Lecture 13

- **S**tandard **T**emplate **L**ibrary (**STL**) [12]
  - Standard class **`vector`** [2.8, 12.1.2]
  - Standard class **`list`** [12.4]
  - **`string`**s and iterators
  - Algorithms [12.2, 12.3.1, appendix C]
  - Standard class **`pair`** *(par)* [5.11]
  - Standard class **`map`** and **`multimap`** *(avbildningar)* [12.5.1]
  - Standard class **`set`** and **`multiset`** *(mängder)* [12.5.2]
  - Iterators and streams [12.1.3]
  - Numerical algorithms [appendix C.9]

Info

- Dugga 3 -- posted through Lisam
  - Friday, 12:00 – Monday, 12:00
  - Deliver one source file (**`.cpp`**) for each exercise
  - Covers Fö 10 - Fö 12, lessons 3&4, lab 3
    - **inheritance**, virtual functions, dynamic binding, abstract classes, STL
    - Note: I am not saying that you need to used all these concepts, just that possibility exists
- Labs
  - 18 persons have not yet presented lab 1
    - Sessions of last week (week 49) had very low attendance
    - There were sessions with zero students
    - Reminder: at most two labs can be presented in one lab session
    - Late labs are not prioratized

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>`

Iterators with different functionality

Read sec. 12.1

- Random-access iterators

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

  - pointers are random-access iterators

- Bidirectional operators

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

- Forward iterators

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

- Input iterators

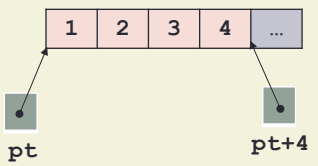
```
… = *it;
it++
++it
```

Output iterators

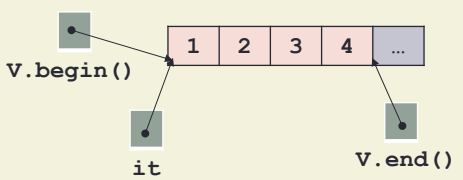
```
*it = …;
it++
++it
```


Iterators

```
int V[4] = {1, 2, 3, 4};
for(int* pt = V; pt != V+4; pt++)
        cout << *pt << endl;
```




```
vector<int> V = {1, 2, 3, 4};
for(vector<int>::iterator it = V.begin(); it != V.end(); ++it)
        cout << *it << endl;
```



random-access iterators

`vector`'s member functions

`vector::push_back` `vector::pop_back`	`vector::insert`	`vector::erase`

```
vector<int> V2;

v2.push_back(11);
v2.push_back(12);
v2.push_back(3);
v2.pop_back();
//v2 = {11, 12}
```

Empty vector, i.e. size = capacity = 0

Insert in the end of the vector -- size increases
- May imply new memory allocation

```
V2.erase(++V2.begin());
//V2 = {11, 12}
```

V.erase(iterator);

```
vector<int> V1 = {4, 5, 6};
V2.insert(V2.begin(), V1.rbegin(), V1.rend());
//V2 = {6, 5, 4, 11}
```

V.insert(iterator, iterator_from, iterator_to);

Insert **before** the position indicated by the *iterator* -- implies shifting elements

Algorithms

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

Five 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
- …

- Algorithms are not member functions of any container class
- Can operator over different types of containers
- Have often iterators as arguments
- Always check that the type of iterator required by the algorithm is also supported by the container

See appendix **C** of the course book

http://www.cplusplus.com/reference/algorithm/

Exercise

- Write a program that reads a sequence of user given integers and then displays the values sorted increasingly and without repeations
  - **`Example1.cpp:`** uses a **`vector`** to store the user values
  - **`Example2.cpp:`** uses a **`list`** to store the user values
  - **`Example3.cpp:`** uses a **`set`** to store the user values

Algorithms

```
vector<int> V1 = {11, 5, 10, 7, 9};
vector<int>::iterator itr;
itr = find( V1.begin(), V1.end(), 10);
if ( itr != V1.end() )
    cout << *ptr << endl;
```

Search interval

Value to search for

Non-successful search

- iterator to the element after the last is returned

```
vector<int> V2 = {1, 8, 3};
copy(V1.begin(), itr, ++V2.begin());
//V2 = {1, 11, 5, 8, 3}
sort(V2.begin(), V2.end());
//V2 = {1, 3, 5, 8, 11}
```

copy(iterator, iterator_from, iterator_to);

See class **`Clock`** and **`test_clock_algorithms.cpp`**

Algorithms

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

Function pointer

```
for(itr = V2.begin(); itr != V2.end(); itr++)
      if ( even(*itr) )   break;
```

Read sec. 12.3.1

Algorithms

```
vector<int> V1 = {11, 2, 10, 7, 9};
replace_if(V1.begin(), V1.end(), even, -1);
//V1 = {11, -1, -1, 7, 9}
```

Pairwise comparison
abs_equal(V1[i],V2[i])

```
vector<int> V2 = {-11, -2, -10, -7, -9};
if ( equal(V1.begin(), V1.end(), V2.begin(), abs_equal) )
    cout << "V1 == -1*V2" << endl;
else
    cout << "V1 != -1*V2" << endl;
```

equal(it_first1, it_last1, it_first2, test);

```
bool abs_equal(int a, int b)
{ return ( fabs(a) == fabs(b) ); }
```

Algorithms

```
vector<int> V1 = {11, 2, 10, 7, 9};
sort(V1.begin(), V1.end(), larger_than);
//V1 = {11, 10, 9, 7, 2}
```

Pointer to a comparison function to be used for sorting

```
bool larger_than(int a, int b)
{ return (a > b); }
```

Apply function **display** to every element in the interval

```
for_each(V1.begin(), V1.end()-1, display);
```

```
for(ptr = V1.begin(); ptr != V1.end()-1; ptr++)
      display(*ptr);
```

```
void display(int a)
{ cout << a << " ";  }
```


Standard class `list`

```
#include <list>
list<int> L1;
```

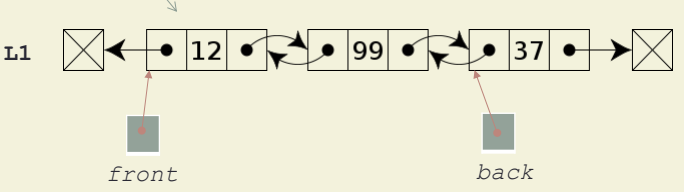
Empty list

```
L1.push_back(12);
L1.push_back(99);
L1.push_back(37);
```

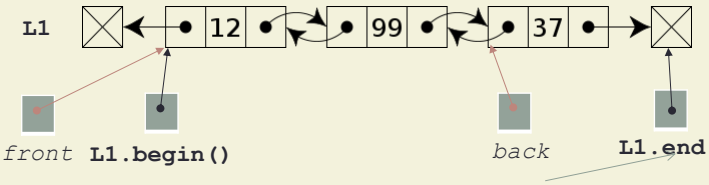
Insert at the back

```
L1.push_front(-1);
L1.push_front(-2);
```

Insert at the front



Standard class `list`




```
L1.begin();
L1.end();
```

Bidirectional iterators

Attention do not dereference

```
list<int>::iterator it;
it = L1.end();
cout << *it;
```

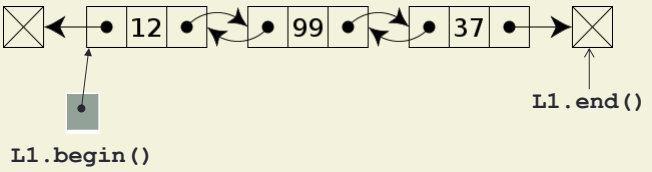
~~`sort(L1.begin(), L1.end());`~~

`sort` algorithm requires random-access iterators

```
cout << "First: " << L1.front(); //12
cout << "Last: " << L1.back();   //37
```

References to elements in the list, not iterators

Standard class `list`



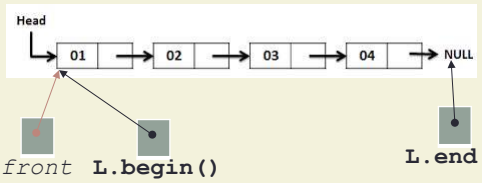
~~`sort(L1.begin(), L1.end());`~~

```
L1.sort();
list<int>::iterator it = L1.begin();
while ( it != L1.end() )
{
    cout << *it << " ";
    it++;
}
```

`List::sort()`

For more `List` member functions see table in pag. 467

forward_list (C++11)




```
L.begin();
L.end();
```

Forward iterators

```
#include <forward_list>
forward_list<int> L = {2, 3, 4};
L.push_front(1);
L.front() = -5;
```

Return a reference to the first element in **L**

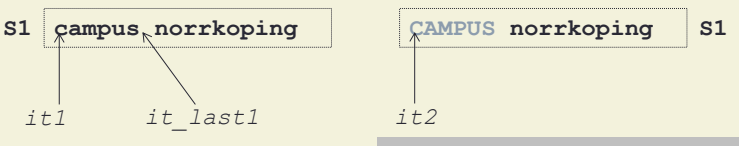
```
cout << "First: " << L1.front(); //-5
```


iterators and strings

- A **string** object can be seen as a container
  - Iterators can be used, then -- ☺ possible to use algorithms for **string**

string::begin()	string::end()
string::rbegin()	string::rend()

Random access iterators




```
transform(it1, it_last1, it2, func);
```

```
string::iterator it1, it_last1;
string::iterator it2;
it1 = S1.begin();
it2 = S1.begin();
it_last1 = it1 + 6;
```

```
while ( it1 != it_last1 )
{
   *it2 = func(*it1);
   it1++; it2++;
}
```

iterators and **strings**

```
#include <cctype>
char to_upper(char c)
{ return toupper(c); }
```

Transform all characters of **S** to upper case letters

```
string S = "campus norrkoping";
transform(S.begin(), S.end(), S.begin(), to_upper);
```

Transform first 3 characters of **S** to upper case letters

```
string S = "norrkoping";
string::iterator it3 = S.begin()+3;
transform(S.begin(), it3, S.begin(), to_upper);
```

member type of class **string**

Iterators and streams

- Iterators can be used to read and write from streams

```
#include <iterator>
istream_iterator<T> in_it(cin);
```

Type of elements in the buffer of the stream

Input iterator points to the next element of type *T* in the buffer

Stream from where to read (**cin**)

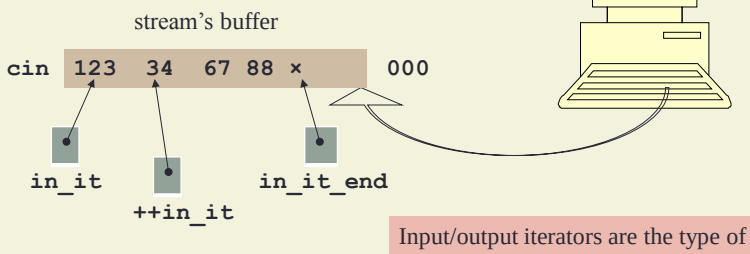
```
istream_iterator<T> in_it_end;
```

If no stream is given the the iterator points to the *end of the stream*

Iterators and streams

```
#include <iterator>
istream_iterator<int> in_it(cin);
```

```
istream_iterator<int> in_it_end;
```



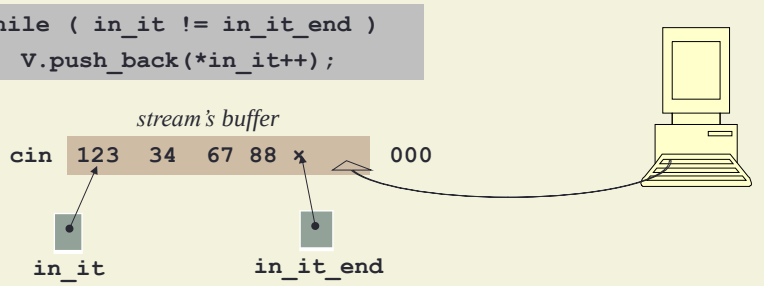
Input/output iterators are the type of iterators that support less operations

Iterators and streams: reading

Read the **int**s typed by the user and store them in a vector

```
istream_iterator<int> in_it(cin), in_it_end;
vector<int> V; //empty vector
```

```
while ( in_it != in_it_end )
   V.push_back(*in_it++);
```




```
copy(it, it_end, back_inserter(V));
```

Output iterator that inserts new elements automatically at the end of the container

Iterators and streams: writting

```
ostream_iterator<int> out_it(cout, " ");
vector<int> V = {2, 3, 5, 7, 9};
copy(V.begin(), V.end(), out_it);
```

Delimiter character: which char to write between every element

```
ostream_iterator<Clock> out_it(cout, "\n");
vector<Clock> V = {Clock(8,30), Clock(10,0), Clock(10,30)};
copy(V.begin(), V.end(), out_it);
```

Needed **operator<<(…)** for class **Clock**

See **exercise5.cpp**

Standard class **pair**

```
#include <utility>
pair<string, int> p1 ("Maria", 17);
```

A person's name

A person's age

```
typedef pair<string, int> Person_type;
Person_type p2("Erik", 25); //see also make_pair
cout << "Name: " << p2.first << " ";
cout << "Age: " << p2.second << endl;
```

```
p1 = p2;
if ( p1 == p2 ) …;
```

Assignment operator and comparison **operator==** are defined for class **pair<…>**

```
typedef pair<string, string> Lab_group;
vector<Lab_group> MT2;
```

Read sec. 5.11

Standard class **map**

- Table of pairs **<key,value>**
  - Sorted by **key**
  - Pairs with repeated keys NOT allowed

Phone Extensions

Name	Phone
Erik	1234
Lisa	3399
…	…

```
#include <map>
map<string,int> Phone_ext;
```

Empty table

key value

```
Phone_ext["Erik"] = 1234;
Phone_ext["Lisa"] = 3399;
```

Pair **<"Erik",1234>** is inserted in the table

```
Phone_ext["Erik"] = 1323;
```

Change the value associated to the key "**Erik**"
- **<"Erik",1323>**

Read sec. 12.5.1

Standard class **map**

```
cout << Phone_ext["Erik"] << endl;
cout << Phone_ext["Mike"] << endl;
```

Phone Extensions

Name	Phone
Erik	1234
Lisa	3399
Mike	*0*

Getting the value of a key not existing in the table implies that a new pair is inserted
- **<"Mike", 0>**

```
map<string,int>::iterator it;
it =  Phone_ext.find("Mike");
if ( it != Phone_ext.end() )
    cout << it->second << endl;
else
    cout << "Mike is not in the table";
```

Returns a bidirectional iterator that points to the pair with key "**Mike**"

Test if a pair with key "**Mike**" was found

Standard class **map**

```
#include <map>
#include "Clock.h"
typedef map<Clock, string> Schema;
Schema table;
```

Time	Destiny
08:30	Uppsala
09:00	Stockholm
…	…

Table is sorted by **key** = **Clock**
Need **Clock::operator<(…)**

```
table[ Clock(9,0) ] = "Stockholm";
table[ Clock(8,30) ] = "Uppsala";
```

operator<< for class **Clock**

```
Schema::iterator it;
cout << "Time" << "    " << "Destiny" << endl;
for(it = table.begin(); it != table.end(); it++)
       cout << it->first << "    " << it->second << endl;
```


Problem

- What if we need to display the schedule sorted by **Destiny**
  - **map** can only be sorted by the key (first element of each pair)

Time	Destiny
09:00	Stockholm
08:30	Uppsala
…	…

- Solution
  1. Copy the pairs to a vector of pairs `<Clock, Destiny>`
  2. Sort the vector by `Destiny`

Useful for **Lab 4**

See **time_table.cpp**
Class **Clock** is needed

Standard class **map**

```
typedef pair<Clock, string> Table_Entry;
vector<Table_Entry> vec_table(table.size());
```

1. Copy the pairs in the **map** into a **vecto**r

```
copy(table.begin(), table.end(), vec_table.begin());
```

2. Sort the vector by destiny

```
sort(vec_table.begin(), vec_table.end(), compare);
```

Sorting criteria

```
bool compare(const Table_Entry &e1, const Table_Entry &e2)
{ return (e1.second <= e2.second); }
```


Standard class **map**

What if we need to have pairs with repeated keys?

Phone Extensions

Name	Phone
Erik	1234
Erik	1323
Lisa	3399
…	…

Solution 1: **map< string, vector<int> >**

```
typedef vector<int> Extensions;
map<string, Extensions> Phone_ext;
```

is a vector

```
Phone_ext["Erik"].push_back(1234);
Phone_ext["Erik"].push_back(1323);
Phone_ext["Lisa"].push_back(3399);
```

Phone_ext

Name	Phone
Erik	<1234,1323>
Lisa	<3399>
…	…

Standard class **multimap**

- Table of pairs **<key,value>**
  - Sorted by **key**
  - Pairs with repeated keys are allowed

Phone Extensions

Name	Phone
Erik	1234
Erik	1323
Lisa	3399
…	…

Solution 2: **multimap<string, int>**

```
multimap<string, int> Phone_ext;
```

Cannot use indexing **operator[]** with multimap

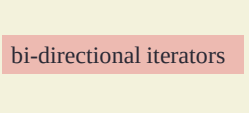
```
Phone_ext.insert(make_pair("Erik", 1234));
Phone_ext.insert(make_pair("Erik", 1323));
Phone_ext.insert(make_pair("Lisa", 3399));
```


Standard class **multimap**

Phone Extensions

Name	Phone
Anna	1010
Erik	1234
Erik	1323
Lisa	3399
…	…

```
multimap<string, int> Phone_ext;
```



bi-directional iterators

Display the extension numbers for Erik

```
multimap(string, int>::iterator it1, it2;
it1 = Phone_ext.lower_bound("Erik");
it2 = Phone_ext.upper_bound("Erik");
while (it1 != it2 ) {
       cout << it1->second << " ";
       it1++;
}
```

See table in pag. 471, 472 of course book

Standard class **set** and **multiset**

```
#include <set>
set<int> s1 = {2, 4, 6};
```

- Sets are sorted
- Cannot have repeated elements

```
s1.insert(-1); //s1 = {-1, 2, 4, 6};
s1.erase(6); //s1 = {-1, 2, 4};
cout << s1.size(); //number of elements
if ( s1.empty() ) //test if set is empty
  …;
```

For more **set** member functions see table in pag. 480

```
multiset<int> s2 = {2, 4, 6, 4};
s2.insert(4);
```

- Multisets are sorted
- Can have repeated elements

Read sec. 12.5.2

Homework

- Study the following examples
  - **TimeTable.cpp** -- create a project with class **Clock**
  - **stream_iterators.cpp** -- create another project with class **Clock**

Next …

- Lesson 4
  - Read exercises
  - Attempt the exercises
    - little code to write, more focus on the understanding and correct use of concepts
- Start Lab 4 -- about **STL**
- Fo 14
  - Final exam
  - Course ending