Introduction to the STL

Eric Elfving

1 Containers
2 Iterators
3 Algorithms

There are three groups of containers in the STL:

Sequence containers Stores values in sequence. Can usually access an element by index (but not always)

Sequence adaptors Built upon a specific sequence container but changes the interface.

Associative Stores a value that is associated with a specific key.

Sequence containers

`vector`	Stores values in a contiguous block of memory (array-based). Can change size as needed.
`array`	Works like vector, but has a fixed (compile-time) size.
`deque`	Double-ended queue, usually implemented with linked arrays. Useful to store elements in the beginning or end of the range.
`list`	A double-linked list
`forward_list`	A single-linked list made to be as efficient as possibly

Sequence adaptors

`stack` Usually based on deque, can only access the element that was added last.

`queue` FIFO queue

`priority_queue` A queue that stores values according to some given priority function.

Associative containers

`map` Associates a value to a key of a given type. `map<string, int>` uses a string as a key to fetch a value of type int

`set` Works like `map`, but will only use the key.

Also exists as `multi-` and `unordered-` variants

```cpp
#include <map>
#include <iostream>
using namespace std;

int main()
{
    map<string, int> table { {"Key 1", 2},
                             {"Key 2", 5} };
    table["Hello"] = 6;
    for ( pair<string const, int> p : table )
    {
        cout << p.first << ": " << p.second << endl;
    }
    // With auto:
    for ( auto && p: table )
    {
        cout << p.first << ": " << p.second << endl;
    }
    //
    // C++17 below (using structured bindings):
    for ( auto && [key, value] : table )
         cout << key << ": " << value << endl;
}
```

1 Containers

2 Iterators

3 Algorithms

An iterator is a common interface for referring to an element in a given container. The container supplies a specialized iterator that knows how to traverse the underlying data structure.

Has the same interface as built-in pointers.

There are five different categories that specifies what we can use the iterator for.

	Iterator Category				
Operations	**Input**	**Output**	**Forward**	**Bidirectional**	**Random Access**
`==, !=`	✓	✓	✓	✓	✓
`*, ->`	Read	Write	Read/Write	Read/Write	Read/Write
`++`	✓	✓	✓	✓	✓
`--`	-	-	-	✓	✓
`+, +=, -, -=`	-	-	-	-	✓
`<, <=, >, >=`	-	-	-	-	✓
`i[n]`	-	-	-	-	✓

The type of iterator given by a specific container is decided by the underlying data structure.

If the category given by a container is bidirectional or random access, we'll also have access to reverse iterators to traverse the data structure in reverse order.

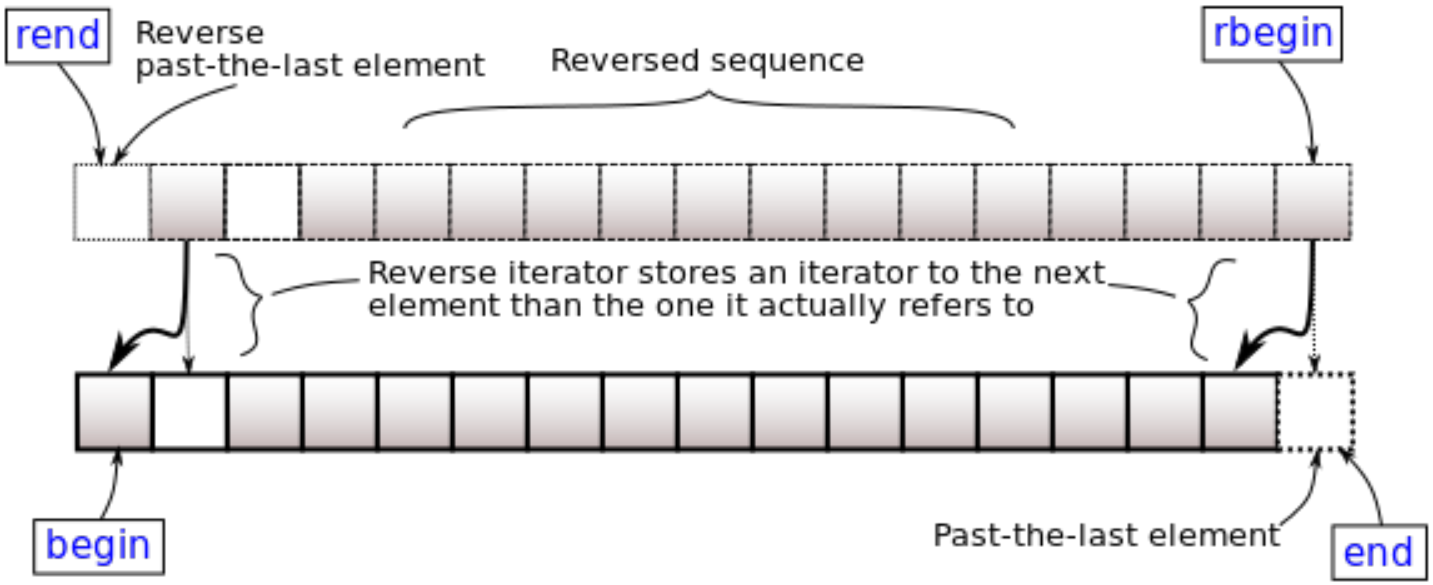

Figure : Relationship between an iterator type and corresponding reverse iterator cppreference

```
vector<int> values {1, 2, 3, 5, 8};
// explicitly declaring an iterator
vector<int>::iterator it { values.begin() };
*it = 5;  // change first element (1) to 5
// iterate over range using iterators
for ( auto it { begin(values) }; it != end(values); ++it )
{
    cout << *it << endl;
}
```

1 Containers
2 Iterators
3 Algorithms

- Iterators are often used to do general calculations on ranges of data with help of generalized algorithms.
- Simple example:

```
vector<int> vals {1, 2, 5, 2, 7};
sort(begin(vals), end(vals));
// vals is in order 1 2 2 5 7
```

- The algorithm specifies the minimum requirement on iterators passed.
- Some containers have member functions that work for that specific container:

```
list<int> lst {1, 45, 2, 5};
lst.sort();
```

- You can change the default behavior of most algorithms with a *callable object*

```
bool comp(int a, int b) { return b < a; }
vector<int> v {1, 2, 4, 2, 7};
```

  - Function pointer

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

- You can change the default behavior of most algorithms with a *callable object*

```
bool comp(int a, int b) { return b < a; }
vector<int> v {1, 2, 4, 2, 7};
```

  - Function pointer

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

  - Function object (class type with function call operator)

    ```
    sort(begin(v), end(v), std::greater<int>{});
    ```

- You can change the default behavior of most algorithms with a *callable object*

```
bool comp(int a, int b) { return b < a; }
vector<int> v {1, 2, 4, 2, 7};
```

  - Function pointer

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

  - Function object (class type with function call operator)

    ```
    sort(begin(v), end(v), std::greater<int>{});
    ```

  - Lambda expression - temporary anonymous function object

    ```
    sort(begin(v), end(v), [](int l, int r) { return r < l; });
    ```

Eric Elfving

www.liu.se

LINKÖPING UNIVERSITY