Vector:

Explanation: Dynamic array that can resize itself.
Unique Property: Efficient random access with indexing.
Why Choose: Best for frequent insertion/deletion at the end.

Common Methods:

push_back(): Adds an element to the end.
pop_back(): Removes the last element.
at(): Accesses an element at a specified position.
size(): Returns the number of elements.
empty(): Checks if the vector is empty.
front(): Accesses the first element.
back(): Accesses the last element.
clear(): Clears all elements.

Example:

```
#include <vector>
#include <iostream>
int main() {
std::vector<int> numbers = {1, 2, 3};
numbers.push_back(4);
numbers.pop_back();
std::cout << numbers.at(1);  // Outputs the element at inde
std::cout << numbers.size(); // Outputs the size of the vector
return 0;
}
```

Iteration:

```
for (const auto& num : numbers) {
std::cout << num << " ";
}
```

List:

Explanation: Doubly linked list.
Unique Property: Efficient insertion/deletion anywhere in the list.
Why Choose: Best for frequent mid-list operations.

Common Methods:

push_front(): Adds an element to the front.
push_back(): Adds an element to the end.
pop_front(): Removes the first element.
pop_back(): Removes the last element.
size(): Returns the number of elements.
empty(): Checks if the list is empty.
front(): Accesses the first element.
back(): Accesses the last element.

Example:

```
#include <list>
#include <iostream>

int main() {
std::list<std::string> words = {"apple", "banana", "cherry"};
words.push_front("orange");
words.pop_back();
return 0;
}
```

Iteration:

```
for (const auto& word : words) {
std::cout << word << " ";
}
```

Set:

Explanation: Collection of unique, sorted elements.
Unique Property: Automatically sorted in ascending order.
Why Choose: Best for maintaining a sorted collection of unique elements.

Common Methods:
insert(): Adds an element.
erase(): Removes an element.
find(): Finds an element.
size(): Returns the number of elements.
empty(): Checks if the set is empty.
count(): Counts occurrences of an element.
clear(): Clears all elements.
lower_bound(): Returns an iterator to the first element not less than the given value.
upper_bound(): Returns an iterator to the first element greater than a certain value.

Example:

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

int main() {
std::set<int> numbers = {3, 1, 4};
numbers.insert(2);
numbers.erase(4);
return 0;
}
```

Iteration and Printing:

```
for (const auto& num : numbers) {
std::cout << num << " ";
}
```

Map:

Explanation: Key-value pairs with unique keys.
Unique Property: Stores elements sorted by keys.
Why Choose: Ideal for associative arrays, dictionary-like data structures.

Common Methods:

insert(): Adds a key-value pair.
erase(): Removes a key-value pair.
find(): Finds a key.
size(): Returns the number of elements.
empty(): Checks if the map is empty.
clear(): Clears all elements.
count(): Counts occurrences of a key.
operator[]: Accesses or inserts elements using a key.

Example:

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

int main() {
std::map<std::string, int> ages = {{"Alice", 25}, {"Bob", 30}};
ages.insert({"Charlie", 22});
ages.erase("Bob");
return 0;
}
```

Iteration and Printing:

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

Deque:

Explanation: Double-ended queue.
Unique Property: Efficient insertion/deletion at both ends.
Why Choose: Best for frequent insertion/deletion at front/back.

Common Methods:

push_front(): Adds an element to the front.
push_back(): Adds an element to the end.
pop_front(): Removes the first element.
pop_back(): Removes the last element.
size(): Returns the number of elements.
empty(): Checks if the deque is empty.
front(): Accesses the first element.
back(): Accesses the last element.

Example:

```
#include <deque>
#include <iostream>

int main() {
std::deque<double> values = {3.14, 2.71, 1.618};
values.push_front(2.0);
values.pop_back();
return 0;
}
```

Iteration:

```
for (const auto& val : values) {
std::cout << val << " ";
}
```

Stack:

Explanation: LIFO (Last-In-First-Out) data structure.
Unique Property: Access limited to the top element.
Why Choose: Best for managing function calls or undo functionality.

Common Methods:
push(): Adds an element to the top.
pop()*:* Removes the top element.
top(): Accesses the top element.

Example:

```
#include <stack>
#include <iostream>
int main() {
std::stack<int> s;
s.push(5);
s.pop();
return 0;}
```

Iteration and Printing:

AS STACKS DON'T SUPPORT DIRECT ITERATION, TO PRINT ELEMENTS, YOU NEED TO POP AND PRINT UNTIL EMPTY.

Queue:

Explanation: FIFO (First-In-First-Out) data structure.
Unique Property: Elements are accessed in the order they were added.
Why Choose: Best for modeling real-world scenarios like task scheduling.

Common Methods:

push(): Adds an element to the end.
pop(): Removes the first element.
front(): Accesses the first element.

Example:

```
#include <queue>
#include <iostream>

int main() {
std::queue<int> q;
q.push(3);
q.pop();
return 0;
}
```

Iteration and Printing:

Similar to stacks, queues require popping elements for printing.

Priority Queue:

Explanation: Queue where elements have a priority.
Unique Property: Element with the highest priority always comes first.
Why Choose: Best for tasks that need to be processed in order of priority.

Common Methods:

push(): Adds an element according to priority.
pop(): Removes the top priority element.
top(): Accesses the top priority element.

Example:

```
#include <queue>
#include <iostream>
int main() {
std::priority_queue<int> pq;
pq.push(10);
pq.pop();
return 0;
}
```

Iteration and Printing:

Similar to stacks and queues, priority queues require popping elements for printing.

Multiset:

Explanation: Set that allows duplicate elements.
Unique Property: Allows multiple occurrences of the same value.
Why Choose: Best for scenarios where duplicates need to be preserved.
Common Methods:
insert(): Adds an element.
erase(): Removes an element.
count(): Counts occurrences of an element.

Example:

```
#include <set>
#include <iostream>
int main() {
std::multiset<int> numbers = {3, 1, 4, 1};
numbers.insert(2);
numbers.erase(1);
return 0; }
```

Iteration and Printing:

```
for (const auto& num : numbers) {
std::cout << num << " "; }
```

Multimap:

Explanation: Map that allows duplicate keys.

Unique Property: Allows multiple elements with the same key.

Why Choose: Ideal for scenarios where multiple values can be associated with the same key.

Common Methods:

insert(): Adds a key-value pair.

erase(): Removes a key-value pair.

count(): Counts occurrences of a key.

Example:

```
#include <map>
#include <iostream>
int main() {
std::multimap<std::string, int> ages = {{"Alice", 25}, {"Bob", 30}, {"Alice", 22}};
ages.insert({"Charlie", 27});
ages.erase("Alice");
return 0;
}
```

Iteration and Printing:

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

Unordered Set:

Explanation: Set with unordered storage for faster access.

Unique Property: Faster search, insertion, and deletion in comparison to ordered sets.

Why Choose: Great for scenarios where ordering is not essential but speed is crucial.

Common Methods:

insert(): Adds an element.

erase(): Removes an element.

find(): Finds an element.

size(): Returns the number of elements.

empty(): Checks if the set is empty.

clear(): Clears all elements.

Example:

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

int main() {
std::unordered_set<int> numbers = {3, 1, 4};
numbers.insert(2);
numbers.erase(4);
return 0;
}
```

Iteration and Printing:

```
for (const auto& num : numbers) {
std::cout << num << " ";
}
```

Unordered Map:

Explanation: Map with unordered storage for faster access.

Unique Property: Faster search, insertion, and deletion in comparison to ordered maps.

Why Choose: Suitable for applications where the order of elements is not crucial.

Common Methods:

insert(): Adds a key-value pair.

erase(): Removes a key-value pair.

find(): Finds a key.

size(): Returns the number of elements.

empty(): Checks if the map is empty.

clear(): Clears all elements.

Example:

```
#include <unordered_map>
#include <iostream>
int main() {
std::unordered_map<std::string, int> ages = {{"Alice", 25}, {"Bob", 30}};
ages.insert({"Charlie", 22});
ages.erase("Bob");
return 0;
}
```

Iteration and Printing:

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