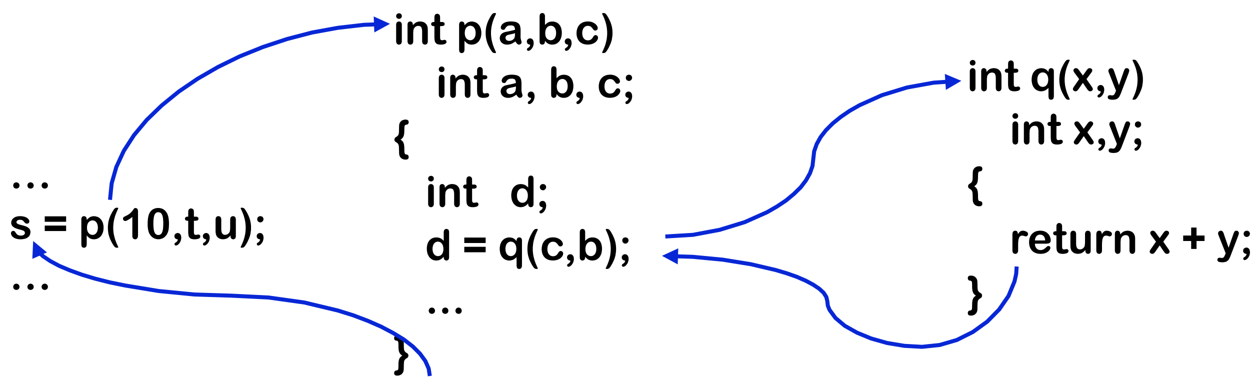
The Procedure Abstraction
Part II: Symbol Tables, Storage

The Procedure Abstractions: Last Lecture

- Control Abstraction
  - → Well defined entries & exits
  - → Mechanism to return control to caller



The Procedure Abstractions: Today

- Name Space
- External Interface

The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding variables
- Lets the programmer introduce "local" names

How can the compiler keep track of all those names?

```
procedure p {
    int a, b, c
        ....
    {
      int v, b, x, w
        ....
    }
}
```

The Procedure as a Name Space

The Problem

- At point *X*, which declaration of *b* is current?
- At run-time, where is *b* found?
- As parser goes in & out of scopes, how does it delete *b*?

The Answer

- The compiler must model the name space
- Lexically scoped symbol tables (see § 5.7.3)

```
procedure p {
    int a, b, c
        ....
    {
     int v, b, x, w
        ....
    }
}
```

Lexically-scoped Symbol Tables

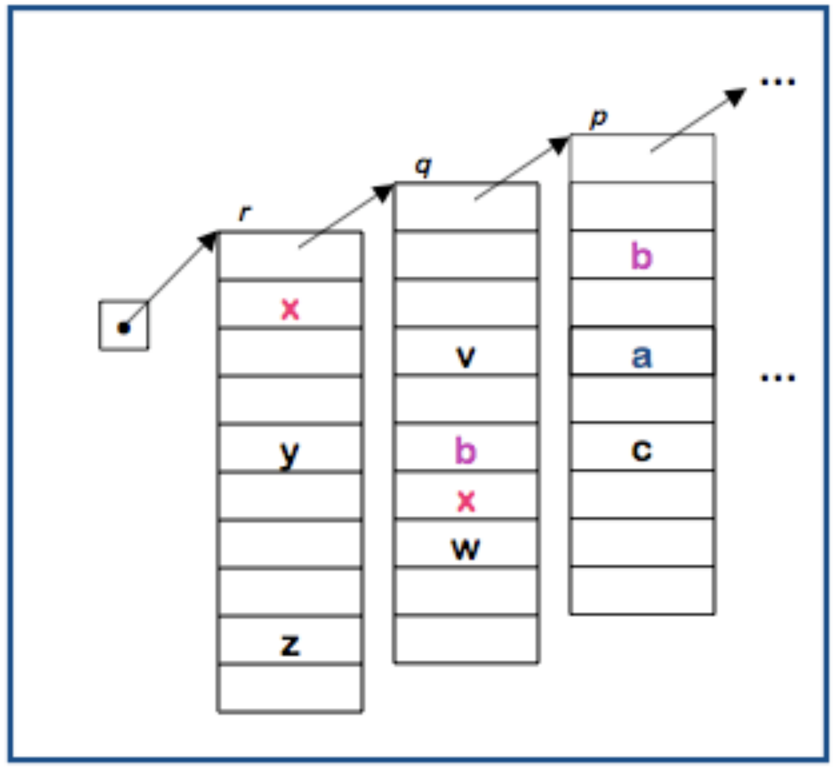
§ 5.7 in EaC

The problem
- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface
- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Example

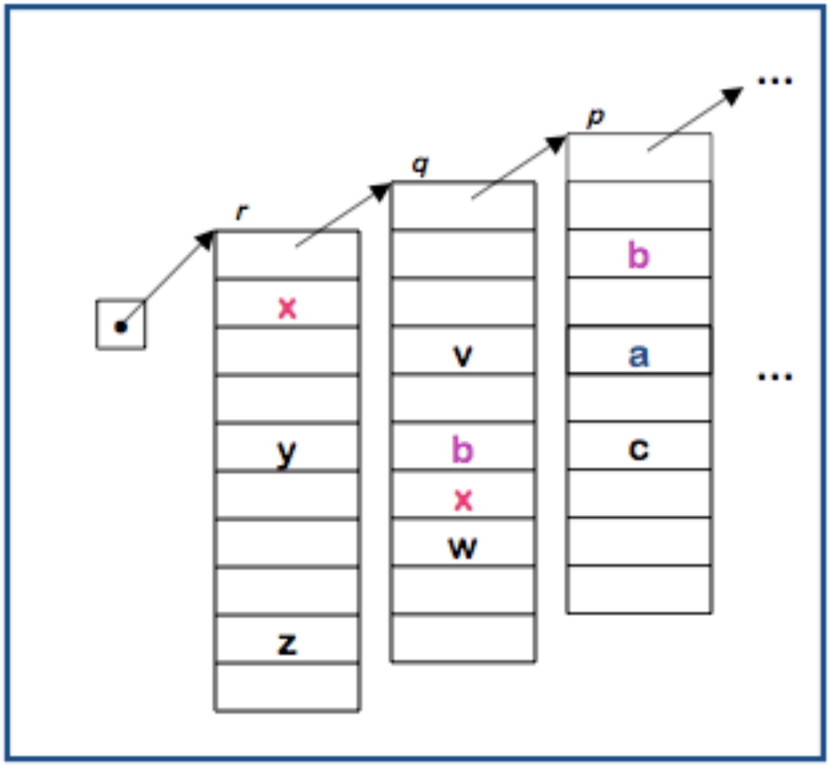
```
B0: procedure b {
        int a, b, c
B1:     {
            int v, b, x, w

B2:         {
                int x, y, z
                ....
            }

B3:         {
                int x, a, v
                ...
            }
            ...
        }
    ...
}
```



Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



"Sheaf of tables" implementation

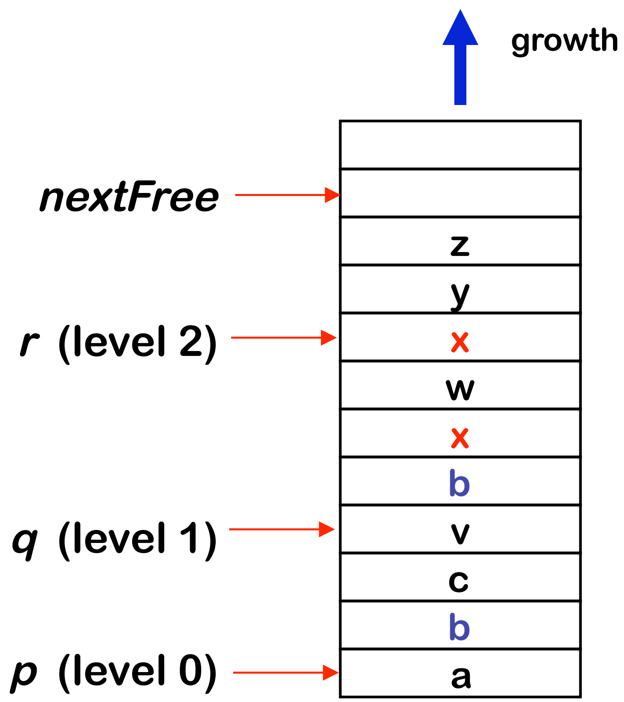
- *insert*() may need to create table
- it always inserts at current level
- *lookup*() walks chain of tables & returns first occurrence of name
- *delete*() throws away table for level *p*, if it is top table in the chain

If the compiler must preserve the table (*for, say, the debugger*), this idea is actually practical.

Individual tables can be hash tables.

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- insert () creates new level pointer if needed and inserts at nextFree
- lookup () searches linearly from nextFree-1 forward
- delete () sets nextFree to the equal the start location of the level deleted.

Advantage

- Uses <u>much</u> less space

Disadvantage

- Lookups can be expensive

The Procedure as an External Interface

OS needs a way to start the program's execution

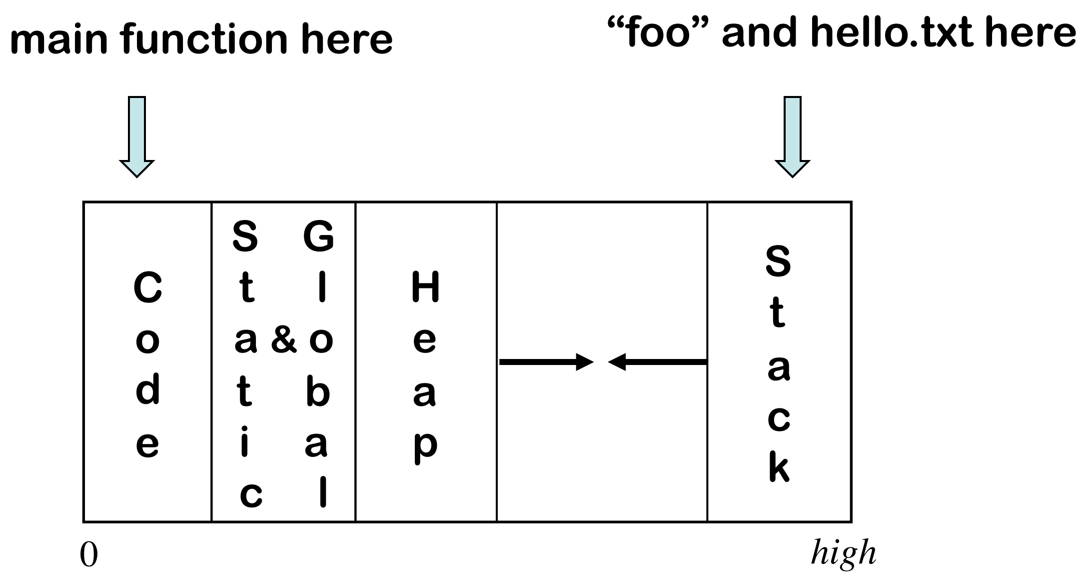
- Programmer needs a way to indicate where it begins
  - → The "main" procedure in most languaages
- When user invokes "grep" at a command line
  - → OS finds the executable
  - → OS creates a process and arranges for it to run "grep"
  - → "grep" is code from the compiler, linked with run-time system
    - ♦ Starts the run-time environment & calls "main"
    - ♦ After main, it shuts down run-time environment & returns
- When "grep" needs system services
  - → It makes a system call, such as fopen()

UNIX/Linux specific discussion

The Procedure as an External Interface

OS needs a way to start the program's execution

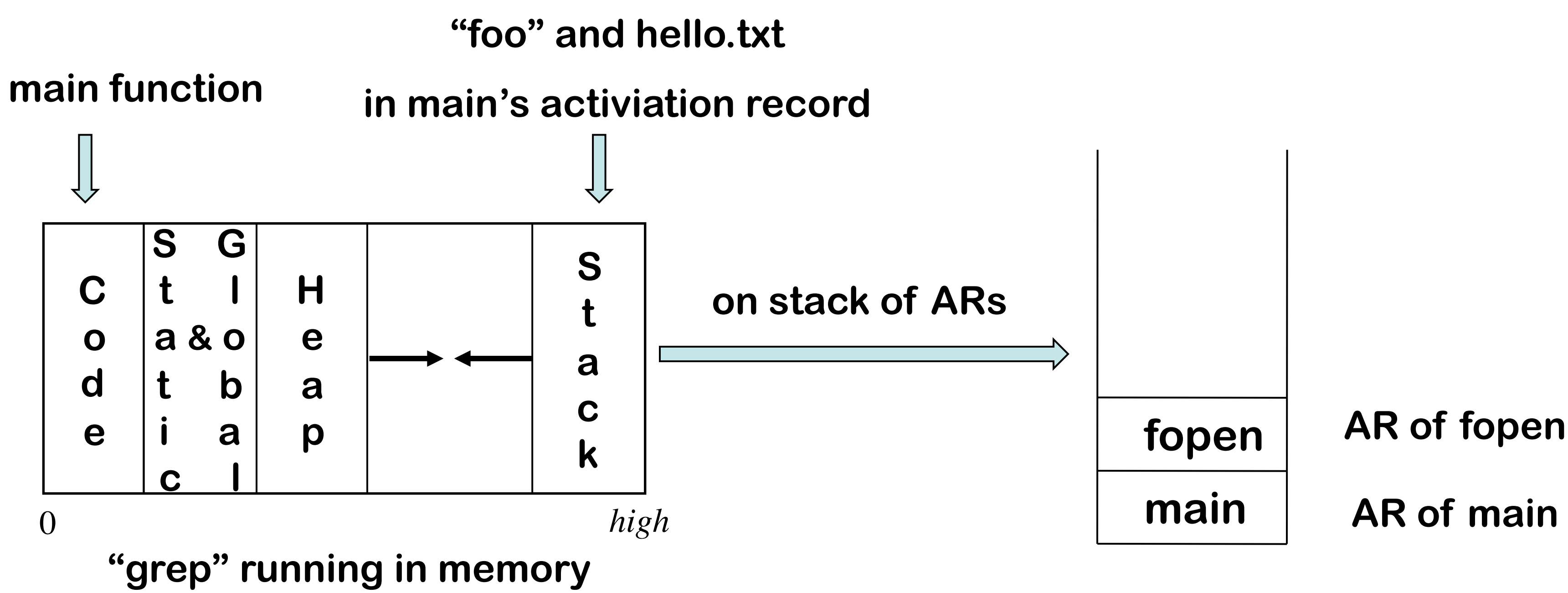
> grep "foo" hello.txt



"grep" running in memory

The Procedure as an External Interface

Grep may call fopen with “hello.txt”



Where Do All These Variables Go?

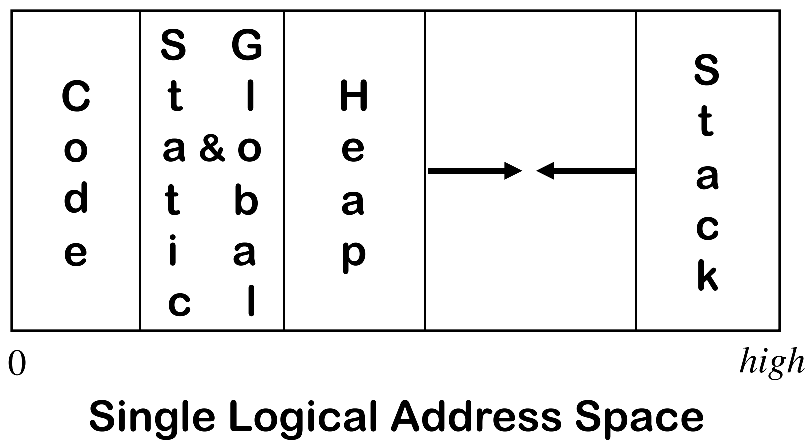
Local
- Keep them in the procedure activation record or in a register
- Automatic $\Rightarrow$ lifetime matches procedure's lifetime

Static
- File scope $\Rightarrow$ storage area affixed with file name
- Lifetime is entire execution

Global
- One or more named global data areas
- One per variable, or per file, or per program, …
- Lifetime is entire execution

Placing Run-time Data Structures

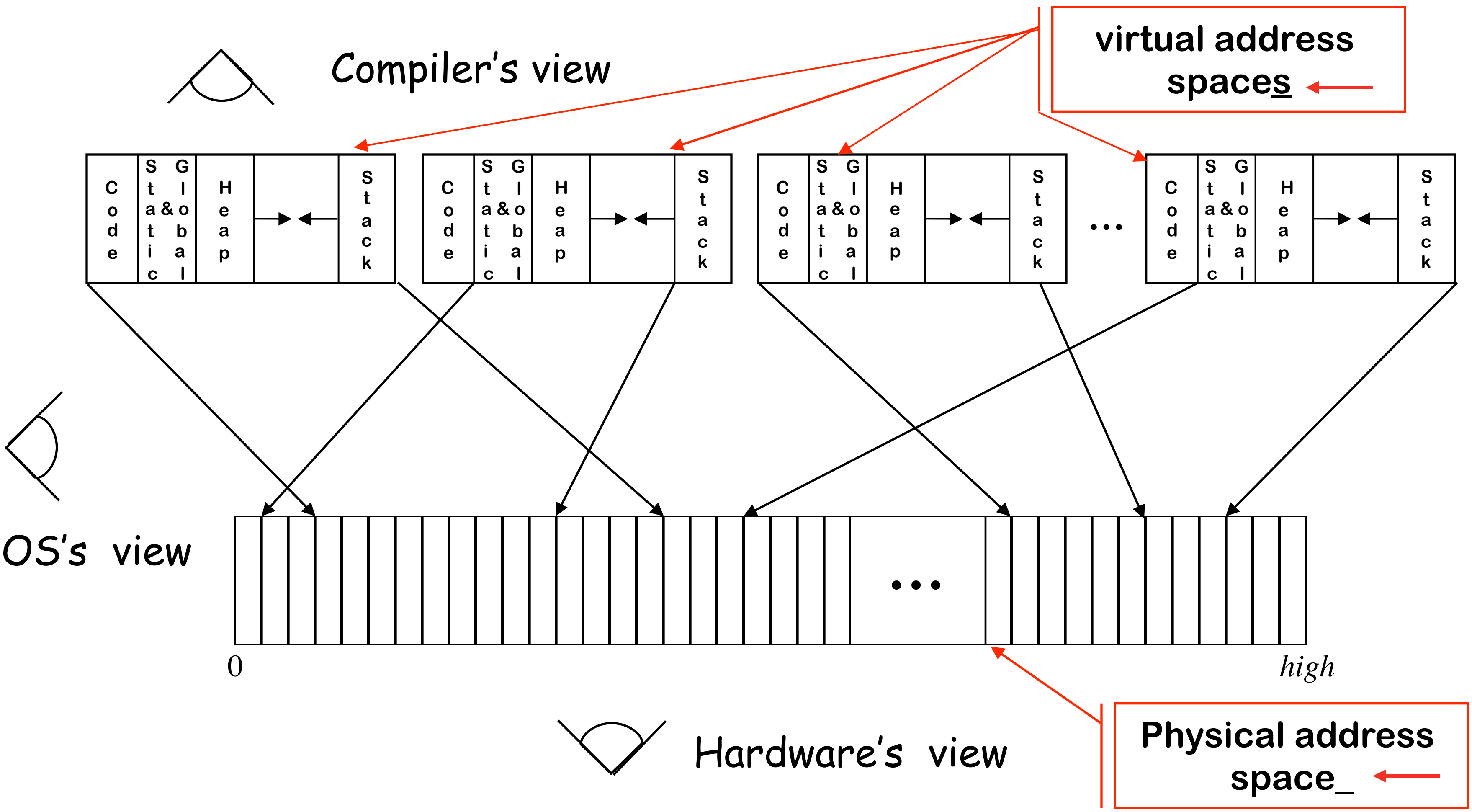
Classic Organization



- Code, static, & global data have known size
- Heap & stack both grow & shrink over time
- This is a <u>virtual</u> address space

How Does This Really Work?

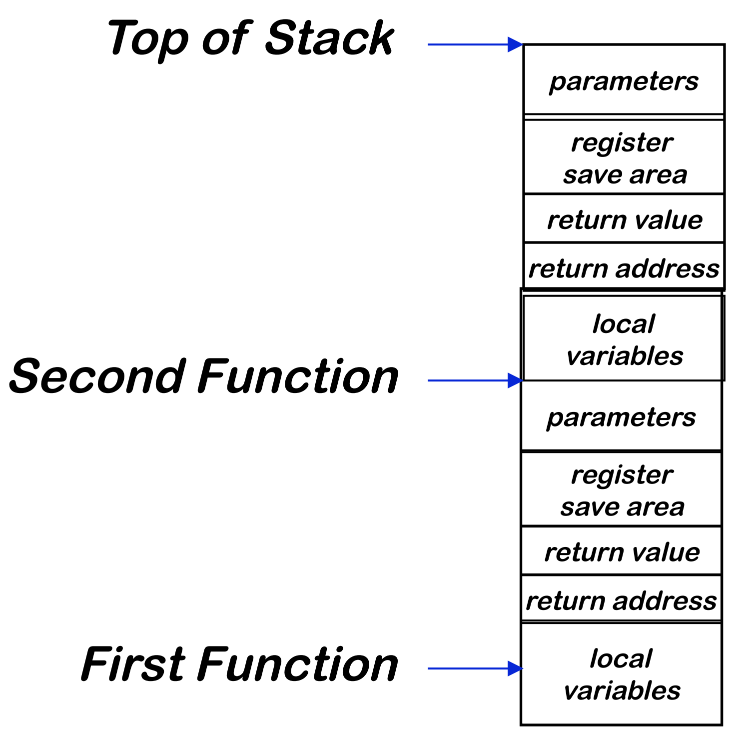
The Big Picture



Where Do Local Variables Live?

A Simplistic model

- Allocate a data area for each distinct scope
- Need a data area per invocation (or activation) of a scope
- We call this the scope's activation record
- The compiler can also store control information there !

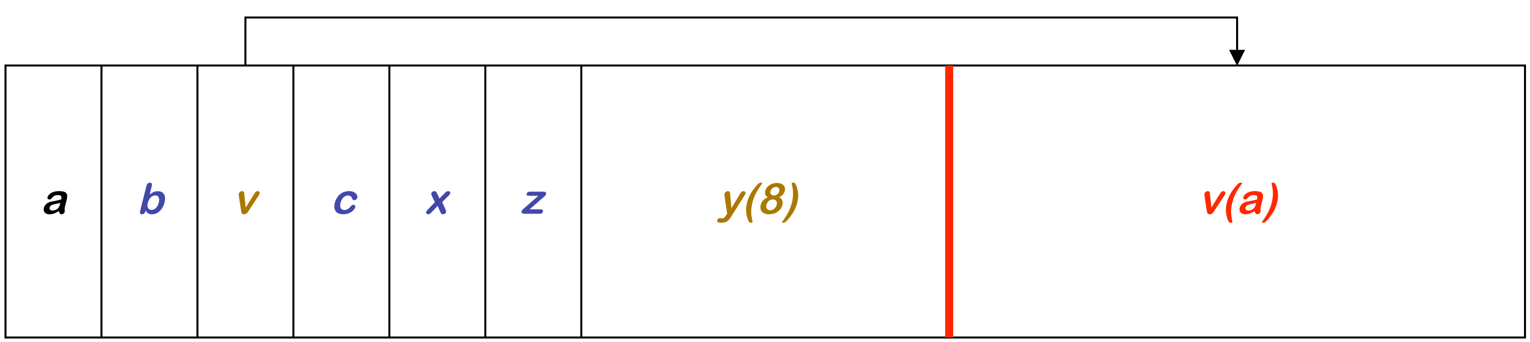


Variable-length Data

```
B0: {
        int a, b
        int v(a), c, x
        int z, y(8)
              ....
         }
```

Arrays

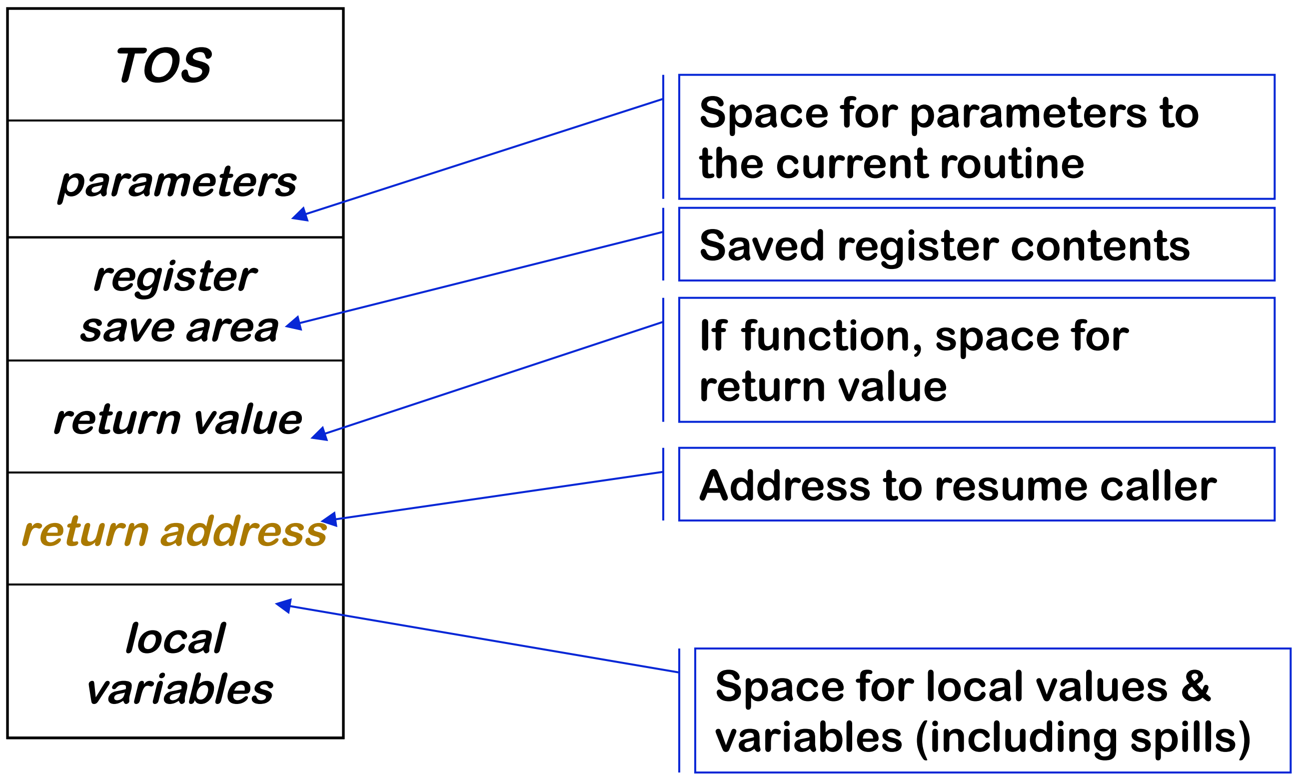
- → If size is fixed at compile time, store in fixed-length data area
- → If size is variable, store descriptor in fixed length area, with pointer to variable length area
- → Variable-length data area is assigned at the end of the fixed length area for block in which it is allocated



Includes variable length data for all blocks in the procedure …

Variable-length data

Activation Record Basics



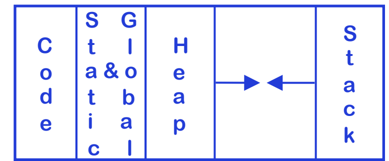
One AR for each invocation of a procedure

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, *AND*
- If code normally executes a "return"

⇒ Keep ARs on a stack



Yes! This stack.

- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

⇒ ARs must be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

Communicating Between Procedures

Most languages provide a parameter passing mechanism
⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- Call-by-reference passes a pointer to actual parameter
  - → Requires slot in the AR (for address of parameter)
  - → Multiple names with the same address?

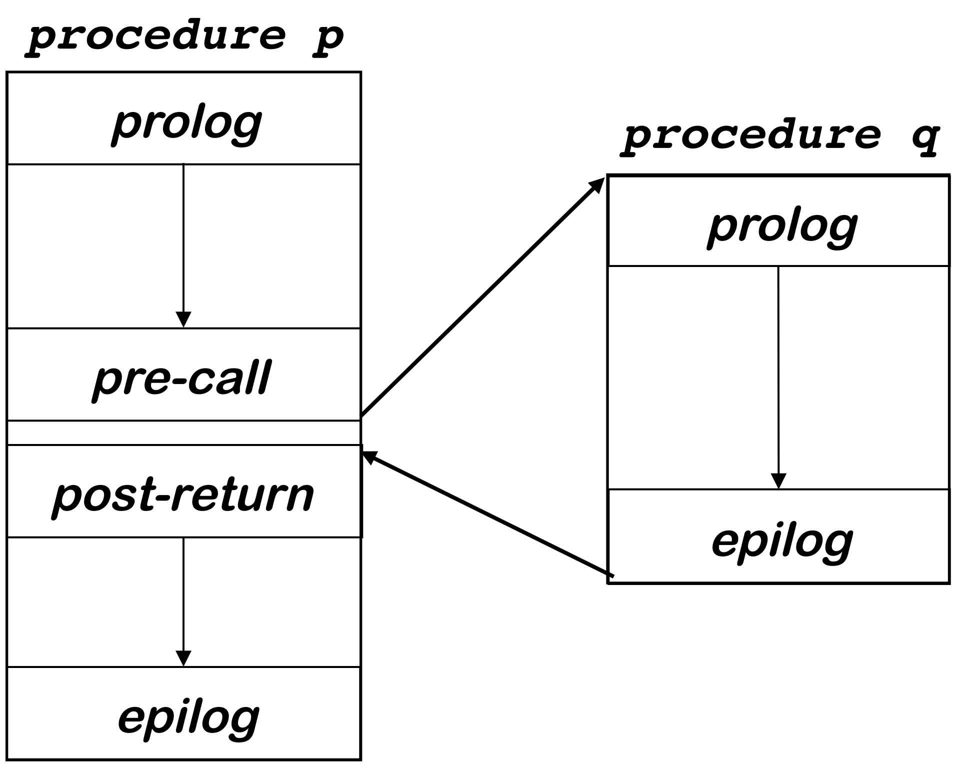
call fee(x,x,x);

- Call-by-value passes a copy of its value at time of call
  - → Requires slot in the AR
  - → Each name gets a unique location *(may have same value)*
  - → Arrays are mostly passed by reference, not value

- Can always use global variables …

Procedure Linkages

Standard procedure linkage



Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters

Procedure Linkages

Pre-call Sequence

- Sets up callee's basic AR
- Helps preserve its own environment

The Details

- Allocate space for the callee's AR
  - → except space for local variables
- Evaluates each parameter & stores value or address
- Saves return address, caller's ARP into callee's AR
- If access links are used
  - → Find appropriate lexical ancestor & copy into callee's **AR**
- Save any caller-save registers
  - → Save into space in caller's AR
- Jump to address of callee's prolog code

Procedure Linkages

Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
  - → Also copy back call-by-value/result parameters
- Continue execution after the call

Procedure Linkages

Prolog Code
- Finish setting up the callee's environment
- Preserve parts of the caller's environment that will be disturbed

The Details
- Preserve any callee-save registers
- If display is being used
  - → Save display entry for current lexical level
  - → Store current ARP into display for current lexical level
- Allocate space for local data
  - → Easiest scenario is to extend the AR
- Find any static data areas referenced in the callee
- Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables

Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value? No, this happens on the return statement
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)

Extra Slides Start Here

Establishing Addressability

Must create base addresses

- Global & static variables
  - → Construct a label by mangling names (*i.e.*, &_fee)
- Local variables
  - → Convert to static data coordinate and use **ARP** + offset
- Local variables of other procedures
  - → Convert to static coordinates
  - → Find appropriate **ARP**
  - → Use that **ARP** + offset

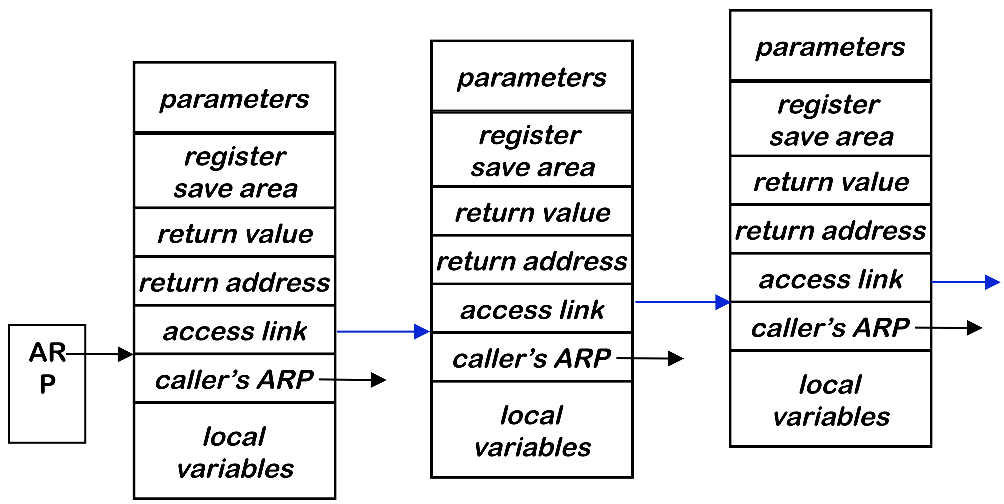
{ **Must find the right AR**

Need links to nameable ARs

Establishing Addressability

Using access links

- Each AR has a pointer to AR of lexical ancestor
- Lexical ancestor need not be the caller



Some setup cost on each call

- Reference to ‹*p*,16› runs up access link chain to *p*
- Cost of access is proportional to lexical distance

Establishing Addressability

Using access links

SC	Generated Code
<2,8>	loadAI r_0, 8 ⇒ r_2
<1,12>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0,16>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, -4 ⇒ r_1 loadAI r_1, 16 ⇒ r_2

Assume
- Current lexical level is 2
- Access link is at **ARP** - 4

Maintaining access link
- Calling level *k*+1
→ Use current **ARP** as link
- Calling level *j* < *k*
→ Find ARP for *j* –1
→ Use that ARP as link

Access & maintenance cost varies with level

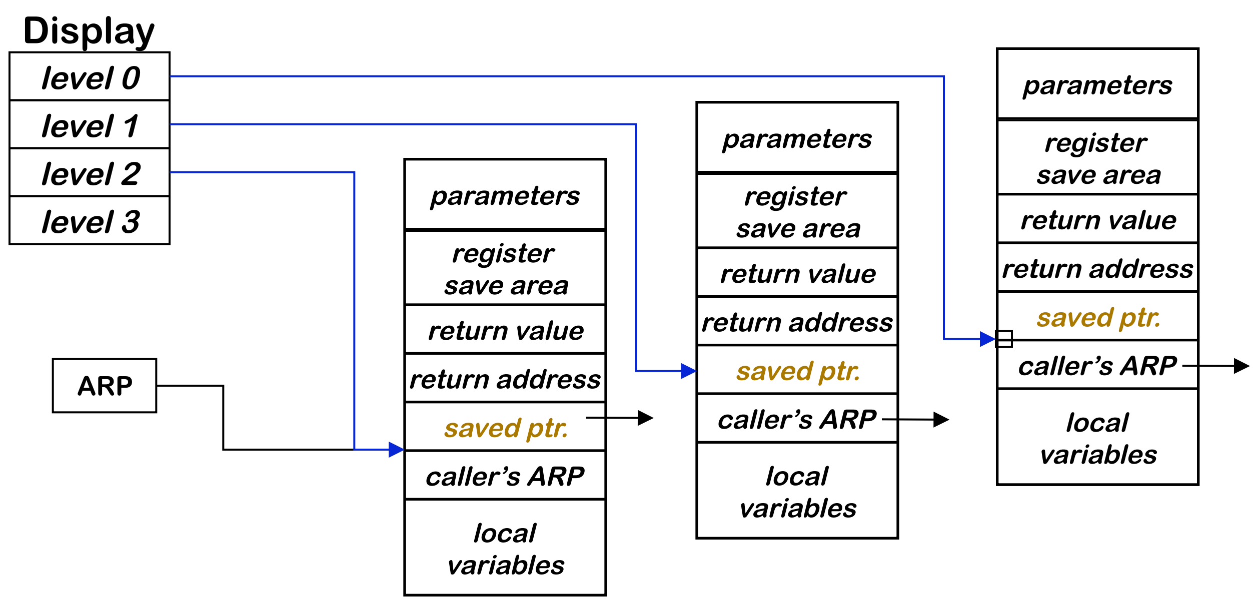
All accesses are relative to ARP (r_0)

Establishing Addressability

Using a display

- Global array of pointer to nameable ARs
- Needed ARP is an array access away

Some setup cost on each call



- Reference to ‹*p*,16› looks up *p*'s ARP in display & adds 16
- Cost of access is constant (ARP + offset)

Establishing Addressability

Using a display

SC	Generated Code
<2,8>	loadAI r_0, 8 ⇒ r_2
<1,12>	loadI _disp ⇒ r_1 loadAI r_1, 4 ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0,16>	loadI _disp ⇒ r_1 loadAI r_1, 16 ⇒ r_2

Assume
- Current lexical level is 2
- Display is at label _disp

Maintaining access link
- On entry to level *j*
  → Save level *j* entry into AR (Saved Ptr field)
  → Store ARP in level *j* slot
- On exit from level *j*
  → Restore level *j* entry

Access & maintenance costs are fixed
Address of display may consume a register

Desired AR is at _disp + 4 x *level*

Establishing Addressability

Access links versus Display

- Each adds some overhead to each call
- Access links costs vary with level of reference
  - → Overhead only incurred on references & calls
  - → If ARs outlive the procedure, access links still work
- Display costs are fixed for all references
  - → References & calls must load display address
  - → Typically, this requires a register *(rematerialization)*

Your mileage will vary

- Depends on ratio of non-local accesses to calls
- Extra register can make a difference in overall speed

For either scheme to work, the compiler must insert code into each procedure call & return

Procedure Linkages

How do procedure calls actually work?

- At compile time, callee may not be available for inspection
  - → Different calls may be in different compilation units
  - → Compiler may not know system code from user code
  - → All calls must use the same protocol

Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement *(for interoperability)*