

# STL: Standard Template Library

P. Ronchese

Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

## A set of tools

The STL is a set a of tools based on a system of class templates, satisfying efficiently the most common needs in C++ programs.

- Strings
- Containers
  - Sequences
  - Associations
- Algorithms

The STL provide a lot of objects, only a few of them will be described.  
Documentation: <http://www.sgi.com/tech/stl/>

# Strings

STL strings (`std::string`) are objects providing several functionalities not available in C-strings. To use them an `#include <string>` is required.

- Copy
- Concatenation
- Insertion
- Comparison
- Length determination
- Substring find

Whenever C-strings are needed, they can be obtained from STL-strings:

```
const char* c = s.c_str();
```



## String copy and concatenate

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    // create a string
    string s("abcdefghi");
    cout << s << endl;
    // copy the string
    string t=s;
    cout << t << endl;
    // concatenate two strings
    string u=s+"1234";
    cout << u << endl;
    return 0;
}
```

## Substring insertion and extraction

```
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char* argv[]) {
    // create a string
    string s("abcdefghi");
    cout << s << endl;
    // insert a substring after 3 characters
    s.insert(3, "zyx");
    cout << s << endl;
    // extract a substring after 4 characters
    string t=s.substr(4,2);
    cout << t << endl;
    return 0;
}
```

## String comparison and inspection

```
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char* argv[]) {
    string s(argv[1]);
    // string length
    cout << s.length() << endl;
    // compare strings
    if(s=="abcdef")cout << "equal" << endl;
    // look for a substring
    cout << s.find("cde") << endl;
    cout << (s.find("cde")==string::npos?
            "not ":"")<< "found" << endl;
    return 0;
}
```

## Sequences and associations

STL provides several containers, with different properties.

- Sequences: contents are stored in sequential order.
  - Queues:
    - data are contained in contiguous memory locations,
    - random access is supported,
    - linear time to insert/remove data in the middle  
(i.e. the time to insert an element in a container with 1000 elements is 10 times longer than the time to insert an element in a container with 100 elements).
  - Lists:
    - data are linked to the previous/next ones,
    - random access is not supported,
    - constant time to insert/remove data in the middle  
(i.e. the time to insert an element in a container does not depend on the size of the container itself).
- Associations: content are stored with a key.  
They will be shown later.

## Sequences: creation and filling

The most used sequences in STL (`std::vector`) are array-like objects.

To use them an `#include <vector>` is required.

```
unsigned int n=10; // create a vector
vector<int> v(n); // with 10 elements
unsigned int i;
for(i=0;i<n;++i)v[i]=2*i;
v.push_back(987); // add at the end
for(i=0;i<v.size();++i)cout << i << " "
                        << v[i] << endl;
vector<int>* pv=&v;
for(i=0;i<pv->size();++i)cout << i << " "
                        << pv->at(i) << endl;
```

Contrarily to native arrays,  
vectors can be copied as whole objects.

## Initializer lists

### C++11 only

Sequences can be initialized with lists, as native array.

```
vector<int> v {20,18,16,14,12,10,8,6,4,2};
unsigned int i;
for(i=0;i<v.size();++i) cout << i << " "
                        << v[i] << endl;
```

### C++11 only

Fixed-size arrays can be created  
(an `#include <array>` is required).

```
array<int,10> a {20,18,16,14,12,10,8,6,4,2};
unsigned int i;
for(i=0;i<a.size();++i) cout << i << " "
                        << a[i] << endl;
```

## Memory management

Elements are stored sequentially:  
insertion of elements (eventually) requires  
an element shift and/or a memory reallocation.

- Constant time to insert/remove elements at the end.
- Linear time to insert/remove elements in the middle.
- Automatic reallocation triggered when all memory locations are used: available memory is doubled at each time.
- `v.reserve(n)` forces the allocation of memory for `n` elements, without changing the visible `size`.
- `v.capacity()` returns the available memory.
- All iterators are invalidated after a memory reallocation.

## vector functions and operators: set content

- **Constructors:**
  - `vector<T> v` : create an empty vector
  - `vector<T> v(n)` : create a vector with  $n$  elements
  - `vector<T> v(n, t)` :  
create a vector with  $n$  elements initialized at  $t$
  - `vector<T> v(u)` : create a vector by copying vector  $u$
- `v[i]` : reference to element at position  $i$   
(unchecked range)
- `v.at(i)` : reference to element at position  $i$   
(checked range)
- `v.push_back(t)` : add an element at the end
- `v.pop_back()` : remove an element at the end
- `v.reserve(n)` : allocate memory for  $n$  elements
- `v.resize(n)` , `v.resize(n, t)` : set the number of elements by adding or removing elements at the end and leaving unchanged the others
- `v.clear()` : remove all elements

## vector functions and operators: get

- `v.size()` : number of elements
- `v.capacity()` : available memory
- `v.empty()` : true if the number of elements is 0
- `v.front()` : reference to the first element
- `v.back()` : reference to the last element
- `v[i]` : reference to element at position `i`  
(unchecked range)
- `v.at(i)` : reference to element at position `i`  
(checked range)

## Direct object creation

### C++11 only

Composite objects can be created inside the `vector` directly from the parameters used by the constructors.

```
vector<Point> v;  
v.emplace_back( 5.7, 6.2 );
```

## Navigation across containers

Access to containers elements can be obtained through “smart pointers” allowing the scan of the container in both directions: `iterators`.

- `vector<T>::iterator i=v.begin()` :  
an iterator pointing to the first element
- `vector<T>::iterator i=v.end()` :  
an iterator pointing to the next-to-last element
- `*i` : reference to the pointed element
- `++i , i++` : move the iterator to the next element
- `--i , i--` : move the iterator to the previous element

```
vector<int> v;
...
vector<int>::iterator it=v.begin();
vector<int>::iterator ie=v.end();
while(it<ie) cout << *it++ << endl;
```

## Iterators arithmetic

- `vector<...>::iterator ii=it+n:`  
create an iterator `ii` pointing `n` positions after `it`
- `vector<...>::iterator ii=it-n:`  
create an iterator `ii` pointing `n` positions before `it`
- `it+=n, it-=n:`  
move forward or backward of `n` positions
- `int d = distance(ip, in):`  
compute how much `ip` must be advanced to reach `in`
- `advance(it, n):`  
move `it` forward of `n` positions

## Operations with sequences and iterators

- `v.insert(it, t)` :  
insert the element `t` at the position `it`
- `v.insert(it, ib, ie)` :  
insert the elements pointed by `[ib, ie)` at the position `it`
- `v.erase(it)` , `v.erase(ib, ie)` :  
erase the element(s) pointed by `it` or `[ib, ie)`
- after an insertion or erase, iterators pointing to elements of the sequence are invalidated
- `const vector<T> v` :  
a vector whose size and elements cannot be modified
- `vector<T>::const_iterator` :  
analogous to “pointer to const”
- `vector<T>::reverse_iterator` :  
allow the scan in the backward direction
- `v.rbegin()` , `v.rend()` :  
begin and end of the reversed vector

## Double-entry containers

Insertion and removal at both ends is supported by `std::deque` objects.

To use them an `#include <deque>` is required.

```
unsigned int n=10; // create a deque
deque<int> v(n); // with 10 elements
unsigned int i;
for(i=0;i<n;++i)v[i]=2*i;
v.push_front(123); // add at the beginning
for(i=0;i<v.size();++i)cout << i << " "
                                << v[i] << endl;
```

- `v.push_front(t)`: add an element at the beginning
- `v.pop_front()`: remove an element at the beginning
- constant time to insert/remove elements at the beginning and the end
- linear time to insert/remove elements in the middle
- no `reserve` and `capacity` functions

# Lists

A “smooth” insertion and removal of elements is supported by `std::list` objects.

To use them an `#include <list>` is required.

- Constant time for insertion and removal at any point.
- Linear time for `size()` function.
- Constant time for `empty()` function.
- No random access, i.e. no `[]` operator.
- Iterators are not invalidated by insertion or removal, but the iterator(s) pointing to the erased element(s).
- No `distance` and `advance` functions.
- No `+`, `+=`, `-`, `-=` operators for `list::iterator`

## Sort and search

STL provides useful algorithms acting over containers, accessed through iterators.

To use them an `#include <algorithm>` is required.

- Sort
- Binary search

Arrays and pointers can be used  
in place of containers and iterators

## Sort by operator <

```
sort(first, last);  
sort elements in ascending order
```

- Elements are compared by using the < operator.
- Iterators to the first and next-to-last element taken as arguments.
- $\mathcal{O}(N \log(N))$  comparisons are done, where  $N = \text{last} - \text{first}$ .

```
vector<int>v;  
...  
sort(v.begin(), v.end());
```

## Sort by global function

Elements can be compared using a global function

```
sort (first, last, comp);
```

```
bool comp(const Vector2D* v1,
          const Vector2D* v2) {
    float q1=pow(v1->getX(),2)+
           pow(v1->getY(),2);
    float q2=pow(v2->getX(),2)+
           pow(v2->getY(),2);
    return q1 < q2;
}
...
vector<Vector2D*>v;
...
sort (v.begin(), v.end(), comp);
```

## Sort by function object

Elements can be compared  
operator () in a function object

```
class Comp {
public: // accessible by all functions
    bool operator()(const Vector2D* v1,
                    const Vector2D* vr) {
        float ql=pow(v1->getX(),2)+
              pow(v1->getY(),2);
        float qr=pow(vr->getX(),2)+
              pow(vr->getY(),2);
        return ql < qr;
    }
};

...
Comp comp;
sort(v.begin(),v.end(),comp);
```

## Sort by lambda functions

### C++11 only

Simple functions to be used in algorithms  
can be coded directly where needed

```
sort (v.begin(), v.end(),
      [] (const Vector2D* v1,
          const Vector2D* v2) {
          return (pow(v1->getX(), 2) +
                  pow(v1->getY(), 2)) <
                 (pow(v2->getX(), 2) +
                  pow(v2->getY(), 2));
      });
```

Variables in the environment can be “captured”:

- [] capture nothing
- [&] capture all by reference
- [=] capture all by value
- [=, &i] capture all by value, but i by reference

## Function wrappers

### C++11 only

An uniform interface to wrap function  
is given by `std::function` objects.

To use them an `#include <functional>` is required.

```
function<bool (Vector2D* v1, Vector2D* v2)>
    comp([] (const Vector2D* v1,
            const Vector2D* v2) {
        return (pow(v1->getX(), 2) +
                pow(v1->getY(), 2)) <
                (pow(v2->getX(), 2) +
                pow(v2->getY(), 2));
    });
sort(v.begin(), v.end(), comp);
```

## Binary search

Binary-search functions look for an element in a container and return the corresponding/nearest iterator.

- `iter=lower_bound(first, last, i)` : the first element such that it and the following ones are not smaller than `i`
- `iter=upper_bound(first, last, i)` : the first element such that it and the following ones are bigger than `i`
- `iter=lower_bound(first, last, i, comp)` and `iter=upper_bound(first, last, i, comp)` : perform the search by using the `comp` function object in place of the `<` operator.
- At most  $N \log(N) + 1$  comparisons are done.

# Associations

Associative containers support efficient retrieval of elements based on keys.

- `sets`: value coincident with key
- `maps`: association of a value with each key
- No two elements have the same key (but for `multiset` and `multimap`)
- Insertion and erase of elements do not invalidate iterators
- Need a sorting algorithm (< operator or compare object)

# Sets

The simplest associative container in STL (`std::set`) stores objects of type `Key`.

To use them an `#include <set>` is required.

- `set<T> s` : create an empty set
- `set<T,C> s(comp)` : create an empty set using `comp` as function object to compare keys
- `s.insert(x)` : insert the element `x`
- `s.insert(first, last)` : insert the elements pointed by a range of iterators
- `s.erase(x)` : erase the element `x`
- `s.erase(it)` , `s.erase(ib, ie)` : erase the element(s) pointed by `it` or `[ib, ie)`
- `size()` : return the number of elements

## Elements insertion and search

Elements are looked for by the function `find`

- if `s` contains `x`  
return an iterator pointing to the corresponding element
- if `s` does not contain `x`  
return `s.end()`

```
set<int> s;  
...  
int i;  
cin>>i;  
set<int>::const_iterator it=s.find(i);  
set<int>::const_iterator ie=s.end();  
if(it!=ie)cout << *it << endl;  
else      cout << "not found" << endl;
```

# Maps

The most used associative container in STL (`std::map`) stores pairs of objects of type `Key` and `Data`.

To use them an `#include <map>` is required.

- `map<Key, Data> m` : create an empty map
- `map<Key, Data, Comp> m(comp)` : create an empty map using `comp` as function object to compare keys
- each element is a `std::pair<const Key, Data>` (or `map<Key, Data>::value_type`) having two members:
  - first with type `Key` (or `map<Key, Data>::key_type`)
  - second with type `Data` (or `map<Key, Data>::mapped_type`)
- `m.insert(make_pair(k, x))` : insert `x` with key `k`
- `m.erase(k)` : erase the element whose key is `k`
- `m.erase(it)` , `m.erase(ib, ie)` : erase the element(s) pointed by `it` or `[ib, ie)`
- `size()` return the number of elements

## Elements insertion and search

Elements are looked for by the function `find`

- if `m` contains an element with key `k`  
return an iterator pointing to it
- if `m` does not contains an element with key `k`  
return `m.end()`

```
map<string,int> m;
...
string s;
cin>>s;
map<string,int>::const_iterator it=m.find(s);
map<string,int>::const_iterator ie=m.end();
if(it!=ie)cout << it->second << endl;
else      cout << "not found" << endl;
```

## Elements access via [] operator

The `m[k]` expression:

- returns a reference to the element whose key is `k`, if it does exist,
- otherwise it creates it with a default and returns it.

`operator[]` is non-const,  
and cannot be used for const maps

```
map<string,int> m;
...
string s;
cin>>s;
cout << m[s] << endl;
const map<string,int> c(m);
cout << c[s] << endl; // ERROR
```

## Elements access via `at` function

The `m.at(k)` expression:

- returns a reference to the element whose key is `k`, if it does exist,
- otherwise it throws an exception.

`at` function can be used for `const` maps

```
map<string,int> m;
...
string s;
cin>>s;
cout << m.at(s) << endl;
const map<string,int> c(m);
cout << c.at(s) << endl;
```

## Object ownership

A “smart pointer” is an object behaving as a pointer and holding the ownership of the pointed object.

- The pointed object is automatically deleted when the smart pointer owning it is destroyed.
- If a smart pointer can be copied the ownership must be defined.

### C++98/03 - deprecated

An `auto_ptr<T>` object holds a pointer to an object of type `T` and, when copied, passes the ownership to the new copy.

An `#include <memory>` is required to use it.

## Different smart pointers

The C++11 STL allows the creation of different smart pointers, with different properties.

### C++11 only

- An `unique_ptr<T>` object cannot be copied, but it can be “moved” (i.e. copied from a temporary object).
- A `shared_ptr<T>` can be copied; the pointed objects is destroyed when the last pointer is destroyed.