Lecture 6b: Implementing Iterators

CS 70: Data Structures and Program Development

Thursday, Februrary 27, 2020

C++ Details

Operator Overloading

```
x[17]     // calls  x.operator[](17)
x = 17    // calls  x.operator=(17)
x == 17   // calls  x.operator==(17)
x != 17   // calls  x.operator!=(17)
x * 17    // calls  x.operator*(17)
*x        // calls  x.operator*()
++x       // calls  x.operator++()
--x       // calls  x.operator--()
```

Accessing `private` data

```
class C {
 public:
   int getData();
 private:
   int data_;
};


int C::getData() {
   // OK: Code for class C has full access to data_
   return this->data_;  // or just "return data_;"
}
```

Accessing `private` data in other objects

```
class C {
 public:
   bool operator==(const C& rhs) const;
 private:
   int data_;
};


bool C::operator==(const C& rhs) const {
   // OK: Code in C has full access to any C object
   return (this->data_ == rhs.data_);
}
```

Accessing `private` data

```
class C {
  private:
   int data_;
};
class D {
   int peek(C other);
};

int D::peek(C other) {
   // ERROR: Code in D can't access a C's data
   return other.data_;
}
```

friend-ship

```
class C {
 private:
   int data_;
 friend class D;
};
class D {
   int peek(C other);
};

int D::peek(C other) {
   // OK: C has announced we're its friend
   return other.data_;
}
```

`friend`-ship is one-way

```
class C {
 private:
   int data_;
};
class D {
   int peek(const C& other);
 friend class C;
};

int D::peek(C other) {
   // ERROR: C does not agree that D is its friend
   return other.data_;
}
```

Nested Classes

```
class LinkedList {
    // ...LinkedList stuff...

    class Node {
      // ...Node stuff...
    };

    // ...LinkedList stuff...
};
```

- Defines classes `LinkedList` and `LinkedList::Node`.
- `LinkedList::Node` can be `public` or `private` as we choose.
- Nesting doesn't imply `friend`-ship in either direction.

Review: Iterators

The Problem

We want to allow access to all members of some collection.

Constraints:

- We want a “standard” interface applicable to many collections
- Random access (subscripting) may be wildly inefficient.

Idiomatically looping through collections

```
// Print the integers in vector<int> v
for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
    cout << *i << endl;

// Print characters of string s
for (string::iterator i = s.begin(); i != s.end(); ++i)
    cout << *i << endl;

// Print strings of set<string> t
for (set<string>::iterator i = t.begin(); i != t.end(); ++i)
    cout << *i << endl;
```

Suppose `CollectionA` is a collection of `doubles`

```
// Print all the doubles in c, an object of class CollectionA
for (CollectionA::iterator i = c.begin(); i != c.end(); ++i)
    cout << *i << endl;
```

1. What must we implement in the class `CollectionA`?
2. What must we implement in the nested class `CollectionA::iterator`?
3. How can we check whether `CollectionA` is empty?

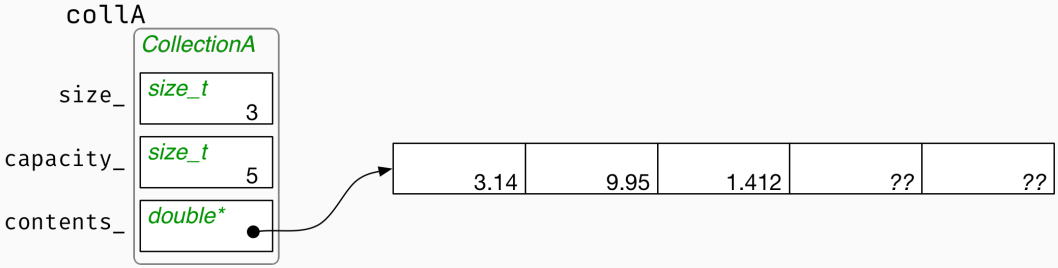
Thinking about iterators: key questions

- What data should we store in an `iterator` (to keep track of where we are)?
- What data is in the `begin()` iterator?
- What do the `iterator` operations do with this data?
  - In `operator!=`
  - In `operator*`
  - In `operator++`
- What data is in the `end()` iterator?

Implementing Iterators

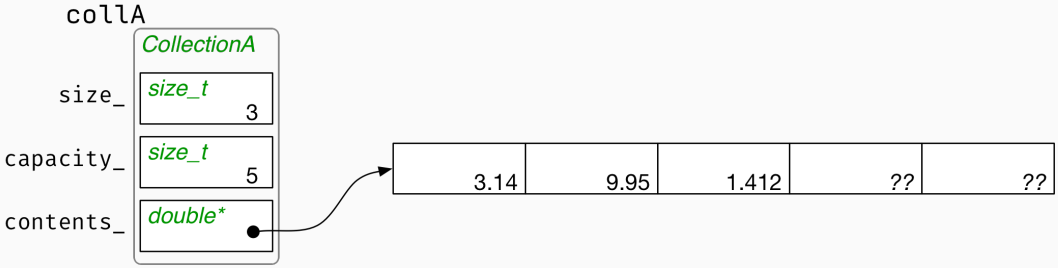
CollectionA

Suppose the class `CollectionA` stores `doubles` in a vector-like fashion.



CollectionA

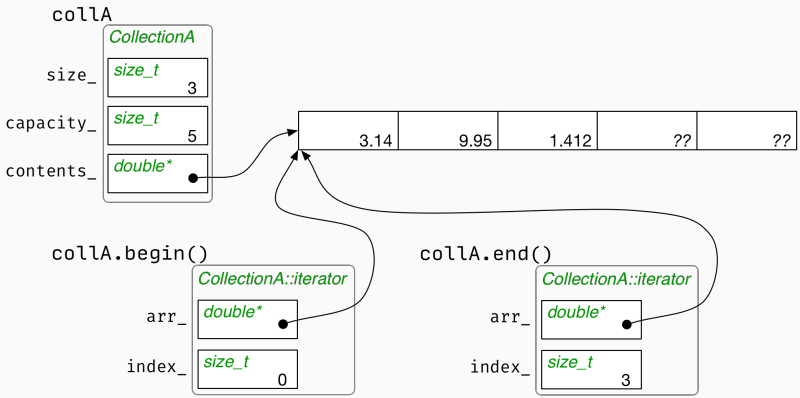
Suppose the class `CollectionA` stores `doubles` in a vector-like fashion.



There are lots of possible iterator designs.

- Today: a pointer to the array plus an integer index.

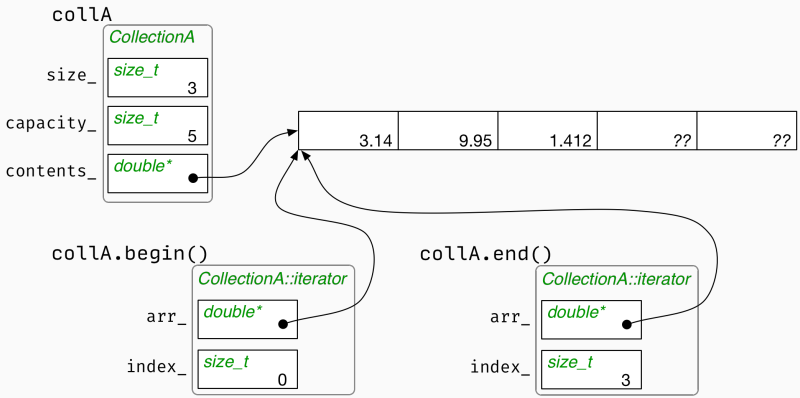
Implementing `begin` and `end`




```
// e.g., collA.begin()
CollectionA::iterator CollectionA::begin()
{
    iterator b{  ???, ???  };
    return b;
}
```

```
// e.g., collA.end()
CollectionA::iterator CollectionA::end()
{
    iterator e{  ???, ???  };
    return e;
}
```

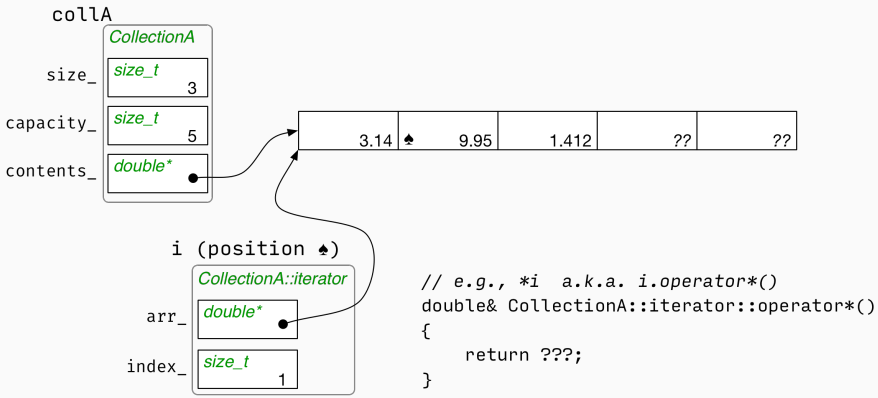
Implementing `begin` and `end`




```
// e.g., collA.begin()
CollectionA::iterator CollectionA::begin()
{
    iterator b{ contents_, 0 };
    return b;
}
```

```
// e.g., collA.end()
CollectionA::iterator CollectionA::end()
{
    iterator e{ contents_, size_ };
    return e;
}
```

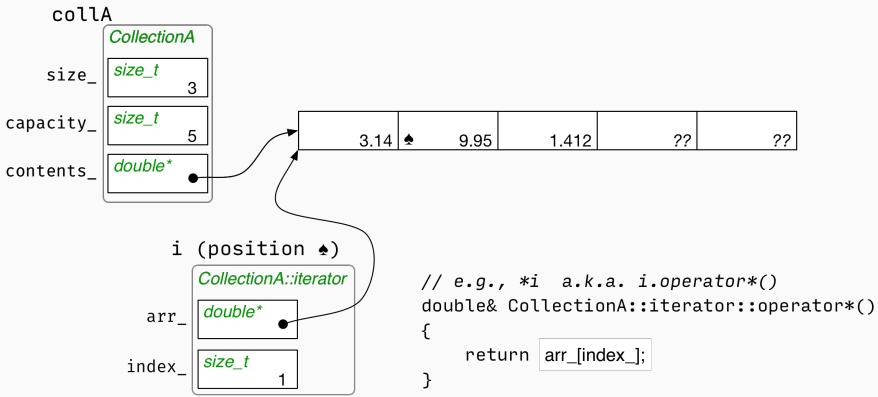
Implementing `operator*` and `operator++`




```
// e.g., *i  a.k.a. i.operator*()
double& CollectionA::iterator::operator*()
{
    return ???;
}
```

```
// e.g., ++i  a.k.a. i.operator++()
CollectionA::iterator& CollectionA::iterator::operator++()
{
    ???
    return *this;
}
```

Implementing operator* and operator++




```
// e.g., *i  a.k.a. i.operator*()
double& CollectionA::iterator::operator*()
{
    return arr_[index_];
}
```

```
// e.g., ++i  a.k.a. i.operator++()
CollectionA::iterator& CollectionA::iterator::operator++()
{
    ++index_;
    return *this;
}
```

Polishing Our Iterators

C++ Feature: Type Abbreviations

We create helpfully-named synonyms for existing types.

```
using cowptr_t = Cow*;

cowptr_t cp = new Cow;
```

But we capitalize our class names. . .

```
class CollectionA {
public:
    class iterator { ... };
    iterator begin();
    iterator end();
    // ...etc...
};

class CollectionA {
private:
    class Iterator { ... };
public:
    using iterator = Iterator;
    iterator begin();
    iterator end();
    // ...etc...
};
```

Useful STL algorithms

```cpp
#include <algorithm>
#include <numeric>
#include <vector>

void demo(const std::vector<int>& v) {
   int sum = std::accumulate(v.begin(), v.end(), 0);

   int has_42 =
        std::find(v.begin(), v.end(), 42)  !=  v.end();
   // ...
}
```

Making an STL-compatible iterator

```
#include <iterator>
#include <cstddef>

class CollectionA {

  class iterator {
   public:
     using value_type        = double;       // (!)
     using reference         = value_type&;
     using pointer           = value_type*;
     using difference_type   = ptrdiff_t;
     using iterator_category = std::forward_iterator_tag;
     ...as before...
  };

  ...as before...
};
```

Valid and Invalid Iterators

Which are OK?

```
std::string s{"hello"};
std::string::iterator i;

std::cerr << s << std::endl;       // OK?
std::cerr << *i << std::endl;      // OK?
```

Which are OK?

```
void processAnyString(std::string s)
{
   std::string::iterator i = s.begin();

   // debugging output
   std::cerr << s << std::endl;   // OK?
   std::cerr << *i << std::endl;  // OK?

   // ...do the work...
}
```

Which are OK?

```
std::string::iterator i;

{
    std::string s{"hello"};
    i = s.begin();
    std::cerr << *i << std::endl;   // OK?
}

std::cerr << s << std::endl;        // OK?
std::cerr << *i << std::endl;       // OK?
```

Which are OK?

```
std::string s{"hello"};
std::string::iterator i = s.begin();

s.push_back('!');

std::cout << s << std::endl;   // OK?
std::cout << *i << std::endl;  // OK?
```

But it can depend on the data structure...

```
std::vector<int>::iterator i = v.begin();

v.push_back(42);

std::cout << v[0] << std::endl;    // OK?
std::cout << *i << std::endl;  // OK?
```

But it can depend on the data structure...

```
std::list<int> s{1,2,3};
std::list<int>::iterator i = s.begin();

s.push_front(0);
s.push_back(4);

std::cout << *i << std::endl;  // OK!
```

Which are OK?

```
std::list<int> s{1,2,3};
std::list<int>::iterator i = s.begin();

++i;
s.erase(i);

std::cout << *i << std::endl; // OK?
```

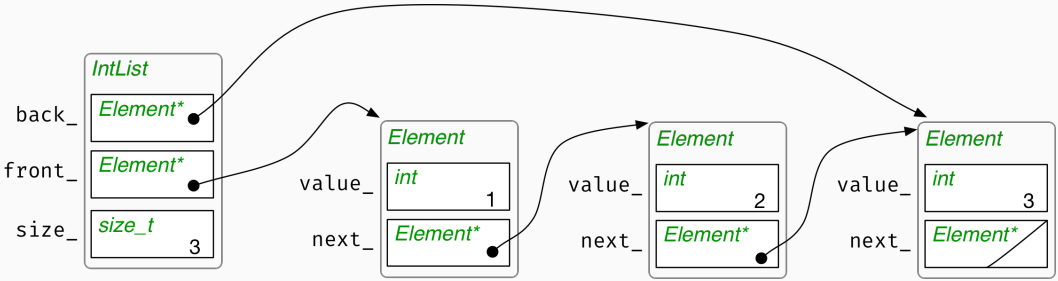
Clever Workaround

```
std::list<int> s{1,2,3};
std::list<int>::iterator i = s.begin();

++i;
std::list<int>::iterator j = s.erase(i);
//erase returns a valid iterator (to the next position)!

std::cout << *j << std::endl; // OK!
```

What data should an `IntList::iterator` object contain?



Learning Targets

1. I can write a C++ class that supports the iterator idiom.
2. I can write C++ code that uses iterators.
3. I am ready to start Homework 5.