Review: Lexical Scope vs. Dynamic Scoping

Lexical Scoping

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

Dynamic Scoping

- Non-local variables are associated with declarations at *run* time
- Find the most recent, currently active run-time stack frame containing a declaration of the variable

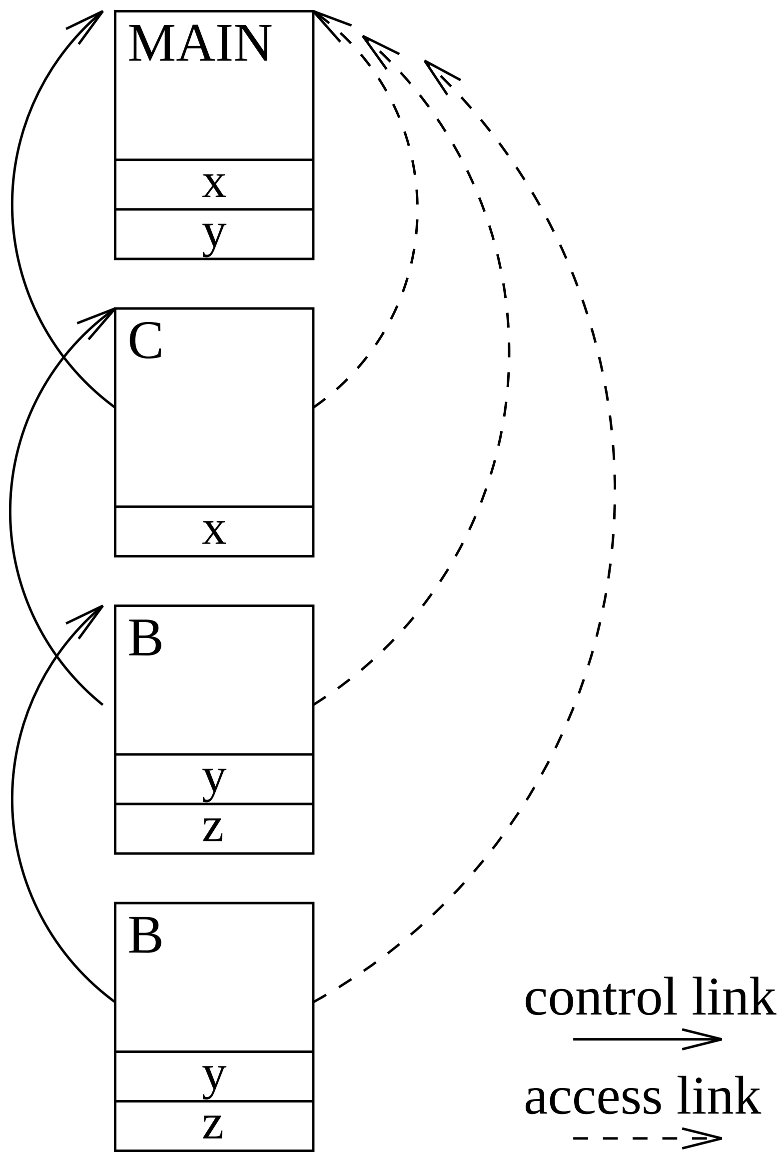
Lexical Scoping Example

scope of a declaration: Portion of program to which the declaration applies

```
Program
   x, y:  integer    // declarations of x and y
   begin
      Procedure B    // declaration of B
         y, z:  real    // declaration of y and z
         begin
            ...
            y = x + z    // occurrences of y, x, and z
            if (...)  call B         // occurrence of B
         end
      Procedure C    // declaration of C
         x:  real       // declaration of x
         begin
            ...
            call B    // occurrence of B
         end
      ...
      call C    // occurrence of C
      call B    // occurrence of B
   end
```

Lexical Scoping Example

Calling chain: MAIN $\Rightarrow$ C $\Rightarrow$ B $\Rightarrow$ B



Scoping and the Run-time Stack

Access links and **control links** may be used to look for non-local variable references.

Static Scope:

> Access link points to stack frame of the most recently activated lexically enclosing procedure

⇒ Non-local name binding is determined at *compile time*, and implemented at *run-time*

Dynamic Scope:

> Control link points to stack frame of caller

⇒ Non-local name binding is determined and implemented at *run-time*

Lexical scoping (de Bruijn notation)

Symbol table matches declarations and occurrences.

⇒ Each name can be represented as a pair

(nesting_level, local_index).

```
Program
   (1,1), (1,2):  integer    // declarations of x and y
   begin
      Procedure (1,3)         // declaration of B
         (2,1), (2,2):  real   // declaration of y and z
         begin
            ...    // occurrences of y, x, and z
            (2,1) = (1,1) + (2,2)
            if (...)  call (1,3)      // occurrence of B
         end
      Procedure (1,4)    // declaration of C
         (2,1):  real      // declaration of x
         begin
            ...
            call (1,3)    // occurrence of B
         end
      ...
      call (1,4)    // occurrence of C
      call (1,3)    // occurrence of B
   end
```

Access to non-local data

How does the code find non-local data at *run-time*?

Real globals

- visible *everywhere*
- translated into an address at compile time

Lexical scoping

- view variables as (*level,offset*) pairs (**compile-time symbol table**)
- look-up of (*level,offset*) pair uses chains of access links (**at run-time**)
- optimization to reduce access cost: display

Dynamic scoping

- variable names are preserved
- look-up of variable name uses chains of control links (**at run-time**)
- optimization to reduce access cost: reference table

Access to non-local data (lexical scoping)

Two important problems arise

1. *How do we map a name into a* (level,offset) *pair?*

   We use a block structured symbol table (**compile-time**)

   - when we look up a name, we want to get the most recent declaration for the name
   - the declaration may be found in the current procedure or in any nested procedure

2. *Given a* (level,offset) *pair, what's the address?*

   Two classic approaches (**run-time**)

   ⇒ access links (*static links*)

   ⇒ displays

Access to non-local data (lexical scoping)

To find the value specified by (l, o)

- need current procedure level, k
- if $k = l$, is a local value
- if $k > l$, must find l's activation record
  $\Rightarrow$ follow $k - l$ access links
- $k < l$ cannot occur

Maintaining access links:

If procedure p is nested immediately within procedure q, the access link for p points to the activation record of the most recent activation of q.

- calling level $k + 1$ procedure
  1. pass my FP as access link
  2. my backward chain will work for lower levels
- calling procedure at level $l \leq k$
  1. find my link to level $l - 1$ and pass it
  2. its access link will work for lower levels

The display

To improve run-time access costs, use a *display*.

- table of access links for lower levels
- lookup is index from known offset
- takes slight amount of time at call
- a single display or one per frame

Access with the display

assume a value described by (l, o)

- find slot as DP$[l]$ in display pointer array
- add offset to pointer from slot

"setting up the activation frame" now includes display manipulation.

Display management

Single global display: *simple method*

on entry to a procedure at level l

save the level l display value

push FP into level l display slot

on return

restore the level l display value

Run-time storage organization

To maintain procedure abstractions, the compiler must adopt some conventions to govern memory use.

Code space

- fixed size
- statically allocated

Data space

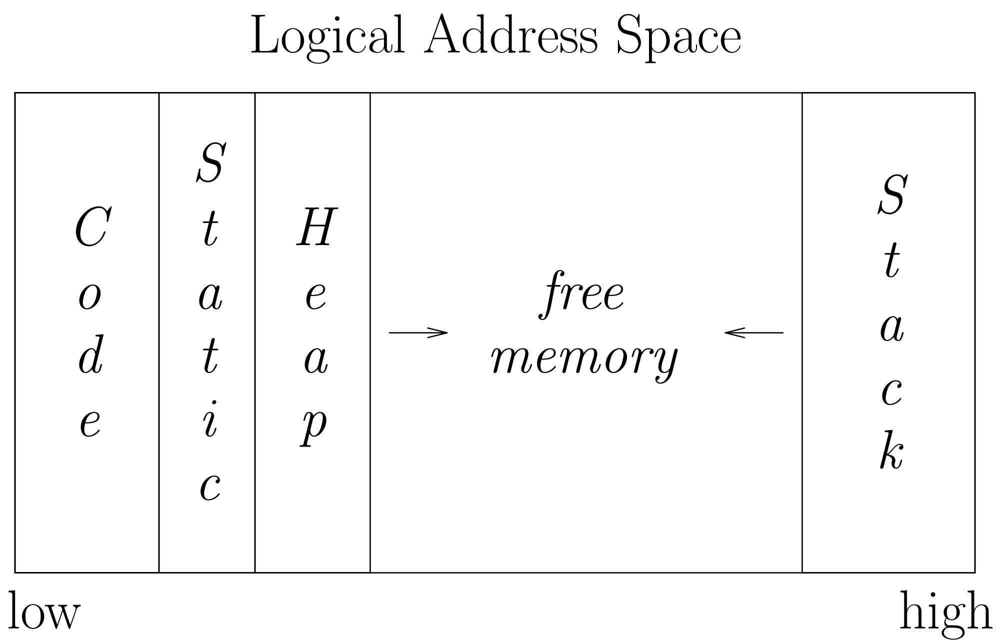
- fixed size data may be statically allocated
- variable size data must be dynamically allocated
- dynamic allocation on stack or heap depending on lifetime of data item (e.g.: variable number of arguments to procedure)

Runtime (Control) stack

- dynamic slice of activation tree
- usually supported in hardware

Run-time storage organization

Typical memory layout



The classical scheme

- allows both stack and heap maximal freedom
- code and static may be separate or intermingled

Run-time storage organization

Where do local variables go?

When can we allocate them on a stack?

Key issue is lifetime of local names

Downward exposure:

- called procedures may reference my variables
- dynamic scoping
- lexical scoping

Upward exposure:

- can I return a reference to my variables?
- functions that return functions

With only *downward exposure*, the compiler can allocate the frames on the run-time stack

Run-time storage organization

Each variable must be assigned a storage class
(*base address* for static area, stack, heap)

Static or global variables

- addresses compiled into code (*relocatable*)
- allocated at compile-time
- limited to fixed size objects

Procedure local variables

Put them on the stack —

- *if* sizes are fixed, or known at procedure invocation time, and
- *if* lifetimes are limited, i.e., values are not preserved

Run-time storage organization

Storage classes (*con't*):

Dynamically allocated variables

Put them on the heap —

- pointers may lead to non-local lifetimes
- (*usually*) an explicit allocation
- explicit or implicit deallocation (garbage collection)

Next Lecture

Things to do:

Start working on project as soon as possible. Will be posted by Friday evening.

Next time:

- aliases and dangling references
- garbage collection
- read Louden, Ch. 5 (5.5-5.7)