

CSC 347 - Concepts of Programming Languages

Closures

Instructor: Stefan Mitsch



Learning Objectives

❓ How to bundle data and functions?

- Understand closures
- Understand classes vs. closures



What Problem do Closures Solve?

How to create a "container" for data and functions?

Object-oriented programming: classes

```
1 public class Incrementor {
2     private int i;
3     public Incrementor(int i) {
4         this.i = i;
5     }
6     public int increment(int x) {
7         return x+i;
8     }
9 }
10 // use object
11 Incrementor inc = new Incrementor(2);
12 inc.increment(4); // returns 6
13 inc.increment(5); // returns 7
```

❓ Functional programming

- Closures

```
1 def incrementor(i:Int) : Int=>Int = {
2     def increment(x:Int) = x+i
3     return increment;
4 }
5 // use closure
6 val inc = incrementor(2)
7 inc(4) // returns 6
8 inc(5) // returns 7
```

- ❓ What are the challenges of making closures work?



Closures

- Runtime support for nested functions
 - particularly when lifetimes do not nest
 - Only applies to *static / lexical* scope



Nested Functions

- Nested functions allow for reuse of inner function name
- Allowed by GCC, but not C standard
- Allowed in Scala

```
1 int fact (int n) {  
2     int loop (int n, int result) {  
3         if (n <= 1) {  
4             return result;  
5         } else {  
6             return loop (n - 1, n * result);  
7         }  
8     }  
9     return loop (n, 1);  
10 }
```

```
1 $ gcc -c nested-fact.c  
2 $ gcc -pedantic -c nested-fact.c  
3 function.c: In function 'fact':  
4 function.c:2:3: warning: ISO C forbids nested functions [-pedantic]
```

```
1 def fact(n:Int) : Int =  
2     def loop(n:Int, result:Int) : Int =  
3         if n<=1 then result  
4         else loop(n-1, n*result)  
5     loop(n, 1)
```



Nested Functions

- Access variables from enclosing context: requires some runtime support

```
1 def fact(n:Int) : Int =  
2   def loop(i:Int, result:Int) : Int =  
3     if i>n then result  
4     else loop(i+1, i*result)  
5   loop(1, 1)
```



Nested functions: Scoping

- Access variable `x` from which context?
- Requires runtime support
- With explicit functions

```
1 val x = 4
2
3 def f(y: Int) = x*y
4
5 def g(h: Int=>Int) =
6     val x = 7
7     h(3) + x
8
9 g(f)
```

- With lambda expressions

```
1 val x = 4
2
3 def g(h: Int=>Int) =
4     val x = 7
5     h(3) + x
6
7 g(y => x*y)
```



Nested Functions: Scope vs Lifetime

- Limit *scope* of inner function
- *Lifetime* of inner function vs. *lifetime* of outer function?
- Potentially unsafe, and requires *more* runtime support than accessing variables from enclosing function
- Lifetime problems!
- Lexical Closures for C++



Nested Functions: GCC

- Lifetime problems caused by nested functions

```
1 typedef void (*funcptr) (int);
2
3 funcptr f (int x) {
4     void g (int y) {
5         printf ("x = %d, y = %d\n", x, y);
6     }
7     g (1);
8     return &g;
9 }
10
11 int main (void) {
12     funcptr h = f (10);
13     (*h) (2);
14     f (20);
15     (*h) (3);
16 }
```

Unsafe calls may or may not work

```
1 $ gcc -std=c99 nested-gcc.c
2 $ ./a.out
3 x = 10, y = 1 <- g(1): safe to call g, with x=10
4 x = 10, y = 2 <- (*h)(2): unsafe to call h, created with x=10
5 x = 20, y = 1 <- g(1): safe to call g
6 x = 20, y = 3 <- (*h)(3): unsafe to call h, created with x=10
```



Nested Function: Java and Scala

- Nested functions work correctly in Java and Scala

Scala

```
1 def f (x:Int) : Int=>Unit =
2   def g (y:Int) : Unit =
3     println ("x = %d, y = %d".format (x, y))
4   g (1)
5   g
6
7 def main () =
8   val h = f (10)
9   h (2)
10  val h2 = f (20)
11  h (3)
```

Java

```
1 import java.util.function.IntConsumer;
2
3 public static IntConsumer f (int x) {
4   IntConsumer g =
5     y -> System.out.format ("x = %d, y = %d\n", x, y);
6   g.accept (1);
7   return g;
8 }
9 public static void main (String[] args) {
10   IntConsumer h = f (10);
11   h.accept (2);
12   IntConsumer h2 = f (20);
13   h2.accept (3);
14 }
```

```
1 x = 10, y = 1 // g(1) // g.accept(1)
2 x = 10, y = 2 // h(2) // h.accept(2)
3 x = 20, y = 1 // g(1) // g.accept(1)
4 x = 10, y = 3 // h(3) // h.accept(3)
```



Nested Function: Java

- With explicit types

```
1 import java.util.function.Function;
2
3 static Function<Integer,Void> f (int x) {
4     Function<Integer,Void> g = y -> {
5         System.out.format ("x = %d, y = %d%n", x, y);
6         return null;
7     };
8     g.apply (1);
9     return g;
10 }
11
12 public static void main (String[] args) {
13     Function<Integer,Void> h = f (10);
14     h.apply (2);
15     f (20);
16     h.apply (3);
17 }
```



Nested Function: Java

With explicit object instantiation

```
1 import java.util.function.Function;
2
3 static Function<Integer,Void> f (int x) {
4     Function<Integer,Void> g = new Function<Integer,Void>() {
5         public Void apply(Integer y) {
6             System.out.format ("x = %d, y = %d%n", x, y);
7             return null;
8         }
9     };
10    g.apply (1);
11    return g;
12 }
13
14 public static void main (String[] args) {
15     Function<Integer,Void> h = f (10);
16     h.apply (2);
17     f (20);
18     h.apply (3);
19 }
```



Nested Function: Problem Summary

```
1 def outer (x:A) : B=>C =  
2   def inner (y:B) : C =  
3     //...use x and y...  
4   inner
```

1. Enclosing function `outer` is called
2. AR contains data `x`
3. Function `outer` returns nested function `inner`
4. Function `inner` references `x` from `outer`'s AR
5. Lifetime of `outer`'s AR and `x` ends
6. Nested function `inner` is called
7. Function `inner` needs `x` from `outer`'s AR



Nested Function: Closures

- *Closures* store inner function and environment
- Environment contains variables from enclosing scope
- Lifetime of environment = lifetime of inner function
- Environment is allocated on the heap
- Different implementations in different PLs
- Recurring implementation choice: copy or share?



Closures: Copy or Share

```
1 def outer (x:A) : B=>C =  
2   def inner (y:B) : C =  
3     ...use x and y...  
4   inner
```

- Closure contains
 - pointer/reference to code for `inner`
 - a copy of `x`



Closures: Copy or Share

```
1 def outer (x:A) : B=>C =  
2   var u:A = x  
3   def inner (y:B) : C =  
4     //...use u and y...  
5     u = u + 1  
6     inner
```

- Closure contains
 - pointer/reference to code for `inner`
 - copies of `x` and `u`
 - `inner` sees updated `u`?
 - require `u` to be immutable?



Closures: Copy or Share

```
1 def outer (x:A) : B=>C =  
2   var u:A = x  
3   def inner (y:B) : C =  
4     //...use u and y...  
5     u = u + 1  
6     inner
```

- Alternatively, share `u`
- Closure contains
 - pointer/reference to code for `inner`
 - copy of `x`
 - reference to shared `u` (on heap)



Closures: Scala

Scala function closure

```
1 object Demo:
2   def outer (x:Int) : Boolean=>Int =
3     def inner (y:Boolean) : Int =
4       x + (if y then 0 else 1)
5     inner
```

Java object-oriented implementation

```
1 public final class Demo {
2   public static Function1<Boolean, Integer> outer(int x) {
3     return new Closure(x);
4   }
5 }
6
7 public final class Closure extends AbstractFunction1<Boolean, Integer> {
8   private final int x;
9   public final Integer apply(Boolean y) {
10    return x + (y ? 0 : 1);
11  }
12  public Closure(int x) { this.x = x; }
13 }
```



Closures: Scala

Scala function closure

```
1 object Demo:
2   def outer (x:Int) : Boolean=>Int =
3     var u:Int = x
4     def inner (y:Boolean) : Int =
5       x + u + (if y then 0 else 1)
6     u = u+1;
7     inner
```

Java object-oriented implementation

```
1 import scala.runtime.*;
2
3 public final class Demo {
4   public static Function1<Boolean, Integer> outer(int x) {
5     IntRef u = new IntRef(x);
6     var c = new Closure(x, u);
7     u.elem = u.elem+1;
8     return c;
9   }
10 }
11
12 public final class Closure extends AbstractFunction1<Boolean, Integer> {
13   private final int x;
14   private final IntRef u;
15   public final Integer apply(Boolean y) {
16     return x + u.elem + (y ? 0 : 1);
17   }
18   public Closure(int x, IntRef u) {
19     this.x = x;
20     this.u = u;
21   }
22 }
```

- `u` is a `var` declaration, so is mutable: shared on heap



Closures: Example

- With nested method

```
1 val f : ()=>Int =  
2   var x = -1  
3   def g() = { x = x + 1; x }  
4   g
```

- With nested function

```
1 val f : ()=>Int =  
2   var x = -1  
3   () => { x = x + 1; x }
```

- Initializes `x` to `-1` when initializing variable `f`
- Returns incremented `x` on every call `f()`

```
1 scala> f()  
2 res0: Int = 0  
3 scala> f()  
4 res1: Int = 1
```



Closures: Example

- With methods

```
1 def g(y: Int) : ()=>Int =  
2   var z = y  
3   def h() = { z = z + 1; z }  
4   h
```

- `g(i)` returns a function `h: ()=>Int`,
its own `z` initialized to `i`

```
1 scala> val h1=g(10)  
2 h1: () => Int = $$Lambda$1098/39661414@54d8c20d  
3 scala> val h2=g(20)  
4 h2: () => Int = $$Lambda$1098/39661414@5bc7e78e
```

- With functions

```
1 val g: Int=>()=>Int =  
2   y => {  
3     var z = y  
4     () => { z = z + 1; z }  
5   }
```

- `h()` returns its own incremented `z`

```
1 scala> h1()  
2 res3: Int = 11  
3 scala> h1()  
4 res4: Int = 12  
5 scala> h2()  
6 res5: Int = 21  
7 scala> h2()  
8 res6: Int = 22  
9 scala> h1()  
10 res7: Int = 13
```



Closures: Example

➤ Implement a function `printer` that adds an initializable string prefix; wanted use:

```
1 val p = printer("> ")
2 println(p("hello"))
3 println(p("world!"))
```

- Return a function for printing that accesses the prefix argument

```
1 def printer(prefix: String) : String=>String =
2   (s: String) => prefix + s
```



Closures: Example

➤ Implement a map data structure from arrays and index access; wanted use:

```
1 val (get, put) = map(10)
2 put(3, "Hello")
3 put(6, "world!")
4 println(get(3))
```

- Return a tuple of functions that access a shared array

```
1 def map(size: Int) : (Int=>String, (Int, String)=>Unit) =
2   val elems : Array[String] = Array.ofDim(size)
3   def get(key: Int) =
4     if key < size then elems(key)
5     else throw new NoSuchElementException("Unknown key " + key)
6
7   def put(key: Int, value: String) =
8     elems(key) = value
9
10  (get, put)
```



Summary

- Closures combine functions with data from the context
- Align lifetime of functions and accessed context
- [Closures in Javascript](#)