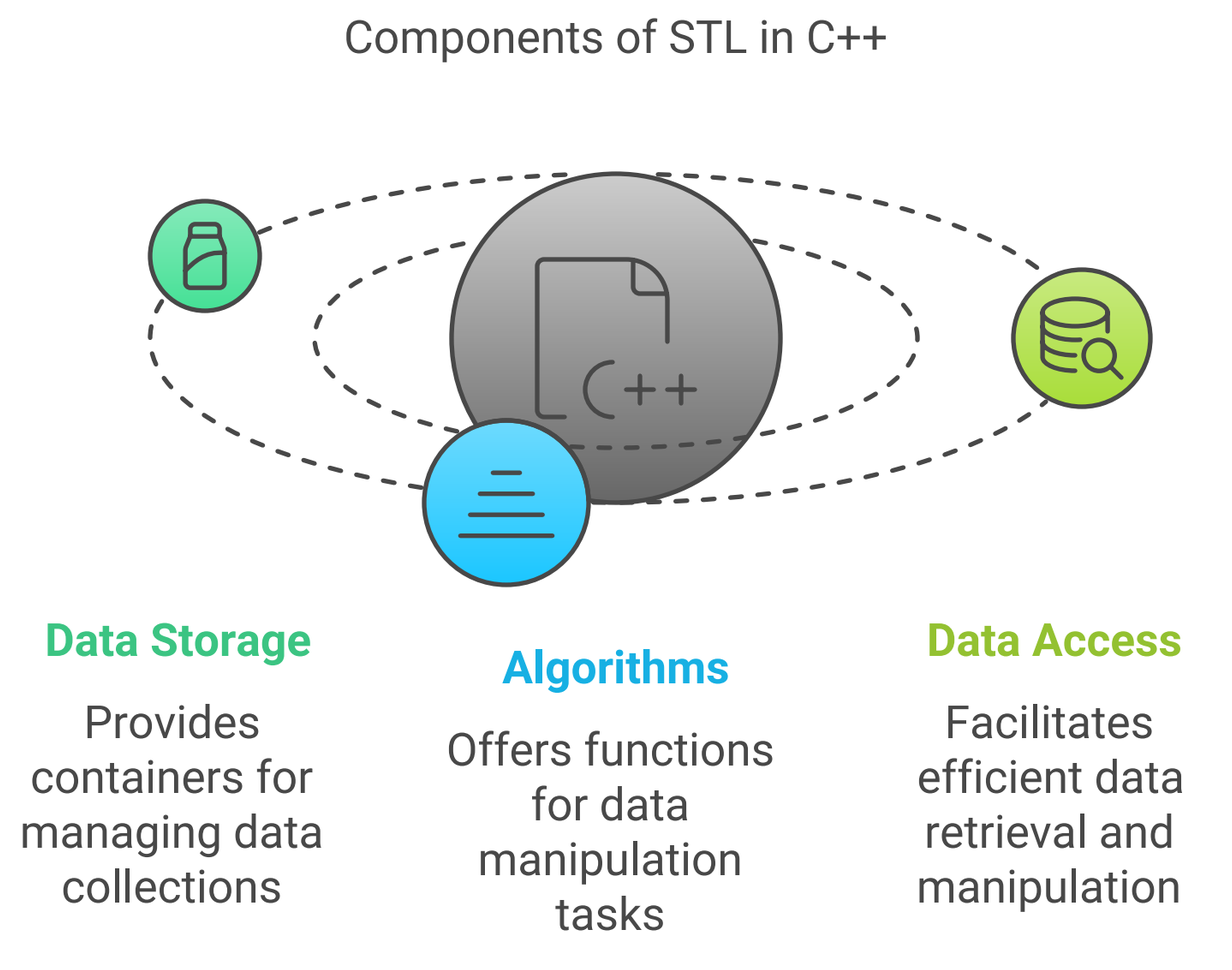
Lecture 7: Standard Template Library (STL) in C++

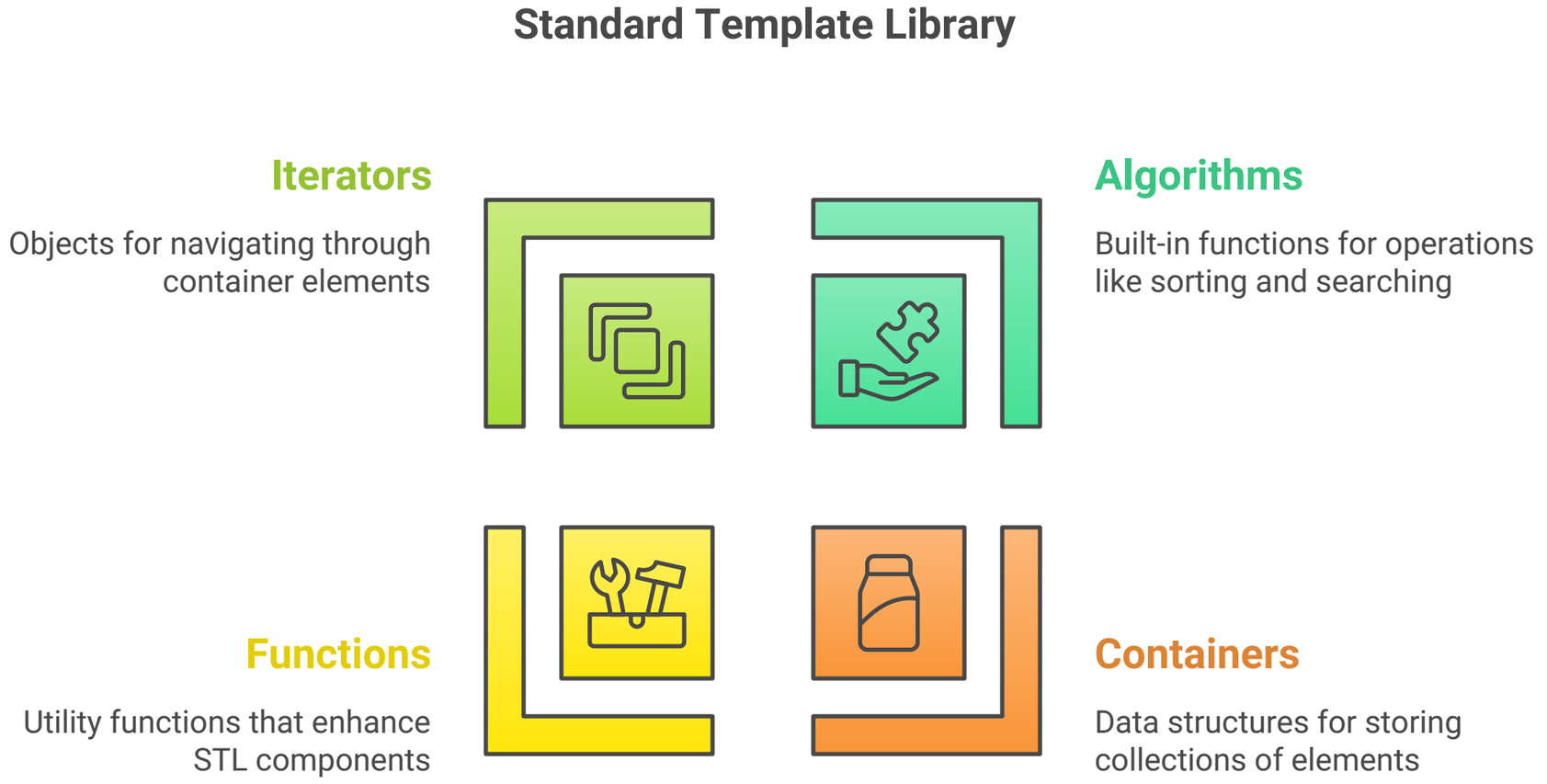
Overview

The **Standard Template Library (STL)** in C++ is a collection of template classes and functions, making it easier to manage data collections and perform common tasks like searching, sorting, and manipulating data structures. STL brings consistency, efficiency, and convenience to C++ development, offering components for data storage, algorithms for data manipulation, and tools for accessing data.



1. Key Components of STL

1. **Algorithms**: Built-in functions that perform operations like sorting, searching, counting, etc.
2. **Containers**: Data structures that store collections of elements.
3. **Functions**: Utility functions that work alongside STL components.
4. **Iterators**: Objects that allow navigation through container elements.

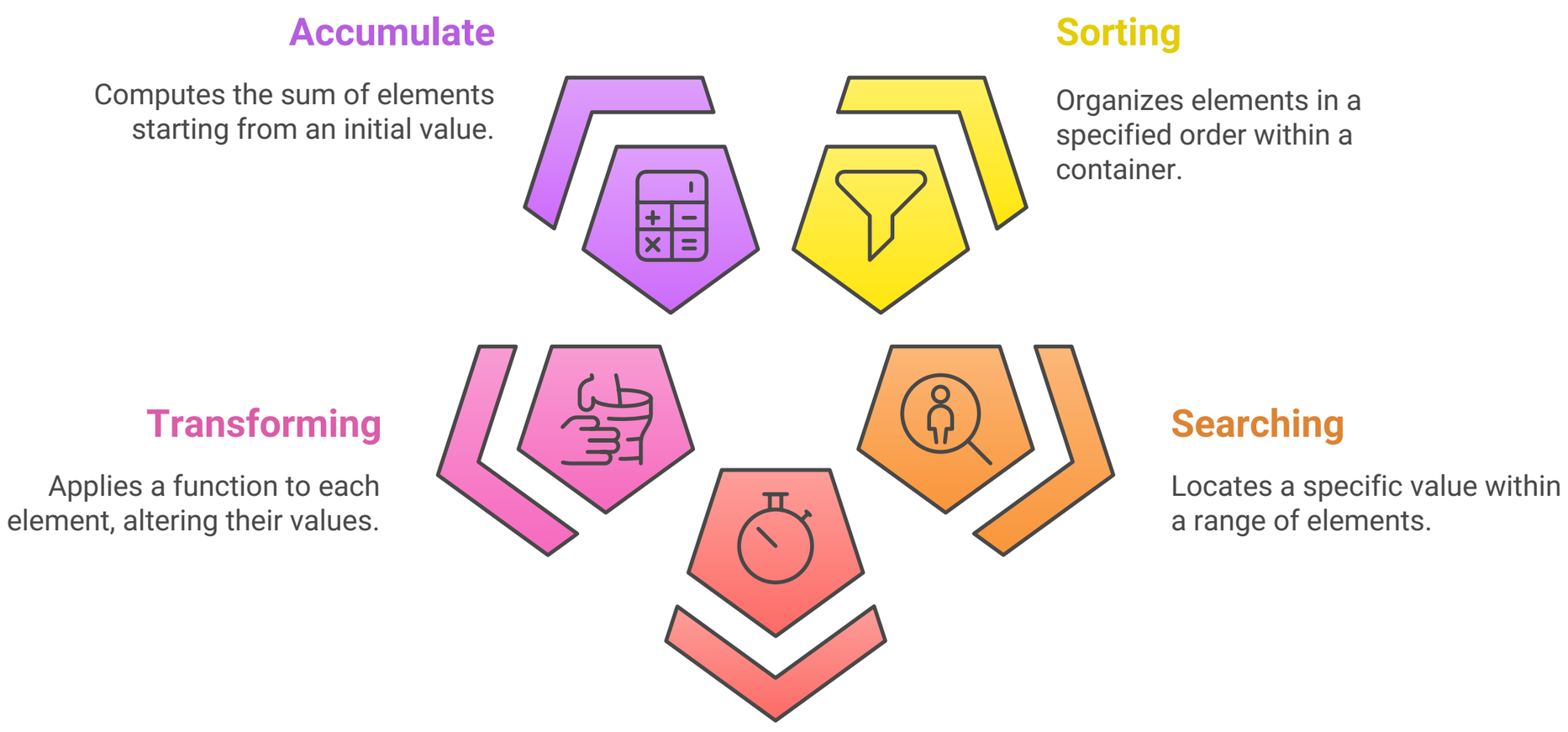


These components work together, with algorithms typically operating on data within containers using iterators.

2. Algorithms

STL algorithms are a set of functions that operate on ranges of elements. Algorithms are general-purpose and can work on any STL container. Common algorithms include:

1. **Sorting** - **sort(begin, end)** to sort a container.
2. **Searching** - **find(begin, end, value)** to locate a value.
3. **Counting** - **count(begin, end, value)** to count occurrences.
4. **Transforming** - **transform(begin, end, result, function)** to apply a function to each element.
5. **Accumulate** - **accumulate(begin, end, init)** to compute the sum of elements.



Example of Sorting and Counting:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric> // For accumulate
using namespace std;

int main() {
    vector<int> numbers = {5, 2, 8, 6, 3, 4};

    // Sorting
    sort(numbers.begin(), numbers.end());

    // Counting occurrences of '6'
    int countSix = count(numbers.begin(), numbers.end(), 6);

    // Summing all elements
    int sum = accumulate(numbers.begin(), numbers.end(), 0);

    cout << "Sorted numbers: ";
```

```
    for (int num : numbers) cout << num << " ";
    cout << "\nCount of 6: " << countSix << endl;
    cout << "Sum of all numbers: " << sum << endl;

    return 0;
}
```

Example of find

The **find** algorithm searches for the first occurrence of a specified value within a container. If found, it returns an iterator to the element; otherwise, it returns **end()**.

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

int main() {
    vector<int> numbers = {10, 20, 30, 40, 50};

    // Finding the number 30
    auto it = find(numbers.begin(), numbers.end(), 30);

    if (it != numbers.end()) {
        cout << "Found 30 at position: " << distance(numbers.begin(), it) << endl;
    } else {
        cout << "30 not found in the vector." << endl;
    }

    return 0;
}
```

Example of transform

The **transform** algorithm applies a specified function to each element in a container, storing the result in another container (or modifying the original one if specified). Here, we'll use **transform** to square each element in a vector.

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

// Function to square each element
int square(int x) {
    return x * x;
}

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    vector<int> squares(numbers.size());

    // Transforming each element to its square
    transform(numbers.begin(), numbers.end(), squares.begin(), square);

    cout << "Original numbers: ";
    for (int num : numbers) cout << num << " ";
    cout << "\nSquared numbers: ";
    for (int sq : squares) cout << sq << " ";
    cout << endl;

    return 0;
```

```
}
```

Example of accumulate

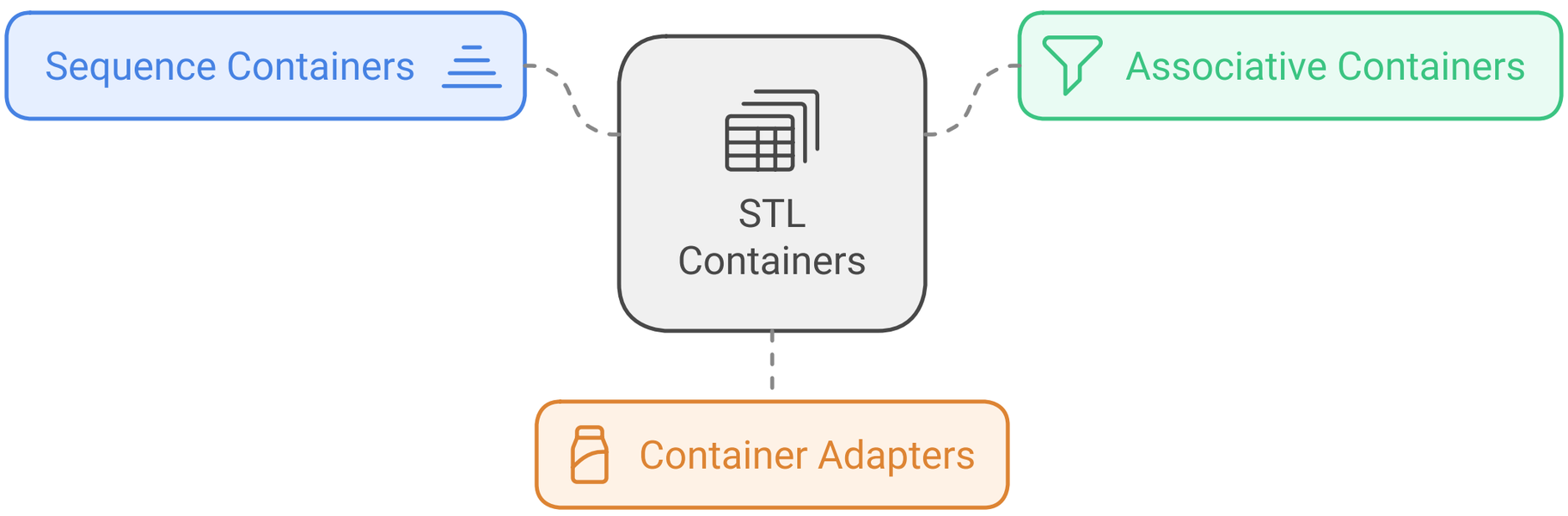
The **accumulate** algorithm is used to compute the sum (or other operations) of all elements in a container. Here, we'll calculate the sum of elements in a vector.

```
#include <iostream>
#include <vector>
#include <numeric> // For accumulate
using namespace std;

int main() {
    vector<int> numbers = {10, 20, 30, 40, 50};

    // Calculating the sum of all elements
    int sum = accumulate(numbers.begin(), numbers.end(), 0);

    cout << "Sum of all numbers: " << sum << endl;

    return 0;
}
```

3. Containers

Containers are data structures for organizing collections of elements. STL containers are divided into:

- **Sequence Containers**: Maintain ordering and include containers like **vector**, **deque**, and **list**.
- **Associative Containers**: Automatically sort elements and include containers like **set** and **map**.
- **Container Adapters**: Modify underlying containers to provide stack, queue, and priority queue functionality.

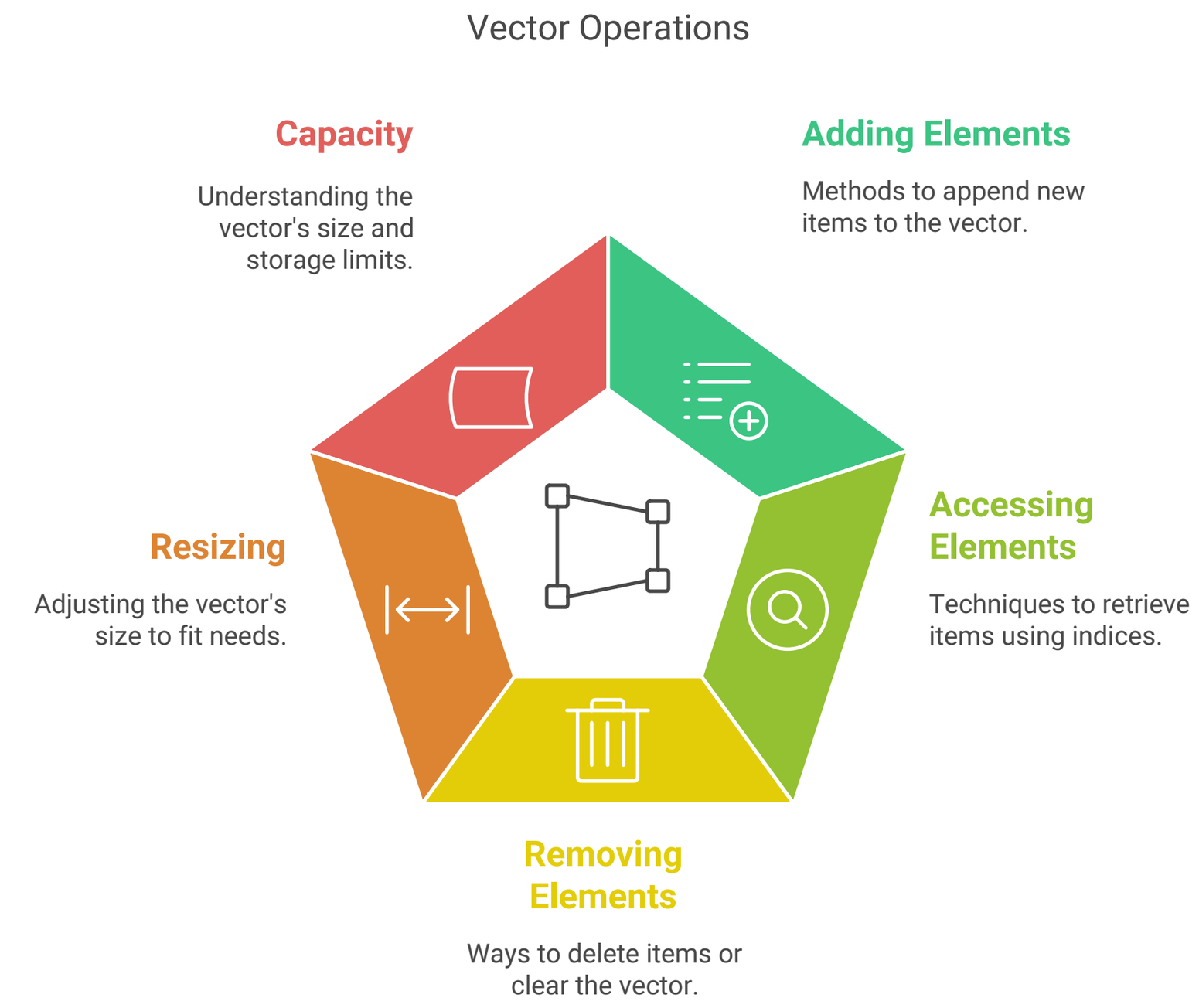


4. Vector

A **vector** is a dynamic array, meaning its size can grow or shrink as needed. It allows efficient random access to elements, making it a preferred choice for many applications.

Common Vector Operations

1. **Adding elements**: **push_back(value)**, **emplace_back(value)**
2. **Accessing elements**: **[]** operator, **at(index)**
3. **Removing elements**: **pop_back()**, **erase(position)**, **clear()**
4. **Resizing**: **resize(new_size)**
5. **Capacity**: **size()**, **capacity()**, **empty()**



Example of Vector Manipulation:

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

int main() {
    // Initializing a vector
    vector<int> numbers = {10, 20, 30};

    // 1. Adding elements
    numbers.push_back(40);         // Adds 40 to the end
    numbers.emplace_back(50);      // Adds 50 to the end, more efficient than push_back for
objects

    // 2. Accessing elements
    cout << "First element: " << numbers[0] << endl;
    cout << "Second element (using at): " << numbers.at(1) << endl;

    // 3. Inserting elements at a specific position
    auto it = numbers.begin() + 1; // Iterator pointing to the second position
    numbers.insert(it, 15);        // Inserts 15 at the second position

    // 4. Removing elements
    numbers.pop_back();            // Removes the last element (50)
    numbers.erase(numbers.begin()); // Removes the first element (10)

    // 5. Resizing the vector
    numbers.resize(3);             // Changes the vector size to 3, truncating excess elements if
necessary
    numbers.resize(5, 99);         // Increases size to 5, new elements initialized to 99

    // 6. Sorting elements in descending order
```

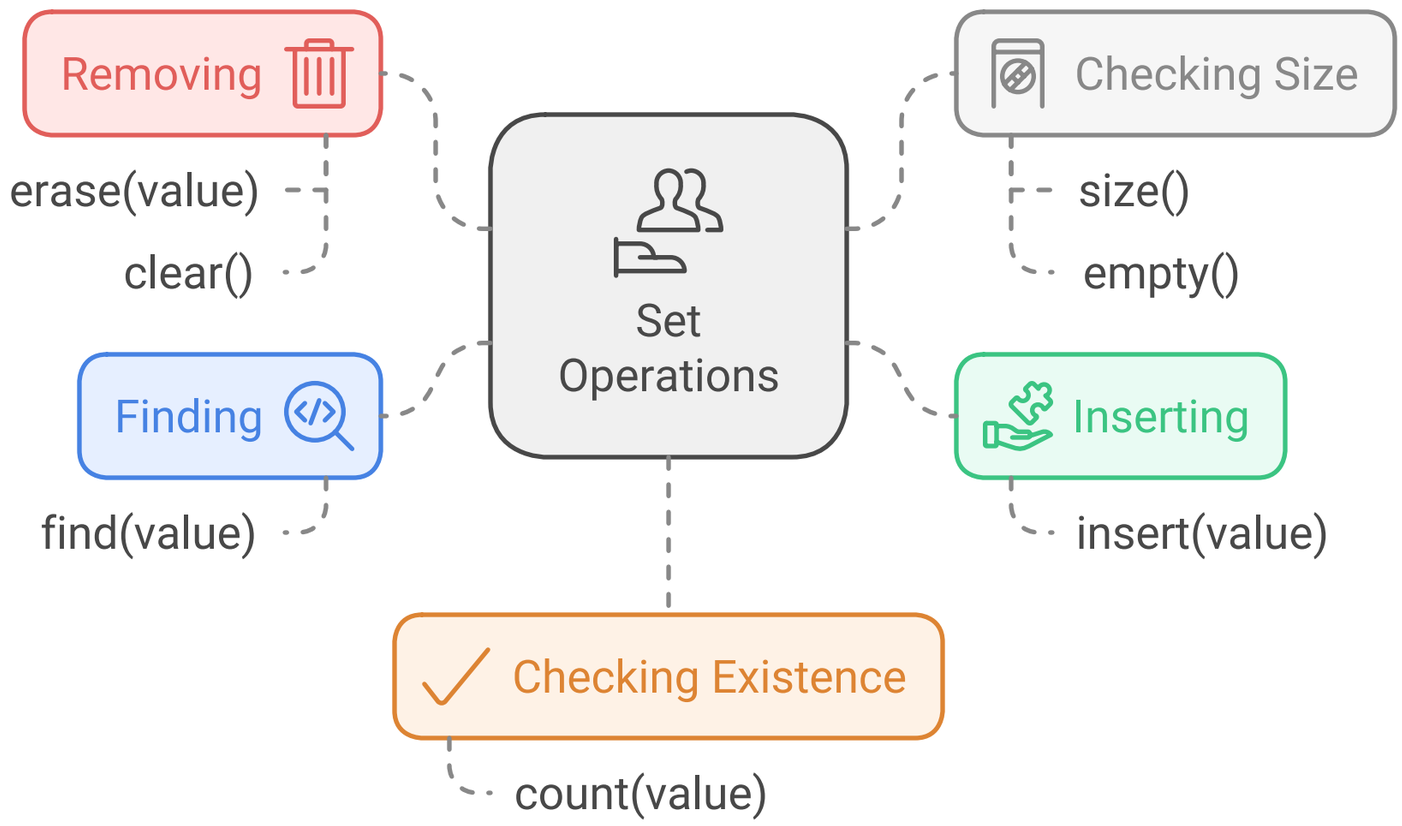
```
sort(numbers.begin(), numbers.end(), greater<int>());

// 7. Checking size and capacity
cout << "Size of vector: " << numbers.size() << endl;
cout << "Capacity of vector: " << numbers.capacity() << endl;

// 8. Clearing all elements
numbers.clear();
cout << "Size after clear: " << numbers.size() << " (should be 0)" << endl;

// 9. Checking if the vector is empty
if (numbers.empty()) {
    cout << "Vector is empty!" << endl;
}

// 10. Using iterators to traverse
numbers = {5, 15, 25, 35, 45}; // Re-initializing with values
cout << "Using iterators to traverse: ";
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    cout << *it << " ";
}
cout << endl;

return 0;
}
```

5. Set

A **set** is an associative container that contains unique elements and automatically sorts them. Sets do not allow duplicate values, making them ideal for storing distinct items. Elements are ordered in ascending order by default.

Common Set Operations

1. **Inserting elements**: **insert(value)**
2. **Removing elements**: **erase(value)**, **clear()**
3. **Finding elements**: **find(value)** returns an iterator
4. **Checking size**: **size()**, **empty()**
5. **Checking if an element exists**: **count(value)**



Example of Set Manipulation:

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

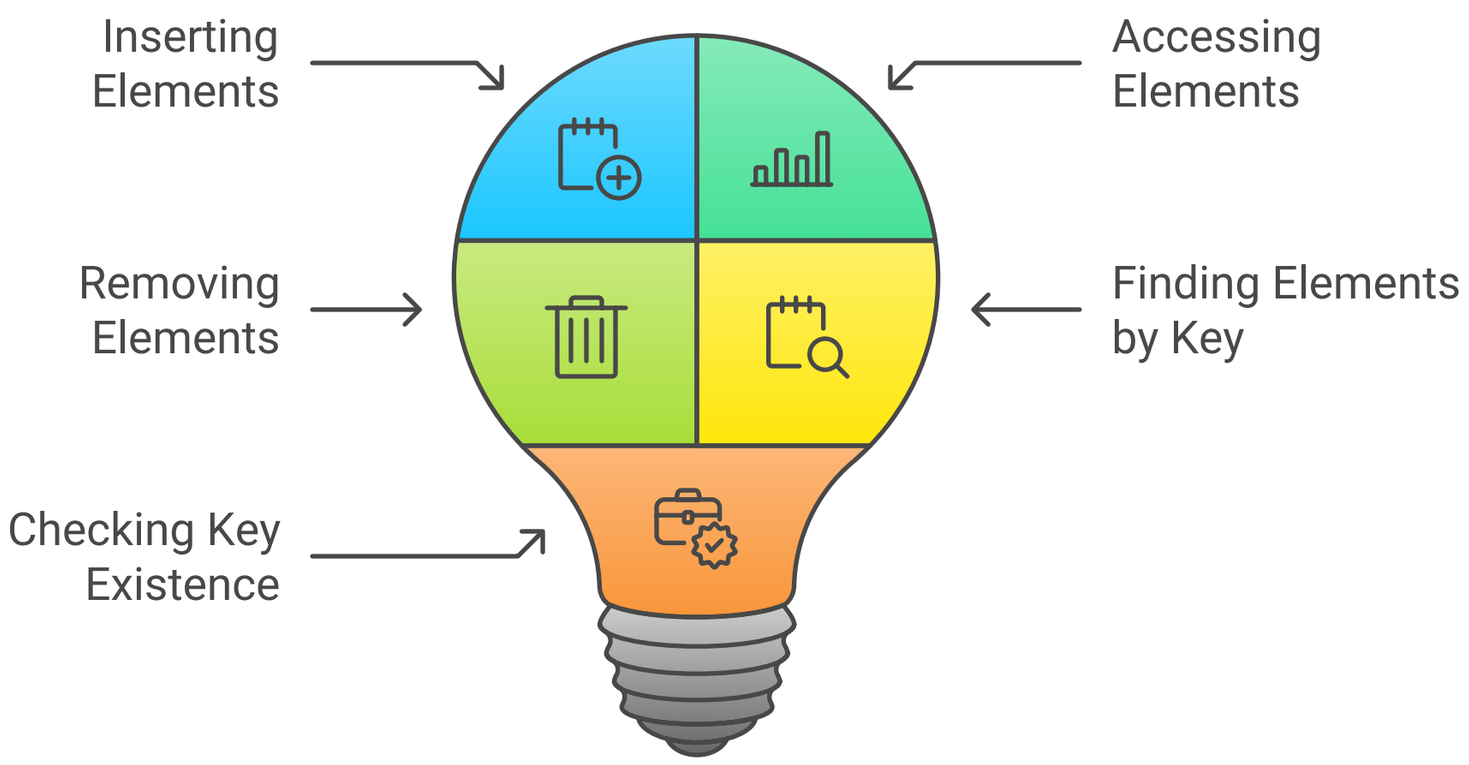
```
int main() {
    // Initializing a set
    set<int> numbers = {20, 10, 30};

    // 1. Inserting elements
    numbers.insert(25);    // Adds 25 to the set
    numbers.insert(10);    // Duplicate; ignored by the set

    // 2. Accessing elements (using iterator)
    cout << "Elements in set: ";
    for (int num : numbers) {
        cout << num << " ";
    }
    cout << endl;

    // 3. Checking if an element exists
    int value = 30;
    if (numbers.find(value) != numbers.end()) {
        cout << value << " is in the set." << endl;
    } else {
        cout << value << " is not in the set." << endl;
    }

    // 4. Erasing elements
    numbers.erase(20);    // Removes 20 from the set
    numbers.erase(numbers.begin()); // Removes the first element

    cout << "Set after erasing elements: ";
    for (int num : numbers) {
        cout << num << " ";
    }
    cout << endl;

    // 5. Clearing all elements
    numbers.clear();
    cout << "Size after clear: " << numbers.size() << " (should be 0)" << endl;

    // 6. Using count to check if an element exists
    numbers = {5, 10, 15}; // Reinitializing set
    if (numbers.count(10) > 0) {
        cout << "10 exists in the set." << endl;
    } else {
        cout << "10 does not exist in the set." << endl;
    }

    // 7. Checking size and emptiness
    cout << "Size of set: " << numbers.size() << endl;
    if (numbers.empty()) {
        cout << "Set is empty!" << endl;
    } else {
        cout << "Set is not empty!" << endl;
    }

    return 0;
}
```

6. Map

A **map** is an associative container that stores elements in key-value pairs. Each key is unique, allowing for efficient lookup by key. Maps are often used when storing related information, such as names and scores.

Common Map Operations

1. **Inserting elements**: **insert(pair<key, value>)**, **operator[]**
2. **Accessing elements**: **operator[]**
3. **Removing elements**: **erase(key)**, **clear()**
4. **Finding elements by key**: **find(key)**
5. **Checking if a key exists**: **count(key)**



Example of Map Manipulation:

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

int main() {
    // Initializing a map
    map<string, int> scores;

    // 1. Inserting elements
    scores["Alice"] = 90;                // Insert using [] operator
    scores.insert({"Bob", 85});          // Insert using insert() with a pair

    // 2. Accessing elements
    cout << "Alice's score: " << scores["Alice"] << endl;
    cout << "Bob's score: " << scores.at("Bob") << endl;

    // 3. Updating values
    scores["Alice"] = 95;                // Updates Alice's score to 95

    // 4. Checking if a key exists
    string key = "Charlie";
    if (scores.find(key) != scores.end()) {
        cout << key << " is in the map with score: " << scores[key] << endl;
    } else {
        cout << key << " is not in the map." << endl;
    }

    // 5. Erasing elements
```

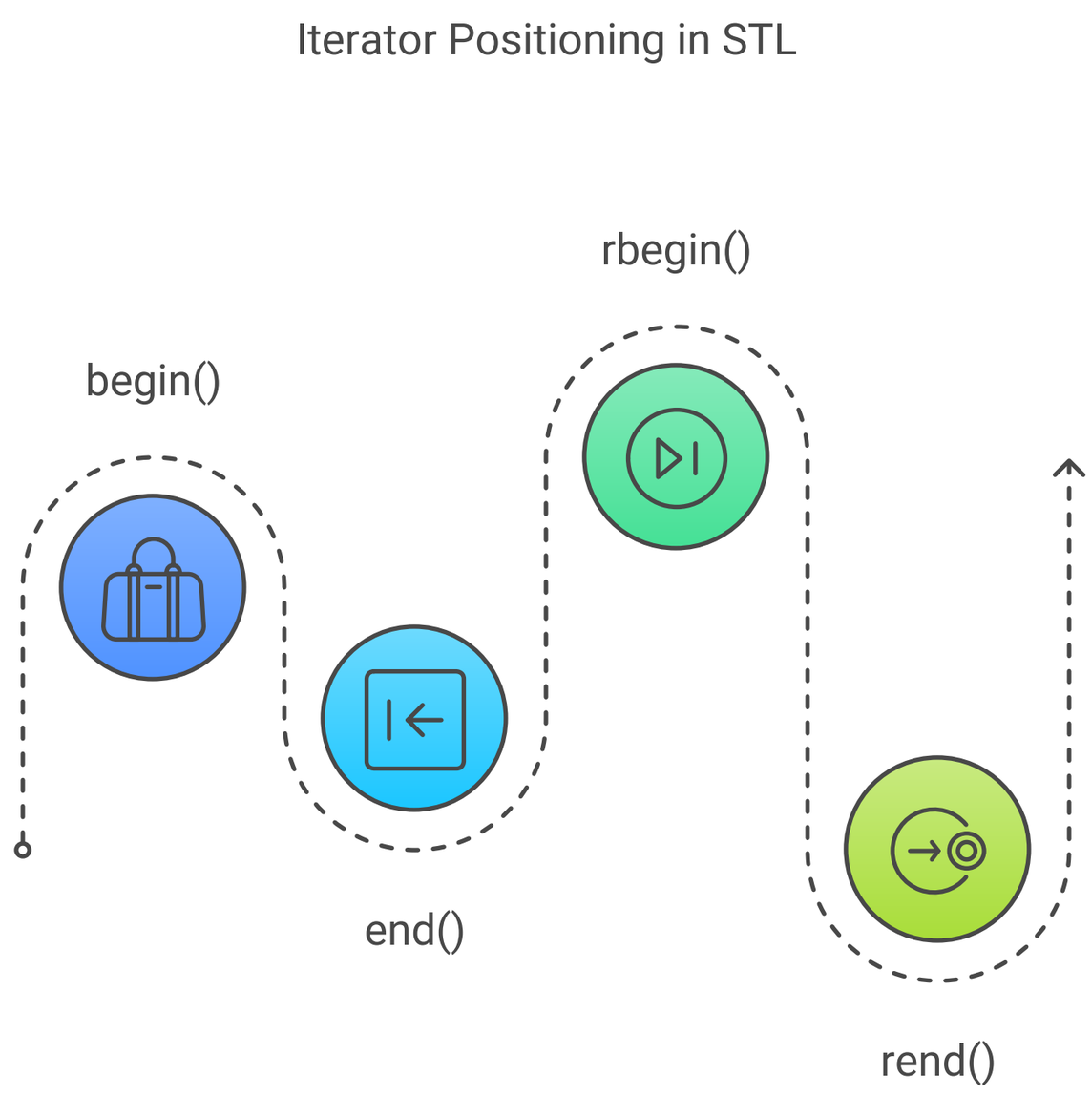
```
scores.erase("Bob");                    // Erases the entry with key "Bob"

// 6. Iterating through the map
cout << "Scores in the map:" << endl;
for (const auto& pair : scores) {
    cout << pair.first << ": " << pair.second << endl;
}

// 7. Checking size and emptiness
cout << "Size of map: " << scores.size() << endl;
if (scores.empty()) {
    cout << "Map is empty!" << endl;
} else {
    cout << "Map is not empty!" << endl;
}

    return 0;
}
```

7. Iterators

Iterators are objects that point to elements within a container and allow sequential access to elements. Iterators are used with algorithms and containers to traverse elements in various ways.

Common Iterator Types:

- **begin()**: Points to the first element in a container.
- **end()**: Points to one past the last element in a container.
- **rbegin()** and **rend()**: Reverse iterators that point to the last element and one before the first element, respectively.



Types of Iterators

1. **Input Iterator**: Reads data from a container. It can move only forward.
2. **Output Iterator**: Writes data to a container. It can move only forward.
3. **Forward Iterator**: Can read and write data. Moves only forward.
4. **Bidirectional Iterator**: Moves forward and backward.
5. **Random Access Iterator**: Can move in any direction and jump multiple positions.

Examples of Using Iterators

1. Basic Iteration through a Vector

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

int main() {
    vector<int> numbers = {10, 20, 30, 40, 50};

    cout << "Elements in vector: ";
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

In this example, **begin()** returns an iterator to the first element, and **end()** returns an iterator to one past the last element. Using **++it** increments the iterator, moving to the next element.

2. Reverse Iterator in Vector

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

int main() {
    vector<int> numbers = {10, 20, 30, 40, 50};

    cout << "Elements in reverse order: ";
    for (auto rit = numbers.rbegin(); rit != numbers.rend(); ++rit) {
        cout << *rit << " ";
    }
    cout << endl;

    return 0;
}
```

Here, **rbegin()** gives a reverse iterator pointing to the last element, and **rend()** gives a reverse iterator pointing one before the first element. This lets us iterate backward through the vector.

3. Iterating through a Map

With maps, iterators allow access to both the key and the value of each element in the container.

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

int main() {
    map<string, int> scores = {{"Alice", 90}, {"Bob", 85}, {"Charlie", 88}};

    cout << "Scores in map:" << endl;
    for (auto it = scores.begin(); it != scores.end(); ++it) {
        cout << it->first << ": " << it->second << endl;
    }

    return 0;
}
```

In a map, **it->first** refers to the key, and **it->second** refers to the value. This allows direct access to each key-value pair in the container.

4. Constant Iterators

If you only need to read elements without modifying them, use **const_iterator**. This prevents modification of elements through the iterator.

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

int main() {
    vector<int> numbers = {10, 20, 30};

    cout << "Constant iterator output: ";
    for (vector<int>::const_iterator it = numbers.cbegin(); it != numbers.cend(); ++it) {
        cout << *it << " ";
        // *it = 40; // Error: cannot modify element through const_iterator
    }
    cout << endl;

    return 0;
}
```

cbegin() and **cend()** provide **const_iterator**s. Attempting to modify elements through a **const_iterator** will cause a compilation error.

5. Iterators with Set (Bidirectional)

Sets do not support random access iterators, but they do support bidirectional iterators, which means you can increment or decrement iterators to move forward or backward.

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

int main() {
    set<int> numbers = {5, 10, 15, 20};

    cout << "Elements in set: ";
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

Since sets are ordered, the iterator accesses elements in ascending order by default. However, you cannot access elements by index, as sets don’t support random access.

6. Random Access with Vector

Random access iterators allow you to perform arithmetic operations like **+**, **-**, **+=**, and **-=** to jump directly to a specific position.

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

int main() {
    vector<int> numbers = {10, 20, 30, 40, 50};

    auto it = numbers.begin();
    it += 2; // Jump to the third element (index 2)

    cout << "Third element in vector: " << *it << endl;
```

```
    return 0;
}
```

In this example, **it += 2** moves the iterator forward by two positions, allowing direct access to the element at that position.

7. Using advance and distance with Iterators

- **advance**: Moves an iterator forward or backward by a specified number of steps.
- **distance**: Calculates the number of steps between two iterators.

```
#include <iostream>
#include <vector>
#include <iterator> // For advance and distance
using namespace std;

int main() {
    vector<int> numbers = {10, 20, 30, 40, 50};

    auto it = numbers.begin();
    advance(it, 3); // Move iterator to the fourth element

    cout << "Fourth element: " << *it << endl;

    // Calculate distance between two iterators
    int dist = distance(numbers.begin(), it);
    cout << "Distance from beginning to fourth element: " << dist << endl;

    return 0;
}
```

advance(it, n) shifts the iterator **it** by **n** positions, while **distance(start, end)** calculates the number of elements between two iterators.

Summary

The STL provides tools to manage data efficiently, combining containers for storage, iterators for navigation, and algorithms for data manipulation. Familiarity with STL components enables you to write code that is concise, maintainable, and performance-optimized.