Programming Languages

Storage Management

Janyl Jumadinova

September 29, 2020
A **scope** is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted.
Scope Rules

A **scope** is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted.

In most languages with subroutines (functions, methods, procedures), we open a new scope on subroutine entry:

- create bindings for new local variables,
- deactivate bindings for global variables that are re-declared (these variables are said to have a “hole” in their scope)
- make references to variables
Ways Around “Hole in Scope”

Some languages allow access to scopes that are “hidden” by new declarations. E.g., Java:

```java
public class MyClass {
    private int x;
    // This creates a hole in the scope
    // of the instance variable x
    public void myMethod(int x) {
        x = 10; // Parameter x
        this.x = 20; // instance variable x
        ...
    }
}
```

C++ has a similar construct.
Scope Rules

On subroutine exit:

- destroy bindings for local variables
- reactivate bindings for global variables that were deactivated

The book uses the term “elaboration” for the process of allocating memory and creating bindings associated with a declaration.
Memory segments on the x86-32 architecture

Image credit: https://notes.shichao.io/tlpi/ch6/
Java Example

Jamboard example: Example.java bytecode

0: ic0nst_1: Push the integer constant 1 onto the operand stack

1: istore_1: Pop the top operand (an int value) and store it in local variable at index 1, which corresponds to variable num1

2: ic0nst_3: Push the integer constant 3 onto the operand stack

3: istore_2: Pop the top operand (an int value) and store it in local variable at index 2, which corresponds to variable num2

4: iload_1: Load the int value from local variable at index 1 and push it onto the operand stack

5: iload_2: Load the int value from local variable at index 2 and push it onto the operand stack

6: iadd: Pop the top two int values from the operand stack, add them, and push the result back onto the operand stack

7: istore_3: Pop the top operand int value and store it in local variable at index 3, which corresponds to variable num

8: istore_3: Return from the void main method
public class MyClass {
    private int a;
    public int getA() {
        return a;
    }
    public void setA(int x) {
        a = x;
    }
}

public void someOtherMethod() {
    MyClass x = new MyClass();
    MyClass y = new MyClass();
    ...
}

Whenever `someOtherMethod` is invoked, a new activation record (frame) is created for `someOtherMethod` and the declarations of `x` and `y` are elaborated into locations in this frame; the names `x` and `y` are bound to these locations.

(NOTE: creation of the frame itself is an elaboration of the declaration of function `someOtherMethod`)

... etc ...
pointer to y
pointer to x
return address

frame for `someOtherMethod`
public class MyClass {
    private int a;
    public int getA() {
        return a;
    }
    public void setA(int x) {
        a = x;
    }
}

public void someOtherMethod() {
    MyClass x = new MyClass();
    MyClass y = new MyClass();
    ...
}

Furthermore, the declarations of the instance variable `a` are elaborated into memory locations in the heap and the names `x.a` and `y.a` are bound to these locations.
Heap Allocation (Dynamic allocation)

Example (Java):

```java
int values[];
System.out.print("How big is the array? ");
int n = scan.nextInt();
values = new int[n];
```

No way to know at compile time how much space will be needed for the array "values" – determined at run time.

So... how can we know how much memory to save on the stack?
We can't!
We must allocate it dynamically from a special memory area called the heap.
Heap Allocation (Dynamic allocation)

Example (Java):

```java
int values[];
System.out.print("How big is the array? ");
int n = scan.nextInt();
values = new int[n];
```

- No way to know at compile time how much space will be needed for the array “values” – determined at run time.
- So... how can we know how much memory to save on the stack?
Heap Allocation (Dynamic allocation)

Example (Java):

```java
int values[];
System.out.print("How big is the array? ");
int n = scan.nextInt();
values = new int[n];
```

- No way to know at compile time how much space will be needed for the array “values” – determined at run time.
- So... how can we know how much memory to save on the stack? We can’t!
- We must allocate it dynamically from a special memory area called the **heap**.
Heap Allocation

Stack grows and shrinks ("push" and "pop"); easy to generate code for this at compile time.

Heap: no pattern—no "last-in, first-out" or similar rule:

```java
MyClass x = new MyClass();
MyClass y = new MyClass();
if (count == 10)
    x = new MyClass();
else
    y = new MyClass();
```

"Garbage"
Any time we use the “new” operator in Java, we allocate space from the heap.

In C, use of the malloc function allocates from the heap.

Harder to maintain than a stack; many techniques used.
Simplest way to maintain heap—“free list” of blocks of memory:
Heap Allocation

Re-assign z:

\[ z = x; \]
Heap Allocation

Create new object:

\[ x = \text{new} \ldots ; \]

Not garbage---something is still pointing to it
Create new object:

\[ y = \text{new} \ldots; \]
Heap Allocation

“Collect Garbage”

free:

1 -> 1 -> 2

x = new ...
y = new ...
z = new ...
z = x;
x = new ...
y = new ...
Heap Allocation

Merge adjacent blocks:

free: 4

z = x;

x = new ...

y = new ...
"First fit": take the first block in the free list that is big enough to satisfy the requested amount:

Request for block of size 1:

"Best fit": take the smallest block in the free list that is big enough to satisfy the request:

Request for block of size 1:
**Binding**

is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol. static vs. dynamic
**Binding**

is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol. static vs. dynamic

Example: \( count = count + 5; \)
Example: \( \text{count} = \text{count} + 5; \)
Some of the bindings and their binding times:

- The type of count is bound at compile time.
- The set of possible values of count is bound at compiler design time.
- The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.
- The internal representation of the literal 5 is bound at compiler design time.
- The value of count is bound at execution time with this statement.
function big() {
    function sub1() {
        var x = 7;
        sub2();
    }
    function sub2() {
        var y = x;
    }
    var x = 3;
    sub1();
}

Under static scoping, the reference to the variable \( x \) in \( \text{sub2} \) is to the \( x \) declared in the procedure \( \text{big} \).
Example: Static vs. Dynamic Scoping

```javascript
function big() {
    function sub1() {
        var x = 7;
    }
    function sub2() {
        var y = x;
        var z = 3;
    }
    var x = 3;
}
```

Under dynamic scoping, the meaning of x may reference the variable from either declaration of x, depending on the calling sequence.
How is scope implemented at execution time?

- Pointers on the activation record stack refer to surrounding scope.
- Lexical: “static link”.
- Dynamic: “dynamic link”.

Janyl Jumadinova
Programming Languages
Storage Manager
September 29, 2020
Activity: Static chains in JavaScript and Scoping

1. Go to http://goo.gl/hcrqmE for a working version of Figure 3.5 (in JavaScript).
2. Complete class activity (Google form on Slack).

Figure 3.5  **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B’s surrounding scope, A, by dereferencing its static chain twice and then applying an offset.