CSE 165/ENGR 140
Intro to Object Orient Program

Lecture 25 – Standard Library(2)
Announcement

- Lab #10
  - Due 4/30 at 11:59PM
- Final on Thursday (5/7)
  - Last lecture of semester
  - Review and quiz on Tuesday (5/5)
- Final project is out (in UCMCROPS)
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM
Generic algorithms

- `for_each()`
  - It pulls out each element in a container (through iterators) and passes it as an argument to a function object:

```cpp
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function fn) {
    while (first!=last) {
        fn (*first);
        ++first;
    }
    return fn;   // or, since C++11: return move(fn);
}
```
Generic algorithms

- for_each example using mem_fun():

```cpp
struct Shape {
    virtual void draw() = 0;
    virtual ~Shape() {}
};

struct Circle : public Shape {
    virtual void draw() { cout << "Circle::Draw()" << endl; }
};

struct Square : public Shape {
    virtual void draw() { cout << "Square::Draw()" << endl; }
};

int main() {
    vector<Shape*> vs;
    vs.push_back(new Circle);
    vs.push_back(new Square);
    for_each ( vs.begin(), vs.end(), mem_fun(&Shape::draw) );
    // ...
}
```
Generic algorithms

- Example of additional features
  - Fill functions: to add values to ranges
  - Sequence manipulation
    - copy, reverse_copy, swap, rotate, random_shuffle, etc.
  - Counting, sorting, searching, and replacing
    - find, find_if, binary_search, sort, partial_sort, etc.
  - Heap operations
    - heaps are “binary trees” organized in an array format
    - efficient to represent priority queues
      - make, push, pop, etc.
  - Numeric algorithms
    - accumulate, transform, inner_product, etc.
Generic Containers (Ch. 7)

- All basic data structures are available in the standard library in the form of containers.
- Containers will perform full copies of the elements they contain.
  - You can always store pointers in containers
    - this is very common
  - If you store pointers to objects in a container
    - there is no mechanism for automatic deletion of the elements in your containers
      - so you must do it by hand
Vector and deque

- **Vector**
  - Quick access to indexed items (random access)
  - Often quick insertion at the end with `push_back()`
  - Expensive middle insertion

- **Deque (double-ended-queue)**
  - Similar to vector, it is based on indexed access
  - Also provides quick insertion at the front with `push_front()`
  - It is not based on a single continuous array
    - Multiple blocks used, at least two “vector blocks”
List

- Based on a linked list
- Provides fast `push_front()` and `push_back()`
- But no indexed access, access only with iterators
  - No `operator[]` defined
- Some operations will simply manipulate pointers instead of copying values
  - Specific `sort()` and `reverse()` members do not copy values but only manipulate pointers
Stack and queue

- **Stack**
  - The basic operations are `push()` and `pop()`
  - You can choose its underlying container:
    
    ```
    typedef stack<string> Stack1; // Default case, will use: deque<string>
    typedef stack<string, vector<string> > Stack2; // based on vector
    typedef stack<string, list<string> > Stack3; // based on list
    ```

- **Queue**
  - It is just a restricted form of a deque
  - `push`: insert in one end
  - `pop`: remove from the other end
Bitset
- Holds a fixed number of bits
- Some bit manipulation methods are included

vector<bool>
- This is a template specialization of vector<>:
  - recall what is template specialization, for ex:
    template<> class X<char>() { ... };
- It will compress the booleans in the vector so that it uses less space (one int may have 32 bits!)
- It essentially maintains a vector of bits
Associative containers: set and multiset

Set

- Allows to insert and lookup elements (ex. indexing words in a book)
- Stores only one copy of each inserted element
  - method `count()`: how many instances (0 or 1)
- Needs a comparison function
  - uses function object “`less<>`”
- Stores elements in a balanced tree in order to provide fast search and access to elements (in O(log n) time)
- Uses `insert_iterator<>` to add elements

Multiset

- Same as set but an element can be inserted multiple times
  - `count()`: will return 0 or more for a given element
Associative containers: map and multimap

- Map
  - Similar to **set** but elements have “keys”
  - Allows only one element per key
  - Main operations: insert and lookup
  - Associates keys with elements to be stored
    
    Ex: `mymap.insert(make_pair(47, elem));`

- Multimap
  - Similar operations as in a map
  - But allows multiple elements per key
Map and multimap

- Maps and multimaps are implemented as “balanced binary trees”
  - But their functionality is exactly what would be expected from a hash table
    - Hash tables were not originally available
      - Hash tables are often much faster
    - The external STL library has the following classes:
      - hash_set, hash_multiset, hash_map, hash_multimap
      - slist (singly linked list)
      - “rope” (a container for large strings)
        (note: STL is how the standard library was before it became a standard)
The newest C++ specification, C11, has many more features that are gradually becoming available in compilers, for example:

- Hash tables become available with class `unordered_map<>`
  - hashing function has to be provided (as a function object)
- System-independent functionality for allowing parallel programming
  - thread creation etc.
typedefs often used to simplify template definitions.

Customization of objects is achieved by using default arguments in template definitions.
  ◦ Iterators, allocators, and traits objects often used.

Function objects often used in Algorithm class.

All basic data structures are available as containers.
  ◦ Interface of containers is heavily based on iterators.
  ◦ Some containers are wrappers to other containers.
  ◦ Some containers are specializations of other containers (recall vector<bool> case).