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

```
void f (int x) {
  int y = x + 1;
  if (x > y) {
    int z = y + 1;
    printf ("z = %d\n", z);
  }
}
```

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

Occurrences of Identifiers

- *free* occurrence has no matching binding

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

- *binding* occurrence declares the identifier

```
int y;    // binding occurrence of y
```

- *bound* occurrence follows matching declaration

```
int y;    // Binding occurrence of y
int x;    // Binding occurrence of x

x = 6;    // Bound occurrence of x
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?

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

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

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

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

Shadowing

❓ Should reusing names be allowed?

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

- See Java Language Specification

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

```scala
object C:
  def f () =
    var x = 1;
      var y = x + 1;
        var x = y + 1;
        println ("x = " + x)
  end f

  def main (args:Array[String]) =
    f ()
  end main
```

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

Shadowing

- Shadowing occurs in the Scala REPL

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

scala> def f (a:Int) = x + a
f: (a: Int)Int

scala> f (10)
res0: Int = 11

scala> val x = 2
x: Int = 2

scala> x
res1: Int = 2

scala> f (10)
res2: Int = 11
```

- Scala REPL behavior corresponds to

```scala
{
  val x = 1;
  def f (a:Int) = x + a
  f (10)
  {
    val x = 2;
    x
    f (10)
    ...
  }
}
```

Shadowing and Recursion

Is `x` in scope?

```c
int main (void) {
  int x = 10;
  {
    int x = x + 1;
    printf ("x = %08x\n", x);
  }
  return 0;
}
```

```
$ gcc -o scope scope.c

$ gcc -Wall -o scope scope.c
scope.c: In function ‘main’:
scope.c:5:7: warning: unused variable ‘x’ [-Wunused-variable]
scope.c:7:9: warning: ‘x’ is used uninitialized in this function [-Wuninitialized]

$ ./scope
x = 00000001
```

Shadowing and Recursion

- Java requires that all variables be initialized before use.

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

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

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

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

Shadowing and Recursion

Does that work with complex datatypes?

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

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

Shadowing and Recursion

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

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

- Need to delay evaluation of tail

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

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

Scala Streams

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

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

```
scala> ones.take (5)
res0: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> ones.take (5).toList
res1: List[Int] = List(1, 1, 1, 1, 1)</code></pre>
```

Scala Streams

- *Lazy* evaluation of stream elements

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

```
scala> val xs:Stream[Int] = f(10)
f(10)
xs: Stream[Int] = Stream(10, <not computed>)

scala> xs.take(4).toList
f(11)
f(12)
f(13)
res12: List[Int] = List(10, 11, 12, 13)

scala> xs.take(4).toList
res13: List[Int] = List(10, 11, 12, 13)

scala> xs.take(6).toList
f(14)
f(15)
res14: List[Int] = List(10, 11, 12, 13, 14, 15)
```

Scala Lazy Lists

- *Lazy* evaluation of list elements

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

```
scala> val xs:LazyList[Int] = f(10)
xs: LazyList[Int] = &lt;not computed>

scala> xs.take(4).toList
f(10)
f(11)
f(12)
f(13)
res12: List[Int] = List(10, 11, 12, 13)

scala> xs.take(4).toList
res13: List[Int] = List(10, 11, 12, 13)

scala> xs.take(6).toList
f(14)
f(15)
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*

```
var x:Int = 10
def f () =
  x = 20

def g () =
  var x:Int = 30
  f ()

g ()
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?

```
...
def g (x:Int) : Int =
  var y:Int = x * 2
  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

```scala
var x:Int = 10
def f () : Unit =
  x = 20

def g () : Unit =
  var x:Int = 30
  f ()

g ()
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: C macros

- C macros (prints 10):

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

- Macros expand in-place

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

Static vs. Dynamic Scope: Python

- Python (prints 20):

```
def main():
  def f ():
    nonlocal x
    x = 20

  def g ():
    x = 30
    f ()

  x = 10
  g ()
  print (x)

main()
```

Static vs. Dynamic Scope: Python

- Python (prints 20):

```
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

```python
def useX():
  print (x)

def defX():
  global x
  x = 1
```

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

```
(let ((x 10))
  (defun f ()
    (setq x 20))
  (defun g ()
    (let ((x 30))
      (f)))
  (g)
  (message (int-to-string x)))
```

Common Lisp (prints 20)

```
(let ((x 10))
  (defun f ()
    (setq x 20))
  (defun g ()
    (let ((x 30))
      (f)))
  (g)
  (print x))
```

Scheme (prints 20)

```
(let ((x 10))
  (define (f)
    (set! x 20))
  (define (g)
    (let ((x 30))
      (f)))
  (g)
  (display x)
  (newline))
```

Static vs. Dynamic Scope: Perl

- Perl (prints 10):

```perl
local $x = 10;
sub f {
    $x = 20;
}
sub g {
    local $x = 30;
    f ();
}
g ();
print ($x);
```

- `local` : dynamic scope

- Perl (prints 20):

```perl
my $x = 10;
sub f {
    $x = 20;
}
sub g {
    my $x = 30;
    f ();
}
g ();
print ($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

Control Links

❓ How should activation records be connected?

- Some systems, e.g., 32-bit x86, use *control links*
- *Control link* in each AR points to previous AR
- Control links provide linked list / stack view of ARs
- `ebp` register points to AR for most recently called function

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'd, 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 GPL'd, 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 free'd
==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