The standard template library (STL) contains:
- Containers
- Algorithms
- Iterators

- A container is a way that stored data is organized in memory, for example an array of elements.
- Algorithms in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.
- Iterators are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array.

Containers, Iterators, Algorithms

Algorithms use iterators to interact with objects stored in containers.

Containers

- A container is a way to store data, either built-in data types like int and float, or class objects.
- The STL provides several basic kinds of containers:
  - <vector>: one-dimensional array
  - <list>: double linked list
  - <deque>: double-ended queue
  - <queue>: queue
  - <stack>: stack
  - <set>: set
  - <map>: associative array

Sequence Containers

- A sequence container stores a set of elements in sequence, in other words each element (except for the first and last one) is preceded by one specific element and followed by another. <vector>, <list> and <deque> are sequential containers.
- In an ordinary C++ array the size is fixed and can not change during run-time. It is also tedious to insert or delete elements. Advantage: quick random access.
- <vector> is an expandable array that can shrink or grow in size, but still has the disadvantage of inserting or deleting elements in the middle.
Sequence Containers

- `<list>` is a double linked list (each element has points to its successor and predecessor), it is quick to insert or delete elements but has slow random access
- `<deque>` is a double-ended queue, that means one can insert and delete elements from both ends, it is a kind of combination between a stack (last in first out) and a queue (first in first out) and constitutes a compromise between a `<vector>` and a `<list>`

Associative Containers

- An associative container is non-sequential but uses a key to access elements. The keys, typically a number or a string, are used by the container to arrange the stored elements in a specific order, for example in a dictionary the entries are ordered alphabetically.

Associative Containers

- A `<set>` stores a number of items which contain keys. The keys are the attributes used to order the items, for example a set might store objects of the class Person which are ordered alphabetically using their name.
- A `<map>` stores pairs of objects: a key object and an associated value object. A `<map>` is somehow similar to an array except instead of accessing its elements with index numbers, you access them with indices of an arbitrary type.
- `<set>` and `<map>` only allow one key of each value, whereas `<multiset>` and `<multimap>` allow multiple identical key values.

Vector Container

int array[5] = {12, 7, 9, 21, 13};
vector<int> v(array, array+5);

v.begin();
v.pop_back();
v.push_back(15);
v[3]
#include <vector>
#include <iostream>

vector<int> v(3);  // create a vector of ints of size 3
v[0]=23;
v[1]=12;
v[2]=9;    // vector full
v.push_back(17);   // put a new value at the end of array
for (int i=0; i<v.size(); i++)   // member function size() of vector
    cout << v[i] << " ";   // random access to i-th element
cout << endl;

int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array
while ( ! v.empty()) // until vector is empty
    {
        cout << v.back() << " ";   // output last element of vector
        v.pop_back();                 // delete the last element
    }
    cout << endl;

vector<Date> x(1000); // creates vector of size 1000,
                      // requires default constructor for Date
vector<Date> dates(10,Date(17,12,1999)); // initializes
                                         // all elements with 17.12.1999
vector<Date> y(x); // initializes vector y with vector x

Iterators are pointer-like entities that are used to access individual elements in a container. Often they are used to move sequentially from element to element, a process called iterating through a container.

The iterator corresponding to the class vector<int> is of the type vector<int>::iterator
# Iterators

The member functions begin() and end() return an iterator to the first and past the last element of a container.

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

int arr[] = {12, 3, 17, 8}; // standard C array
vector<int> v(arr, arr+4); // initialize vector with C array

int max(vector<int>::iterator start, vector<int>::iterator end) {
    int m=*start;
    while (start != end) {
        if (*start > m)
            m=*start;
        ++start;
    }
    return m;
}

int main() {
    vector<int>::iterator iter=v.begin(); // iterator for class vector
    // define iterator for vector and point it to first element of v
    cout << "first element of v= " << *iter; // de-reference iter
    iter++; // move iterator to next element
    iter=v.end()-1; // move iterator to last element
    cout << "max of v = " << max(v.begin(),v.end());
    return 0;
}
```

One can have multiple iterators pointing to different or identical elements in the container.
Iterators

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

int arr[] = { 12, 3, 17, 8 }; // standard C array
vector<int> v(arr, arr+4); // initialize vector with C array
for (vector<int>::iterator i=v.begin(); i!=v.end(); i++)
    // initialize i with pointer to first element of v
    // i++ increment iterator, move iterator to next element
    {
        cout << *i << " "; // de-referencing iterator returns the
    // value of the element the iterator points at
    }
cout << endl;
```

Iterator Categories

- Not every iterator can be used with every container for example the list class provides no random access iterator
- Every algorithm requires an iterator with a certain level of capability for example to use the [ ] operator you need a random access iterator
- Iterators are divided into five categories in which a higher (more specific) category always subsumes a lower (more general) category, e.g. An algorithm that accepts a forward iterator will also work with a bidirectional iterator and a random access iterator

```
input  forward  bidirectional  random access
output
```

For_Each() Algorithm

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

void show(int n)
{
    cout << n << " ";
}

int arr[] = { 12, 3, 17, 8 }; // standard C array
vector<int> v(arr, arr+4); // initialize vector with C array
for_each(v.begin(), v.end(), show); // apply function show
    // to each element of vector v
```

Find() Algorithm

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

int key;
int arr[] = { 12, 3, 17, 8, 34, 56, 9 }; // standard C array
vector<int> v(arr, arr+7); // initialize vector with C array
vector<int>::iterator iter;
cout << " enter value: ";
cin >> key;
iter=find(v.begin(),v.end(),key); // finds integer key in v
if (iter != v.end()) // found the element
    cout << "Element " << key << " found" << endl;
else
    cout << "Element " << key << " not in vector v" << endl;
```
# Find_If() Algorithm

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

Bool mytest(int n) { return (n>21) && (n <36); }

int arr[] = { 12, 3, 17, 8, 34, 56, 9 };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array

vector<int>::iterator iter = find_if(v.begin(), v.end(), mytest);
// finds element in v for which mytest is true
if (iter != v.end()) // found the element
    cout << "found " << *iter << endl;
else
    cout << "not found" << endl;
```

# Count_If() Algorithm

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

Bool mytest(int n) { return (n>14) && (n <36); }

int arr[] = { 12, 3, 17, 8, 34, 56, 9 };  // standard C array

vector<int> v(arr, arr+7);  // initialize vector with C array

int n = count_if(v.begin(), v.end(), mytest);
// counts element in v for which mytest is true
cout << "found " << n << " elements" << endl;
```

## List Container

- An STL list container is a double linked list, in which each element contains a pointer to its successor and predecessor.
- It is possible to add and remove elements from both ends of the list.
- Lists do not allow random access but are efficient to insert new elements and to sort and merge lists.
**Insert Iterators**

- If you normally copy elements using the copy algorithm you overwrite the existing contents

```
#include <list>
int arr1[] = { 1, 3, 5, 7, 9 };
int arr2[] = { 2, 4, 6, 8, 10 };
list<int> l1(arr1, arr1+5); // initialize l1 with arr1
list<int> l2(arr2, arr2+5); // initialize l2 with arr2
copy(l1.begin(), l1.end(), l2.begin());
// copy contents of l1 to l2 overwriting the elements in l2
// l2 = { 1, 3, 5, 7, 9 }
```

**Insert Iterators**

- With insert operators you can modify the behavior of the copy algorithm
  - `back_inserter` : inserts new elements at the end
  - `front_inserter` : inserts new elements at the beginning
  - `inserter` : inserts new elements at a specified location

```
#include <list>
int arr1[] = { 1, 3, 5, 7, 9 };
int arr2[] = { 2, 4, 6, 8, 10 };
list<int> l1(arr1, arr1+5); // initialize l1 with arr1
list<int> l2(arr2, arr2+5); // initialize l2 with arr2
copy(l1.begin(), l1.end(), back_inserter(l2)); // use back_inserter
// adds contents of l1 to the end of l2 = { 9, 7, 5, 3, 1, 2, 4, 6, 8, 10 }
```

**Sort & Merge**

- Sort and merge allow you to sort and merge elements in a container

```
#include <list>
int arr1[] = { 6, 4, 9, 1, 7 };
int arr2[] = { 4, 2, 1, 3, 8 };
list<int> l1(arr1, arr1+5); // initialize l1 with arr1
list<int> l2(arr2, arr2+5); // initialize l2 with arr2
l1.sort(); // l1 = {1, 4, 6, 7, 9}
l2.sort(); // l2 = {1, 2, 3, 4, 8}
l1.merge(l2); // merges l2 into l1
// l1 = { 1, 1, 2, 3, 4, 4, 6, 7, 8, 9 }, l2 = {}
```

**Functions Objects**

- Some algorithms like sort, merge, accumulate can take a function object as argument.
  - A function object is an object of a template class that has a single member function : the overloaded operator ()
  - It is also possible to use user-written functions in place of pre-defined function objects

```
#include <list>
#include <functional>
int arr1[] = { 6, 4, 9, 1, 7 };
list<int> l1(arr1, arr1+5); // initialize l1 with arr1
l1.sort(greater<int>()); // uses function object greater<int>
// for sorting in reverse order l1 = { 9, 7, 6, 4, 1 }
```
Function Objects

The accumulate algorithm accumulates data over the elements of the containing, for example computing the sum of elements

```
#include <list>
#include <functional>
#include <numeric>

int arr1[] = { 6, 4, 9, 1, 7 };
list<int> l1(arr1, arr1+5); // initialize l1 with arr1
int sum = accumulate(l1.begin(), l1.end(), 0, plus<int>());
int sum = accumulate(l1.begin(), l1.end(), 0);  // equivalent
int fac = accumulate(l1.begin(), l1.end(), 0, times<int>());
```

User Defined Function Objects

```
class squared_sum // user-defined function object
{
   public:
      int operator()(int n1, int n2) { return n1+n2*n2; }
   };
int sq = accumulate(l1.begin(), l1.end(), 0, squared_sum()); // computes the sum of squares
```

User Defined Function Objects

```
template <class T>
class squared_sum // user-defined function object
{
   public:
      T operator()(T n1, T n2) { return n1+n2*n2; }
   };
vector<complex> vc;
complex sum_vc;
vc.push_back(complex(2,3));
vc.push_back(complex(1,5));
vc.push_back(complex(2,4));
sum_vc = accumulate(vc.begin(), vc.end(),
                     complex(0,0), squared_sum<complex>());
// computes the sum of squares of a vector of complex numbers
```

Associative Containers

In an associative container the items are not arranged in sequence, but usually as a tree structure or a hash table.

- The main advantage of associative containers is the speed of searching (binary search like in a dictionary)
- Searching is done using a key which is usually a single value like a number or string
- The value is an attribute of the objects in the container
- The STL contains two basic associative containers
  - sets and multisets
  - maps and multimaps
#include <set>

string names[] = {"Ole", "Hedvig", "Juan", "Lars", "Guido"};
set<string, less<string> > nameSet(names, names+5);
// create a set of names in which elements are alphabetically
// ordered string is the key and the object itself
nameSet.insert("Patric"); // inserts more names
nameSet.erase("Juan"); // removes an element
nameSet.insert("Maria"); // sets iterator to lower start value "K"
while (iter != nameSet.upper_bound("Q"))
cout << *iter++ << endl;
// displays Lars, Maria, Ole, Patric

#include <map>

string names[] = {"Ole", "Hedvig", "Juan", "Lars", "Guido", "Patric", "Maria", "Ann"};
int numbers[] = {75643, 83268, 97353, 87353, 19988, 76455, 77443, 12221};
map<string, int, less<string> > phonebook;
for (int j=0; j<8; j++)
phonebook[names[j]] = numbers[j]; // initialize map phonebook
for (iter = phonebook.begin(); iter != phonebook.end(); iter++)
cout << (*iter).first << " : " << (*iter).second << endl;
cout << "Lars phone number is " << phonebook["Lars"] << endl;
class person
{
  private:
    string lastName;
    string firstName;
    long phoneNumber;
  public:
    person(string lana, string fina, long pho) :
      lastName(lana), firstName(fina), phonenumber(pho) {}
    bool operator<(const person& p);
    bool operator==(const person& p);
}

Maps & Multimaps

person p1("Neuville", "Oliver", 5103452348);
person p2("Kirsten", "Ulf", 5102782837);
person p3("Larssen", "Henrik", 8904892921);
multiset<person, less<person>> persSet;
multiset<person, less<person>>::iterator iter;
persSet.insert(p1);
persSet.insert(p2);
persSet.insert(p3);