C++ Usefull Notes on STL and Introduction to Multi-Threading in C++

Standard Template Library (STL) in C++

- Vector in C++ STL
  - Definition of Vector
    - A vector in C++ is a dynamic array provided by the Standard Template Library (STL).
    - Unlike a static array, a vector can grow or shrink dynamically as per the need. - It stores elements in contiguous memory locations and supports random access.
    - Vectors are part of the header file.
- Syntax: Declaration of Vector

```
#include <vector>
using namespace std;
vector<datatype> vectorName;
```

- Example: Declaring a vector of Student class

```
vector<Student> students;
```

- Student Class Implementation: We will define a Student class with data members RollNo, Name, Fees, and Age:

```
#include <iostream>
#include <vector>
#include <algorithm> // For sorting
#include <string>
using namespace std;
class Student {
public:
    int RollNo;
    string Name;
    double Fees;
    int Age;

    // Constructor
    Student(int RollNo, string Name, double Fees, int Age) : RollNo(RollNo),
Name(Name), Fees(Fees), Age(Age) {}

    // Display Function
    void display() const {
        cout << "RollNo: " << RollNo << ", Name: " << Name
             << ", Fees: " << Fees << ", Age: " << Age << endl;
    }
};
```

- Operations on Vector
- Insertion of Records
    - You can use methods like push_back(), insert(), and emplace_back() to insert records.
    - push_back(): Adds an element to the end of the vector.
    - emplace_back(): Constructs the object in-place to avoid unnecessary copies.
- Example: Inserting Student Records

```
vector<Student> students;
// Adding students using push_back
students.push_back(Student(101, "John", 1500.50, 20));
students.push_back(Student(102, "Alice", 1600.75, 21));

// Adding students using emplace_back
students.emplace_back(103, "Bob", 1700.80, 22);
students.emplace_back(104, "Emma", 1800.60, 19);

// Displaying all records
cout << "Student Records After Insertion:\n";
for (const auto& student : students) {
    student.display();
}
```

- Advantages of Using Vectors
    - Dynamic Size: Vectors can grow and shrink automatically.
    - Random Access: Provides constant time access to any element using the index.
    - Flexible Operations: Supports insertion, deletion, and traversal using built-in methods.
    - Performance: Better performance than linked lists for accessing elements.
    - Compatibility: Integrates with STL algorithms like sort(), find_if(), and remove_if().
- Limitations of Vectors
    - Memory Overhead: Allocates more memory than required to optimize dynamic growth.
    - Slow Insert/Delete at Middle: Operations other than at the end require shifting of elements.
    - Not Suitable for Large Data Manipulation: If frequent insertions/deletions occur at random positions, linked lists may be a better choice.
    - Contiguous Storage: If memory fragmentation occurs, allocating large vectors may fail.
- Conclusion
    - Vectors are a powerful container in C++ STL that provide dynamic memory management and efficient data manipulation.
    - They are ideal when random access and dynamic resizing are needed, but they can be less efficient for frequent insertions/deletions in the middle of the collection.
- list<> STL in C++
    - A list in C++ STL is a doubly linked list.
    - It allows dynamic insertion and deletion of elements from both ends and at any position efficiently.
    - Unlike vectors, lists do not require contiguous memory locations.
    - It is part of the header file.
- Syntax: Declaration of List

```
#include <list>
using namespace std;
list<datatype> listName;
```

- Example: Declaring a list of Student class

```
list<Student> students;
```

- Student Class Implementation: We define a Student class with data members RollNo, Name, Fees, and Age:

```
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

class Student {
public:
    int RollNo;
    string Name;
    double Fees;
    int Age;

    Student(int RollNo, string Name, double Fees, int Age) {
        this->RollNo = RollNo;
        this->Name = Name;
        this->Fees = Fees;
        this->Age = Age;
    }

    void display() {
        cout << "RollNo: " << RollNo << ", Name: " << Name
             << ", Fees: " << Fees << ", Age: " << Age << endl;
    }
};
```

- Operations on List:
- Insertion of Records
    - Lists allow insertion using push_back(), push_front(), insert(), and emplace_back().
    - push_back(): Adds an element at the end of the list.
    - push_front(): Adds an element at the beginning of the list.
- Example: Inserting Student Records

```

```

```
list<Student> students;

students.push_back(Student(101, "John", 1500.50, 20));
students.push_back(Student(102, "Alice", 1600.75, 21));

students.push_front(Student(103, "Bob", 1700.80, 22));

cout << "Student Records After Insertion:\n";
for (auto student : students) {
    student.display();
}
```

- Searching for a Record
  - To search for a record, use the find_if() algorithm with a custom condition.
- Example: Searching for a Student by RollNo

```
int searchRollNo = 102;
auto it = find_if(students.begin(), students.end(),
                  [searchRollNo](Student& s) { return s.RollNo == searchRollNo;
});

if (it != students.end()) {
    cout << "Student Found:\n";
    it->display();
} else {
    cout << "Student with RollNo " << searchRollNo << " not found.\n";
}
```

- Sorting Records
  - Lists can be sorted using the sort() method, which accepts a custom comparator.
- Example: Sorting by RollNo

```
students.sort([](Student& a, Student& b) { return a.RollNo < b.RollNo; });
cout << "Students Sorted by RollNo:\n";
for (auto student : students) {
    student.display();
}
```

- Example: Sorting by Fees in Descending Order

```
students.sort([](Student& a, Student& b) { return a.Fees > b.Fees; });
cout << "Students Sorted by Fees (Descending):\n";
for (auto student : students) {
    student.display();
}
```

- Updating a Record
  - To update a record, iterate through the list and modify the desired element.
- Example: Updating Fees of a Student by RollNo

```
int updateRollNo = 103;
double newFees = 2000.00;

for (auto& student : students) {
    if (student.RollNo == updateRollNo) {
        student.Fees = newFees;
        cout << "Updated Record:\n";
        student.display();
        break;
    }
}
```

- Deleting a Record
  - To delete a record, use remove_if() or erase().
  - remove_if(): Removes all elements satisfying a condition.
- Example: Deleting a Student by RollNo

```
int deleteRollNo = 101;
students.remove_if([deleteRollNo](Student& s) { return s.RollNo == deleteRollNo;
});
cout << "Records After Deletion:\n";
for (auto student : students) {
    student.display();
}
```

- Advantages of Using List
  - Dynamic Size: Lists grow or shrink dynamically as needed.
  - Efficient Insertions/Deletions: Inserting or deleting elements at any position is faster compared to vectors.
  - No Memory Reallocation: Memory is allocated as nodes, avoiding frequent reallocation.
  - Bidirectional Traversal: Lists are implemented as doubly linked lists, allowing traversal in both directions.
- Limitations of Using List
  - No Random Access: Unlike vectors, lists do not support direct access via indices.
  - Extra Memory Overhead: Each node requires additional memory for pointers.
  - Slower Traversal: Sequential traversal is slower compared to contiguous memory structures like vectors.
  - Performance: Sorting and searching operations are slower compared to arrays or vectors.

Comparison of Vector and List in C++ STL

- Storage Structure:
    - A vector uses contiguous memory similar to an array, while a list uses non-contiguous memory implemented as a doubly linked list.
    - This gives vectors better memory efficiency, while lists provide flexibility in dynamic allocation.
- Accessing Elements:
    - Vectors support random access with constant time complexity O(1) using the [] operator or at() function. - Lists, on the other hand, do not support random access; accessing elements requires traversal, leading to linear time complexity O(n).
- Insertion and Deletion:
    - Vectors are efficient for insertions and deletions at the end using push_back() or pop_back() (O(1) amortized). However, inserting or deleting elements in the middle is costly (O(n)) because all subsequent elements need to be shifted.
    - Lists allow efficient insertions and deletions at any position (O(1)) as they use pointers to link nodes. The list's insert() and remove() functions work directly with iterators, which avoids the need for shifting elements.
- Memory Management:
    - Vectors are more memory-efficient since they store elements contiguously without any additional overhead.
    - Lists, however, require extra memory for pointers in each node, resulting in higher memory consumption.
- Iterators:
    - Vectors support random access iterators, which allow direct access to any element. Lists support only bidirectional iterators, meaning traversal is sequential in either direction. Additionally, inserting elements into a vector may invalidate existing iterators due to memory reallocation, whereas list iterators remain valid unless the pointed node is deleted.
- Sorting:
    - Vectors can use the std::sort() algorithm from the header, which is efficient due to random access and contiguous memory. Lists cannot use std::sort() but instead provide their own list::sort() method, which works directly on nodes but is generally slower.
- Performance:
    - Access: Vectors outperform lists for accessing elements because they provide constant time random access.
- Insertion/Deletion:
    - Lists outperform vectors when inserting or deleting elements frequently in the middle because vectors require shifting of elements.
- Sorting:
    - Vectors are faster for sorting due to better cache locality, while lists take more time as they require node traversal.
- Use Cases:
    - Vectors are ideal when random access is required, and operations like insertions or deletions occur at the end.
    - Lists are preferred when frequent insertions and deletions are needed in the middle, and stable iterators are important.
- Conclusion Vectors are better for scenarios requiring fast access and efficient memory usage, while lists excel when insertions, deletions, and pointer stability are priorities. The choice between them depends on the specific needs of the application.

- Other STL Classes
    - stack<>
    - queue<>
    - set<>
    - dequeue<>

map<K,V>

- A map in C++ STL is an associative container that stores key-value pairs, where each key is unique, and keys are automatically sorted in ascending order by default.
- The map is implemented as a self-balancing binary search tree (e.g., Red-Black Tree), making key-based operations efficient.
- It provides efficient lookup, insertion, and deletion, typically with a time complexity of O(log n).
- Syntax: Declaration

```
#include <map>
using namespace std;
map<KeyType, ValueType> mapName;
```

- Example with Student and Address:

```
map<Student, Address> StudentAddressMap;
```

- Implementation of map<Student,Address>
    - Classes: Student and Address
        - Define a Student class with data members: RollNo, Name, Fees, and Age.
        - Define an Address class with data members: City and Country.

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

class Student {
public:
    int RollNo;
    string Name;
    double Fees;
    int Age;
    Student(int RollNo, string Name, double Fees, int Age) {
        this->RollNo = RollNo;
        this->Name = Name;
        this->Fees = Fees;
        this->Age = Age;
    }
    void Display() {
```

```
        cout << "RollNo: " << RollNo << ", Name: " << Name << ", Fees: " << Fees
<< ", Age: " << Age<<endl;
    }
    bool operator<(const Student& s) const {
        return RollNo < s.RollNo;               // Compare by RollNo
    }
};
class Address {
public:
    string City;
    string Country;
    Address(string City, string Country) {
        this->City = City;
        this->Country = Country;
    }
    void Display() {
        cout << ", City: " << City << ", Country: " << Country << endl;
    }
};
```

- Operations on Map
  - Insertion of Records
    - Use the insert() or [] operator to add key-value pairs to the map.
    - Example:

```
    map<Student, Address> StudentAddressMap;

    // Insert using `insert` method
    StudentAddressMap.insert({Student(101, "Malkeet", 1500.50,
20), Address("Delhi", "India")});
    StudentAddressMap.insert({Student(102, "Saket", 1600.75, 21),
Address("Mumbai", "India")});

    // Insert using `[]` operator
    StudentAddressMap[Student(103, "Johnson", 1700.80, 22)] =
Address("Berlin", "Germany");

    // Display all records
    cout << "Student-Address Map After Insertion:"<<endl;
    for (auto& pair : StudentAddressMap) {
        pair.first.Display();
        pair.second.Display();
    }
```

- Searching for a Record
  - Use the find() method to search for a record based on the key.
  - Example:

```
    Student searchStudent(102, "Saket", 1600.75, 21); // Same key
details must match
    auto it = StudentAddressMap.find(searchStudent);
    if (it != StudentAddressMap.end()) {
        cout << "Record Found:\n";
        it->first.Display();
        it->second.Display();
    } else {
        cout << "Record not found.\n";
    }
```

- Sorting Records
  - Maps automatically sort their keys. To customize sorting, use a comparator function.
  - Example: Sorting by Age

```
    struct CompareByAge {
        bool operator()(const Student& a, const Student& b) {
            return a.Age < b.Age; // Sort by Age
        }
    };
    // Declare map with custom comparator
    map<Student, Address, CompareByAge> sortedByAgeMap;
    sortedByAgeMap[Student(101, "Malkeet", 1500.50, 20)] =
Address("Delhi", "India");
    sortedByAgeMap[Student(102, "Saket", 1600.75, 21)] =
Address("Mumbai", "India");
    // Display records
    cout << "Map Sorted by Age:\n";
    for (auto& pair : sortedByAgeMap) {
        pair.first.Display();
        pair.second.Display();
    }
```

- Updating a Record
  - To update a record, use the [] operator or modify the value directly.
  - Example: Update Address for a Student

```
Student updateStudent(103, "Johnson", 1700.80, 22);
auto it = StudentAddressMap.find(updateStudent);
if (it != StudentAddressMap.end()) {
    it->second = Address("Paris", "France"); // Update Address
    cout << "Record Updated:\n";
    it->first.Display();
    it->second.Display();
}
```

- Deleting a Record

    - Use the erase() method to remove a record based on the key.
    - Example:

```
Student deleteStudent(101, "Malkeet", 1500.50, 20);
StudentAddressMap.erase(deleteStudent); // Erase by key
cout << "Map After Deletion:"<<endl;
for (auto& pair : StudentAddressMap) {
    pair.first.Display();
    pair.second.Display();
}
```

- Advantages of Using Map
    - Key-Based Access: Maps provide efficient lookups, insertions, and deletions using keys.
    - Automatic Sorting: Keys are always sorted, which simplifies operations like range queries.
    - Custom Comparators: Maps allow custom sorting using comparator functions.
    - Iterators: Stable iterators allow easy traversal of elements.
- Limitations of Using Map
    - Slower than Unordered Map: Maps have O(log n) complexity due to tree-based implementation, whereas unordered maps offer O(1) average-case complexity.
    - Key Duplication Not Allowed: Each key in a map must be unique.
    - Memory Overhead: Maps require extra memory for maintaining tree structures.
- Example: map<Student,Address>

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
class Student {
public:
    int RollNo;
    string Name;
    double Fees;
    int Age;

    Student(int RollNo, string Name, double Fees, int Age) {
        this->RollNo = RollNo;
        this->Name = Name;
        this->Fees = Fees;
        this->Age = Age;
    }

    void Display() {
        cout << "RollNo: " << RollNo << ", Name: " << Name << ", Fees: " << Fees
<< ", Age: " << Age<<endl;
    }

    bool operator<(const Student& s) const {
        return RollNo < s.RollNo; // Compare by RollNo
    }
```

```
};

class Address {
public:
    string City;
    string Country;

    Address(string City, string Country) {
        this->City = City;
        this->Country = Country;
    }

    void Display() {
        cout << ", City: " << City << ", Country: " << Country << endl;
    }
};

int main() {
    map<Student, Address> StudentAddressMap;

    // Insertion
    StudentAddressMap[Student(101, "Malkeet", 1500.50, 20)] = Address("Delhi",
"India");
    StudentAddressMap[Student(102, "Saket", 1600.75, 21)] = Address("Mumbai",
"India");

    // Display
    cout << "Initial Records:"<<endl;
    for (auto& pair : StudentAddressMap) {
        pair.first.Display();
        pair.second.Display();
    }

    // Searching
    Student searchStudent(102, "Saket", 1600.75, 21);
    auto it = StudentAddressMap.find(searchStudent);
    if (it != StudentAddressMap.end()) {
        cout << "Record Found:"<<endl;
        it->first.Display();
        it->second.Display();
    }

    // Updating
    Student updateStudent(101, "Malkeet", 1700.80, 22);
    auto updateIt = StudentAddressMap.find(updateStudent);
    if (updateIt != StudentAddressMap.end()) {
        updateIt->second = Address("Paris", "France");
        cout << "Updated Record:"<<endl;
        updateIt->first.Display();
        updateIt->second.Display();
    }

    // Deleting
    StudentAddressMap.erase(Student(101, "Malkeet", 1700.80, 22));
```

```
    cout << "Records After Deletion:"<<endl;
    for (auto& pair : StudentAddressMap) {
        pair.first.Display();
        pair.second.Display();
    }
    return 0;
}
```

Multi-Threaing in C++

- What is Multithreading?
  - Multithreading allows a program to run multiple parts (called threads) at the same time. This helps make the best use of the CPU. Each thread acts like a small, lightweight process within the main program.
- Multithreading in C++
  - Before C++11, we used libraries like POSIX threads (). However, these were not standard, leading to compatibility issues.
  - With C++11, the std::thread library was introduced, making multithreading easier and portable.
- What is std::thread?
  - std::thread is a class that represents a single thread in C++.
  - You can create and start a thread by passing a task to a thread object.
- Syntax to Create a Thread

```
std::thread
ThreadName(CallableObject:FunctionPointer/Function/Lambda/ClassObject);
```

- Here, ThreadName is the name of the thread, and CallableObject is the code the thread will execute.
- What Can Be Passed as CallableObject?
  - A callable is the task the thread will run. It can be one of the following:
    - A Function Pointer: A normal function.
    - A Lambda Expression: A small, inline function written using [ ].
    - A Function Object: An object with an overloaded operator().
    - A Non-Static Member Function: A member function of a class, called on an object.
    - A Static Member Function: A class-level function that does not need an object.
- How to Start a Thread?
  - Define the callable.
  - Example:

```
void MyFunction()
{
    for(int i=0;i<5;i++)
    {
        cout<<"Value is:   "<<i<<endl;
    }
}
```

- Pass the callable to the thread object's constructor.
- Example:

```
thread Th1(MyFunction);     //Creating Thread by creating object of
thread class
```

- The thread will automatically start running once the object is created.
- Example:

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

// function to be used in callable
void MyFun(int N)
{
    for (int i = 0; i < N; i++) {
    cout <<i<<" by Thread"<<this_thread::get_id()<<endl;
    }
}
int main()
{
    thread th(MyFun,5);     //First Thread will execute MyFun Function

    //Creating a lambda Function
    auto MyFun1=[](int X)
    {
        for (int i = 0; i < X; i++)
        {
        cout <<i<<" by Thread"<<this_thread::get_id()<<endl;
        }
    };

    thread th2(MyFun1,5);   //Assigning lambda to Thread th2

    for(int i=0;i<5;i++)
    {
        cout <<i<<" by Thread"<<this_thread::get_id()<<endl;
    }
    return 0;
}
```

- Benefits of Multithreading
  - Executes multiple tasks at the same time.
  - Makes programs faster and more efficient by using the CPU fully.
- Examples:

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

void MyFun1(int X)
{
    for(int i=0;i<X;i++)
    {
        cout<<i<<" By: "<<this_thread::get_id()<<endl;
    }
}
void MyFun(int X)
{
    for(int i=0;i<X;i++)
    {
        cout<<i<<" By: "<<this_thread::get_id()<<endl;
        //this_thread::sleep_for(chrono::milliseconds=200);
    }
}
class Test
{
public:

    void operator()(int X)
    {
        for(int i=0;i<X;i++)
        {
            cout<<i<<" By"<<this_thread::get_id()<<endl;
            this_thread::sleep_for(chrono::milliseconds(300));
        }
    }
};
int main()
{
    Test t;

    thread th(t,10);
    th.join();
    return 0;
}
int main2()
{
    auto MyFun=[](int X){

        for(int i=0;i<X;i++)
        {
            cout<<i<<" By"<<this_thread::get_id()<<endl;
            this_thread::sleep_for(chrono::milliseconds(300));
        }
    };
    thread th(MyFun,10);        //Creating and starting of thread th
    th.join();                  //Here Main will be wait be for the execution of
```

```
th thread
    return 0;
}
int main1()
{

    thread th1(MyFun,10);
    thread th2(MyFun1,10);
    cout<<"Am in Main"<<this_thread::get_id()<<endl;
    return 0;
}
```