601.220 Intermediate Programming

Writing iterators

Linked list of ints

```
// ListNode.h:
#include <iostream>

class ListNode {
  public:
    ListNode(int val, ListNode *nxt) : data(val), next(nxt) {}

//  private:  //usually private but public for this example
    int data;
    ListNode *next;
};
```

Linked list of ints

```
// ListDriver.cpp:

#include <iostream>
#include <string>
#include "ListNode.h"

int main() {

  ListNode l3(3, nullptr);
  ListNode l2(2, &l3);
  ListNode l1(1, &l2);

  //Run through all items in list, output them one by one
  for (ListNode* cur = &l1; cur != nullptr; cur = cur->next) {
     std::cout << cur->data << " ";
  }

}
```

```
$ g++ -std=c++11 -Wall -Wextra -pedantic ListDriver.cpp
$ ./a.out
1 2 3
```

MyVector class example

```
// MyVector.h:
#include <iostream>
#include <string>

class MyVector {
  public:
    MyVector(): data(new int[5]), capacity(5), num_elts(0) { }
    void add(int item);

//  private:  //but public for this example
    int* data;
    int capacity;
    int num_elts;
};

void MyVector::add(int item) {
     if (num_elts >= capacity) {
        /* then double the size of the array - code not shown */
     }
     data[num_elts++] = item;
}
```

MyVector class example

```
// MyVectorDriver.cpp:

#include <iostream>
#include "MyVector.h"

int main() {

  MyVector v = MyVector();
  v.add(1);
  v.add(2);
  v.add(3);


  //Run through all items in list, output them one by one
  for (int i = 0; i != v.num_elts; i++) {
     std::cout << v.data[i] << " ";
  }

}
```

```
$ g++ -std=c++11 -Wall -Wextra -pedantic MyVectorDriver.cpp
$ ./a.out
1 2 3
```

Iterating over containers is common

- In both classes, we needed to loop over all elements in the "list"
  - In our example, we printed items, but we might have been, say, searching for a value
- Code to "run through all elements" looks very different (cur pointer that advances through linked list vs. for loop over integer indices of vector)
- C++ iterators unify these different code segments
  - Regardless of the container specifics, an iterator feels like a pointer to successive individual elements, that we can easily advance

Iterators

- We use an `iterator` over a container to traverse elements in the container in order from beginning to end
- A `reverse_iterator` can be used to traverse elements in a backwards direction
- A `const_iterator` is an iterator which promises not to modify individual elements as it progresses through them

How can we define our own iterator for a custom class?

- Suppose we write a new container class from scratch to represent, say, a deck of cards.
  - It would be nice to have an iterator for the deck!
- Let's write one. . .

How can we define our own iterator?

- Is an iterator really just a pointer?
  - A pointer might work for a container where elements are laid out contigulously in memory, e.g. for an array
  - But a pointer doesn't work well for say, std::map. How would ++it advance properly?
- Instead, we actually define an entirely new class to represent an iterator...

Using a nested class to define an iterator

- We can write our own `iterator` (or `const_iterator` or `reverse_iterator`) as a *nested class* inside the container class
- A nested class sits inside another class definition, and has access to the members of the enclosing class, including private members
    - For our purposes, we don't need access to the private members; each iterator class simply wraps a layer of operator overloads around a pointer

How do we use an iterator?

Suppose we want to output the elements in some container c:

```
for (MyContainerType::iterator it = c.begin();
     it != c.end();
     ++it) {

  //*it can now be used to refer to each successive element
  std::cout << *it << " ";
}
```

What operators does our `iterator` class need to overload?

Minimally:

- inequality:
  - `operator!=`
- dereference:
  - `operator*`
- preincrement:
  - `operator++`

That's all we need for today, but a real-world iterator might additionally handle:

- equality: `operator==`
- arrow (for class member access): `operator->`

Implementing an iterator, continued

- Our enclosing (container) class should then also define methods named `begin` and `end`, which return iterators to the first item in the collection, and the just-past-last element in the collection, respectively

Other types of iterators

- What would need to be different for a `const_iterator`?
  - Hint: definition of `operator*` needs to change
- What would need to be different for a `reverse_iterator`?
  - Hint: definition of `operator++` needs to change, begin and end too