C++

STL: sequences and iterators

Sequences

- Homogeneous linear collection of objects
- STL offers: vector, list, deque (and slist and bit_vector)
- Vector: one dimensional array
  - Fast random access
  - Fast add / del of element at the back only
- List: double linked list
  - Fast access at front and back
  - Fast add / del of element at any position
- Deque: double ended array
  - Fast random access
  - Fast add / del of element at the front and back

Some operations on sequences

- Access element (for all containers):
  - front(): get the first element
  - back(): get the last element
- Access element (vector and deque only):
  - operator[]: subscript operator to access element at position i
- Add / remove at back (for all):
  - push_back(): append ele at back
  - pop_back(): remove ele from back (be careful: pop_back() return void)
- Add / remove at front (list and deque):
  - push_front()
  - pop_front()
- Add / remove everywhere (list):
  - insert()
  - erase()

Some operations on sequences

- comparison: “==“, “<“, “!=“
- size(): returns the size of the container
- empty(): returns true if seq. is empty
- resize(int i): modify the size of seq to I
- clear(): erase all elements
  - Be careful:
    - clear() will call the destructor of the contained objects
    - If contained objects are pointers, clear() does not de-allocate the memory pointed to by the pointers

List operations on seq.

- insert() and erase(): allow to insert and remove an element at a given position in the seq.
- They are available for all seq. but:
  - For list this can be done fast (O(1))
  - Not for vector and deque (O(n))

Example

```
#include <deque>
#include <vector>
#include <list>
#include <iostream>

using namespace std;

int main()
{
 vector<int> v;
 v.push_back(1); v.push_back(2);
 cout << v.size() << endl;
 cout << v[0] << v[1] << endl;

 deque<double> d;
 d.push_back(1.0); d.push_front(2.0);
 d[0] = 3.0;

 list<int> l;
 l.push_back(1); l.push_back(2);
 int t = l.back(); l.pop_back(); // now only 1 elt
}
```


Introduction to iterator

- Suppose we have an array of int and we want to iterate through its elements

```
int a[10] = { /* …*/ }; //…
int* end = &a[10];

for (int i = 0; i < 10; i++) {
  // do something with a[i]
}

for (int* it = a; it != end; it++) {
  // do something with *it
}
```


Iterator

- We would like to generalize pointers (and their arithmetic)
- This is exactly what iterators do:
  - They are objects that points to other objects
  - They allow to iterate over a range of objects
- Operations on pointers:
  - Access: *p, p->
  - Go to next ele: ++p, --p
  - Compare iterators: == !=
- Each container in the STL provide an iterator type:
  - vector<T>::iterator it;
  - list<T>::iterator it;
  - deque<T>::iterator it;

Iterator

- Iterators are important to generic programming:
  - They are an interface between containers and algorithms
  - With iterators it is possible to write generic algorithms working on many kind of containers: vector, deque, etc

Example

- Display the content of a vector using an iterator

```
vector<int> v;
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); it++)
  cout << *it << " ";
```

Note:

v.begin() returns an iterator to the beginning (front) of the vector

v.end() returns an iterator to the end (back) of the vector

Example: general display()

- Suppose we want to generalize the previous display code

```
template <class ContainerT>
void display(ContainerT c) {
  typename ContainerT::iterator it;
  for (it = c.begin(); it!= c.end(); it++) {
    cout << *it << " ";
  }
}

// …
vector<int> v;
// …
display(v);
```

```
template <class IteratorT>
void display(IteratorT begin, IteratorT end) {
  IteratorT it;
  for (it = begin; it!= end; it++) {
    cout << *it << " ";
  }
}

// …
vector<int> v;
// …
display(v.begin(), v.end());
```


Example: find an element in a container

```
template <class IteratorT, class T>
IteratorT
find (IteratorT begin, IteratorT end, const T& el)
{
  while(begin != end && *begin != el)
  {
    ++begin;
  }
  return begin;
}
```

find() will return the position of the element el in the container if it found it,
otherwise it will return the end of the container.

find() is a generic algorithm that can be applied to any type of container.