Lecture 12

- **Templates** (*intro.*) *(mallar)*
  - Function templates (*funktionsmallar*) [14.2.1]
  - Class templates *(klassmallar)* [14.1.1]
  - Examples
- **S**tandard **T**emplate **L**ibrary (**STL**) [12]
  - Container classes
    - **`vector`** -- self-study [2.8]
  - Iterators [12.1]
  - Algorithms [12.2]
  - Examples

Templates

- Classes like **`Set`** and **`Matrice`** can store several variables/objects -- but, all of the same type
  - class **`Set`** of Lab 2 stores **`int`**s
    - *What if we want a* **`Set`** *of* **`string`***s or* **`Clock`**s*?*
  - class **`Matrice`** of Fö 7 stores **`double`**s
    - What if we want a **`Matrice`** of **`long double`**s*?*
- Do we need to write several functions to sort an array of **`int`**s, **`Clock`**s, or **`string`**s?
  - very identical code for **different types**
- Solution: use class and function templateS
  - Generic classes and functions used as a template *(mall)*
  - Complicated syntax

Function templates

See **funTemplates.cpp**

```
template<class T>
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Function template

Function template arguments
- A *generic type* **T**

```
string S1 = "Floyd", S2 = "Pink";
swap(S1,S2);
//swap<string>(s1,S2);
int i = 5, j = 10;
swap(i,j);
```

Compiler automatically creates a specialisation of template function **swap** for type **string**

Compiler automatically creates a specialisation of template function **swap** for type **int**

STL has a template function **swap**

Function template's specializations

Specialisations of template function **swap** automatically generated by the compiler

```
void swap(string& a, string& b)
{
    string temp = a;
    a = b;
    b = temp;
}
```

```
void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
int main() {
  string S1 = "Floyd", S2 = "Pink";
  swap(S1,S2);
  int i = 5, j = 10;
  swap(i,j);
}
```

Function templates versus functions

```
double min(double a, double b)
{
    if (a < b)
       return a;
    else
       return b;
}
```

```
template<class T>
T min(T a, T b)
{
    if (a < b)
       return a;
    else
       return b;
}
```

```
int i = 5;
double d = 4.4;

cout << min(d, i);
```

Automatic type conversion occurs

```
int i = 5;
double d = 4.4;

cout << min(d, i);
```

Don't mix the types!!

```
cout << min<double>(d,i);
```

Create a specialization where **T** is replaced by **double**

Function templates

```
template<class T>
const T& min(const T &a, const T &b)
{
    if (a < b)
       return a;
    else
       return b;
}
```

Read sec. 14.2.1

Function **min** can also be called for objects like **Clock** (see Fö 9)

Need **Clock::operator<(…)**

Improved version

- Can be applied to large objects
- Can be applied to constant objects

```
const Clock K1(12,30), K2(10,0);

cout << min(K1, K2);
```

Function templates

See **funTemplates.cpp**

```
template<class T>
const T& min(const T* A, int n)
{
    const T *p_min = A;

    for(int i = 0; i < n; i++)
        if (A[i] < *p_min)
            p_min = A+i;

    return *p_min;
}
```

An array of objects of type **T**

T A[]

A reference (address) of the smallest value stored in array **A** is returned -- efficiency!!

```
int V1[] = {3,-1,7,9};
cout << min(V1,4);
Clock V2[] = {Clock(10,30,0), Clock(0,30,0), Clock(2,0,0)};
cout << min(V2,3);
```


Templates

- Templates can have several generic arguments

```
template<class Type1, class Type2>
struct Pair
{
    Type1 first;
    Type2 second;
};
```

See *http://www.cplusplus.com/reference/utility/pair/*

```
Pair<int,int> point;
point.first = 10;
point.second = 20;
```

specialization

```
struct Pair<int,int>
{
    int first;
    int second;
};
```

```
Pair<int, double> SqRoot;
SqRoot.first = 90;
SqRoot.second = 9.4868329;
```

```
struct Pair<int,double>
{
    int first;
    double second;
};
```

Class templates

```
template <class T>
class Matrice
{
    public:
      Matrice(int l, int n);
      Matrice(const Matrice &M); //copy constructor
      …
    private:
        int lines;
        int cols;
        T *table;
};
```

```
Matrice<int> M1(2,2);
Matrice<double> M2(3,3);
```

Specializations of the class template are automatically generated by the compiler

- class **Matrice<int>**
- class **Matrice<double>**

Class templates

```
class Matrice<int>
{
    public:
      Matrice<int>(int l, int n);
      …
    private:
        int lines;
        int cols;
        int *table;
};
```

Name of the class generated by the compiler

```
Matrice<int> M1(2,2);
```

Specializations of the class template generated by the compiler

- class **Matrice<int>**

Class templates

```
class Matrice<double>
{
    public:
      Matrice<double>(int l, int n);
      …
    private:
        int lines;
        int cols;
        double *table;
};
```

```
Matrice<double> M2(3,3);
```

Specializations of the class template generated by the compiler

- class **Matrice<double>**

Class member functions templates

```
template <class T>
Matrice<T>::Matrice(int l, int c)
{
   lines = l;
   col = c;

   //Allocate memory space
   table = new T [lines*cols];
}
```

Name of the class generated by the compiler

```
Matrice<int>::Matrice(int l, int c)
{
   lines = l;
   col = c;

   //Allocate memory space
   table = new int [lines*cols];
}
```

```
Matrice<int> M1(2,2);
```

Specialization generated by the compiler

Matrice<int>::Matrice(int l, int c)

Class templates

See MatriceT.cpp

```
template <class T, int LINES = 10, int COLS = 10>
class Matrice
{
    public:
      Matrice(T init_val);
       …
      ostream& display(ostream& os);

    private:
       T table[LINES][COLS]; //no dynamic memory allocation
};
```

```
Matrice<int,2,3> M1(0);
Matrice<double> M2(-1.0);
//Matrice<double,10,10> M2(1.0);
```

Specializations compiler generated
- class **Matrice<int,2,3>**
- class **Matrice<double,10,10>**

Class templates without dynamic memory allocation

```
class Matrice<int,2,3>
{
   public:
      Matrice<int,2,3>(int init_val);
       …
      ostream& display(ostream& os);

    private:
       int table[2][3];
};
```

Name of the class generated by the compiler

```
Matrice<int,2,3> M1(0);
```

Specialization compiler generated
- class **Matrice<int,2,3>**

Class Templates
without dynamic memory allocation

```
class Matrice<double,10,10>
{
   public:
      Matrice<double,10,10>(double init_val);
       …
      ostream& display(ostream& os);
    private:
       double table[10][10];
};
```

Name of the class generated by the compiler

```
Matrice<double,10,10> M2(1.0);
```

Specialization compiler generated
- class **`Matrice<double,10,10>`**

Disadvantage: code bloating!!

Class Templates

- Class template definition (interface) and implementation of the class member functions (templates) need to be in the same file
  - Header file (**`.h`**) should also contain member functions implementation
  - Compiler needeeds to see all code that describes the class to be able to create a class specialization

Read sec. 14.1.1

Standard Template Library

- Ready to use program components
- **Container classes**
  - Classes to represent collections of data (data structures)
  - e.g. lists, sets
- **Algorithms**
  - Functions that perform operations over the container classes
  - e.g. copy, sort, count, find
- **Iterators**
  - *Similar to pointers* for the objects stored in a container class
  - Used often as arguments of algorithms
- The implementation of **STL** relies on templates
  - Template syntax is used

Containers

- **Sequence containers**
  - Linear data structures
    - Dynamic array `#include <vector>`
    - Linear list `#include <list>`
- **Associative containers**
  - Store key/value pairs
  - Retrieve a value given its key

Map Multimap	`#include <map>`
Set Multiset	`#include <set>`

- Unordered associative containers
  - Available only for the new `C++11` standard
  - hash tables ☺ -- TND004, Data structures course

Problems with Arrays

```
const int SIZE = 100;
int V1[SIZE], V2[SIZE];
```

1. Arrays assignment not allowed ~~`V1 = V2;`~~
2. Comparison of arrays not allowed ~~`V1 == V2`~~
3. Array size cannot be changed
4. No size is associated with an array
   - Access outside array boundaries is possible

- Class **`vector`** tackles these problems
  - Part of the standard library `#include <vector>`
  - Many useful functions
  - Syntax a *bit* awkward -- based on templates
  - You are expected to study class **`<vector>`** Read sec. 2.8

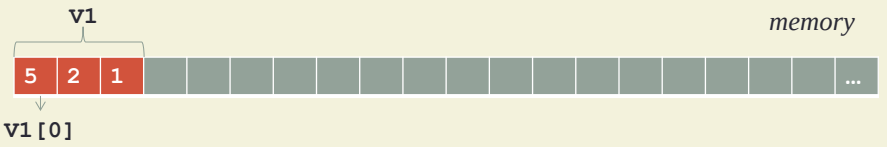
Vectors: declaration and initialization

```
#include <vector>
vector<char> V;                    //vector of size zero
vector<int> V1(3);                 //V1 = {0, 0, 0}
vector<double> V2 = {1.1, 2.2, 3.3};
vector<int> V3(4, 2);              //V3 = {2, 2, 2, 2}
vector<double> V4(V2);             //V4 is a copy of V2

//default constructor Clock::Clock() called for each slot
vector<Clock> K(4);

V4[0] = 3.14; //indexing
```

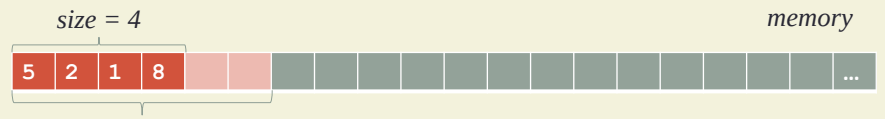
```
int a[10] = {1, 2, 3, …, 10};
vector<int> V5(a, a+10);
```

Initializes a vector from an array

```
vector <int> V1(3) = {5, 2, 1};
```

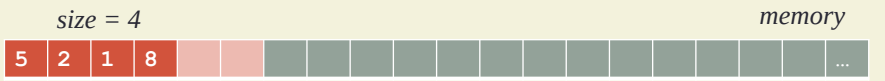



```
V1.push_back(8);  //add a value to the end of V1
```



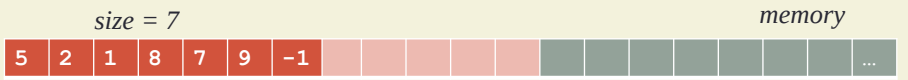

```
V1.pop_back();
```

```
for(int i = 0; i < V1.size(); i++)
  cout << V1[i];

cout << "V1 capacity: " << V1.capacity() << endl;
```

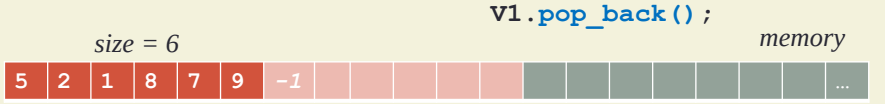




```
V1.push_back(7);
V1.push_back(9);
```


```
V1.push_back(-1);
```




```
V1.pop_back();
```



push_back(…)
- changes the size
- may change the capacity

pop_back(…)
- changes the size

```
vector<int> V1;
```

size = 0

capacity = 0

```
V1[0] = 5;
V1[1] = 5;
V1[2] = 5;
```

- NOT correct!
- Access out of **V1** bounds
- Neither size nor capacity increase

```
V1.push_back(5);
V1.push_back(5);
V1.push_back(5);
//V1.resize(3,5);
```

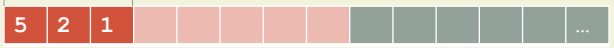
- Correct!

size = 3 *memory*

capacity = 4

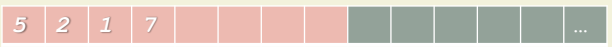
size = 3

V1 *memory*




```
V1.clear();
```

size = 0 *memory*




```
if ( V1.empty() )
  cout << "Vector is empty!!";

V1[3] = 66; //out of the vector bounds access

V1.at(3) = 66;
```

Signals if the requested position is ≥ **V1.size()** by throwing an **out_of_range** exception

Vectors: assignment and comparison

See **Clock** class + **test_clock.cpp**

```
#include <vector>
vector<Clock> V1(4);
vector<Clock> V2(3);
V1 = V2;     //V2 becomes a copy of V1
if (V1 == V2)
  cout << "V1 and V2 are equal" << endl;
else if (V1 < V2)
      cout << "V1 is smaller than V2" << endl;
```

Clock::operator< is used to compare each pair of **Clock**s

Vectors: 2-Dimensional Tables

	Jan	Feb	Mar	April	Maj	Juni	Juli	Aug	Sept	Okt	Nov	Dec
Min	11.5	12.0	12.9	13.5	17.0	20.5	22.5	25.0	21.5	18.0	13.5	12.3
Max	19.0	20.5	20.5	23.0	26.6	27.0	28.7	32.5	28.0	27.0	22.0	20.0
Avg	15.0	15.5	17.0	19.0	22.0	23.1	25.0	28.3	24.1	22.0	26.6	14.5
Vatten	17.0	17.5	20.5	21.0	22.0	23.5	25.0	26.0	24.5	22.3	21.0	17.5

```
//double table[4][12];
vector<vector<double>> table(4); //vector of vectors
```

table (( ), ( ), ( ), ( ))

table[0] is an empty vector

See **vector_tables.cpp**

Vectors: 2-Dimensional Tables

	Jan	Feb	Mar	April	Maj	Juni	Juli	Aug	Sept	Okt	Nov	Dec
Min	11.5	12.0	12.9	13.5	17.0	20.5	22.5	25.0	21.5	18.0	13.5	12.3
Max	19.0	20.5	20.5	23.0	26.6	27.0	28.7	32.5	28.0	27.0	22.0	20.0
Avg	15.0	15.5	17.0	19.0	22.0	23.1	25.0	28.3	24.1	22.0	26.6	14.5
Vatten	17.0	17.5	20.5	21.0	22.0	23.5	25.0	26.0	24.5	22.3	21.0	17.5

```
//double table[4][12];
vector<vector<double>> table(4); //vector of vectors

//Create lines of 12 columns initialized with 0
    for(int line = 0; i < table.size(); line++)
        table[line].resize(12);
```

table ((0,…,0), (0,…,0), (0,…,0), (0,…,0))

table[0] is a vector of 12 slots

Exercise

1. Add to the program in **vector_tables.cpp** a function that creates a table and stores the values given by the user
   - each line in the file corresponds to one line in the table
   - Use input redirection to read from a file

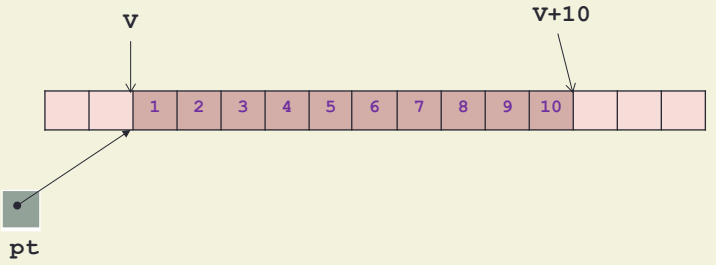
Number_of_lines	3
Values for line 1	4 2.2 6
Values for line 2	2.2 3.3 4.4 5.5
Values for line 3	-1 0
…	1 2 3.5 6 7 0

2. Solve exercise 21 of course book (pag. 66)
3. Create a class **Matrice** using **vector**s, instead of using dynamic memory allocation

Iterators

```
int V[10] = {1, 2, 3, …, 10};
for(int* pt = V; pt != V+10; pt++)
        cout << *pt << endl;
```

Looping through an array using a pointer and pointer arithmetic



Iterators

```
int V[10] = {1, 2, 3, …, 10};
for(int* pt = V; pt != V+10; ++pt)
        cout << *pt << endl;
```

Similar to a pointer to walk through the elements stored in the vector

Returns an iterator that points to the first element of the vector

Returns an iterator that points to an element **after** the last

```
vector<int> V = {1, 2, 3, …, 10};
for(vector<int>::iterator it = V.begin(); it != V.end(); ++it)
        cout << *it << endl;
```

Access the element *pointed* by the iterator

Increment the iterator

Iterators with different functionality

- Random-access iterators

```
it++      ++it   --it    it--
it+n      it-n
it[n]     *it    it->
```

  - pointers are random-access iterators

Read sec. 12.1

- Bidirectional operators

```
it++      ++it   --it  it--
*it       it->
```

- Forward iterators

```
it++      ++it         *it          it->
```

- Input iterators

```
… = *it;
it++
++it
```

Output iterators

```
*it = …;
it++
++it
```


Iterators and containers

- Different containers support different types of iterators

```
vector<int> V = {1, 2, 3, …, 10};
vector<int>::iterator it = V.begin();
```

Returns a random-access iterator
vectors support random-access iterators

```
cout << *(it+2);
```

```
list<int> L = {1, 2, 3};
list<int>::iterator it = L.begin();
```

Returns a bidirectional iterator
lists support bidirectional iterators

```
cout << *(it+2);
cout << ++(++it);
```

Reverse Iterators

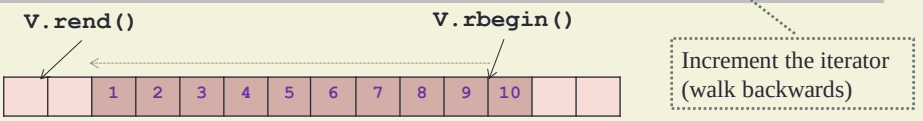
See **vectors_it.cpp**

Similar to a pointer to walk backwards through the elements stored in the vector

Returns an iterator that points to the last element of the vector

Returns an iterator that points to an element **before** the first

```
vector<int> V = {1, 2, 3, …, 10};
for(vector<int>::reverse_iterator it = V.rbegin();
                                  it != V.rend(); ++it)
        cout << *it << endl;
```



Increment the iterator (walk backwards)

const Iterators

```
int V[10] = {1, 2, 3, …, 10};
for(const int* pt = V; pt != V+10; ++pt)
        cout << *pt << endl;
```

```
vector<int> V = {1, 2, 3, …, 10};
for( vector<int>::const_iterator it = V.begin(); it != V.end(); ++it )
        cout << *it << endl;
        *it = 8;
```

Iterators

- Why do we need iterators?
  - Many **vector** member functions and algorithms use iterators as parameters
  - **vector::insert** and **vector::erase**
  - **http://www.cplusplus.com/reference/stl/vector/**

Read sec. 12.1.2

```
vector<int> V1 = {1, 2, 3, …, 10};
vector<int> V2 = {11, 12, 16};
```

```
V2.insert(V2.begin(), 3, 8);
//V2 = {8, 8, 8, 11, 2, 16}
```

V.insert(iterator, repeat, value)

```
V2.insert(V2.begin(), V1.rbegin(), V1.rend());
//V2 = {10, 9, 8, …, 1, 11, 2, 16}
```

```
V2.erase(++V2.begin());
//V2 = {10, 8, …, 1, 11, 2, 16}
```

V.erase(iterator)

Algorithms

```
#include <algorithm>
#include <numeric>
```

Four algorithms specifically designed to operate on numeric sequences

- Copying
- Searching
- Replacing and removing elements
- Reordering a sequence
- Sorting
- Sorted Sequence Searching
- Merging sorted sequences
- Minimum and maximum
- …

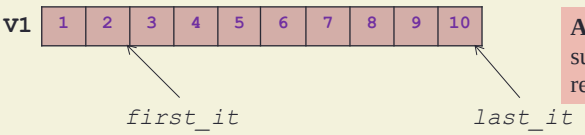
It's often possible to replace hand-written loops by calling an algorithm function

See appendix C in the end of the course book and online library

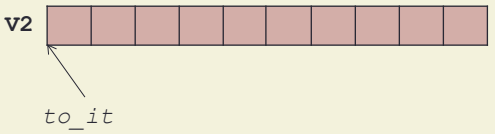
Algorithms: general idea

```
#include <algorithm>
copy(first_it, last_it, to_it);
```

Algorithms are not member functions



Attention: check that the container supports the type of iterators required by the algorithm




```
vector<int> V1 = {1, 2, …, 10}
vector<int> V2(10);
copy(V1.begin()+2, V1.end(), V2.begin());
```


Algorithms

```
vector<int>::iterator ptr;
ptr = find(V2.begin(), V2.end(), 10);
if (ptr != V2.end())
    cout << *ptr << endl;
```

Non-successful search

- iterator pointing to the slot after the last is returned

```
//find first even number
ptr = find_if(V2.begin(), V2.end(), even);
if (ptr != V2.end())
    cout << *ptr << endl;
//find second even number
ptr = find_if(++ptr, V2.end(), even);
if (ptr != V2.end())
    cout << *ptr << endl;
```

Algorithms

```
vector<int> V1 = {1, 3, 2, 8};
vector<int> V2 = {11, 2, 10, 7, 9};
vector<int>::iterator ptr;
ptr = find(V2.begin(), V2.end(), 10);
copy(V2.begin(), ptr, ++V1.begin());
//V1 = {1, 11, 2, 8}
sort(V1.begin(), V1.end());
//V1 = {1, 2, 8, 11}
```

```
int xx[10] = {1, 3, 2, 4, 5, 6, 7, 7, 7, 7};
vector<int> V3(10);
replace_copy(xx, xx+10, V3.begin(), 7, 0);
```

Read sec. 12.2

Exercise

- Write a program that reads several times (two **ints**, $0 \leq$ **hh** ≤ 23 and $0 \leq$ **mm** ≤ 59**)** and then displays
  - All morning times (AM), sorted incresingly
    - Use the format **hh:mm:00AM**
  - All afternoon times (PM), sorted increasingly
    - Use the format **hh:mm:00PM**
- Times are stored first in a **vector** (**times**) of **Clock** objects
  - All morning times copied to **vector times_AM**
  - All afternoon times copied to **vector times_PM**
  - Algorithms used: **copy_if**, **sort**, **transform**, and **for_each**
  - See **test_clock_algorithms.cpp**
  - **copy_if** only in **C++11**

Next …

- **Fö 12**
  - **S**tandard **T**emplate **L**ibrary (**STL**) (*cont.*)
    - Pairs [sec. 5.11]
    - Iterators and streams [sec. 12.1.3]
    - Iterators and **`string`**s
    - Standard class **`list`** [sec. 12.4]
    - Standard class **`map`** and **`multimap`** [sec. 12.5.1]
    - Standard class **`set`** and **`multiset`** [sec. 12.5.2]
- Lab 4 -- about STL