Associative Containers, Algorithms, and Functors

CS 250 - Session 15
C++ Programming

Topics Covered

- Reading 835-859, 997-1027 [Appendix G]
- Associative containers
- Function objects [aka Functors]
  - Types of functors [unary, binary, predicate]
  - Writing and adapting your own functors
  - Predefined functors and function adapters
- Algorithms
- Homework

Associative Containers

- Associative containers are not "linear"
  - Internally stored as a "tree" structure
    - Similar to a linked list; parent and child nodes
    - Several possible meanings to first, last, next
- Associatives store "keys" and "value"
  - The key is used to search and order container
  - Example: name used as key in phone book
- STL has 4: set, multiset, map, multimap
The set Classes

- Logically, the set and multiset classes look like this:

- Internally, they are usually implemented as "red-black" trees.

The set Classes I

- The set and multiset classes are sorted
  - The key and value are the same
  - Set keys are unique, multiset are not
  - Creating sets and multisets

```c++
char a[] = "Stephen Gilbert";
set<char> s1(a, a + 15); // from range
set<char> s2(s1); // from another set
set<char> s3; // Empty set
multiset<char> ms1(a, a + 15); // Empty
multiset<char> ms2; // Empty
```

The set Classes II

- Use insert(v) to add items to a set or multiset
  - Containers are sorted, no push_back(), etc.
  - insert(iterator, value) used to provide "hint"
  - You can also use an inserter iterator with copy

```c++
multiset<char>::iterator i = ms1.begin();
while (i != ms1.end())
  s3.insert(*i++);
```

```c++
while (i != s2.end())
s3.insert( *i++);
```

```c++
copy(s2.begin(), s2.end(), inserter(ms2, ms2.begin()));
```
The set Classes III

- **Searching a set and multiset**
  - `find(v)` returns an iterator or `end()`
    - if `(s1.find('e') != s1.end())` cout << "found";
  - Use a loop to find all copies in a multiset
    - multiset<char>::iterator i = ms1.find('e');
      while (i != ms1.end() && *i == 'e') i++;
  - `count(v)` returns number of times item occurs
    - cout << "# 'e's : " << ms1.count('e');

The set Classes IV

- **Remove items with the erase() member function**
  - **Version 1**: supply a single iterator
    - Only erases single value in multiset
  - **Version 2**: Supply a key
    - Erases all copies of the key from set or multiset
    - Returns number of copies removed
  - **Version 3**: supply an iterator range
    - No return value

The map Classes I

- **Maps store all their data as pair objects**
  - The pair class has public fields first, second
    - Note that this is different than the Pair class
  - You construct a pair using the constructor like this:
    - `pair<string, int> num("Bob", 23);`
  - Or, you can use the `make_pair()` function:
    - `make_pair(string("Bob"), 23);`
You'll normally construct an empty map
- Can't easily convert sequence containers
  ```cpp
  map<string, int> m; // This is OK
  char a[] = "A char array"; // This is NOT
  map<char, char> nogo(a, a+sizeof a);
  ```

Use insert() and make_pair() add an item
- ```cpp
  m.insert(make_pair(string("Five"), 5));
  ```

Use an iterator to access the elements
- Use `itr->first`, `itr->second` fields
  ```cpp
  map<int, string>::iterator i;
  for (i = m.begin(); i != m.end(); i++)
      cout << i->first << " : "
           << i->second << endl;
  ```
- Note that the `iter->first` is [essentially] read-only
  - Can change value, but not key
- Examples: map01.cpp, map02.cpp

The map class overrides the subscript operator
- Allows you to create "associative arrays" [map03.cpp]
  ```cpp
  map<string, double> test;
  while (getline(cin, student, ',')) {
      cin >> grade;
      test[student] = grade;
  }
  ```
- Can both read [insert] and write
  ```cpp
  cout << s << " grade is " << test[s] << endl;
  ```
Iterators facilitate generic algorithms like this:

```cpp
template <class InItr, class T>
T accumulate(InItr first, InItr last, T init) {
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

- Standard library has this in header `<numeric>`

Can sum values in any container: [generic01]

```cpp
int a[5] = { 1, 2, 3, 4, 5 };
list<int> lst(a, a+5);
int sum1 = accumulate(a, a+5, 0);
int sum2 = accumulate(lst.begin(), lst.end(), 0);
```

Can also pass "function" argument for processing

```cpp
int mult( int x, int y ) { return x * y; }
int product = accumulate(a, a+5, 1, mult);
```

What is a Functor?

- A functor is STL-speak for a "function object"
  - So??? What is a "function object"?
    "Any entity that can be applied to zero or more arguments to obtain a value and/or modify the state of a computation"
  - Shorthand: anything you can "call" or "invoke"
    » Includes function pointers [like `mult`]
    » Also includes classes that override `operator()`
- Functors implement `generic function concept`
Function Objects I

- Here is the "class-based" version of mult: [generic02]

  ```cpp
template <typename T>
  class mult {
  public:
    T operator() (const T& x, const T& y) const
    { return x * y; }
  }
  ```

- Use it like this:

  ```cpp
  int x = accumulate(a, a+5, 1, mult<int>());
  ```

Function Objects II

- Here is how you overload operator():

  ```cpp
class X {
  public:
    return-value operator()(arguments) const;
  }
  ```

- Use objects of class just like functions:

  ```cpp
  X fo;                 // A function object
  fo(arg1, arg2); // fo.operator()(arg1, arg2);
  ```

Why Bother?

- Functors are better than function pointers:
  - They are faster
    » Function pointers must be dereferenced
    » Function objects can be inlined
  - They are safer
    » Function objects each have their own type
  - They are smarter
    » Function objects can maintain state
How do you add a value to every object?
- Store value to add as a private member:
  ```cpp
  template <class T> class AddValue {
    T valueToAdd;
    public:
      AddValue(const T&): valueToAdd(T){};
      void operator()(T& element) const
      { element += valueToAdd; }
  };
  ```
- for_each(a, a+5, AddValue<int>(10));

Maintaining State

Functor Concepts

- The STL defines several functor concepts
  - These are categories [like iterator categories]
  - generators, unary, and binary functors
    - Functions that can be called with 0, 1, or 2 args
  - predicates and binary predicates
    - Functions that return bool [1 arg and 2 arg]
  - STL algorithms specify category of functor required
    - for_each(), for instance, requires a unary functor

STL Algorithms

- The STL contains a number of generic functions:
  - Work on any container that meets requirements
    - Can't apply sort() to a list, for instance
- These algorithms fall into four categories
  - Non-mutating sequence operations [no change]
  - Mutating sequence operations [changes container]
  - Sorting and related operations
  - General numeric operations
I
Some algorithms work "in-place" [generic04]
- sort(a, a+5);

I
Some algorithms send their output elsewhere
- copy(a, a+5, b);

I
Some algorithms can do both
- transform(a, a+5, b, sort);
- transform(a, a+5, a, sort);

In-place vs. Copying I

I
Some algorithms have both versions
- Depends upon efficiency considerations
  - No version of sort_copy() [just do both]
  - reverse(), reverse_copy()

I
Some algorithms take an optional predicate
- Two functions take same number of arguments?
  - Then create version with _if suffix [copy_if]
- Additional argument? Don't use suffix

In-place vs. Copying II

I
The STL defines several elementary functors
- List on page 846 of your text [not in Appendix G]
- Examples: [from header <functional>]
  - less<T> binary predicate: returns true if a < b
  - negate<T> unary : returns -a
  - plus<T> binary : returns a + b
- Example: the sort() generic function [generic05]
  sort(v1.begin(), v1.end(), greater<int*>());

Predefined Functors

Note: The arrows (») indicate examples and notes that are not included in the content.
An adapter is a class that changes the interface.

Suppose you have a binary predicate like this:

```cpp
template <class T>
bool Greater(const T& val, const T& min)
{ return val > min; }
```

Want to use it where a unary predicate is required?

```cpp
remove_copy_if(a, a+5, b, Greater<int>());
```

Doesn't work, `remove_copy_if()` passes single arg.

If you wrap this up in a class, it works:

```cpp
template <class T> class Greater { T val;
public:
    Greater(const T& v=T()) : val(v) {} 
    bool operator()(const T& val2) { return val2 > val; }
};
```

Then you call `remove_copy_if()` like this:

```cpp
remove_copy_if(a, a+5, b, Greater<int>(10));
```

The built-in functors already allow this.

These are called adaptable types.

Create a new class of type `binder1st` or `binder2nd`.

Normally use functions `bind1st()` and `bind2nd()`.

Here's the same example using `bind2nd()`:

```cpp
remove_copy_if(a, a+5, out bind2nd(less_equal<double>(), 100.0));
```
Homework

- Due Sunday, May 23
- Homework 28: Concordance
  - Use map and set classes
  - Print line numbers where each word appears
- Homework 29: Functors
  - Write functors average and variance
  - Two versions of operator()
    - Single argument version used with for_each
    - No-arg version returns calculated value