CMSC202
Computer Science II for Majors

Lecture 19 and 20 –
STL and Iterators

Dr. Katherine Gibson

Last Class We Covered

- Templates
  - How to implement them
  - Possible problems (and solutions)
  - Compiling with templates
- Bits & Pieces
  - Initialization lists
  - The “grep” command
  - Redirecting input and output

Any Questions from Last Time?

Today's Objectives

- STL
  - Standard Template Library
  - Containers
- Iterators
  - Purpose
  - Manipulating

- STL is the Standard Template Library

- STL contains many useful things, including…
  - Containers
  - Iterators

  - Both are ***templated***, which means we can use them with any type of data we want

Why Use the STL?

- Good programmers know what to write
- Great programmers know what to reuse

- STL provides reusable code
- Linked list, vector, map, multimap, pair, set, multiset, queue, stack, etc.
- Don't reinvent the wheel
  - Unless we tell you to!

STL Containers

STL Containers

- All containers support a few basic methods
  - `size()`
  - `empty()`
  - `clear()`
- All containers are implemented as a class

STL Containers

- Vectors
  - Dynamic (size can be changed)
  - Sequential container (elements in an order)
  - Allows random access
    - Using `[]` or `.at()`

STL Containers

- Lists
  - Linked List, (not the "list" in Python)
  - Sequential (elements in an order)
  - Does **not** support random access
  - Basic functions include:
    - `insert()`
    - `push_back()` / `push_front()`
    - `pop_back()` / `pop_front()`
    - `erase()`

STL Containers

- Sets
  - Elements are sorted when added to the set
    - Uses **`operator<`** by default
  - Cannot change the value of an element once added
  - No random access
  - Basic functions include:
    - **`insert()`**
    - **`count()`**
    - **`find()`**
    - **`erase()`**

STL Containers

- Multisets
  - Same as a set, but…
    - Allow duplicate elements
  - Elements are sorted when added to the set
    - Uses **`operator<`** by default
  - Cannot change the value of an element once added
  - No random access
  - Same basic functions as well

STL Containers

- Pairs
  - Connects two items into a single object
  - (Sort of like a tuple in Python)
  - Member variables:
    - `first`
    - `second`
  - Pair containers are used by other containers

Examples of Using Pair

- To combine an int and a string into a pair

```
pair<int, string> ex1( 5, "hello");
```

- You can then access the values in the pair using standard "dot" notation

```
cout << ex1.second << endl;  // "hello"
```

Examples of Using Pair

- A function template named **`make_pair()`** can be used to create pair objects

```
pair<int, string> ex2 =
                  make_pair(7, "ciao");
```

- A pair can be made with any two pieces of information (doesn't have to be int and string)

STL Containers

- Maps
  - Stores key/value pairs
  - Sorts by key
    - Key must be unique
    - Key is not modifiable
    - Value is modifiable

STL Containers

- Multimaps
  - Stores key/value pairs
  - Sorts by key (allows duplicate keys)
    - Key **does not** need to be unique
    - Key is not modifiable
    - Value is modifiable

Map and Multimap Functions

- Basic functions of Maps include:
  - `insert()`
  - `count()`
  - `find()`
  - `erase()`

Iterators

Iterators

- Problem
  - Not all STL classes provide random access
  - How do we do “for each element in X”?
- Solution
  - Iterators
    - “Special” pointers
  - “Iterate” through each item in the collection
- Also: encapsulation
  - User shouldn’t need to know how it works

About Iterators

- Allows the user to access elements in a data structure using a familiar interface, regardless of the internal details of the data structure

- An iterator should be able to:
  - Move to the beginning (first element)
  - Advance to the next element
  - Return the value referred to
  - Check to see if it is at the end

Kinds of Iterators

- Forward iterators:
  - Using ++ works on iterator
- Bidirectional iterators:
  - Both ++ and -- work on iterator
- Random-access iterators:
  - Using ++, --, and random access all work with iterator
- These are "kinds" of iterators, not types!

Iterators

- Essential operations
  - **`begin()`**
    - Returns an iterator to first item in collection
  - **`end()`**
    - Returns an iterator ONE BEYOND the last item in collection
    - Why does it do this?
      - If the collection is empty, begin() == end()

Constant and Mutable Iterators

- Behavior of the dereferencing operator dictates if an iterator is constant or mutable

- Constant iterator:
  - Cannot edit contents of container using iterator

- Mutable iterator:
  - Can change corresponding element in container

Constant Iterators

- Constant iterator:
  - * produces read-only version of element
  - Can use *p to assign to variable or output, but cannot change element in container

- *e.g.*, *p = <anything>; is illegal
  - *p can only be on the right hand side of the assignment operator

Mutable Iterators

- Mutable iterator:
  - *p can be assigned value
  - Changes corresponding element in container

- *i.e.*: *p returns an lvalue
  - *p can be on the left hand side of the assignment operator
  - (and the right hand side)

Vector Example

- Here's a very basic example of using an iterator to move through a vector:

```
vector<int> v; // fill up v with data...

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

- This basic example should work regardless of the container type!

Set Example

```
int main ( )
{
   set<int> iSet;

   iSet.insert(4);
   iSet.insert(12);
   iSet.insert(7);


   // this looping construct works for all containers

   set<int>::const_iterator position;

   for (position = iSet.begin(); position != iSet.end();
        ++position)
   {
      cout << *position << endl;
   }
   return 0;
}
```

Map Example

```
int main ( )
{
   // create an empty map using strings
   // as keys and floats as values
   map<string, float> stocks;

   // insert some stock prices
   stocks.insert( make_pair("IBM",  42.50));
   stocks.insert( make_pair("XYZ",   2.50));
   stocks.insert( make_pair("WX",    0.50));

   // instantiate an iterator for the map
   map<string, float>::iterator position;

   // print all the stocks
   for (position = stocks.begin(); position != stocks.end(); ++position)
      cout << "( " << position->first << ", " << position->second << " )\n";

   return 0;
}
```

Iterators - Overloaded Operators

- `*` Dereferences the iterator

- `++` Moves forward to next element
- `--` Moves backward to previous element

- `==` True if two iterators point to ***same*** element
- `!=` True if two iterators point to ***different*** elements
- `=` Assignment, makes two iterators point to same element

Reverse Iterators

- The easiest way to iterate through a container in reverse is to use a **`reverse_iterator`**

```
reverse_iterator p;
for (rp =  container.rbegin();
     rp != container.rend();  rp++)
   cout << *rp << " " ;
```

- When using a reverse iterator, use **`rbegin()`** and **`rend()`** instead of **`begin()`** and **`end()`**

Practice Problems

- Create a vector of integers
- Using an iterator and a loop
  - Change each integer to be the value of its square
- Using an iterator and a second loop
  - Print each item in reverse order

Announcements

- SCEQs next time
  - Very important metric – please fill them out!
- Project 5 is out
  - Due May 5th by 9:00 PM

- Final Exam is…
  - May 17th (Tuesday) 3:30 to 5:30 PM
  - Lecture Hall 1 (here)
  - Comprehensive!