Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Functors

First-class objects

- There are lots of things we can do with integers in a running program, such as:
  - Create them
  - Destroy them
  - Pass them as arguments
  - Return them as values
- In fact, we can do these four things with **all** values of the simplest types in the language, including pointers and even types we create ourselves with the `enum`, `struct`, or `class` mechanisms.
- For any entity in a programming language, we say that the entity is *first-class* if you can do all four of these things.

First-class objects

- Unlike values, types and functions are **not** first class objects in C++; but, they can sometimes come close.

- The template mechanism allows us to pass types as arguments.
- The function pointer mechanism allows us to pass functions as arguments and return them as results.

- Specifically, we cannot create and destroy functions dynamically.
  - Sometimes, that's a problem.
- Fortunately, C++ gives us a mechanism that is **almost** exactly like first-class functions, called "function objects".

Review: List

- We'll illustrate this problem by using the `List` ADT to write a generic `any_of` function using function pointers.

- Remember the operations on a singly-linked `List`, with it's associated `Iterator` class:

```
bool empty() const;
T & front() const;
void push_front(const T &datum);
void pop_front();
Iterator begin() const;
Iterator end() const;
```

Motivation

- Implement this function using the List Iterator

```
// EFFECTS: returns true if l has any odd
// element, false otherwise
bool any_odd(const List<int> &l);
```

Motivation

- Implement this function using the List Iterator

```
// EFFECTS: returns true if l has any odd
// element, false otherwise
bool any_odd(const List<int> &l) {
  for(List<int>::Iterator i=l.begin();
      i != l.end(); ++i)
    if (*i % 2) return true;

  //reached end without finding elt
  return false;
}
```

Motivation

- Implement this function using the List Iterator
- Another solution, using C++11

```
// EFFECTS: returns true if l has any odd
// element, false otherwise
bool any_odd(const List<int> &l) {
  for(auto i:l)
    if (*i % 2) return true;

  //reached end without finding elt
  return false;
}
```

Motivation

- If we write `any_even`, it looks almost exactly the same:

```
// EFFECTS: returns true if l has any even
// element, false otherwise
bool any_even(const List<int> &l) {
  for(List<int>::Iterator i=l.begin();
      i != l.end(); ++i)
    if (!(*i % 2)) return true;

  //reached end without finding elt
  return false;
}
```

We can abstract away the test inside the `if()` statement as a function and generalize it.

Review: using function pointers

- The generic function header would be something like this:

```
bool any_of(const List<int> &l, bool (*pred)(int));
```

- `any_of` takes two arguments, `l` and `pred`.
  - `l` is a reference to a `List`
  - `pred` is a pointer to a function taking a single integer argument, returning a boolean result
- A *Predicate* is used to control an algorithm
  - Input: one element
  - Output: `bool`

Predicates

- A *Predicate* is used to control an algorithm
  - Input: one element
  - Output: `bool`
- We can write a simple predicate to control a complicated algorithm that somebody else wrote
- For example, `any_of` is part of the STL (Standard Template Library), which ships with C++ implementations
  `#include <algorithm>`

Review: using function pointers

- Then, the generic implementation is:

```
bool any_of(const List<int> &l, bool (*pred)(int)) {
  for(List<int>::Iterator i=l.begin();
      i!=l.end(); ++i)
    if (pred(*i)) return true;

  //reached end without finding elt
  return false;
}
```

Review: using function pointers`

- We can use `any_of` to see if some list `l` has odd elements, by writing the appropriate predicate:

```
bool is_odd(int n) {
  return (n%2);
}

List<int> l;
// fill l ...
bool contains_odd = any_of(l, is_odd);
```

- So far, so good – nothing new here...

Motivation

How would you use `any_of` to see if a list `l` has any elements greater than 2? How would you use it to see if a list has any elements greater than 42?

```
bool any_of(const List<int> &l, bool (*pred)(int)) {
  for(List<int>::Iterator i=l.begin();
      i!=l.end(); ++i)
    if (pred(*i)) return true;

  //reached end without finding elt
  return false;
}
```

Motivation

- Write new predicates!

```
bool greater2(int n) {
  return (n>2);
}

bool greater42(int n) {
  return (n>42);
}
```

This is not good. We really ought to be able to generalize more by writing a single predicate that is "`GreaterN`", and use that single predicate no matter what the larger than target is.

Worse yet, we'd have to know how much "greater" the elements needed to be at compile time – and we might not know that!

Motivation

- Unfortunately, solving these problems of creating a `greaterN` function with function pointers requires an unsightly global variable:

```
int greaterN_limit;  //global variable
bool greaterN(int n) {
  return n > greaterN_limit;
}
```

- To use this, we would do something like this:

```
List<int> l;  // fill l ...
cin >> greaterN_limit; // get limit
cout << any_of(l, greaterN);
```

Motivation

- To avoid using a global variable, we'd want a "function-creating function“ – one that, given an integer `limit`, returns a predicate that takes one integer argument, `N`, and returns true if `N > limit`.

- Because C++ functions are not first-class, there's no way to do this with functions.

- But, we **can** do it with the class mechanism plus one neat trick – operator overloading.

Operator overloading

- Recall that many of our classes required us to define the assignment operator:

```
const List & operator=(const List &l);
```

- In our Iterator class, we defined more operators:

```
T& operator* () const;    // Get
Iterator & operator++ (); // Move
bool operator!= (Iterator rhs) const;//Compare
```

Function call operator

- It turns out that we can redefine almost any operator for a class, and the "function-call" is just another operator

- Suppose we have a class called `Greater2`, with no private members and only one public member function:

```
class Greater2 {
public:
  bool operator() (int n);
};
```

Function call operator

```
class Greater2 {
public:
  bool operator() (int n) {
    return n > 2;
  }
};
```

- This public method overloads the "function-call" operator on instances of `Greater2` – the method takes a single integer argument, and returns a Boolean result.
- It takes an integer argument and returns true if the input is greater than 2.
- Because it is a very short function, we implemented it *in line*

Using function objects

- So, we can do this:

```
Greater2 g2;
cout << g2(4) << endl;
```

```
./a.out
1
```

- The second line looks like a function call; however, `g2` is **not** a function. Rather, it is an instance of the class `Greater2`
- The class `Greater2` has defined the "function-call operator", and so that member function runs when we use the syntax `g2(4)`, passing the argument 4.
- Objects that overload the function-call operator are called "function objects", or sometimes "functors".

Implementing function objects

- So far, this is a lot like function pointers
- However, unlike functions with function pointers, objects can have per-object state, which allows us to specialize on a per-object basis.
- Let's add a member variable to `GreaterN`

```
class GreaterN {
  int limit;
public:
  bool operator() (int n) {
    return n > limit;
  }
};
```

When we create instances of `GreaterN`, we store the `limit` in a member variable, and later can test inputs against that `limit`.

Implementing function objects

- Now, we need a way to set the `limit`. Let's initialize a `GreaterN` object with a `limit` using a constructor.

```
class GreaterN {
  int limit;
public:
  GreaterN(int limit_in) : limit(limit_in) {}
  bool operator() (int n) {
    return n > limit;
  }
};
```

Using function objects

- Constructors and function call operators easy to mix up
- Constructors create new variables

```
GreaterN g2(2);              //ctor
GreaterN g6(6);              //ctor
```

- Function call operators work with existing variables

```
cout << g2(4) << endl;  //function call
  operator
cout << g6(4) << endl;  //function call
  operator
```

Using function objects

- Now, we can use this new class to "generate" specialized functors

```
GreaterN g2(2);
GreaterN g6(6);
cout << g2(4) << endl;
cout << g6(4) << endl;
```

Exercise: LessN and InRange

- Write a functor called `LessN` that returns true if an input integer is less than an input limit
  - Overload the function call operator
  - Create a constructor to initialize `limit`
- Write a functor called `InRange` that returns true if an input integer is within an inclusive range: `[min, max]`
  - Overload the function call operator
  - Create a constructor to initialize `min` and `max`
- Write a `main()` function that uses instances of `LessN`, `GreaterN` and `InRange` to check if water is a `solid`, `liquid` or `gas` at a given temperature
  - Read an integer `temp` from standard input
  - Freezing point = 32, boiling point = 212

Function objects as function inputs

- Now we have covered how to create function objects
- In order to complete our generic `any_of` function, we need to pass a functor as a function input
- We did this before using function pointers:

```
bool any_of(const List<int> &l, bool (*pred)(int));
```

- Problem: since a functor is a custom type, each functor will have its own type.
- Solution: generalize the type using templates

Templated functions review

- Recall templated functions
- This sum function works for any type `T` that can be initialized to `0` and can be added with the "+" operator.

```
template <typename T>
T sum(T a[], int size);

double a[100] = {... }
int    b[200] = {... }

double aSum = sum(a, 100);
int bSum = sum(b, 200);
```

Templated functions

- We used the sum function with any type `T`

```
template <typename T>
T sum(T a[], int size);
```

- We want to use the `any_of` function with any functor `Predicate`
- Classes are first-class types, so we can use templates to solve this problem

```
template <typename Predicate>
bool any_of(const List<int> &l, Predicate pred);
```

Templated functions

- The template is the only thing we have to change in `any_of`

Function template substitutes functor

```
template <typename Predicate>
bool any_of(const List<int> &l, Predicate pred){
  for(List<int>::Iterator i=l.begin();
      i!=l.end(); ++i)
    if (pred(*i)) return true;

  return false;
}
```

The body of `any_of` is **exactly the same** as it was before. But, rather than take a function pointer, it takes a functor.

Using function objects

- Now, we can use `any_of` with any of our Predicate functors

```
List<int> l; // fill with ( 1 2 3 )
GreaterN g2(2);
GreaterN g6(6);
cout << any_of(l, g2) << endl;
cout << any_of(l, g6) << endl;
```

```
./a.out
1
0
```

Using function objects

- Now, we can use `any_of` with any of our Predicate functors

```
List<int> l; // fill l
GreaterN g2(2);
cout << any_of(l, g2) << endl;

LessN l2(2);
cout << any_of(l, l2) << endl;

InRange ir(32, 212);
cout << any_of(l, ir) << endl;
```

Using function objects

- Another way to do it:

```
List<int> l; // fill l

cout << any_of(l, GreaterN(2)) << endl;


cout << any_of(l, LessN(2)) << endl;


cout << any_of(l, InRange(32, 212)) << endl;
```

Using function objects

- We can even get limits from the user:

```
List<int> l; // fill l

// ask user for limits
int less_limit=0, greater_limit=0;
cin >> less_limit >> greater_limit;

// create functors
LessN less(less_limit);
GreaterN greater(greater_limit);

// use functors
cout << any_of(l, less) << endl;
cout << any_of(l, greater) << endl;
```

So, the ability of objects to carry per-object state **plus** overriding the "function call" operator gives us the equivalent of a "function factory".

Comparison functors

- So far, we have written *predicates* using functors
  - A *Predicate* is used to **control an algorithm**. The input is one element, the output is a `bool`

- Another use for functors is a *comparison* operator
  - A *comparison* is used to **define order**. The inputs are two elements of the same type, the output is a `bool`

Predicate vs. comparison functors

- A *Predicate* is used to control an algorithm. The input is one element, the output is a `bool`

```
class MyPredicate {
  public:
  bool operator()(T input) { /* ... */ }
};
```

- A *comparison* is used to define order. The inputs are two elements of the same type, the output is a `bool`

```
class MyComparison {
  public:
  bool operator()(T a, T b) { /* ... */ }
};
```

Comparison functor

- A totally superfluous functor

```
class IntLess {
public:
  bool operator() (int a, int b) {
    return a < b;
  }
};

IntLess less;
cout << less(1, 21) << endl;
```

```
./a.out
1
```

Comparison functors

- An `Animal` class:

```
class Animal {
  string name;
public:
  // constructors, etc. ...
  virtual string get_name() const { return name; }
};
```

- What if we want to compare two `Animal` objects and put them in order?

Comparison functors

- Write a functor to compare two `Animal` objects by name
- Returns true if name of `a` is before `b` in alphabetical order

```
class AnimalLess {
public:
  bool operator()(const Animal &a,
                  const Animal &b) const {
    return a.get_name() < b.get_name();
  }
};

AnimalLess less;
Animal fergie("Fergie");
Animal myrtleII("Myrtle II");
cout << less(fergie, myrtleII) << endl;
```

```
./a.out
1
```

"Fergie" comes before "Myrtle II" in ASCII order

Exercise: functors + pointers

- Write `AnimalLess_ptr`, a functor that takes two `Animal` pointers as input, and produces a `bool` as output. The functor should not change the pointed-to objects
- When should you use a functor that passes its input by pointer? By const reference? By value?

```
class AnimalLess {
public:
  bool operator()(const Animal &a,
                  const Animal &b) {
    return a.get_name() < b.get_name();
  }
};
```

Exercise: comparison functors

- Now that we have `AnimalLess_ptr`, we can compare two the objects pointed to by two pointers and determine their order

```
AnimalLess_ptr less;
Animal *fergie = new Animal("Fergie");
Animal *myrtleII = new Animal("Myrtle II");
cout << less(fergie, myrtleII) << endl;
```

- What if you want to check if two `Animal*` are equal?
- Use an instance of "`AnimalLess_ptr less;`" to check if two `Animal` objects are equal