The C++ Standard Template Library

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

What is the STL?

The STL (Standard Template Library) is a set of template classes and functions composed of:

- containers data structures, e.g. `std::vector`;
- iterators for container data structure traversal;
- algorithms for common operations, e.g. `std::sort` for sorting

STL Documentation:

- Containers library
- Iterators library
- Algorithms library

A few code examples are available online at GitHub.

STL container types

- **Sequence containers**: `vector`, `list`, `deque` ...
- **Associative containers**: `map`, `set`, `multiset`, `multimap`
- **Container adapters**: common data structures, that internally use other containers for their implementation: `stack`, `queue`, ...

Advantages and disadvantages

+ STL provides common data structures and associated algorithms that can be used with most programs. Programmers do not have to "re-invent the wheel", and in particular deal with use of dynamic memory allocation and pointers used internally by STL containers; special care still has to be taken when container elements if they are pointers to dynamically allocated memory.

– Somewhat irregular design. For example, containers with approximate functionality have different interfaces for no apparent reason (e.g. `list` and `vector`), and are not not "glued" by a class hierarchy. Error messages during compilation are often difficult to interpret.

Containers - some common operations

Function	Meaning
Default constructor	Creates empty container
Copy constructor	Creates container copying elements from another container.
Construtor using “initializer list”	Creates container with elements given by an initializer list (since C++ 11).
Destructor	Frees dynamically allocated memory associated to the container.
`size`	Yields number of elements in the container.
`empty()`	Indicates if container is empty.
`operator=`	Assignment operator.
`operator==`	Equality test operator.

Containers - some common operations

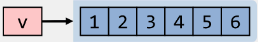
Function	Meaning
`at`, `operator[]`	Accesses element (`const` or mutable references).
`insert`	Inserts element.
`erase`	Erases element.
`clear`	Erases all elements.
`begin`	Get iterator for the first element the container,for “begin-to-end” iteration according to some order.
`end`	Get iterator marking the “end” of the container.
`rbegin`	Get iterator for the last element of the container, for “end-to-begin” reverse iteration.
`rend`	Get iterator marking the “end” of the container in a reverse

Sequence containers

Class	Implementation type
`vector`	Dynamic array - elements stored contiguously in an array whose size is dynamically adjusted.
`list`	Doubly linked list - elements stored in a list of doubly linked nodes (i.e., with a pointer to previous and next nodes).
`forward_list`	Simply linked list - elements stored in a list of simply linked nodes (i.e., only with a pointer to the next node).
`deque`	Vector of vectors (double indirection scheme).

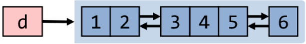
Sequence containers (cont.)

`vector<T>`



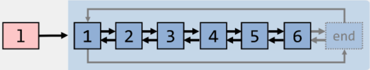
dynamic contiguous array; amortized $O(1)$ growth strategy; C++'s “default” container

`deque<T>`



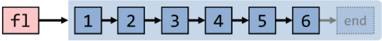
double-ended queue; fast insert/erase at both ends

`list<T>`



doubly-linked list; $O(1)$ insert, erase & splicing; in practice often slower than vector

`forward_list<T>`



singly-linked list; $O(1)$ insert, erase & splicing; needs less memory than `list`; in practice often slower than vector

Image from hackingcpp.com

Sequence containers - example

We have already seen several examples with `vector`. The use of `list` or `deque` is analogous in regard to the most basic member functions:

```
#include <deque>
....
deque<string> dq {"a", "b", "c", "d", "e"};
dq.push_back("f");
dq.push_front("g");

cout << dq.size() << " [";
for (size_t i = 0; i < dq.size(); i++)
  cout << ' ' << dq.at(i);
cout << " ]" << '\n';
```

Output:

```
7 [g a b c d e f]
```

Sequence containers - front and end operations

Function	Description
`front`	Access first element
`back`	Access last element (not defined for `forward_list`).
`push_back`	Adds element to the end of the sequence (not defined for `forward_list`).
`pop_back`	Removes element from the end of the sequence (not defined for `forward_list`).
`push_front`	Adds element to start of sequence (not defined for `vector`).
`pop_front`	Removes element from the beginning of the sequence (not defined for `vector`).

Algorithmic complexity

Operation	`vector`	`list`[1]	`deque`
`pop_front` / `push_front`	N/A	$O(1)$	$O(1)$
`pop_back` / `push_back`	$O(1)$	$O(1)$	$O(1)$
`at` / `operator[]`	$O(1)$	$O(n)$	$O(1)$
`insert`/`remove` [2]	$O(n)$	$O(1)$	$O(n)$

[1] `forward_list` similar but does not define `pop_back` or `push_back`.

[2] `insert` and `remove` work via iterator to position.

$O(1)$ - **constant time** (or amortized constant in some cases): computational effort is independent of the number of elements in the container. Relocation time, if necessary (as in `vector`), is linear but amortized over subsequent calls.

$O(n)$ - **linear time**: the computational effort is proportional to the number of elements (n) in the container.

Iterators

An **iterator** is an object that allows you to iterate elements of a container. They are used to traverse a container but also in association with container member functions (e.g. `erase` and `insert`) or STL algorithms (e.g. `std::sort`).

An iterator has an associated notion of position within the container it refers to, and the corresponding element if the position is valid. For an iterator `itr` `*itr` accesses the current element referred to by the iterator and `itr++` advances the iterator to the next position.

Iterators are used to iterate a container `c` by an iteration from `c.begin()` to `c.end()`:

```
for (iterator_type itr = c.begin();
     itr != c.end();
     itr++) {
   // Do something using *itr to get the element
   ...
}
```

Iterators - an example

Calculating the sum of `int` elements in a `vector`:

```
vector<int> c { 2, -5, 6, 7, -9 };
int sum = 0;
for(vector<int>::iterator itr = c.begin();
    itr != c.end();
    itr++) {
 sum += *itr;
}
cout << sum << '\n';
```

- Initial iterator obtained with `c.begin()`.
- Iteration ends by comparing `itr` with `c.end()` (iterator without associated element that "marks" the end).
- Element associated with the iterator obtained with the dereferencing operator: `*itr`.
- Iterator advances to next position with `itr++`.

Use of auto

It is common to employ `auto` to avoid declaring the iterator type, which is typically quite verbose.

In the previous example, the iterator type is `vector<int>::iterator`, but using `auto` is more succinct and does not hurt the readability of the code.

```
vector<int> c { 2, -5, 6, 7, -9 };
int sum = 0;
for( auto itr = c.begin();
     itr != c.end();
     itr++) {
   sum += *itr;
}
cout << sum << '\n';
```

Iterators and range-based `for` loops

Range-based `for` loops are often more convenient than the explicit use of iterators for loops. The previous example can be expressed instead more simply as:

```
vector<int> c { 2, -5, 6, 7, -9 };
int sum = 0;
for( int v : c) {
  sum += v;
}
cout << sum << '\n';
```

But in fact a range-based `for` loops corresponds to an implicit use of iterators.

Iterators and range-based `for` loops (cont.)

A container must define `begin()` and `end()` iterators for range-based for loops to be defined.

When we write

```
for (type v : c) {
    ...
}
```

this is implicitly equivalent to

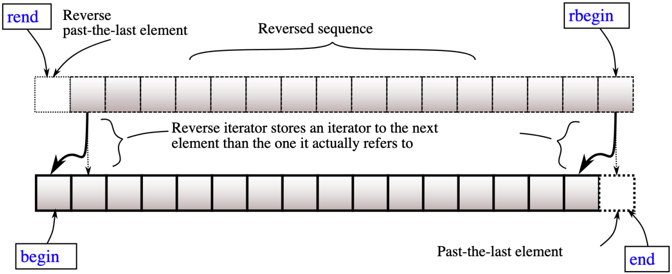
```
for (itr = c.begin(); itr != c.end(); itr++) {
    type v = *itr;
    ...
}
```

Reverse iterators

Reverse iterators allow element traversal in the reverse order of a standard iterator. The member functions `rbegin()` and `rend()` must be employed for this purpose.

```
for( auto itr = c.rbegin();
     itr != c.rend();
     itr++) { ... }
```

Illustration for `std::vector` (image from cplusplus.com):



Reverse iterators (cont.)

Example:

```
vector<char> c { 'R', 'O', 'M', 'A' };
for( auto itr = c.rbegin();
     itr != c.rend();
     itr++) {
  cout << *itr;
}
```

Output:

```
AMOR
```

Iterators and container changes

Iterators can also be used to update the contents of a container.

Example - increment of all values in a vector of integers:

```
vector<int> c { 2, -5, 6, 7, -9 };
for( auto itr = c.begin();
     itr != c.end();
     itr++) {
   *itr = *itr + 1;
}
// values at the end: 3, -4, 7, 8, -8
```

Iterators and container changes (cont.)

For sequence containers we can use iterators to insert and erase elements through `insert` and `erase`. This can happen outside a loop:

```
vector<int> c {  10, 20, 30, 40, 50, 60 };
...
// Remove 3rd element (30)
c.erase(c.begin() + 2);
// Inserts 123 before the penultimate position (50)
c.insert(c.end() - 2, 123);
for (int v : c) cout << v << ' ';
```

Output:

```
10 20 40 123 50 60
```

Insertions and removals (cont.)

Insertions and removals can happen during an iteration:

```
list<int> c { 2, 0, 1, 0, 2 };
auto itr = c.begin();
while (itr != c.end()) {
  if (*itr == 0) itr = c.erase(itr);
  else itr++;
}
// At the end c has elements: 2, 1, 2
```

`insert` and `erase` make the given iterator invalid or others that may exist on the container, but return a new one that can be used to continue (in `erase` above: the one in the following position).

Index-based iteration vs. iterators

If `c` is a `std::vector` then

```
for( auto itr = c.begin(); itr != c.end(); itr++) {
  sum += *itr;
}
```

has the same logical effect and efficiency of

```
for( auto i = 0;  i < c.size(); i++) {
  sum += c.at(i); // or c[i]
}
```

since `at` / `operator[]` are $O(1)$.

However, if `c` is a `std::list` then `at` / `operator[]` are $O(n)$, while using the iterator is $O(1)$ (faster).

Types of iterator - summary

- **Forward iterators**: only can move forward one position;
  - e.g. `forward_list` iterators
- **Bi-directional iterators** - also move backwards
  - e.g. `list` iterators
- **Random access iterators**: move backward or forward an arbitrary number of elements
  - e.g. `vector`, `queue` iterators

Overloaded operators for `iterator`

Operators	Meaning
`*itr`	Gets reference to element.
`itr1 = itr2`	Assignment.
`itr1 == itr2`	Comparison.
`itr1 != itr2`	
`itr1 < itr2`	
`++itr`	Advance to next position.
`itr++`	
`--itr`	Returns to previous position (bidirectional iterators)
`itr--`	
`itr += n`	Moves `n` positions (random access iterators).
`itr -= n`	
`itr + n`	
`itr - n`	

STL algorithms - overview

STL algorithms:

- set of template functions defined in the `<algorithm>` header;
- functions implement common functionalities such as sorting, searching, copying elements, filling containers...
- often take a container range defined by a start and end iterators as arguments, rather than containers directly.

Overview of algorithms

Functions	Description
`sort`	Ordering (covered next)
`find`	Linear search
`binary_search`	Binary search
`for_each`	Iteration
`transform`	Transformation of values
`unique`	Duplicate removal
`min`, `max`	Obtaining minimum and maximum
...	... see many more online ...

Example algorithm - `std::sort`

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

`std::sort` takes as arguments two iterators resp. for the beginning and end of elements to iterate. First variant orders elements according to operator `<` defined for `T`, and second variant takes an argument for specify the comparison function.

Iterators must be of the "random access" type, e.g. iterators of a `vector` or `deque`; for `list` we can use the `list::sort` member function (which does not exist for `vector` and `deque`).

Example algorithm - `std::sort` (cont.)

Comparison function must have 2 arguments `a` and `b` of the type used with the “template” function and return a `bool` value indicating whether `a` must precede `b`.

```
bool increasing_order(int a, int b)
  { return a < b; }
bool decreasing_order(int a, int b)
  {  return a > b; }
...
vector<int> v { 1, 7, 3, -1, 9, 1, 10 };
sort(v.begin(), v.end()); // (1)
sort(v.begin(), v.end(), decreasing_order); // (2)
sort(v.begin(), v.end(), increasing_order); // (3)
```

Example algorithm - `std::sort` (cont.)

For the previous example:

```
bool increasing_order(int a, int b)
  { return a < b; }
bool decreasing_order(int a, int b)
  {  return a > b; }
...
vector<int> v { 1, 7, 3, -1, 9, 1, 10 };
sort(v.begin(), v.end()); // (1)
sort(v.begin(), v.end(), decreasing_order); // (2)
sort(v.begin(), v.end(), increasing_order); // (3)
```

we get in cases `(1)`, `(2)` and `(3)`

```
(1) [ -1 1 1 3 7 9 10 ]
(2) [ 10 9 7 3 1 1 -1 ]
(3) [ -1 1 1 3 7 9 10 ]
```

Example algorithm - `std::sort` (cont.)

The range for an algorithm function can refer to a sub-section of a container.

For

```
vector<int> v { 1, 7, 3, -1,  9, 1, 10 };
sort(v.begin() + 1, v.end() - 1, decreasing_order);
for (int x : v ) cout << x << ' ';
```

elements at positions `0` and `v.size() - 1` remain unchanged ...

```
1 9 7 3 1 -1 10
```

Example algorithm - `std::sort` (cont.)

```
struct Date { int day, month, year; };
bool by_year(const Date& a, const Date& b) {
    return a.year < b.year;
}
...
vector<Date> vec { ... };
sort(vec.begin(), vec.end(), by_year);
```

For elements

```
[ 1/7/2021 30/12/2020 30/11/2020 14/5/2020
  14/5/2021 12/5/2020 13/5/2020 ]
```

the `by_year` ordering yields

```
[ 30/12/2020 30/11/2020 14/5/2020 12/5/2020
  13/5/2020 1/7/2021 14/5/2021 ]
```

(the relative order of dates with the same year is maintained)

Example algorithm - `std::sort` (cont.)

Considering now ...

```
bool by_year_month_and_day(const Date& a, const Date& b) {
    if (a.year != b.year) return a.year < b.year;
    if (a.month != b.month) return a.month < b.month;
    return a.day < b.day;
}
```

For elements

```
[ 1/7/2021 30/12/2020 30/11/2020 14/5/2020
  14/5/2021 12/5/2020 13/5/2020 ]
```

the `by_year_month_and_day` ordering yields

```
[ 12/5/2020 13/5/2020 14/5/2020 30/11/2020
  30/12/2020 14/5/2021 1/7/2021 ]
```

Associative containers

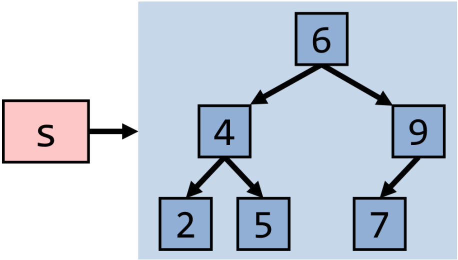
STL provides implementations for:

- **Sets** - `set`, `unordered_set`
- **Maps** (also called dictionaries): set of key-value pairs where keys are unique values - `map`, `unordered_map`
- **Multi-sets and multi-maps** - sets/maps with repetition of elements/keys - `multi_set`, `multi_map`, `unordered_multi_set`, `unordered_multi_map`

Ordered implementations

`set`, `map`, `multi_map`, `multi_set`

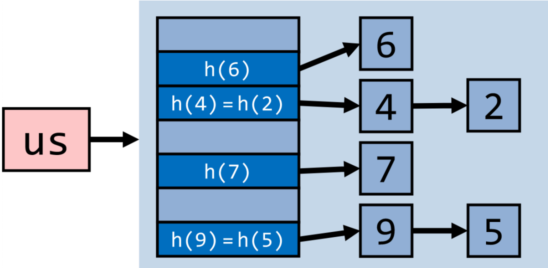
- elements have an associated **order**;
- implementations based on **“Red-Black” binary search trees**
- element search, insertion and removal has **logarithmic** complexity - $O(\log n)$



Unordered implementations

`unordered_set`, `unordered_map`, `unordered_multi_map`, `unordered_multi_set`

- elements have **no order**;
- implementation based on **hash tables**
- time for searching, inserting and removing elements is **constant** - $O(1)$



Sets - elementary functions

`set`, `unordered_set`

Operation	Description
`insert(v)`	Inserts element `v` (no effect if `v` is already in the set).
`erase(v)`	Removes element `v` (no effect if `v` is not in the set).
`find(v)`	Checks whether element `v` is in the set (returns iterator pointing to element).

Sets - elementary functions (cont.)

```
set<int> s; // or unordered_set<int>
s.insert(1);
s.insert(2);
s.insert(3);
s.erase(2);
for (int i = 0; i <= 3; i++)
  if (s.find(i) != s.end())
    cout << i << " is in the set" << '\n';
  else
    cout << i << " is not in the set" << '\n';
```

```
0 is not in the set
1 is in the set
2 is not in the set
3 is in the set
```

set vs. unordered_set

set follows an ordering criterium associated to store and traverse the elements.

For instance

```
set<string> a
  {  "fcup", "prog", "feup", "leic" };
for (auto v : a ) { cout << v << ' ';
...
```

will print, in accordance to the lexicographical order of strings, the following:

```
fcup feup leic prog
```

The output could follow any order for the elements in the case of unordered_set.

set vs. unordered_set (cont.)

- set<T> requires that operator < is defined for T
  - two elements a and b are considered equivalent if ! (a < b) && ! (b < a)
  - comparison function can be given via constructor alternatively
- unordered_set<T> requires that the == operator is defined for T to test equality of elements and also a hashing function
  - for primitive types and classes like std::string support for "hashing" is already given via std::hash

Maps

`map`, `unordered_map`

Operation	Description
`insert({k,v})`	Inserts key-value pair `(k,v)` (no effect if there is already a pair with key `k`).
`erase(k)`	Removes pair with key `k` (no effect if key `k` does not exist in the map)
`find(k)`	Checks if an entry with key `k` exists (returns iterator)
`at(k)`	Gets value for key `k` (which must exist).
`operator[](k)`	Gets reference to value for key `k`, creating association if it does not exist.

Map example

Executing ...

```
map<string, int> m;
m.insert({"a", 1});
m.insert(pair<string,int>("b", 2));
m.insert({"a", 3}); // no effect
m.at("a") ++; // update
m.erase("b"); // removal
m["c"] = 4; // implicit insertion
m["c"] += m["a"]; // update
```

we get `(a,2)` and `(c,6)` at the end. Note that key `b` is defined but erased subsequently.

Map example (cont.)

Previous example, step by step:

Step	Map contents
Initially	`{ }` (empty)
`m.insert({"a", 1})`	`{ (a,1) }`
`m.insert({"b", 2})`	`{ (a,1), (b,2) }`
`m.insert({"a", 3})`	`{ (a,1), (b,2) }` (unchanged)
`m.at("a") ++`	`{ (a,2), (b,2) }`
`m.erase("b")`	`{ (a,2) }`
`m["c"] = 4`	`{ (a,2), (c,4) }`
`m["c"] += m["a"]`	`{ (a,2), (c,6) }`

Map example (cont.)

A map can be iterated to access key-value pairs represented via `std::pair`. In continuation of the previous example, the following code

```
for (auto itr = m.begin(); itr != m.end(); itr++)
  cout << (*itr).first << " --> "
       << (*itr).second << '\n';
```

or, in an equivalent and simpler way, the range-based for loop ...

```
for (auto kv: m)
  cout << kv.first << " --> "
       << kv.second << '\n';
```

will give us the following output

```
a --> 2
c --> 6
```

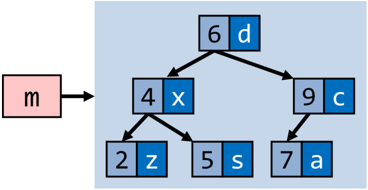
`map` vs. `unordered_map`

The differences between `map` and `unordered_map` are similar to those between `set` and `unordered_set`:

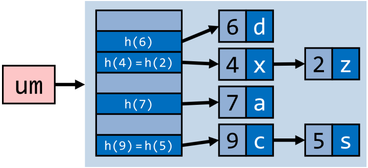
- `map` guarantees an ordered iteration (by key values), `unordered_map` does not.
- Elementary operations are $O(\log n)$ for `map`, and $O(1)$ for `unordered_map`
- `map<K,V>` requires `<` operator defined for `K` or a comparison function, while `unordered_map` requires `==` operator for `K` and an hash function for `K`.

map vs. unordered_map (cont.)

map



unordered_map



Images from hackingcpp.com

Multi-sets

`multiset` and `unordered_multiset` support multi-sets, where we can have multiple instances of the same element. The `count` member function returns the number of instances for a given element. Member functions are otherwise similar to `set` and `unordered_set`.

```
multiset<int> ms;
for (int i = 1; i <= 3; i++)
  for (int j = 1; j <= i; j++) ms.insert(i);
for (auto v : ms)
  cout << v << " count: " << ms.count(v) << '\n';
```

Output:

```
1 count: 1
2 count: 2
2 count: 2
3 count: 3
3 count: 3
3 count: 3
```

Multi-maps

`multimap` and `unordered_multimap` are collections of key-value pairs in which the same key can be repeated several times.

```
multimap<int,int> ms;
for (int i = 1; i <=3; i++)
  for (int j = 1; j <= i; j++) ms.insert({i, i * 10 + j});
for (auto v : ms)
  cout << v.first << " --> " << v.second
            <<  " (count: "<< ms.count(v.first)
            << ")" << '\n';
```

Output:

```
1 --> 11 (count: 1)
2 --> 21 (count: 2)
2 --> 22 (count: 2)
3 --> 31 (count: 3)
3 --> 32 (count: 3)
3 --> 33 (count: 3)
```