Standard Template Library

Introduction

STL (Standard Template Library) in C++ is a powerful library that provides a set of generic data structures and algorithms. It promotes code reusability and efficiency by offering containers, algorithms, iterators, and general-purpose functions.

Algorithms

STL provides a wide range of algorithms such as sorting, searching, reversing, and more. These algorithms allow you to efficiently manipulate and process data within containers, making tasks like finding the maximum element, sorting elements, or searching for specific values easier.

Containers

STL offers various containers like vectors, pairs, arrays, and sets. Vectors provide dynamic arrays with flexible size, pairs allow you to combine two values into a single entity, arrays provide fixed-size arrays, and sets ensure sorted and unique elements. These containers offer different ways to store and access data based on your program's requirements.

Iterators

STL iterators act as generalized pointers, enabling traversal and manipulation of container elements. They provide a uniform interface to access and modify data, regardless of the underlying container type. Iterators allow you to efficiently iterate over elements, perform operations, and support algorithms within the STL.

General Purpose Functions

STL includes helpful general-purpose functions like min, max, swap, size, and empty. These functions offer convenient ways to perform common operations. For example, min and max return the minimum and maximum of two values, swap exchanges the values of two variables, while size and empty provide information about the size and emptiness of containers.

Array functions in STL

The array functions in the STL provide convenient ways to manipulate arrays efficiently in C++. Here are some array related functions in the STL:

Memset

The memset function in the STL can be used to fill an array with specific values. It is commonly used to fill an array with 0 or -1 because memset works on byte level. It is defined in the **<cstring>** header file.

Syntax:

```
memset(arr_name,value,sizeof(arr_name));
```

Example:

```
#include <iostream>
#include <cstring>
Using namespace std;

int main() {
    int arr[4];
    memset(arr,-1,sizeof(arr));
    for(int i = 0;i<4;i++){
        cout<<arr[i]<<" ";
    }
}
```

Output

```
-1 -1 -1 -1
```

Fill

The fill function in the STL allows you to fill an array with any specific value. It provides a convenient way to initialize or overwrite elements in an array.The fill function in the STL is defined in the <algorithm> header file. It takes 3 parameters. Starting pointer of the array, end pointer of the array and the value.

Syntax:

```
fill(arr,arr+size,value);
```

Example:

```
#include <iostream>
#include <algorithm>
Using namespace std;

int main() {
    int arr[4];
    fill(arr,arr+4,5);
    for(int i = 0;i<4;i++){
        cout<<arr[i]<<" ";
    }
}
```

Output

```
5 5 5 5
```

Reverse

The reverse function in the STL is used to reverse the order of elements in an array. It swaps the elements in a way that the first element becomes the last, the second becomes the second last, and so on.The reverse function in the STL is defined in the <algorithm> header file.

It takes starting and end pointers as its parameters.

Syntax:

```
reverse(arr,arr+size);
```

Example:

```
#include <iostream>
#include <algorithm>
Using namespace std;

int main() {
    int arr[4] = {1,2,3,4}
    reverse(arr,arr+4);
    for(int i = 0;i<4;i++){
        cout<<arr[i]<<" ";
    }
}
```

Output

```
4 3 2 1
```

Sort

The sort function in the STL is used to sort the elements of an array in ascending order.The sort function in the STL is defined in the <algorithm> header file. It takes starting and end pointers as its parameters.

Syntax:

```
sort(arr,arr+size);
```

Example:

```
#include <iostream>
#include <algorithm>
Using namespace std;

int main() {
    int arr[4] = {8,6,3,4}
    sort(arr,arr+4);
    for(int i = 0;i<4;i++){
        cout<<arr[i]<<" ";
    }
}
```

Output

```
3 4 6 8
```

Swap

The swap function in the STL is used to swap the values of two elements in an array. It provides a convenient way to exchange the values of two variables.

Syntax:

```
swap(arr[i],arr[j]);
```

Example:

```
#include <iostream>
#include <algorithm>
Using namespace std;

int main() {
    int arr[4] = {8,6,3,4}
    swap(arr[1],arr[3])
    for(int i = 0;i<4;i++){
        cout<<arr[i]<<" ";
    }
}
```

Output

```
8 4 3 6
```

Sizeof

The sizeof keyword serves as a compile-time operator that calculates the size, measured in bytes, of a variable or a container.

Syntax:

```
sizeof(arr)
```

Example:

```
#include <iostream>
#include <algorithm>
Using namespace std;

int main() {
    int arr[4] = {8,6,3,4}
    cout<<sizeof(arr)<<" "<<sizeof(arr[0]);
    cout<<"Size of the array is "<<sizeof(arr)/sizeof(arr[0]);
}
```

Output

```
16 4
Size of the array is 4
```

Vectors

Vectors in C++ are dynamic arrays provided by the STL. They offer a flexible and efficient way to store and manipulate collections of elements. One advantage of vectors over arrays is their ability to dynamically resize, allowing for easy insertion and deletion of elements without explicitly managing memory. Vector is defined in <vector> header file.

Syntax:

```
vector<int> v;
```

There are two more custom ways to declare vectors.

Syntax 1:

```
vector<int> v(size);
```

Syntax 2:

```
vector<int> v(size,value)
```

Example:

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

int main() {
    vector<int> a(4);
    vector<int> b(6,5);
    for(int i = 0;i<4;i++){
        cout<<a[i]<<" ";
    }
    cout<<endl;
    for(int i = 0;i<6;i++){
        cout<<b[i]<<" ";
    }
}
```

Output:

```
0 0 0 0
5 5 5 5 5 5
```

Size

The size function returns the size of the vector it is called upon.

Syntax:

```
vector_name.size();
```

Push_back and Pop_back

The push_back function is used to add elements to the end of a vector, while pop_back removes the last element from the vector.

Syntax:

```
push_back(element);
```

Syntax:

```
pop_back();
```

Example:

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

int main() {
    vector<int> a(4,8);
    a.pop_back();
    for(int i = 0;i<a.size();i++){
        cout<<a[i]<<" ";
    }
    a.push_back(9);
    a.push_back(10);
    cout<<endl;
    for(int i = 0;i<a.size();i++){
        cout<<a[i]<<" ";
    }
}
```

Output:

```
8 8 8
8 8 8 9 10
```

More on Vectors

Iteration allows you to traverse and process each character in a string sequentially. By leveraging the accessibility features and methods available for strings, we can employ loops to iterate over and perform manipulations on the elements of a string.

at() function

The at() function is used to access elements in a vector by their index. It performs bounds checking and throws an out_of_range exception if the index is invalid, providing safer element access compared to the normal [] operator.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec;
    vec.push_back(4);
    vec.push_back(2);
    vec.push_back(5);
    int element = vec.at(1);
    cout << "Element: " << element << endl;
    return 0;
}
```

Output:

```
Element: 2
```

empty() function

The empty() function is used to check if a vector is empty. It returns a boolean value, true if the vector is empty (contains no elements), and false otherwise.

Example:

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

int main() {
    vector<int> vec;
    bool isEmpty = vec.empty();
    cout << "Is Empty: " << (isEmpty ? "Yes" : "No") << endl;
}
```

Output:

```
Is Empty: Yes
```

clear() function

The clear() function is used to remove all elements from a vector, effectively emptying it. It resets the vector to its initial state, with size zero.

Example:

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

int main() {
```

```
    vector<int> vec;
    vec.push_back(4);
    vec.push_back(2);
    vec.push_back(5);
    vec.clear();
    cout << "Size: " << vec.size() << endl;
}
```

Output:

```
Size: 0
```

reverse() function

The reverse() function is used to reverse the order of elements in a vector. It swaps the elements so that the first becomes the last, the second becomes the second last, and so on.

Example:

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

int main() {
    vector<int> vec;
    vec.push_back(4);
    vec.push_back(2);
    vec.push_back(5);
    reverse(vec.begin(), vec.end());
    for (int i = 0;i<vec.size();i++) {
         cout << vec[i] << " ";
    }
}
```

Output:

```
5 2 4
```

front() and back() functions

The front() function is used to access the first element in a vector, while the back() function is used to access the last element.

Example:

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

int main() {
    vector<int> vec;
    vec.push_back(4);
    vec.push_back(2);
    vec.push_back(5);
    int firstElement = vec.front();
    int lastElement = vec.back();
    cout << "First Element: " << firstElement << endl;
    cout << "Last Element: " << lastElement << endl;
}
```

Output:

```
First Element: 1
Last Element: 3
```

erase() function

The erase() function is used to remove elements from a vector based on the provided position or range. It adjusts the size of the vector accordingly.

Example:

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

int main() {
    vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);
    vec.push_back(5);
    vec.erase(vec.begin() + 2);
    for (int i = 0;i<vec.size();i++) {
         cout << vec[i] << " ";
    }
}
```

Output:

```
1 2 4 5
```

sort() function

The sort() function is used to sort the elements of a vector in ascending order. It rearranges the elements based on the comparison operator.

Example:

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

int main() {
    vector<int> vec;
    vec.push_back(5);
    vec.push_back(2);
    vec.push_back(4);
    vec.push_back(6);
    vec.push_back(3);
    vec.sort(vec.begin(),vec.begin());
    for (int i = 0;i<vec.size();i++) {
         cout << vec[i] << " ";
    }
}
```

Output:

```
2 3 4 5 6
```

Array of vectors

Arrays of vectors are data structures that store multiple vectors in a contiguous block of memory.

resize() function

The resize() function can be used with an array of vectors to resize each vector to a specified size. It allows you to adjust the size of each vector independently.

Example:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> arr[3];
    for (int i = 0; i < 3; i++) {
        arr[i].resize(5);
    }
    for (int i = 0; i< 3; i++) {
         cout << arr[i].size() << " ";
    }
}
```

Output:

```
5 5 5
```

Vector of vectors

Vectors of vectors, also known as nested vectors or 2D vectors, refer to a data structure that contains vectors as its elements. In other words, each element of the main vector is itself a vector. This concept is often used in programming and mathematics to represent a two-dimensional grid or matrix.

Initialization 1

A vector of vectors can be initialized by pushing back vectors into the main vector. This approach allows for dynamic creation and addition of vectors.

Example :

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

int main() {
    vector<vector<int>> vecOfVecs;
    vector<int> vec1(4,5);
    vector<int> vec2(4,3);
    vector<int> vec3(4,8);
    vecOfVecs.push_back(vec1);
    vecOfVecs.push_back(vec2);
    vecOfVecs.push_back(vec3);
    for (int i = 0; i< vecOfVecs.size(); i++) {
         for(int j = 0;j<vecOfVecs[i].size();j++){
              cout<<vecOfVecs[i][j]<<” ”;
         }
         cout<<endl;
    }
}
```

Output:

```
5 5 5 5
3 3 3 3
8 8 8 8
```

Initialization 2

A vector of vectors can be pre-initialized with a specific size and default values. This allows for efficient memory allocation and predetermined dimensions of the inner vectors.

Example:

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

int main() {
    vector<vector<int>> vecOfVecs(3, vector<int>(4, 0));
        for (int i = 0; i< vecOfVecs.size(); i++) {
         for(int j = 0;j<vecOfVecs[i].size();j++){
              cout<<vecOfVecs[i][j]<<” ”;
         }
         cout<<endl;
    }
}
```

Output:

```
0 0 0 0
0 0 0 0
0 0 0 0
```

Auto Keyword

Auto is a C++ keyword introduced in C++11 that allows the compiler to automatically deduce the data type of a variable based on its initializer. The auto keyword is used to simplify type declarations and reduce verbosity, making code more concise and maintainable. It is particularly useful when the data type is complex or its name is lengthy.

Variable substitution using the 'auto' keyword

With the auto keyword, the data type of a variable can be automatically inferred, eliminating the need for explicitly specifying the type.

Example :

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

int main() {
     auto num = 42;
     auto name = “John”;
     cout << "num: " << num << endl;
     cout << "name: " << name << endl;
}
```

Output:

```
num: 42
name: John
```

Replacing iterators with the 'auto' keyword

With the auto keyword, the data type of a variable can be automatically inferred, eliminating the need for explicitly specifying the type.

Example :

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

int main() {
    vector<int> vec;
    vec.push_back(5);
    vec.push_back(2);
    vec.push_back(4);
    vec.push_back(6);
    vec.push_back(3);
    // Previously: vector<int> :: iterator it = vec.begin();
    auto it = vec.begin();
    for (; it != vec.end(); ++it) {
         cout << *it << " ";
    }
}
```

Output:

```
5 2 4 6 3
```

Replacing Function data types with the 'auto' keyword

The auto keyword can be used to deduce the return type of a function, allowing for more flexible and expressive function declarations.

Example :

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

auto add(int a, int b) {
    return a + b;
}

auto multiply(float x, float y) {
    return x * y;
}

int main() {
     cout << "add(2, 3): " << add(2, 3) << endl;
     cout << "multiply(2.5, 3.5): " << multiply(2.5, 3.5) << endl;

}
```

Output:

```
add(2, 3): 5
multiply(2.5, 3.5): 8.75
```

Pairs in STL

Pairs are a built-in container in C++ that allow you to store a pair of values together. They are significant for situations where you need to associate two different values, such as coordinates, key-value pairs, or any other related information. Pairs provide a convenient way to work with two values as a single entity, making code more readable and concise. Pairs are present in <utility> header file

Initialization

Syntax 1:

```
pair<char, int> a;
```

This declares a pair named a with a first element of type char and a second element of type int. The values of a are uninitialized.

Syntax 2:

```
pair<char, int> a = {'b', 1};
```

This declares a pair named a and initializes its first element to 'b' and its second element to 1.

Using make_pair function:

```
pair<char, int> a = make_pair(‘A’,10);
```

The make_pair() function is a convenient way to create a pair by deducing the types of its elements based on the provided arguments.

Accessibility in Pairs:

The first and second member variables of a pair allow you to access the individual elements of the pair.

Example :

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

int main() {
     pair<char, int> a = {'b', 1};
     cout << "First: " << a.first << endl;
     cout << "Second: " << a.second << endl;
}
```

Output:

```
First: b
Second: 1
```

Vectors of pairs:

Pairs can be stored in vectors, allowing you to create a collection of pairs.

Example:

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

int main() {
    vector<pair<char,int>> vec;
    pair<char,int> b = {‘A’,1};
    pair<char,int> c = {‘B’,2};
    pair<char,int> d = {‘C’,3};
    vec.push_back(b);
    vec.push_back(c);
    vec.push_back(d);
    for (int i = 0;i<vec.size();i++) {
        cout << "Letter:" << vec[i].first << ", Number:" << vec[i].second;
    }
}
```

Output:

```
Letter: A, Number: 1
Letter: B, Number: 2
Letter: C, Number: 3
```

Sorting vectors of pairs::

Vectors of pairs can be sorted based on the first or second elements of the pairs using sorting algorithms from the C++ Standard Library, such as sort(). By default sort function uses the first elements of all the pairs to compare and sort them in ascending order.

Example:

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

int main() {
    vector<pair<int,char>> vec;
    pair<int,char> b = {3, ‘A’};
    pair<int,char> c = {1, ‘B’};
    pair<int,char> d = {2, ‘C’};
    vec.push_back(b);
    vec.push_back(c);
    vec.push_back(d);
    for (int i = 0;i<vec.size();i++) {
        cout << pair.first << ": " << pair.second << endl;
    }
}
```

Output:

```
1: Bob
2: Charlie
3: Alice
```