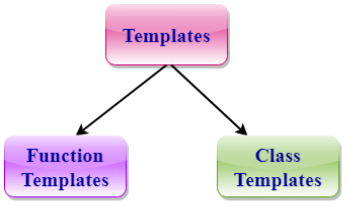
C++ Templates 2
    How Do Templates Work? 2
    Function Templates 3
    Class Templates 3
Standard Template Library (STL) 4
    Containers 4
    Iterators 5
    Algorithms 6
    Functors / function object 10
Array 11
Vectors 13
Stack 17
Queues 18
Pair 18
tuple 20
signed and unsigned 22
bitset 23
type casting 25
lambda 27
List 28
Forward List 29
Deque (Double-ended queues) 31
map 32
Red-Black Tree (self-balancing Binary Search tree) 34
    AVL tree and RED BLACK TREE 34
    Is every AVL tree can be a Red-Black tree? 35
    some rules used to create the Red-Black tree: 35
unordered_map 37
Hashing Method 39
    Separate Chaining: 41
    Open Addressing: 42
    Quadratic Probing 43
    Double Hashing 43
Types of Hash functions: 44
Sets 44
C++ Namespaces 46

C++ Templates

It allows you to define the generic classes and generic functions .

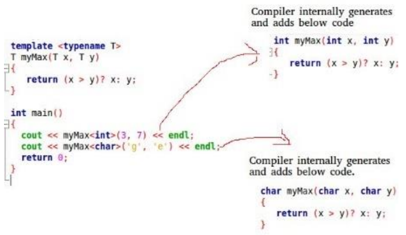
Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types.



How Do Templates Work?

Templates are expanded at compiler time. The difference is, that the compiler does type-checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.



```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

```
#include<iostream>
using namespace std;
template<class t>
t sum(t a,t b){
   return a + b;
};
int main()
{  cout<<sum(10,10)  <<endl;
   cout<<sum(10.5,10.5)
<<endl;
   cout<<sum('a','a') <<endl;
} output:
  20
  21
  T
```

```
#include<iostream>
using namespace std;
template<class t>
t sum(t a){
   return a + 1;
};
int main()
{
   cout<<sum("a")
<<endl;
 return 0;
}
```

```
#include<iostream>
using namespace std;
template<class t1 ,class
t2>
t1 sum(t1 a , t2 b){
   return a + b;
};
int main()
{
   cout<<sum(4,4.5)
<<endl;
 return 0;
}

output: 8
```

```
#include<iostream>
using namespace std;
template<class t1 ,class
t2>
t1 sum(t1 a , t2 b){
   return a + b;
};
int main()
{
   cout<<sum(4.5,4)
<<endl;
 return 0;
}
output: 9
```

```
#include<iostream>
using namespace std;
template<class t1 ,class t2>
t2 sum(t1 a , t2 b){
   return a + b;
};
int main()
{
   cout<<sum(4.5,4) <<endl;
 return 0;
}
output : 8
```

default argument

```
#include<iostream>
using namespace std;
template<class t1=int ,class t2>
t2 sum(t1 a , t2 b){
   return a + b;
};
int main()
{
   cout<<sum(4.5,4) <<endl;
 return 0;
}
output : 8
```

Function Templates

We write a generic function that can be used for different data types.
Examples of function templates are sort(), max(), min(), printArray().

Class Templates

Class templates like function templates, class templates are useful when a class defines something that is independent of the data type.
Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

#include<iostream>
using namespace std;
template<class t>
class a{
   t a,b;
public:
t show(t x ,t y){
   a=x;
   b=y;
   return (a+b); };
};
int main()
{a <float>obj;
cout<< obj.show(10.5,10.5)<<endl ;
//21
 return 0;
}
	by defalut , in vs code defalut template is not running
	#include<iostream>
	using namespace std;
	template <class t=float >
	class a{
	   t a,b;
	public:
	t show(t x ,t y)
	{ a=x;
	   b=y;
	   return (a+b);
	    } };
	int main()
	{a obj;
	cout<< obj.show(10.5,10.5)<<endl ;
	}
		#include<iostream>
		using namespace std;
		template<class r>
		class boss {
		   public:
		   r show(r a){
		     return a+1; } };
		template<class t>
		class empl :public boss<t> {
		   public:
		   t get(t a){
		     return a +1;
		   } };
		int main()
		{empl <int>obj;
		cout<<obj.show(101) <<endl;
		//102
		cout<<obj.get(101); //102
		}	

```
#include<iostream>
using namespace std;
template<class r>
class boss {  public:
                    r show(r a){
                    return a+1;  }
r fun(){ r a=2;
        return a+2; }   };
template<class t>
class empl :public boss<t> {
  using boss<t>::fun;
   public:
   t get(t a){

      cout<<"boss : " <<fun() <<endl;
     return a +1;
   }
};
int main()
{empl <int>obj;
cout<<obj.get(101) <<endl;  }
output :
boss : 4
102
```

What is the difference between function overloading and templates?
Both function overloading and templates are examples of polymorphism features of OOP. Function overloading is used when multiple functions do quite similar (not identical) operations, templates are used when multiple functions do identical operations.

Standard Template Library (STL)

- The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs.
- The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.
- One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types.
- This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type.

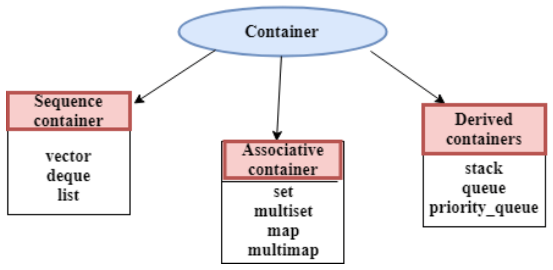
STL has 4 components:

1. Algorithms
2. Containers
3. Functors
4. Iterators

Containers

Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures for example arrays, list, trees, etc.

Container	Description	Header file	iterator
vector	vector is a class that creates a dynamic array allowing insertions and deletions at the back.	<vector>	Random access
list	list is the sequence containers that allow the insertions and deletions from anywhere.	<list>	Bidirectional
deque	deque is the double ended queue that allows the insertion and deletion from both the ends.	<deque>	Random access
set	set is an associate container for storing unique sets.	<set>	Bidirectional
multiset	Multiset is an associate container for storing non- unique sets.	<set>	Bidirectional
map	Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping).	<map>	Bidirectional
multimap	multimap is an associate container for storing key- value pair, and each key can be associated with more than one value.	<map>	Bidirectional
stack	It follows last in first out(LIFO).	<stack>	No iterator
queue	It follows first in first out(FIFO).	<queue>	No iterator
Priority-queue	First element out is always the highest priority element.	<queue>	No iterator



Iterators

- Iterators are used to point at the memory addresses of STL containers.
- They are primarily used in sequences of numbers, characters etc. They reduce the complexity and execution time of the program.
- Iterators are pointer-like entities used to access the individual elements in a container.
- Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.

Operations of iterators :-

1. begin() :- This function is used to return the beginning position of the container.
2. end() :- This function is used to return the after end position of the container.
3. advance() :- This function is used to increment the iterator position till the specified number mentioned in its arguments.
4. next() :- This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.
5. prev() **:-** This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.
6. 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.

Iterator Categories

Iterators are mainly divided into five categories:

1.Input iterator:

1. An Input iterator is an iterator that allows the program to read the values from the container.
2. Dereferencing the input iterator allows us to read a value from the container, but it does not alter the value.
3. An Input iterator is a one way iterator.
4. An Input iterator can be incremented, but it cannot be decremented.
5.

2.Output iterator:

1. An output iterator is similar to the input iterator, except that it allows the program to modify a value of the container, but it does not allow to read it.
2. It is a one-way iterator.
3. It is a write only iterator.

3.Forward iterator:

1. Forward iterator uses the ++ operator to navigate through the container.
2. Forward iterator goes through each element of a container and one element at a time

4.Bidirectional iterator:

1. A Bidirectional iterator is similar to the forward iterator, except that it also moves in the backward direction.
2. It is a two way iterator.
3. It can be incremented as well as decremented.

5.Random Access Iterator:

1. Random access iterator can be used to access the random element of a container.
2. Random access iterator has all the features of a bidirectional iterator, and it also has one more additional feature, i.e., pointer addition. By using the pointer addition operation, we can access the random element of a container.

Algorithms

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

STL algorithms can be categorized

1.Nonmutating algorithms:

Nonmutating algorithms are the algorithms that do not alter any value of a container object nor do they change the order of the elements in which they appear. These algorithms can be used for all the container objects, and they make use of the forward iterators.

2.Mutating algorithms:

Mutating algorithms are the algorithms that can be used to alter the value of a container. They can also be used to change the order of the elements in which they appear.

3.Sorting algorithms:

Sorting algorithms are the modifying algorithms used to sort the elements in a container.

4.Set algorithms:

Set algorithms are also known as sorted range algorithm. This algorithm is used to perform some function on a container that greatly improves the efficiency of a program.

5.Relational algorithms:

Relational algorithms are the algorithms used to work on the numerical data. They are mainly designed to perform the mathematical operations to all the elements in a container.

Non-Manipulating Algorithms

1. sort(first_iterator, last_iterator) – To sort the given vector.
2. sort(first_iterator, last_iterator, greater<int>()) – To sort the given container/vector in descending order
3. reverse(first_iterator, last_iterator) – To reverse a vector. (if ascending -> descending OR if descending -> ascending)
4. max_element (first_iterator, last_iterator) – To find the maximum element of a vector.
5. min_element (first_iterator, last_iterator) – To find the minimum element of a vector.
6. accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation of vector elements
7. count(first_iterator, last_iterator,x) – To count the occurrences of x in vector.
8. find(first_iterator, last_iterator, x) – Returns an iterator to the first occurrence of x in vector and points to last address of vector ((name_of_vector).end()) if element is not present in vector.
9. binary_search(first_iterator, last_iterator, x) – Tests whether x exists in sorted vector or not.
10. 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'.
11. 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

1. arr.erase(position to be deleted) – This erases selected element in vector and shifts and resizes the vector elements accordingly.
2. arr.erase(unique(arr.begin(),arr.end()),arr.end()) – This erases the duplicate occurrences in sorted vector in a single line.
3. next_permutation(first_iterator, last_iterator) – This modified the vector to its next permutation.
4. prev_permutation(first_iterator, last_iterator) – This modified the vector to its previous permutation.

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.

Non-modifying sequence operations

1. all_of : Test condition on all elements in range
2. any_of : Test if any element in range fulfills condition
3. none_of : Test if no elements fulfill condition
4. for_each : Apply function to range
5. find : Find value in range
6. find_if : Find element in range
7. find_if_not : Find element in range (negative condition)
8. find_end : Find last subsequence in range
9. find_first_of : Find element from set in range
10. adjacent_find : Find equal adjacent elements in range
11. count : Count appearances of value in range
12. count_if : Return number of elements in range satisfying condition
13. mismatch : Return first position where two ranges differ
14. equal : Test whether the elements in two ranges are equal
15. is_permutation : Test whether range is permutation of another
16. search : Search range for subsequence
17. search_n : Search range for element

Modifying sequence operations

1. copy : Copy range of elements
2. copy_n : Copy elements
3. copy_if : Copy certain elements of range
4. copy_backward : Copy range of elements backward
5. move : Move range of elements
6. move_backward : Move range of elements backward
7. swap : Exchange values of two objects
8. swap_ranges : Exchange values of two ranges
9. iter_swap : Exchange values of objects pointed to by two iterators
10. transform : Transform range
11. replace : Replace value in range
12. replace_if : Replace values in range
13. replace_copy : Copy range replacing value
14. replace_copy_if : Copy range replacing value
15. fill : Fill range with value
16. fill_n : Fill sequence with value
17. generate : Generate values for range with function
18. generate_n : Generate values for sequence with function
19. remove : Remove value from range
20. remove_if : Remove elements from range

21. remove_copy : Copy range removing value
22. remove_copy_if : Copy range removing values
23. unique : Remove consecutive duplicates in range
24. unique_copy : Copy range removing duplicates
25. reverse : Reverse range
26. reverse_copy : Copy range reversed
27. rotate : Rotate left the elements in range
28. rotate_copy : Copy range rotated left
29. random_shuffle : Randomly rearrange elements in range
30. shuffle : Randomly rearrange elements in range using generator

<u>**Partition Operations**</u>

1. is_partitioned : Test whether range is partitioned
2. partition : Partition range in two
3. stable_partition : Partition range in two – stable ordering
4. partition_copy : Partition range into two
5. partition_point : Get partition point

<u>**Sorting**</u>

1. sort : Sort elements in range
2. stable_sort : Sort elements preserving order of equivalents
3. partial_sort : Partially sort elements in range
4. partial_sort_copy : Copy and partially sort range
5. is_sorted : Check whether range is sorted
6. is_sorted_until : Find first unsorted element in range
7. nth_element : Sort element in range

<u>**Binary search (operating on partitioned/sorted ranges)**</u>

1. lower_bound : Return iterator to lower bound
2. upper_bound : Return iterator to upper bound
3. equal_range : Get subrange of equal elements
4. binary_search : Test if value exists in sorted sequence

<u>**Merge (operating on sorted ranges)**</u>

1. merge : Merge sorted ranges
2. inplace_merge : Merge consecutive sorted ranges
3. includes : Test whether the sorted range includes another sorted range
4. set_union : Union of two sorted ranges
5. set_intersection : Intersection of two sorted ranges
6. set_difference : Difference of two sorted ranges
7. set_symmetric_difference : Symmetric difference of two sorted ranges

Heap Operations

1. push_heap : Push element into heap range
2. pop_heap : Pop element from heap range
3. make_heap : Make heap from range
4. sort_heap : Sort elements of heap
5. is_heap : Test if range is heap
6. is_heap_until : Find first element not in heap order
7. max : Return the largest
8. minmax : Return smallest and largest elements
9. min_element : Return smallest element in range
10. max_element : Return largest element in range
11. minmax_element : Return smallest and largest elements in range

Other Operations

1. lexicographical_compare : Lexicographical less-than comparison
2. next_permutation : Transform range to next permutation
3. prev_permutation : Transform range to previous permutation

Functors / function object

Please note that the title is Functors (Not Functions)!!
Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function
Functors are objects that can be treated as though they are a function or function pointer.
Functors are most commonly used along with STLs in a scenario like following:

```
// A C++ program uses transform()
in STL to add
// 1 to all elements of arr[]
#include <bits/stdc++.h>
using namespace std;

int increment(int x) { return (x+1);
}

int main()
{
        int arr[] = {1, 2, 3, 4, 5};
        int n =
sizeof(arr)/sizeof(arr[0]);

        // Apply increment to all
elements of
        // arr[] and store the
modified elements
        // back in arr[]
        transform(arr, arr+n, arr,
increment);

        for (int i=0; i<n; i++)
                cout << arr[i] <<"
";

        return 0;
}

//2 3 4 5 6
```

Suppose 'd' is an object of a class, operator() function can be called as:

```
d();
which is same as:

d.operator() ( );
```

Let's see a simple example:

```
  #include <iostream>
    using namespace std;
   class function_object
   {
      public:
     int operator()(int a, int b)
    {
       return a+b;
    }
  };

   int main()
  {
    function_object f;
    int result = f(5,5);
   cout<<"Addition of a and b is : "<<result;

  return 0;
}
```

Output:

```
Addition of a and b is : 10
```

In the above example, 'f' is an object of a function_object class which contains the definition of operator() function. Therefore, 'f' can be used as an ordinary function to call the operator() function.

Array

The array is a collection of homogeneous objects
This array container is defined for constant size arrays or (static size).
This container wraps around fixed-size arrays and the information of its size are not lost when declared to a pointer.
In order to utilize arrays, we need to include the array header: #include <array>

homogeneous :The data structures which will accept same type of data type is called a homogeneous datastructure. Examples are arrays,strings etc.
heterogenous :The data structure which will accept different type of data types is called a heterogenous data structures.

Syntax:	array<object_type, arr_size> arr_name;

1. **[ ] Operator :** This is similar to the normal array, we use it to access the element store at index 'i' .
2. **front( ) and back( ) function:** These methods are used to access the first and the last element of the array directly.
3. **swap( ) function:** This swap function is used to swap the content of the two arrays.
4. **empty( ) function:** This function is used to check whether the declared STL array is empty or not, if it is empty then it returns true else false.
5. **at( ) function:** This function is used to access the element stored at a specific location, if we try to access the element which is out of bounds of the array size then it throws an exception.
6. **fill( ) function:** This is specially used to initialize or fill all the indexes of the array with a similar value.
7. **size( ) or max_size( ) and sizeof( ) function:** Both size( ) or max_size( ) are used to get the maximum number of indexes in the array while sizeof( ) is used to get the total size of array in bytes.
8. **data( ):** This function returns the pointer to the first element of the array object. Because elements in the array are stored in contiguous memory locations. This data( ) function return us the base address of the string/char type object.

```
#include<iostream>
#include<array>
using namespace std;
int main()
{array<int,4> arr;
 cout<<arr.size()
<<endl;
arr={1,2,3,4};

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

cout<<arr.at(i)<<endl;
} }
output:
1
2
3
4
```

```
at
int main()
{array<int,4> arr;
arr={1,2,3,4};

for (int i = 0; i < 6; i++)
{
   cout<<arr.at(i)<<endl;
}
 return 0;
}
output:
1
2
3
4
terminate called after
throwing an instance of
'std::out_of_range'
  what():  array::at: __n
(which is 4) >= _Nm (which
is 4)
```

```
font() and back()

#include<iostream>
#include<array>
using namespace std;
int main()
{array<int,4> arr;
arr={1,2,3,4};

cout<<arr.front() <<endl;  //1
cout<<arr.back()<<endl;   //4
 return 0;
}
```

```

#include<iostream>
#include<array>
using namespace std;
int main()
{array<int,4> arr;
arr={1,2,3,4};

for (int i = 0; i < 4; i++)
{
   cout<<arr.at(i)<<endl;
}


for (int i = 0; i < 4; i++)
{   arr.fill(20);
   cout<<arr.at(i)<<endl
<<"\t";
}
 return 0;
}
operator delete[]
1 2 3 4
20 20 20 20
```

// swap()

```
#include<iostream>
#include<array>
using namespace std;
int main()
{array<int,4> a{10,20,30,40};
array<int,4> b{100,200,300,400};
a.swap(b);
for (int i = 0; i < a.size(); i++)
{
   cout <<"value of a :" <<a.at(i)<<endl; }
for (int i = 0; i < b.size(); i++)
{
   cout <<"value of b :" <<b.at(i)<<endl;
} }
```

```
output:
value of a :100
value of a :200
value of a :300
value of a :400
value of b :10
value of b :20
value of b :30
value of b :40
```

Data()

```
#include <iostream>
#include <cstring>
#include <array>
using namespace std;
int main ()
{
const char* str = "GeeksforGeeks";
array<char,13> arr;
memcpy (arr.data(),str,13);
cout << arr.data() << '\n';
return 0;
} //GeeksforGeeks
```

```
#include <iostream>
#include <array>
using namespace std;

int main() {
array <int , 10> arr;
cout<<arr.size()<<'\n'; // total num of indexes
cout<<arr.max_size()<<'\n'; // total num of indexes
cout<<sizeof(arr); // total size of array
return 0;
}
10
10
40
```

```
#include <iostream>
#include <array>
using namespace std;

int main()
 {
array <int , 5> arr;
arr.fill(1);
for(int i: arr)
cout<<arr[i]<<" ";
return 0;
}
// 1 1 1 1 1
```

at

```
#include <iostream>
#include <array>
using namespace std;

int main() {
        array <int , 3> arr={'G','f','G'}; // ASCII val of 'G' =71
        array <int , 3> arr1={'M','M','P'}; // ASCII val of 'M' = 77 and 'P' = 80
        cout<< arr.at(2) <<" " << arr1.at(2);
        //cout<< arr.at(3); // exception{Abort signal from abort(3) (SIGABRT)}
        return 0;
}
```

```
                   iterator
#include<iostream>
#include<array>
using namespace std;
int main()
{array<int,4> a{10,20,30,40};
//iterat //iterator
array<int,4>::iterator b=a.begin() ;
array<int,4>::iterator c=a.end()-1 ;
// cout<<b;   //0x61fefc
cout<<*b;    //0x61fefc
cout<<*c;    //40
 return 0;}
```

```
#include<iostream>
#include<array>
using namespace std;
int main()
{array<int,4> a{10,20,30,40};
//iterat //iterator
array<int,4>::iterator b=a.begin() ;

cout<<*(b);  //20
cout<<*(b+1); //20
 return 0;
}
```

Vectors

- Vectors are the 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.

std::vector in C++ is the class template that contains the vector container and its member functions.
It is defined inside the <vector> header file.
The member functions of std::vector class provide various functionalities to vector containers.
<vector>

Iterators

1. **begin() –** Returns an iterator pointing to the first element in the vector
2. **end() –** Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. **rbegin() –** Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. **rend() –** Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
5. **cbegin() –** Returns a constant iterator pointing to the first element in the vector.
6. **cend() –** Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
7. **crbegin() –** Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
8. **crend() –** Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

Capacity

1. **size() –** Returns the number of elements in the vector.
2. **max_size() –** Returns the maximum number of elements that the vector can hold.
3. **capacity() –** Returns the size of the storage space currently allocated to the vector expressed as number of elements.
4. **resize(n) –** Resizes the container so that it contains 'n' elements.
5. **empty() –** Returns whether the container is empty.
6. **shrink_to_fit() –** Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
7. **reserve() –** Requests that the vector capacity be at least enough to contain n elements.

Element access

1. **reference operator [g] –** Returns a reference to the element at position 'g' in the vector
2. **at(g) –** Returns a reference to the element at position 'g' in the vector
3. **front() –** Returns a reference to the first element in the vector
4. **back() –** Returns a reference to the last element in the vector
5. **data() –** Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

Modifiers

1. **assign() –** It assigns new value to the vector elements by replacing old ones
2. **push_back() –** It push the elements into a vector from the back
3. **pop_back() –** It is used to pop or remove elements from a vector from the back.
4. **insert() –** It inserts new elements before the element at the specified position
5. **erase() –** It is used to remove elements from a container from the specified position or range.
6. **swap() –** It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
7. **clear() –** It is used to remove all the elements of the vector container
8. **emplace() –** It extends the container by inserting new element at position
9. **emplace_back() –** It is used to insert a new element into the vector container, the new element is added to the end of the vector

```
        #include<iostream>
          #include<vector>
          using namespace std;
          int main()
          {vector<int> b;
           cout<<b.size() <<endl; //0
           return 0;
}
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{vector<int> b;
 cout<<b.capacity() <<endl; //0  capacity() - >element present in array
 return 0;
}
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{vector<int> b;
b.push_back(100);
cout<<b.size(); //1
 cout<<b.capacity() <<endl;
//1
 return 0;
}
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{vector<int> b;
b.push_back(100);
b.push_back(200);
cout<<b.size(); //2
 cout<<b.capacity() <<endl;
//2
 return 0;
}
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{vector<int> b;
b.push_back(100);
b.push_back(200);
b.push_back(300);
cout<<b.size(); //3
 cout<<b.capacity() <<endl;
//4
 return 0;
}
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{vector<int> b;
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
b.push_back(100);
cout<<b.size(); //5
 cout<<b.capacity() <<endl; //8
 return 0;
}


//when you use popback element it del the element not memory
```

```
#include<iostream>
#include<vector>  //fifo
using namespace std;
int main()
{vector<int> a;
a.push_back(1);
a.push_back(2);
a.push_back(3);
//this is range function
//this for copy array value   ,call by value
for(int &b:a){
   cout<<b++ <<"\t";    //123
}

//this for copy array value   ,call by reference
for(int b:a){
   cout<<b <<"\t";   // 2 3 4
}
 return 0;
}
```

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{vector <string> a;
a.push_back("mon");
a.push_back("tue");
a.push_back("web");

for(auto b:a){
   cout<<b <<endl;

}
cout<<a.size();
cout<<a.capacity();
a.pop_back();
cout<<a.size();
cout<<a.capacity();
return 0;}
op:
mon
tue
web
3424
```

```
#include<iostream>
#include<vector>  //fifo
using namespace std;
int main()
{vector<int> a;
a.push_back(1);
a.push_back(2);
a.push_back(3);
for (auto b:a){
   cout<<b <<endl;
}
vector<int>::iterator r=a.begin();
cout<<*++r;

 return 0;
}

output:
1
2
3
2
```

```
#include<iostream>
#include<vector>  //fifo
using namespace std;
int main()
{vector<int> a{1,2,3,4,5};              //static array
for (auto b:a){
   cout<<b <<endl;
}
cout<<endl;
vector<int>::iterator r=a.begin();
cout<<*++r;

 cout<<endl;
 cout<<a.size() <<endl; //5
 cout<<a.capacity();    //5


 return 0;
}
```

```

//when we use  a.push_back(10); it behave like dynamic array but
it is static array (coz we fix the size of array)
#include<iostream>
#include<vector>  //fifo
using namespace std;
int main()
{vector<int> a{1,2,3,4,5};              //static array
a.push_back(10);                        //dynamic
for (auto b:a){
   cout<<b <<endl;
}
cout<<endl;
 cout<<endl;
 cout<<a.size() <<endl; //6
 cout<<a.capacity();    //10
 return 0;
}
```

insert and erase	#include<iostream>	#include<iostream>	#include<iostream>
	#include<vector>	#include<vector>	#include<vector>
#include<iostream>	#include<algorithm>	#include<algorithm>	#include<algorithm>
#include<vector>	using namespace std;	using namespace std;	using namespace std;
#include<algorithm>	int main()	int main()	int main()
using namespace std;	{vector<int> a{1,2,3,4,5};	{vector<int> a{1,2,3,4,5};	{vector<int> a{1,2,3,4,5};
int main()	a.insert(a.begin()+5,3,300)	a.erase(a.begin(),a.begin()+2)	//static array
{vector<int> a{1,2,3,4,5};	;	;	a.assign({100});
a.insert(a.begin()+5,300)	for (auto b:a){	for (auto b:a){	for (auto b:a){
;	cout<<b <<"\t";	cout<<b <<"\t"; //3 4	cout<<b <<"\t";
for (auto b:a){	}	5	//100
cout<<b <<"\t";	return 0;	}	}
//1 2 3 4 5 300	}	return 0;	cout<<a.size() <<endl;
}	//1 2 3 4 5 300 300 300	}	//1
return 0;}			cout<<a.capacity();
			//5
			return 0;
			}

Stack

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 (top) and an element is removed from that end only.
For creating a stack, we must include the <stack> header file in our code.

The functions associated with stack are:

1. **empty() –** Returns whether the stack is empty – Time Complexity : O(1)
2. **size() –** Returns the size of the stack – Time Complexity : O(1)
3. **top() –** Returns a reference to the top most element of the stack – Time Complexity : O(1)
4. **push(g) –** Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)
5. **pop() –** Deletes the most recent entered element of the stack – Time Complexity : O(1)

```
#include<iostream>
#include<stack>                   //it is a template
using namespace std;
int main()
{
  stack<int> s;
  for (int i = 1; i <=10; i++)
    {s.push(i);}
    while(!s.empty())                         //this loop untill the s in empty
 {
  cout<<s.top() <<"\t";
//10      9       8       7       6       5       4       3       2       1
  s.pop();
 }
 return 0;
}
```

Queues

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

1. **empty()** Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.
2. **size()** Returns the size of the queue.
3. **swap()** Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ.
4. **emplace()** Insert a new element into the queue container, the new element is added to the end of the queue.
5. **front()** Returns a reference to the first element of the queue.
6. **back()** Returns a reference to the last element of the queue.
7. **push(g)** Adds the element 'g' at the end of the queue.
8. **pop()** Deletes the first element of the queue.

Time complexity for all function in O(1)

```
#include<iostream>
#include<queue>             //it is a template
using namespace std;
int main()
{
 queue<int> s;
 for (int i = 1; i <=10; i++)
  {s.push(i);}
  while(!s.empty())                    //this loop untill the s in empty
 {
 cout<<s.front() <<"\t"; //1    2    3    4    5    6    7    8    9    10
 s.pop();
 }
 return 0; }
```

Pair

1. Pair is used to combine together two values that may be of different data types.
2. Pair provides a way to store two heterogeneous objects as a single unit.
3. It is basically used if we want to store tuples.
4. The pair container is a simple container defined in <utility> header consisting of two data elements or objects.
5. The first element is referenced as 'first' and the second element as 'second' and the order is fixed (first, second).
6. python dict is made of pair
7. pair is a class template

```
#include<iostream>
using namespace std;
int main()
{pair<int,string>p;
p.first=10;
p.second="amarth";
cout<<p.first<<" "<<p.second<<endl;   //10 amarth

// first ,second in keyword
 return 0;
}
```

how to use pair with vector

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<pair<int,int>> p;
int n ,roll,age;
cout<<"how many record u want ";
cin>>n;
for(int i=0;i<n;i++){
    cin>> roll >> age ;
    p.push_back(make_pair(roll,age));
    p.push_back({roll,age});  //you can do this to
}
for(int i=0;i<n;i++){
    cout<< "roll number :"<<p[i].first <<", age :" <<p[i].second ;
    cout<<endl;
}  return 0;}
```

```
output:
how many record u want 2
101
1
102
2
roll number :101, age :1
roll number :102, age :2
```

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<pair<int, int>> p1{{101, 20}, {102, 25}, {103, 19}};
    vector<pair<int, int>> p2{{101, 21}, {102, 25}, {103, 11}};
    int c = 0;
    for (int i = 0; i < 3; i++)
    {

        if (p1[i].first == p2[i].first && p1[i].second == p2[i].second)
        {
            cout<<p1[i].first <<" " <<p1[i].second <<endl;
            c++;
        }
    };
    cout << c;

    return 0;
}
```

```
output:
102 25
1
```

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<pair<int, int>> p1{{101, 20}, {102, 25}, {103, 19}};
    vector<pair<int, int>> p2{{201, 21}, {202, 25}, {203, 11}};

   p1.swap(p2);
for (int i = 0; i < 3; i++)
{
    cout<<p1[i].first <<" " <<p1[i].second <<endl;
}
    return 0;
}
```

```
output:
201 21
202 25
203 11
```

make_pair(): This template function allows to create a value pair without writing the types explicitly.
Syntax: Pair_name = make_pair (value1,value2);

```
#include <iostream>
#include <utility>
using namespace std;
int main()
{         PAIR3 = make_pair("GeeksForGeeks is Best", 4.56);
          cout << PAIR3.first << " ";
          cout << PAIR3.second << endl;}
```

output: GeeksForGeeks is Best 4.56

swap: This function swaps the contents of one pair object with the contents of another pair object. The pairs must be of the same type.
Syntax: pair1.swap(pair2) ;

tuple

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.

Operations on tuple :-
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.

1. **make_tuple()** is used to assign tuple with values. The values passed should be in order with the values declared in tuple.
2. **tuple_size :-** It returns the number of elements present in the tuple.
3. **swap() :-** The swap(), swaps the elements of the two different tuples.
4. **tuple_cat() :-** This function concatenates two tuples and returns a new tuple.

```
#include <iostream>
#include <tuple>
using namespace std;
int main()
{  tuple<int,int,int> t;
    t=make_tuple(10,20,30);
    cout<<get<0>(t)  <<" " <<get<1>(t)  <<" " <<get<2>(t);   //  10 20 30
}
```

```
#include<iostream>
#include<tuple> // for tuple
using namespace std;
int main()
{          tuple <char, int, float> geek;

           geek = make_tuple('a', 10, 15.5);

           cout << "The initial values of tuple are : ";

           cout << get<0>(geek) << " " << get<1>(geek);

           cout << " " << get<2>(geek) << endl;

           // Use of get() to change values of tuple
           get<0>(geek) = 'b';
           get<2>(geek) = 20.5;

           // Printing modified tuple values
           cout << "The modified values of tuple are : ";
           cout << get<0>(geek) << " " << get<1>(geek);
           cout << " " << get<2>(geek) << endl;
}
```

```
The initial values of tuple are : a 10
15.5
The modified values of tuple are : b 10
20.5
```

tuple store more than two data type is like advance structer

```
#include <iostream>
#include <tuple>
using namespace std;
int main()
{
   pair<int, int> p;
   p.first = 10;
   p.second = 220;
   cout << p.first << " " << p.second << endl;
   tuple<int, int, int> t;
   // t=make_tuple(10,20,30);
   t = {10, 20, 30};
   cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t);
}
```

output:

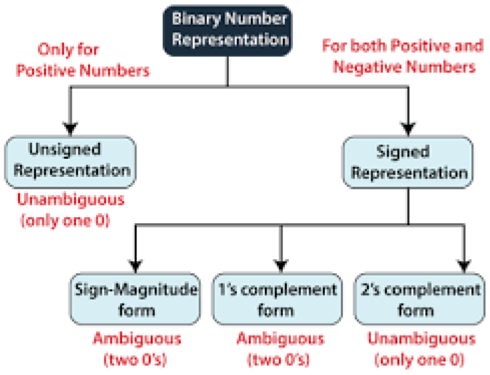
```
10 220
10 20 30
```

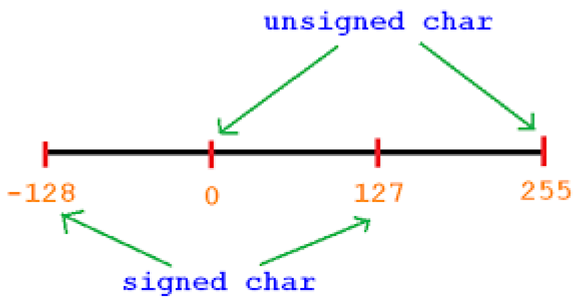
```
// how to use pair with vector
#include <iostream>
#include <vector>
using namespace std;
int main()
{
vector<pair<int,pair<int,int>>> vt;
vt.push_back({1,{101,102}});
cout<<vt[0].first<<endl;   //1
cout<<vt[0].second.first<<endl; //101
cout<<vt[0].second.second<<endl; //102

    return 0;
}
```

signed and unsigned

two data type
signed and unsigned

signed - by defalut





```
signed
#include<iostream>
using namespace std;
int main()
{ int a=10;
cout<<a;      //10
 return 0;
}

#include<iostream>
using namespace std;
int main()
{
signed int a=10;
cout<<a;      //10
 return 0;
}
```

```
unsigned
#include<iostream>
using namespace std;
int main()
{unsigned int a=-2;
cout<<a;      //65534
 return 0;
}

a= -2
 in turbo
 1 byte=8 bit
 so
 (in turbo it takes 2 )*8=16
 2^16   = 65536
 than
 65536- 2   = 65534
```

```
#include<iostream>
using namespace std;
int main()
{unsigned int a=-8;

 int b;
 b=a;

cout<<a;     //4294967288

cout<<b;  //-8
 return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{unsigned int a=-8;

 unsigned int b;
 b=a;
cout<<a
<<endl;     //4294967288

cout<<b;  //-4294967288
 return 0;}
```

```
#include<iostream>
using namespace std;
int main()
{unsigned int a;

  int b=-8;
 a=b;

cout<<b <<endl; //-8

cout<<a;  //-4294967288
 return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{unsigned int a = -65537;

int b;
b=a;

cout<<b <<endl; //
cout<<a;  //
return 0;
}
```

bitset

1. A limitation of the bitset is that size must be known at compile time i.e. size of the bitset is fixed.
2. bitset is the class template for bitset that is defined inside <bitset> header file so we need to include the header file before using bitset in our program.
3. only work on static memory
4. rat maze problem

```
bitset<size> variable_name(initialization);
```

```
#include<iostream>
#include<bitset>
using namespace std;
int main()
{
    bitset<8>  b(35);               //bitset<8> = 8 is 8 bit , it take 8 bit
    bitset<6>  c(35);
    cout<<b <<endl; //00100011 output given in binary
    cout<<c;    //100011
 return 0;
}
```

```
#include<iostream>
#include<bitset>
using namespace std;
int main()
{
    bitset<8>  b(35);                              //bitset<8> = 8 is 8 bit , it take 8 bit
   cout<<b <<endl;          //00100011
    cout<<b.count() <<endl; //3 (count only 1 in 35 binary)
    cout<<b.set() <<endl; //11111111 (set convet all zero in 1 , replace 0 --> 1)

    cout<<b.any() <<endl; //1 (true)
(if 1 is  present in 35 (00100011) binary number it return true , if not presnet return
false )

 return 0;
}
```

```
#include <iostream>
#include <bitset>
using namespace std;
int main()
{    bitset<8> b(35);   // bitset<8> = 8 is 8 bit , it take 8 bit
    cout << b << endl; // 00100011
    cout << b.test(0) << endl; // 1
    cout << b.test(1) << endl; // 1
    cout << b.test(2) << endl; // 0
    cout << b.test(3) << endl; // 0
    cout << b.test(4) << endl; // 0
    cout << b.test(5) << endl; // 1
    cout<<b.reset() <<endl; //00000000 (reset  change all value in 0)
}

#include <iostream>
#include <bitset>
using namespace std;
int main()
{   bitset<8> b(35);   // bitset<8> = 8 is 8 bit , it take 8 bit
    cout << b << endl; // 00100011
    cout<<b.set(3) <<endl; //00101011 (to set particular value change into 1 use set(3))
}



#include <iostream>
#include <bitset>
using namespace std;
int main()
{   bitset<8> b(257);   // 2^8=256 (range crossed , than return )
    cout << b << endl; // 00000001
}


#include <iostream>
#include <bitset>
using namespace std;
int main()
{
    bitset<16> b(257);   // 2^8=256 (range crossed , than return )
    cout << b << endl; //0000000100000001
}
```

```
all()  if all number in binary in 1 , than all return true(1)
#include <iostream>
#include <bitset>
using namespace std;
int main()
{
    bitset<6> b(35);
    cout << b << endl;   //100011
    cout<<b.set() <<endl; //111111
    cout<<b.all() ;// 1
    return 0;
}
```

```
fill()  chnage 1 into 0 and 0 into 1
#include <iostream>
#include <bitset>
using namespace std;
int main()
{
    bitset<6> b(35);
    cout << b << endl;   //100011
    cout<<b.set() <<endl; //111111
    cout<<b.flip() ;//  000000
    return 0;
}
```

<u>type casting</u>

4 types of type casting

1.static_cast

2.reinterpret_cast

3.constant_cast

4.dynamic_cast

```
#include<iostream>
using namespace std;
int main()
{int a=98.4;
cout<<a;  //98
 return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{int a=98.4;
cout<<float(a);  // 98
 return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{int a=98.4;
float f;
f=a;
cout<<f;  // 98
 return 0;
}
```

static_cast

```
#include<iostream>
using namespace std;
int main()
{
unsigned int a=-2;
cout<<static_cast<int>(a); //  static_cast<int> static is a tempalte , is assigning int
data type in runtime
 return 0;
}
```

without variable display value using pointer

```
#include<iostream>
using namespace std;
int main()
{int *p=new int(56);
cout<<*p <<endl;//56
cout<<p <<endl; //0x8c7ec8
 return 0;
}
```

reinterpret_cast

when doing pointer type casting we using reinterpret_cast
pointer conversion

```
#include<iostream>
using namespace std;
int main()
{int *p=new int(65);
//chnage into character
char *c= reinterpret_cast<char *>(p);
/// char is the data type of c so it will be converted to p and then it will convert that
into char
cout<<*c ; //A
 return 0;
}
```

constant_cast

to change the value of const type of data we use constant_cast only work on pointer .

```
#include<iostream>
using namespace std;
int main()
{const  int a =9;
cout<<a; //9

 return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{const  int a =9;
 const int *p=&a;
 cout<<"value "<<*p <<endl; //value 9
 cout<<"address " <<&(*p) << endl;
//address 0x61ff08
 cout<<*p+10; //19
 cout<<a ; //9
 return 0;
}
```

lambda

C++ 11 introduced lambda expressions to allow inline functions which can be used for short snippets of code that are not going to be reused and therefore do not require a name.
In their simplest form a lambda expression can be defined as follows:

```
[ capture clause ] (parameters) -> return-type
{
  definition of method
}
```

`#include <iostream> using namespace std; int main() { cout<<  [](int a) { return a + 2; }(8); //10 return 0; }`	`#include <iostream> using namespace std; int main() { auto p =[](int a) { return a + 2; }(8); cout<<p; //10 return 0; }`	`#include <iostream> using namespace std; int main() { auto p =[](int a) { return a + 2; }; cout<<p(20); //22 return 0; }`	`#include <iostream> using namespace std; int main() { auto p =[](int a , int b) // (int a , int b) this is called captcha { return a + b + 2; }; cout<<p(20,20);     //44 return 0; }`

generic lambda function

address use in this

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{vector<int>v{1,2,3,4,5};
auto k=[&v](int t)
{
     for (int i = 0; i < v.size(); i++)
     {
        v[i] +=t;
     }
};
k(10);
for(auto k:v){
    cout <<k<<" ";
}                                   //11 12 13 14 15

}
```

List

1. List is a contiguous container while vector is a non-contiguous container i.e list stores the elements on a contiguous memory and vector stores on a non-contiguous memory.
2. Insertion and deletion in the middle of the vector is very costly as it takes lot of time in shifting all the elements. Linklist overcome this problem and it is implemented using list container.
3. List supports a bidirectional and provides an efficient way for insertion and deletion operations.
4. Traversal is slow in list as list elements are accessed sequentially while vector supports a random access.

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
  list<int> l;
}
```

It creates an empty list of integer type values.

List can also be initalised with the parameters.

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
  list<int> l{1,2,3,4};
}
```

List can be initialised in two ways.

```
list<int>  new_list{1,2,3,4};
or
list<int> new_list = {1,2,3,4};
```

C++ List Functions

Method	Description
insert()	It inserts the new element before the position pointed by the iterator.
push_back()	It adds a new element at the end of the vector.
push_front()	It adds a new element to the front.
pop_back()	It deletes the last element.
pop_front()	It deletes the first element.
empty()	It checks whether the list is empty or not.
size()	It finds the number of elements present in the list.
max_size()	It finds the maximum size of the list.
front()	It returns the first element of the list.
back()	It returns the last element of the list.
swap()	It swaps two list when the type of both the list are same.
reverse()	It reverses the elements of the list.
sort()	It sorts the elements of the list in an increasing order.
merge()	It merges the two sorted list.
splice()	It inserts a new list into the invoking list.
unique()	It removes all the duplicate elements from the list.
resize()	It changes the size of the list container.
assign()	It assigns a new element to the list container.
emplace()	It inserts a new element at a specified position.
emplace_back()	It inserts a new element at the end of the vector.
emplace_front()	It inserts a new element at the beginning of the list.

Forward List

1. Forward list in STL implements singly linked list. Introduced from C++11,
2. Forward list are more useful than other containers in insertion, removal, and moving operations (like sort) and allow time constant insertion and removal of elements.
3. It differs from the list by the fact that the forward list keeps track of the location of only the next element while the list keeps track of both the next and previous elements.
4. The drawback of a forward list is that it cannot be iterated backward and its individual elements cannot be accessed directly.
5. Forward List is preferred over the list when only forward traversal is
6. Some example cases are, chaining in hashing, adjacency list representation of the graph, etc.

Method	Definition
assign():	This function is used to assign values to the forward list, its other variant is used to assign repeated elements and using the values of another list.
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 the 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 the 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. There are two variants of 'erase after' function.
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.
clear():	. This function deletes all the elements from the list.
splice_after():	This function transfers elements from one forward list to other.
front()	This function is used to reference the first element of the forward list container.
begin()	This function is used to return an iterator pointing to the first element of the forward list container.
end()	This function is used to return an iterator pointing to the last element of the list container.
cbegin()	Returns a constant iterator pointing to the first element of the forward_list.
cend()	Returns a constant iterator pointing to the past-the-last element of the forward_list.
before_begin()	Returns an iterator that points to the position before the first element of the forward_list.
cbefore_begin()	Returns a constant random access iterator which points to the position before the first element of the forward_list.
max_size()	Returns the maximum number of elements that can be held by forward_list.
resize()	Changes the size of forward_list.
unique()	Removes all consecutive duplicate elements from the forward_list. It uses a binary predicate for comparison.
reverse()	Reverses the order of the elements present in the forward_list.

```
#include <forward_list>
#include <iostream>
using namespace std;
int main()
{           forward_list<int> flist1;
            forward_list<int> flist2;
            forward_list<int> flist3;
            flist1.assign({ 1, 2, 3 });                 // Assigning values using assign()

            //// Assigning repeating values using assign() // 5 elements with value 10
            flist2.assign(5, 10);
            flist3.assign(flist1.begin(), flist1.end());  //Assigning values of list 1 to list 3

            cout << "The elements of first forward list are : ";
            for (int& a : flist1)
             {cout << a << " "  << endl;}

            cout << "The elements of second forward list are : ";
            for (int& b : flist2)
                        {cout << b << " " << endl;}
           cout << "The elements of third forward list are : ";
            for (int& c : flist3)
                        {cout << c << " " << endl; }  }
```

```
OUTPUT :
The elements of first forward list are : 1 2 3
The elements of second forward list are : 10 10 10 10 10
The elements of third forward list are : 1 2 3
```

push_front(), emplace_front() and pop_front()

```
#include <forward_list>
#include <iostream>
using namespace std;
int main()
{           forward_list<int> flist = { 10, 20, 30, 40, 50 };
            flist.push_front(60);  // Inserting value using push_front()  // Inserts 60 at front
            cout<< "The forward list after push_front operation : ";
            for (int& c : flist)
                        {cout << c << " " << endl;}

            flist.emplace_front(70);  // Inserting value using emplace_front()  // Inserts 70 at front

            // Displaying the forward list
            cout << "The forward list after emplace_front operation : ";
            for (int& c : flist)
             {cout << c << " "  << endl;}

            flist.pop_front();  // Deleting first value using pop_front() // Pops 70


            cout << "The forward list after pop_front operation : ";
            for (int& c : flist)
           {cout << c << " "  << endl; }
}
```

```
OUTPUT:
The forward list after push_front operation : 60 10 20 30 40 50
The forward list after emplace_front operation : 70 60 10 20 30 40 50
The forward list after pop_front operation : 60 10 20 30 40 50
```

Deque (Double-ended queues)

Double-ended queues are sequence containers with the feature of expansion and contraction on both ends.
Double-ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.
The functions for deque are same as vector, with an addition of push and pop operations for both front and back.

The time complexities for doing various operations on deques are-
Accessing Elements- O(1)
Insertion or removal of elements- O(N)
Insertion or removal of elements at start or end- O(1)

```
#include <deque>
#include <iostream>
 using namespace std;
 void showdq(deque<int> g)
{
   deque<int>::iterator it;
   for (it = g.begin(); it != g.end(); ++it)
      cout << '\t' << *it;
   cout << '\n';
}

int main()
{
   deque<int> gquiz;
   gquiz.push_back(10);
   gquiz.push_front(20);
   gquiz.push_back(30);
   gquiz.push_front(15);
   cout << "The deque gquiz is : ";
   showdq(gquiz);

   cout << "\ngquiz.size() : " << gquiz.size();
   cout << "\ngquiz.max_size() : " << gquiz.max_size();

   cout << "\ngquiz.at(2) : " << gquiz.at(2);
   cout << "\ngquiz.front() : " << gquiz.front();
   cout << "\ngquiz.back() : " << gquiz.back();

   cout << "\ngquiz.pop_front() : ";
   gquiz.pop_front();
   showdq(gquiz);

   cout << "\ngquiz.pop_back() : ";
   gquiz.pop_back();
   showdq(gquiz);

   return 0;
}
```

```
Output
The deque gquiz is :    15    20    10    30

gquiz.size() : 4
gquiz.max_size() : 4611686018427387903
gquiz.at(2) : 10
gquiz.front() : 15
gquiz.back() : 30
gquiz.pop_front() :    20    10    30

gquiz.pop_back() :    20    10
```

map

1. Maps are part of the C++ STL (Standard Template Library).
2. Maps are the associative containers that store sorted key-value pair, in which each key is unique and it can be inserted or deleted but cannot be altered. Values associated with keys can be changed.
3. Data is always sorted
4. The time complexity is O(log2n)
5. Its supports red black tree concepts algo that is also called self-balanced binary search tree
6. We can perform insert , erase , and find operation .
7. It is used in graph
8. pair is used in map and unordermap

1. **begin() –** Returns an iterator to the first element in the map.
2. **end() –** Returns an iterator to the theoretical element that follows the last element in the map.
3. **size() –** Returns the number of elements in the map.
4. **max_size() –** Returns the maximum number of elements that the map can hold.
5. **empty() –** Returns whether the map is empty.
6. **pair insert(keyvalue, mapvalue) –** Adds a new element to the map.
7. **erase(iterator position) –** Removes the element at the position pointed by the iterator.
8. **erase(const g)–** Removes the key-value 'g' from the map.
9. **clear() –** Removes all the elements from the map.

Begin() , end()

```
#include <iostream>
#include <map>
using namespace std;
int main()
{map<string, int> map;
map["one"] = 1;
map["two"] = 2;
map["three"] = 3;
// Get an iterator pointing to the first element in the map
std::map<std::string, int>::iterator it = map.begin();
while (it != map.end())
{cout << "Key: " << it->first << ", Value: " << it->second <<endl;
++it;
} }
```

```
Key: one, Value: 1
Key: three, Value: 3
Key: two, Value: 2
```

Size()

```
#include <iostream>
#include <map>
using namespace std;
int main()
{map<string, int> map;
map["one"] = 1;
map["two"] = 2;
map["three"] = 3;
cout << "Size of map: " << map.size();        //Size of map: 3
}
```

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
map<string, int> map;
map["one"] = 1;
map["two"] = 2;
map["three"] = 3;
// Check if a key is in the map
if (map.count("four") > 0)
{  cout << "Key 'four' is in the map";}
else{cout << "Key 'four' is not in the map";} //Key 'four' is not in the map
}
```

Constructor/Destructor

Functions	Description
constructors	Construct map
destructors	Map destructor
operator=	Copy elements of the map to another map.

Iterators

Functions	Description
begin	Returns an iterator pointing to the first element in the map.
cbegin	Returns a const iterator pointing to the first element in the map.
end	Returns an iterator pointing to the past-end.
cend	Returns a constant iterator pointing to the past-end.
rbegin	Returns a reverse iterator pointing to the end.
rend	Returns a reverse iterator pointing to the beginning.
crbegin	Returns a constant reverse iterator pointing to the end.
crend	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
empty	Returns true if map is empty.
size	Returns the number of elements in the map.
max_size	Returns the maximum size of the map.

Element Access

Functions	Description
operator[]	Retrieve the element with given key.
at	Retrieve the element with given key.

Modifiers

Functions	Description
insert	Insert element in the map.
erase	Erase elements from the map.
swap	Exchange the content of the map.
clear	Delete all the elements of the map.
emplace	Construct and insert the new elements into the map.
emplace_hint	Construct and insert new elements into the map by hint.

Observers

Functions	Description
key_comp	Return a copy of key comparison object.
value_comp	Return a copy of value comparison object.

Operations

Functions	Description
find	Search for an element with given key.
count	Gets the number of elements matching with given key.
lower_bound	Returns an iterator to lower bound.
upper_bound	Returns an iterator to upper bound.
equal_range	Returns the range of elements matches with given key.

Allocator

Functions	Description
get_allocator	Returns an allocator object that is used to construct the map.

Red-Black Tree (self-balancing Binary Search tree)

1. Red-Black tree is a binary search tree in which every node is colored with either red or black.
2. It is a type of self balancing binary search tree.
3. It has a good efficient worst case running time complexity.
4. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.
5. red black tree - need TWO rotation
6. It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
7. This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.

AVL tree and RED BLACK TREE

why do we require a Red-Black tree if AVL is also a height-balanced tree.
The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree.
The main difference between the AVL tree and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced. So, the AVL tree is more balanced than the Red-Black tree, but the Red-Black tree guarantees O(log2n) time for all operations like insertion, deletion, and searching.

Is every AVL tree can be a Red-Black tree?

Yes, every AVL tree can be a Red-Black tree if we color each node either by Red or Black color.
But every Red-Black tree is not an AVL because the AVL tree is strictly height-balanced while the Red-Black tree is not completely height-balanced.

some rules used to create the Red-Black tree:

1. If the tree is empty, then we create a new node as a root node with the color black.
2. If the tree is not empty, then we create a new node as a leaf node with a color red.
3. If the parent of a new node is black, then exit.
4. If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.
   a) If the color is Black, then we perform rotations and recoloring.
   b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node.

NOTE:
1.The root of the tree is always black.
2.There are no two adjacent red nodes (A red node cannot have a red parent or red child).
3.count number of black node in each path

Sr. No.	**Algorithm**	**Time Complexity**
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

“n” is the total number of elements in the red-black tree.

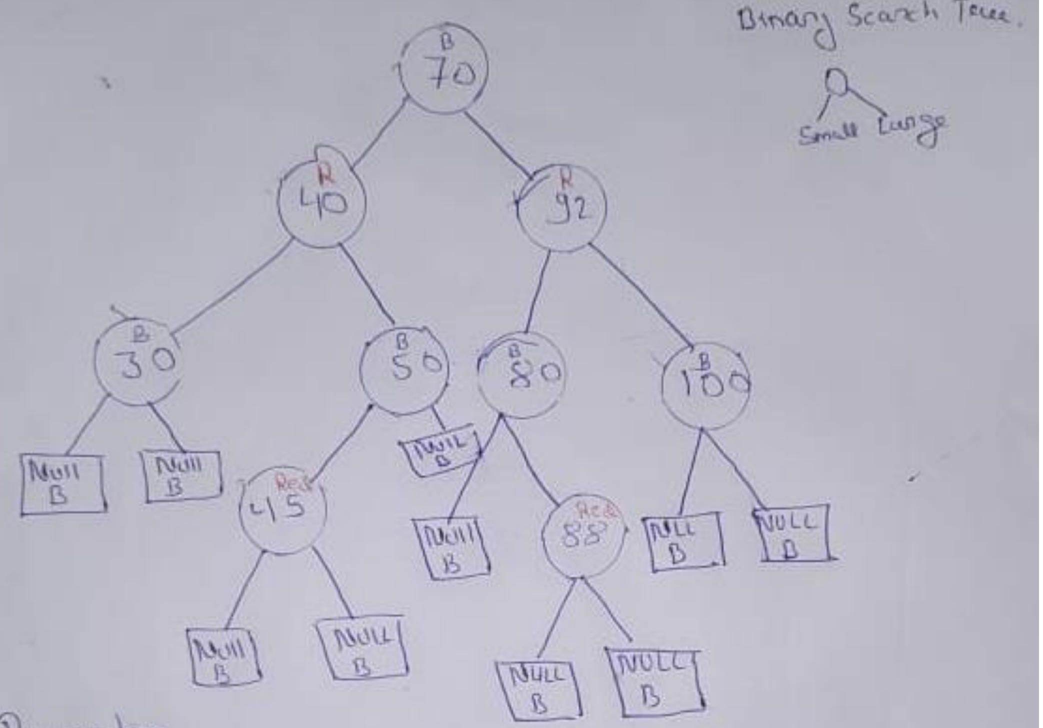
Properties of Red Black Tree:

The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –
1. Root property: The root is black.
2. External property: Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.
3. Internal property: The children of a red node are black. Hence possible parent of red node is a black node.
4. Depth property: All the leaves have the same black depth.
5. Path property: Every simple path from root to descendant leaf node contains same number of black nodes.

Rules That Every Red-Black Tree Follows:

Every node has a color either red or black.
The root of the tree is always black.
There are no two adjacent red nodes (A red node cannot have a red parent or red child).
Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
Every leaf (e.i. NULL node) must be colored BLACK.

Red- Black - Tree.



Properties

① It is a binary Search Tree

2) The root node is black.

3) The child of red node are black.

4) No root to external node path has 2 Consecutive ~~two~~ red node [70 - 40 - 30 - Null]
B R B B

B=3

[~~70 - 40 - 50~~

5) All the root to external node Contain. Same Number of black node

(3 node on each path

70 — 40 - 30 - Null = 3 black.

70 - 40 - 50 - 45 - Null = 3

70 - 40 - 50 - Null = 3.

70 - 92 - 80 - Null = 3

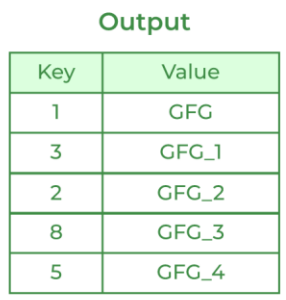
70 - 92 - 100 - Null = 3.

70 - 92 - 80 - 88 - Null = 3.

GO TO JAVA POINT FOR MORE - https://www.javatpoint.com/red-black-tree

unordered_map

Unordered_map	**Map**
The unordered_map key can be stored in any order.	The map is an ordered sequence of unique keys
Unordered_Map implements an unbalanced tree structure due to which it is not possible to maintain order between the elements	Map implements a balanced tree structure which is why it is possible to maintain order between the elements (by specific tree traversal)
The time complexity of unordered_map operations is O(1) on average.	The time complexity of map operations is O(log n)


Output

Key	Value
1	GFG
3	GFG_1
2	GFG_2
8	GFG_3
5	GFG_4

unorder map we use -- hashing technique
// two type of array
// associative (map)
// and numberic

Methods/Functions	Description
at()	This function in C++ unordered_map returns the reference to the value with the element as key k
begin()	Returns an iterator pointing to the first element in the container in the unordered_map container
end()	Returns an iterator pointing to the position past the last element in the container in the unordered_map container
bucket()	Returns the bucket number where the element with the key k is located in the map
bucket_count	Bucket_count is used to count the total no. of buckets in the unordered_map. No parameter is required to pass into this function
bucket_size	Returns the number of elements in each bucket of the unordered_map
count()	Count the number of elements present in an unordered_map with a given key
equal_range	Return the bounds of a range that includes all the elements in the container with a key that compares equal to k
find()	Returns iterator to the element
empty()	Checks whether the container is empty in the unordered_map container
erase()	Erase elements in the container in the unordered_map container

```
#include<iostream>
#include<unordered_map>
using namespace std;
int main()
{  unordered_map <string ,int> up;
     up["sun"]=101;
    up["mon"]=102;
    up["tue"]=103;
    up["wed"]=104;
     for(auto k:up){
    cout<<k.first <<"\t" <<k.second <<endl; }
//   or
for (auto itr=up.begin() ;itr!=up.end(); itr++)
{ cout<<itr->first <<"\t" << itr->second <<endl; }
```

```
OUTPUT:
wed       104
tue       103
sun       101
mon       102
```

```
OUTPUT:
wed       104
tue       103
sun       101
mon       102
```

```
string s="mon";
if (up.find(s) != up.end())
{    cout<<"key found";
    auto t=up.find(s);
    cout<<"key is =" <<t->first <<"\t";
    cout<<"value is =" <<t->second <<"\t"; }
 else{   cout<<"not found";}
```

```
OUTPUT:
key foundkey is =mon
value is =102
```

```
up.insert({"jan",1011});
up.insert({"mar",456});
s="mon",
up.erase(s);
for (auto itr=up.begin() ;itr!=up.end(); itr++)
{
    cout<<itr->first <<"\t" << itr->second <<endl; }
cout<<up.size() <<endl;                                  //5
cout<<up.max_size()<<endl;                              // 59652323
```

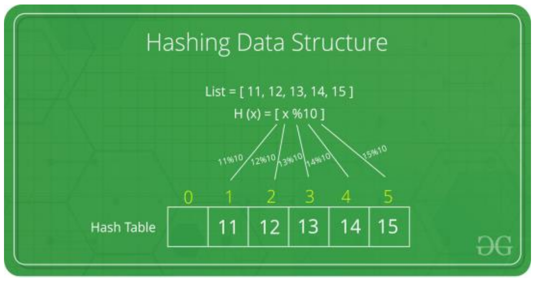
```
OUTPUT:
sun       101
tue       103
wed       104
jan       1011
```

```
// count frequency

int arr[]={2,3,4,7,12,2,2,2,4,5,6,3,3,4};
unordered_map<int,int>upfr;
for (int i = 0; i < 14; i++)
{
    int k=arr[i];
     upfr[k]++;
}
for (auto itr=upfr.begin() ;itr!=upfr.end(); itr++)
{
    cout<<itr->first <<"\t" << itr->second <<endl;
}
}
```

```
Output :

6       1
5       1
3       3
2       4
7       1
12      1
4       3
```

Hashing Method

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function.
Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions.
This technique determines an index or location for the storage of an item in a data structure.



Components of Hashing

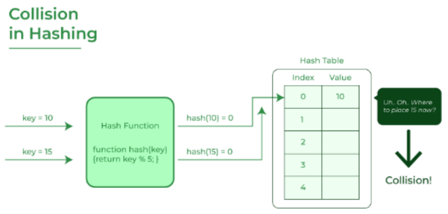
There are majorly three components of hashing:
Key: A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.

Hash Function: The hash function receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index.

Hash Table: Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

What is Collision?

The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



Hashing example key = 22,40,11,9,18,53 | size= n=8 (size should be given) | hashing formula n=n-1 |so n=8-1 = 7
Now we find the mod of all key and jo remainder hoga uski jagh index pr key ko set kr degy
40%8=0
22%8=6
11%8=3
9%8=1
18%8=2
53%8=5

N=7	index	table
1	0	40
2	1	9
3	2	18
4	3	11
5	4	
6	5	53
7	6	22

When the two remainder is same called collision

How to handle Collisions?

<u>There are mainly two methods to handle collision:</u>

1. Separate Chaining
2. Open Addressing or closed hashing.
   Open hashing 3 type
   1.Linear Probing
   2. Quadratic Probing
   3. Double hashing

<u>Load Factor</u>

m = Number of slots in hash table

n = Number of keys to be inserted in hash table

Load factor α = n/m

Load factor = total key / total slot

N= 5

Key =12,16,29,23 ,10 = total 5

Load factor = 5/5 = 1

Separate Chaining:

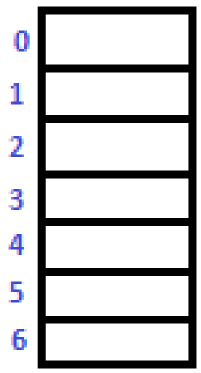
the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

The idea behind separate chaining is to implement the array as a linked list called a chain.

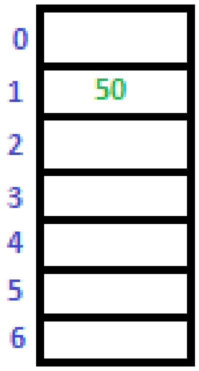
The linked list data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

Here, all those elements that hash into the same slot index are inserted into a linked list.
Now, we can use a key K to search in the linked list by just linearly traversing.
If the intrinsic key for any entry is equal to K then it means that we have found our entry.
If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

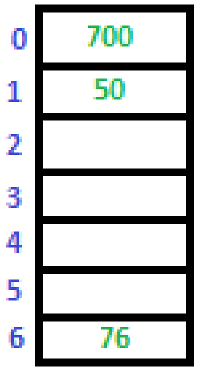
Example: Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101



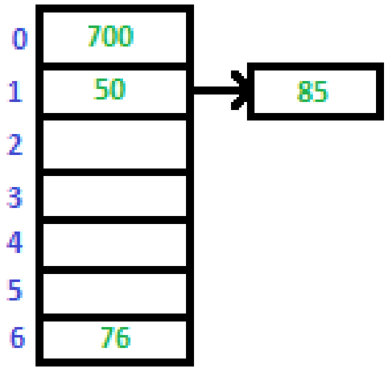
Initial Empty Table



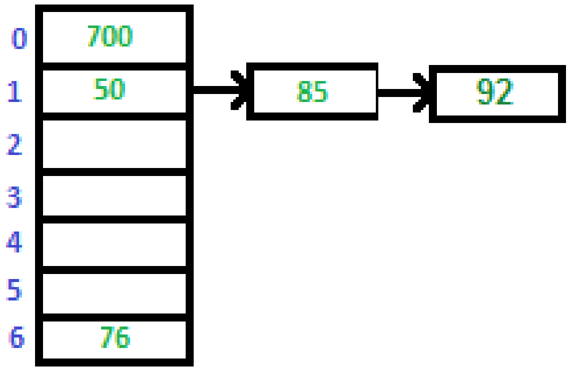
Insert 50



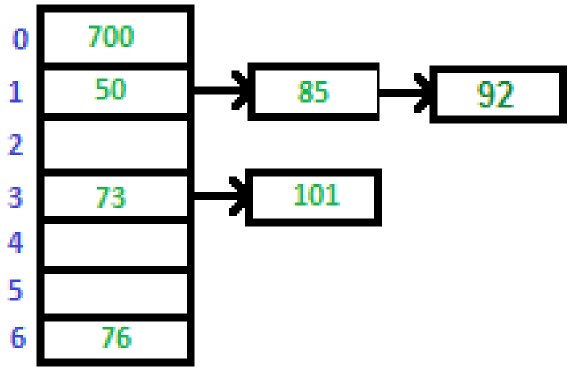
Insert 700 and 76



Insert 85: Collision Occurs, add to chain



Inser 92 Collision Occurs, add to chain



Insert 73 and 101

Open Addressing:

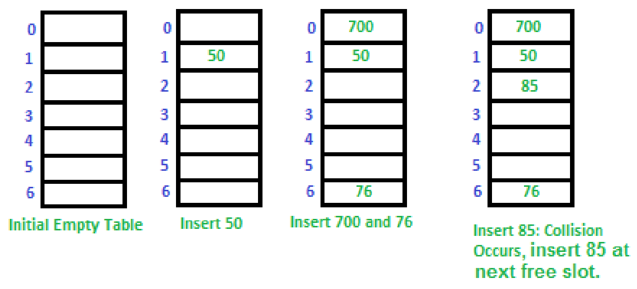
Like separate chaining, open addressing is a method for handling collisions.
In Open Addressing, all elements are stored in the hash table itself.
So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).
This approach is also known as closed hashing.

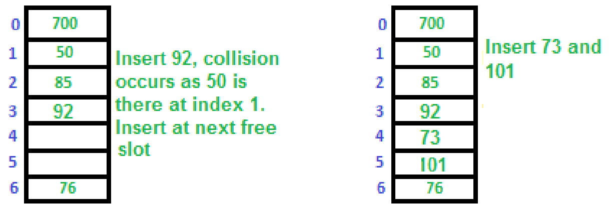
1. Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash.
If in case the location that we get is already occupied, then we check for the next location.

Formula used
H(k) = k mod n
H2(k) = (h(k) + 1) mod n
If not find slot continue
H3(k) = (h(k) + 2) mod n
H4(k) = (h(k) + 3) mod n

Key =50, 700, 76, 85, 92, 73, 101,
Size = 7
Table=7 - 1 = 6
50%7 =1
700%7 =0
76%7=6
85%7 =1
92%7 =1
73%7 =3
101%7 =3




700 50 85 92 76
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
700 50 85 92 73 101 76
Insert 73 and 101

85 collision occurs
H(k) = k mod n
H2(k) = (h(k) + 1) mod n

H(k)= 85 mod 7 = 1
H2(k) = (1 + 1) 7 = 2
Ans=2
2 index is free so we put 85 in second .

Follow this for all collision

Quadratic Probing

Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed above.

This method is also known as the mid-square method.

In this method, we look for the i2'th slot in the ith iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

Formula used H(k) = k mod n H2(k) = (h(k) + 1^2) mod n If not find slot continue H3(k) = (h(k) + 2^2) mod n H4(k) = (h(k) + 3^2) mod n

Double Hashing

let hash(x) be the slot index computed using hash function.

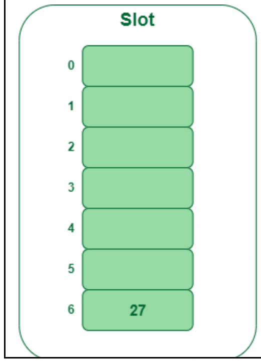
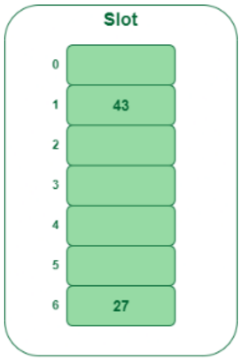
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7.

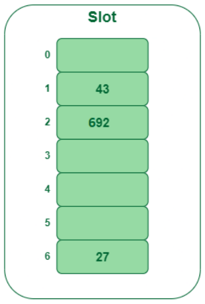
first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Step 1: Insert 27	Step 2: Insert 43
27 % 7 = 6, location 6 is empty so insert 27 into 6 slot. 	43 % 7 = 1, location 1 is empty so insert 43 into 1 slot 

Step 3: Insert 692

692 % 7 = 6, but location 6 is already being occupied and this is a collision

So we need to resolve this collision using double hashing.



*h_{new} = [h1(692) + i * (h2(692)] % 7*

*= [6 + 1 * (1 + 692 % 5)] % 7*

= 9% 7

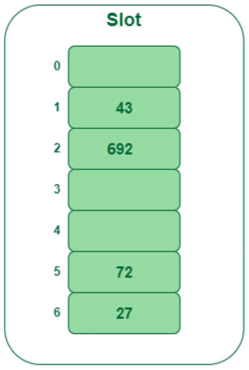
= 2

Now, as 2 is an empty slot,

so we can insert 692 into 2nd slot.

Insert key 692 in the hash table

Step 4: Insert 72
72 % 7 = 2, but location 2 is already being occupied and this is a collision.
So we need to resolve this collision using double hashing.
*h_{new} = [h1(72) + i * (h2(72)] % 7*
*= [2 + 1 * (1 + 72 % 5)] % 7*
= 5 % 7
= 5,
Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.



Insert key 72 in the hash table

Types of Hash functions:

There are many hash functions that use numeric or alphanumeric keys.

1. Division Method.
2. Mid Square Method.
3. Folding Method.
4. Multiplication Method

Sets

Sets are a type of associative container in which each element has to be unique because value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending.
The set class is the part of C++ Standard Template Library (STL) and it is defined inside the <set> header file.
Syntax:
set <data_type> set_name;

set<int> val; // defining an empty set
set<int> val = {6, 10, 5, 1}; // defining a set with values

```
#include <iostream>
#include <set>
using namespace std;
int main()
{   set<int> a={1,2,3,4};
    for (auto str : a) {
        cout << str << ' ';
    }
    cout << '\n';         //1 2 3 4
    return 0;
}
```

iterators

Functions	Description
Begin	Returns an iterator pointing to the first element in the set.
cbegin	Returns a const iterator pointing to the first element in the set.
End	Returns an iterator pointing to the past-end.
Cend	Returns a constant iterator pointing to the past-end.
rbegin	Returns a reverse iterator pointing to the end.
Rend	Returns a reverse iterator pointing to the beginning.
crbegin	Returns a constant reverse iterator pointing to the end.
Crend	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
empty	Returns true if set is empty.
Size	Returns the number of elements in the set.
max_size	Returns the maximum size of the set.

Modifiers

Functions	Description
insert	Insert element in the set.
Erase	Erase elements from the set.
Swap	Exchange the content of the set.
Clear	Delete all the elements of the set.
emplace	Construct and insert the new elements into the set.
emplace_hint	Construct and insert new elements into the set by hint.

Observers

Functions	Description
key_comp	Return a copy of key comparison object.
value_comp	Return a copy of value comparison object.

Operations

Functions	Description
Find	Search for an element with given key.
count	Gets the number of elements matching with given key.
lower_bound	Returns an iterator to lower bound.
upper_bound	Returns an iterator to upper bound.
equal_range	Returns the range of elements matches with given key.

Allocator

Functions	Description
get_allocator	Returns an allocator object that is used to construct the set.

C++ Namespaces

Namespaces in C++ are used to organize too many classes so that it can be easy to handle the application.
For accessing the class of a namespace, we need to use namespacename::classname.
We can use using keyword so that we don't have to use complete name all the time.
In C++, global namespace is the root namespace.
The global::std will always refer to the namespace "std" of C++ Framework.

Let's see the simple example of namespace which include variable and functions.

```
#include <iostream>
using namespace std;
namespace First {
   void sayHello() {
      cout<<"Hello First Namespace"<<endl;
   }
}
namespace Second  {
     void sayHello() {
        cout<<"Hello Second Namespace"<<endl;
     }
}
int main()
{
 First::sayHello();
 Second::sayHello();
return 0;
}
```

```
Output:
Hello First Namespace
Hello Second Namespace
```

C++ namespace example: by using keyword
Let's see another example of namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

```
#include <iostream>
using namespace std;
namespace First{
  void sayHello(){
    cout << "Hello First Namespace" << endl;
  }
}
namespace Second{
  void sayHello(){
    cout << "Hello Second Namespace" << endl;
  }
}
using namespace First;
int main () {
  sayHello();
}
```

Output: Hello First Namespace