STL C++

1.Vector:

S. No.	Function	Description	Syntax
1.	begin()	Returns an iterator to the first element.	v.begin()
2.	end()	Returns an iterator to the theoretical element after the last element.	v.end()
3.	size()	Returns the number of elements present.	v.size()
4.	empty()	Returns true if the vector is empty, false otherwise.	v.empty()
5.	at()	Return the element at a particular position.	v.at(index)
6.	assign()	Assign a new value to the vector elements.	v.assign(size, value);
7.	push_back()	Adds an element to the back of the vector.	v.push_back()
8.	pop_back()	Removes an element from the end.	v.pop_back()

S. No.	Function	Description	Syntax

9.	insert()	Insert an element at the specified position.	1.v.insert (begin()+pos, val);(single insert)
			2.v.insert(begin()+pos, n, val);(multiple insertion of same value)
			3. int main() {
			vector<int> vec{ 1, 2, 3, 4, 5 };
			vector<int> temp{ 2, 5, 9, 0, 3, 10 };
			// Defining range as the elements between the 1st
			// and 4th index of temp vector
			auto first = temp.begin() + 1;
			auto last = temp.begin() + 5;
			// Inserting elements from the range defined above at
			// the index 3
			vec.insert(vec.begin() + 3, first,
			last);

10.	erase()	Delete the elements at a specified position or range.	v.erase(pos); // Remove single element
			v.erase(first, last); // Erase multiple elements

11.	clear()	Removes all elements.	v.clear()
12.	swap()	To swap 2 vector	v1.swap(v2)
13.	swap()	To make the element of a vector reverse	reverse(vec.begin(), vec.end());

2.SET(set <data_type> set_name;)

Function	Description	
begin()	Returns an iterator to the first element in the set.	
end()	Returns an iterator to the theoretical element that follows the last element in the set.	
rbegin()	Returns a reverse iterator pointing to the last element in the container.	
rend()	Returns a reverse iterator pointing to the theoretical element right before the first element in the set container.	
crbegin()	Returns a constant iterator pointing to the last element in the container.	
crend()	Returns a constant iterator pointing to the position just before the first element in the container.	
cbegin()	Returns a constant iterator pointing to the	

Function	Description	
	first element in the container.	
cend()	Returns a constant iterator pointing to the position past the last element in the container.	
size()	Returns the number of elements in the set.	
max_size()	Returns the maximum number of elements that the set can hold.	
empty()	Returns whether the set is empty.	
insert(const g)	Adds a new element 'g' to the set.	
iterator insert (iterator position, const g)	Adds a new element 'g' at the position pointed by the iterator.	
erase(iterator position)	Removes the element at the position pointed by the iterator.	
erase(const g)	Removes the value 'g' from the set.	

Function	Description	
clear()	Removes all the elements from the set.	
key_comp() / value_comp()	Returns the object that determines how the elements in the set are ordered ('<' by default).	
find(const g)	Returns an iterator to the element 'g' in the set if found, else returns the iterator to the end.	
count(const g)	Returns 1 or 0 based on whether the element 'g' is present in the set or not.	
lower_bound(const g)	Returns an iterator to the first element that is equivalent to 'g' or definitely will not go before the element 'g' in the set.	
upper_bound(const g)	Returns an iterator to the first element that will go after the element 'g' in the set.	
equal_range()	The function returns an iterator of pairs. (key_comp). The pair refers to the range that includes all the elements in the container which	

Function	Description	
	have a key equivalent to k.	
emplace()	This function is used to insert a new element into the set container, only if the element to be inserted is unique and does not already exist in the set.	
emplace_hint()	Returns an iterator pointing to the position where the insertion is done. If the element passed in the parameter already exists, then it returns an iterator pointing to the position where the existing element is.	
swap()	This function is used to exchange the contents of two sets but the sets must be of the same type, although sizes may differ.	
operator=	The '=' is an operator in C++ STL that copies (or moves) a set to another set and set::operator= is the corresponding operator function.	

Function	Description	
get_allocator()	Returns the copy of the allocator object associated with the set.	

```
int main() {
    std::set<int> mySet = {5, 10, 15, 20, 25};

    // 1. begin() and end()
    std::cout << "Elements in mySet: ";
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 2. rbegin() and rend()
    std::cout << "Elements in reverse: ";
    for (auto it = mySet.rbegin(); it != mySet.rend(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 3. cbegin() and cend()
    std::cout << "Elements in mySet (using cbegin and cend): ";
    for (auto it = mySet.cbegin(); it != mySet.cend(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 4. crbegin() and crend()
    std::cout << "Elements in reverse (using crbegin and crend): ";
    for (auto it = mySet.crbegin(); it != mySet.crend(); ++it) {
```

```
    std::cout << *it << " ";
}
std::cout << std::endl;

// 5. size() and max_size()
std::cout << "Size of mySet: " << mySet.size() << std::endl;
std::cout << "Maximum possible size of mySet: " << mySet.max_size() << std::endl;

// 6. empty()
std::cout << "Is mySet empty? " << (mySet.empty() ? "Yes" : "No") << std::endl;

// 7. insert() - single element
mySet.insert(30);
std::cout << "After inserting 30, mySet: ";
for (int x : mySet) std::cout << x << " ";
std::cout << std::endl;

// 8. insert() - with position hint
auto it = mySet.insert(mySet.begin(), 7);
std::cout << "After inserting 7 at a position, mySet: ";
for (int x : mySet) std::cout << x << " ";
std::cout << std::endl;

// 9. erase() by iterator
mySet.erase(it); // Erases the element '7'
std::cout << "After erasing 7, mySet: ";
for (int x : mySet) std::cout << x << " ";
std::cout << std::endl;

// 10. erase() by value
mySet.erase(30); // Erases the element '30'
std::cout << "After erasing 30, mySet: ";
for (int x : mySet) std::cout << x << " ";
std::cout << std::endl;
```

```
// 11. clear()
std::set<int> tempSet = {1, 2, 3};
tempSet.clear();
std::cout << "tempSet cleared. Is it empty? " << (tempSet.empty() ? "Yes" : "No") << std::endl;

// 12. key_comp() and value_comp()
std::cout << "Using key comparison in mySet: ";
auto comp = mySet.key_comp();
for (auto it = mySet.begin(); comp(*it, 20); ++it) {
   std::cout << *it << " ";
}
std::cout << std::endl;

// 13. find()
auto found = mySet.find(15);
if (found != mySet.end()) {
   std::cout << "Element 15 found in mySet" << std::endl;
}

// 14. count()
std::cout << "Count of 10 in mySet: " << mySet.count(10) << std::endl;

// 15. lower_bound() and upper_bound()
std::cout << "Lower bound of 15: " << *mySet.lower_bound(15) << std::endl;
std::cout << "Upper bound of 15: " << *mySet.upper_bound(15) << std::endl;

// 16. equal_range()
auto range = mySet.equal_range(15);
std::cout << "Equal range of 15: " << *range.first << " to " << *range.second << std::endl;

// 17. emplace()
mySet.emplace(35);
std::cout << "After emplacing 35, mySet: ";
```

```
    for (int x : mySet) std::cout << x << " ";
    std::cout << std::endl;

    // 18. emplace_hint()
    mySet.emplace_hint(mySet.begin(), 40);
    std::cout << "After emplacing 40 with hint, mySet: ";
    for (int x : mySet) std::cout << x << " ";
    std::cout << std::endl;

    // 19. swap()
    std::set<int> anotherSet = {100, 200};
    mySet.swap(anotherSet);
    std::cout << "After swapping, mySet: ";
    for (int x : mySet) std::cout << x << " ";
    std::cout << "\nanotherSet: ";
    for (int x : anotherSet) std::cout << x << " ";
    std::cout << std::endl;

    // 20. operator=
    std::set<int> copySet = mySet;
    std::cout << "copySet (copied from mySet): ";
    for (int x : copySet) std::cout << x << " ";
    std::cout << std::endl;

    // 21. get_allocator()
    int* p = mySet.get_allocator().allocate(5);
    std::cout << "Allocated array using mySet's allocator." << std::endl;
    mySet.get_allocator().deallocate(p, 5);

    return 0;
}
```

MAP(Use to store unique pairs)

Function	Definition
map::insert()	Insert elements with a particular key in the map container –> O(log n)
map:: count()	Returns the number of matches to element with key-value 'g' in the map. –> O(log n)
map equal_range()	Returns an iterator of pairs. The pair refers to the bounds of a range that includes all the elements in the container which have a key equivalent to k.
map erase()	Used to erase elements from the container –> O(log n)
map rend()	Returns a reverse iterator pointing to the theoretical element right before the first key-value pair in the map(which is considered its reverse end).
map rbegin()	Returns a reverse iterator which points to the last element of the map.
map find()	Returns an iterator to the element with key-value 'g' in the map if found, else returns the iterator to end.
map crbegin() and crend()	crbegin() returns a constant reverse iterator referring to the last element in the map container. crend() returns a constant reverse iterator pointing to the theoretical element before the first element in the map.
map cbegin() and cend()	cbegin() returns a constant iterator referring to the first element in the map container. cend() returns a constant iterator pointing to the theoretical element that follows the last element in the multimap.
map emplace()	Inserts the key and its element in the map container.
map max_size()	Returns the maximum number of elements a map container can hold –> O(1)

map upper_bound()	Returns an iterator to the first element that is equivalent to mapped value with key-value 'g' or definitely will go after the element with key-value 'g' in the map
map operator=	Assigns contents of a container to a different container, replacing its current content.
map lower_bound()	Returns an iterator to the first element that is equivalent to the mapped value with key-value 'g' or definitely will not go before the element with key-value 'g' in the map –> O(log n)
map emplace_hint()	Inserts the key and its element in the map container with a given hint.
map value_comp()	Returns the object that determines how the elements in the map are ordered ('<' by default).
map key_comp()	Returns the object that determines how the elements in the map are ordered ('<' by default).
map::size()	Returns the number of elements in the map.
map::empty()	Returns whether the map is empty
map::begin() and end()	begin() returns an iterator to the first element in the map. end() returns an iterator to the theoretical element that follows the last element in the map
map::operator[]	This operator is used to reference the element present at the position given inside the operator.
map::clear()	Removes all the elements from the map.
map::at() and map::swap()	at() function is used to return the reference to the element associated with the key k. swap() function is used to exchange the contents of two maps but the maps must be of the same type, although sizes may differ.

```
// Creating the map (noteMap) and adding notes using insert() and emplace()
map<int, string> noteMap;
noteMap.insert({1, "Learn C++ basics"});
noteMap.insert({2, "Review map functions"});
noteMap.emplace(3, "Write code using map");

// Check if a note with ID 2 exists using count()
cout << "Number of notes with ID 2: " << noteMap.count(2) << endl;

// Using equal_range() to get range of elements with a specific key
auto range = noteMap.equal_range(2);
if (range.first != noteMap.end()) {
   cout << "Equal range for key 2: " << range.first->first << " -> " << range.first->second << endl;
}

// Using find() to locate a specific note
auto it = noteMap.find(3);
if (it != noteMap.end()) {
   cout << "Found note with ID 3: " << it->first << " -> " << it->second << endl;
}

// Adding another note and then using erase() to remove it
noteMap.insert({4, "Read about merge() function"});
noteMap.erase(4);
cout << "After erasing note with ID 4, size of noteMap: " << noteMap.size() << endl;

// Using rbegin(), rend(), crbegin(), and crend() for reverse traversal
cout << "Last note using rbegin(): " << noteMap.rbegin()->first << " -> " << noteMap.rbegin()->second << endl;
cout << "Constant reverse begin: " << noteMap.crbegin()->first << " -> " << noteMap.crbegin()->second << endl;

// Using cbegin() and cend() for constant iterators
cout << "First note using cbegin(): " << noteMap.cbegin()->first << " -> " << noteMap.cbegin()->second << endl;

// Using max_size() to check the theoretical maximum size of the map
cout << "Max size of noteMap: " << noteMap.max_size() << endl;
```

```
// Using lower_bound() and upper_bound() to get boundary notes
cout << "Lower bound of ID 2: " << noteMap.lower_bound(2)->first << " -> " << noteMap.lower_bound(2)->second << endl;
cout << "Upper bound of ID 1: " << noteMap.upper_bound(1)->first << " -> " << noteMap.upper_bound(1)->second << endl;

// Using operator= to assign noteMap to a new map (anotherMap)
map<int, string> anotherMap;
anotherMap = noteMap;
cout << "Another map after assignment:" << endl;
for (const auto& [key, value] : anotherMap) {
    cout << key << " -> " << value << endl;
}

// Using emplace_hint() to insert with a hint
auto hint = noteMap.find(3);
noteMap.emplace_hint(hint, 5, "Practice coding challenges");
cout << "After emplacing with hint, note with ID 5: " << noteMap[5] << endl;

// Using value_comp() and key_comp() to compare elements
auto comp = noteMap.value_comp();
if (comp(*noteMap.begin(), *noteMap.rbegin())) {
    cout << "First note is less than last note based on value_comp()." << endl;
}

// Using size() and empty()
cout << "Current size of noteMap: " << noteMap.size() << endl;
cout << "Is noteMap empty? " << (noteMap.empty() ? "Yes" : "No") << endl;

// Using begin() and end() to print all notes
cout << "All notes:" << endl;
for (auto it = noteMap.begin(); it != noteMap.end(); ++it) {
    cout << it->first << " -> " << it->second << endl;
}

// Using operator[] to update a note
noteMap[1] = "Learn C++ thoroughly";
cout << "Updated note with ID 1: " << noteMap[1] << endl;
```

```
// Using clear() to remove all notes from anotherMap
anotherMap.clear();
cout << "After clearing, is anotherMap empty? " << (anotherMap.empty() ? "Yes" : "No") << endl;

// Using at() and swap() functions
cout << "Note at ID 3: " << noteMap.at(3) << endl;
noteMap.swap(anotherMap);
cout << "After swapping, is noteMap empty? " << (noteMap.empty() ? "Yes" : "No") << endl;
cout << "Size of anotherMap after swap: " << anotherMap.size() << endl;

// Using try_emplace() to insert a note if ID does not exist
anotherMap.try_emplace(6, "Study advanced C++ topics");
cout << "After try_emplace with ID 6: " << anotherMap[6] << endl;

// Using merge() to combine two maps
map<int, string> additionalNotes = {{7, "Review data structures"}, {8, "Prepare for exams"}};
anotherMap.merge(additionalNotes);
cout << "After merging additional notes:" << endl;
for (const auto& [key, value] : anotherMap) {
    cout << key << " -> " << value << endl;
}

// Using contains() (C++20) to check existence of a note
if (anotherMap.contains(7)) {
    cout << "anotherMap contains note with ID 7." << endl;
}

// Using extract() to remove a note and manipulate it separately
auto node = anotherMap.extract(7);
if (!node.empty()) {
    cout << "Extracted note with ID " << node.key() << ": " << node.mapped() << endl;
    // Modify the note outside the map and reinsert
    node.mapped() = "Data structures review updated";
    anotherMap.insert(move(node));
    cout << "Updated note with ID 7: " << anotherMap[7] << endl;}
```

STRING

```
int main() {
    // Creating a string
    string str = "Hello, World!";

    // Capacity Functions
    cout << "Initial string: " << str << endl;
    cout << "String capacity: " << str.capacity() << endl;
    cout << "String length: " << str.length() << endl;
    cout << "String max size: " << str.max_size() << endl;
    cout << "Is string empty? " << (str.empty() ? "Yes" : "No") << endl;

    // Modifiers
    str.clear(); // clear
    cout << "After clear, is string empty? " << (str.empty() ? "Yes" : "No") << endl;

    str = "Hello, World!";
    str.insert(7, "beautiful "); // insert
    cout << "After insert: " << str << endl;

    str.push_back('!'); // push_back
    cout << "After push_back: " << str << endl;

    str.pop_back(); // pop_back
    cout << "After pop_back: " << str << endl;

    str.append(" Let's code."); // append
    cout << "After append: " << str << endl;

 str.replace(7, 9, "awesome"); // replace
    cout << "After replace: " << str << endl;
```

```
str1.replace(pos, n, m, c)                          // Replace with character
str1.replace(pos, n, str2)                          // Replace with string
str1.replace(pos1, n, str2, pos2,m)                 // Replace with substring
str1.replace(first, last, n, c);                    // Replace Character
str1.replace(first, last, str2)                     // Replace String
str1.replace (first, last, str2_first, str2_last);  // Replace Substring
```

```
str.erase(13, 5); // erase
cout << "After erase: " << str << endl;

str += " and learn."; // operator+=
cout << "After operator+=: " << str << endl;

// Capacity Resizing
str.resize(5); // resize
cout << "After resize(5): " << str << endl;

str.shrink_to_fit(); // shrink_to_fit
cout << "After shrink_to_fit, capacity: " << str.capacity() << endl;

str = "Hello, World!";

// String Operations
cout << "Character at index 4 (str.at(4)): " << str.at(4) << endl; // at

cout << "First character (str.front()): " << str.front() << endl; // front
cout << "Last character (str.back()): " << str.back() << endl; // back

const char* cstr = str.c_str(); // c_str
cout << "C-string representation: " << cstr << endl;

const char* data = str.data(); // data
cout << "Data representation: " << data << endl;

// Comparison
string str2 = "Hello, World!";
cout << "Compare str and str2: " << str.compare(str2) << endl; // compare

str.swap(str2); // swap
```

```
cout << "After swap with str2, str: " << str << endl;
cout << "After swap with str2, str2: " << str2 << endl;

// Substring
string subStr = str.substr(7, 5); // substr
cout << "Substring (7, 5): " << subStr << endl;

// Find Functions
size_t pos = str.find("World"); // find
if (pos != string::npos) {
    cout << "'World' found at position: " << pos << endl;
}

pos = str.rfind("o"); // rfind
if (pos != string::npos) {
    cout << "Last occurrence of 'o' at position: " << pos << endl;
}

pos = str.find_first_of("aeiou"); // find_first_of
if (pos != string::npos) {
    cout << "First vowel found at position: " << pos << endl;
}

pos = str.find_last_of("aeiou"); // find_last_of
if (pos != string::npos) {
    cout << "Last vowel found at position: " << pos << endl;
}

pos = str.find_first_not_of("Hello, "); // find_first_not_of
if (pos != string::npos) {
    cout << "First non 'Hello, ' character at position: " << pos << endl;
}
```

```
    pos = str.find_last_not_of("!"); // find_last_not_of
    if (pos != string::npos) {
        cout << "Last non '!' character at position: " << pos << endl;
    }

    // Conversion
    string numStr = "12345";
    int num = stoi(numStr); // stoi
    cout << "Integer from string: " << num << endl;

    float fnum = stof(numStr); // stof
    cout << "Float from string: " << fnum << endl;

    numStr = to_string(num); // to_string
    cout << "String from integer: " << numStr << endl;

    // Iterator Functions
    cout << "Using begin and end iterators: ";
    for (auto it = str.begin(); it != str.end(); ++it) {
        cout << *it;
    }
    cout << endl;

    cout << "Using reverse iterators: ";
    for (auto rit = str.rbegin(); rit != str.rend(); ++rit) {
        cout << *rit;
    }
    cout << endl;

    return 0;
}
```

STACK

```
// Creating a stack of integers
stack<int> myStack;

// Checking if the stack is empty
cout << "Is the stack empty? " << (myStack.empty() ? "Yes" : "No") << endl;

// Using push() to add elements to the stack
myStack.push(10);
myStack.push(20);
myStack.push(30);

// Using size() to get the number of elements
cout << "Stack size after pushing 3 elements: " << myStack.size() << endl;

// Accessing the top element using top()
cout << "Top element: " << myStack.top() << endl;

// Using emplace() to add an element
myStack.emplace(40);
cout << "Top element after emplace(40): " << myStack.top() << endl;

// Using pop() to remove the top element
myStack.pop();
cout << "Top element after pop: " << myStack.top() << endl;

// Creating another stack and using swap()
stack<int> anotherStack;
anotherStack.push(100);
anotherStack.push(200);

cout << "Before swapping, top of myStack: " << myStack.top() << endl;
cout << "Before swapping, top of anotherStack: " << anotherStack.top() << endl;

// Swapping stacks
myStack.swap(anotherStack);
```

```
    cout << "After swapping, top of myStack: " << myStack.top() << endl;
    cout << "After swapping, top of anotherStack: " << anotherStack.top() << endl;

    // Clearing out the stack
    while (!myStack.empty()) {
        cout << "Popping element: " << myStack.top() << endl;
        myStack.pop();
    }

    cout << "Is myStack empty after popping all elements? " << (myStack.empty() ? "Yes" : "No") << endl;

    return 0;
}
```

QUEUE

```
    // Creating a queue of integers
    queue<int> myQueue;

    // Checking if the queue is empty
    cout << "Is the queue empty? " << (myQueue.empty() ? "Yes" : "No") << endl;

    // Using push() to add elements to the queue
    myQueue.push(10);
    myQueue.push(20);
    myQueue.push(30);

    // Using size() to get the number of elements
    cout << "Queue size after pushing 3 elements: " << myQueue.size() << endl;

    // Accessing the front and back elements
    cout << "Front element: " << myQueue.front() << endl;
    cout << "Back element: " << myQueue.back() << endl;

    // Using emplace() to add an element
    myQueue.emplace(40);
    cout << "Back element after emplace(40): " << myQueue.back() << endl;
```

```
    // Using pop() to remove the front element
    myQueue.pop();
    cout << "Front element after pop: " << myQueue.front() << endl;

    // Creating another queue and using swap()
    queue<int> anotherQueue;
    anotherQueue.push(100);
    anotherQueue.push(200);

    cout << "Before swapping, front of myQueue: " << myQueue.front() << endl;
    cout << "Before swapping, front of anotherQueue: " << anotherQueue.front() << endl;

    // Swapping queues
    myQueue.swap(anotherQueue);
    cout << "After swapping, front of myQueue: " << myQueue.front() << endl;
    cout << "After swapping, front of anotherQueue: " << anotherQueue.front() << endl;

    // Clearing out the queue
    while (!myQueue.empty()) {
        cout << "Popping element: " << myQueue.front() << endl;
        myQueue.pop();
    }

    cout << "Is myQueue empty after popping all elements? " << (myQueue.empty() ? "Yes" : "No") << endl;

    return 0;
}
```

Pair

```
    // 1. Constructors
    pair<int, string> p1; // default constructor
    pair<int, string> p2(10, "hello"); // parameterized constructor
    pair<int, string> p3(p2); // copy constructor
    pair<int, string> p4 = make_pair(20, "world"); // make_pair function

    cout << "p2: (" << p2.first << ", " << p2.second << ")" << endl;
    cout << "p3: (" << p3.first << ", " << p3.second << ")" << endl;
```

```
    cout << "p4: (" << p4.first << ", " << p4.second << ")" << endl;

    // 2. Accessing elements
    cout << "p2 first: " << p2.first << ", second: " << p2.second << endl;

    // 3. swap()
    cout << "Before swap: p2: (" << p2.first << ", " << p2.second << "), p4: (" << p4.first << ", " << p4.second << ")" << endl;
    p2.swap(p4);
    cout << "After swap: p2: (" << p2.first << ", " << p2.second << "), p4: (" << p4.first << ", " << p4.second << ")" << endl;

    // 4. Relational Operators
    pair<int, char> p5(1, 'a');
    pair<int, char> p6(2, 'b');

    cout << "p5 == p6: " << (p5 == p6 ? "true" : "false") << endl;
    cout << "p5 != p6: " << (p5 != p6 ? "true" : "false") << endl;
    cout << "p5 < p6: " << (p5 < p6 ? "true" : "false") << endl;
    cout << "p5 <= p6: " << (p5 <= p6 ? "true" : "false") << endl;
    cout << "p5 > p6: " << (p5 > p6 ? "true" : "false") << endl;
    cout << "p5 >= p6: " << (p5 >= p6 ? "true" : "false") << endl;

    return 0;
}
```

Unordered Set

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> s = {1, 2, 3, 4, 5};

    // Insert elements
    s.insert(6);
    s.emplace(7);

    // Find element
    if (s.find(3) != s.end()) {
```

```
    std::cout << "3 found in the set\n";
  }

  // Check if element exists using count
  if (s.count(4)) {
    std::cout << "4 is in the set\n";
  }

  // Bucket information
  std::cout << "Number of buckets: " << s.bucket_count() << std::endl;
  std::cout << "Load factor: " << s.load_factor() << std::endl;

  // Hash function and key equality
  auto hash_func = s.hash_function();
  std::cout << "Hash for 10: " << hash_func(10) << std::endl;

  // Erase element
  s.erase(2);

  // Display elements
  for (const auto& elem : s) {
    std::cout << elem << " ";
  }
  std::cout << std::endl;

  return 0;
}
```

Multiset

```
#include <iostream>
#include <set>

int main() {
  std::multiset<int> ms = {1, 2, 2, 3, 3, 3, 4};

  // Inserting elements
  ms.insert(5);
```

```
    ms.insert(3); // Duplicate allowed

    // Displaying elements
    std::cout << "Elements in the multiset: ";
    for (auto it = ms.begin(); it != ms.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // Count occurrences of an element
    std::cout << "Count of 3: " << ms.count(3) << std::endl;

    // Find an element
    auto it = ms.find(3);
    if (it != ms.end()) {
        std::cout << "Found 3 at position: " << *it << std::endl;
    }

    // Erase an element
    ms.erase(2);  // Erases all occurrences of 2

    // Display elements after erase
    std::cout << "After erasing 2: ";
    for (auto it = ms.begin(); it != ms.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // Lower and Upper Bound example
    auto lb = ms.lower_bound(3); // First element not less than 3
    auto ub = ms.upper_bound(3); // First element greater than 3
    std::cout << "Lower bound of 3: " << *lb << std::endl;
    std::cout << "Upper bound of 3: " << *ub << std::endl;

    return 0;
}
```

Priority queue

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional> // For std::greater and custom comparator

// Custom comparator for min-heap
struct CompareMinHeap {
    bool operator()(int a, int b) {
        return a > b;  // Min-heap: smaller values have higher priority
    }
};

int main() {
    // Max-Heap Priority Queue (default)
    std::priority_queue<int> maxHeap;

    // Insert elements into max heap
    maxHeap.push(10);
    maxHeap.push(30);
    maxHeap.push(20);
    maxHeap.push(5);

    std::cout << "Max-Heap Priority Queue:" << std::endl;
    std::cout << "Top element: " << maxHeap.top() << std::endl; // Access top element
    std::cout << "Size of max heap: " << maxHeap.size() << std::endl; // Get size
    std::cout << "Is max heap empty? " << (maxHeap.empty() ? "Yes" : "No") << std::endl; // Check if empty

    // Pop elements from max heap
    while (!maxHeap.empty()) {
        std::cout << maxHeap.top() << " ";  // Access top element
        maxHeap.pop();  // Remove top element
    }
    std::cout << std::endl;

    // Min-Heap Priority Queue (using custom comparator)
```

```
std::priority_queue<int, std::vector<int>, CompareMinHeap> minHeap;

// Insert elements into min heap
minHeap.push(10);
minHeap.push(30);
minHeap.push(20);
minHeap.push(5);

std::cout << "Min-Heap Priority Queue:" << std::endl;
std::cout << "Top element: " << minHeap.top() << std::endl; // Access top element
std::cout << "Size of min heap: " << minHeap.size() << std::endl; // Get size
std::cout << "Is min heap empty? " << (minHeap.empty() ? "Yes" : "No") << std::endl; // Check if empty

// Pop elements from min heap
while (!minHeap.empty()) {
    std::cout << minHeap.top() << " ";  // Access top element
    minHeap.pop();  // Remove top element
}
std::cout << std::endl;

// Priority queue with custom data type and comparator
struct Task {
    int priority;
    std::string name;

    Task(int p, std::string n) : priority(p), name(n) {}
};

struct CompareTask {
    bool operator()(const Task& t1, const Task& t2) {
        return t1.priority > t2.priority;  // Lower priority number = higher priority (min-heap behavior)
    }
};

std::priority_queue<Task, std::vector<Task>, CompareTask> taskQueue;
```

```
    // Insert tasks
    taskQueue.push(Task(3, "Task 3"));
    taskQueue.push(Task(1, "Task 1"));
    taskQueue.push(Task(2, "Task 2"));

    std::cout << "Task Queue (Min-Heap based on priority):" << std::endl;
    while (!taskQueue.empty()) {
        Task t = taskQueue.top();
        std::cout << t.name << " with priority " << t.priority << std::endl;
        taskQueue.pop();
    }

    return 0;
}
```

Multimap

```
#include <iostream>
#include <map>

int main() {
    // Declare a multimap of int keys and string values
    std::multimap<int, std::string> myMultimap;

    // Insert elements into the multimap
    myMultimap.insert({1, "Apple"});
    myMultimap.insert({2, "Banana"});
    myMultimap.insert({2, "Blueberry"});
    myMultimap.insert({3, "Cherry"});
    myMultimap.insert({3, "Coconut"});
    myMultimap.insert({4, "Date"});

    // Display all elements in the multimap
    std::cout << "All elements in the multimap:" << std::endl;
    for (const auto& pair : myMultimap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second << std::endl;
    }
    std::cout << std::endl;
```

```
// 1. `count` function to count elements with a specific key
int key_to_count = 2;
int count = myMultimap.count(key_to_count);
std::cout << "Count of elements with key " << key_to_count << ": " << count << std::endl;

// 2. `find` function to find the first occurrence of a specific key
int key_to_find = 3;
auto it = myMultimap.find(key_to_find);
if (it != myMultimap.end()) {
    std::cout << "Found element with key " << key_to_find << ": " << it->second << std::endl;
} else {
    std::cout << "Key " << key_to_find << " not found." << std::endl;
}

// 3. `equal_range` to retrieve all elements with a specific key
int key_to_equal_range = 3;
auto range = myMultimap.equal_range(key_to_equal_range);
std::cout << "Elements with key " << key_to_equal_range << ":" << std::endl;
for (auto it = range.first; it != range.second; ++it) {
    std::cout << "Value: " << it->second << std::endl;
}

// 4. `erase` function to remove elements by key
int key_to_erase = 2;
myMultimap.erase(key_to_erase);
std::cout << "Multimap after erasing elements with key " << key_to_erase << ":" << std::endl;
for (const auto& pair : myMultimap) {
    std::cout << "Key: " << pair.first << ", Value: " << pair.second << std::endl;
}

// 5. `clear` function to remove all elements from the multimap
myMultimap.clear();
std::cout << "Multimap size after clear: " << myMultimap.size() << std::endl;

// 6. `insert` with hint (insert element at position given by hint)
```

```
    auto hint = myMultimap.insert(myMultimap.begin(), {5, "Elderberry"});
    std::cout << "Inserted with hint at key " << hint->first << " with value " << hint->second << std::endl;

    // Display all elements in the multimap after inserting with hint
    std::cout << "All elements after hint insertion:" << std::endl;
    for (const auto& pair : myMultimap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second << std::endl;
    }

    return 0;
}
```