Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: The standard library

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

The standard library (partI)
- What do we have in the Standard Library?
- A glance on containers
- Iterators
- Methods and types in a container

Ranges

pair and tuple

Some more details on contenitors
- Sets and multisets
- Maps and multimaps
- Unordered container
- Stack, queue, priority queues

Final considerations

Algorithms
- Types of algorithms

Inserters

What do we have in the Standard Library? (I)

- Containers Generic containers and iterator. Containers adapters to perform specific actions:(e.g stack<T>). Containers differ on the type and computational complexity of access to the contained elements.
- Utilities: Smart pointers (C++11). pair and tuple (C++11): fixed-size collection of heterogeneous values. Clocks and timers (C++11). Function wrappers (C++11) and predefined functors. Binders, Lambdas (C++11). The template class ratio for constant rationals (C++11).
- Support for internationalization: locale and wide_char.
- Algorithms They operate on ranges of values (usually stored in std containers) to perform specific actions like, sorting, transformations, copying etc.
- Strings and text processing. The class string (and derived classes). Regular expressions (C++11).

What do we have in the Standard Library? (II)

- I/O. The i/o streams and related utilities.
- Numerics complex<T>, numeric limits, random numbers and distributions (C++11), main math operators.
- Error and exception handling. Standard exception classes, support for handling floating point exceptions (C++11), exception pointers to store exception class objects (C++11).
- Support for generic programming. type_traits (C++11).
- Support for reference and move semantic. Reference wrappers, std::move() Tools for forwarding function arguments.
- Support for multithreading and concurrency. Threads, mutexes, locks etc.
- Allocator. They allow to change how objects are allocated in containers.

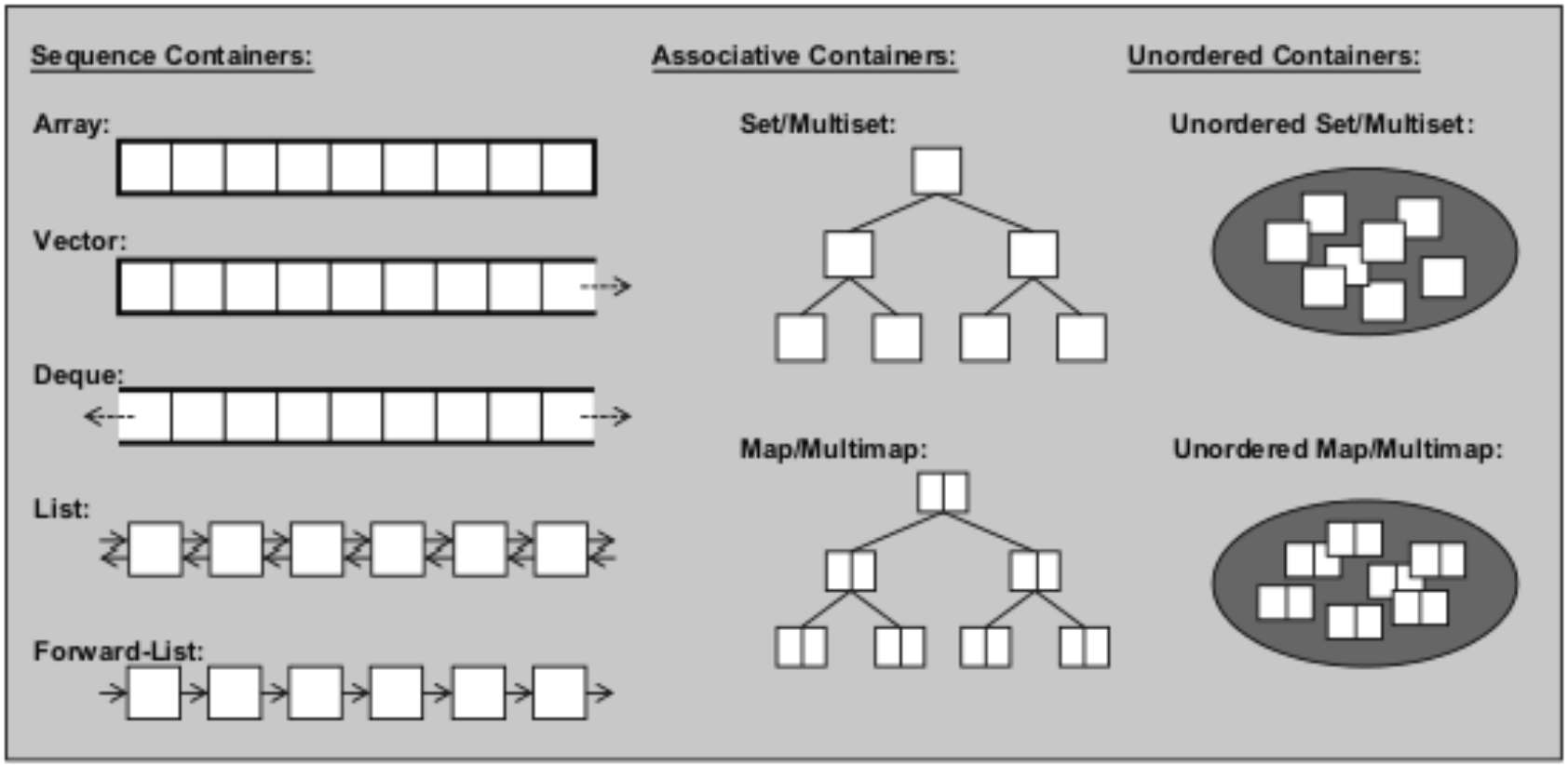
Containers (I)

- Sequence containers: vector<T>, array<T,N> (C++11) deque<T>, list <T>, forward_list<T> (C++11). Ordered collection of elements whose position is independent from the value of the element. In vector and array elements are contiguous in memory. It is possible to modify a stored element directly.
- Adaptors. Built on top of other containers they provide special operations: stack<T>, queue<T> and priority_queue<T>.
- (Ordered) Associative containers. set<T> e map<T,V> (no repetition), and multiset<T> and multimap<T,V> (with repetition). Collections of elements (or of pairs Key-Value) where a strict ordering relation has been defined through a specialization of the functor less<T> or by overloading **operator**<(). The position of an element depends on its value. Elements are accesses read-only, modifications require cancellation and re-insertion.

Containers (II)

- Unordered associative containers (C++11). unordered_set<T>, unordered_multiset<T>, unordered_map<T,V> and unordered_multimap<T,V>. Collections of elements (or of pairs Key-Value). A hashing function, i.e a map from the domain of T values and positive integers in a range $[0, max]$, should be provided together with an equivalence relation (by specializing equal_to<T> or overloading **operator** =(Tconst &,Tconst &), ...) For all standard types a default hash function is provided. Elements are accesses read-only, modifications require cancellation and re-insertion.

The main containers



Picture taken from The C++ Standard Library, (II ed) by N.M. Josuttis, Addison-Wesley, 2012

Iterators

The elements of all containers are addressable with the use of iterators. In particular all containers have the methods begin() and end() which return the iterator to the beginning and to the end (+1) of the container. Iterators behave semantically in a way similar to pointers. For all iterators the following methods are defined.

- **operator** $*$ Dereferencing. For associative containers it returns a constant object.
- **operator** $->$ Addressing a member. In associative containers only const members can be called.
- **operator** $++$ Advance to the next element.
- Operators $=$, $==$ and $!=$.

All containers define the types Container::iterator, Container::reverse_iterator and the corresponding **const** versions (Container::const_iterator etc.)

Special iterators: stream iterators

You may traverse a stream using an iterator. We show here an example of ostream_iterator.

```
#include <iostream>
#include <iterator>
using namespace std;
...
 ostream_iterator<int> out_it (cout,", ");
 int vals[]={1,2,3,4,5,6};
 std::copy(begin(vals),end(vals),out_it);
```

We have defined an iterator to output stream capable to deal with integers, which is connected to the output stream cout and uses a space as separator. The program outputs

```
1 2 3 4 5 6
```

See STL/beginend.cpp.

Methods common to all containers

	Default, copy, and move constructors
Cont c(beg,end)	Constructor from the range [beg,end)
size()	Number of stored elements
empty()	**true** if empty
max_size()	Max number of elements that can be stored
	Comparison operators
c1 = c2	Copy assignment, c1 may be a container of type different from c2
swap(c2)	Swaps data (c2 may be a container of different type)
swap(c1,c2)	As above (as free function)
begin()	Iterator to the first element
end()	Iterator to the position after the last element
rbegin()	Reverse iterator for reverse iteration (initial pos)
rend()	Reverse iterator (position after the last element)
insert(pos,elem)	Inserts a copy of elem (return value may differ)
erase(beg,end)	"Removes" all elements of the range [beg,end)
clear()	Removes all elements (makes container empty)

Methods common to all containers (C++11)

cbegin()	Constant iterator to the first element
cend()	Constant iterator to the position after the last element
emplace(Args...)	Constructs elements in place using the constructor that take Args... as arguments. Its actual signature varies from sequence and associative containers.

C++11 has also introduced the free functions begin(Cont &) and end(Cont &), with the same functionality of the analogous member function to help the creation of user-defined containers. They work also on C-style arrays.

Types defined by all containers

C::value_type	The type of the object stored in a container. Value_type must be Assignable and CopyConstructible, but need not be DefaultConstructible
C::iterator	The type of the iterator used to iterate through a container's elements
C::const_iterator	A type of iterator that may be used to examine, but not to modify, a container's elements
C::reference	type that behaves as a reference to the container's value type
C::const_reference	A type that behaves as a const reference to the container's value type
C::pointer	A type that behaves as a pointer to the container's value type
C::difference_type	A signed integral type used to represent the distance between two of the container's iterators.
C::size_type	An unsigned integral type that can represent any nonnegative value of the container's distance type

Having a **typedef** to the contained type may look strange. After all the type elements in a vector<T> is just T T ! However, this technique helps in generic programming:

```
template <typename Container>
myfun(Container &c){
 typedef typename Container::value_type VT;
 ...
 VT a;...}
```

The introduction of the auto keyword and of decltype() in C++11 may reduce however this need

distance

The distance between iterators is equal to the number of elements in the range defined by them

```
#include <iterator>
...
std::set<double>::iterator a;
std::set<double>::iterator b;
...
int d=std::distance(a,b); number of elements in [a,b[
```

With *random access iterators* (iterators to vector, deque and array we are also allowed to do

```
std:vector<int>::iterator a;
..
int hwomany=v.end()-a; // n. el between a and the end
```

Container::size_type in a sequence container is the type used as argument in **operator** [](), defined for these containers.
It is guaranteed to be an unsigned integral type. Use it instead of just using **int** or **unsigned int** if you think there may be problem with implicit conversions. size_type is in fact implementation dependent.

Computational complexity ($O()$)

Container	[ ]	Iterators	Insert	Erase	Find	Sort
list	n/a	Bidirect'l	1	1	N	NlogN
deque	1	Random	1 at begin or end; else N/2	1 at begin or end; else N	N	NlogN
vector	1	Random	C at end; else_N	1 at end; else_N	N	NlogN
set	n/a	Bidirect'l	logN	logN	logN	1
multiset	n/a	Bidirect'l	logN	d log (N+d)	logN	1
map	log N	Bidirect'l	logN	logN	logN	1
multimap	n/a	Bidirect'l	logN	d log (N+d)	logN	1
stack	n/a	n/a	1	1	n/a	n/a
queue	n/a	n/a	1	1	n/a	n/a
priority_queue	n/a	n/a	logN	logN	n/a	n/a

Computational complexity ($O()$)

Container	[ ]	Iterators	Insert	Erase	Find	Sort
array	1	Random	n/a	n/a	N	NlogN
unordered set	n/a	Unidirec'l	C	C	C	n.a.
multiset	n/a	Unirirec'l	C	C	C	n.a.
map	1	Unidirec'l	C	C	C	n.a.
multimap	1	Unidirec'l	C	C	C	n.a.
forward list	n/a	Unidirec'l	1	1	N	NlogN

C indicate a constant that depends on the hashing function and bucket size of the container.

Headers and compulsory template parameters

Here `T` is the type of the contained element and `K` the type of the key used to address the elements of associative containers, `N` an unsigned integer equal to the size of the array.

vector<T>	<vector>
array<T,N>	<array>
list <T>	<list>
deque<T>	<deque>
set<T>	<set>
multiset<T>	<set>
map<K,T>	<map>
multimap<T,T>	<map>
stack<T>	<set>
queue<T>	<queue>
priority_queue<T>	<queue>
unordered_set<T>	<unordered_set>
unordered_multiset<T>	<unordered_set>
unordered_map<K,T>	<unordered_map>
unordered_multimap<K,T>	<unordered_map>

Ranges (or sequence)

A range indicates an interval of elements "logically contiguous" in a container.
We give a working definition. Two iterators (or pointers) `b` and `e` define a valid range if the instruction

```
for(iterator p=b; p<e;++p) *p;
```

is valid, and `*p` returns the value of elements of the container. The algorithms of the standard library operate on ranges.

The utility <pair>

L'header <utility> introduces the struct pair<T1,T2> (loaded also by <map>), used to contain two values of possible different type.

```
#include <utility>
...
std::pair<double,int> a;
// A very useful utility is make_pair
a=std::make_pair(4.5,2);
// first and second return the values
auto c=a.first; //c is a double
int d=a.second;
auto z=make_pair(3,Polygon());// z is a pair<int,Polygon>
```

The utility <tuple> (C++11)

Tuples are extension of <pair>

```
#include <tuple>
...
// Create a 4 elements tuple.
tuple<string,int,int,complex<double>> t;
// create and initialize a tuple explicitly
tuple<int,float,string> t1(41,6.3,"nico");
///extract an element
cout << get<1>(t1) << " ";
get<2>(t1)="mario";// change an element
// use the utility make_tuple
auto t2 = make_tuple(22,44,"nico");
```

See the examples in STL/tuple1.cpp and STL/tuple2.cpp.

We have already explained vector<T> in detail. We skip it here.

Useful if you have the necessity to insert elements at both ends. It implements

```
T& back(); //first element
T& front(); //last element
iterator push_back(T const &);//Add to the back
iterator push_front(T const &);//Add to the front
void pop_back();//Erase last element
void pop_front();//Erase first element
```

It implements a double-linked linked list. It implements the following methods

```
iterator push_front(T const &);
iterator push_back(T const &);
T& front();
T& back();
iterator insert(iterator pos, T const &);
// remove all elements with value v
void remove(T const v);
// Removes all elements satisfying a predicate
void remove_if(Predicate p);
```

The predicate has to take an element of the type contained in the list and return a bool.

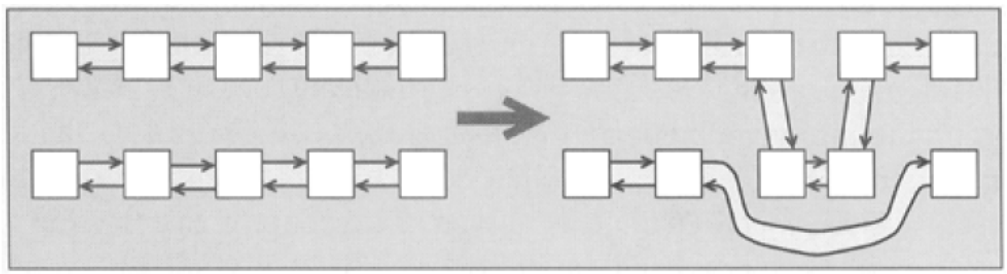
An example

```
// Initialize with an initializer list
// Only c++11!
list<int> l={1,2,4,5,6,7,3};
...
//! using a lambda expression
l.remove_if([](int x){return x>3;});
```

Now the list contains $[1, 2, 4, 3]$.

Operations with lists

c.unique()	Removes duplicates of consecutive elements with the same value
c.unique(op)	Removes duplicates of consecutive elements, for which op() yields true
c1.splice(pos,c2)	Moves all elements of c2 to c1 in front of the iterator position pos
c1.splice(pos,c2,c2pos)	Moves the element at c2pos in c2 in front of pos of list c1 (c1 and c2 may be identical)
c1.splice(pos,c2,c2beg,c2end)	Moves all elements of the range [c2beg,c2end) in c2 in front of pos of list c1 (c1 and c2 may be identical)



Picture taken from The C++ Standard Library, (II ed) by N.M. Josuttis, Addison-Wesley, 2012

It implements a singly-linked list. It provides only part of the functionalities of a list<T> but it is more efficient and less memory consuming.

Sets and mulisets

To use a set or a multiset on elements of type T you need that a strict weak ordering relation is defined on the elements. Let's indicate with $<$ the relation, we must have, for any couple of elements of type T

$$\neg(a < b) \wedge \neg(b < a) \Leftrightarrow a \equiv b$$
$$a < b \vee \neg(a < b) \vee (a \equiv b)$$
$$(a < b) \wedge (b < c) \Rightarrow a < c$$
$$(a < b) \wedge (b \equiv c) \Rightarrow a < c$$

where $\equiv$ is an equivalence relation (non necessarily the equality!) which is in fact defined by the first relation.

How to define the ordering relation (I)

Specializing the function object less<T>

```
template<>
struct less<MyClass>{
bool operator()(MyClass const & a, MyClass const & b){
..., //returns the result of the comparison a<b }
};
```

Overloading the < operator

```
bool operator <(MyClass const & a, MyClass const & b){
..., //returns the result of the comparison a<b }
```

Use the first strategy if you want to limit the use of the comparison to Standard Library associative containers and algorithms. Use the second choice if you want to give it a global scope (in this case you may want to overload the other comparison operators as well!

A Note

For all POD types, pairs, tuples and containers of POD types, as well as strings, pointers, complex<T> the standard library already provides an implementation of less<T>.

How to define the ordering relation (II)

The comparison operator can be specified as second template argument, which obviously defaults to less<T>.

```
struct CompOp{
bool operator()(MyClass const & a, MyClass const & b){...}
};
..
set<int,greater<int>> z;// I use greater instead of less
set<MyClass,CompOp> a; //I use my special functor
```

It is possible also to pass an object in the constructor (it may be useful if the comparison depends on the state of the object

```
CompOp c;
c.setS(..);// do something that affects comparison
set<MyClass,CompOp> a(c());
```

A note

The position of an element in the set is linked to its value, that's why you can access it read-only and you cannot modify it. The memory address of an element does not change after the addition/deletion of another element of the set. Thus, those operations do not invalidate iterators or pointers. (obviously a part those possibly pointing to the deleted element!).

In a set<T> we have no repetition of equivalent elements. Trying to insert an element already existing in the set (in the sense of equivalence) is not an error, it is just a no-operation.
Elements are inserted either with the method insert(Tconst &) or emplace(Args...), the latter builds the element using the constructor with the given arguments (so it is more efficient). Both return a pair<iterator,**bool**> contaning the iterator to the inserted element (o to the already existing element) and a bool which is **true** only if a new element has been inserted.

```
  set<int> s;
  s.insert(5);
  s.insert(10);
  ..
  pair<set<int>::iterator,bool> s.insert(5);
  // pair.second is false!
...
 set<int>::iterator i=s.find(10);
```

find() returns s.end() if the element is not in the set.

Transversing a set

Iterators transverse a set using the inorder transversing. The consequence is that the following code prints the elements of the set into ascending order (with respect to the given ordering relation!!).

```
set<double> a;
typedef set<double>::iterator It;
...
for(It i=a.begin();i<a.end();++i)cout<< *i<<" ";
```

In C++11 the last line could have been written as

```
for(auto i : a )cout<< i<<" ";
```

avoiding the need of the typedef (isn't it nice?).

An example: a set of mesh edges

```
#include<set>
#include<algorithm> //for max() and min()
   bool operator < (Edge const & a, Edge const & b){
   int a1=max(a[0].Id(),a[1].Id());
   int a2=min(a[0].Id(),a[1].Id());
   //The same for b
   if(a1==b1) return a2<b2;
   return a1<b1;
}
...
set<Edge> a;
```

This defines a lexicographic ordering where two edges are equivalent if the vertexes have the same Id (independently from the ordering).

multiset

A multiset is a set where elements may be repeated. More precisely where we may have more than one equivalent elements. The latter are stored in "logically contiguous" locations.

```
#include <set>
...
multiset<double> a;
typedef multiset<double>::iterator It;
a.insert(5.0);
a.insert(10.4);
a.insert(5.0);
pair<It,It> x=a.equal_range(5.0);
```

equal_range() returns a pair of iterators that define a range of the found elements. If x.first==x.second the range is empty (element not found). Method find() returns instead the iterator at the first element found (or a.end()). In C++ the assignment can be made via an initializer list: a.insert({5.0,10.4,5.0}).

Map

A map<key,Value> is the classic dictionary that associates a value to a key. An ordering relation must be defined for the keys, analogously to what already illustrated for a set<T>. A classic example is the telephone directory:

```
#include<map>
...
 map<string,string> rub;
 rub.insert(make_pair("Marco","022344789");
 rub.insert(make_pair("Giorgio","0119876656");
 ..
 rub.insert(make_pair("Marco","02339877");
 // The second Marco entry is NOT inserted
```

Iterators to maps and find()

Dereferencing an iterator to a map<Key,Value> you obtain a pair<Key,Value>. The insert method returns a pair<Iterator,**bool**> as in a set<T>. Also the method find() behaves similarly to a set<T>.

```
typedef map<string,string>::iterator It;
..
auto x=rib.find("Marco");
if (x==rib.end())
    cout<<"Not found";
else
    cout<<"Numero di "<<x->first<<" : "<<x->second;
```

To search for a value you may also use the algorithm find_if, which works on all containers but with $O(N)$ complexity!, by passing a functor (or a lambda expression)

```
#include <algorithm>
 string number("022344789");
 auto i=find_if(rub.begin(),rub.end(),
   [&number](pair<string,string> const & i){
       return i.second==number});
if(i !=rub.end())cout<<"Found: "<<i->first;
```

Here I have used a lambda expression to test is the entry value is equal to the given number. You may prefer a functor.

The operator [] on maps

A map<T> provides an implementation of operator [], which makes life easier (but be careful!):

```
#include<map>
...
 map<string,string> rub;
 rub["Marco"]="022344789";
 rub["Giorgio"]="0119876656";
 ..
 // Returns the corresponding value
 string a=rub["Marco"];
 string b=rub["Maria"];// ATTENTION!
```

If a value with key Maria is not contained in rub the instruction b=rub[“Maria”] adds the entry aria,string()Maria,string() to rub.

So beware when using [] as rvalue!, prefer find().

Multimap

A multimap is a map where we may have more than one entry associated to equivalent keys. Multimaps do not provide **operator** [].

We do not say much more here about multimaps: methods to access them are analogous to those in a multiset. Remember however that dereferenced iterators provide a pair<Key,Value>.

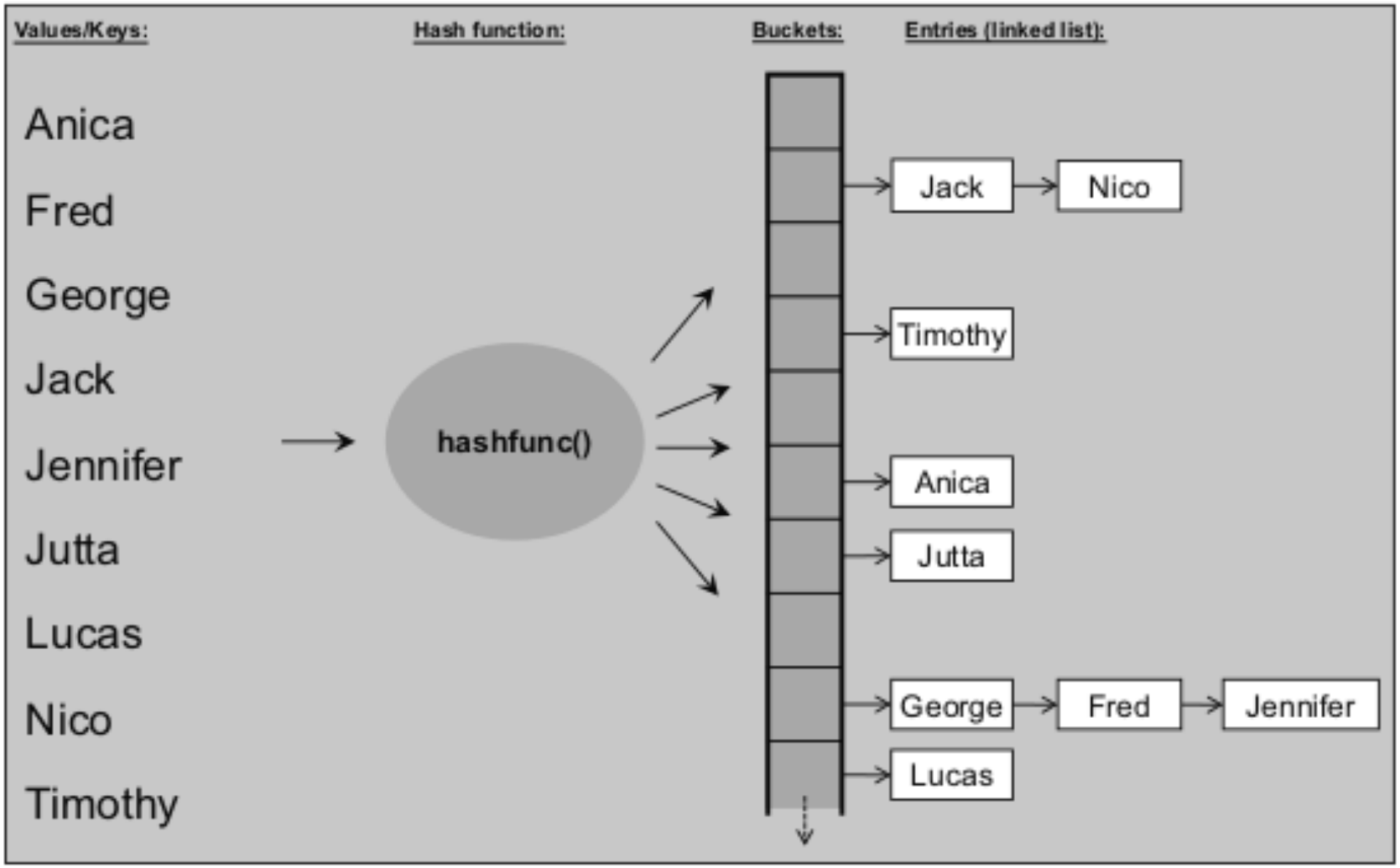
Unordered containers

Unordered containers are template classes of the form

```
template <typename T,
typename Hash = hash<T>,
typename EqPred = equal_to<T>,
typename Allocator = allocator<T> >
class unordered_XXX;
```

where `XXX` is either `set`, `multiset`, `map`, `multimap`. They mimic the functionality of the analogous ordered containers, but they do not require to define an ordering relation (you need an equivalence relation instead).

The general internal layout



Picture taken from The C++ Standard Library, (II ed) by N.M. Josuttis, Addison-Wesley, 2012

Requirements to use unordered containers

To use an unordered container in a type `T` you need to define an equality operator, by overloading **operator** ==() or by specializing the standard functor equal_to<T>, and an hashing function, by specializing hash<T>.

Note: For most common types an implementation of the hashing function (as well as of the equality operator) is already provided by the language (but you can change it if you need to).

Equivalence relation

It has to satisfy the following properties for any elements in the range of T

$$a == b \Rightarrow b == a$$
$$(a == b) \wedge (b == c) \Rightarrow a == c$$
$$a == a$$

Hash function

The hash function is a map

$$\mathcal{H} : \text{Range}(\mathtt{T}) \to [0, N[$$

where N is (usually a big) unsigned integer. Internally the container further transforms (via the modulo operator) the interval $[0, N[$ into $[0, N_b[$, N_b being the number of buckets.
In a good hash function

$$k_1 \not\equiv k_2 \Rightarrow \text{prob}\left(\mathcal{H}(k_1) = \mathcal{H}(k_2)\right) \simeq 1/N$$

To implement a hash function for your type you need to specialize

```
template< class Key > struct hash;
```

We omit the details, which may be found in any good reference manual (also on the web).

Ordered or unordered associative containers?

The public interface of ordered and unordered associative containers is rather similar. Remember however that inserting new elements in an unordered container can cause rehashing, i.e the re-arrangement of the internal layout by changing the number of buckets. Consequently, iterators to elements may be invalidated.

Use the ordered version if you plan to exploit the ordering, or you need to guarantee that iterators/pointers to the contained elements are not invalidated by insertions/deletions. Otherwise, prefer unordered containers since they are more efficient (if you have a decent hash function!).

A stack<T> is an adapter to container that implements a LIFO list (but it has a default!). We have three methods:

```
T & top(); // The element at the top
void push(const T &);// add an element
void pop(); //Delete top element
void emplace(Args...); //Construct element at the top
```

You can change the underlying container by passing it in the constructor:

```
stack c; //Empty stack
vector<int> a;
...
stack<int> s(a);//uses a to initialize s
```

Note: emplace() is only C++11.

Implements a FIFO. Methods are

```
// first element inserted
T& front()
T& back() //last element inserted
void push(const T&) //add element
void pop(); //Delete first (oldest) element
void emplace(Args...)// Construct a new element
```

The considerations made for a stack<T> apply also here.

Priority queue

Same methods as a stack<T> but returns the "largest" element w.r.t. an ordering relation, which defaults to less<T> (like in a set<T> the ordering relation can be changed.)

It can be very useful to track elements for which a certain value is maximal (remember that "maximal" is related to the chosen ordering!).

Initialization by a range

All containers (but not the "adapters") can be initialized providing a range of values. There is also a method assign() that allows to do the same on already existing containers. This makes easy to copy a container into another:

```
set<Edge> e;
// I find the mesh edges using a set
..
// when found I put them into a vector
vector<Edge> ev(e.begin(),e.end());
e.clear();// Clear the set.
```

Parameter list initialization (C++11)

The new standard allows to initialize containers via parameter lists

```
set<int> a={2,3,4,5};
vector<double> b={3.5,6.7};
// Note the double {{}}
map<int,double> c={{3,4.4},{6,7.8}};
c.insert({6,8.0});
```

Which container to use?

Of course it depends on the type of operations you need to perform on the containers. Sets, for instance, are very useful when you need to collect data eliminating repetitions (in this case you may also use unordered sets), or when you need to perform efficiently operations like intersection, union, difference (there are special algorithms for that!). You need ordered sets in this case.

Maps are very useful to implement dictionaries and object factories.

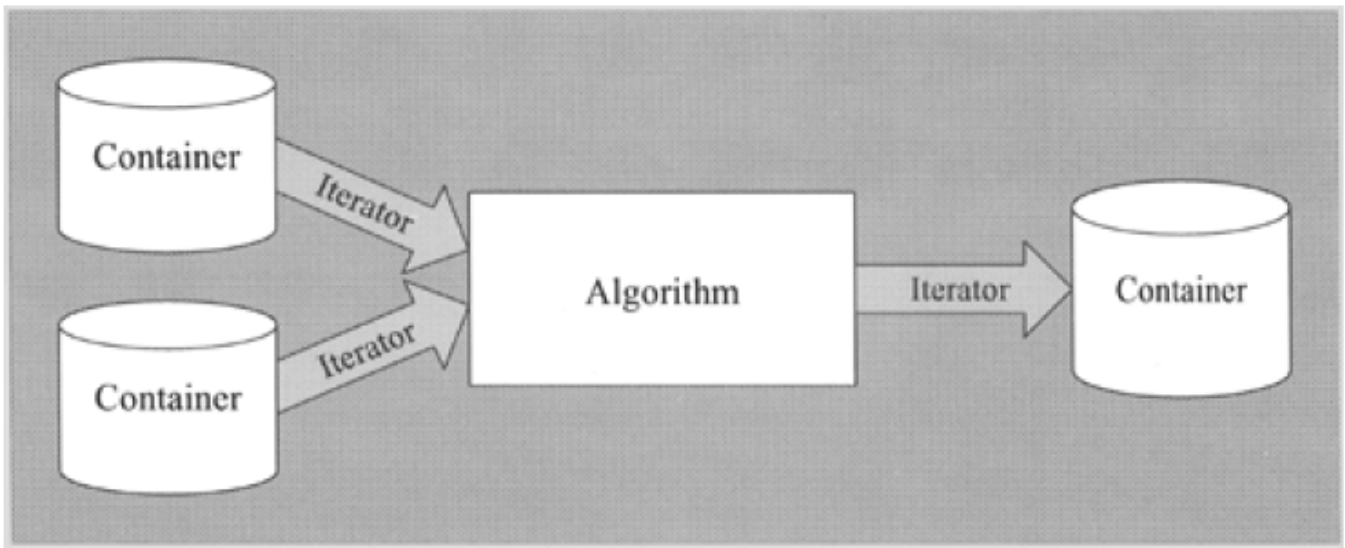
Keep in mind, however, that having contiguous memory reduces cache misses, and cache misses can easily become the major source of inefficiency in your code! So, if you do not have special reasons to do otherwise, prefer to use vector<T> and array<T>.

Examples

Examples are in STL/cont

Algorithms

The standard library provides an extensive set of algorithm to operate on containers, or more precisely on ranges. A full list is here.



Type of algorithms

- Non modifying. Do not modify the value of the range. They work also on constant ranges. Example:

```
It find(It first, It last, T const & value)
```

finds the first occurrence of value in the range [first,last).

- Modifying They either modify the given range, like

```
void fill(ForwardIt first, ForwardIt last, const T& value );
```

that assigns the given `value` to the elements in the range `[first, last)`. Or, they copy the result of an operation into another (existing!) range (beware of the possible need of inserters, as we will discuss later). For instance

```
OutIt copy (InIt first, InIt last, OutIt result );
```

copies [first,last) in the range that starts in result.

Types of algorithms

- Sorting. Particular *modifying algorithms* operating on a range to order it according to a ordering relation (less<T> by default):

```
#include <functional>
 vector<double> a;
 ...
 // decreasing order a[i+1]<=a[i]
 sort(a.begin(),a.end(),greater<double>());
 // increasing order a[i+1]>=a[i]
 sort(a.begin(),a.end());
```

  stable_sort preserves order between equivalent elements (but it is slower)

- Operating on sorted range: search algoritms

```
bool binary_search(It first,It last,T const& value)
```

  returns true is the value is present.

Types of algorithms

- Operating on sorted range. Set union, intersection and difference (but they do not need to be a set<T>, it is sufficient that the range is ordered):

```
#include <iterator>// for the inserter
   set<int> a;
   set<int> b;
   ...
   set<int>c;
   set_union(a.begin(),a.end(),
        b.begin(),b.end(),inserter(c,c.begin()));
// I need the inserter to write on a set
```

Now $c = a \cup b$.

Types of algorithms

- Heap. To create and manipulate heaps.
- Min e max. A series of algorithms to find minimum and maximum element in a range:

```
template< class T >
const T& max( const T& a, const T& b );
template< class T, class Compare >
const T& max( const T& a, const T& b, Compare comp );
// Only C++11
template< class T >
std::pair<const T&,const T&> minmax( const T& a,
                                     const T& b );
template< class InputIt1, class InputIt2 >
bool lexicographical_compare(InIt1 first1,InIt1 last1,
                             InIt2 first2,InIt2 last2 );
```

Types of algorithms

- Partitioning operations. To partition a range according to a given criterion passes as a function object.
- Numeric Operations (you need <numeric>). Examples

```
vector<double> v;
vector<double>w;
// Sums a range
auto sum = std::accumulate(v.begin(), v.end(), 0);
// Product of a range
auto product =
    std::accumulate(v.begin(), v.end(), 1,
                                    std::multiplies<double>());
auto r1 = std::inner_product(v.begin(), v.end(),
                                      w.begin(), 0);
```

Types of algorithms

In <numeric> we have other useful algorithms:

```
template< class ForwardIt, class T >
void iota(ForwardIt first,ForwardIt last, T value );
```

that fills the range `[first, last)` with sequentially increasing values, starting with `value` and repetitively evaluating `++value` (only C++11):

```
std::list<int> l(10);
std::iota(l.begin(), l.end(), -4);
// l contains -4 -3 -2 ....
```

or adjacent_difference that computes the differences between the second and the first of each adjacent pair of elements of a range.

A use of the for_each algorithm

A functor that accumulates statistics

```
template<classs T>
class Statistic{
public:
  Statistic():M_n(0),M_mean(0),M_var(0){};
  T mean()const {return M_mean;}
  T var()const {return M_n==0?0:M_var/M_n;}
  void operator()(double const & x){
    ++M_n; double delta = x - M_mean;
    M_mean+= delta/M_n; M_var+=delta*(x-M_mean);
  }
private:
  int M_n;
  T M_mean, M_var;};
```

```
#include<numeric>
    vector<double>v;
  // fill v with values
  ...
  Statistic<double> s;
  for_each(v.begin(),v.end(),s);
  cout<<" Mean"<<s.mean()<<" Variance "<<s.var()
                                  <<endl;
```

We have applied s() to each element of the container,

Transform

Another very flexible algorithm is transform, present in two forms

```
OutIt transform(InIt first1, InIt last1,OutIt result,
  UnaryOperator op );
OutIt transform (InIt1 first1, InIt1 last1,
                 InIt2 first2, OutIt result,
                 BinaryOperator binary_op );
```

where op is a unary or binary function (typically implemented as a functor, or lambda) with signature and return type

```
Tout operator () (Tin const & a);
```

or

```
Tout operator () (Tin1 const & a, Tin2 const & b);
```

where Tin and Tout are the types of the elements in the input and output container, respectively. Beware that the length of the ranges must be consistent (no check is made!)

Un example of transform

```
set<double> a;
list<double> l;
....// fill a and l with values
vector<double> b(a.size());
transform(a.begin(),a.end(),l.begin(),
b.begin(),std::plus<double>());
// b now contains  a+l
```

Other interesting algorithms

	Non-modifying operations
for_each	Apply function to range
find_if	Find first element satisfying a predicate
count	Count appearances of value in range
count_if	Return number of elements in range satisfying a predicate
	Modifying sequence operations
replace	Replace value
replace_if	Replace values in range satisfying a predicate
replace_copy	Copy range while replacing values
replace_copy_if	Copy range replacing value satisfying a predicate
fill	Fill range with value
fill_n	Fill n elements with value
generate	Generate values according to given unary function
remove_if	Remove elements satisfying predicate
remove_copy	Removing values and copy them to another range
remove_copy_if	Remove elements satisfying a predicate and copy
unique	Remove consecutive duplicates
random_shuffle	Rearrange elements in range randomly
partition	Partition range in two

Inserters

Inserters are special iterators used to insert values into a container. We have three main type

```
\\Insert at the back
back_inserter (Container& x);
\\insert in the front
front_inserter(Container& x);
\\Insert after indicated position
inserter(Container & x, It position);
```

Example:

```
copy(a.begin(),a.end(),front_inserter(c));
```

Beware the cost depends on the type of container!

A example of use of an inserter

Several algorithms require he iterator to the beginning of a valid, non constant range where the output is written. This would make them impossible to use directly on a non sequential container, or on a empty container.

```
#include <algorithm>
using namespace std;
...
vector<double> a;
...// copy the vector on the set
set<double> b;
copy(set.begin(),set.end(),b.begin(); //ERROR!!
```

You need an inserter:

```
copy(set.begin(),set.end(),
                inserter(b,b.begin())); //OK!
```

For an associative container the second argument of is taken only as suggestion.