بِسْمِ اللهِ الرَّحْمٰنِ الرَّحِيْمِ

GU TECH

Object Oriented Programming

Lab 12

July 30, 2025

Contents

STL Containers

Recommended reading: C++ Data Structures and STL

C++ Standard Template Library (STL) containers are template classes that provide various data structures for storing and organizing collections of objects. They offer efficient ways to manage data and are categorized into three main types:

1. **Sequence containers:** These containers store elements in a linear order, allowing sequential access.

    - `std::vector`: A dynamic array that can grow or shrink in size. Offers fast random access and efficient additions and deletions at the end. Additions and deletions in the middle are inefficient.
    - `std::deque`: A **double-ended queue** that allows efficient insertions and deletions at **both** ends. Allows random access.
    - `std::list`: A doubly linked list that provides efficient insertions and deletions anywhere. **No** random access.
    - `forward_list`: A singly linked list, more memory-efficient than `std::list` but only allows forward traversal.
    - `array`: A fixed-size array that wraps a C-style array, providing bounds checking and other utility functions.

2. **Associative containers:** These containers store elements in a linear order, allowing sequential access.

    - `std::set`: Stores unique elements in a sorted order.
    - `std::multiset`: Similar to `std::set` but allows duplicate elements.
    - `std::map`: Stores key-value pairs, where keys are unique and sorted.
    - `std::multimap`: Similar to `std::map` but allows duplicate keys.

3. **Unordered associative containers:** These containers store elements in an unordered fashion using hash tables, offering average constant-time complexity for search, insertion, and deletion.

- `std::unordered_set`: Stores unique elements in an unordered fashion.
- `std::unordered_multiset`: Similar to `std::unordered_set` but allows duplicate elements.
- `std::unordered_map`: Stores key-value pairs in an unordered fashion, where keys are unique.
- `std::unordered_multimap`: Similar to `std::unordered_map` but allows duplicate keys.

4. **Container adapters:** These are not standalone containers but provide a specific interface built on top of existing containers.

   - `std::stack`: A LIFO (Last-In, First-Out) data structure.
   - `std::queue`: A FIFO (First-In, First-Out) data structure.
   - `std::priority_queue`: A queue where elements are retrieved based on their priority.

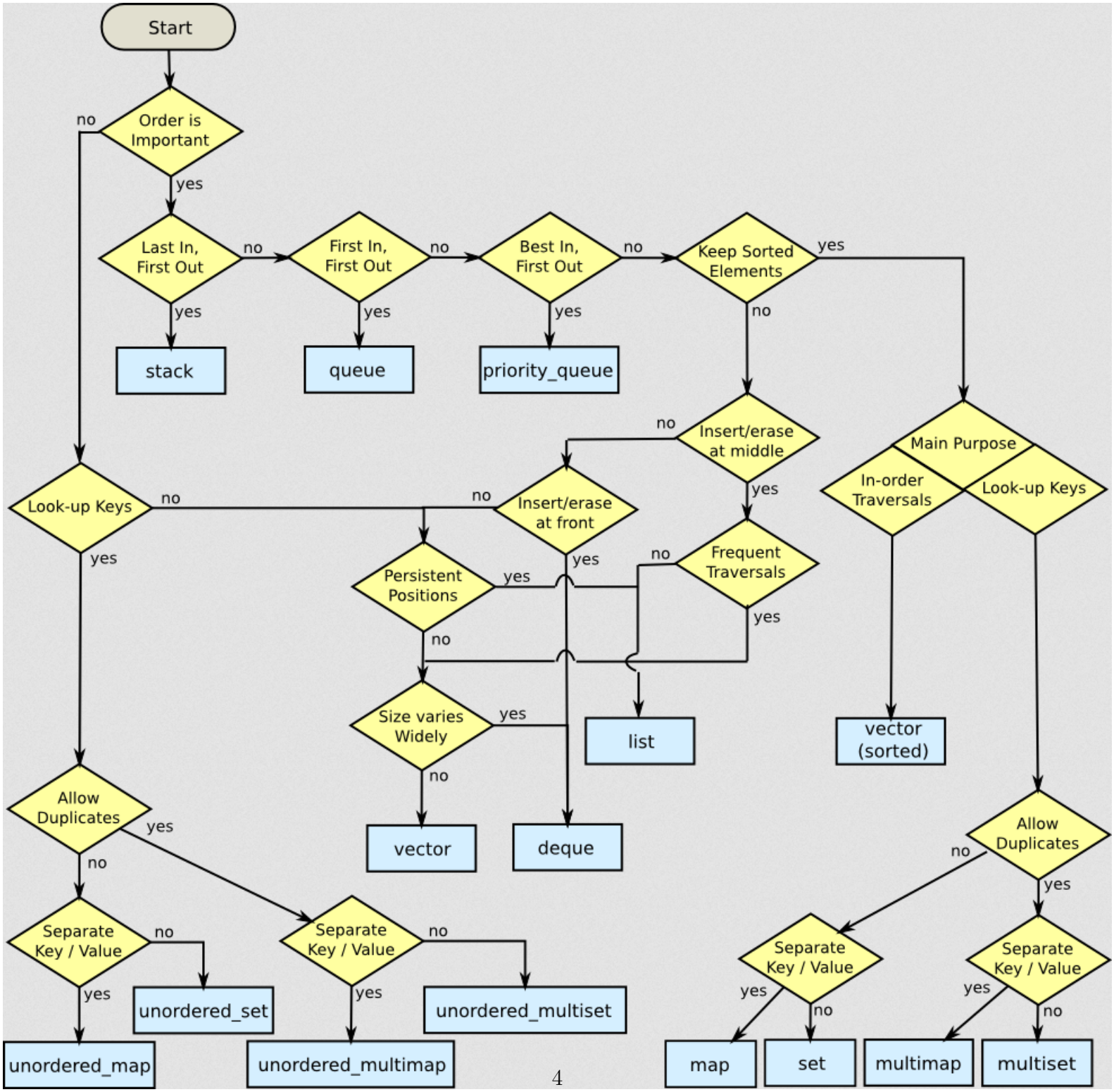
Start
Order is Important
no
yes
Last In, First Out
First In, First Out
Best In, First Out
Keep Sorted Elements
stack
queue
priority_queue
Look-up Keys
Insert/erase at middle
Insert/erase at front
Persistent Positions
Frequent Traversals
Main Purpose
In-order Traversals
Look-up Keys
Size varies Widely
list
vector (sorted)
vector
deque
Allow Duplicates
Separate Key / Value
unordered_map
unordered_set
unordered_multimap
unordered_multiset
map
set
multimap
multiset

Chart source: Stackoverflow

Vectors

`std::vector` is the most used data structure and is typically used as a replacement for arrays.

- It lets us add elements without specifying the number of elements beforehand.
- It allows random access (by using subscript `[]` or `.at()` function).
- Unlike arrays, it allows insertion and deletion in the middle but it is extremely inefficient to do so.
  - On insertion, all succeeding elements are shifted forward.
  - On deletion, all succeeding elements are shifted backward.

For comparision, an `std::list` allows efficient insertions and deletions in the middle as well but does not allow random access. The access has to be sequential.

Vector Usage

Before we can use vectors or other STL containers, we need to include their respective headers.

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

void main()
{
    vector<int> equation;   //3x^2 + 2x^1 - 8x^0 = 0
    equation.push_back(3);
    equation.push_back(2);
    equation.push_back(-8);

    for (int i = 0; i < equation.size(); i++)
    {
        cout << equation[i] << " ";
    }
    cout << "\n";

    for (int i = 0; i < equation.size(); i++)
    {
        cout << equation.at(i) << " ";
    }
    cout << "\n";
}
```

It is important to note that STL `pop()` functions do not return values.

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

void main()
{
    vector<int> equation;
    //9x^4 + 12x^3 - 44x^2 - 32x^1 + 64x^0 = 0
    equation.push_back(9);
    equation.push_back(12);
    equation.push_back(-44);
    equation.push_back(-32);
    equation.push_back(64);

    for (int i = 0; i < equation.size(); i++)
    {
        cout << equation[i] << " ";
        //9 12 -44 -32 64
    }
    cout << "\n";

    //  cout << equation.pop_back() << "\n"; //Error
    equation.pop_back();

    for (int i = 0; i < equation.size(); i++)
    {
        cout << equation[i] << " ";
        //9 12 -44 -32
    }
    cout << "\n";
}
```

The follwoing code demonstrates insertion and deletion in the middle.

Reminder: Insertion and deletion in the middle of a vector are extremely inefficient.

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

void printVector(vector<int> v)
{
    for (int i = 0; i < v.size(); i++)
    {
            cout << v[i] << " ";
        }
        cout << "\n";
}

int main()
{
    vector<int> numbers = {0, 1, 2, 3, 4};

    printVector(numbers);
    //0 1 2 3 4
    numbers.insert(numbers.begin() + 2, 5);
    printVector(numbers);
    //0 1 5 2 3 4
    numbers.erase(numbers.begin() + 2);
    printVector(numbers);
    //0 1 2 3 4

    //numbers.insert(2, 5); //Does not work
    //numbers.erase(2);     //Does not work

    return 0;
}
```

Runtime Polymorphism with STL Containers

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

class Animal
{ public: virtual void speak(){cout << "gibberish\n";} };

class Cat : public Animal
{ public: void speak(){cout << "Meow\n";} };

class Dog : public Animal
{ public: void speak(){cout << "Woof\n";} };

int main()
{
    vector <Animal*> jungle;
    jungle.push_back(new Cat());
    jungle.push_back(new Dog());
    jungle.push_back(new Cat());

    for (int i = 0; i < jungle.size(); i++)
    {
        jungle[i]->speak();
    }

    for (int i = 0; i < jungle.size(); i++)
    {
        delete jungle[i];
    }

    return 0;
}
```

Reminder: Every single `new` should have a corresponding `delete` and every single `new[]` should have a corresponding `delete[]`.

Iterating through STL Containers

- STL containers that allow random access may be iterated with `for` loops using the `.size()` function, just like iterating regular arrays.
  - These containers include `std::vector`, `std::deque` and `std::array`.
  - `std::array` should not be confused with regular arrays.
- All iterable containers can be iterated with `for each` loops, (not to be confused with `std::for_each` loop).
  - This includes iterating by value and iterating by reference.
- All iterable containers can also be iterated with `iterators`.
- `std::stack` and `std::queue` cannot be iterated at all (by design).
- The `std::for_each` loop from the `<algorithm>` header may also be used for all iterable containers but this manual does not cover that.

Iterating with For Loops

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

int main()
{
    vector<int> numbers = {0, 1, 2, 3, 4};
    deque<string> words =
    {"comic", "sans", "reigns", "supreme"};

    for (int i = 0; i < numbers.size(); i++)
    {
        cout << numbers[i] << " ";
    }
    cout << "\n";

    for (int i = 0; i < words.size(); i++)
    {
        cout << words[i] << " ";
    }
    cout << "\n";

    return 0;
}
```

Iterating with For Each Loops

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

int main()
{
    vector<int> numbers = {0, 1, 2, 3, 4};

    for (const int i : numbers)
    {
        cout << i << " ";
        //0 1 2 3 4
    }
    cout << "\n";

    for (int& i : numbers)
    {
        i = i * i;
    }

    for (const int i : numbers)
    {
        cout << i << " ";
        //0 1 4 9 16
    }
    cout << "\n";

    return 0;
}
```

When iterating STL containers where each element holds a single data type, we can simply use a variable of that data type in our `for each` loop.

For example, when iterating `vector<int> numbers = 0, 1, 2, 3, 4`, we can simply use `for (int i : numbers)`.

However, for STL containers that contain a pair of values, we need to use `for (std::pair<data_type_1, data_type_2> i) : containerName`.

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

int main()
{
    map<string, string> students = {{"K164078", "KMT"},
                                    {"K163859", "SMR"},
                                    {"K163860", "STM"}};

    for (std::pair<string, string> i : students)
    {
        cout << i.first << ", " << i.second << "\n";
    }

    return 0;
}
```

Note: When using associative containers, the `<` operator is used by default for sorting. This implies that when custom classes are used with associative containers, the `<` operator should be overloaded.

Associative containers store elements in a sorted order.

Iterating with Iterators

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

int main()
{
    vector<int> numbers = {0, 1, 2, 3, 4};

    for (vector<int>::iterator it =
        numbers.begin(); it != numbers.end(); it++)
    {
        cout << *it << " ";
        //0 1 2 3 4
    }
    cout << "\n";

    for (vector<int>::reverse_iterator rit =
        numbers.rbegin(); rit != numbers.rend(); rit++)
    {
        cout << *rit << " ";
        //4 3 2 1 0
    }
    cout << "\n";

    return 0;
}
```

Also see: https://cplusplus.com/reference/vector/vector/rbegin/

Lab Tasks

1. **Library Book Catalog:** Create a `Book` class with appropriate constructor, getters, and the following attributes:

   - `isbn (string)`
   - `author (string)`
   - `title (string)`

   Store books in an `unordered_map<string, Book>` (key = ISBN).

   Implement a `search` function that returns the number of books by a given author and takes a `multimap<string, Book>` parameter for author indexing.

   Hints:

   - You will need the following headers.
     - `<unordered_map>`
     - `<map>`
   - Your `search` function can have the following prototype:
     `int search(unordered_map<string, Book>& library, string author, multimap<string, Book>& searchResult)`
   - You can use the `.insert()` function for insertions. You will need to use braces inside the function, like this: `.insert({author, Book})`.
   - Refer to the `map` example given in the manual for further help.
   - You may consult the Internet, but do not copy code from ChatGPT.
   - You may use the following books to test your code (you can also come up with your own).

```
Book b0("9635280258938", "Mercury", "Asad");
Book b1("4702599831795", "Venus", "Asad");
Book b2("8417392305289", "Earth", "Asad");
Book b3("4323267170016", "Mars", "Taimoor");
Book b4("8951093252041", "Saturn", "Unknown");
Book b5("5580166798252", "Jupiter", "Taimoor");
```

2. Create a class `Student` with attributes `string name` and `string rollNo`. Read names and roll number from a csv (or `.txt`) file, and store the class objects in an ordered container. Overload the `<` operator so that the `Student` objects are sorted by roll numbers lexicographically, and **make sure** that the comparision is case **insensitive**.

   You may use the following file for testing:

```
24g-bCs007, hUzAiFa,
24g-bCs003, bArIrA,
24G-BcS001, Abdur Rehman,
24G-BcS002, AhSaN,
24g-bCs004, fAtImA,
```

3. Create a class `Polynomial` utilizing `std::vector` and overload the `+`, `-` and `*` operators and preferably the `/` operator as well for a bonus of 2 marks. Hints to follow later.