EECS 280 – Lecture 20

Functors, Function Pointers, and Impostor Syndrome

3/28/2022

Review: Traversal by Iterator

- Walk an **iterator** across the elements.
- To get an element, just dereference the iterator!
- Get iterators that define the range from the container using `begin()` and `end()` functions.

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);


List<int>::Iterator end = list.end();
for (List<int>::Iterator it = list.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

Ask the List for iterators that define the sequence of elements.

Review: The Iterator Interface

- Iterators provide a common interface for iteration.
  - A generalized version of traversal by pointer.
  - An iterator "points" to an element in a container and can be "incremented" to move to the next element.

- Iterators[1] support these operations:
  - Dereference – access the current element.
    `*it`
  - Increment – move forward to the next element.
    `++it`
  - Equality – check if two iterators point to the same place.
    `it1 == it2`
    `it1 != it2`

1 There are many different kinds of iterators. These operations are specifically required for *input iterators*.

Review: List Iterator

```
template <typename T>
class List {
public:
  ...
  class Iterator {
    friend class List;
  public:
    Iterator() : node_ptr(nullptr) { }
    T & operator*() const;
    Iterator & operator++();
    bool operator==(Iterator rhs) const;
    bool operator!=(Iterator rhs) const;
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }
  Iterator end() { return Iterator(); }
  ...
};
```

Example: any_of_odd

- Implement this function that checks if there are any odd elements in a sequence.

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if any element in the
//           sequence [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  for (Iter_type it = begin; it != end; ++it) {
    if (*it % 2 != 0) { return true; }
  }
  return false;
}
```

We could use some other container here too!

```
int main() {
  List<int> list; // Fill with numbers
  cout << any_of_odd(list.begin(), list.end());
}
```

Alternate Solutions: any_of_odd


```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if any element in the
//           sequence [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  while (begin != end) {
    if (*begin % 2 != 0) { return true; }
    ++begin;
  }
  return false;
}
```

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if any element in the
//           sequence [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  for ( ; begin != end; ++begin) {
    if (*begin % 2 != 0) { return true; }
  }
  return false;
}
```

A General any_of Function

- If we were to write an any_of_even function, it would look very similar.
  - Code duplication is bad!
- The only piece of code that needs to change is the test for oddness/evenness.

```
if (*it % 2 != 0) { return true; }
```

Reducing Code Duplication

Bad

```
int times2(int x) {
  return x * 2;
}

int times3(int x) {
  return x * 3;
}

int times4(int x) {
  return x * 4;
}

int main() {
  cout << times2(42);
  cout << times3(42);
  cout << times4(42);
}
```

Good

```
int times(int x, int n) {
  return x * n;
}

int main() {
  times(42, 2);
  times(42, 3);
  times(42, 4);

  for (int i = 0; i < 10; ++i) {
    cout << times(42, i);
  }

}
```

Idea: Function Parameters

```
bool is_even(int x);
bool is_odd(int x);
bool is_prime(int x);

template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end, ____ fn) {

  for (Iter_type it = begin; it != end; ++it) {
    if ( fn(*it) ) { return true; }
  }

  return false;
}

int main() {
  List<int> list; // Fill with numbers
  cout << any_of(list.begin(), list.end(), is_prime);
}
```

A parameter that takes a function!

The function tells us "yes" or "no" for each item.

Pass in the function we want to use.

Function Pointers

- Variables that can store a function.
- The instructions for executing a function are stored somewhere…a function pointer actually stores the **address** where the function is stored.

Unlike a regular pointer, you don't need to dereference a function pointer to use it.

The syntax for declaring a function pointer looks a lot like the functions it can hold.

In a declaration, the * means "a pointer to". Parentheses are needed due to precedence.

```
int max(int x, int y) { return x > y ? x : y; }
int min(int x, int y) { return x < y ? x : y; }

int (*fn)(int, int) = max; // fn starts as max
fn = min; // set fn to be min
cout << fn(2, 5); // uses min, prints 2
```

Takes two ints.

Returns an int.

Declaration Syntax

- C++ declarations are read "inside out".
- Postfix operators ([], ()) have higher precedence than prefix (*, &).

- Example: `double (*func)(int)`

`double (*func)(int)`	func is
`double (*func)(int)`	a pointer to
`double (*func)(int)`	a function
`double (*func)(int)`	that takes an `int`
`double (*func)(int)`	and returns a `double`

Function Pointers for any_of

- Basically, "A variable that holds a function."

```
bool is_odd(int x) {
  return x % 2 != 0;
}
```

"fn is a pointer to a function that takes an int and returns a bool."

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if there is an element in
//           [begin, end) for which fn returns true.
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end,
            bool (*fn)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (fn(*it)) { return true; }
  }
  return false;
}
```

The type of is_odd matches the type of the fn parameter.

```
int main() {
  List<int> list; // Fill with numbers
  cout << any_of(list.begin(), list.end(), is_odd);
}
```

Predicates

- A **predicate** is a function that returns either `true` or `false` depending on whether its argument has some property.
  - `is_odd`, `is_even`, `is_prime`, etc.
- Thus, a better name for the third parameter to `any_of` might be `pred`.

```
bool any_of(Iter_type begin, Iter_type end,
            bool (*pred)(int));
```

- The strategy here is to write a general function like `any_of` and customize its behavior with a predicate.

Making Predicates

- What if we wanted to use `any_of` to check for elements greater than some threshold?
  - Greater than 0?

```
bool greater0(int x) {
  return x > 0;
}
```

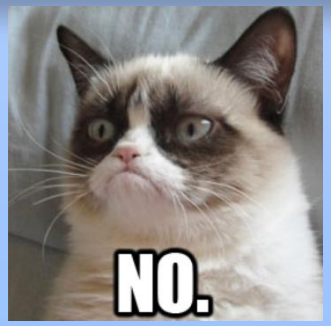
  - Greater than 32?

```
bool greater32(int x) {
  return x > 32;
}
```

  - Greater than 212?

```
bool greater212(int x) {
  return x > 212;
}
```

- Is this a good approach?



Exercise

```
int main() {
  List<int> list; // Fill with numbers
  cout << any_of(list.begin(), list.end(), is_odd);
}
```

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if there is an element in
//           [begin, end) for which fn returns true.
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end,
            bool (*fn)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (fn(*it)) { return true; }
  }
  return false;
}
```

```
bool is_odd(int x) {
  return x % 2 != 0;
}
```

```
bool greater32(int x) {
  return x > 32;
}
```

```
bool greater212(int x) {
  return x > 212;
}
```

Question 19.1

What can we do to fix the problem?

A) Make a single `greater` function that uses a global variable to store the threshold.

B) Make a single `greater` function with an extra parameter to pass in the threshold.

C) Add an extra parameter to the `any_of` function to pass in the threshold.

D) These are all good ideas.

E) None of these are good ideas.

Functions and First-Class Objects

- A **first-class object**…
  - …has state. (i.e. stores some information)
  - …can be created at runtime.
  - …can be passed as an argument.
  - …etc.
- C++ does not have first-class functions. This gives us two big limitations:
  - Functions can't really store information.
  - You can't just make a function at runtime.

Functors

- We would like to be able to make functions on the fly, but we can't.
- Instead, we'll make class-type objects that pretend to be functions.
  - We'll overload the **function call operator**.
  - Yes, you can do this!

- An **iterator** is an object that acts like a **pointer**.
- A **functor** is an object that acts like a **function**.

Implementing a GreaterN Functor

```
class GreaterN {
private:
  int threshold;

public:
  GreaterN(int threshold_in)
    : threshold(threshold_in) { }

  bool operator()(int n) const {
    return n > threshold;
  }
};
```

A GreaterN object stores a threshold as a member variable.

The threshold is passed into the constructor.

The overloaded function call operator takes an int and tests it.

```
int main() {
  GreaterN g0(0);
  GreaterN g32(32);
  GreaterN g212(212);

  cout << g0(-5); // false
  cout << g0(3);  // true

  cout << g32(9); // false
  cout << g32(45); // true
}
```

```
int main() {
  List<int> list; // Fill with numbers
  GreaterN g0(0);
  cout << any_of(list.begin(), list.end(), g0);
  cout << any_of(list.begin(), list.end(), GreaterN(32));
}
```

See file L19.1_GreaterN at lobster.eecs.umich.edu.

Functors as Parameters

- To pass a functor into **any_of**, we add a template parameter for the type of the functor.

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if there is an element in
//           [begin, end) for which pred returns true.
template <typename Iter_type, typename Predicate>
bool any_of(Iter_type begin, Iter_type end,
            Predicate pred) {
  for (Iter_type it = begin; it != end; ++it) {
    if (pred(*it)) { return true; }
  }
  return false;
}
```

```
int main() {
  List<int> list; // Fill with numbers
  GreaterN g0(0);
  cout << any_of(list.begin(), list.end(), is_odd);
  cout << any_of(list.begin(), list.end(), g0);
  cout << any_of(list.begin(), list.end(), GreaterN(32));
}
```

We can still use any_of with a function pointer.

Break Time!

We'll start again in five minutes.

Are you an impostor?

What is impostor syndrome?

And how can it affect you?

Impostor Syndrome

A concept describing high-achieving individuals who are marked by an inability to internalize their accomplishments and a persistent fear of being exposed as a "fraud".[1]

1 https://en.wikipedia.org/wiki/Impostor_syndrome

You may have impostor syndrome if you…

- Attribute your success to external factors
- Fear being revealed as a fraud
- Convince yourself you are not good enough
- Have a hard time accepting compliments for accomplishments

Do you often doubt yourself?

- *"I don't belong here...everyone seems so smart"*
- *"If I'm not getting an A in 280, how am I going to survive 281?"*
- *"Nobody else here is like me – I don't belong in this class".*
- *"I don't like asking questions in discussion because I'm afraid others will realize I don't know what I'm doing."*
- *"Maybe EECS isn't for me...I should drop."*

The truth is…

- Almost no one knows what they're doing and we're all trying to figure it out!
- "…70 percent of people from all walks of life — men and women — have felt like impostors for at least some part of their careers."[1]
- Chances are you, or the person next to you, may experience impostor syndrome at some point.

Question

Have you felt like an impostor in your classes here at UM?

A) Never
B) Rarely
C) Sometimes
D) Often
E) All the time

1 Gravois, J. (2007). *You're not fooling anyone*. The Chronicle of Higher Education, 54(11), A1.

What can I do?

Stop comparing yourself to others

Encourage each other

Join student orgs and connect

Accept your accomplishments

Find a mentor

Resources

- UMICH CAPS article on Impostor Syndrome http://bit.ly/2nhy2UG
- TED talk on "How Students of Color Confront Impostor Syndrome" http://bit.ly/2i1OpxP
- American Psychological Association article on Impostor Syndrome http://bit.ly/1gH4JOs

Recall: max_element

```
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {

  Iter_type maxIt = begin;

  for (Iter_type it = begin; it != end; ++it) {
    if (*maxIt < *it) {
      maxIt = it;
    }
  }
  return maxIt;
}

int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;
}
```

Start by assuming first element is the max.

Use traversal by iterator to check each element.

If we find a larger element, update maxIt to point to it.

Which element types can this function be used with?
Those that support the < operator.

Dereference returned iterator to get the element itself.

max_element with Ducks

- Using `max_element` with a container of `Ducks` doesn't work because there is no overloaded > operator for `Duck`.
- But there are still situations where we'd like to find the "maximum" `Duck`.
  - The `Duck` whose name is last alphabetically.
  - The `Duck` with the most ducklings.
- Here's the idea:

A functor that compares two Ducks according to their names.

```
int main() {
  vector<Duck> vec; // fill with Ducks
  max_element(vec.begin(), vec.end(), DuckNameLess());
}
```

Comparators

- A **comparator** is a function that compares two arguments and returns true/false depending on their ordering.
- It's common to use "less" comparators that return true if the first argument is less than the second argument.
  - `DuckNameLess`, `DuckDucklingsLess`, etc.

```
class DuckNameLess {
public:
  bool operator()(const Duck &d1, const Duck &d2) const {
    return d1.getName() < d2.getName();
  }
};
```

max_element with Ducks

```
template <typename Iter_type, typename Comparator>
Iter_type max_element(Iter_type begin, Iter_type end,
                      Comparator less) {
  Iter_type maxIt = begin;
  for (Iter_type it = begin; it != end; ++it) {
    if (less(*maxIt,*it)) {
      maxIt = it;
    }
  }
  return maxIt;
}

int main() {
  vector<Duck> vec; // fill with Ducks
  max_element(vec.begin(), vec.end(), DuckNameLess());
}
```

Call comparator rather than using < operator.

for_each

- Many different functions iterate over a sequence and do something with each element.
  - Functors serve as an abstraction for "do something".
  - Iterators serve as an abstraction for a sequence.
- Here's code for `for_each`, which follows this idea:

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Applies func to each of the elements in
//           the sequence [begin, end) and returns func.
template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;
}
```

We return the functor in case it contains some information about the result.

Printing with for_each

```
template <typename T>
class Printer {
public:
  void operator()(const T &n) const {
    cout << n;
  }
};

template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;
}

int main() {
  List<int> list; // Fill with numbers
  for_each(list.begin(), list.end(), Printer<int>());
}
```

Exercise

Question

Which of these do you need to add?

A) Member variable
B) Constructor
C) Destructor
D) operator*
E) More than one of these

- Modify the `Printer` functor so that it can be used with any stream, and not just `cout`.
  - `Printer` should store a reference to the stream.

```
template <typename T>
class Printer {
// MODIFY THIS CLASS
public:
  void operator()(const T &x) const { os << x; }
};


int main() {
  List<int> list; // Fill with numbers
  ofstream fout("list.out");
  for_each(list.begin(), list.end(),
           Printer<int>(fout));
}
```

Solution: Printer

References must be initialized to alias an object.

```
template <typename T>
class Printer {
public:
  Printer(ostream &os_in) : os(os_in) { }

  void operator()(const T &x) const { os << x; }
private:
  ostream &os;
};

int main() {
  List<int> list; // Fill with numbers
  ofstream fout("list.out");
  for_each(list.begin(), list.end(),
           Printer<int>(fout));
}
```