CSC 347 - Concepts of Programming Languages

Scope and Lifetime

Instructor: Stefan Mitsch

Learning Objectives

❓ How should identifiers relate to memory locations?

- Understand the difference between a memory location and an identifier pointing to it
- Understand the difference between the lifetime of a memory location and the lifetime of a pointer to it

Scope

- *Scope* of an identifier: region of text in which it may be used

```
1 def f (x: Int) =
2   val y = x+1
3   if x>y then
4     val z = y+1
5     println(s"z = $x")
```

- `x` and `y` are in scope after their declaration until end of method `f`
- `z` is in scope after its declaration until end of `if`-block

Occurrences of Identifiers

- *free* occurrence has no matching binding

```
1 y = 5*x;   // Free occurrences of x and y
```

- *binding* occurrence declares the identifier

```
1 val y : Int;    // binding occurrence of y
```

- *bound* occurrence follows matching declaration

```
1 var y : Int;    // Binding occurrence of y
2 var x : Int;    // Binding occurrence of x
3
4 x = 6;     // Bound occurrence of x
5 y = 5*x;  // Bound occurrences of x and y
```

Occurrences of Identifiers

- Complete programs usually have no free occurrences of identifiers
- How do IDEs treat free occurrences?

Scope of Identifiers

- Scope rules not limited to just variables
- Apply to identifiers for
  - variables
  - function arguments
  - function type parameters
  - function/method names
  - class names
  - and more

Circular Dependencies

❓ What to do with circular dependencies?

```
1 char f (int x) { return x>0 ? g (x-1) : 1; }
```

```
1 char g (int x) { return f (x) + f (x); }
```

- Most modern languages allow any order
- C, C++ require *forward declarations*

```
1 char f (int x);
2 char g (int x);
3 // f and g definitions can now be in any order
```

Shadowing

❓ Should reusing names be allowed?

```
 1 static void f () {
 2   int x = 1;
 3   {
 4     int y = x + 1;
 5     {
 6       int x = y + 1;
 7       System.out.println ("x = " + x);
 8     }
 9   }
10 }
```

- See Java Language Specification

```
1 $ javac C.java
2 C.java:7: error: variable x is already defined in method f()
3         int x = y + 1;
4             ^
5 1 error
```

Shadowing

- Fields in Java have different treatment

```java
public class C {
  static int x = 1;

  static void f () {
    int y = x + 1;
    {
      int x = y + 1;
      System.out.println ("x = " + x);
    }
  }

  public static void main (String[] args) {
    f ();
  }
}
```

```
$ javac C.java
$ java C
x = 3
```

Shadowing

- C is less strict than Java (on shadowing)

```c
int main () {
  int x = 1;
  {
    int y = x + 1;
    {
      int x = y + 1;
      printf ("x = %d\n", x);
    }
  }
}
```

```
$ gcc -o scope scope.c
$ ./scope
x = 3
```

Shadowing

- Scala is less strict than Java (on shadowing)

```
1  object C:
2    def f () =
3      var x = 1
4        var y = x + 1
5          var x = y + 1
6          println ("x = " + x)
7    end f
8
9    def main (args:Array[String]) =
10     f ()
11   end main
```

```
1 $ scalac C.scala
2 $ scala C
3 x = 3
```

Shadowing and Pattern Matching

```
1 def sum(x: List[Int]) = x match
2   case Nil       => 0
3   case x :: tail => x + sum(tail)
```

- What can be problematic about shadowing?
- Shadowing can make programs more difficult to read and maintain

```
1 def f(x : Int) = x match
2   case 1 => 0
3   case 2 => 0
4   case x => -x // renaming to "case y" not caught by compiler
```

- ❓ Why allow shadowing at all?

Shadowing and Nested Classes

💡 Shadowing is necessary to allow nested classes

```
1 class Outer
2   // Inner shadows Outer's this, hashCode, equals, toString
3   class Inner
```

Shadowing and Recursion

❓ Is `x` in scope?

```
1 int main (void) {
2   int x = 10;
3   {
4     int x = x + 1;
5     printf ("x = %08x\n", x);
6   }
7   return 0;
8 }
```

```
1 $ gcc -o scope scope.c
2
3 $ gcc -Wall -o scope scope.c
4 scope.c: In function ‘main’:
5 scope.c:5:7: warning: unused variable ‘x’ [-Wunused-variable]
6 scope.c:7:9: warning: ‘x’ is used uninitialized in this function [-Wuninitialized]
7
8 $ ./scope
9 x = 00000001
```

Shadowing and Recursion

- Java requires that all variables be initialized before use

```
1 class C {
2     public static void main (String[] args) {
3         int x = 1 + x;
4         System.out.printf ("x = %08x\n", x);
5     }
6 }
```

```
1 x.java:3: error: variable x might not have been initialized
2         int x = 1 + x;
3                     ^
```

Shadowing and Recursion

- Scala variables and fields are set to default values (e.g., `0`) before the initialization code is run
- Recursion is allowed when initializing fields

Scala

```
1 scala> val x:Int = 1 + x
2 x: Int = 1
```

Expressed in Java

```
1 public class C {
2   private final int x; // default-initialized to 0
3   public int x() { return x; }
4   public C() { x = 1 + x; }
5 }
```

Shadowing and Recursion

❓ Does that work with complex datatypes?

```
1 val xs:List[Int] = 1 :: xs
2 // java.lang.NullPointerException
```

- `xs` default-initialized to `null`
- `null != Nil`: exception occurs because `1 :: null` is `null.::(1)`

Shadowing and Recursion

```
1 case class S(head:Int, tail:S)
```

```
1 scala> val ss:S = S(1, ss)
2 ss: S = S(1,null)
```

- Need to delay evaluation of tail

```
1 case class T(head:Int, tail:()=>T)
```

```
1 scala> val ts:T = T(1, ()=>ts)
2 ts: T = T(1,$$Lambda$1324/2038353966@4d500865)
3 scala> ts.tail().tail().head
4 res14: Int = 1
```

Scala Streams

- Streams `#::` are non-strict in right-hand argument
- Deprecated, use `LazyList` instead

```
1 val ones:Stream[Int] = 1 #:: ones
2 // ones: Stream[Int] = Stream(1, ?)
```

```
1 scala> ones.take (5)
2 res0: scala.collection.immutable.Stream[Int] = Stream(1, ?)
3
4 scala> ones.take (5).toList
5 res1: List[Int] = List(1, 1, 1, 1, 1)
```

Scala Streams

- *Lazy* evaluation of stream elements

```
1 def f (x:Int) : Stream[Int] =
2   println (s"Called f($x)")
3   x #:: f(x+1)
4 // f: (x: Int)Stream[Int]
```

```
 1 scala> val xs:Stream[Int] = f(10)
 2 Called f(10)
 3 xs: Stream[Int] = Stream(10, <not computed>)
 4
 5 scala> xs.take(4).toList
 6 Called f(11)
 7 Called f(12)
 8 Called f(13)
 9 res12: List[Int] = List(10, 11, 12, 13)
10
11 scala> xs.take(4).toList
12 res13: List[Int] = List(10, 11, 12, 13)
13
14 scala> xs.take(6).toList
15 Called f(14)
16 Called f(15)
17 res14: List[Int] = List(10, 11, 12, 13, 14, 15)
```

Scala Lazy Lists

- *Lazy* evaluation of list elements

```
1 def f (x:Int) : LazyList[Int] =
2   println (s"Called f($x)")
3   x #:: f(x+1)
4 // f: (x: Int)LazyList[Int]
```

```
 1 scala> val xs:LazyList[Int] = f(10)
 2 xs: LazyList[Int] = <not computed>
 3
 4 scala> xs.take(4).toList
 5 Called f(10)
 6 Called f(11)
 7 Called f(12)
 8 Called f(13)
 9 res12: List[Int] = List(10, 11, 12, 13)
10
11 scala> xs.take(4).toList
12 res13: List[Int] = List(10, 11, 12, 13)
13
14 scala> xs.take(6).toList
15 Called f(14)
16 Called f(15)
17 res14: List[Int] = List(10, 11, 12, 13, 14, 15)
```

Static and Dynamic Scope

❓ What does this program do?

- Using Scala *syntax*, but various different *semantics*

```
1  var x:Int = 10
2  def f () =
3    x = 20
4
5  def g () =
6    var x:Int = 30
7    f ()
8
9  g ()
10 println (x)
```

Static Scope

- **Static scope**: identifiers are bound to the closest binding occurrence in an enclosing block of the program code
- **Static scoping property**: We can rename any identifier, so long as we rename it consistently throughout its scope (and so long as the new name we have chosen does not appear in the scope)
- Also known as **lexical scope**

Static and Dynamic Scope

- **Dynamic scope**: identifiers are bound to the binding occurrence in the *closest* activation record
- Consistent renaming may break a working program!

Static and Dynamic Scope

- Where could `z` come from?

```
1 ...
2 def g (x:Int) : Int =
3   var y:Int = x * 2
4   z * x * y           // x and y are local; z is non-local
```

- Dynamic scope:
  - non-locals are not resolved (bound) until runtime
  - to resolve non-local identifier, look at the callers

Static vs. Dynamic Scope: Scala

- Scala uses static scope (prints 20)
- Most languages do use static scope

```
1 var x:Int = 10
2 def f () : Unit =
3   x = 20
4
5 def g () : Unit =
6   var x:Int = 30
7   f ()
8
9 g ()
10 println (x)
```

Static vs. Dynamic Scope: Bash

- Bash (prints 10):

```
x=10
function f() {
    x=20
}
function g() {
    local x=30
    f
}
g
echo $x
```

Static vs. Dynamic Scope: C

- C functions (prints 20):

```c
int x = 10;
void f () {
    x = 20;
}
void g () {
    int x = 30;
    f ();
}
int main () {
    g ();
    printf ("x=%d\n", x);
}
```

Static vs. Dynamic Scope: Python

- Python (prints 20)

```
1 def main():
2   def f ():
3     nonlocal x
4     x = 20
5
6   def g ():
7     x = 30
8     f ()
9
10  x = 10
11  g ()
12  print (x)
13
14 main()
```

Static vs. Dynamic Scope: Python

- Python (prints 20)

```python
def f ():
  global x
  x = 20

def g ():
  x = 30
  f ()

x = 10
def main():
  g ()
  print (x)

main()
```

Static vs. Dynamic Scope: Python

- Python global scope is not static

```
1 def useX():
2   print (x)
3
4 def defX():
5   global x
6   x = 1
```

```
1 >>> useX()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 2, in useX
5 NameError: name 'x' is not defined
6 >>> defX()
7 >>> useX()
8 1
```

Static vs. Dynamic Scope

- Well-known PLs have included dynamic scoping…
  - Lisp, Perl, …
  - …and later added static scoping!

Static vs. Dynamic Scope

Emacs Lisp (prints 10)

```
1 (let ((x 10))
2   (defun f ()
3     (setq x 20))
4   (defun g ()
5     (let ((x 30))
6       (f)))
7   (g)
8   (message (int-to-string x)))
```

Common Lisp (prints 20)

```
1 (let ((x 10))
2   (defun f ()
3     (setq x 20))
4   (defun g ()
5     (let ((x 30))
6       (f)))
7   (g)
8   (print x))
```

Scheme (prints 20)

```
1 (let ((x 10))
2   (define (f)
3     (set! x 20))
4   (define (g)
5     (let ((x 30))
6       (f)))
7   (g)
8   (display x)
9   (newline))
```

Static vs. Dynamic Scope: Perl

- Perl (prints 10):

```
 1 local $x = 10;
 2 sub f {
 3     $x = 20;
 4 }
 5 sub g {
 6     local $x = 30;
 7     f ();
 8 }
 9 g ();
10 print ($x);
```

- `local` : dynamic scope

- Perl (prints 20):

```
 1 my $x = 10;
 2 sub f {
 3     $x = 20;
 4 }
 5 sub g {
 6     my $x = 30;
 7     f ();
 8 }
 9 g ();
10 print ($x);
```

- `my` : static scope

Lifetime

- *Lifetime* of an area of memory: duration during which it is allocated
- 📖 Chapter 7 of Mitchell textbook
- Recall activation records from Systems I

Activation Records

- *Activation records*: storage space for local variables and intermediate values that the runtime system generates
- Also known as *stack frames*
- ARs almost always placed on a call stack

Storage Options

Global

- Static storage
- Available for lifetime of program

Call Stack

- *In AR in call stack* (stack-allocated)
- Available whilst function active (called but not returned)

Heap

- *In heap* (heap-allocated)
- Available until deallocated (manually or via garbage collection)

Lifetime Issues

- 🐞 Lifetime too short
  - reads return other value
  - writes overwrite other value
  - resource state incorrect, e.g., file handle closed
  - can cause security problems
- 🐞 Lifetime too long
  - uses too much memory (*memory leak*)
  - too late in freeing other resources / finalization
  - can cause vulnerability to denial of service attacks

Call Stack of Activation Records

- *Call stack* of ARs allows
  - fast *allocation* of fresh AR on function call
  - fast *deallocation* of AR on function return
  - Contrast with heap allocation
- *Stack discipline* ensures ordering of AR
  - (call f) allocate AR for f
  - (call g) allocate AR for g
  - (return from g) deallocate AR for g
  - (return from f) deallocate AR for f

Call Stacks in Multi-Threaded Applications

❓ How should we maintain activation records in multi-threaded applications?

- Each thread needs a separate call stack
- Calls and returns in separate threads are independent

Heap Allocation

- Heap allocation can use any allocation pattern (not strict like stack discipline)
- For example, allocate *M* bytes → allocate *N* bytes → deallocate *M* bytes → deallocate *N* bytes
- Allocations may be long-lived, others short-lived
- Gives freedom, but more costly than call stack

Common Problems

- PLs with garbage collection
  - Java, Scala, C#, Python, Ruby, JS, Scheme, etc.
  - Lifetime too long (not GCed)
- PLs with manual memory management
  - C, C++
  - Pointers to storage whose lifetime has ended
  - *Dangling pointers* to an old AR
  - *Dangling pointers* to `free`d heap memory (*use after free*)
  - Double `free`ing of heap memory

Dangling Pointers: Stack

🐞 What is wrong with this program?

```c
#include <stdio.h>
#include <stdlib.h>

int *f (int x) {
  int y = x;
  return &y;
}

int main (void) {
  int *p = f (1);
  printf ("*p = %d\n", *p);
  return 0;
}
```

- Compile warning

```
$ gcc -o ar ar.c
ar.c: In function ‘f’:
ar.c:6:3: warning: function returns address of local variable
  [enabled by default]

$ ./ar
*p = 1
```

Dangling Pointers

- Static analysis tools can help

```
$ valgrind ./ar
==5505== Memcheck, a memory error detector
==5505== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==5505== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==5505== Command: ./ar
==5505==
==5505== Conditional jump or move depends on uninitialised value(s)
==5505==    at 0x4E7C1A1: vfprintf (vfprintf.c:1596)
==5505==    by 0x4E85298: printf (printf.c:35)
==5505==    by 0x400536: main (in /tmp/ar)
==5505==
==5505== Use of uninitialised value of size 8
==5505==    at 0x4E7A49B: _itoa_word (_itoa.c:195)
==5505==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
==5505==    by 0x4E85298: printf (printf.c:35)
==5505==    by 0x400536: main (in /tmp/ar)
==5505==
==5505== Conditional jump or move depends on uninitialised value(s)
==5505==    at 0x4E7A4A5: _itoa_word (_itoa.c:195)
==5505==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
==5505==    by 0x4E85298: printf (printf.c:35)
==5505==    by 0x400536: main (in /tmp/ar)
==5505==

```

```
*p = 1
==5505==
==5505== HEAP SUMMARY:
==5505==     in use at exit: 0 bytes in 0 blocks
==5505==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==5505==
==5505== All heap blocks were freed -- no leaks are possible
==5505==
==5505== For counts of detected and suppressed errors, rerun with: -v
==5505== Use --track-origins=yes to see where uninitialised values come from
==5505== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 2 from 2)
```

Dangling Pointers: Heap

🐞 What is wrong with this program?

```c
#include <stdio.h>
#include <stdlib.h>

int *f (int x) {
  int *result = (int *) malloc (sizeof (int));
  *result = x;
  return result;
}

int main (void) {
  int *p = f (1);
  printf ("*p = %d\n", *p);
  return 0;
}
```

- Program compiles

```
$ gcc -Wall -o ar ar.c && ./ar
*p = 1
```

- but...

Dangling Pointers: Heap

```
$ valgrind ./ar
==10962== Memcheck, a memory error detector
==10962== Copyright (C) 2002-2011, and GNU GPL, by Julian Seward et al.
==10962== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==10962== Command: ./ar
==10962==
*p = 1
==10962==
==10962== HEAP SUMMARY:
==10962==     in use at exit: 4 bytes in 1 blocks
==10962==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==10962==
==10962== LEAK SUMMARY:
==10962==    definitely lost: 4 bytes in 1 blocks
==10962==    indirectly lost: 0 bytes in 0 blocks
==10962==      possibly lost: 0 bytes in 0 blocks
==10962==    still reachable: 0 bytes in 0 blocks
==10962==         suppressed: 0 bytes in 0 blocks
==10962== Rerun with --leak-check=full to see details of leaked memory
==10962==
==10962== For counts of detected and suppressed errors, rerun with: -v
==10962== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Dangling Pointers: Heap

🐞 What is wrong with this program?

```c
#include <stdio.h>
#include <stdlib.h>

int *f (int x) {
  int *result = (int *) malloc (sizeof (int));
  *result = x;
  return result;
}

int main (void) {
  int *p = f (1);
  free (p);
  printf ("*p = %d\n", *p);
  return 0;
}
```

- Program compiles

```
$ gcc -Wall -o ar ar.c && ./ar
*p = 0
```

- but...

Dangling Pointers: Heap

```
$ valgrind ./ar
==13594== Memcheck, a memory error detector
==13594== Copyright (C) 2002-2011, and GNU GPLd, by Julian Seward et al.
==13594== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13594== Command: ./ar
==13594==
==13594== Invalid read of size 4
==13594==    at 0x4005D2: main (in /tmp/ar)
==13594==  Address 0x51f0040 is 0 bytes inside a block of size 4 freed
==13594==    at 0x4C2A82E: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==13594==    by 0x4005CD: main (in /tmp/ar)
==13594==
*p = 1
==13594==
==13594== HEAP SUMMARY:
==13594==     in use at exit: 0 bytes in 0 blocks
==13594==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==13594==
==13594== All heap blocks were freed -- no leaks are possible
==13594==
==13594== For counts of detected and suppressed errors, rerun with: -v
==13594== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

Summary

- Scope: how an identifier refers to a memory location
  - Static scope: closest lexical appearance in source code
  - Dynamic scope: closest activation record
- Lifetime: how long a memory location is available
  - Dangling pointers: point to freed memory