SSE2034: System Software Experiment 3 Spring 2016

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
STL Collection Types

- Template based set of collection classes
- STL collection types (container types)
  - Sequences
    - vector - collection of elements of type T
    - list - doubly linked list, only sequential access
    - deque - fast insert at either end
  - Associative containers
    - map
    - set
### Skeleton class for multiple types

```cpp
template < class T >
class Value
{
    T _value;
    public:
    Value ( T value ) { _value = value; }

    T getValue ();

    void setValue ( T value );
};

template < class T >
T Value<T>::getValue () { return _value; }

template < class T >
void Value<T>::setValue ( T value ) { _value = value; }

Value<float> values[10];  // array of values of type float
```
Iterators

- Allows access of elements in a collection
  - forward iterator
  - bi-directional iterator
  - random access iterator
    - bi-directional + constant time access
STL Collections Methods

- General class methods
  - empty – True if collection is empty
  - size – number of elements in the collection
  - begin – start of collection in forward iterator
  - end – one past the last in forward iterator
  - rbegin – last of collection in backward iterator
  - rend – one before the start in backward iterator
  - clear – erases all elements
  - erase – erase an element or range of collection
  - Insert – insert an element
Add and Remove

- vector, deque, list
  - front - get a reference to first element
  - back - get a reference to last element
  - push_front - add to the start (NOT for vector)
  - push_back - add to the end
  - pop_front - remove from the start (NOT for vector)
  - pop_back - remove from the end
Iterators

vector<int> v;

// input from console
while (cin >> input)
    v.push_back(input);

// sort elements in vector
sort(v.begin(), v.end());

// print out - version 1
for (int i=0; i<v.size(), i++)
    cout << v[i] << "\n";

// print out - version 2
vector<int>::iterator ii;
for (ii=v.begin(); ii!=v.end(); ii++)
    cout << *ii << "\n";
Operator [ ]

- Vector, deque, map can use [ ] to access element
  - Similar semantics in C array element access
  - Not available for list

- at - similar to operator [ ]
  - `v.at(index)` vs. `v[index]`
  - Available for vector, deque
  - Perform bound checking unlike operator [ ]
Vector

- Automatically take care of resizing
- Constructor
  - `vector<T> vec;` // empty vector
  - `vector<T> vec(10);` // capacity of 10 elements
  - `vector<T> vec(10, 1);` // and initialized with 1
- Add / remove elements
  - `push_back(e), pop_back()`
  - `insert(pos, e), erase(pos)`
#include <iostream>
#include <vector>
using namespace std;

main()
{
    vector<int> iv1;
    vector<int> iv2(10);

    cout << "iv1: capacity = " << iv1.capacity() << "\n";
    cout << "iv2: capacity = " << iv2.capacity() << "\n";

    iv1.push_back(10);  // can increase size and capacity
    iv2[0] = 10;
    iv2.push_back(20);  // iv2[1] or iv2[10]
    iv1[1] = 20;        // segmentation fault
}
List

- Doubly linked list
- Constructor
  - `list<T> li;` // empty list (size = 0)
  - `list<T> li(10);` // list with size = 10
  - `list<T> li(10, 1);` // and initialized with 1

- Add / remove elements
  - `push_front(e), push_back(e), insert(pos, e)`
  - `pop_front(), pop_back(), erase(pos)`
List

- **Test if list is empty**
  - // std::list<int> il(100);
  - il.size() takes O(n)
  - il.empty() takes O(1)  // why?

- **Useful algorithms for list**
  - sort() on list - uses O(n log n) algorithm
  - reverse() on list - reverse the order
  - unique() on list - remove repeating elements
List

- **insert** \((pos, e)\) : insert \(e\) before position \(pos\)

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

main()
{
    list<int> L;

    L.push_back(0);
    L.push_back(0);
    L.insert(++L.begin(), 2);
    L.push_back(5);
    L.push_back(6);

    list<int>::iterator i;
    for (i=L.begin(); i!=L.end(); i++) cout << *i << " ";
    cout << endl;
}
```
Set

- Sorted Associative Container
  - Sorted at insert by using default compare-operator

- Unique
  - No two elements are same

- Fast add & remove
  - Add/remove a sorted range in linear time
#include <iostream>
#include <set>
using namespace std;

int main(){
    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);

    set<int>::const_iterator it;
    it = myset.begin();
    while (it != myset.end()) {
        cout << *it << " ";
        ++it;
    }
    cout << endl;
}
Map

- Associative container
  - `<key, value>` pair
  - Unique element (key)
  - Sorted on key value
Map

- Associative key can be any type

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

int main(){
  map<string, double> market;
  market["apple"] = 2.30;
  market["orange"] = 1.20;
  market["melon"] = 3.30;

  map<string, double>::const_iterator it;
  it = market.begin();
  while (it != market.end()) {
    cout << it->first << ": " << it->second << endl;
    ++it;
  }
  cout << endl;
}
```
[Lab – Practice #1]

- Schedule earlier and shorter job first (use vector)
  - A job has to be executed as much as period.
  - The job which starts early has the highest priority.
  - If two jobs’ start time are same, the job which has less period has higher priority.
- Input:
  - two integers (start, period) & string of name of jobs until -1
- Output:
  - Execution order of jobs
  - If there is no job to handle in specific time unit, print *
$ ./sort_job
input: 4 2
A
input: 2 1
B
input: 4 1
C
input: -1
B * C A A

$ ./sort_job
input: 1 2
D
input: 5 1
E
input: 4 3
F
Input: 4 2
G
input: -1
D D * F F F G G E
From left to right of given circular list, list people’s ID in stride

- Use list to manage people’s id
- Input: stride number and people’s id
- Output: the order of people’s id in defined stride, from left to right

```bash
$ ./stride
stride: 3
id : 1 5 9 8 4 3 2 10 7 6 -1
  9 3 7 5 2 1 10 4 6 8
$ ./stride
stride: 2
id : 1 2 3 4 5 6 -1
  2 4 6 3 1 5
```