This chapter focuses on the Standard Template Library (STL) of C++. We describe the general structures of the STL that are mainly composed of container classes, algorithms, functions and iterators. A second section focuses on the description of the container class vector that is used to create vector with elements of a given type and do various operations on them such as adding new elements to a vector.

1. Introduction to the Standard Template Library (STL)

1.1. General content of STL. The C++ programming language contains a library called Standard Template Library that offer many features to programmer. The STL is composed of the following elements:

- containers. They are template classes that are use to define object that contains a collection of other object (their elements).
- functions. Each container contains public functions that can give access to a specific object’s element, the size of an object or other information.
- algorithms. Some containers also present algorithms that perform operation on all the elements of an object of the container. For instance, algorithms can be used to sort elements of an object with respect to a comparison operators like $<$. 
- iterators. They are objects used to point to an element of a container’s object. They allow to navigate through the elements of the object. They are similar to pointer however containers present specific iterator that point to the first or the last element of an object which allows to avoid overread or overwrite memory issue.
- allocators. They provide information on how allocation and deallocation of dynamic memory of container’s object is managed. A default allocator is provided by C++ for each container.

1.2. Some STL containers. Here is a short description of few containers that you may encounter. We refer to the literature for an exhaustive list and description of all the containers provided by C++.

- std::array. This container is used to create a fix sized collection of objects. This objects represents the elements of the array. To instantiate an object of std::array, we need to provide the type of the elements of the array and the number of elements of the array. It can be done as follows:

```cpp
std::array<int, 4> myarray;
```

The element of an array are stored contiguously.
• **std::vector.** This container is used to create vector, i.e. array, that can change of size. The elements of a vector are stored contiguously, the dynamical storage of these elements (adding/removing element) is handled by the container via the use of adequate functions. More information on this container are provided in the next section.

• **std::stack.** This container stores objects of other template class. It uses the LIFO (Last In First Out) rule, meaning objects are added and removed from the end of the collection.

• **std::list.** It is a container that is implemented as a doubly-linked list. Each elements of such a list contains two link that point to the previous and the next element. More information can be found in the literature.

The following section focuses on the description of the class container vector. We note that many features (iterators, functions, constructors) of the class vector are also available in other STL classes.

2. **The class vector**

This section shows how to use vectors defined with the container vector of the C++ programming language. We provide information on the various constructors of the class vector, the functions that give access to elements of a vector, iterators that allows to navigate between elements of a vector. The feature of overloading operator, such as the addition, is also discussed.

2.1. **Declaring a vector.** Object of the class vector can be instantiated with the following constructors.

• **default constructor.** Create an empty vector with size zero.

```cpp
std::vector<double> vec;
```

• **fill constructor.** Create a vector of n elements which are all set to a given value.

```cpp
std::vector<double> vec(5, -4.0);
```

• **Copy constructor.** Create a vector as a copy of a given vector.

```cpp
std::vector<double> vec(vec1); // or vec = vec1
```

• **Range constructor.** Create a vector based on a given sequence of elements stored contiguously. It takes two iterators as inputs. The iterators respectively point to the first element to read and the last element to read. Here is an example that reproduce the effect of a copy constructor.

```cpp
std::vector<double> vec(vec1.begin(), vec1.end());
```

Unlike the copy constructor, the range constructor can use a sequence of an array’s elements. We note that the standard c++11 introduced two new type of constructor. In addition to the above constructors, the class vector contains an overload of the assignment operator equal (=). This operator equal allows to change the dimension of the left hand side vector if it does not match the one of the right hand side. Eventually the class vector have a destructor that clean up the dynamical memory allocated to stored an object when the object goes out of scope.
2.2. Accessing a vector’s element. The class vector contains various public function members that can be used by an object of this class. Here is a list of these functions.

- **size**. Return the size of the vector (no input).
- **at**. Take an integer i as input and return a reference to the i-th element of the vector.
- **front**. Return a reference to the first element of the vector (no input).
- **back**. Return a reference to the last element of the vector (no input).
- **operator[]**. Allow to access or set an element i of a vector as follows:

\[
\text{vec1}[2]=3.0;
\]

where we assume that i is larger than zero and smaller than the size of the vector vec1.

We remind that a public function of a class is accessible with the operator "." as follows:

\[
\text{int n=vec1.size();}
\]

2.3. Modify a vector. The class vector presents numerous features that allows to bring change to the structure of a vector. In particular, these features allows us to change the size of a vector, add or remove elements of a vector. Here is a non exhaustive list of public functions members of the class vector that lead to a modification of a vector object.

- **resize**. Change the size of a vector.

\[
\text{vec1.resize(n);}
\]

If the size decreases, the content of the first n elements is unchanged. If the size increases, the new element are either set to zero, via a default constructor, or to a given value as follows.

\[
\text{vec1.resize(n, 10.0);}
\]

- **push_back**. Add a new element at the end of the vector. The value of the new element is given to push_back as follows:

\[
\text{vec1.push_back(value_new_element);}
\]

- **pop_back**. Remove the last element of vector.

\[
\text{vec1.pop_back();}
\]

- **erase and insert**. Remove, respectively insert, one or more elements into a vector at a given position.

- **swap**. Swap two vectors.

\[
\text{vec1.swap(vec2);}
\]

- **clear**. Erase all the elements of a vector. The size of the vector becomes zero.

\[
\text{vec1.clear();}
\]
2.4. Iterators.

- **begin.** Iterator to the the beginning of the vector (first element).
- **end.** Iterator to the end of the vector (after last element).
- **rbegin.** Reverse iterator to the last element of the vector.
- **rend.** Reverse iterator before the first element of the vector.

Note that incrementing a reverse iterator with `++` lead to moving to the previous element as it is ordered in the vector. Iterator that can not be changed (const) can be declared by using the above iterators preceded by the letter `c` (cbegin, cend, crbegin and crend). These iterators can be used to print the value of a vector’s elements as follows:

```cpp
for (auto it= vec.begin(); it< vec.end(); it ++){
    std::cout << *it << std::endl;
}
```

where we assume that the vector `vec` has been defined in previous instructions. We refer to the example `ex43_STL.cc` for more details.

2.5. Operators. The class vector does not contain overload of operators such as the addition. If such feature are needed, a programmer has to implement them. Here is one way of overloading the operator addition such that it works with objects of the class vector.

```cpp
using namespace std;

template <typename T>
vector<T> operator+ ( const vector<T>& vec1, const vector<T>& vec2)
{
    if (vec1.size()!= vec2.size()){
        throw "vector dont have same length in operator +";
    }
    else {
        vector<T> vec_out;
        for (int i=0; i<vec1.size(); i ++){
            vec_out.push_back(vec1[i]+vec2[i]);
        }
        return vec_out;
    }
}
```

Similar instructions can be implemented to overload operators such as multiplication operator (*) or stream operator (<< and >>). We remind that the assignment operator equal (=) is already overloaded in the class vector definition.