Lecture 13: Run-Time Storage Management

CSC 131
Kim Bruce

Attributes of Variable

- Scope  
- Lifetime  
- Location  
- Value

Lifetime

- FORTRAN - all allocated statically - \( \infty \)
- Stack-based (C/C++/Java/Pascal/...)
  - local vbles/ parameters: method/procedure/block entry to exit
  - allocate space in activation record on run-time stack
- Heap allocated variable
  - lifetime independent of scope
- Static - global vbles or static vbles

Value & Location

- Sometimes referred to as l-value & r-value
  - \( x = x + 1 \)  
  - What does each occurrence of \( x \) stand for?
  - location normally bound when declaration processed
- Normally values change dynamically
  - if frozen at compilation then called constants
  - Java final variables frozen when declaration processed.
  - Java static final bound at compile time.
**Aliases**

- x and y are aliases if both refer to same location.
- If x, y are aliases then changes to x affect value of y.
- Java has uniform model where assignment is by “sharing”, so create aliases.
- Languages that mix are more confusing.
  - Common mistakes occur when not realize aliases.
  - E.g., add elt to priority queue and then change it ...

**Pointers**

- “Pointers have been lumped with the goto statement as a marvelous way to create impossible to understand programs”
  - K & R, C Programming Language
- Problems
  - Dangling pointers -- leave pointer to recycled space
    - stack frame popped or recycled heap item
  - Dereference nil pointers or other illegal address
  - Unreachable garbage
  - in C: p+1 different from (int)p + 1

**Program Units**

- Separate segments of code allowing separate declarations of variables
  - Ex.: procedures, functions, methods, blocks
  - During execution represented by unit instance
    - fixed code segment
    - activation record with “fixed” ways of accessing items

**Activation Record Structure**

- Return address
- Access info on parameters (how?)
- Space for local vbles
  - How get access to non-local variables?
### Invoking Function

- Make parameters available to callee
  - E.g., put on stack or in registers
- Save state of caller (registers, prog. counter)
- Ensure callee knows where to return
- Enter callee at first instruction

### Returning from Function

- If function, leave result in accessible location
- Get return address and transfer execution
- Caller restores state

### Parameter Passing

- **Call-by-reference (FORTRAN, Pascal, C++)**
  - pass address (l-value) of parameter
- **Call-by-copying (Algol 60, Pascal, C, C++)**
  - pass (r)-value of parameter
  - options: in, out, in-out
- **Call-by-name (Algol 60)**
  - pass actual expression (as “thunk”) - not macro!
  - re-evaluate at each access
  - lazy gives efficient implementation if no side effects

### Call-by-name

```pascal
procedure swap(a, b : integer);
var temp : integer;
begin
  temp := a;
  a := b;
  b := temp
end;
```

- Won’t always work!
- swap(i, a[i]) with i = 1, a[1] = 3, a[3] = 17
- Can’t write swap that always works!
What about Java?

• Conceptually call-by-sharing
• Implemented as call-by-value of a reference

Stack-based Allocation

• Pascal, C, C++, Java, ...
  - Activation records on stack
  - Problem: static (scope) vs dynamic (return address)
  - Activation records pushed on call and popped on return
  - Activation record contains:
    • return address
    • return-result address — if necessary
    • control or dynamic link — to next stack frame
    • access or static link — to nearest stack frame of enclosing scope
    • parameters, local vbles, & intermediate results.

Static Memory allocation

• FORTRAN
  - All storage known at translation time
  - Activation records directly associated with code segments
  - At compile time, instructions and vbles accessed by (unit name, offset)
  - At link time, resolve to absolute addresses.
  - Procedure call and return straightforward

Accessing non-local vbles

```
program main;
type array_type = array [1..10] of real;
var a : integer;
b : array_type;
procedure x (var c : integer; d : array_type);
var e : array_type;
procedure y (f : array_type);
var g : integer;
begin
  z(a+c);
end; {y}
begin {x}
  : ..... := b[6]......
y(e);
end; {x}
```

```
procedure z (h : integer);
var a : array_type;
begin
  : 
  x (h,a);
  : 
end; {main}
begin {main}
  : 
x (a,b);
  : 
end. {main}
```
Assign Variables Offsets

<table>
<thead>
<tr>
<th>Name</th>
<th>Level</th>
<th>Name</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>f</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>g</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>h</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Look at run time stack when: main calls x, which calls y, which calls z, which calls x, which calls y, ...

Accessing non-local Variables

- Length of access chain from any fixed procedure to main is always same length
- Any non-local variable found after some fixed number of access links (independent of activation record)
- # of links = constant determinable at compile time
- Access via <chain position, offset> where 1st is # of access links to traverse, 2nd is offset in activation record.

Allocating Activation Record

- Static - sizes of all local variables and parameters known at compile time
  - Fixed size activation records
- Size known at unit activation
  - Array bounds depend on parameters
  - Space for parameter descriptors at fixed offset
- Dynamic
  - Flexible arrays and pointers
  - Allocate on heap w/reference on stack
Tail-Recursive Functions

- Recursion less efficient (space & time) than iteration because of activation records.

- A call to function f in body of g is tail if g returns immediately after call of f terminates.
  - Ex: fun g x = if x > 0 then f x else f (~x)

- Tail calls use stack space more efficiently

- Tail recursive functions even better!

Fibonacci

```c
int fib(int n) {
    int current = 1;
    int next = 1;
    while (n > 0) {
        int temp = current;
        current = next;
        next = next + temp;
        n = n - 1;
    }
    return current;
}
```

Replace while loops

Can replace while by tail recursive function where all variables used become parameters:

```c
can replace while by tail recursive function where all variables used become parameters:

fastfib n = fibloop n 1 1

where

fibloop 0 current next = current
fibloop n current next =
    fibloop (n-1) next (current + next);
```
Correctness

- Let $a_0, a_1, \ldots$ be list of Fibonacci numbers
- Lemma: For all $n, k \geq 0$,
  \[ \text{fibloop } n \ a_k \ a_{k+1} = a_{k+n} \]
- fastfib $n$ = fibloop $n$ $1$ $1$
  = fibloop $n$ $a_0$ $a_1$
  = $a_n$

Function Parameters

- Harder to cope with because need environment defined in. Two problems:
  - Downward funarg:
    \[
    \begin{align*}
    x &= 47 \\
    f \ y &= x + y; \\
    g(h) &= \text{let } \text{val } x = 17 \\
    &\quad \text{in } h(1) \\
    &> g(f)
    \end{align*}
    \]
  - When evaluate $f(i)$, is in environment where $x = 17$!