The above Unordered Set will store elements with unique size only. Now, suppose a new requirement comes and we also want to store objects in a new unoredered\_set with unique names instead of unique ID. How to do that?

## unordered\_set of User Defined Class with Custom Hasher and Comparator

As, now we need to create a new unordered\_set of type Student that can store unique Student objects based on name instead of ID, so we cannot use the above implementation of std::hash<Student> and operator==.

So, to achieve this we can create custom Hasher and Comparator.

**Custom Hasher for class Student**

It will compute the hash based on name of Student:

// Custom Hash Functor that will compute the hash on the

// passed string objects length

struct StudentHasher{

size\_t operator()(const Student & obj) const{

return std::hash<std::string>()(obj.getName());

}

};

**Custom Comparator for class Student,**

It will compare the two Student objects based on name instead of ID i.e

// Custom comparator that compares the string objects by length

struct StudentComparator{

bool operator()(const Student & obj1, const Student & obj2) const{

if (obj1.getName() == obj2.getName())

return true;

return false;

}

};

// Declaring unordered\_set of Student

std::unordered\_set<Student, StudentHasher, StudentComparator> setOfStudByName;

***Hashing with Chaining Code.***

long HashCode(std::string text) {

long hash = 0;

size\_t strlen = text.length(), i;

char character;

if (strlen == 0)

return hash;

for (i = 0; i < strlen; i++) {

character = text.at(i);

hash = (31 \* hash) + static\_cast<short>(character);

}

return hash;

}

template<typename K,typename V>

class HashNode{

public:

K m\_key;

V m\_value;

HashNode<K, V> \*m\_pNext; // Reference to next node

HashNode(K key, V value) :m\_key(key), m\_value(value),m\_pNext(NULL) {}

};

template<typename K, typename V>

class HashTable {

vector< HashNode<K, V>\* > bucketArray; // bucketArray is used to store array

// of chains

int numBuckets; // Current capacity of array list(bucketArray)

int size; // Current size of array list

// This implements hash function to find index for a key

int getBucketIndex(K key) {

long hashCode = HashCode(to\_string(key));

int index = hashCode % numBuckets;

return index;

}

public:

HashTable(const int sz) {

numBuckets = sz;

size = 0;

//bucketArray.reserve(sz);

for (int i = 0; i < numBuckets; i++) { // Create empty chains

bucketArray.push\_back(NULL);

}

}

~HashTable() {

for (int i = 0; i < numBuckets; i++) {

HashNode<K, V> \*head = bucketArray[i];

while (head) {

HashNode<K, V> \*tmp = head;

cout << "index:: " << i <<"Key:: "<< head->m\_key << endl;

head = head->m\_pNext;

delete tmp;

}

}

}

int getSize() { return size; }

bool isEmpty() { return bucketArray->empty(); }

void insert(pair<K, V> &tupple) {

int index = getBucketIndex(tupple.first);

HashNode<K, V> \*walker = bucketArray[index];

if (walker == NULL) {

walker = new HashNode<K, V>(tupple.first, tupple.second);

size++;

bucketArray[index] = walker;

}

else {

while (walker->m\_pNext) { // If Key is already present

if ((walker)->m\_key == tupple.first) {

(walker)->m\_value = tupple.second; // Only update the value of existing key and return

return;

}

walker = (walker)->m\_pNext; // Otherwise point to the last of list.

}

size++;

walker->m\_pNext = new HashNode<K, V>(tupple.first, tupple.second);

}

// load\_factor processing. If load factor goes beyond threshold, then

// double hash table size.

if ((1.0\*size) / numBuckets >= 0.7){

bucketArray.resize(numBuckets \* 2);

numBuckets = numBuckets \* 2;

/\*

vector< HashNode<K, V>\* > tmp; // Creates the tmp Vector of array

for (int i = 0; i < numBuckets\*2; i++) { // Creates empty chains.

tmp.push\_back(NULL);

}

for (int i = 0; i < numBuckets; i++) {

if( bucketArray[i] )

tmp[i] = bucketArray[i];

}

bucketArray = tmp;

numBuckets = numBuckets \* 2;

\*/

}

}

void displayHashTable() {

for (int i = 0; i < numBuckets; i++) {

cout << "[" << i <<"]:: ";

HashNode<K, V> \*head = bucketArray[i];

while(head) {

cout<<head->m\_key<< " --> "<<head->m\_value.c\_str()<<"--> ";

head = head->m\_pNext;

}

cout << endl;

}

}

V getElement(const K key) {

int index = getBucketIndex(key);

HashNode<K, V> \*head = bucketArray[index];

while (head){

if (head->m\_key == key) {

return head->m\_value;

}

head = head->m\_pNext;

}

return NULL;

}

bool removeElement(const K key) {

int index = getBucketIndex(key);

HashNode<K, V> \*head = bucketArray[index];

if (head == NULL)

return false;

else {

if (head->m\_key == key) { //key found at begning the of the list.

bucketArray[index] = head->m\_pNext; //update the base

//address.

delete head; //Delete node

size--; //decrement Hash table size

return true; //return success.

}

while (head->m\_pNext) { //Key found at mid or end of the list

HashNode<K, V> \*prev = head;//keep track of prev node

head = head->m\_pNext; //Advance the ptr, because we

//already checked it.

if (head->m\_key == key) {

if (head->m\_pNext) { //Key found but not last node.

prev->m\_pNext = head->m\_pNext;

delete head;

}

else { //Key found but last node.

prev->m\_pNext = NULL;

delete head;

}

size--;

return true;

}

}

}

return false;

}

};

int main() {

HashTable<int, std::string> hTbl(11);

hTbl.insert(make\_pair<int, std::string>(123122, "CCU"));

hTbl.insert(make\_pair<int, std::string>(123123, "DDH"));

hTbl.insert(make\_pair<int, std::string>(123129, "RNC"));

hTbl.insert(make\_pair<int, std::string>(123772, "NDLS"));

hTbl.insert(make\_pair<int, std::string>(123984, "BLR"));

hTbl.insert(make\_pair<int, std::string>(186120, "PAT"));

hTbl.insert(make\_pair<int, std::string>(146126, "HYD"));

hTbl.insert(make\_pair<int, std::string>(145425, "KAN"));

hTbl.insert(make\_pair<int, std::string>(733122, "DDN"));

hTbl.displayHashTable();

cout << "----------------------------------------------\n";

//hTbl.removeElement(123984);

//hTbl.removeElement(123122);

//hTbl.removeElement(733122);

cout << hTbl.getElement(733122).c\_str();

//hTbl.displayHashTable();

}