Primitive Types:

Array :

One Dimensional: int p[10]; or int *p = new int[10] or std::array<int, 5> p;

Two Dimensional: int two_d[10][20];

Three Dimensional: int three_d[10][20][30];

Size: int len = sizeof(arr)/sizeof(arr[0]);

Accessing: *for(int i = 0; i<n;i++) cout<<arr[i];*

for(auto i : arr) cout << i;

Deleting array if created using new : delete[] p;

Sorting: sort(arr, arr + n);

Strings:

Declaration:	string str;
Line of input:	getline(cin,str);
Adding character:	s.push_back('a')
Removing character:	s.pop_back()
Size:	s.length()
Accessing:	for(int i=0;i<s.length();i++) cout<<s[i];
	for(auto it;=s.begin();it!=s.end();it++) cout<< *it;
	for(autot it : s) cout<<s[i];
Sorting:	sort(str.begin(), str.end());
Functions on string:	
char array from string:	const char* charstr = str6.c_str();
string from char array:	string s(charstr);
Substring:	s.substr(a,b) // substring of length b starting from pos a
	s.substr(a) // substring starting from pos a ending till end.
Reverse:	reverse(str.begin(), str.end());
Erase:	s.erase(a,b) // erase b characters starting from a.
Compare:	str.compare(str1)
Clear:	s.clear():
If empty:	s.empty() // 1 : true and 0 : false

Vectors:

One-d: vector<int> v;

Vector of vectors: vector<vector<int>> v;

Vector array: vector<int> v[n];

Size: v.size();

Add element: v.push_back(i);

Remove element: v.pop_back();

Clear all elements: v.clear();

Update element: v[i] = new_val;

Iterators: begin(),end(),rbegin(),rend()

Accessing: for(int i=0;i<v.size();i++) cout<<v[i];

for(auto it : v) cout<it;

Sorting: sort(v.begin(), v.end()); // non-decreasing

sort(v.begin(), v.end(),greater<int>); // non-increasing

Pairs:

Single: pair<int, char> PAIR1;

Array Pair: pair<int,int> p[n];

Adding value: p = {1,'a'}

Accessing Value: cout<< p.first << p.second;

Vector pair: vector<pair<int,int>> p;

Sort vector pair: sort(v.begin(),v.end()) // using first element

sort(v.begin(),v.end(),myComp); //using second element where myComp is as below:

bool myComp (const pair<int,int> &a, const pair<int,int> &b){

return (a.second < b.second); }

Sort vector pair non-increasing: sort(vect.rbegin(), vect.rend());

Map:

Creation:	unordered_map<int,int> m;
	map<int,int> m; (similar to treemap in java)
Map array:	unordered_map<int,int> m [n];
Size:	m.size();
Add element:	m['key'] = val;
Remove element:	m.erase(val)
Clear all elements:	m.clear();
Update element:	m[i] = new_val;
Find element	auto it = m.find(val);
	if(it!=m.end()){ cout<<it.first<<it.second;}
Iterators:	begin(),end(),rbegin(),rend()
Accessing:	for(auto it=m.begin();it!=m.end();it++)
	cout<<it.first<<it.second
	for(auto it : m) cout<it.first<it.second;
Sorting:	convert map to vector<pair<>> and follow sorting of vector.

NOTE: Multimap is similar to map with an addition that multiple elements can have same keys. Also, it is NOT required that the key value and mapped value pair has to be unique in this case. One important thing to note about multimap is that multimap keeps all the keys in sorted order always.

Creation:	multimap<int,int> mm;
Add element:	mm.insert(pair <int, int> (1, 40));

Set:

Creation:	unordered_set<int > s;
	set<int,int> s; (similar to treeset in java)
Set array:	unordered_set<int,int> s[n];
Size:	s.size();
Add element:	s.insert(val);
Remove element:	s.erase(val);
Clear all elements:	s.clear();
Find element	auto it = s.find(val);
	if(it!=s.end()){ cout<<it.first<<it.second;}
Iterators:	begin(),end(),rbegin(),rend()
Accessing:	for(auto it=s.begin();it!=s.end();it++)
	cout<<it.first<<it.second
	for(auto it : m) cout<it.first<it.second;
Sorting:	convert set to vector<int> and follow sorting of vector.

Multiset:

Creation:	multiset<int > ms;
	multiset<int, greater<int>> ms;
Add element:	ms.insert(val);
Remove element:	ms.erase(val); // remove all elements of val.
	ms.erase(ms.find(val)); // remove only one instance
Clear all elements:	ms.clear();

Heaps/PriorityQueue:

Creation:	priority_queue<int> pq; //maxheap
	priority_queue<int,greater<int>> pq; //minheap
Size:	pq.size();
Add element:	pq.push(val);
Fetch element:	pq.top();
Clear all elements:	pq = priority_queue<int>();
Remove top element:	pq.pop();

Deque:

Creation:	deque<int> dq;
Size:	dq.size();
Add element:	dq.push_back(val);
	dq.push_front(val);
Fetch element:	dq.front(); //front element
	dq.back() // back element
	dq[index] // indexed element
Clear all elements:	dq.clear();
Remove element:	dq.pop_front(); //front element
	dq.pop_back() // back element
	dq.erase(index); // indexed element
Iterating:	for(auto it = dq.begin();it!=dq.end();it++)
	cout<< *it;

Queue:

Creation:	queue<int> q;
Size:	q.size();
Add element:	q.push(val);
Fetch element:	q.front(); //front element
	q.back() // back element
Clear all elements:	q = queue<int>();
Remove element:	q.pop(); //front element

Stack:

Creation:	stack<int> st;
Size:	st.size();
Add element:	st.push(val);
Fetch element:	st.top(); //top element
Clear all elements:	st = stack<int>();
Remove element:	st.pop();

Singly LinkedList:

Creation:	forward_list<int> lst;
Size:	st.size();
Add element:	lst.push_front(val);
	lst.insert_after(index,val);
Fetch element:	lst.pop_front();
	lst.erase_after(index,val);
	lst.remove(val);

Clear all elements:	st = stack<int>();
Remove element:	st.pop();
Iterating:	for(auto it = lst.begin();it!=lst.end();it++)
	cout<< *it;

Doubly LinkedList:

Creation:	list<int> dl;
Size:	dl.size();
Add element:	dl.push_front(val);
	dl.push_back(val);
Fetch element:	dl.front();
	dl.back();
Clear all elements:	dl.clear();
Remove element:	dl.pop_front();
	dl.pop_back();
Reverse:	dl.reverse();
Sort:	dl.sort();
Two sorted list into one:	dl1.merge(dl2);
Iterating:	for(auto it = dl.begin();it!=dl.end();it++)
	cout<< *it;

NON-MUTATING Algorithms:

Binary search

binary_search(startaddress, endaddress, valuetofind);

returns: true if an element equal to valuetofind is found, else false.

LowerBound

lower_bound(startaddress, endaddress, valuetofind);

returns: Returns pointer to “position of num” if container contains 1 occurrence of num.

Returns pointer to “first position of num” if container contains multiple occurrence of num.

Returns pointer to “position of next higher number than num” if container does not contain occurrence of num.

Subtracting the pointer to 1st position i.e “vect.begin()” returns the actual index.

UpperBound

upper_bound(startaddress, endaddress, valuetofind);

returns: Returns pointer to “position of next higher number than num” if container contains 1 occurrence of num.

Returns pointer to “first position of next higher number than last occurrence of num” if container contains multiple occurrence of num.

Returns pointer to “position of next higher number than num” if container does not contain occurrence of num.

Subtracting the pointer to 1st position i.e “vect.begin()” returns the actual index.

MaxElement and MinElement:

```
auto it = max_element(v.begin(),v.end());

cout<< *it ;

auto it = min_element(v.begin(),v.end());

cout<< *it ;
```

Find:

auto it = find (v.begin(),v.end());

if(it == v.end()) cout << ” Not Found ” ;

else cout << ” Found ” << (it – v.begin())

IsPermutation:

If(is_permutation (v1.begin(),v1.end(),v2.begin())) cout<<”true”;

Lexicographical compare:

if(lexicographical_compare(one, one+13, two, two+3)) cout<<”one small than two”;

f(lexicographical_compare(one, one+13, two, two+3,MyComp)) cout<<”one small than two”;

MUTATING Algorithms:

Sort:

sort (arr,arr+n);

sort (arr,arr+n ,greater);

Reverse:

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

NextPermutation:

next_permutation (v.begin(),v.end()); // input needs to be sorted

PrevPermutation:

prev_permutation (v.begin(),v.end()); // input needs to be sorted

MakeHeap

make_heap (v.begin(),v.end()); // maxheap

make_heap (v.begin(),v.end(),greater()); //minheap

MORE ON: https://www.geeksforgeeks.org/algorithms-library-c-stl/