LECTURE 4

A program using `vector`'s

```cpp
#include <iostream>
#include <vector>
#include <iterator> // for ostream_iterator
#include <algorithm> // for copy
#include <functional> // for greater
#include "MyRand.hpp" // for NormalDist
using namespace std;


void containerExample(){
  vector<double> dVec(5);
  NormalDist(dVec);

  copy(dVec.begin(),dVec.end(),ostream_iterator<double>(cout," "));
  cout<< endl;

  //removing negative entries
  vector<double>::iterator iter=dVec.begin();
  while(iter!=dVec.end()){
    if((*iter)<0)
      iter=dVec.erase(iter);
    else
      ++iter;
  }

  copy(dVec.begin(),dVec.end(),ostream_iterator<double>(cout," "));
  cout<< endl;
}
```

A program using `vector`'s

Output:

```
    1.58883 1.14627 -1.69412 0.295142 -0.895352
    1.58883 1.14627 0.295142
```

```cpp
#include <iostream>
#include <vector>
#include <iterator> // for ostream_iterator
#include <algorithm> // for copy
#include <functional> // for greater
#include "MyRand.hpp" // for NormalDist
```

- `<vector>` contains the implementation of the `vector` container
- `<iterator>` contains the implementation of the `ostream_iterator` container
- `<algorithm>` contains `stl` algorithms acting on containers
- `<functional>` contains `std::less<>` and `std::greater<>`,
- `"MyRand.hpp"` is a header file in which the function `NormalDist` is implemented
- `"MyRand.hpp"` is not part of the standard library

A program using `vector`'s

```
1  vector<double> dVec(5);
2  NormalDist(dVec);
3
4  copy(dVec.begin(),dVec.end(),ostream_iterator<double>(cout," "));
5  cout<< endl;
```

- The first line defines a `vector` containing five default constructed `double`'s
- The second line calls the function `NormalDist` taking the vector by reference and filling with standard normals
- The third line calls the `copy` algorithm (of `<algorithm>`).
- `copy` takes a range from a container determined by the beginning and the end of the range.
- and copies the elements of the range into the range determined by the third argument.
- Ranges are defined by **iterators** which are pointer-like addresses of container entries (see later).

```
1  vector<double>::iterator iter=dVec.begin();
2  while(iter!=dVec.end()){
3    if((*iter)<0)
4      iter=dVec.erase(iter);
5    else
6      ++iter;
7  }
```

- The first line defines an iterator pointing to the first element of the vector `dVec`
- The while loops until `iter` does not point to the element after the last (to where `push_back(...)` would insert the next element)
- In the `if` condition statement we access the value to which `iter` points to using the dereferencing operator `*`, similarly to the use of pointers.
- If the value is negative, we call the `erase(...)` **member function** of `dVec`, which removes the element and returns the address of the next element in the vector.
- If the value is non-negative, the iterator is incremented using the operator ++. The new iterator points to the next element in the vector.

Introduction to the standard template library

The standard library offers containers and algorithms.

- **containers** are (templated) classes holding values of certain types
- **sequential containers**: ordered collections in which every element has a certain position not necessarily depending on the value of the element (but might depending on the time and place of insertion) - `vector`, `deque`, `list`
- **associative containers**: sorted collections in which the actual position of an element depends on its value due to a certain sorting condition - `set`, `multiset`, `map`, `multimap`
- containers have member functions (e.g. `size()`, `push_back(...)`, `erase(...)`, `remove(...)`)
- **iterators** (e.g. `container_type::iterator`) are pointer-like addresses pointing to elements of containers
- one can define sub-ranges of containers in terms of iterators
- **algorithms** are functions, usually taking a range or ranges of elements from containers and act on them
- algorithms can be non-modifying algorithms, but also there are algorithms modifying the number of elements in containers, the values of the entries, etc.
- examples: `for_each`, `transform`, `sort`, `remove`, `remove_if`, `copy`, `insert`, `merge`, `set_union`, etc. etc.

Requirements

The elements of stl containers **must** meet the following requirements

- Each element must be **copyable** by a copy constructor.
- Each element must be **assignable** by the assignment operator =.
- Each element must be **destroyable** by a destructor.

For certain member functions, elements might also have to meet the following requirements

- **default constructor** must be available
- operator == must be defined.
- for sorting algorithms and associative containers a sorting criterion must be provided. By default this is determined by the operator <.

Common properties of stl containers

Constructors and destructor

- `cont_type c` - constructs an empty container of type `cont_type`
- `cont_type c2(c1)` - constructs a container `c2` by copying the elements of `c1`, where `c1` has the same type `cont_type`
- `cont_type c(beg,end)` - constructs a container of type `cont_type` and inserts the element from the range determined by `beg, end`.
- `c.~cont_type()` - deletes all elements and free memory (note: if `c` contains pointers created by the operator `new`, the elements must be explicitly destroyed using `delete` before `c` is destroyed)

Basic description

- `c.size()` - returns the actual number of elements in `c`
- `c.empty()` - returns whether the container is empty (equivalent to `c.size()==0`, but usually is faster)
- `c.max_size()` - returns the maximum number of elements possible

Relational operators

- `c1==c2, c1!=c2`
- `c1>c2, c1>=c2` etc.

Common properties of stl containers

Basic operations

- `c1=c2` - assigns all elements of `c2` to `c1`
- `c.swap(c2)` - swaps data of `c1` and `c2`
- `c.insert(elem)` - inserts the element `elem` into `c`, if `c` is an associative container
- `c.insert(pos,elem)` - inserts the element `elem` to the position `pos` (sequential containers)
- `c.insert(pos,beg,end)` - inserts the range [`beg`, `end`) to the position `pos` (sequential container)
- `c.insert(beg,end)` - inserts the range [`beg`, `end`) into `c` (associative container)
- `c.erase(beg,end)` - removes all elements of the range `beg,end` from `c` (note: in case of some containers `erase` returns an iterator to the next element not removed)
- `c.clear()` - removes all elements from `c`

Common properties of stl containers

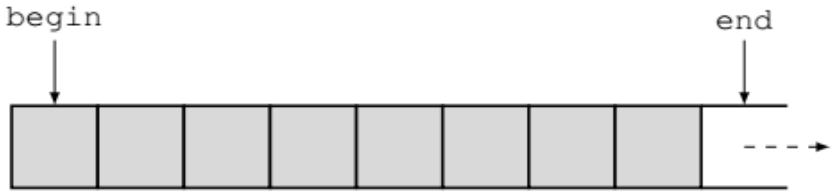
Operations returning iterators

- `c.begin()` - returns an iterator pointing to the first element in `c`
- `c.end()` - returns an iterator to the position after the last element
- `c.rbegin()` - returns an reverse iterator pointing to the first element of the reverse iteration
- `c.rend()` - returns an reverse iterator to the position after the last element of the reverse iteration

Iterator operations

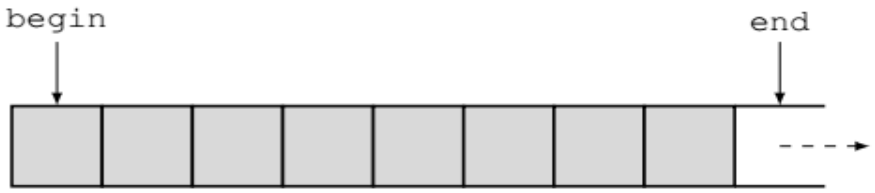
- `++iter`, `iter++` - pre- and post-increment of iterators (`--iter`, `iter--` exists for bidirectional iterators - see later)
- `*iter` - dereferencing iterators, i.e. accessing the value they point to
- `(*iter).mem`, `iter->mem` - accessing members of the value the iterator points to
- `distance(iter1,iter2)` - returns the distance (number of steps) of `iter1` and `iter2`
- `advance(iter,n)` - increments `iter` n-times (either by n-times `++iter` or `iter+n` if available)

Vectors - general properties



- `#include <vector>`
- Vectors are dynamic arrays and a kind of an ordered collection of elements.
- Vectors provide **random access**, elements can be accessed directly in constant time.
- Vectors provide good performance if elements are appended or deleted at the end.
- Inserting or deleting in the middle is not efficient because it might result in copying/moving lots of elements to new position.
- Memory allocation for new elements might take time, however the `reserve` member function can be used to allocate memory for potential elements.
- Note: `vector<bool>` does NOT behave as an stl container, use `deque<bool>` instead!

Vector operations



Construction - beyond the common constructors

- `vector<elem_type> c(n)` - creates a vector of n default constructed elements
- `vector<elem_type> c(n,elem)` - creates a vector of n copies of `elem`

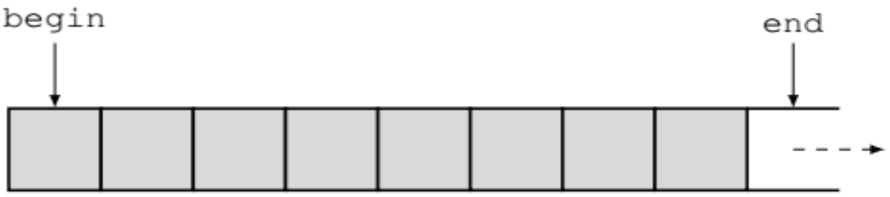
Non-modifying operations - beyond the common ones

- `c.capacity()` - returns the maximum possible number of elements without reallocation
- `c.reserve()` - enlarges the capacity if not enough yet

Assignments - beyond the common ones

- `c.assign(n,elem)`, `c.assign(beg,end)` - assigning elements (n copies of `elem` or elements in a range) to the vector
- `c1.swap(c2)`, `swap(c1,c2)` - member and global swap functions

Vector operations



Accessing elements

- `c.at(index)` - access element at `index` (with range checking)
- `c[index]` - same as above, but without range checking
- `c.front()`, `c.back()` - returns first/last element (without checking if it exists)

Inserting/removing elements - beyond the common ways

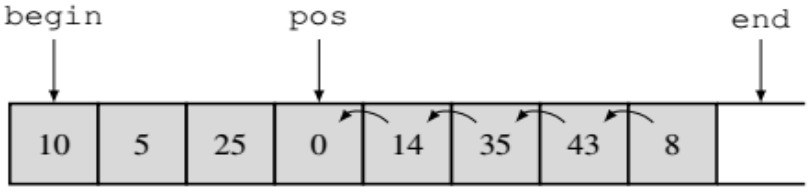
- `c.insert(pos,n,elem)` - inserts n copies of `elem` at position `pos`
- `c.push_back(elem)` - insert `elem` at the end
- `c.pop_back()` - removes element from the and
- `c.resize(n)` - changes the number of elements to n, new elements are created by their default constructor
- `c.resize(n,elem)` - changes the number of elements to n, new elements are copies of `elem`

Vector operations

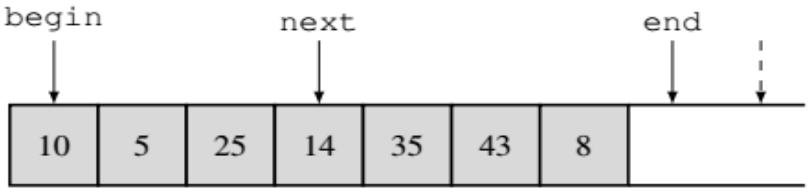
Inserting/removing elements - continued

- `c.erase(pos)` - removes element from position, returns position of the next element
- `c.erase(beg,end)` - removes elements from range [`beg`,`end`), returns position of the next element

Why is erase slow?

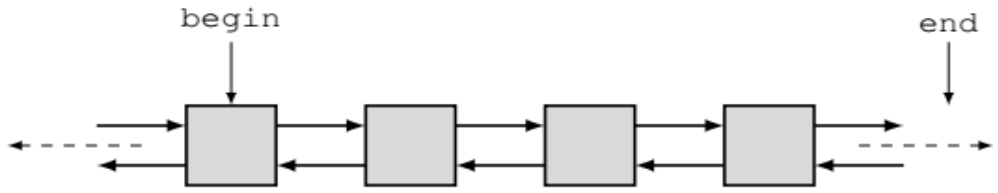


`c.erase(pos)` results in copying elements (using copy their copy constructor):



Deques and Lists

- `#include <deque>`
- Deques are dynamic arrays, similar to vectors, however open at both ends.
- Inserting and removing elements is fast both at the beginning and the end



- `#include <list>`
- Lists don't offer random access. To access the nth element, one must navigate through the first $n - 1$ elements, which might be slow.
- Inserting and removing elements at each position can be done in constant time without moving other elements.

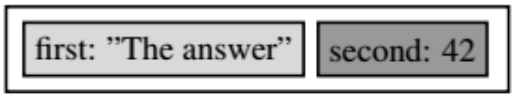
Summary

stl containers	stl algorithms
`vector`	`deque`
`list`	iterators

Associative containers

- Sets and multisets contain elements on which some sorting criterion is defined.
- Maps and multimaps contain pairs of elements: a key and a mapped value, such that some sorting criterion on keys is defined.
- Multisets and multimaps allow duplicates, sets and maps don't.
- Include `<set>` to access sets and multisets, include `<map>` to access maps and multimaps.
- The sorting criterion must have the following properties: antisymmetry, transitive, irreflexive, and equivalence transitive. Note `x` and `y` are defined to be equivalent if neither `x<y` nor `y<x` yield `true`.
- Sets, maps don't provide operations for direct access
- Sets and multisets are very efficient at finding elements with a certain value
- Maps and multimaps are very efficient at finding elements with a certain key
- The value of the elements in sets and multisets and the value of keys in maps and multimaps are not to be changed directly (would mess up the sorting)

Pairs - general properties



- `#include <utility>`
- An instance of the double templated `pair<type1,type2>` is a simple container holding an element of type `type1` and an element of type `type2`

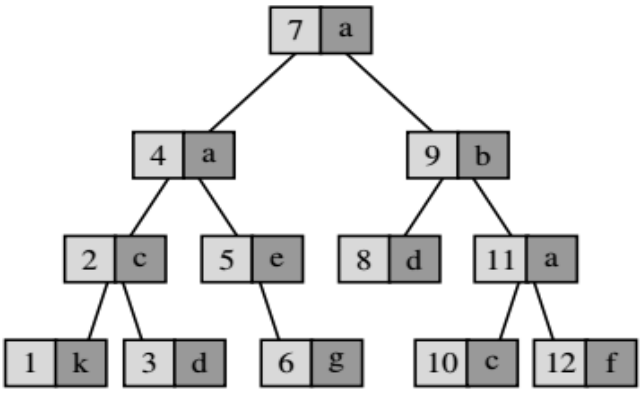
Members

- `pair<type1,type2>::first_type`, `pair<t1,t2>::second_type` are type definitions for the types `type1` and `type2` respectively
- `p.first` is a data member (note, no `()` is used), the contained element of type `type1`
- `p.second` is a data member, the contained element of type `type2`
- `pair<type1,type2> p(a,b)` constructs a pair assuming that `a` and `a` are of (or can be converted to) `type1` and `type2` respectively.

Non member but useful function

- `make_pair(a,b)` - returns a pair containing `a` and `b`.

Maps and multimaps - the typical internal structure



- Maps and multimaps are very efficient in finding elements by their key
- however maps and multimaps are not efficient in finding elements by their value (i.e. the second member of the contained pairs)
- The key is found using equivalence (`x` and `y` are equivalent if `!(x<y) && !(y<x)` is true).

Maps and multimaps - operations

Constructors/destructors - beyond the common ones

- `MAP c(op)` - creates an empty map/multimap of mapping elements of type `mapped_type` to the type `type_key` which uses `op` as sorting criterion
- `MAP c(beg,end,op)` - creates a map/multimap which uses `op` as sorting criterion initialized by the elements in the range [`beg`,`end`).
- where `MAP` can be `map<type_key, mapped_type>`, `map<type_key,mapped_type,type_op>`, `multimap<type_key,mapped_type>` or `multimap<type_key,mapped_type,type_op>`

Maps and multimaps - operations

Special members - optimized search operations

- `c.count(key)` - returns the number of elements in `c` with key `key`
- `c.find(key)` - returns the position of the first element with key `key` if exists, otherwise returns `c.end()`
- `c.lower_bound(key)` - returns the first position, where an element with key `key` would get inserted (fist element `>=elem`)
- `c.upper_bound(key)` - returns the last position where an element with key `key` would get inserted (the first element `>elem`)
- `equal_range(key)` - returns the range (in a `pair` of iterators, see next slide) where an element with key `key` would get inserted.
- Note: these members are more efficient than the corresponding stl algorithms.

Non-modifying operations, iterator functions, insertion, erase

- the common ones available

Associative containers - an example

```
#include <map>
#include <utility>
#include <string>

void mapExample(){
  typedef map<int,string,greater<int> > MyMap;
  MyMap mmExample;
  //inserting elements
  mmExample.insert(make_pair(1,"aaa"));
  pair<MyMap::iterator,bool> pIfDone =
    mmExample.insert(make_pair(1,"bbb"));
  if(!pIfDone.second)
    cout << "Insertion did not succeed" << endl;
```

- <map> is included for map
- <utility> is included for pair
- typedef defines a map type mapping strings to integers with the sorting rule greater, i.e. the keys will be in descending order
- note that insert() returns some information, including if the insertion was successful

Associative containers - an example

```
  mmExample.insert(make_pair(12,"aaa"));
  mmExample.insert(make_pair(3,"bbb"));
  mmExample.insert(MyMap::value_type(23,"ccc"));
  mmExample[10]="c"; //same outcome, but might be slower

  //iterating through a map
  MyMap::const_iterator citer = mmExample.begin();
  MyMap::const_iterator eiter = mmExample.end();
  for(; citer != eiter; ++citer)
    cout << "[" << (*citer).first << "]=\""
     << citer->second << "\"" << endl;
```

- insert() inserts pairs into the map, which can be created either by make_pair() or by the pair constructor (my_map::value_type is exactly the type of the contained pairs)
- the operator [] also can be used to insert elements, however it calls the default constructor first then the assignment. insert() calls the copy constructor only.
- One can iterate through the map incrementing iterators. Each iterator points to a pair, note the two ways of accessing the data in the pair.

Associative containers - an example

```
1  //changing keys: 1 to 100, indirectly!
2  MyMap::iterator iter = mmExample.find(1);
3  MyMap::mapped_type elem(iter->second);
4  mmExample.erase(iter);
5  mmExample.insert(make_pair(100,elem));
6  //checking if 1 is still in
7  iter=mmExample.find(1);
8  if(iter==mmExample.end())
9    cout << "1 is no longer in the map" << endl;
10 //checking if 100 is still in
11 iter=mmExample.find(100);
12 if(iter != mmExample.end())
13   cout << "100 is in the map, with value: \""
14    << iter->second << "\"" << endl;
15 }
```

- since modifying the key of an element would mess up the ordering (most compiler would not allow it anyway), we do the find-erase-reinsert trick
- Finally `find()` is applied to look up keys in the map.

```
Insertion did not succeed
[23]="ccc"
[12]="aaa"
[10]="c"
[3]="bbb"
[1]="aaa"
1 is no longer in the map
100 is in the map, with value: "aaa"
```

STL algorithm - overview 1

- STL algorithms work on ranges defined by iterators, the following categories are available
  - Non-modifying/modifying algorithms
  - Removing algorithms
  - Sorting algorithms
  - Algorithms on sorted ranges
  - Numeric algorithms

- In general, the `<algorithm>` header is required, for certain use of functors the `<functional>` header must be included, the numeric algorithms are defined in the `<numeric>` header
- The algorithms take iterators as arguments (and might take other types too), however they don't know the type of the containers these iterators refer to, i.e algorithms exploit the general properties of iterators, but no more.
- This implies that if an algorithm takes more than one range as argument, those ranges can refer to containers of different types.
- Some of the containers provide member functions, which are specialized and therefore more efficient than the general versions in `<algorithm>`.

STL algorithm - overview 2

- Algorithms usually operate on container elements, these operations are determined by functors or function objects.
- Functors can be pointers to functions, or classes where the functionality used by the algorithm must be provided by the member operator `()`.
- Function objects are usually more efficient than function pointers, furthermore they offer greater flexibility.
- Function object must be monomorphic (not derived from other classes) and must be cheap to construct (see chapter 6 in Effective STL by Scott Meyers for further requirements and details).
- Note that stl provides some built-in templated function objects, e.g. `plus<T>`, `minus<T>`, `multiplies<T>`, `divides<T>`, `less<T>`, `greater<T>` etc (see about templates on day 3).
- The `_if` suffix in the algorithms name is used when a certain criterion is determined by a function, i.e. `find()` searches for elements that has a certain value, whereas `find_if()` searches for elements that meets certain criterion, in which case the criterion functor must be passed as an argument.
- The `_copy` suffix indicates that the elements are not only manipulated but also copied into a destination range.

STL algorithm - `for_each()`

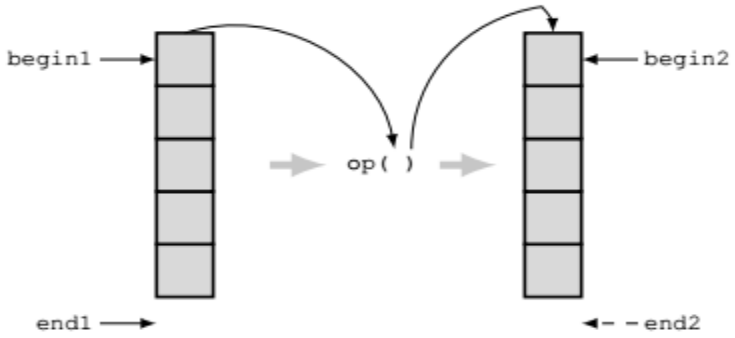
The algorithm `for_each` takes a range and an operation and executes the operation on each element in the range.

```
1  //functor
2  class add_value_byref
3  {
4  public:
5    add_value_byref(double dArg) : dValue_(dArg){}
6    void operator()(double & dArg){ dArg+=dValue_;}
7  private:
8    double dValue_;
9  };
10   //in the main
11   vector<double> Vec(5,3.0);
12   for_each(Vec.begin(),Vec.end(),add_value_byref(2.5));
```

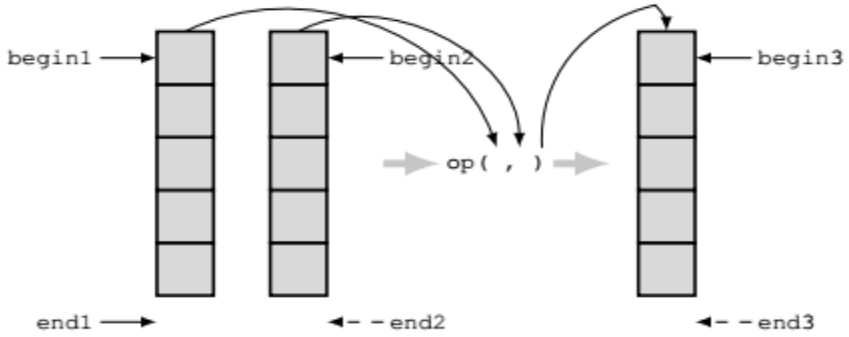
- In this example, the `for_each` algorithm calls the `operator()` member of the function object `add_value_byref`. (It's function object because it has an `operator` member.)
- The `operator()` member of the `add_value_byref` class takes a `double` by reference, and adds a value to the argument.
- The function object is constructed when passed to the `for_each` algorithm. The value to be added is determined by the argument of the constructor.

STL algorithm - `transform()`

`transform(begin1,end1,begin2,op)`



`transform(begin1,end1,begin2,begin3,op)`



STL algorithm - `transform()`

```
1  //functor
2  class add_value_byval : public unary_function<double,double>
3  {
4  public:
5    add_value_byval(double dArg) : dValue_(dArg){}
6    double operator()(double & dArg){return dArg+dValue_;}
7  private:
8    double dValue_;
9  };
10   //in the main
11   vector<double> Vec2(5);
12   Vec2[0]=2;
13   transform(Vec2.begin(),Vec2.end()-1,Vec2.begin()+1,
14        add_value_byval(2.5));
```

- The function object `add_value_byval` has an operator `()` taking a `double` by value and returning a `double`.
- The one-variable version of `transform()` is called; the input range is the elements of `Vec2` from the beginning to the penultimate element, and the output range is `Vec2` from the second element to the last.

Summary

`set, multiset` `map, multimap`

`pair`

`<algorithm>` `<numeric>`

`for_each()` `transform()`