LABORATORY 10
The C++ Standard Template Library

Objectives

The main objective of this work is to acquaint students with the C++ Standard Template Library (STL). The laboratory session will cover the most important STL containers, the usage of iterators to traverse there containers, and will introduce lambda expressions.

Theoretical concepts

Containers in STL

A container is a data structure that stores several objects (of the same type) organized according to some specific rules. Containers can be categorized into sequence containers and associative containers. Sequence containers (such as vectors or arrays) store independent objects in which elements can be accessed sequentially. On the other hand, associative containers (such as maps or dictionaries) store key-value pairs.
The table below contains a brief description of the basic containers from the C++ standard library.

Class	**Description**	**Main operations**
array	Defined in <array> Encapsulates fixed size arrays. Elements are stored contiguously. It is a sequence container.	**at, operator[] –** for retrieving elements at a given position in the array. **size –** the number of elements in the array.
vector	Defined in <vector> Encapsulates dynamic size arrays. Elements stored contiguously. It is a sequence container.	**at, operator[] –** for retrieving elements at a given position in the vector. **push_back**, **emplace_back –** inserts an element at the end of the vector (constant time complexity). **pop_back –** removes the last element from the vector (constant time complexity). **size –** the number of elements in the vector. **capacity –** the capacity of the vector. **insert –** inserts elements on the specified position in the vector. Linear time complexity. **erase –** erases the specified elements from the container. Linear

		time complexity.
deque	Defined in <deque> Encapsulates a container with dynamic size that can be expanded or contracted on both ends (front and back). Elements are **not stored contiguously.** It is a sequence container.	**at, operator[] –** for retrieving elements at a given position in the deque. **push_back**, **emplace_back –** inserts an element at the end of the deque (constant time complexity). **push_front**, **emplace_front –** inserts an element at the front of the deque (constant time complexity). **pop_back –** removes the element from the end of the deque (constant time complexity). **size –** number of elements in the container.
list	Defined in <list> Encapsulates a double-linked list container that allows for constant time insert and erase operations anywhere within the sequence, and iteration in both directions. Does **not allow for direct random access.** It is a sequence container.	**insert –** inserts element(s) at the specified position(s). **erase –** erases element(s) from the specified position(s). **push_back**, **emplace_back, push_front**, **emplace_front, pop_back, pop_front –** Constant complexity. **size –** number of elements in the container.
stack	Defined in <stack> Encapsulates a stack container that operates in a LIFO manner (last-in first-out) – i.e. insertion and removal of the elements is performed at the beginning of the container. It is a container adaptor that "pushes" and "pops" elements on the top of the container.	**top –** access the element on the front of the stack. **push, emplace –** add an element in the stack (on the first position). Constant complexity. **pop –** removes an element from the stack (the first element from the stack). Constant complexity. **size –** number of elements in the container.
queue	Defined in <queue> Encapsulates a queue container that operates in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. It is a container adapter that "pushes" elements on the top, and "pops" them from the end of the container.	**front –** access the first element of the queue. Constant complexity. **front –** access the last element in the queue. Constant complexity. **push, emplace –** add an element in the queue (after the last element). Constant complexity. **pop –** removes an element from the stack (the first element from the queue). Constant complexity. **size –** number of elements in the container.
map	Defined in <map> It is an associative (referenced by their keys and not by their position in the container) and ordered container. Typically implemented as binary search trees.	**insert –** inserts a key-value pair in the map. Logarithmic complexity if a single element is inserted. **operator[], at –** accesses elements in the container based on the specified key. Logarithmic complexity. **find –** searches the container for an element with the specified key. Logarithmic complexity. **size –** number of elements in the container.

set	Defined in <set> Encapsulates a container that stores unique elements following a specific order. It is an ordered container with **unique** elements. Typically implemented as binary search trees.	**insert –** inserts a key-value pair in the set. Logarithmic complexity if a single element is inserted. **find –** searches the container for an element. Logarithmic complexity. **size –** number of elements in the container.
unordered _map	Defined in <unordered_map> Associative container that stores key-value pairs; the elements are not sorted but instead organized internally into "buckets" (hash table). Search, insertion, and removal of elements have average constant-time complexity.	**insert –** inserts a key-value pair in the map. **operator[], at –** accesses elements in the container based on the specified key.

The example below illustrates the usage of two of the sequence containers *std::vector* and *std::array* for storing student grades. Both are used for storing a sequence of elements, but a *std::vector* has a dynamic size, meaning that its size can change at runtime, allowing elements to be added or removed. In other words, the size of a *std::vector* can grow or shrink as needed, though this may involve copying elements to a new memory location if more space is needed. On the other hand, *std::array* encapsulates a fixed size array, so the number of elements from a std::array is known at compilation time and does not change.

```
#include <iostream>
#include <array>
#include <vector>

int main() {
    // Using std::array for the grades of 5 students
    std::array<int, 5> grades = {75, 85, 90, 95, 88}; //
Fixed size of 5
    std::cout << "Using std::array (grades, fixed size):" <<
std::endl;
    for(size_t i = 0; i < grades.size(); ++i) {
        std::cout << grades[i] << " ";
    }
    std::cout << "\nTotal grades (std::array): " <<
grades.size() << std::endl;
```

```
    // Using std::vector for storing the grades of several
students
    std::vector<int> examResults = {75, 85, 90, 95, 88}; //
Initial size of 5, but size can change
    std::cout << "\nUsing std::vector (grades, dynamic
size):" << std::endl;
    // Adding more grades by using the push_back function
    examResults.push_back(92);
    examResults.push_back(81);

    for(size_t i = 0; i < examResults.size(); ++i) {
        std::cout << examResults[i] << " ";
    }
    std::cout << "\nTotal exam results (std::vector): " <<
examResults.size() << std::endl;

    return 0;
}
```

```
Using std::array (grades, fixed size):
75 85 90 95 88
Total grades (std::array): 5

Using std::vector (grades, dynamic size):
75 85 90 95 88 92 81
Total exam results (std::vector): 7
```

STL iterators

STL iterators allow for flexible and efficient manipulation of container data. An iterator is an object that provides a way to access the elements of a container sequentially. In other words, an iterator provides a mechanism to move from one element to another within the container.

Every C++ container offers, at a minimum, two types of iterators: `<container>::iterator` – read/write iterator for both reading and writing operations and `<container>::const_iterator` – read-only iterator for read-only operations, where <container> is the name of the containers (array, vector, map, etc.).

In addition, all STL containers include four basic member functions which return iterators:

- *begin*(): returns an iterator representing the beginning of the elements in the container;

- *end*(): returns an iterator representing the element just past the end of the elements;
- *cbegin*(): returns a const (read-only) iterator representing the beginning of the elements in the container;
- *cend*(): returns a const (read-only) iterator representing the element just past the end of the elements.

The example below illustrates how to iterate over a std::vector<int> using both std::vector<int>::iterator for read-write operations and std::vector<int>::const_iterator for read-only operations.

```
#include <iostream>
#include <vector>

int main() {
    // Initialize a vector of integers
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Using iterator for read-write operations
    std::cout << "Using iterator (read-write):" <<
std::endl;
    for(std::vector<int>::iterator it = vec.begin(); it !=
vec.end(); ++it) {
        // Access and modify the elements (*it)
        *it = *it * 2; // modify each element through the
iterator
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // Using const_iterator for read-only operations
    std::cout << "Using const_iterator (read-only):" <<
std::endl;
    for(std::vector<int>::const_iterator cit = vec.cbegin();
cit != vec.cend(); ++cit) {
        // Access and display the elements (*cit)
        std::cout << *cit << " "; // via a constant
iterator, the values cannot be modified
    }
    std::cout << std::endl;

    return 0;
```

```
}
```

```
Using iterator (read-write):
2 4 6 8 10
Using const_iterator (read-only):
2 4 6 8 10
```

In the example, the object *vec* of type *std::vector<int>* is initialized with the integers from 1 to 5. The first for-loop uses a *std::vector<int>::iterator* (which can be used for read-write operation) to iterate over vec, doubling each element. Then, the second for loop uses a *std::vector<int>::const_iterator* to iterate over the modified vec in read-only mode to display its values.

The next example demonstrates how to iterate over a *std::map<std::string, int>* using *std::map<std::string, int>::iterator* for read-write operations and *std::map<std::string, int>::const_iterator* for read-only operations.
A *std::map* stores elements as key-value pairs, sorted by keys. In the example, the key is a string representing the name of a person and the value is an integer representing the corresponding age. A *std::map* iterator points to elements that are *std::pair<const key_type, value_type>* objects, where *key_type* is the type of the keys in the map and *value_type*is the type of the associated values. Because the keys in a map are unique and immutable, the first element of the pair (the key) is constant (const), meaning that the key of an existing element in a map cannot be changed.

```
#include <iostream>
#include <map>

int main() {
    // Initialize a map where keys are strings and values
are integers
    std::map<std::string, int> ageMap = {
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35}
    };

    // Using iterator for read-write operations
    std::cout << "Modifying values using iterator:" <<
std::endl;
    for(std::map<std::string, int>::iterator it =
ageMap.begin(); it != ageMap.end(); ++it) {
```

```
        // Increment the age by 1
        it->second += 1;
        std::cout << it->first << ": " << it->second <<
std::endl;
    }

    // Using const_iterator for read-only operations
    std::cout << "Displaying updated values using
const_iterator:" << std::endl;
    for(std::map<std::string, int>::const_iterator cit =
ageMap.cbegin(); cit != ageMap.cend(); ++cit) {
        // Display the key-value pairs
        std::cout << cit->first << ": " << cit->second <<
std::endl;
    }

    return 0;
}
```

```
Modifying values using iterator:
Alice: 31
Bob: 26
Charlie: 36
Displaying updated values using const_iterator:
Alice: 31
Bob: 26
Charlie: 36
```

Lambda functions

In C++, a lambda function/expression is an anonymous function that can be defined inline, without polluting the global namespace of the program. The syntax for a lambda expression is the following:

```
[ captureClause ] ( parameters ) -> returnType
{
    // lambda body
}
```

The *captureClause* contains the variables that are accessible inside the lambda function (by default, the lambda function does not have access to the variables defined outside the lambda function). The capture clause can be empty. There

are two special symbols that can be used to capture all the variables outside the lambda function:

- "=" is used to capture all the outside variables by value;
- "&" is used to capture all the outside variables by reference.

parameters can be empty if no parameters are required, just like for a normal function that doesn't take any parameters.

The return type (-> *returnType*) is optional.

The example below shows a scenario in which a vector of students needs to be sorted according to different criteria. To sort this vector both in ascending and descending order by criteria such as name and GPA, four separate functions would typically need to be defined for single-use. However, lambda functions present an alternative by enabling the definition of these sorting criteria directly within the local context.

```
#include <string>
#include <vector>
#include <iostream>
#include <algorithm> // for for_each, sort
using namespace std;

class Student {
public:
   Student(const std::string& name, double gpa)
       : m_name(name), m_gpa(gpa) {}

   std::string name() const { return m_name; }
   double gpa() const { return m_gpa; }
   void setGpa(double gpa){ m_gpa = gpa;}
   friend ostream& operator<<(ostream &os, const Student
&s){
       os<<s.m_name<<" "<<s.m_gpa<<"; ";
       return os;
   }
private:
   std::string m_name;
   double m_gpa;
};

void display(const vector<Student> &students){

   std::for_each(students.begin(), students.end(),
                 // lambda function to display a student
                 [](const Student &stud){cout<<stud<<" ";
   });
```

```
   cout<<endl;
}

int main(){
   std::vector<Student> students = {
       { "Alice", 3.7 },
       { "Bob", 2.9 },
       { "Charlie", 3.2 },
       { "David", 3.9 },
       { "Emma", 3.8 }
   };

   // Lambda function to sort by name in increasing order
   auto sort_by_name_ascending = [](const Student& a, const
Student& b) {
       return a.name() < b.name();
   };

   // Lambda function to sort by GPA in decreasing order
   auto sort_by_gpa_descending = [](const Student& a, const
Student& b) {
       return a.gpa() > b.gpa();
   };

   // Sort by name in increasing order
   std::sort(students.begin(), students.end(),
sort_by_name_ascending);
   cout<<"By name ascending: "<<endl;
   display(students);
   cout<<endl;

   // Sort by name in decreasing order
   std::sort(students.begin(), students.end(),  [](const
Student& a, const Student& b) {
       return a.name() > b.name();
   });
   cout<<"By name descending: "<<endl;
   display(students);
   cout<<endl;

   // Sort by GPA in increasing order
   std::sort(students.begin(), students.end(), [](const
Student& a, const Student& b) {
       return a.gpa() < b.gpa();
   });
   cout<<"By gpa ascending: "<<endl;
   display(students);
```

```
   cout<<endl;

   // Sort by GPA in decreasing order
   std::sort(students.begin(), students.end(),
sort_by_gpa_descending);
   cout<<"By gpa descending: "<<endl;
   display(students);
   cout<<endl;


   // increase the gpa by gpaAdd points
   double gpaAdd = 0.5;
   std::transform(students.begin(), students.end(),
students.begin(),
                  // lambda function to increase the gpa of
each student by gpaAdd (gpaAdd is used in the capture
clause)
      [gpaAdd](Student &s){
            s.setGpa(s.gpa()+gpaAdd);
            return s;
   });
   cout<<"After increasing gpa with  "<<gpaAdd<<endl;
   display(students);
   cout<<endl;
   return 0;
}
```

```
By name ascending:
Alice 3.7;  Bob 2.9;  Charlie 3.2;  David 3.9;  Emma 3.8;

By name descending:
Emma 3.8;  David 3.9;  Charlie 3.2;  Bob 2.9;  Alice 3.7;

By gpa ascending:
Bob 2.9;  Charlie 3.2;  Alice 3.7;  Emma 3.8;  David 3.9;

By gpa descending:
David 3.9;  Emma 3.8;  Alice 3.7;  Charlie 3.2;  Bob 2.9;

After increasing gpa with  0.5
David 4.4;  Emma 4.3;  Alice 4.2;  Charlie 3.7;  Bob 3.4;
```

Proposed problems

1. Create a class called *Song* based on the following UML class diagram:

Song
artist: string # title: string - lyrics: vector<string>
+Song(artist: string, title: string, lyrics: string) + getArtist(): string + getTitle(): string + getLyrics(): vector<string>

The parameterized constructor takes as parameters a string containing the lyrics of the song, but the attribute *lyrics* store the lyrics split into words. All words should be converted to lowercase and any punctuation or numbers should be ignored in the attribute *lyrics*.

2. Create a class *SongCollection* that stores a std::vector of *Song*s. The class should have at least a constructor that loads the song collection from a file. The songs are stored in the following format:

```
__Artist: Artist Name 1
__Song: Song Name 1
__Lyrics:
Line 1 of the lyrics 1
…
Line N of the lyrics 1

__Artist: Artist Name 2
__Song: Song Name 2
__Lyrics:
Line 1 of the lyrics 2
…
Line N of the lyrics 2
```

3. Add a method in the *SongCollection* class that computes and returns the unique artists from the *SongCollection* (use a *std::set* to compute the unique artists).

4. Organize songs by their artists using an STL associative container, creating a "bucket" for each artist to hold all their songs.

5. Use lambda expressions to sort the vector of Songs:
   - in increasing order based on the artist's name;
   - in descending order based on the title;
   - in increasing order based on the number of words in the lyrics.

6. Using containers from the C++ STL, design a compound data structure that allows for efficient searching of songs by a specific word. Describe the time and space complexity of this data structure.