Advanced Methods for Scientific Computing (AMSC)
Lecture title: The Standard Library

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

The C++ Standard Library: a little bit of history

In November 1993 Alexander Stepanov presented a library (Standard Template Library) based on generic programming to the ANSI/ISO committee for C++ standardization. It contained generic containers (and the iterators) and a set of algorithms operating on them.

The proposal was accepted, and the STL is still one of the components of what is now the Standard Library: a huge set of tools that are an integral part of modern C++.

All functionalities provided by the Standard Library are under the namespace `std::`, or, sometimes, sub-namespaces enclosed in `std::`.

The Boost libraries

Many of the components added to the Standard Library during the years have been originally proposed in the Boost C++ Libraries, a collector of libraries that work well with the Standard Library and are usable across a wide spectrum of applications.

A notable component of the Boost is the Boost Graph Library, very much used by people working with graphs and networks.

Boost libraries are all open-source. They are installable, singularly or as a whole, in all Linux platform using the package manager: `sudo apt-get install libboost-all-dev` on Ubuntu, for instance. But you can download the source and compile them yourself.

An overview of the Standard Library (I)

- Porting of C libraries. Several libraries "inherited" from C have been ported under the std namespace, like <cmath>. They all start with a "c".
- Containers: Generic containers and iterators.
- Utilities: Smart pointers. Fixed-size collection of heterogeneous values: pair and tuple . Clocks and timers. Function wrappers and predefined functors. The class ratio for constant rationals.
- 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. Some of them implement parallel execution.
- Strings and text processing. The class **string** (and derived classes). Regular expressions and tools to operate on strings efficiently.

An overview of the Standard Library (II)

- Support for I/O. The i/o streams and related utilities.
- Numerics complex<T>, numeric limits, random numbers and distributions, basic math operators.
- Utilities for diagnostics. Standard exception classes. Exception handlers.
- Support for generic programming. Type traits, declval, as_const
- Support for reference and move semantic. Reference wrappers, std::move(), std::forward<T>.
- Support for multithreading and concurrency. Threads, mutexes, locks etc. parallel algorithms
- Allocators. Change how objects are allocated in containers.
- Utilities for hybrid data: optional, variant and any.
- filesystem. Utilities to examine the file system

C++20 additions

C++20 has made a few very important additions:
-ranges and views
-range based algorithms;
-concepts

In this course we wil just give a brief overview of standard containers and some algorithms. Yet, remember that the Standard Library is full of useful stuff.

Overview of containers (I)

Containers may be subdivided into different categories, linked to how data is stored and handled internally:

- Sequence containers: vector<T>, array<T,N> deque<T>, list <T>, forward_list<T>.

  Ordered collection of elements whose position is independent from the value of the element. In vector and array elements are guaranteed to be contiguous in memory and is possible to access the elements directly with the [] operator.
- Adaptors. Built on top of other containers, they provide special operations: stack<T>, queue<T> and priority_queue<T>.

Overview of containers (II)

Associative containers: Collections of elements where the position of an element depends on its content. They are divided into two main types: maps, whose elements are key-value pairs, and sets, where the element are just values (in a set we may consider that keys and values are the same.) Moreover, we have a further subdivision: ordered and unordered, which depends on the way the elements are stored and impose different requirements on the type of elements.

Note: in a set we use the terms *value* and *key* interchangeably since they are equivalent.

Overview of containers (III)

Ordered associative containers: They are:

- set<K> (no-repetition) and multiset<K> (repetition allowed): they store single values and the value is the key.
- map<K,V> (no repetition of keys) and multimap<K,V> (repetition of keys allowed): they store pairs of (key,value). They act as dictionaries.

A ordering relation must be defined for the key K. It can be done by passing a specific callable object, or by a specialization of the functor std::less<K>, or by defining **operator**<(). Keys can be accesses read-only, modifications of keys require special operations..

Overview of containers (IV)

Unordered associative containers. They are

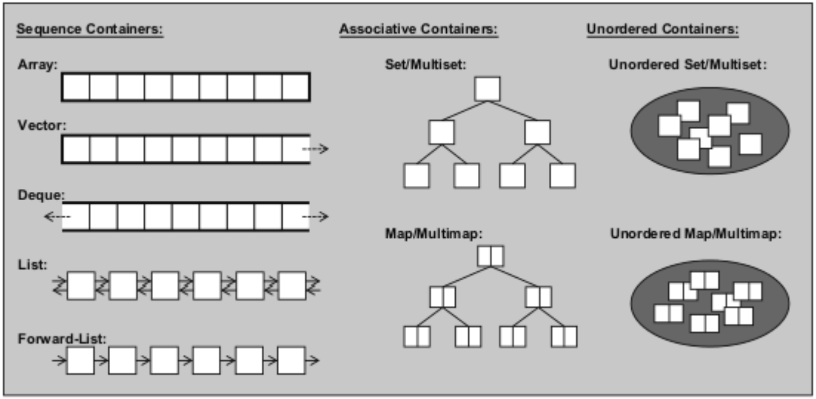
- unordered_set<K>, and unordered_multiset<K>.
- unordered_map<K,V> and unordered_multimap<K,V>.

Their general behavior is similar to that of the ordered counterparts.

A hashing function, that is a map from the domain of the keys and positive integers in a range $[0, max[$, should be provided together with a proper equivalence relation among keys (by specializing equal_to<K> or defining **operator** ==), or passing a user defined functor.

For all standard types a default hash function is provided by the library as well as relational operators.

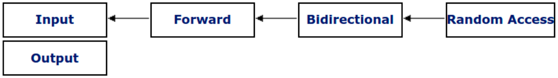
The main containers



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

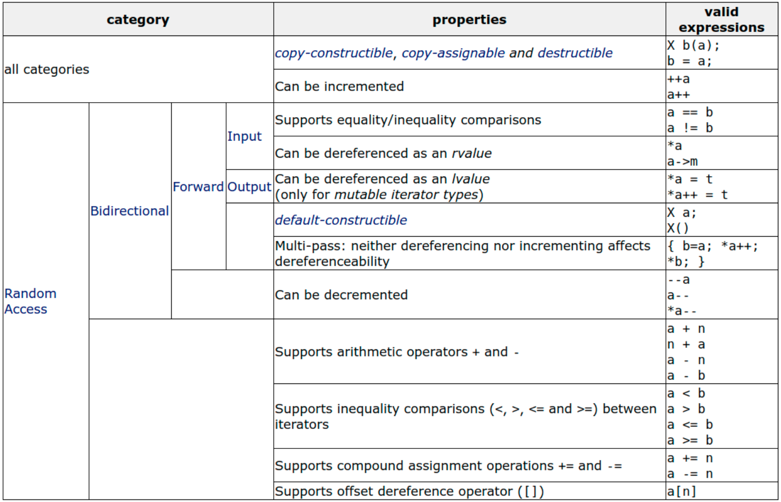
Iterators

The iterators allow to iterate over the elements of a container.
It can be dereferenced with the $*$ operator, returning an element of the range, and incremented (move to the next element), with the $++$ operator.
A special object is sentinel which allows to mark the end of a sequence of elements: often the sentinel is just a past-of-end iterator.
They can be classified into six categories, five of which illustrated in the figure



The sixth is the contigous iterator, which is a random access iterator on contigous memory area.

Operations on iterators



category				properties	valid expressions

all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);`
					`b = a;`

				Can be incremented	`++a` `a++`

Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b`
					`a != b`

				Can be dereferenced as an *rvalue*	`*a`
					`a->m`

			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t`
					`*a++ = t`

				default-constructible	`X a;`
					`X()`

				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n`
					`a -= n`

				Supports offset dereference operator (`[]`)	`a[n]`

Taken form cplusplus.com

Containers and Iterators

All standard containers have iterators and as we will see, also standard algorithms may operate using iterators:

- Forward iterator: `std::forward_list`.
- Bidirectional iterator: `std::list`.
- Random access iterator: `std::deque`
- Contiguous iterators: `std::string`, `std::vector`, `std::array`, `std::span`, `std::valarray`.

Input and output iterators are used to iterate on input and output streams.

std::span

The class template std::span<T,N>, introduced by the header <span>, describes an object that can refer to a contiguous sequence of objects with the first element of the sequence at position zero. It is useful to give a uniform interface to function that may take both c++ contiguous containers or C style arrays,.

```
template<class T, std::size_t N>
auto fun (std::span<T,N> x );
...
double xc[3];
std::array xa={3,4,5};
fun (xc );
fun (xa );
```

Sequences and ranges

With the term sequence (or range) we define a set elements that are “logically adjacent” in a container. It is defined by a couple of objects, formed by an iterator and a ”sentinel” (often another iterator), that allow us to traverse the sequence.
If we call them start and stop the standard assures that

```
for (auto x=start ; x!=stop) a = *x;
```

iterates over all elements of the sequence.

The range concept

Since C++20 the range concept has a precise meaning, is any class object that contains methods begin() and end() that return a valid iterator and a sentinel to a sequence of elements:

```
template<class T>
concept range = requires(T& t) {
  ranges::begin(t);
  ranges::end(t);
};
```

Thus, all standard containers satisfy the range concept. This will become crucial if you use standard views and constrained algorithms. For any objectv of a class that obeys the concept of a range, the statement

```
for (auto x: v) a=x;
```

is equivalent to

```
for (auto x=v.begin();x!=v.end();++x) a=*x;
```

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()	sentinel to the position after the last element
rbegin()	Reverse iterator for reverse iteration (initial pos)
rend()	Reverse sentinel (position after the last element)
insert(pos,elem)	Inserts a copy of elem (return value may differ)
emplace(pos,args...)	Insert element by constructing it in place
erase(beg,end)	"Removes" all elements of the range [beg,end)
clear()	Removes all elements (makes container empty)

Methods common to all containers

cbegin()	Constant iterator to the first element
cend()	Constant iterator to the position after the last element

You have also the free functions std::begin() and std::end(),etc., that take a container in input, with the same functionality of the analogous member functions. They work also on C-style arrays and may be overloaded for their use with user-defined containers.

Most containers provide also comparison operators that perform lexicographic comparison of the elements of the container, and the equivalence operator (==).

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[
```

Distance may be negative if iterators are *Random Access*.

With *Random Access* iterators (like those of vector, deque and array) we are also allowed to subtract iterators to obtain the distance:

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

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.

Beware of computational complexity

Computational complexity gives you just the order by which the number of operations grows with N.

But the actual time depends also on the memory bandwidth. In particular, all CPU have a cache memory with a much faster access than that of the main RAM. The system can guess the next elements that will be accessed, do a pre-fetching and store them in the cache. This is certainly the case for contiguos memory sequential containers (vectors and arrays).

Consequence: if N is not very large it is difficult to beat a `std::vector` even on operations not suited for it.

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>

vector<T> and array<T,N>

You should know vector<T> and array<T,N> in detail. We skip it here.
We only recall that memory area is contiguous and iterators are RandomAccess.

deque<T>

A sequential container useful if you have the need to insert elements at both ends, but not access them in an arbitrary position. 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
iterator emplace_back(Args&&...);//Emplace to the back
iterator emplace_front(Args&&...);//Emplace to the front
void pop_back();//Erase last element
void pop_front();//Erase first element
```

The stack<T> adapter

A stack<T> is an adapter to a container that implements a LIFO (last in, first out) list. We have these 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
```

Very useful when what you need is indeed a LIFO.

The queue<T> adapter

Implements a FIFO (first in, first out) list. 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
```

Very useful when what you need is indeed a FIFO.

The priority_queue adapter

Similar to a stack, but it keeps track of the "greatest" element according to a comparison rule (defaulted to less-than).

```
#include <queue>
 const std::vector data = {1, 8, 5, 6, 3, 4, 0, 9, 7, 2};
 std::priority_queue<int> q1{data.begin(),data.end()}; // Max prior
 auto x=q1.pop(); // x is 9
 auto y=q1.pop(); // y is 7
 q1.push(40);
 x=q1.pop();// x is now 40
```

See cppreference.com for details

list<T>

It implements a double-linked list. Among the main methods (there are many others)

```
iterator push_front(T const &);
iterator push_back(T const &);
T& front();
T& back();
// insert before position pos
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. (Note remove_if is present also as free function taking any range)

An example

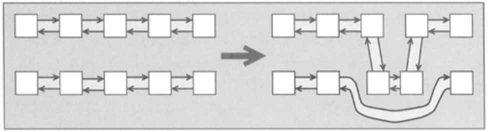
```
// Initialize with an initializer list
list<int> l={1,2,4,5,6,7,3};
...
// using a lambda expression
l.remove_if([](int x){return x>3;});
```

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

Lists are sequential containers and are convenient when you need to do merging or splicing (see next slide). Their iterator are however only `Bidirectional`, since elements are not stored contiguously.

Operations with lists

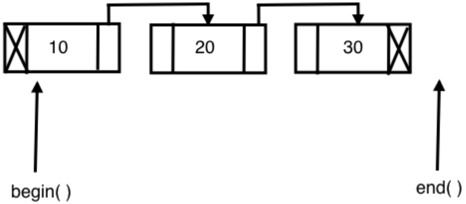
c.unique()	Removes duplicates of consecutive elements with the same value
c.unique(op)	Removes duplicates of consecutive elements, for which op() yields true
c1.splice(pos,c2)	Moves all elements of c2 to c1 in front of the iterator position pos
c1.splice(pos,c2,c2pos)	Moves the element at c2pos in c2 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

forward_list<T>

It implements a singly-linked list. It provides only part of the functionalities of a list<T> but is more less memory demanding. In a forward list you have only a `Forward` iterator.



It is clear from the figure that you can only move "to the next element" and not go backward!

A note to the method insert

The method insert is present in all containers, but in different flavours depending on the type of containers. In all containers Cont we have the version

```
Cont::iterator insert(Cont::iterator hint,const Cont::value_type& value)
```

that, for sequential containers inserts value at position before that indicated by hint. For associative containers hint is indeed just an hint: the position in the container is determined by the value. The version

```
Cont::iterator insert(const Cont::value_type& value),
```

exists only for associative containers (where the position will be found according to the value).

All containers have also a version that takes the hint and a range to be inserted.
The same considerations apply to emplace.

A digression: comparison operators

C++ comparison operators are: the relational operators, $<$, $<=$, $>$, $>=$, which define an ordering, and the equivalence operators $==$ and $=!$, (and the spaceship operator, $<=>$ that we neglect at the moment).
They are consistently defined if, taking $<$ as basic operator,

$$
\begin{aligned}
a == b &\iff \neg(a < b) \wedge \neg(b < a)\\
a\,! = b &\iff \neg(a == b)\\
a \leq b &\iff (a < b) \vee (a == b)\\
a > b &\iff \neg(a \leq b)\\
a \geq b &\iff (a > b) \vee (a == b)
\end{aligned}
$$

They can be defined (and even deleted!) for user classes, since C++20 however, defining $==$ automatically synthesize $=!$ (with obvious result). If a class is an aggregate the compiler tries to synthesize the comparison operators using lexicographic comparison on the aggregate data members.

Another digression: weak ordering and strong ordering

C++20 has introduced a more rigorous classification of ordering relations, allowing to distinguish among weak and strong orderings (and partial ordering). Here, I only recall the main concepts.

We need first to recall the concept of equivalence. Having introduced a strict ordering $<$ the induced equivalence $(==)$ is $\neg(a < b) \wedge \neg(b < a)$. Equality instead means that the internal representation of a is identical to that of b.

A weak ordering relation is a strong ordering relation if the induced equivalence implies that for any function f returning a comparable value, $f(a) == f(b)$ whenever $a == b$.

Example: if I compare two integers modulo 2 I have a weak ordering: according to this ordering $2 == 8$, but if I take $f(i) = i/2$, we have $f(1) \mathrel{!}= f(3)$ according to the given ordering.

Sets and multisets

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

$$\begin{array}{l} (a < b) \wedge (b < c) \Rightarrow a < c \\ \neg[(a < b) \wedge (b < a)] \end{array}$$

These are the obvious properties of less-than, but... are you sure of satisfying them in the comparison operator you are defining for your class.... Beware!

An more complex example

Let's have a class for rational numbers and a some comparison operators defined simply as

```
struct Rational
{ int num;
  int den;
}
bool operator<(Rational const& a, Rational const & b)
{return a.num*b.dem<a.dem*b.num;}
bool operator ==(Rational const& a, Rational const & b)
{return not(a<b) and not(b<a));}
```

If do not keep the rationals normalised, you only a weak ordering since

```
int f(Rational const & a){return a.num;}
```

gives $f(a) \neq f(b)$ if, for instance, I consider the two equivalent Rationals $a = 1/2$ and $b = 2/4$.

Consistency of comparison operators

Having defined the operator $<$ (less) that introduces a strict weak ordering, we have a whole set of consistent comparison operators (we use the C++ syntax), defined recursively as

== (equivalence)	**not**(a<b) **and** **not**(b<a))
<= (less_equal)	a<b **or** a==b
> (greater)	**not** a<b
>= (greater_equal)	a>b **or** a==b
!= (not_equal_to)	**not** a==b

It is safer to define consistent relational operators. C++20 has introduced the spaceship operator, <=>, that may help to this aim.

Specify an ordering relation for a user defined class

In order of generality: i) Overloading the < operator

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

ii) Specializing the function object less<T>

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

iii) passing a functor as template parameter.

```
struct CompOp{
  bool operator()const
     (const MyClass& lhs, const MyClass & rhs);
  ...
  std::set<MyClass,CompOp> s;
```

How to pass a comparison op to a set

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

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

How to pass a comparison op to a set

The comparison operator may have a state, in that case you may pass a particular instance via the constructor.

```
 struct modCompare
  {
    int mod;
    bool operator(int a, int b)
    {return a % mod <b % mod;}
  };
...
  modCompare compOp{3};// comparison modulo 3
  std::set<int,modCompare> s{compOp};
  // or std::set<int,modCompare> s{compOp{3}};
```

Using a lambda as a relational operator

If you change the comparison operator you need to specify the type as second template argument of std::set. This would be impossible if the operator is a lambda expression, but luckily we have **decltype**:

```
// A lambda that defines a comparison operator between ints
auto comp=[](int const & a,int const & b){return a%10 >b%10;};
 //I have to use decltype to deduce the type and pass the object
set<int,decltype(comp)> s{comp};
```

Note that I have also to pass the lambda as argument of the constructor as lambdas are NOT DEFAULT-CONSTRUCTABLE. But they are copy-constructable, so, the passed lambda object will be copied in the private member of set that stores the comparison operator.

What to use

i) Overloading the $<$ operator has a global effect. Use it when you want to implement a full set of comparison operators ($<=$, $>$ $==$) for your class. Beware: it is better to be consistent!.

ii) Specialize std::less<T> template struct if you want your comparison operator limited to algorithms and containers of the Standard Library. The standard will provide a consistent equal_to<T> automatically.

iii) Use your own functor if you want to implement your comparison just for a specific std::set or std::map object.

A note

For all POD (plain old data) types, as well as strings, pointers, complex<T>, the Standard Library already provides an implementation of less<T> (as well as of **operator**<()), and of all the other comparison operators.

For standard vectors, tuples, arrays and other sequential containers the language provides comparison operators that use lexicographic comparison.

Traversing a set

Iterators traverse a set using inorder traversal. The consequence is that the following code prints the elements of the set in ascending order with respect to the given ordering relation.

```
 set<double> a={1,2,3};
 set<double,std::greater<double>> b={1,2,3};
...
   for(auto i : a )cout<< i<<" ";
   for(auto i : b )cout<< i<<" ";
```

elements of a are printed from smallest to largest (1,2,3), those of b from largest to smallest (3,2,1).
More formally, if R is the ordering, when the set is traversed the values are such that $v_0 \, R \, v_1 \, R \, v_2 \cdots$.

A more complex example: a set of mesh/graph 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: Edge (1,2) is equivalent to (2,1).
See STL/SetEdge/Edges.hpp.

multiset

A multiset is a set where equivalent elements may be repeated.

```
#include <set>
...
multiset<double> a;
a.insert(5.0);
a.insert(10.4);
a.insert(5.0);
auto [It1,It2] = a.equal_range(5.0);
```

equal_range() returns a pair of iterators that defines a range of equivalent elements. If It1==It2 the range is empty (element not found). Method find() returns instead the iterator at the first element found (or a.end() if not present).

A note on set (and associative containers in general)

In a set<T> we have no repetition of equivalent elements. In other words a set<T> represents a totally ordered set.

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

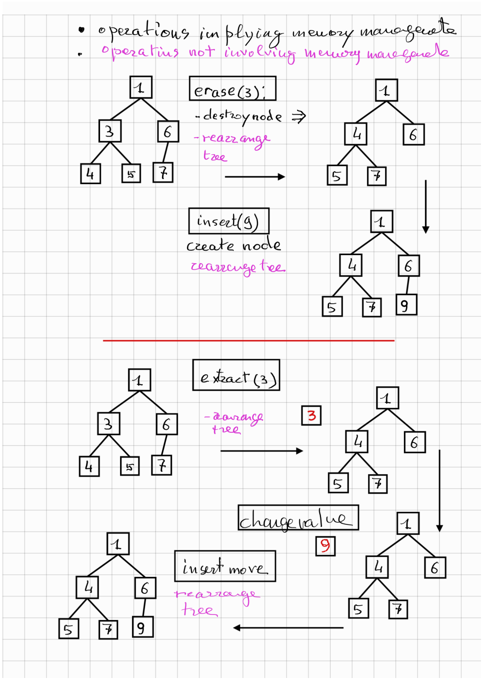
Operations that return a iterator to an element of a set, do return const iterators, i.e. iterators whose pointed element cannot be changed.

Changing an element in an associative container

A possibility is removing the old element and inserting the new one. A more efficient technique adopts the method extract() that returns a handle to a node of the container internal structure:

```
// extracts the node stoning oldvalue
auto node=s.extract(oldvalue);
if(node.empty()) // check if oldvalue is not in the set
  s.insert(newvalue);
else {
  node.key()=newvalue;// change key (=value)
  s.insert(std::move(node))// move back the node
};
```

Notes A node handle is not copyable but it's movable, that's why you need std::move. Being s a set, I could have used node.value()=newvalue; since in a set values and keys coincide. With this technique if oldvalue is present, we avoid useless deallocation/allocation of memory.



In the figure I have reported the different operation performed by the two different strategies for changing the key in an associative container. In black the operations that imply memory management. You may see that in the case of the use of extract there are none!

Map

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

```
#include<map>
...
 map<string,string> rub;
 rub.insert({"Marco","022344789"}); //pair is an aggregate
 // you can insert more than one
 rub.insert({{"Giorgio","0119876656"},{"Silvia","0376688"}});
 rub.insert({"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>, and it may be used to search by a key.

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

Searching a value

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

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

A note: Here, I have used a lambda expression to test is the entry value is equal to the given number. With the new constrained algorithms you can do

```
 auto i=std::ranges::find_if(rub,
   [&number](auto const & i){return i.second==number});
```

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 Maria to rub.

So beware when using [] at the rhs of an expression!, prefer find() in that **case**.

Traversing a map

When traversing a map elements are given ordered according the ordering relation of the Key.
Note that with map structured bindings can be very nice.
Indeed, a std::pair is an aggregate.

```
for (auto [key, value] : aMap) // get key and value
    {
    ...
    }
```

For more functionalities of a map, look at the example in STL/Map

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. Also for multimap dereferenced iterators provide a pair<Key,Value>.

To account for repeated keys, we have the equal_range() method, which returns a pair of iterators defining a sequence containing the elements with the given key.

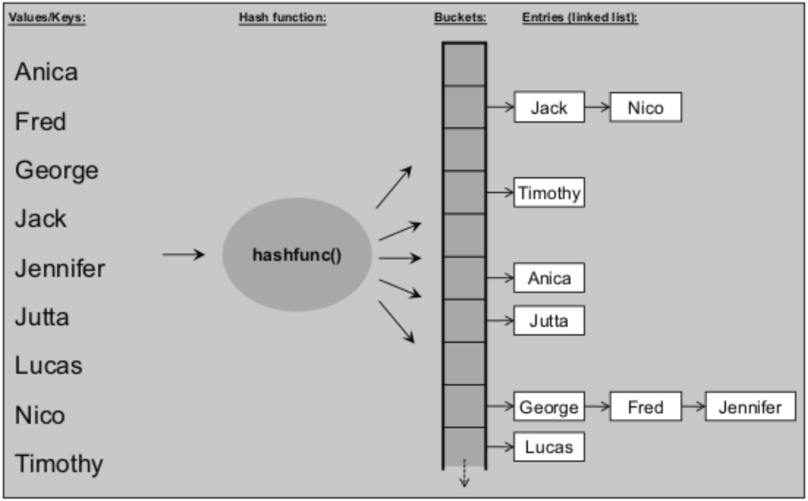
Unordered containers

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, with practically all methods replicated, but they do not require to define an ordering relation, but you need an equivalence relation and a hashing function 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 with a key of type `T`, you need:

- an equivalence operator, by defining **operator** ==() or by specializing the standard functor equal_to<T>, or using a function object;
- an hashing function, by specializing hash<T>, or using a function object.

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

Equivalence relation

An equivalence relation for type T has to satisfy the following properties (weak equivalence relation): for any couple of elements of type T

$$\begin{aligned}
&a == b \Rightarrow b == a \\
&(a == b) \wedge (b == c) \Rightarrow a == c \\
&a == a
\end{aligned}$$

It looks obvious, but... are you satisfying them in your $==$ operator?

Note: to define an unordered container we just need weak comparison, it means that $a == b$ does not necessarily imply that $f(a) == f(b)$.

Hash function

The hash function is a map

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

where N is (usually a big) unsigned integer (of std::size_t type). 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 may specialize std::hash<T> or use any functor taking an object of your class and returning a std::size_t.

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

Defining a hash function for MyType

One can use, for instance, a functor

```
struct MyTypeHash
 {
  std::size_t operator()(MyType const& s) const
  {
    std::size_t result;
    // here the computation of the hash.
    return result;
  }
  };
  ....
std::unordered_set<MyType,MyTypeHash> x;
```

Also in this case, if the callable object has a state that affects the hash function it may be passed as argument in the constructor.

A first concrete example

A possibility is to exploit the hash function provided by the language. A possible hash function for the Rational class

```
struct RationalHash
 {
   std::size_t operator()(Rational const& p) const
   {
     return intHash(p.den())/2 + intHash(p.num())/2;
    }
private:
   std::hash<unsigned int> intHash;
};
...
std::unordered_set<Rational,RationalHash> r;
```

Is this a good choice? Probably not, but it's simple!

A better hashing for the Rationals

To create a more effective hash function exploiting already defined ones, you may use the the Utility/hashCombine.hpp facility (hacked from the boost library)

```
 struct RationalHash
{
std::size_t operator()(Rational const& p) const
 {
  apsc::hash_combine(seed,p.num(),p.den());
  return seed;
 }
private:
  std::size_t seed=0u;
};
...
std::unordered_set<Rational,RationalHash> r;
```

`hash_combine` uses some tricks to increase entropy! It takes an arbitrary number of arguments and uses internally `std::hash<T>` for all argument types.

Ordered or unordered associative containers?

The public interface of ordered and unordered associative containers is very 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, or pointer, to elements may be invalidated. This is not true for ordered containers.

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 for key lookups (if you have a decent hash function!), even if they may be more memory demanding.

Initialization by a sequence

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();// Clears the set.
// Reassign
e.assign(e.begin(),e.end());
```

Moving elements between containers

The new method extract and overload of insert allows to move elements between associative containers (also called splicing) in an efficient and uniform way. We show it here for a multimap, but the technique is analogous for the other associative containers.

```
std::multimap<int,std::string> src{{1 "one"},
                                   {2,"two"},
                                   {2,"onothertwo"},
                                   {3,"three"}};
std::map<int,std::string> dst{{4 "four"}};
// splicing using an iterator
auto it = src.find(2);
dst.insert(src.extract(it));
// but you may also just give the value
dst.insert(src.extract(3));
```

You may splice between containers with elements of the same type.

Examples

Some other examples are in STL/cont

Recap on containers

- We have three major type of containers: sequential, associative ordered, associative unordered;
- Iterators are tools that allow to iterate on standard containers in a uniform way;
- Ordered containers require a weak strict ordering relation on the keys, defaulted to less than (<). It must be well defined. You can define your own ordering in three ways; from more to less general: overloading <, specializing std::less<T>, or passing a callable object to the container at construction. In the scope of the container the equality induced by the ordering: a==b is computed as !(a<b) **and** !(b<a). You do not need to define == to use the container.
- Unordered containers require the equivalence operator == to be defined for the keys and a hash function. You can define your own ordering in three ways; from more to less general: overloading ==, specializing std::equal_to<T>, or passing a callable object to the container at construction . If you need it, you can define your hash function by specializing std::hash<T> or by passing it to the container at construction.

Recap on containers

sequential, contiguos memory containers are normally to be preferred for their cache friendliness. However, query is $O(N)$ and also insertion of new elements non at the end is $O(N)$.

sequential, non-contiguos memory containers (lists). Good if you need to do merge and splicing (but it can be done also with the ordered containers). Addition/deletion is $O(1)$

associative ordered are very nice if you want to keep the data in order. Insertion/deletion are $\log_2(N)$. They use more memory than sequential contiguous containers and are not cache friendly.

associative unordered Excellent if you need to do insert/delete or query very often (all $O(1)$ operations!). Moreover, a complex object, often $==$ is simpler to define than $<$. It can use more memory than the equivalent ordered container. Prefer it if queries for contained values is the most important operation in your code.

The standard library provides also specialized "container type objects", called adaptors for special operations: std::stack, std:queue and std::priority_queue.

The optional utility

Often one needs to indicate that a value has not been set or that it is a "missing datum". Traditionally this is done by selecting a "very unlikely value" as a placeholder for missing/unset value. For instance, in R missing data is represented with a quiet NaN. In modern C++ we have a better way of doing it: optional.

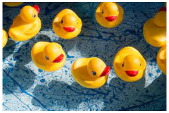
std::optional is a special wrapper to a type that behaves partially similarly to a pointer, but is convertible to bool, and **false** indicates that the value is unset. It contains also other methods to interrogate its content. It needs the header <optional>

An example of optional

```
using Data = std::vector<std::optional<double>>;
Data data(100); // now the elements are all unset
data[10]=45.27;//you set the optional just by assigning the value
auto d = data[7]; // this is unset: you can interrogate it
if(d) // you can also do if(d.has_value())
  value = *d; // or value=d.value()
else
  std::cout<<" Value unset";
```

See also STL/Optional/

The variant<...> utility



You may want a variable able to represent values of different type. In C (and C++) we have **union**, but it is quite complex and unsafe. For this reason, std::variant has been introduced to deal with elements of different data types.

```
std::variant<double,std::string> var;
var="Hello";// It's holding a string
var = 10.5;// now holds a double
double c = std::get<double>(var); // c is now =10.5
// Run-time error: not currently holding a string!!
// std::string s = std::get<std::string>(var);
// But I can check!
   if(var.holds_alternative<std::string>())
   {...//it's a string
```

std:: visit

Variants becomes very powerful when used together with the ‘std::visit‘ facility, which indeed allow us to implement the visitor pattern. Better start with an example.
Let’s have a set of (non necessarily polymorphic) classes representing different geometric objects

```
 class Triangle{
     public:
     double area()const;
    ...
 }
  class Square{
    public:
    double area()const;
    ...
}
  class Point{
    public:
    // No area() method!
    ...
}
// etc. many others...
```

Let's define a visitor as a set of overloaded function operating on our set of polygons (note the use of an automatic function to identify a fallback action).

```
struct TotalArea {
   double totalArea=0;
   void operator()(Triangle const & x){totalArea+=x.area();}
   void operator()(Square const & x){totalArea+=x.area();}
   void operator()(Pentagon const & x){totalArea+=x.area();}
   //... for all other object with area()
   void operator()(auto const & x){}//fallback
}
```

I define a variant able to hold polygons

```
using GeoObj=std::variant<Point,Triangle,Square,Pentagon>;
```

Visiting a set of geometric objects

```
std::vector<GeoObj> listOfGeo; // a vector of variants
//.. I fill the vectos with various objects
//   Now I compute the total area of all objects
TotalArea area;
for (auto const & x:listOfGeo)
 std::visit(area,x);// I visit the variant
 std::cout<<"Total Area="<<area.totalArea;
```

The function `visit(visitor,variant)` automatically selects the correct overload depending on the actual type stored in the variant.

This way, we can implement a new type of dynamic polymorphism that does not need virtual methods.

There is much more...

In STL/Variant folder you find other examples, including using `std::visit` with two variants, and how to make a visitor on the fly using the `overloaded` utility given in Utilities/overloaded.hpp

A more elaborated example is found in the DesignPatterns/VisitorAndVariant/ folder.

std::any

C++17 has also introduced a type-safe container for single values of any copy constructible type.
You have to use std::any_cast<T> to cast a std::any variable to a variable of type T (exception if not possible).

```
#include <any>
...
std::any a = 1;
if (a.has_value())
  int k = std::any_cast<int>(a);
...
a=3.14; // now it holds a double
try
 {
   auto z=std::any_cast<int>(a); //fails!
  }
 catch (const std::bad_any_cast& e)
  {
    std::err << e.what() << '\n';
   }
```

Traversing and examining the filesystem

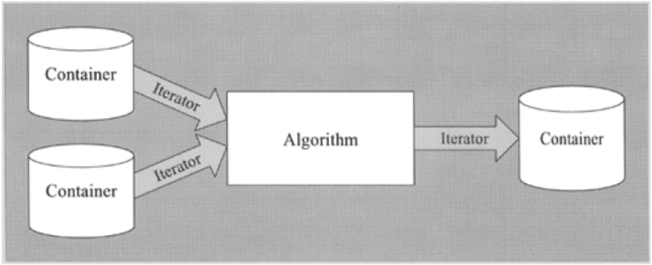
Often one needs to look for files in the filesystem, and maybe check if they exist, or create new directories, renaming files, etc. etc.

The utilities provided by the header <filesystem> of the Standard Library can be used to perform those tasks from within your program.

Have a look to STL/FileSystem/mainFilesystem.cpp for a simple example (but there is much more!).

Algorithms

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



For a full list you may look here for generic and here for numeric algorithms.

Note: C++20 has revised the concept of range and provides a new set of algorithms in the namespace std::ranges, with the same name of the old ones, but simpler to use and sometimes more powerful.

Type of algorithms

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

  ```
  It find(ForwardIt first, ForwardIt 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 position of equivalent elements (but 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 $c = a \cup b$.

Note: Remember that a std::set is already ordered!

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 );
//
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 passed 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>());
// The same with lambdas
auto product =
  std::accumulate(v.begin(), v.end(), 1,
    [](double a, double b){return a*b;});
auto r1 = std::inner_product(v.begin(), v.end(),
                                w.begin(), 0);
```

Numeric algorithm

In <numeric> we have other useful algorithms:

```
auto j= std::gcd(i, l);// greatest common divisor
auto l=std::lcm(i, l);// least common multiplier
auto l3=hypot(x,y,z); 2-norm  of (x,y,z)
auto l2=hypot(x,y); 2-norm  of (x,y)
// difference of adjacent elements (you can change operator)
// also with parallel execution
adjacent_difference(v.begin(),v.end(),o.begin());
```

and many more... Bessel function, elliptic integrals, etc. see see here.

An example: copy_if

```
std::vector<double> a;
// a is filled with values
std::vector<double> b; // an empty vector
std::copy_if(a.begin(),a.end(),std::back_inserter(b),
    [](double const & x) {std::abs(x)>1.e-10;});
```

It is equivalent to

```
for (auto x: a) {
 if (std::abs(x)>1.e-10) b.emplace_back(x);
 };
```

Note the se of back inserter to insert the element at the back of vector b.

And if I want to avoid unnecessary memory relocations?

You can use count_if

```
auto f = [](double const & x) {std::abs(x)>1.e-10;});
std::vector<double> b;
auto howmany=std::count_if(a.begin(),a.end(),f);
b.reserve(howmany);\\ still an empty vector
std::copy_if(a.begin(),a.end(),std::back_inserter(b),f);
```

Or,

```
auto f = [](double const & x) {std::abs(x)>1.e-10;});
auto howmany=std::count_if(a.begin(),a.end(),f);
b.resize(howmany);\\ a vector with howmany elements
std::copy_if(a.begin(),a.end(),b.begin(),f);
// here you do not need back_inserter
```

Why a standard algorithm?

Many standard algorithms can be implemented using a for loop. So what's the advantage? I start by saying that there is nothing wrong with the for-loop version. If you are happy with it, use it. Yet with standard algorithms

- You are more uniform with respect different containers type;
- The algorithm of the Standard Library may do certain optimizations if the contained elements have some characteristics;
- You have a parallel version for free (see next slides).

The parallel version

Here I cannot use back_inserrter

```
  #include <execution> // I need for the parallel version
auto f = [](double const & x) {std::abs(x)>1.e-10;});
auto howmany=count_if(std::execution::par,a.begin(),a.end(),f);
b.resize(howmany);
std::copy_if(std::execution::par,a.begin(),a.end(),b.begin(),f);
```

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
```

Parallel Algorithm

Several algorithms (and new ones) support now parallel execution:

```
#include <execution>
std::vector<double> coll;
...
// parallel for_each (and automatic lambda)
for_each(std::execution::par, coll.begin(), coll.end(),
[](auto& x) {return x*x;});
// sequential (no parallelization)
for_each(std::execution::seq, coll.begin(), coll.end(),
[](auto& x) {return x*x;});
```

You need to link with the libtbb library.

Execution policies

All algorithms that support parallel execution do so by taking an execution policy as first argument. We have 3 possibilities:

- std::execution::seq Sequential execution (no parallelization)
- std::execution::par Parallel sequenced execution
- std::execution::par_unseq Parallel unsequenced execution (vectorization)

The last execution policy is activated only if the hardware supports it. Parallel algorithms have been introduced with C++17. Many old algorithms have now a parallel version, some with restrictions.

See the example in STL/Reduce

It is up to you to make sure that the procedure is parallelizable.

You can have specialised version of the standard library that introduce other execution policies, for instance the NVIDIA extension for GPUs.

Transform_reduce

Another interesting algorithm is transform_reduce. It applies the map-reduce procedure: first the input data is modified elementwise by a given unary transformation, then the result is "reduced" to a scalar value by another transformation defined by a binary function object.
Here an example of its use to compute the $\|v\|_1$ of a given range (bigVector), with also parallel execution

```
auto absolute = [](double const &x) { return std::abs(x); };
double l1norm=
std::transform_reduce(
std::execution::par,// execution policy
std::begin(bigVector), // start
std::end(bigVector),   // end
0.0,                   // initial value
std::plus{},           // binary operator
absolute); // the lambda as unary operator
```

A list of 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

and then all operation on sorted ranges (union, intersection etc). A full list here and here for numerical functions and algorithms.

A final comment on parallel algorithms

Most stl algorithms now support parallel execution (via multi-threading). Some have maintained the pre-C++17 name, and to have parallel execution you need to add only an extra argument, some of them poses some constraints on the category of iterators you may use in the parallel version (but not very relevant).

Some have been added instead: for instance reduce is the parallel version of inner_product.

In conclusion, the addition of parallel algorithm may make some parallelizion cheap, so use them. However: 1) they implement (so far) only multithreading, you will see in the lectures on parallellisation more advanced and powerful techniques, 2) be careful on data race, make sure that you can parallelize the procedure. C++ provides also tools to control parallel execution finely (mutexes etc.), but their use is complex and beyond the scope of this course.

Inserters

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

```
//Insert at the back (only for sequential containers)
back_inserter (Container& x);
//insert in the front (only for sequential containers)
front_inserter(Container& x);
//Insert after indicated position
inserter(Container & x, It position);
```

Example:

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

The computational cost depends on the type of container!

A example of use of an inserter

Several algorithms require to write the output to a (non const) range indicated by the iterator to its beginning. Without inserters it would be impossible to use them on a non-sequential container, or on a sequential container of insufficient size.

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

You need an inserter:

```
std::copy(a.begin(),a.end(),
               std::inserter(b,b.begin())); //OK!
```

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

Traversing and examining the filesystem

Often one needs to look for files in the filesystem, and maybe check if they exist, or create new directories, renaming files, etc. etc.

The utilities provided by the header <filesystem> of the Standard Library can be used to perform those tasks from within your program.

Have a look to STL/FileSystem/mainFilesystem.cpp for a simple example (but there is much more!).

What next?

This is the end of this overview. For some aspects of the STL we have dedicated lectures.