STL CONTAINERS

WITH A FOCUS ON VECTORS AND ITERATORS

Tommaso Piccini,
Computer Vision Laboratory

LINKÖPING
UNIVERSITY

CONTAINERS IN C++11 STL

STL provides a set of container classes which replicate data structures commonly used in programming.

- Implemented as class template
- Transparent memory management
- Member functions to access elements
- Common member functions (BUT DIFFERENT IN COMPLEXITY!!)

- ...plus container adaptors.

LIST OF AVAILABLE CONTAINERS

Sequence containers

- Array
- Vector
- Deque
- Forward_list
- List

- Stack
- Queue
- Priotity_queue

Associative containers

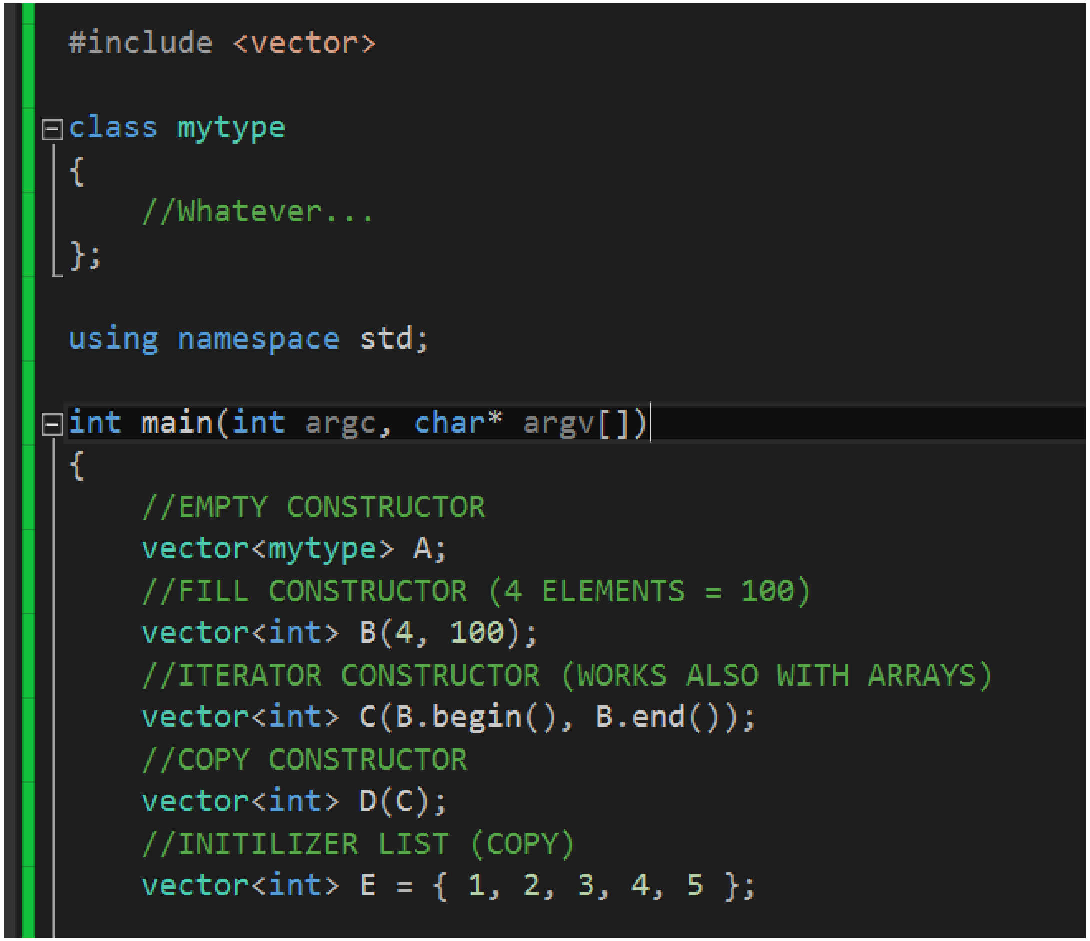
- Set
- Multiset
- Map
- Multimap

- + unordered versions

VECTORS

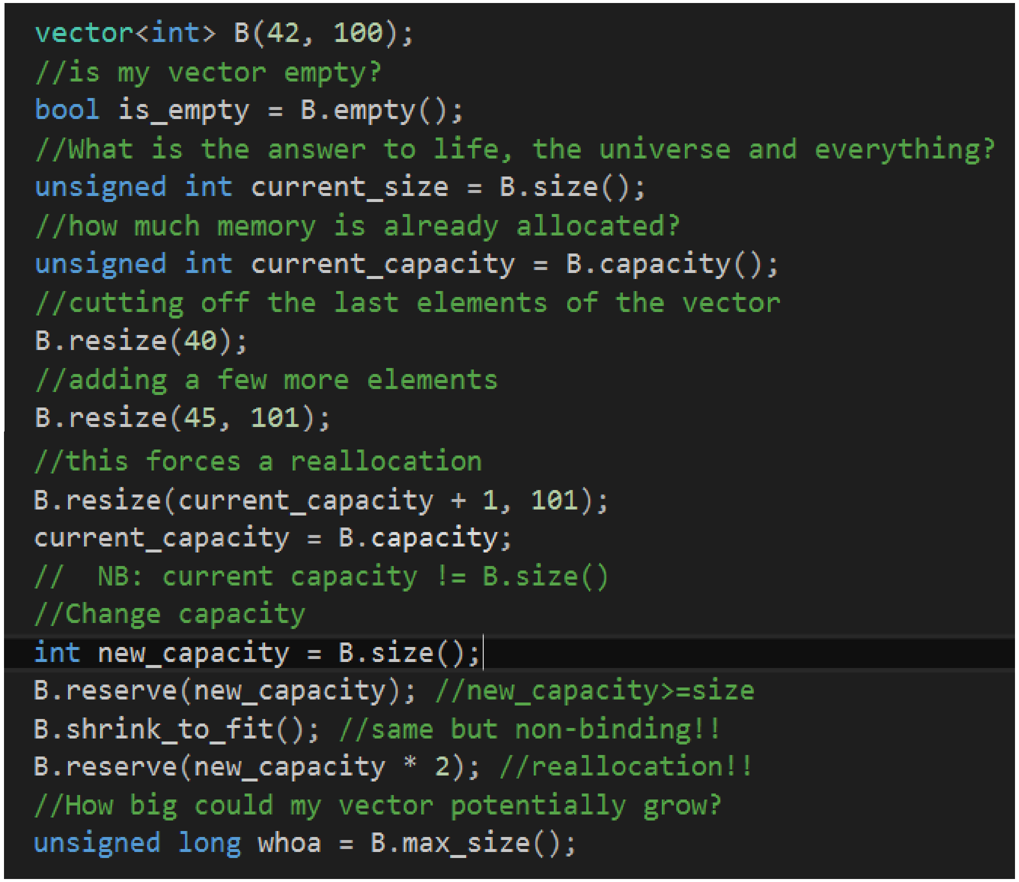
- Dinamically resizing arrays
- Contiguous (efficient access to elements, iterators)
- With size growth may require reallocation (logarithmic growth)
- Low memory efficiency (to limit the cost of reallocation)
- Quite efficient in insertion/removal of elements from the end
- Inefficient in insertion/removal of elements in the middle

VECTORS - constructors



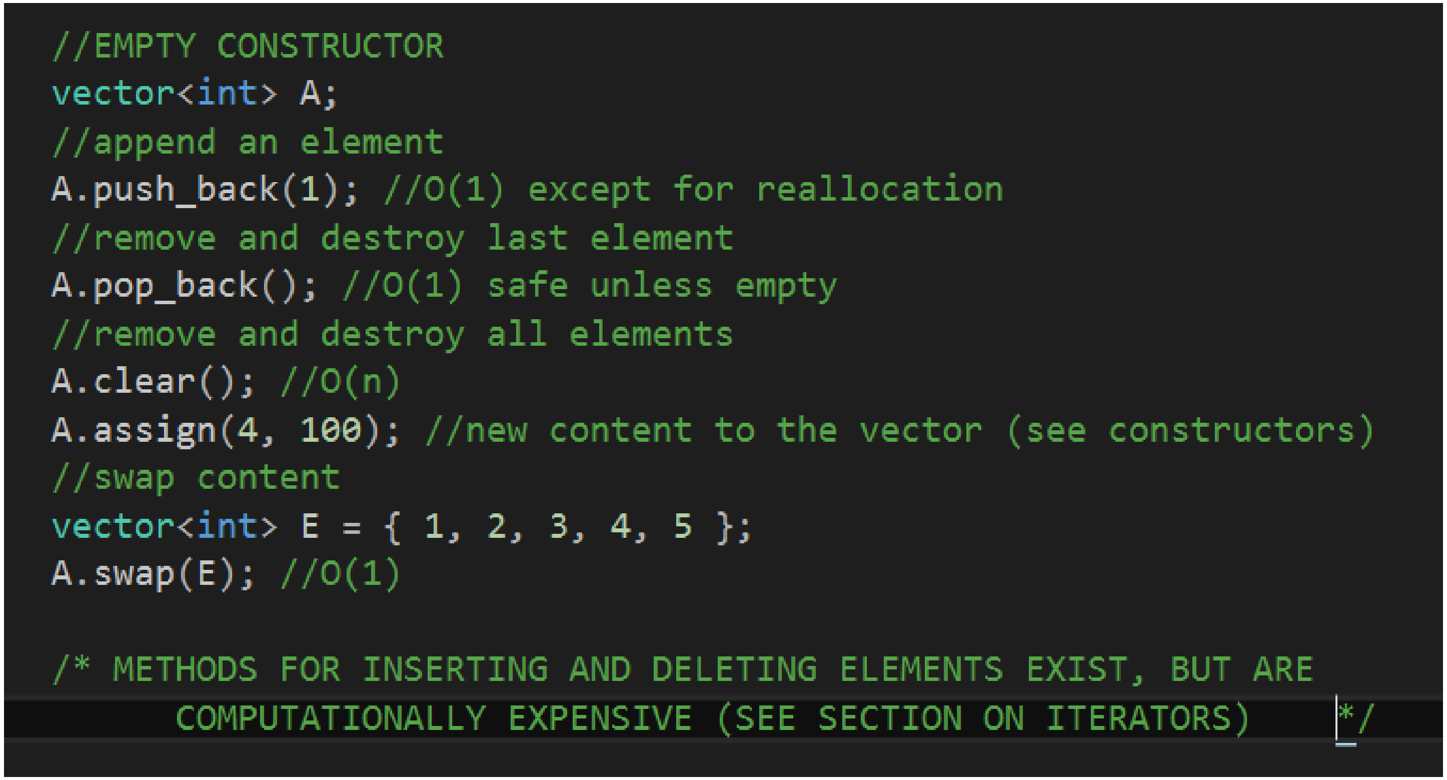
```
#include <vector>

class mytype
{
    //Whatever...
};

using namespace std;

int main(int argc, char* argv[])
{
    //EMPTY CONSTRUCTOR
    vector<mytype> A;
    //FILL CONSTRUCTOR (4 ELEMENTS = 100)
    vector<int> B(4, 100);
    //ITERATOR CONSTRUCTOR (WORKS ALSO WITH ARRAYS)
    vector<int> C(B.begin(), B.end());
    //COPY CONSTRUCTOR
    vector<int> D(C);
    //INITILIZER LIST (COPY)
    vector<int> E = { 1, 2, 3, 4, 5 };
```

VECTORS – member functions (size)



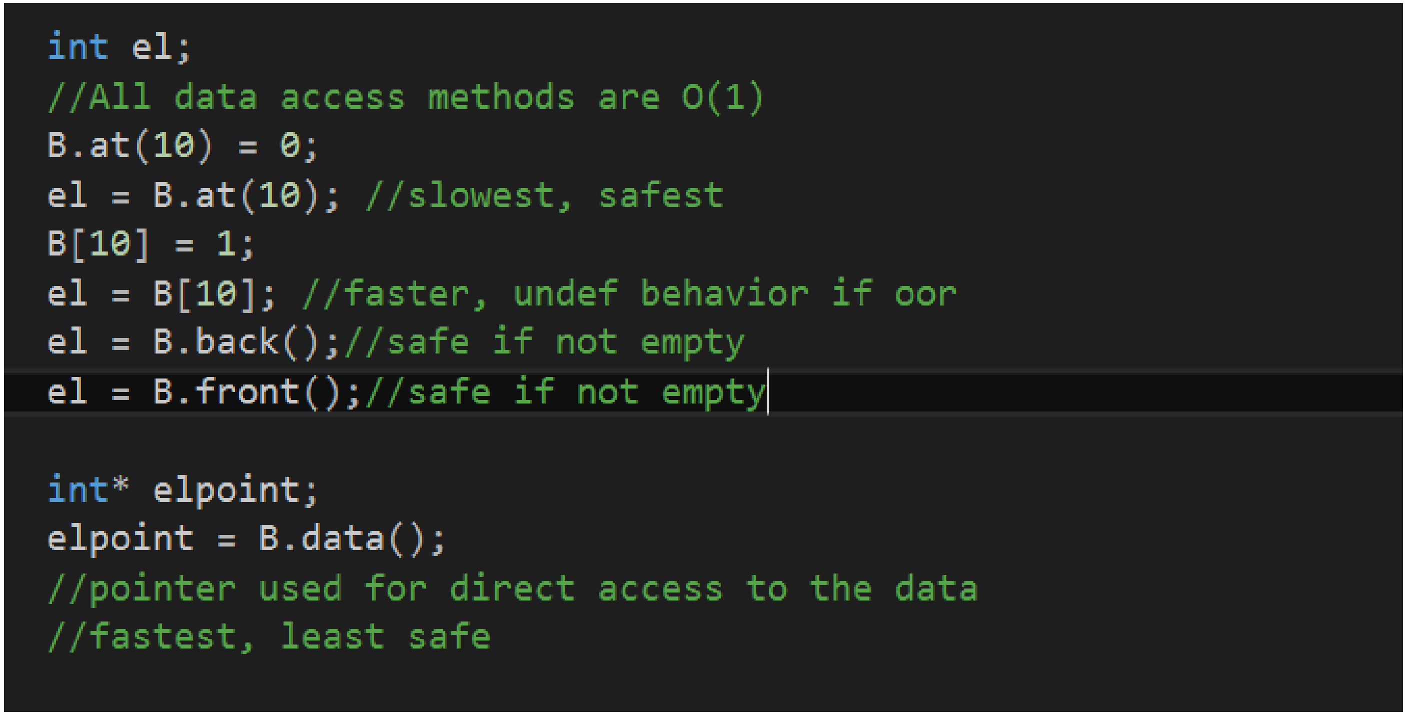
```
vector<int> B(42, 100);
//is my vector empty?
bool is_empty = B.empty();
//What is the answer to life, the universe and everything?
unsigned int current_size = B.size();
//how much memory is already allocated?
unsigned int current_capacity = B.capacity();
//cutting off the last elements of the vector
B.resize(40);
//adding a few more elements
B.resize(45, 101);
//this forces a reallocation
B.resize(current_capacity + 1, 101);
current_capacity = B.capacity;
//  NB: current capacity != B.size()
//Change capacity
int new_capacity = B.size();
B.reserve(new_capacity); //new_capacity>=size
B.shrink_to_fit(); //same but non-binding!!
B.reserve(new_capacity * 2); //reallocation!!
//How big could my vector potentially grow?
unsigned long whoa = B.max_size();
```

VECTORS – member functions (modifiers)



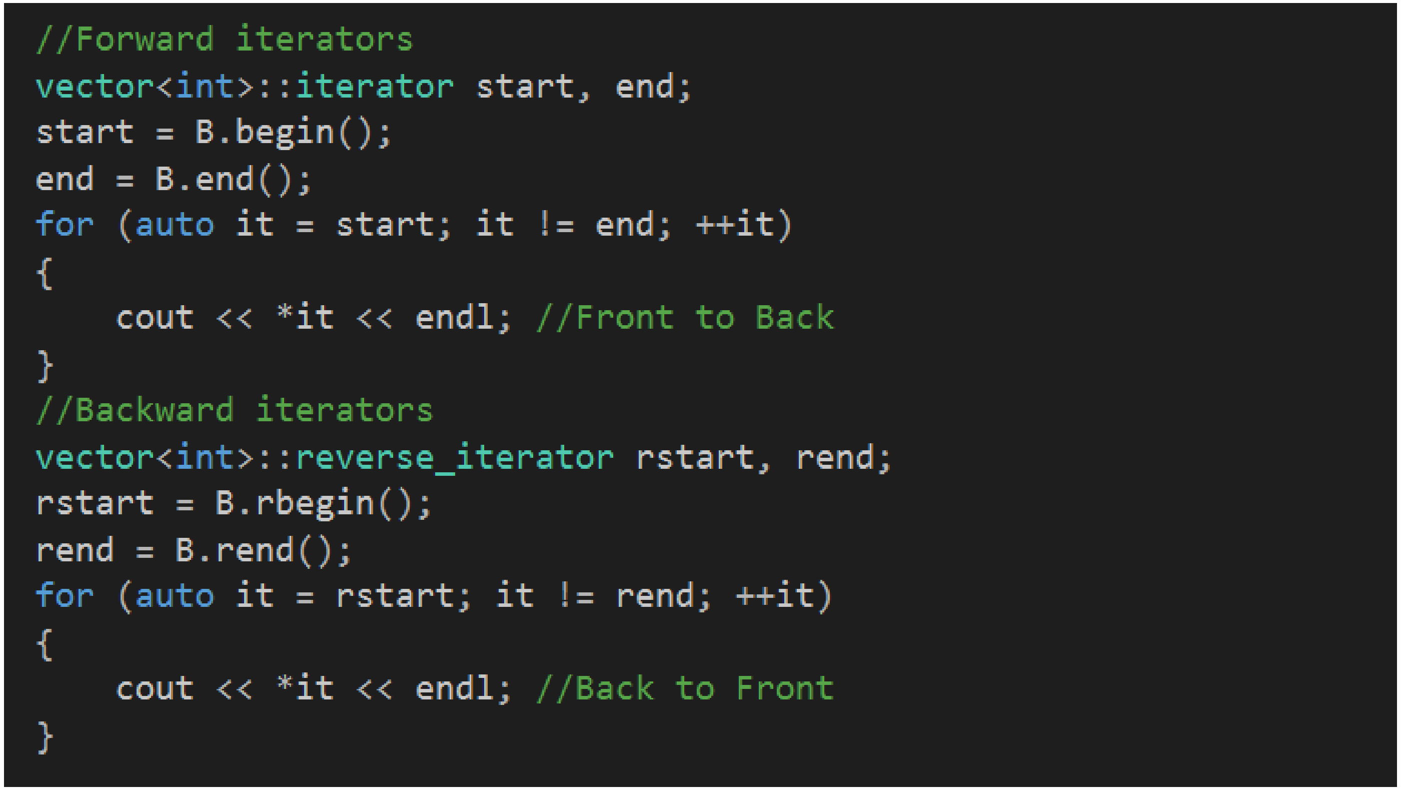
```
//EMPTY CONSTRUCTOR
vector<int> A;
//append an element
A.push_back(1); //O(1) except for reallocation
//remove and destroy last element
A.pop_back(); //O(1) safe unless empty
//remove and destroy all elements
A.clear(); //O(n)
A.assign(4, 100); //new content to the vector (see constructors)
//swap content
vector<int> E = { 1, 2, 3, 4, 5 };
A.swap(E); //O(1)

/* METHODS FOR INSERTING AND DELETING ELEMENTS EXIST, BUT ARE
      COMPUTATIONALLY EXPENSIVE (SEE SECTION ON ITERATORS)    */
```

VECTORS – member functions (access)



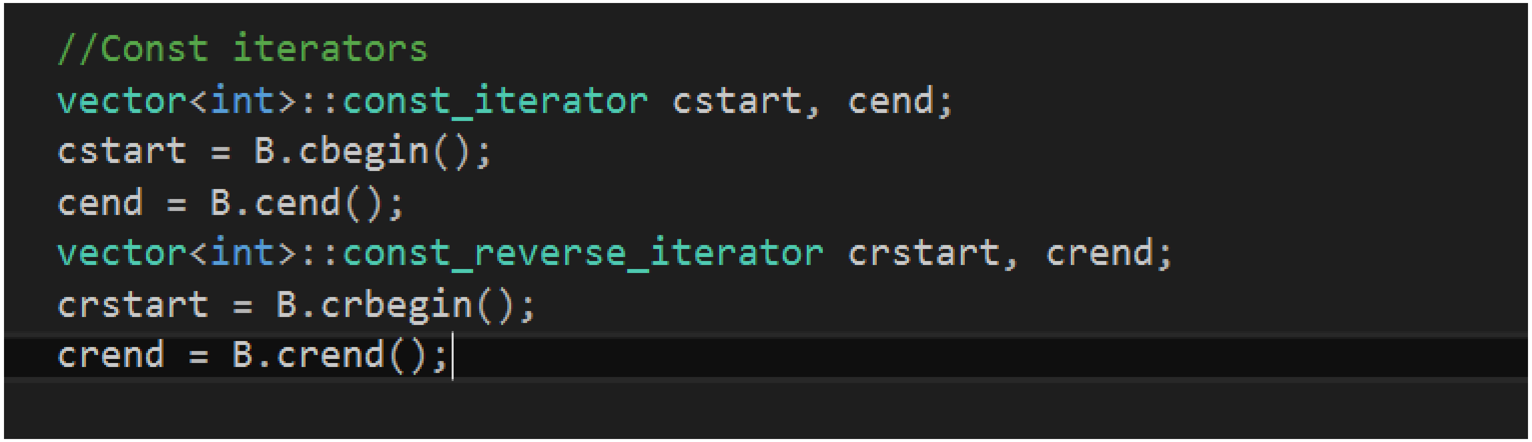
```
int el;
//All data access methods are O(1)
B.at(10) = 0;
el = B.at(10); //slowest, safest
B[10] = 1;
el = B[10]; //faster, undef behavior if oor
el = B.back();//safe if not empty
el = B.front();//safe if not empty

int* elpoint;
elpoint = B.data();
//pointer used for direct access to the data
//fastest, least safe
```

VECTORS – iterators



```
//Forward iterators
vector<int>::iterator start, end;
start = B.begin();
end = B.end();
for (auto it = start; it != end; ++it)
{
    cout << *it << endl; //Front to Back
}
//Backward iterators
vector<int>::reverse_iterator rstart, rend;
rstart = B.rbegin();
rend = B.rend();
for (auto it = rstart; it != rend; ++it)
{
    cout << *it << endl; //Back to Front
}
```

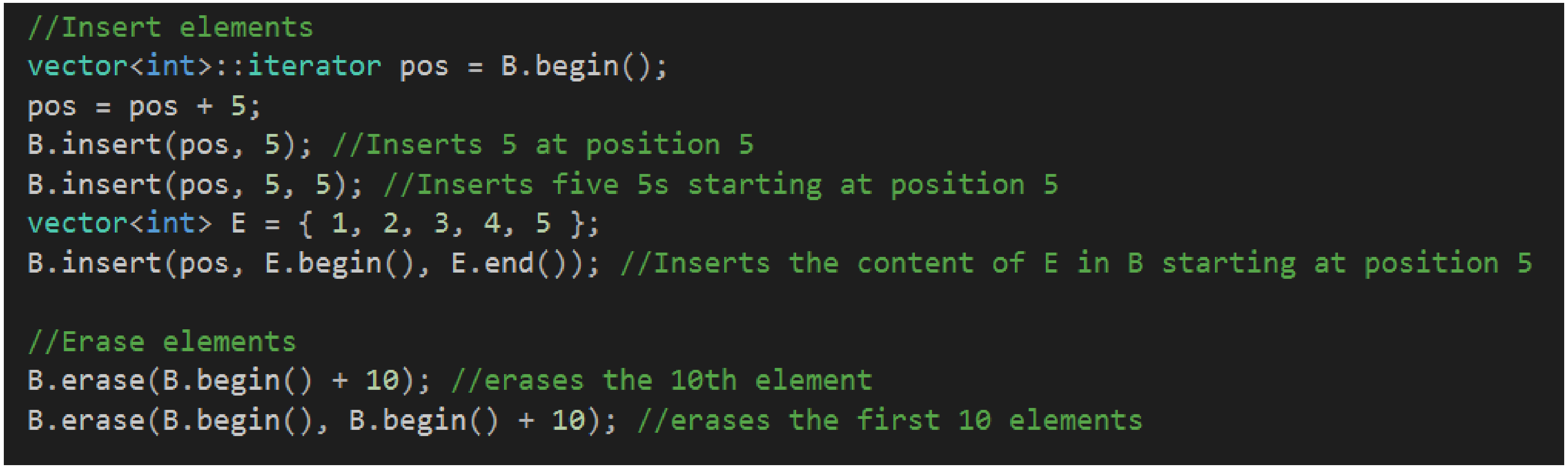
VECTORS – const iterators



```
//Const iterators
vector<int>::const_iterator cstart, cend;
cstart = B.cbegin();
cend = B.cend();
vector<int>::const_reverse_iterator crstart, crend;
crstart = B.crbegin();
crend = B.crend();
```

- Same mechanic, but cannot be used to modify pointed content (even if content is not const!!)

VECTORS – insert/erase



```cpp
//Insert elements
vector<int>::iterator pos = B.begin();
pos = pos + 5;
B.insert(pos, 5); //Inserts 5 at position 5
B.insert(pos, 5, 5); //Inserts five 5s starting at position 5
vector<int> E = { 1, 2, 3, 4, 5 };
B.insert(pos, E.begin(), E.end()); //Inserts the content of E in B starting at position 5

//Erase elements
B.erase(B.begin() + 10); //erases the 10th element
B.erase(B.begin(), B.begin() + 10); //erases the first 10 elements
```

- Mind the complexity!!

OTHER CONTAINERS

- Mind the complexity!!
- ARRAYS – fixed size sequences (contiguous)
- DEQUE – double ended queues (non-contiguous, expanding on both sides)
- FORWARD_LIST – singly linked list (non-contiguous, O(1) insert/delete, O(n) access)
- LIST – doubly linked list (like forward list, but can be browsed backwards)
- STACK – LIFO adaptor (defaults to deque)
- QUEUE – FIFO adaptor (defaults to deque)

OTHER CONTAINERS

- MAP – key/value associative container
  - Typically implemented as binary trees
  - Access values by map[key]
  - Ordered on key (allows subset iterators)
  - Unordered variant available
- MULTIMAP – 1 to N map
- SET – key=value
  - Also binary trees
  - Elements cannot be modified after insertion
  - Unordered variant available (uses buckets)
- MULTISET – allows repetition of elements

THE END

www.liu.se

LINKÖPING
UNIVERSITY