C++ Standard Template Library

1 Vector (Dynamic Array)

1.1 Syntax

```
vector<dataType> vectorName;

vector<int> vectorName = {1, 2, 3, 4}; // Initialization is
    optional

vector<int> vectorName(size, valueAtEveryIndex); // e.g., vector<
    int> vectorName(3, 10) creates a vector of size 3 with each
    element initialized to 10.

vector<int> vector1 = {1, 2, 3};
vector<int> copyVector1(vector1);
```

1.2 Description

A `vector`'s capacity doubles when it gets full. Data from the old vector is copied into a new vector that is double the size of the old vector.
Vectors have two properties:

- **Size**: The number of elements currently present in the vector.
- **Capacity**: The total number of elements a vector can hold before it needs to resize.

1.3 Functions

1. `size()`: Returns the number of elements in the vector.
2. `capacity()`: Returns the current capacity of the vector.
3. `push_back(dataToBePushed)`: Adds an element to the end of the vector.
4. `pop_back()`: Removes the last element of the vector.
5. `emplace_back()`: Constructs an element in-place and adds it to the end of the vector (more efficient than `push_back` for complex objects).

6. `at(indexNumber)`: Returns a reference to the element at the specified position.

7. `front()`: Returns a reference to the first element in the vector.

8. `back()`: Returns a reference to the last element in the vector.

9. `erase(startingIndex, endingIndex (optional))`: Removes elements in the specified range. The ending index is optional.

   ```
   vectorName.erase(vectorName.begin() + indexNumber); //
       Removes element from a specified location
   vectorName.erase(vectorName.begin() + indexNumber,
       vectorName.begin() + indexNumber); // Removes elements
        from a specified range
   vectorName.erase(vectorName.begin(), vectorName.end()); //
        Removes all the elements
   ```

   Note: The `erase()` function changes the size of the vector, not its capacity.

10. `insert(indexNumber, value)`: Inserts an element at the specified position.

    ```
    vectorName.insert(vectorName.begin() + indexNumber, value)
        ;
    ```

11. `clear()`: Removes all elements from the vector.

12. `empty()`: Returns a boolean value, 1 if the condition is true meaning the vector does not contain any elements, and 0 if the condition is false meaning the vector has some elements.

    Note: The functions from 13 to 18 are known as vector iterators.

13. `begin()`: Returns an iterator to the first element of the vector.

    An iterator is an object that can be used like a pointer to traverse elements, but it isn't simply a raw pointer.

14. `end()`: Returns an iterator to the element following the last element of the vector, which acts as a boundary.

    Usage of iterator functions:

    ```
    vector<int>::iterator i;
    for(i = vectorName.begin(); i != vectorName.end(); i++) {
        cout << *(i) << endl;
    }

    for(vector<int>::iterator i = vectorName.begin(); i !=
        vectorName.end(); i++) {
        cout << *(i) << endl;
    }
    ```

```
for(auto i = vectorName.begin(); i != vectorName.end(); i
    ++) {
    cout << *(i) << endl;
}
```

15. `rbegin()`: Returns a reverse iterator to the last element of the vector.
16. `rend()`: Returns a reverse iterator to the element preceding the first element of the vector.

    Usage of reverse iterator functions:

    ```
    vector<int>::reverse_iterator i;
    for(i = vectorName.rbegin(); i != vectorName.rend(); i++)
        {
        cout << *(i) << endl;
    }
    ```

17. `cbegin()`: Returns a constant iterator to the first element of the vector.
18. `cend()`: Returns a constant iterator to the element following the last element of the vector.

    Usage of constant iterator functions:

    ```
    vector<int>::const_iterator i;
    for(i = vectorName.cbegin(); i != vectorName.cend(); i++)
        {
        cout << *(i) << endl;
    }
    ```

2 List

2.1 Syntax

```
list<dataType> listName;

list<int> listName = {1, 2, 3, 4}; // Initialization is optional

list<int> listName(size, valueAtEveryIndex); // e.g., list<int>
    listName(3, 10) creates a list of size 3 with each element
    initialized to 10.

list<int> list1 = {1, 2, 3};
list<int> copylist1(list1);
```

2.2 Description

List is implemented as a doubly linked list.
Lists have two properties:

- **Size**: The number of elements currently present in the list.
- **Capacity**: Lists do not have a capacity in the same way vectors do, as elements are dynamically allocated when needed, and they can grow and shrink as required.

2.3 Functions

1. `size()`: Returns the number of elements in the list.
2. `push_front(dataToBePushed)`: Adds an element to the front of the list.
3. `pop_front()`: Removes the first element of the list.
4. `emplace_front()`: Constructs an element in-place and adds it to the front of the list (more efficient than `push_front` for complex objects).
5. `push_back(dataToBePushed)`: Adds an element to the end of the list.
6. `pop_back()`: Removes the last element of the list.
7. `emplace_back()`: Constructs an element in-place and adds it to the end of the list (more efficient than `push_back` for complex objects).
8. `front()`: Returns a reference to the first element in the list.
9. `back()`: Returns a reference to the last element in the list.
10. `erase(startingIndex, endingIndex (optional))`: Removes elements in the specified range. The ending index is optional.

```
listName.erase(listName.begin() + indexNumber); // Removes
     element from a specified location
listName.erase(listName.begin() + indexNumber, listName.
    begin() + indexNumber); // Removes elements from a
    specified range
listName.erase(listName.begin(), listName.end()); //
    Removes all the elements
```

Note: The `erase()` function changes the size of the list, not its capacity.

11. `insert(indexNumber, value)`: Inserts an element at the specified position.

```
listName.insert(listName.begin() + indexNumber, value);
```

12. `clear()`: Removes all elements from the list. This is an efficient operation for lists, as they do not require reallocation like vectors.

13. `empty()`: Returns a boolean value, 1 if the condition is true meaning the list does not contain any elements, and 0 if the condition is false meaning the list has some elements.

    Note:The functions from 13 to 18 are known as list iterators.

14. `begin()`: Returns an iterator to the first element of the list. Lists support bidirectional iteration, allowing you to traverse both forward and backward.

    An iterator is an object that can be used like a pointer to traverse elements, but it isn't simply a raw pointer.

15. `end()`: Returns an iterator to the element following the last element of the list, which acts as a boundary.

    Usage of iterator functions:

```
list<int>::iterator i;
for(i = listName.begin(); i != listName.end(); i++) {
    cout << *(i) << endl;
}

for(list<int>::iterator i = listName.begin(); i !=
    listName.end(); i++) {
    cout << *(i) << endl;
}

for(auto i = listName.begin(); i != listName.end(); i++) {
    cout << *(i) << endl;
}
```

16. `rbegin()`: Returns a reverse iterator to the last element of the list.

17. `rend()`: Returns a reverse iterator to the element preceding the first element of the list.

    Usage of reverse iterator functions:

```
list<int>::reverse_iterator i;
for(i = listName.rbegin(); i != listName.rend(); i++) {
    cout << *(i) << endl;
}
```

18. `cbegin()`: Returns a constant iterator to the first element of the list.

19. `cend()`: Returns a constant iterator to the element following the last element of the list.

    Usage of constant iterator functions:

    ```
    list<int>::const_iterator i;
    for(i = listName.cbegin(); i != listName.cend(); i++) {
        cout << *(i) << endl;
    }
    ```

3 Deque

3.1 Syntax

```
deque<dataType> dequeName;

deque<int> dequeName = {1, 2, 3, 4}; // Initialization is optional

deque<int> dequeName(size, valueAtEveryIndex); // e.g., deque<int>
    dequeName(3, 10) creates a deque of size 3 with each element
    initialized to 10.

deque<int> deque1 = {1, 2, 3};
deque<int> copyDeque1(deque1);
```

3.2 Description

Deque (double-ended queue) is a sequence container that allows fast insertions and deletions at both the beginning and the end. It is implemented using dynamic array.
Deques have two properties:

- **Size**: The number of elements currently present in the deque.
- **Dynamic Growth**: Deques grow dynamically as needed, but this is managed internally and does not have a "capacity" function. Unlike vectors, deques don't guarantee a contiguous block of memory.

3.3 Functions

1. `size()`: Returns the number of elements in the deque.
2. push_front(dataToBePushed): Adds an element to the front of the deque.
3. `pop_front()`: Removes the first element of the deque.
4. `push_back(dataToBePushed)`: Adds an element to the end of the deque.
5. `pop_back()`: Removes the last element of the deque.
6. `front()`: Returns a reference to the first element in the deque.
7. `back()`: Returns a reference to the last element in the deque.
8. `operator[]`: Allows direct access to elements at a specified index. Does not check for out-of-bounds access.

   ```
   cout << dequeName[2];  // Outputs the element at index 2
   ```

9. `at()`: Allows access to elements at a specified index with bounds checking. Throws `std::out_of_range` exception if the index is invalid.

```
cout << dequeName.at(2);  // Outputs the element at index
    2
try {
    cout << dequeName.at(10);  // Throws out_of_range
        exception
} catch (const std::out_of_range& e) {
    cout << e.what();  // Catching exception
}
```

10. `insert(indexNumber, value)`: Inserts an element at the specified position in the deque.

    ```
    dequeName.insert(dequeName.begin() + indexNumber, value);
    ```

11. `erase(startingIndex, endingIndex (optional))`: Removes elements in the specified range. The ending index is optional.

    ```
    dequeName.erase(dequeName.begin() + indexNumber); //
        Removes element from a specified location
    dequeName.erase(dequeName.begin() + startIndex, dequeName.
        begin() + endIndex); // Removes elements from a
        specified range
    dequeName.erase(dequeName.begin(), dequeName.end()); //
        Removes all the elements
    ```

    Note: The `erase()` function changes the size of the deque, not its capacity.

12. `clear()`: Removes all elements from the deque. This operation does not require reallocation like vectors.

13. `empty()`: Returns a boolean value: 1 if the deque is empty, and 0 if it contains elements.

14. `begin()`: Returns an iterator to the first element of the deque. Deques support bidirectional iteration, allowing you to traverse both forward and backward.

    An iterator is an object that can be used like a pointer to traverse elements, but it isn't simply a raw pointer.

15. `end()`: Returns an iterator to the element following the last element of the deque, which acts as a boundary.

    Usage of iterator functions:

    ```
    deque<int>::iterator i;
    for(i = dequeName.begin(); i != dequeName.end(); i++) {
        cout << *(i) << endl;
    }

    for(deque<int>::iterator i = dequeName.begin(); i !=
        dequeName.end(); i++) {
        cout << *(i) << endl;
    }
    ```

```
for(auto i = dequeName.begin(); i != dequeName.end(); i++)
     {
    cout << *(i) << endl;
}
```

16. `rbegin()`: Returns a reverse iterator to the last element of the deque.
17. `rend()`: Returns a reverse iterator to the element preceding the first element of the deque.

    Usage of reverse iterator functions:

    ```
    deque<int>::reverse_iterator i;
    for(i = dequeName.rbegin(); i != dequeName.rend(); i++) {
        cout << *(i) << endl;
    }
    ```

18. `cbegin()`: Returns a constant iterator to the first element of the deque.
19. `cend()`: Returns a constant iterator to the element following the last element of the deque.

    Usage of constant iterator functions:

    ```
    deque<int>::const_iterator i;
    for(i = dequeName.cbegin(); i != dequeName.cend(); i++) {
        cout << *(i) << endl;
    }
    ```

4 Pair

4.1 Syntax

```
pair<dataType1, dataType2> pairName;

pair<int, string> pairName = {1, "apple"}; // Initialization with
    values

pair<int, string> pair1 = {1, "apple"};

pair<int, string> pair2(pair1);  // Copy initialization
```

4.2 Description

A `pair` is a simple container in C++ that holds exactly two elements. These elements can be of different types, making `pair` useful for storing related data, such as key-value pairs in a map or two coordinates.
Pairs have the following characteristics:

- **First Element**: The first element of the pair, accessible via `first`.
- **Second Element**: The second element of the pair, accessible via `second`.
- **Lexicographical Comparison**: Pairs can be compared lexicographically, meaning the first element is compared first, and if equal, the second element is compared.

4.3 Functions

1. `first`: Accesses the first element of the pair.

   ```
   pair<int, string> p = {1, "apple"};
   cout << p.first;  // Outputs: 1
   ```

2. `second`: Accesses the second element of the pair.

   ```
   cout << p.second;  // Outputs: "apple"
   ```

3. `make_pair(value1, value2)`: Creates and returns a pair initialized with `value1` and `value2`.

   ```
   auto p = make_pair(10, "banana");  // Creates a pair <int,
        string>
   cout << p.first << ", " << p.second;  // Outputs: 10,
       banana
   ```

4. Comparison Operators (`==`, `!=`, `<`, `>`, `<=`, `>=`): Compares two pairs lexicographically, starting with the first element and then comparing the second if the first is equal.

```
pair<int, string> p1 = {1, "apple"};
pair<int, string> p2 = {2, "banana"};
cout << (p1 < p2);  // Outputs: 1 (true, since 1 < 2)
```

5. `swap(pair1, pair2)`: Swaps the contents of two pairs.

```
pair<int, string> p1 = {1, "apple"};
pair<int, string> p2 = {2, "banana"};
swap(p1, p2);
cout << p1.first << ", " << p1.second;  // Outputs: 2,
    banana
cout << p2.first << ", " << p2.second;  // Outputs: 1,
    apple
```

6. `tie(variable1, variable2)`: Unpacks the pair into separate variables.

```
pair<int, string> p = {10, "apple"};
int x;
string y;
tie(x, y) = p;
cout << x << ", " << y;  // Outputs: 10, apple
```

7. `get<N>(pairName)`: Accesses the `N`-th element (where `N` is either 0 or 1) of the pair.

```
pair<int, string> p = {1, "apple"};
cout << get<0>(p);  // Outputs: 1 (first element)
cout << get<1>(p);  // Outputs: "apple" (second element)
```

8. `const` pairs: Declares a pair as constant so that its values cannot be modified.

```
const pair<int, string> p = {1, "apple"};
// p.first = 2;   // Error: cannot modify a const pair
```

9. `pair` in `std::map` or `std::unordered_map`: Used to represent key-value pairs in these containers.

```
std::map<int, string> myMap;
myMap[1] = "apple";
myMap[2] = "banana";
for (const auto& p : myMap) {
    cout << p.first << ": " << p.second << endl;
}
// Outputs:
// 1: apple
// 2: banana
```

10. Sorting Pairs: Sorting based on either the first or second element.

    Using `std::sort()` to sort pairs:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    vector<pair<int, string>> v = {{2, "banana"}, {1, "
        apple"}, {3, "cherry"}};

    // Sorting by the first element (default behavior)
    sort(v.begin(), v.end());
    for (const auto& p : v) {
        cout << p.first << ": " << p.second << endl;
    }
    // Outputs:
    // 1: apple
    // 2: banana
    // 3: cherry
}
```

Custom Sorting by the Second Element: To sort pairs based on the second element, you can provide a custom comparator.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool compareBySecond(const pair<int, string>& p1, const
    pair<int, string>& p2) {
    return p1.second < p2.second;  // Sorts based on the
        second element (alphabetically)
}

int main() {
    vector<pair<int, string>> v = {{2, "banana"}, {1, "
        apple"}, {3, "cherry"}};

    // Sorting by the second element using custom
        comparator
    sort(v.begin(), v.end(), compareBySecond);
    for (const auto& p : v) {
        cout << p.first << ": " << p.second << endl;
    }
    // Outputs:
    // 1: apple
    // 3: cherry
    // 2: banana
}
```

Sorting in Descending Order with `std::greater<int>`:

The `std::greater<int>` comparator can be used to sort in descending order, either based on the first element of the pair or the second.

- Sort by First Element (Descending):

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>  // For std::greater

using namespace std;

int main() {
    vector<pair<int, string>> v = {{2, "banana"}, {1, "
        apple"}, {3, "cherry"}};

    // Sorting by the first element in descending order
    sort(v.begin(), v.end(), greater<pair<int, string>>())
        ;
    for (const auto& p : v) {
        cout << p.first << ": " << p.second << endl;
    }
    // Outputs:
    // 3: cherry
    // 2: banana
    // 1: apple
}
```

- Sort by Second Element (Descending): You can also use ‘std::greater‘ to sort based on the second element of the pair.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>  // For std::greater

using namespace std;

bool compareBySecondGreater(const pair<int, string>& p1,
    const pair<int, string>& p2) {
    return p1.second > p2.second;  // Sorts based on the
        second element in descending order
}

int main() {
    vector<pair<int, string>> v = {{2, "banana"}, {1, "
        apple"}, {3, "cherry"}};

    // Sorting by the second element in descending order
    sort(v.begin(), v.end(), compareBySecondGreater);
    for (const auto& p : v) {
        cout << p.first << ": " << p.second << endl;
    }
    // Outputs:
    // 2: banana
    // 3: cherry
    // 1: apple
}
```

11. `push_back() vs emplace_back() in vector pair`: emplace_back function creates an object by itself and insert it but we have to create an object while using push_back().

```
vector<pair<int, int>> vectorName = {{1,2}, {6,5}, {7,4},
    {3,2}};
vectorName.push_back({8, 9});
vectorName.emplace_back({3, 9});
```

5 Stack

5.1 Syntax

```
stack<dataType> stackName;

stack<int> stackName = {1, 2, 3, 4}; // Initialization is optional

stack<int> stackName; // Create an empty stack

stack<int> stackName; // Create an empty stack, no initialization
stackName.push(10);  // Push elements onto the stack
stackName.push(20);
```

5.2 Description

A stack is a container adapter that follows the **Last In, First Out (LIFO)** principle. Elements can only be added or removed from the top of the stack. Stacks are typically implemented using deques or dynamic arrays under the hood. In C++, the `stack` container adapter does not expose the underlying container, and only specific operations are available. Stacks have two primary properties:

- **Size**: The number of elements currently present in the stack.
- **Dynamic Growth**: Stacks grow dynamically as elements are pushed onto them. The underlying container manages memory allocation automatically, but the stack itself does not provide visibility into or control over the memory structure (e.g., it does not have a ”capacity” function).

5.3 Functions

1. `size()`: Returns the number of elements in the stack.
2. `push(dataToBePushed)`: Adds an element to the top of the stack.
   ```
   stackName.push(10);  // Adds 10 to the top of the stack
   ```
3. `pop()`: Removes the element from the top of the stack.
   ```
   stackName.pop();  // Removes the top element of the stack
   ```
4. `top()`: Returns a reference to the element at the top of the stack.
   ```
   cout << stackName.top();  // Outputs the element at the
       top of the stack
   ```
5. `empty()`: Returns a boolean value: `true` if the stack is empty, `false` otherwise.

```
if (stackName.empty()) {
    cout << "Stack is empty" << endl;
}
```

6. `clear()`: The `stack` container adapter does not have a `clear()` function. To clear a stack, you must pop elements one by one or recreate the stack object.

7. `swap(otherStack)`: Swaps the contents of the current stack with another stack of the same type.

```
stack<int> stack1, stack2;
stack1.push(10);
stack2.push(20);
stack1.swap(stack2);  // Swaps the contents of stack1 and
    stack2
```

5.4 Usage Example

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

int main() {
    stack<int> stackName;

    // Push elements onto the stack
    stackName.push(5);
    stackName.push(10);
    stackName.push(15);

    // Access the top element
    cout << "Top element: " << stackName.top() << endl; // Outputs:
         15

    // Pop the top element
    stackName.pop();
    cout << "Top after pop: " << stackName.top() << endl; //
        Outputs: 10

    // Check if the stack is empty
    if (stackName.empty()) {
        cout << "Stack is empty" << endl;
    } else {
        cout << "Stack is not empty" << endl;
    }

    return 0;
}
```

5.5 Important Notes

- **LIFO Principle**: Stacks follow the **Last In, First Out** principle, meaning the last element added is the first one to be removed.

- **Access**: Stacks only allow access to the top element using the `top()` function. It does not provide direct access to other elements within the stack.
- **Internal Container**: By default, stacks in C++ are implemented using deques, but this is abstracted away from the user. You cannot directly access or manipulate the underlying container.

6 Queue

6.1 Syntax

```
queue<dataType> queueName;

queue<int> queueName = {1, 2, 3, 4}; // Initialization is optional

queue<int> queueName; // Create an empty queue
```

6.2 Description

A queue is a container adapter that follows the **First In, First Out (FIFO)** principle. Elements are added at the back (or rear) and removed from the front. In a queue, the first element added is the first to be removed.
Like stacks, queues are typically implemented using deques or dynamic arrays, but the operations available are restricted to enqueueing and dequeueing elements from opposite ends of the container. Queues have two primary properties:

- **Size**: The number of elements currently present in the queue.
- **Dynamic Growth**: Queues grow dynamically as elements are pushed onto them. The underlying container manages memory allocation automatically, but the queue itself does not provide visibility into or control over the memory structure (e.g., it does not have a "capacity" function).

6.3 Functions

1. `size()`: Returns the number of elements in the queue.
2. `push(dataToBePushed)`: Adds an element to the back (rear) of the queue.
   ```
   queueName.push(10);  // Adds 10 to the back of the queue
   ```
3. `pop()`: Removes the front (first) element from the queue.
   ```
   queueName.pop();  // Removes the front element of the
       queue
   ```
4. `front()`: Returns a reference to the front (first) element of the queue.
   ```
   cout << queueName.front();  // Outputs the element at the
       front of the queue
   ```
5. `back()`: Returns a reference to the back (last) element of the queue.
   ```
   cout << queueName.back();  // Outputs the element at the
       back of the queue
   ```
6. `empty()`: Returns a boolean value: `true` if the queue is empty, `false` otherwise.

```
if (queueName.empty()) {
    cout << "Queue is empty" << endl;
}
```

7. `clear()`: The `queue` container adapter does not have a `clear()` function. To clear a queue, you must pop elements one by one or recreate the queue object.

8. `swap(otherQueue)`: Swaps the contents of the current queue with another queue of the same type.

```
queue<int> queue1, queue2;
queue1.push(10);
queue2.push(20);
queue1.swap(queue2);  // Swaps the contents of queue1 and
    queue2
```

6.4 Usage Example

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

int main() {
    queue<int> queueName;

    // Push elements onto the queue
    queueName.push(5);
    queueName.push(10);
    queueName.push(15);

    // Access the front element
    cout << "Front element: " << queueName.front() << endl; //
        Outputs: 5

    // Pop the front element
    queueName.pop();
    cout << "Front after pop: " << queueName.front() << endl; //
        Outputs: 10

    // Access the back element
    cout << "Back element: " << queueName.back() << endl; //
        Outputs: 15

    // Check if the queue is empty
    if (queueName.empty()) {
        cout << "Queue is empty" << endl;
    } else {
        cout << "Queue is not empty" << endl;
    }

    return 0;
}
```

6.5 Important Notes

- **FIFO Principle**: Queues follow the **First In, First Out** principle, meaning the first element added is the first one to be removed.
- **Access**: Queues allow access only to the front and back elements using the `front()` and `back()` functions. It does not allow direct access to other elements within the queue.
- **Internal Container**: By default, queues are implemented using deques or dynamic arrays, but this is abstracted away from the user. You cannot directly access or manipulate the underlying container.

7 Priority Queue

7.1 Syntax

```
priority_queue<dataType> priorityQueueName;

priority_queue<int> priorityQueueName = {10, 20, 30, 40}; //
    Initialization is optional

priority_queue<int> priorityQueueName; // Create an empty priority
    queue
```

7.2 Description

A priority queue is a container adapter that operates based on the **priority** of its elements. It is a specialized type of queue where elements are dequeued in order of their priority, not in the order they were enqueued. The element with the highest priority is always at the top and is removed first.
By default, in a C++ priority queue, the largest element is given the highest priority (using the ‘std::less‘ comparison). However, you can also customize the priority order by using a comparison function.
Priority queues are typically implemented using a binary heap (often a max heap by default). The underlying container is usually a vector, but this is abstracted from the user.
Priority queues have two primary properties:

- **Size**: The number of elements currently present in the priority queue.
- **Dynamic Growth**: Priority queues grow dynamically as elements are pushed onto them. The underlying container manages memory allocation automatically, but the priority queue itself does not provide visibility into or control over the memory structure (e.g., it does not have a ”capacity” function).

7.3 Functions

1. `size()`: Returns the number of elements in the priority queue.

2. `push(dataToBePushed)`: Adds an element to the priority queue.

   ```
   priorityQueueName.push(10);  // Adds 10 to the priority
       queue
   ```

3. `pop()`: Removes the highest-priority element from the priority queue (the top element).

   ```
   priorityQueueName.pop();  // Removes the highest-priority
       element from the queue
   ```

4. `top()`: Returns a reference to the highest-priority element in the priority queue.

   ```
   cout << priorityQueueName.top();  // Outputs the highest-
       priority element in the queue
   ```

5. `empty()`: Returns a boolean value: `true` if the priority queue is empty, `false` otherwise.

   ```
   if (priorityQueueName.empty()) {
       cout << "Priority queue is empty" << endl;
   }
   ```

6. `clear()`: The `priority_queue` container adapter does not have a `clear()` function. To clear a priority queue, you must pop elements one by one or recreate the priority queue object.

7. `swap(otherPriorityQueue)`: Swaps the contents of the current priority queue with another priority queue of the same type.

   ```
   priority_queue<int> priorityQueue1, priorityQueue2;
   priorityQueue1.push(10);
   priorityQueue2.push(20);
   priorityQueue1.swap(priorityQueue2);  // Swaps the
       contents of priorityQueue1 and priorityQueue2
   ```

7.4 Usage Example

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

int main() {
    priority_queue<int> priorityQueueName;

    // Push elements onto the priority queue
    priorityQueueName.push(5);
    priorityQueueName.push(10);
    priorityQueueName.push(15);

    // Access the highest-priority element
    cout << "Top element: " << priorityQueueName.top() << endl; //
        Outputs: 15

    // Pop the highest-priority element
    priorityQueueName.pop();
    cout << "Top after pop: " << priorityQueueName.top() << endl;
        // Outputs: 10

    // Check if the priority queue is empty
    if (priorityQueueName.empty()) {
        cout << "Priority queue is empty" << endl;
    } else {
        cout << "Priority queue is not empty" << endl;
```

```
    }

    return 0;
}
```

7.5 Important Notes

- **Priority Order**: In a priority queue, the highest-priority element (as defined by the comparator) is always at the top. By default, ‘std::greater‘ is used to create a max heap, meaning the largest element is given the highest priority. You can also use custom comparators for other priority orders (e.g., min-heap).
- **Access**: The priority queue allows access only to the highest-priority element using the `top()` function. It does not allow direct access to other elements within the queue.
- **Internal Container**: By default, priority queues are implemented using a heap-based container (usually a ‘std::vector‘), but the implementation details are abstracted away from the user.

7.6 Custom Comparator Example

If you want a min-heap priority queue (where the smallest element has the highest priority), you can provide a custom comparator. Here’s how to do it:

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

// Custom comparator for min-heap
struct Compare {
    bool operator()(int a, int b) {
        return a > b; // Swap comparison to make it a min-heap
    }
};

int main() {
    // Declare a priority queue with a custom comparator
    priority_queue<int, vector<int>, Compare> minHeap;

    minHeap.push(10);
    minHeap.push(20);
    minHeap.push(5);

    // Access the top element (which is the smallest in min-heap)
    cout << "Top element (min-heap): " << minHeap.top() << endl; //
         Outputs: 5

    return 0;
}
```

8 Map

8.1 Syntax

```
map<keyType, valueType> mapName;

map<int, string> mapName = {{1, "apple"}, {2, "banana"}}; //
    Initialization with pairs

map<int, string> mapName; // Create an empty map
```

8.2 Description

A map is an associative container that stores key-value pairs in a sorted order. Each element in a map is a pair consisting of a key and a value, where the key is unique. In a map, the keys are automatically sorted according to the key type's comparison function (by default, using the 'operator¡').
The main purpose of a map is to associate a unique key with a specific value and to provide fast access to the value using the key.
Maps have two primary properties:

- **Size**: The number of key-value pairs currently present in the map.
- **Sorted Order**: The elements in the map are stored in sorted order based on the key. The sorting order is determined by the key's comparison function (default is 'operator¡').

8.3 Functions

1. `size()`: Returns the number of key-value pairs in the map.
2. `insert(pair)`: Adds a key-value pair to the map. If the key already exists, the pair will not be inserted.

   ```
   mapName.insert({1, "apple"});  // Adds a key-value pair to
        the map
   ```

3. `erase(key)`: Removes the key-value pair with the specified key.

   ```
   mapName.erase(1);  // Removes the pair with key 1
   ```

4. `find(key)`: Returns an iterator to the element with the specified key. If the key is not found, it returns `mapName.end()`.

   ```
   auto it = mapName.find(1);  // Find element with key 1
   if (it != mapName.end()) {
       cout << it->second << endl;  // Access value using
           iterator
   }
   ```

5. `at(key)`: Returns a reference to the value associated with the specified key. Throws `std::out_of_range` exception if the key is not found.

```
cout << mapName.at(1);  // Outputs the value associated
    with key 1
try {
    cout << mapName.at(10);  // Throws out_of_range
        exception
} catch (const std::out_of_range& e) {
    cout << e.what();  // Catching exception
}
```

6. `operator[]`: Allows direct access to the value associated with the key. If the key does not exist, it will insert a default value for that key.

```
cout << mapName[1];  // Outputs the value associated with
    key 1
mapName[2] = "orange";  // Adds or updates the value for
    key 2
```

7. `empty()`: Returns a boolean value: `true` if the map is empty, `false` otherwise.

```
if (mapName.empty()) {
    cout << "Map is empty" << endl;
}
```

8. `clear()`: Removes all elements from the map.

```
mapName.clear();  // Clears all elements in the map
```

9. `swap(otherMap)`: Swaps the contents of the current map with another map of the same type.

```
map<int, string> map1, map2;
map1.insert({1, "apple"});
map2.insert({2, "banana"});
map1.swap(map2);  // Swaps the contents of map1 and map2
```

8.4 Usage Example

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

int main() {
    map<int, string> mapName;

    // Insert key-value pairs into the map
    mapName.insert({1, "apple"});
    mapName.insert({2, "banana"});
    mapName[3] = "cherry";  // Another way to insert key-value pair

    // Access and print value associated with a specific key
    cout << "Value associated with key 2: " << mapName[2] << endl;
         // Outputs: banana

    // Access value using the 'at()' function
    try {
        cout << "Value at key 3: " << mapName.at(3) << endl;  //
            Outputs: cherry
    } catch (const std::out_of_range& e) {
        cout << e.what();
    }

    // Find a key in the map
    auto it = mapName.find(1);
    if (it != mapName.end()) {
        cout << "Found key 1, value: " << it->second << endl;  //
            Outputs: apple
    }

    // Erase a key-value pair
    mapName.erase(2);  // Removes pair with key 2
    cout << "Map size after erase: " << mapName.size() << endl;  //
         Outputs: 2

    // Check if map is empty
    if (mapName.empty()) {
        cout << "Map is empty" << endl;
    } else {
        cout << "Map is not empty" << endl;
    }

    return 0;
}
```

9 Multimap

9.1 Syntax

```
multimap<keyType, valueType> multimapName;

multimap<int, string> multimapName = {{1, "apple"}, {1, "banana"},
    {2, "cherry"}}; // Initialization
```

9.2 Description

A ‘multimap‘ is an associative container that stores key-value pairs in a sorted order. Unlike a ‘map‘, multiple elements in a ‘multimap‘ can have the same key. The main purpose of a ‘multimap‘ is to allow duplicate keys while maintaining sorted order of the elements.
Properties of a ‘multimap‘:

- **Duplicate Keys**: Multiple elements can have the same key.
- **Sorted Order**: The elements in the ‘multimap‘ are stored in sorted order based on the key.

9.3 Functions

1. `size()`: Returns the number of key-value pairs in the multimap.
2. `insert(pair)`: Adds a key-value pair to the multimap.

   ```
   multimapName.insert({1, "apple"});  // Adds a key-value
       pair
   ```

3. `find(key)`: Returns an iterator to the first element with the specified key. If the key is not found, it returns `multimapName.end()`.

   ```
   auto it = multimapName.find(1);  // Find first element
       with key 1
   if (it != multimapName.end()) {
       cout << it->second << endl;  // Access value using
           iterator
   }
   ```

4. `erase(key)`: Removes all pairs with the specified key.

   ```
   multimapName.erase(1);  // Removes all pairs with key 1
   ```

   Using the ‘erase‘ function with an iterator will only remove the specific element pointed to by the iterator, rather than all elements with the same key. Here’s an example:

```
std::multimap<int, std::string> multiMap = {
    {1, "A"},
    {1, "B"},
    {2, "C"}
};

auto it = multiMap.find(1);  // Find the first element
    with key 1
multiMap.erase(it);  // Removes the first element with key
     1

for (const auto& pair : multiMap) {
    std::cout << pair.first << " " << pair.second << std::
        endl;
}

/*
Outputs

1 B
2 C
*/
```

5. `equal_range(key)`: Returns a pair of iterators representing the range of elements with the specified key.

```
auto range = multimapName.equal_range(1);
for (auto it = range.first; it != range.second; ++it) {
    cout << it->second << endl;
}
```

6. `clear()`: Removes all elements from the multimap.

```
multimapName.clear();  // Clears all elements
```

9.4 Usage Example

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

int main() {
    multimap<int, string> multimapName;

    // Insert key-value pairs
    multimapName.insert({1, "apple"});
    multimapName.insert({1, "banana"});
    multimapName.insert({2, "cherry"});

    // Access all values with the same key
    auto range = multimapName.equal_range(1);
    cout << "Values with key 1:" << endl;
    for (auto it = range.first; it != range.second; ++it) {
        cout << it->second << endl;
    }
    // Check size
    cout << "Multimap size: " << multimapName.size() << endl;  //
        Outputs: 3

    return 0;
}
```

10 Unordered Map

10.1 Syntax

```
unordered_map<keyType, valueType> unorderedMapName;

unordered_map<int, string> unorderedMapName = {{1, "apple"}, {2, "
    banana"}}; // Initialization
```

10.2 Description

An ‘unordered_map‘ is an associative container that stores key-value pairs in an unordered fashion using a hash table. Each key in the ‘unordered_map‘ is unique.
Properties of an ‘unordered_map‘:

- **Unordered Storage**: Elements are not stored in any particular order.
- **Unique Keys**: Each key is unique.
- **Average O(1) Access**: Provides constant time complexity for search, insert, and delete operations on average.

10.3 Functions

1. `size()`: Returns the number of key-value pairs in the unordered map.

2. `insert(pair)`: Adds a key-value pair to the unordered map. If the key already exists, the pair will not be inserted.

```
unorderedMapName.insert({1, "apple"});  // Adds a key-
    value pair
```

3. `erase(key)`: Removes the key-value pair with the specified key.

```
unorderedMapName.erase(1);  // Removes the pair with key 1
```

4. `find(key)`: Returns an iterator to the element with the specified key. If the key is not found, it returns `unorderedMapName.end()`.

```
auto it = unorderedMapName.find(1);  // Find element with
    key 1
if (it != unorderedMapName.end()) {
    cout << it->second << endl;  // Access value using
        iterator
}
```

5. `operator[]`: Allows direct access to the value associated with the key. If the key does not exist, it will insert a default value for that key.

```
unorderedMapName[1] = "apple";  // Adds or updates the
    value for key 1
```

6. `empty()`: Returns `true` if the unordered map is empty, `false` otherwise.

```
if (unorderedMapName.empty()) {
    cout << "Unordered map is empty" << endl;
}
```

7. `clear()`: Removes all elements from the unordered map.

```
unorderedMapName.clear();  // Clears all elements
```

10.4 Usage Example

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

int main() {
    unordered_map<int, string> unorderedMapName;

    // Insert key-value pairs
    unorderedMapName.insert({1, "apple"});
    unorderedMapName[2] = "banana";

    // Access value using key
    cout << "Value associated with key 1: " << unorderedMapName[1]
        << endl;

    // Check if key exists
    if (unorderedMapName.find(3) == unorderedMapName.end()) {
        cout << "Key 3 not found" << endl;
    }

    // Erase a key-value pair
    unorderedMapName.erase(2);
    cout << "Unordered map size: " << unorderedMapName.size() <<
        endl;

    return 0;
}
```

11 Set

11.1 Syntax

```
set<valueType> setName;

set<int> setName = {1, 2, 3, 4}; // Initialization

set<int> setName; // Create an empty set
```

11.2 Description

A ‘set‘ is an associative container that stores unique elements in a sorted order. Each element in a ‘set‘ is unique, and the elements are automatically arranged in ascending order by default.
Properties of a ‘set‘:

- **Unique Elements**: Each element in the set is unique. Duplicate elements are not allowed.
- **Sorted Order**: The elements in the set are stored in sorted order.
- **Efficient Operations**: Provides logarithmic time complexity for search, insert, and delete operations.

11.3 Functions

1. `size()`: Returns the number of elements in the set.

    ```
    cout << setName.size();  // Outputs the size of the set
    ```

2. `insert(value)`: Adds an element to the set. If the element already exists, it will not be added.

    ```
    setName.insert(5);  // Adds the value 5 to the set
    ```

3. `erase(value)`: Removes the element with the specified value.

    ```
    setName.erase(3);  // Removes the element 3
    ```

4. `find(value)`: Returns an iterator to the element with the specified value. If the value is not found, it returns `setName.end()`.

    ```
    auto it = setName.find(2);  // Find element 2
    if (it != setName.end()) {
        cout << *it << " found in the set";  // Access the
            value
    }
    ```

5. `lower_bound(value)`: Returns an iterator to the first element that is not less than the given value.

```
auto it = setName.lower_bound(3);  // Iterator to the
    first element >= 3
if (it != setName.end()) {
    cout << "Lower bound: " << *it;  // Outputs the value
}
```

6. `upper_bound(value)`: Returns an iterator to the first element that is greater than the given value.

```
auto it = setName.upper_bound(3);  // Iterator to the
    first element > 3
if (it != setName.end()) {
    cout << "Upper bound: " << *it;  // Outputs the value
}
```

7. `empty()`: Returns `true` if the set is empty, `false` otherwise.

```
if (setName.empty()) {
    cout << "Set is empty";
}
```

8. `clear()`: Removes all elements from the set.

```
setName.clear();  // Clears the set
```

9. `swap(otherSet)`: Swaps the contents of the current set with another set of the same type.

```
set<int> set1 = {1, 2, 3};
set<int> set2 = {4, 5};
set1.swap(set2);  // Swaps the contents of set1 and set2
```

11.4 Usage Example

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

int main() {
    set<int> setName = {1, 2, 3, 4, 5};

    // Insert an element
    setName.insert(6);

    // Access elements
    cout << "Elements in the set: ";
    for (int elem : setName) {
        cout << elem << " ";
    }
    cout << endl;

    // Find an element
    if (setName.find(3) != setName.end()) {
        cout << "Element 3 found in the set." << endl;
    }

    // Lower bound
    auto lower = setName.lower_bound(3);
    if (lower != setName.end()) {
        cout << "Lower bound of 3: " << *lower << endl;
    }

    // Upper bound
    auto upper = setName.upper_bound(3);
    if (upper != setName.end()) {
        cout << "Upper bound of 3: " << *upper << endl;
    }

    // Erase an element
    setName.erase(2);
    cout << "Set size after erasing 2: " << setName.size() << endl;

    return 0;
}
```

12 Multiset

12.1 Syntax

```
multiset<valueType> multisetName;

multiset<int> multisetName = {1, 2, 2, 3}; // Initialization

multiset<int> multisetName; // Create an empty multiset
```

12.2 Description

A ‘multiset‘ is an associative container that stores elements in sorted order and allows duplicate values. The ordering is maintained by a comparison function, which is ‘operator¡‘ by default.
Properties of a ‘multiset‘:

- **Duplicates Allowed**: Multiple occurrences of the same value are permitted.
- **Sorted Order**: Elements are stored in ascending order (or any custom sorting criteria defined by the comparison function).
- **Efficient Operations**: Provides logarithmic time complexity for search, insert, and delete operations.

12.3 Functions

1. `size()`: Returns the number of elements in the multiset.
2. `insert(value)`: Adds an element to the multiset. Duplicate elements are allowed.

   ```
   multisetName.insert(2);  // Adds 2 to the multiset
   ```

3. `erase(value)`: Removes all occurrences of the specified value.

   ```
   multisetName.erase(2);  // Removes all 2s from the
       multiset
   ```

4. `find(value)`: Returns an iterator to the first occurrence of the specified value.

   ```
   auto it = multisetName.find(2);  // Finds first occurrence
        of 2
   if (it != multisetName.end()) {
       cout << *it << endl;  // Access the value
   }
   ```

5. `count(value)`: Returns the number of occurrences of a specified value.

```cpp
cout << multisetName.count(2);  // Outputs the count of 2
    in the multiset
```

6. `lower_bound(value)`: Returns an iterator to the first element that is not less than the given value.

7. `upper_bound(value)`: Returns an iterator to the first element that is greater than the given value.

```cpp
auto lower = multisetName.lower_bound(2);  // Iterator to
    first >= 2
auto upper = multisetName.upper_bound(2);  // Iterator to
    first > 2
for (auto it = lower; it != upper; ++it) {
    cout << *it << " ";  // Access values in the range
}
```

8. `clear()`: Removes all elements from the multiset.

12.4 Usage Example

```cpp
#include <iostream>
#include <set>
using namespace std;

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

    // Insert elements
    multisetName.insert(3);

    // Count occurrences of an element
    cout << "Count of 2: " << multisetName.count(2) << endl;

    // Access range with lower and upper bound
    auto lower = multisetName.lower_bound(2);
    auto upper = multisetName.upper_bound(3);
    cout << "Elements in range [2, 3): ";
    for (auto it = lower; it != upper; ++it) {
        cout << *it << " ";
    }
    cout << endl;

    // Erase all occurrences of 2
    multisetName.erase(2);

    return 0;
}
```

13 Unordered Set

13.1 Syntax

```
unordered_set<valueType> unorderedSetName;

unordered_set<int> unorderedSetName = {1, 2, 3, 4}; //
    Initialization
```

13.2 Description

An ‘unordered_set‘ is an associative container that stores unique elements in an unordered fashion using a hash table. It provides average constant time complexity for search, insert, and delete operations.
Properties of an ‘unordered_set‘:

- **Unique Elements**: Each element in the ‘unordered_set‘ is unique. Duplicate values are not allowed.
- **Unordered Storage**: Elements are stored in no particular order.
- **Efficient Operations**: Provides average O(1) time complexity for search, insert, and delete operations.

13.3 Functions

1. `size()`: Returns the number of elements in the unordered set.
2. `insert(value)`: Adds an element to the unordered set. Duplicate values are ignored.

   ```
   unorderedSetName.insert(5);  // Adds 5 to the set
   ```

3. `erase(value)`: Removes the specified value.

   ```
   unorderedSetName.erase(2);  // Removes 2 from the set
   ```

4. `find(value)`: Returns an iterator to the specified value. If the value is not found, it returns `unorderedSetName.end()`.

   ```
   auto it = unorderedSetName.find(3);
   if (it != unorderedSetName.end()) {
       cout << *it << " found in the unordered set";
   }
   ```

5. `count(value)`: Returns 1 if the value exists, 0 otherwise.

   ```
   cout << unorderedSetName.count(4);  // Outputs 1 if 4
       exists
   ```

6. `clear()`: Removes all elements from the unordered set.
7. `empty()`: Returns `true` if the set is empty.

13.4 Usage Example

```cpp
#include <iostream>
#include <unordered_set>
using namespace std;

int main() {
    unordered_set<int> unorderedSetName = {1, 2, 3, 4};

    // Insert an element
    unorderedSetName.insert(5);

    // Check existence of an element
    if (unorderedSetName.count(3)) {
        cout << "Element 3 exists in the set" << endl;
    }

    // Erase an element
    unorderedSetName.erase(2);

    // Iterate over the unordered set
    cout << "Elements in the unordered set: ";
    for (int elem : unorderedSetName) {
        cout << elem << " ";
    }
    cout << endl;

    return 0;
}
```

14 Sorting and Utility Functions

14.1 Sort

14.2 Syntax

```
#include <algorithm>
sort(beginIterator, endIterator);
sort(beginIterator, endIterator, comparator);
```

14.3 Description

The ‘sort‘ function is part of the C++ Standard Library’s ‘¡algorithm¿‘ header. It sorts the elements in the range ‘[beginIterator, endIterator)‘ in ascending order by default. You can also provide a custom comparator to define a custom sorting order.
Properties:

- Sorting is done in-place using the IntroSort algorithm, which is a combination of QuickSort, HeapSort, and InsertionSort.
- Time complexity: $O(n \log n)$.

14.4 Custom Comparator

A custom comparator is a callable (function, lambda, or functor) that defines the sorting criteria. The comparator must return ‘true‘ if the first argument should come before the second, and ‘false‘ otherwise.

```
// Example: Sort in descending order
bool comparator(int a, int b) {
    return a > b; // Custom comparator for descending order
}

sort(arr.begin(), arr.end(), comparator);

// Lambda Function Example
sort(arr.begin(), arr.end(), [](int a, int b) {
    return a > b; // Descending order
});
```

14.5 Functions for Sorting and Search

1. `sort(beginIterator, endIterator)`: Sorts the range in ascending order by default.
2. sort(beginIterator, endIterator, comparator): Sorts the range based on a custom comparator.
3. binary_search(beginIterator, endIterator, value): Checks if a value exists in a sorted range.

```
sort(arr.begin(), arr.end());
if (binary_search(arr.begin(), arr.end(), 5)) {
    cout << "5 exists in the array";
}
```

4. `max_element(beginIterator, endIterator)`: Returns an iterator to the largest element in the range.

   ```
   auto maxIt = max_element(arr.begin(), arr.end());
   cout << "Max element: " << *maxIt;
   ```

5. `min_element(beginIterator, endIterator)`: Returns an iterator to the smallest element in the range.

   ```
   auto minIt = min_element(arr.begin(), arr.end());
   cout << "Min element: " << *minIt;
   ```

14.6 Usage Example

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Custom comparator function
bool customComparator(int a, int b) {
    return a > b; // Sort in descending order
}

int main() {
    vector<int> arr = {4, 2, 8, 1, 3};

    // Default sort (ascending order)
    sort(arr.begin(), arr.end());
    cout << "Sorted in ascending order: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    // Custom sort (descending order)
    sort(arr.begin(), arr.end(), customComparator);
    cout << "Sorted in descending order: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    // Lambda for sorting by even first, then odd
    sort(arr.begin(), arr.end(), [](int a, int b) {
        if (a % 2 == b % 2) return a < b; // Ascending within
            groups
        return (a % 2 == 0); // Even numbers come first
    });
    cout << "Sorted with custom lambda: ";
```

```
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    // Find max and min elements
    auto maxIt = max_element(arr.begin(), arr.end());
    auto minIt = min_element(arr.begin(), arr.end());
    cout << "Max element: " << *maxIt << endl;
    cout << "Min element: " << *minIt << endl;

    // Binary search
    sort(arr.begin(), arr.end());
    if (binary_search(arr.begin(), arr.end(), 3)) {
        cout << "3 is found in the array" << endl;
    } else {
        cout << "3 is not found in the array" << endl;
    }

    return 0;
}
```