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

```
public class Incrementor {
  private int i;
  public Incrementor(int i) {
    this.i = i;
  }
  public int increment(int x) {
    return x+i;
  }
}
// use object
Incrementor inc = new Incrementor(2);
inc.increment(4); // returns 6
inc.increment(5); // returns 7
```

Functional programming

- Closures

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

Top-Level Functions

- Function declarations made at top level
- Not hidden by scope

```
int loop (int n, int result) {
  if (n <= 1) {
    return result;
  } else {
    return loop (n - 1, n * result);
  }
}
int fact (int n) {
  return loop (n, 1);
}
```

Nested Functions: GCC

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

```c
int fact (int n) {
  int loop (int n, int result) {
    if (n <= 1) {
      return result;
    } else {
      return loop (n - 1, n * result);
    }
  }
  return loop (n, 1);
}
```

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

Nested Functions: GCC

- Access variables from enclosing context: requires some runtime support

```c
int fact (int n) {
  int loop (int i, int result) {
    if (i > n) {
      return result;
    } else {
      return loop (i+1, i * result);
    }
  }
  return loop (1, 1);
}
```

Nested functions: Scoping

- Access variable `x` from which context?
- requires some runtime support

```
typedef int (*funcptr) (int);
{
  int x = 4;
  {
    int f(int y) { return x*y; }
    {
      int g(funcptr h) {
        int x = 7;
        return h(3) + x;
      }
      g(f)
    }
  }
}
```

Nested Functions in Scala

With nested function

```scala
def mapDebug [A,B] (xs:List[A], f:A=>B) : List[B] =
  def printElt (x:A) : B =
    println (x)
    f (x)  // use f from enclosing context
  xs.map (printElt)
```

With lambda expression

```scala
def mapDebug [A,B] (xs:List[A], f:A=>B) : List[B] =
  xs.map ((x:A) => { println (x); f (x) })
```

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

```c
typedef void (*funcptr) (int);

funcptr f (int x) {
  void g (int y) {
    printf ("x = %d, y = %d\n", x, y);
  }
  g (1);
  return &g;
}

int main (void) {
  funcptr h = f (10);
  (*h) (2);
  f (20);
  (*h) (3);
}
```

Unsafe calls may or may not work

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

Nested Function: Clang

- Clang and LLVM
- Apple's *Blocks* extension to C = nested functions
- Ars Technica - Snow Leopard review (2009)
- Apple Developer Library: Introduction to Blocks
- Applications (iOS and OSX)
  - Graphical user interface callbacks
  - Collections processing
  - Concurrent tasks

Nested Function: Clang

```c
#include <Block.h>

// ^funcptr for blocks; *funcptr for function pointers
typedef void (^funcptr) (int);

funcptr f (int x) {
  funcptr g;
  g = ^(int y) {
    // use x from enclosing defn
    printf ("x = %d, y = %d\n", x, y);
  };
  g = Block_copy (g);
  g (1); // OK, f's activation record still allocated
  return g;
}

int main (void) {
  funcptr h = f (10);
  h (2); // OK, because of Block_copy
  f (20);
  h (3); // OK, because of Block_copy
  Block_release (h);
}
```

Blocks need additional runtime support

```
$ sudo apt-get install libblocksruntime-dev
```

```
$ clang -fblocks nested-clang.c -lBlocksRuntime
$ ./a.out
x = 10, y = 1 <- g(1)
x = 10, y = 2 <- h(2): safe to call h, created with x=10, GOOD!
x = 20, y = 1 <- g(1)
x = 10, y = 3 <- h(3): safe to call h, created with x=10, GOOD!
```

- Missing `Block_copy` and `Block_release` is like missing `malloc` and `free`

Nested Function: Java and Scala

- Nested functions work correctly in Java and Scala

Scala

```
def f (x:Int) : Int=>Unit =
  def g (y:Int) : Unit =
    println ("x = %d, y = %d".format (x, y))
  g (1)
  g

def main () =
  val h = f (10)
  h (2)
  f (20)
  h (3)
```

Java

```
import java.util.function.IntConsumer;

public static IntConsumer f (int x) {
  IntConsumer g =
    y -> System.out.format ("x = %d, y = %d%n", x, y);
  g.accept (1);
  return g;
}
public static void main (String[] args) {
  IntConsumer h = f (10);
  h.accept (2);
  f (20);
  h.accept (3);
}
```

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

Nested Function: Java

- With explicit types

```java
import java.util.function.Function;

static Function<Integer,Void> f (int x) {
  Function<Integer,Void> g = y -> {
    System.out.format ("x = %d, y = %d%n", x, y);
    return null;
  };
  g.apply (1);
  return g;
}

public static void main (String[] args) {
  Function<Integer,Void> h = f (10);
  h.apply (2);
  f (20);
  h.apply (3);
}
```

Nested Function: Java

With explicit object instantiation

```java
import java.util.function.Function;

static Function<Integer,Void> f (int x) {
  Function<Integer,Void> g = new Function<Integer,Void>() {
    public Void apply(Integer y) {
      System.out.format ("x = %d, y = %d%n", x, y);
      return null;
    }
  };
  g.apply (1);
  return g;
}

public static void main (String[] args) {
  Function<Integer,Void> h = f (10);
  h.apply (2);
  f (20);
  h.apply (3);
}
```

Nested Function: Problem Summary

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

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

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

Closures: Copy or Share

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

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

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

Java object-oriented implementation

```java
public final class Demo {
  public static Function1<Boolean, Integer> outer(int x) {
    return new Closure(x);
  }
}

public final class Closure extends AbstractFunction1<Boolean, Integer> {
  private final int x;
  public final Integer apply(Boolean y) {
    return x + (y ? 0 : 1);
  }
  public Closure(int x) { this.x = x; }
}
```

Closures: Scala

Scala function closure

```scala
object Demo:
  def outer (x:Int) : Boolean=>Int =
    var u:Int = x
    def inner (y:Boolean) : Int =
      x + u + (if y then 0 else 1)
    u = u+1;
    inner
```

Java object-oriented implementation

```java
import scala.runtime.*;

public final class Demo {
  public static Function1<Boolean, Integer> outer(int x) {
    IntRef u = new IntRef(x);
    var c = new Closure(x, u);
    u.elem = u.elem+1;
    return c;
  }
}

public final class Closure extends AbstractFunction1<Boolean, Integer> {
  private final int x;
  private final IntRef u;
  public final Integer apply(Boolean y) {
    return x + u.elem + (y ? 0 : 1);
  }
  public Closure(int x, IntRef u) {
    this.x = x;
    this.u = u;
  }
}
```

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

Closures: Example

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

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

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

Closures: Example

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

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

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

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

```
scala> h1()
res3: Int = 11
scala> h1()
res4: Int = 12
scala> h2()
res5: Int = 21
scala> h2()
res6: Int = 22
scala> h1()
res7: Int = 13
```

Summary

- Closures combine functions with data from the context
- Align lifetime of functions and accessed context
- Closures in Javascript