601.220 Intermediate Programming

Spring 2023, Day 37 (April 24th)

Today's agenda

- Day 37 recap questions
- Exercise 37

Reminders/Announcements

- Final project due by 11pm on **Friday, April 28th**

Day 37 recap questions

1. Why use iterators?
2. What are the bare minimum operators that need to be overloaded by an iterator?
3. When won't a simple pointer correctly iterate through a collection?
4. Given a container class, how/where should its iterator class be specified?
5. In addition to defining the iterator class, what else should the container do to support iterators?
6. What might go wrong if we don't also define a `const_iterator` for a container?

1. Why use iterators?

When implementing a *collection class* (a class whose instances are intended to store a collection of data values), implementing iterators provides a uniform way to access the data values in the collection.

Advantages:

- User of the container accesses its data using the same techniques as other container types
  - . . . regardless of the underlying data structure
- Container works with STL algorithms and other generic functions designed to work with containers

2. What are the bare minimum operators that need to be overloaded by an iterator?

At a mininum:

- `*` (dereference)
- `++` (advance)
- `==` and `!=` (compare iterators)

3. When won't a simple pointer correctly iterate through a collection?

A pointer type works as an iterator only if the underlying storage for the values in the collection is an array.

If the collection uses a different data structure (such as a linked list), the member where elements are stored isn't guaranteed to be contiguous.

4. Given a container class, how/where should its iterator class be specified?

The `iterator` and `const_iterator` types for the collection must be *members* of the collection.

Example of defining iterator types

```
class MyCollection {
public:
  class iterator {
    // ... members of iterator ...
  };

  class const_iterator {
    // ... members of const_iterator ...
  };

  iterator begin() { return /* begin iterator */ }
  iterator end() { return /* end iterator */ }

  const_iterator begin() const { return /* begin iterator */ }
  const_iterator end() const { return /* end iterator */ }

  // ...
```

What an iterator type could look like

```
// Assume that EltType is the
// element type of the collection
class iterator {
public:
  // ...constructor(s), destructor...

  EltType& operator*();
  iterator& operator++();
  bool operator!=(const iterator &rhs) const;
  bool operator==(const iterator &rhs) const;

private:
  // ...member variables ...
};
```

5. In addition to defining the iterator class, what else should the container do to support iterators?

The container should have appropriate `begin()` and `end()` member functions (as shown on previous slide.)

The `begin()` function returns an `iterator` (or `const_iterator`) positioned at the first element of the collection.

The `end()` function returns an `iterator` (or `const_iterator`) positioned just past the last element of the collection. (I.e., advancing an iterator which is positioned at the last element yields an iterator which is equal to the end iterator.)

6. What might go wrong if we don't also define a `const_iterator` for a container?

Since normal `iterators` can modify data values in the collection, they can't be used on a `const` collection object.

This is a problem if the collection is passed by const reference:

```
void analyze_data(const MyCollection &coll) {
  // this won't work because coll is const
  for (MyCollection::iterator i = coll.begin();
       i != coll.end();
       ++i) {
    // ...
  }
}
```

Exercise 37

- Implement an `iterator` type for the `MyList` class
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes