CS 421 Lecture 19 –Higher-order functions in object-oriented languages

- Function objects
- Java examples
- Parser combinators in Java
- C++ examples
- Higher-order functions in the Standard Template Library

Function objects

Also called “functors.”

Function object = object that defines one method, called **apply**.

Can use function object as if they were functions in a functional language, except that you always have to write f.apply(…) instead of f(…).

Purposes of Function Objects

- More Expressiveness
  - Sometimes simpler and more convenient than other approaches
  - High-Order Functions
- Resilient to Design Changes
  - We don’t need to change the interface
  - Create New Functionality without writing new functions.

Example: increment object

```
class Succ {
   int apply (int x) { return x+1; }
}
    …
    Succ incr = new Succ();
    …  incr.apply(3) …
```

Example: Plus class – build function objects

```
class Plus {
   int i;
   public Plus(int i) { this.i = i; }
   int apply (int x) { return x+i; }
}
    …
    Plus incr = new Plus(1);
    …  incr.apply(3) …
    Plus add4 = new Plus(4);
    … add4.apply(3) …
```

Higher-order functions: map

```
void map (Plus p, int A[]){
 for(int i = 0, i < A.length; i++)
      A[i] = p.apply(A[i]);
}
```

```
 ... map (new Plus(3), w);  // add 3 to each element of w
           (OCaml: plus = fun x -> fun y -> x+y
                  map (plus 3) lis)
```

Could also define map(Plus, List<int>)

Interfaces

This version of map can be used only to increment elements of an array by some amount. To define map so that it can apply other functions to the elements, need an interface to give the type of any integer-to-integer function object.

```
interface IntFun {
   int apply (int x);
}
```

Need to declare Plus to say that it implements this interface. Then can redefine map to take an IntFun:

Interfaces

```
interface IntFun {
   int apply (int x);
}

class Plus implements IntFun {
   ... same as above ... }

void map(IntFun f, int A[]){
    for(int i = 0, i < A.length; i++)
        A[i] = f.apply(A[i]);
}
```

Interfaces

Now, can define other integer-valued functions as function objects and use them in map:

```
class MultBy implements IntFun {
    int m;
    public MultBy (int m) {this.m = m; }
    public int apply (int x) { return m*x; }
}
... map (new MultBy(5), A);
```

More higher-order functions

Can define classes that create functions from other functions, just as in OCaml:

Java

```
class SumFuns implements IntFun {
  IntFun f, g;
  public SumFuns (IntFun f, IntFun g) {
     this.f = f; this.g = g;
  }
  public int apply (int x) {
     return f.apply(x) + g.apply(x);
  }
}

   map(new SumFuns(new Plus(3),
                                new MultBy(5), A);
```

OCaml:

```
fun add f g =
   fun x -> f x + g x;;

map (add (plus 3) (multby 5)) A
```

More examples of higher-order functions

```
class MaxOfFuns implements IntFun {
  IntFun[] funs;
  public MaxOfFuns (IntFun[] funs) {
     this.funs = funs;
  }
  public int apply (int x) {
     int max = funs[0].apply(x);
     for (int i=1; i<funs.length; i++) {
         int m = funs[i].apply(x);
         max = max>m ? max : m;
     return max;
  }
}

IntFun[] flis = new IntFun
   map(new SumFuns(new Plus(3),
                                  new MultBy(5), A);
```

Anonymous inner classes

Given interface IntFun, can define classes that implement it anonymously. These are defined within methods (of any class) using the syntax:

```
new IntFun () { int apply (int x) { … }}
```

E.g

```
map (new IntFun() { int apply (int x) { return x*x; }},
        A);
```

Aside: Event-oriented programming in Java

Inner classes are often used in Java to provide “callbacks” – methods that are called when an external event (e.g. mouseclick) occurs, e.g.

```
interface ActionListener {
        public void actionPerformed (ActionEvent e); }
…
  Button b = new Button(“Blue”);
…
  b.addActionListener(new ActionListener (){
      public void actionPerformed(ActionEvent e) {
              scr.SetBackgroundColor(Color.Blue);
      }});
```

Parser combinators in Java

```
interface Parser{  ArrayList apply(ArrayList cl); }

Parser token(String s){
    return new Parser () {
     ArrayList apply(ArrayList cl){
          if (cl == null || cl.size() == 0) then return null;
          if (s.compareTo((String)cl.get(0)) ==0) then {
               cl.remove(0);
               return cl;
          }
          return null;
     }
    }
}
... Parse parserx = token("x");
```

Parser combinators in Java

```
Parser plusplus(Parser p, Parser q){
   return new Parser () {
    ArrayList apply(ArrayList cl){
        ArrayList cl2 = p.apply(cl);
        if(cl2 == null) return null;
        else return q.apply(cl2);
}}}

Parse parserxy = plusplus(token(“x”), token(“y”));
```

Parser combinators in Java

```
Parser or(Parser p, Parser q){
    return new Parser () {
     ArrayList apply(ArrayList cl){
          ArrayList cl2 = p.apply(cl);
          if(cl2 == null) return q.apply(cl);
          else return cl2;
}}}

Parse parserxyorz = or(parserxy, token(“z”));
```

Parser combinators in Java

```
Parser parserA = or(plusplus(token(“a”), parserA), token(“b”));
```

Is it correct?

```
Parser parserwrap =
    new Parser () { Parser recParser = null;
      ArrayList apply(ArrayList cl){
            if(recParser != null) return recParser.apply(cl);
            return null;
    }};

Parser parserA = or(plusplus(token(“a”), parserwrap), token(“b”));
parserwrap.recParser = parserA;
```

Function object in C++

Idea of function objects can be used in any o-o language (although type-checking issues may be different).

C++ has feature of operator overloading. Can even overload function application. In C++, function object is sometimes defined as an object that overloads function application:

```
class Succ{
    int operator() (int x){
     return x + 1;
    }}
```

```
Succ succ = new Succ();   ...  a = succ(3); ...
```

Function object in C++

C++ does not have interfaces, but the type of the function object can be given as a template argument:

```
template<class IntFun>
void map(IntFun f, int *A, int n){
   for(int i = 0; i < n; i++)
    A[i] = f ( A[i] );
}

... map(new Succ(), arr, leng) ...
```

Higher-order functions in C++

```
template<class IntFun>
class Combine_Add{
   IntFun f, g;
   Combine_Add(IntFun f, IntFun g){
    this.f = f; this.g = g;
   }
   int operator() (int x){
    return f(x) + g(x);
   }
}
```

Higher-order functions in C++

```
template<class IntFun>
class Funmod{
   IntFun f; int x, y;
   Combine_Add(IntFun f, int x, int y){
    this.f = f; this.x = x; this.y = y;
   }
   int operator() (int z){
    if(x == z) return y;
    else return f(z);
   }
}
```

Higher-order functions in the Standard Template Library (STL)

A large section of the STL spec is devoted to function objects. Their use is complicated by type-checking issues which we have ignored in the above, but the basic ideas are as we have discussed.

One place function objects are used is as arguments to functions like sort:

sort (*collection_start*, *collection_end*, *comparator*)

where comparator is an object that redefined operator() as a bool-valued function of two arguments...

Higher-order functions in the Standard Template Library (STL)

```
class GT {
   bool operator() (double x, double y) { return x>y; }
}

... vector<double> u; .. sort (u.begin(), u.end(), GT());
```

This provide flexibility. Could define another comparator:

```
class GT_mag {
   bool operator() {double x, double y) {
                  return fabs(x) > fabs(y);
}}
```

Template parameters

For the record, we have simplified the above by omitting required type declarations. GT is actually:

```
class GT : public binary_function<double, double, bool> {
  bool operator() (double x, double y) { return x>y; }
}
```

The binary_function class doesn't have any operations, but just provides typedefs for the type parameters above; this goes beyond the scope of today's discussion. In any case, sort requires a comparator which is a subclass of binary_function, which means just that it overloads operator() as a two-argument function.

Higher-order functions in the STL

The STL provides some functions that construct function objects from other function objects. We give one example:

bind1st: Given a function object that overloads operator() as a two-argument function, and given a value for the first argument, this produces a function object that overloads operator() as a one-argument function. E.g.

Higher-order functions in the STL

```
Given:  class plus {
              double operator() (double x, double y) {
                   return x+y; }}
   can call:  transform(v1.begin(), v1.end(), v2.begin(),
                        bind1st(plus(), 3.0));
```

This is exactly analogous to using a curried function in OCaml and partially applying it:

```
let plus = fun x -> fun y -> x+.y
transform(…, plus 3.0)
```