Comp151

STL: Sequences & Iterators

“STL” Sequence Containers from the Standard C++ Library

- Here are some homogeneous container classes commonly used to represent a sequence of objects of the same type.
- `vector` : one-dimensional array
  - Fast access at any position.
  - Add and remove elements only at the back.
- `list` : doubly-linked list
  - Fast access to front and back only.
  - Add and remove elements at any position.
- `deque` : double-ended array
  - Fast access at any position.
  - Add and remove elements only at the front and back.

Standard Sequence Containers

- They differ in how quickly different access operations can be performed. *n* is the number of elements currently in the container. O(1) means *constant time*.

Container	Access/Retrieval	Insert, Erase
`vector` (1D array)	O(1) random access	O(1) at back only O(*n*) at front, in middle
`list` (doubly-linked list)	O(1) at front/back only <u>No</u> random access [would be O(*n*)]	O(1) at any position
`deque` (doubly-ended queue)	O(1) random access	O(1) at front/back only O(*n*) in middle

```
#include <vector>
#include <deque>
#include <list>
#include <string>
#include <iostream>
using namespace std;

main()
{
    vector<double> vd;
    deque<string> ds;
    list<int> li;

    vd.push_back(5.0); vd.push_back(10.0); vd.push_back(15.0);
    cout << vd[1] << endl;

    ds.push_back(" World "); ds.push_front(" Hello ");
    cout << ds[0] << ds[1] << endl;

    li.push_back(1); li.push_back(2);
    cout << li[0];                         // error: list doesn't support random access
}
```

Sequence Containers: Access, Add, Remove

- Element access for all:
  - `front()`: First element
  - `back()`: Last element
- Element access for `vector` and `deque`:
  - `[ ]`: Subscript operator, index not checked.
- Add/remove elements for all:
  - `push_back()`: Append element.
  - `pop_back()`: Remove last element.
- Add/remove elements for `list` and `deque`:
  - `push_front()`: Insert element at the front.
  - `pop_front()`: Remove first element.

Sequence Container: Other Operations

- Miscellaneous operations for all:
  - `size()`: Returns the number of elements.
  - `empty()`: Returns true if the sequence is empty.
  - `resize(int i)`: Change size of the sequence.

- Comparison operators `==  != <` etc. are also defined.
  - i.e., you can compare if two containers are equal.

- "List" operations are fast for `list`, but also available for `vector` and `deque`:
  - `insert(p, x)`: Insert an element at a given position.
  - `erase(p)`: Remove an element.
  - `clear()`: Erase all elements.

… but what is p ?!

Example: Print with an Array

```
const int LEN = 10;
int x[LEN];
int* const x_end = &x[LEN];

for (int* p = x; p <= x_end; ++p) {
    cout << *p;
}
```

- We use an `int` pointer to access the elements of an `int` sequence with some basic operations:

Operation	Goal
`p = x`	initialize to the beginning of an array
`*p`	access the element being pointed to
`++p`	point to the next element
`p != x_end`	compare with the end of an array

Example: Printing a List Sequentially

- Similarly, to access `list<int>` elements sequentially, one may define p as an <u>iterator</u> `list<int>::iterator`, and use functions `begin()` and `end()` to get iterators pointing to the beginning and end of the container.

```
#include <list>          // "list" class of STL
using namespace std;

list<int> x;
list<int>::iterator p;

for (p = x.begin(); p != x.end(); ++p) {
    cout << *p;
}
```

Example: Printing a Vector Sequentially

- One can similarly define an iterator `vector<double>::iterator` to sequentially go through items in a `vector<double>` .

```
#include <vector>                  // "vector" class of STL
using namespace std;

vector<double> x;
vector<double>::iterator p;

for (p = x.begin(); p ! = x.end(); ++p) {
    cout << *p;
}
```

Iterators

- For each kind of container in the STL there is an *iterator type*.

```
list<int>::iterator ip;
vector<string>::iterator vp;
deque<double>::iterator dp;
```

- Iterators are a generalization of pointers, and are used much like pointers:
  - They can be used to indicate elements in the sequence.
  - A *pair* of iterators can be used to indicate a subsequence.
- Operations on iterators are just like pointers in arrays:
  - Access element: `*p` `p->`
  - Go to next or previous element: `++p` `--p`
  - Compare iterators: `==` `!=`

```
#include <list>
#include <iostream>
using namespace std;

void display(list<int>::iterator first, list<int>::iterator end)
{
    list<int>::iterator p;
    for (p = first; p != end; ++p) {
        cout << *p << " ";
    }
}

main()
{
    list<int> my_list, small, large;
    list<int>::iterator ip;

    for (int i = 1; i < 13; ++i) {                    // initialize values in the list
        my_list.push_back(i*i % 13);
    }
    for (ip = my_list.begin(); ip < = my_list.end(); ++ip) {
        if (*ip < 7) {
            small.push_back(*ip);
        } else {
            large.push_back(*ip);
        }
    }
    cout << "‘my_list: ";  display(my_list.begin(), my_list.end());  cout << endl;
    cout << "‘small: ";  display(small.begin(), small.end());  cout << endl;
    cout << "‘large: ";  display(large.begin(), large.end());  cout << endl;
}
```

Example: `locate()` with an `int` Iterator

- Iterators provide a systematic way of looking at elements of sequence container without differentiating between different container classes.
- The same code works correctly for all sequence container classes.

```
// File: "locate_int_iterator.cpp"
typedef int* Int_Iterator;

Int_Iterator_locate( Int_Iterator begin, Int_Iterator end, const int& value )
{
    while (begin != end && *begin != value) {
        ++begin;
    }
    return begin;
}
```

Example: `locate()` with an `int` Iterator…

```
#include <iostream>
#include "locate_int_iterator.cpp"
using namespace std;

int main()
{
    const int SIZE = 100; int x[SIZE]; int num;
    Int_Iterator begin = x; Int_Iterator end = &x[SIZE];

    for (int i = 0; i < SIZE; ++i) {
       x[i] = 2 * i;
    }

    while (true) {
       cout << "Enter number: "; cin >> num;
       Int_Iterator position = locate(begin, end, num);

    if (position != end) {
       ++position;
       if (position != end) cout << "Found before " << *position << endl;
       else cout << "Found as last element" << endl;
    }
       else cout << "Not found" << endl;
    }
}
```

Why Are Iterators So Great?

- Because they allow us to separate algorithms from containers.
  - If we change the `locate()` function as follows, it still works:

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

  - The new `locate()` function contains no information about the implementation of the container, or how to move the iterator from one element to the next.
  - The same `locate()` function can be used for any container that provides a suitable iterator.
  - Using iterators lets us turn `locate()` into a re-useable <u>generic function</u>!

Example: `locate()` with an Iterator

```
#include <iostream>
#include <vector>                          // "vector" class from STL
using namespace std;

int main()
{
    vector<int> x(SIZE); int num;
    for (int i = 0; i < SIZE; ++i) {
        x[i] = 2 * i;
    }

    while (true) {
        cout << "Enter number to locate: "; cin >> num;
        vector<int>::iterator position = locate(x.begin(), x.end(), num);

        if (position != x.end()) {
            ++position;
            if (position != x.end()) cout << "Found before " << *position << endl;
            else cout << "Found as last element." << endl;
        } else {
            cout << "Not found" << endl;
        }
    }
}
```

```
#include <list>
#include <vector>
#include <string>
#include <iostream>
using namespace std;

template<class IteratorT>
void display(IteratorT start, IteratorT end)        // now display() becomes our own generic algorithm!
{
    for( IteratorT p = start; p != end; ++p ) {
        cout << *p << " ";
    }
}

main()
{
    list<int> li;
    vector<string> vs;

    for (int i = 1; i < 13; ++i) {
        li.push_back(i*i % 13);
    }
    vs.push_back("Now"); vs.push_back("Is"); vs.push_back("The");
    vs.push_back("Time"); vs.push_back("For"); vs.push_back("All");

    cout << “li: "; display(li.begin(), li.end()); cout << endl;
    cout << “vs: "; display(vs.begin(), vs.end()); cout << endl;
}
```

More on STL…

- Today, all modern C++ compilers include at least a basic implementation of STL (Standard Template Library)
- Beware: some implementations still do not fully support all the specifications in the Standard C++ Library

- The classic standard reference for STL (including the extended SGI version of STL):
  - http://www.sgi.com/tech/stl/

- An excellent portable open-source implementation, if you’re not satisfied with the one that came with your C++ compiler:
  - http://www.stlport.org/