CS-1201 Object Oriented Programming

Standard Template Library

Arbish Akram

Department of Computer Science
Government College University

Standard Template Library

- The Standard Template Library contains many templates for useful algorithms and data structures.
- C++ STL is a set of data structures and algorithms that are commonly used during coding.
- For example, when solving a problem where a linked list is required, we can utilize the built-in `list` in the C++ STL library, instead of creating a linked list from scratch.
- The STL consists of three main components:
  - **Algorithms**
  - **Containers**
  - **Iterators**

Containers

A container is a generic class which implements a certain data structure.

1. Sequence container
   - Array
   - Vector
   - Queue
   - List
2. Associative containers
   - Set
   - MultiSet
   - Map
   - MultiMap

Vector

A `vector` has several advantages over an array:

- **No need to declare size**: Unlike an array, you do not need to declare the number of elements.
- **Dynamic resizing**: `Vectors` automatically increase their size when new elements are added.
- **Simpler syntax**: You can retrieve the number of elements using simpler syntax than with an array.

To use `vector`, include the header file:

```
#include <vector>
```

To declare a `vector` of integers:

```
vector<int> myVector;
```

Vectors

The following are some other useful methods for working with vectors:

- `at(int)`: Returns the value at a specific position in the vector. For example, given `vector<int> numbers = {4, 6, 8}`,
  - `numbers.at(0)` is 4
  - `numbers.at(1)` is 6
  - `numbers.at(2)` is 8
- `push_back(value)`: Adds a new value at the end of the vector. Example: `numbers.push_back(10)` will add 10 to the end of the vector, increasing its size.
- `pop_back()`: Removes the last element of the vector, reducing its size.
- `size()`: Returns the current number of elements in the vector.
- `clear()`: Empties the vector, changing the `size()` to 0.
- `empty()`: Returns `true` if the vector contains 0 elements, `false` otherwise.

Vector: Example

```
1  #include<iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6      vector<int> myVector;
7      int enteredVal, position;
8      cout << "Enter an integer ";
9      cin >> enteredVal;
10     myVector.push_back(enteredVal);
11     cout << "Size of the list is " << myVector.size() << endl;
12     cout << "The list: " << endl;
13     for (int i = 0; i < myVector.size(); ++i)
14         cout << " " << myVector[i] << endl;
15     cout << "Enter a position to display ";
16     cin >> position;
17     cout << "The item at position " << position << " is: ";
18     cout << myVector.at(position) << endl;
19     return 0;
20 }
```

Iterator

Iterators in C++ provide a way to access and traverse elements in containers like vectors. Some important iterator methods include:

- `begin()`: Returns an iterator pointing to the first element in the container. If the container is empty, `begin()` returns the same as `end()`.
- `end()`: Returns an iterator pointing just beyond the last element in the container. It is important to note that `end()` does not point to the last element itself, but to the position after it.
- `insert(position, value)`: Inserts a value at the specified position within the container. It returns an iterator pointing to the newly inserted element.
- Example: Insert the value 15 at the third position in a vector `nums`:

```
nums.insert(nums.begin() + 2, 15);
```

- Example: Insert the value 100 just before the last element of the vector:

Iterator: Example

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

int main() {

    // initialize vector of int type
    vector<int> numbers {1, 2, 3, 4, 5};

    // initialize vector iterator to point to the first element
    vector<int>::iterator itr = numbers.begin();
    cout << "First Element: " << *itr << "  "<<endl;

    // change iterator to point to the last element
    itr = numbers.end() - 1;
    cout << "Last Element: " << *itr;

    return 0;
}
```

Algorithm

- An algorithm is a series of instructions to solve a particular problem.
- In C++, we can use the Standard Template Library (STL) to implement commonly used algorithms.
- The algorithms in the STL are known as the **Algorithms Library**. Some of the most commonly used algorithms are:
  - **Sorting algorithms**: Used to sort the elements in a container, like `sort()`.
  - **Searching algorithms**: Used to search for a specific element, like `find()`.
  - **Copying algorithms**: Used to copy data from one container to another, like `copy()`.
  - **Counting algorithms**: Used to count occurrences of an element, like `count()`.

Algorithm: Example

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // For sort, find, copy, count
#include <iterator>  // For ostream_iterator
using namespace std;
int main() {
    vector<int> nums = {10, 20, 30, 20, 40, 50, 10, 60};
    // 1. Sorting the vector
    sort(nums.begin(), nums.end());
    cout << "Sorted vector: ";
    for (int num : nums) {
        cout << num << " ";
    }
    cout << endl;
    // 2. Finding an element in the vector
    int searchValue = 30;
    auto it = find(nums.begin(), nums.end(), searchValue);
    if (it != nums.end()) {
        cout << "Found "<<searchValue<<" at position "<<(it - nums.begin());
        cout<<endl;
    } else {
        cout << searchValue << " not found in the vector." << endl;
    }
```

Algorithm: Example

```
1
2  // 3. Copying the vector to another vector
3  vector<int> copiedVec(nums.size());
4  copy(nums.begin(), nums.end(), copiedVec.begin());
5  cout << "Copied vector: ";
6  for (int num : copiedVec) {
7      cout << num << " ";
8  }
9  cout << endl;
10 // 4. Counting occurrences of a value in the vector
11 int countValue = 20;
12 int count = std::count(nums.begin(), nums.end(), countValue);
13 cout << "The value "<<countValue<<"occurs "<<count<<"time(s) in the vector.";
14 cout<< endl;
15 return 0;
16 }
```

Associative Containers

What Are Associative Containers?

- Associative containers store data in key-value pairs.
- Provide fast retrieval, insertion, and deletion based on keys.
- Unlike sequential containers (like `vector`), associative containers maintain order and allow fast key-based access.

Types of Associative Containers

- `set`: Stores unique keys in sorted order.
- `map`: Stores key-value pairs with unique keys.
- `multiset`: Stores multiple keys (allows duplicates).
- `multimap`: Stores key-value pairs with duplicate keys allowed.

Common Operations

- `Insert`: Add elements to the container.
- `Find`: Search for an element by key.
- `Erase`: Remove an element by key.
- `Count`: Count occurrences of a key (useful for `multiset` and `multimap`).
- `Begin/End`: Return iterators to the first/last elements.

Associative Container: Example

```
1  #include <iostream>
2  #include <set>
3  #include <map>
4  #include <iterator>
5  using namespace std;
6  int main() {
7      // Set Example (Unique keys)
8      set<int> mySet = {1, 2, 3, 4, 5};
9      // Insert operation
10     mySet.insert(6);
11     cout << "Set after inserting 6: ";
12     for (const auto& val : mySet) {
13         cout << val << " ";
14     }
15     cout << "\n";
16     // Find operation
17     auto it = mySet.find(3);
18     if (it != mySet.end()) {
19         cout << "Found 3 in the set.\n";
20     } else {
21         cout << "3 not found in the set.\n";
22     }
23
```

Associative Container: Example

```
1       // Erase operation
2       mySet.erase(2);
3       cout << "Set after erasing 2: ";
4       for (const auto& val : mySet) {
5           cout << val << " ";
6       }
7       cout << "\n";
8       // Count operation
9       cout << "Count of 4 in the set: " << mySet.count(4) << "\n";
10      cout << "Count of 7 in the set: " << mySet.count(7) << "\n";
11      // Map Example (Key-Value pairs)
12      map<int, std::string> myMap;
13      myMap[1] = "One";
14      myMap[2] = "Two";
15      myMap[3] = "Three";
16      // Insert operation
17      myMap[4] = "Four";
18      cout << "Map after inserting (4, 'Four'): ";
19      for (const auto& pair : myMap) {
20          cout << pair.first << " => " << pair.second << " ";
21      }
22      cout << "\n";
23
```

Associative Container: Example

```
1       // Find operation
2       auto mapIt = myMap.find(2);
3       if (mapIt != myMap.end()) {
4           cout << "Found key 2 with value: " << mapIt->second << "\n";
5       } else {
6           cout << "Key 2 not found in the map.\n";
7       }
8       // Erase operation
9       myMap.erase(1);
10      cout << "Map after erasing key 1: ";
11      for (const auto& pair : myMap) {
12          cout << pair.first << " => " << pair.second << " ";
13      }
14      cout << "\n";
15      // Multiset Example (Allowing duplicates)
16      multiset<int> myMultiSet = {1, 1, 2, 3, 3, 3};
17      // Insert operation (duplicate values allowed)
18      myMultiSet.insert(2);
19      cout << "Multiset after inserting 2: ";
20      for (const auto& val : myMultiSet) {
21          cout << val << " ";
22      }
23      cout << "\n";
```

Associative Container: Example

```
1       // Count operation (Count occurrences)
2       cout << "Count of 3 in the multiset: " << myMultiSet.count(3) << "\n";
3       // Multimap Example (Key-Value pairs, allowing duplicate keys)
4       multimap<int, string> myMultiMap;
5       myMultiMap.insert({1, "One"});
6       myMultiMap.insert({1, "Uno"});
7       myMultiMap.insert({2, "Two"});
8       // Insert operation (duplicate keys allowed)
9       myMultiMap.insert({1, "Eins"});
10      cout << "Multimap after inserting duplicate keys: ";
11      for (const auto& pair : myMultiMap) {
12          cout << pair.first << " => " << pair.second << " ";
13      }
14      cout << "\n";
15      // Find operation (first occurrence)
16      auto multiMapIt = myMultiMap.find(1);
17      if (multiMapIt != myMultiMap.end()) {
18          cout << "First element with key 1: " << multiMapIt->second << "\n";
19      }
```

Associative Container: Example

```
1       // Erase operation
2       myMultiMap.erase(2);
3       cout << "Multimap after erasing key 2: ";
4       for (const auto& pair : myMultiMap) {
5           cout << pair.first << " => " << pair.second << " ";
6       }
7       cout << "\n";
8       return 0;
9       }
```