RICE

COMP 412
FALL 2010

The Procedure Abstraction, Part III
Storage Layout & Addressability

Comp 412

Example

Reminder from last lecture

```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure  r {
            int x, y, z
            ….
        }
        procedure s {
            int x, a, v
            …
        }
        … r … s
    }
  … q …
}
```

```
B0: {
          int a, b, c
B1:        {
              int v, b, x, w
B2:           {
                  int x, y, z
                  ….
              }
B3:           {
                  int x, a, v
                  …
              }
              …
          }
     …
}
```

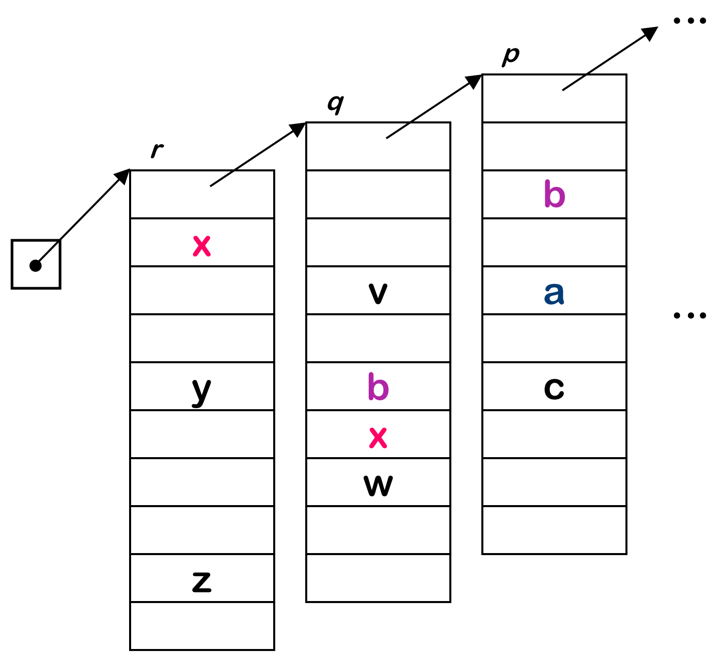
This example is included to remind you of the many different scopes in which names can live. It does not show global or static names.

Lexically-scoped Symbol Tables

Reminder from last lecture

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 level *p* table 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 are hash tables.

This high-level idea can be implemented as shown, or it can be implemented in more space-efficient (albeit complex) ways.

Where Do Local Variables Live?

A Simplistic Model (*the obvious model*)

- Allocate a data area for each distinct scope
- One data area per "sheaf" in scoped table

What about recursion?

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

More complex scheme

- One activation record (AR) per procedure instance
- All the procedure's scopes share a single AR *(may share space)*
- Static relationship between scopes in single procedure

Used this way, "static" means knowable at *compile time* (and, therefore, fixed).

Where Do All These Variables Go?

Automatic & Local

- Automatic ⇒ lifetime matches procedure's lifetime
- Keep them in the procedure's activation record or in a register

Static

- Procedure scope ⇒ name a storage area for the procedure
  - &_p.x for variable x in procedure p
- File scope (C) ⇒ name a storage area for 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

Lifetime does not match procedure's lifetime ⇒ allocate it on the heap

Data Areas

If variables go into data areas, where do data areas go

If lifetime of data area matches procedure invocation *AND* if procedures normally return then

⇒ Stack them with control information (return addresses)

If lifetime of data area is entire execution then

⇒ Allocate space for data area statically (assembler directive)

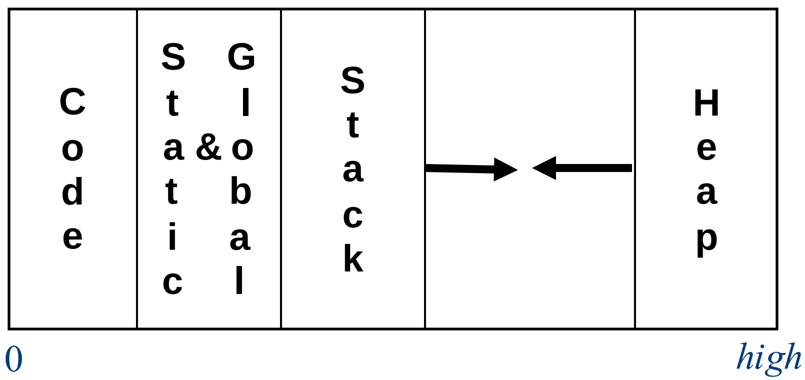
If lifetime of data is less than entire execution *BUT* longer than procedure invocation then

⇒ Allocate space for data area on the heap

Where do the stack, the heap, and static data areas go?

Placing Run-time Data Structures

Classic Organization



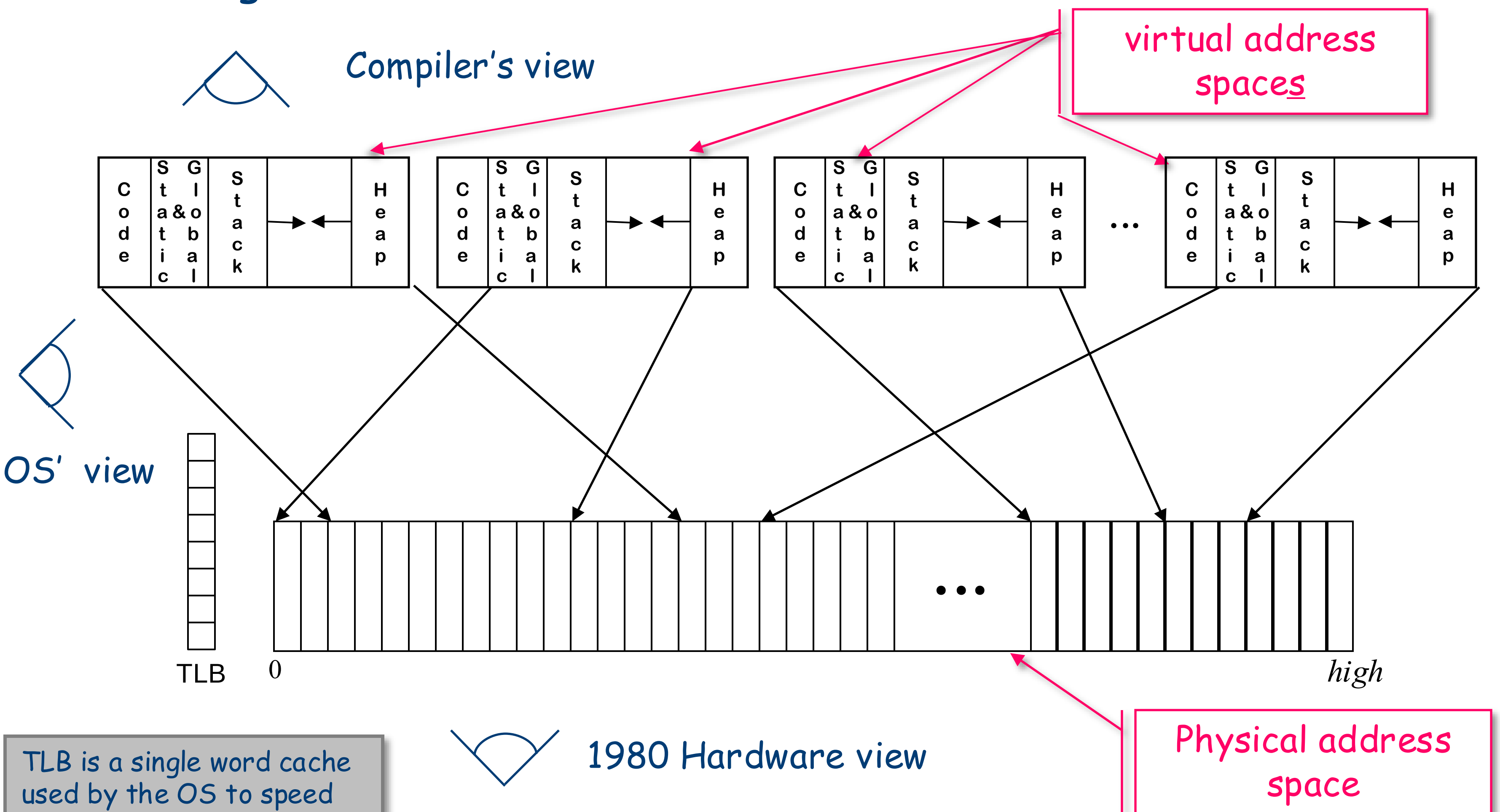
Single Logical Address Space

- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved
- Uses address space, not allocated memory

- Code, static, & global data have known size
  - Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a <u>*virtual*</u> address space

How Does This Really Work?

The Big Picture

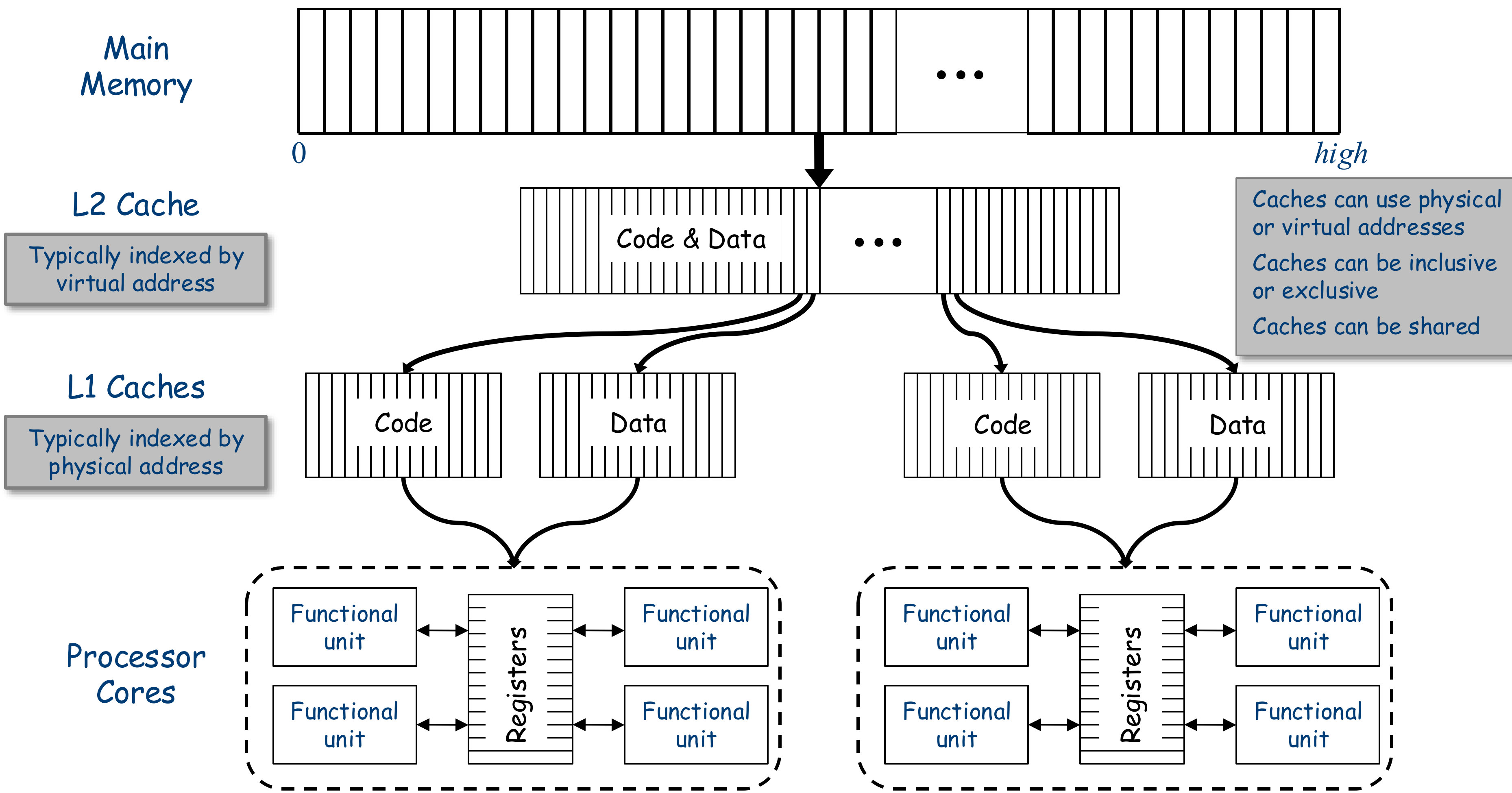


TLB is a single word cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of TLB.

How Does This Really Work?

Most systems now include L3 caches. L4 is on its way.

Of course, the "Hardware view" is no longer that simple



Cache structure matters for performance, not correctness

Translating Local Names

How does the compiler represent a specific instance of *x*?

- Name is translated into a *static coordinate*
  - ‹ *level*,*offset* › pair
  - "*level*" is lexical nesting level of the procedure
  - "*offset*" is *unique* within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "*level*" is a function of the table in which *x* is found
  - Stored in the entry for each *x*
- "*offset*" must be assigned and stored in the symbol table
  - Assigned at *compile time*
  - Known at *compile time*
  - Used to generate code that *executes* at *run-time*

Storage for Blocks within a Single Procedure

```
B0:  {
         int a, b, c
B1:      {
            int v, b, x, w
B2:         {
               int x, y, z
               …
            }
B3:         {
               int x, a, v
               …
            }
            …
         }
         …
     }
```

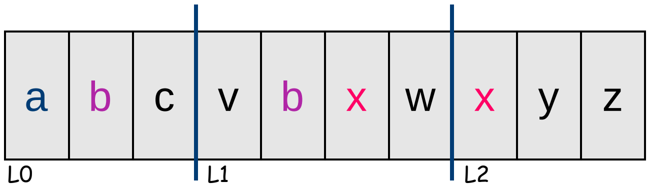
Fixed length data can always be at a constant offset from the beginning of a procedure's data area

- In our example, the *a* declared at level 0 will always be the first data element, stored at byte 0 in the fixed-length data area
- The *x* declared at level 1 will always be the sixth data item, stored at byte 20 in the fixed data area
- The *x* declared at level 2 will always be the eighth data item, stored at byte 28 in the fixed data area
- But what about the a declared in block B3, the second block at level 2?

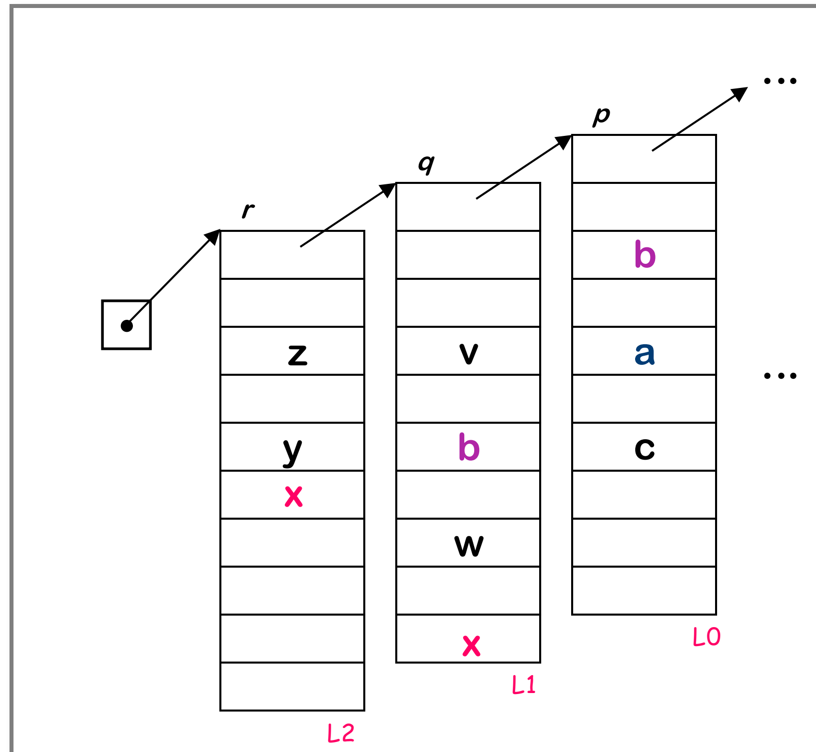
a	b	c	v	b	x	w	x	y	z
L0			L1				L2		

Storage in block B2

Tying It Back to the Scoped Symbol Table



Storage in block B2



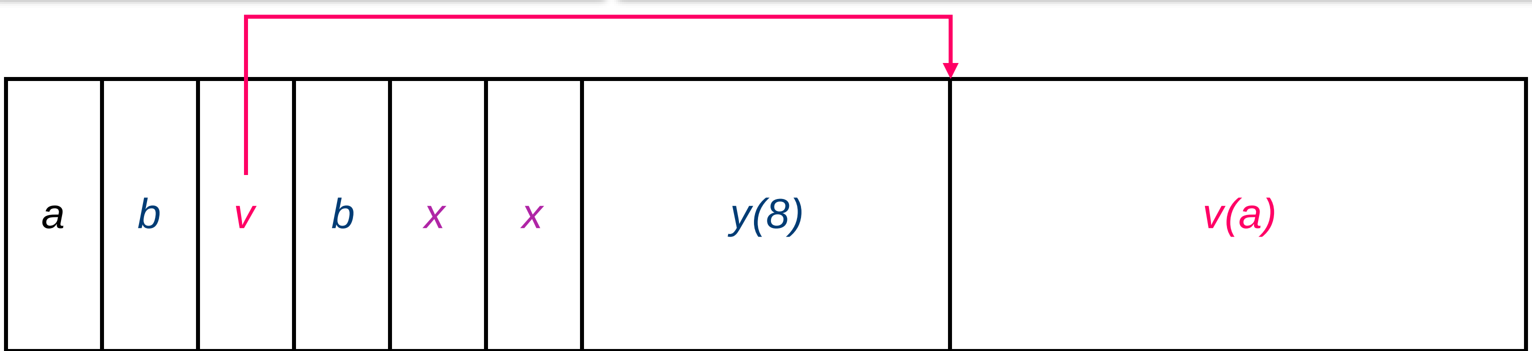
Variable-length Data

Note: The example code has changed from last slide.

```
B0: {int a, b
        …
        assign value to a
        …
B1:     {  int v(a), b, x
           …
B2:          {  int x, y(8)
                …
             }
        }
    }
```

Arrays

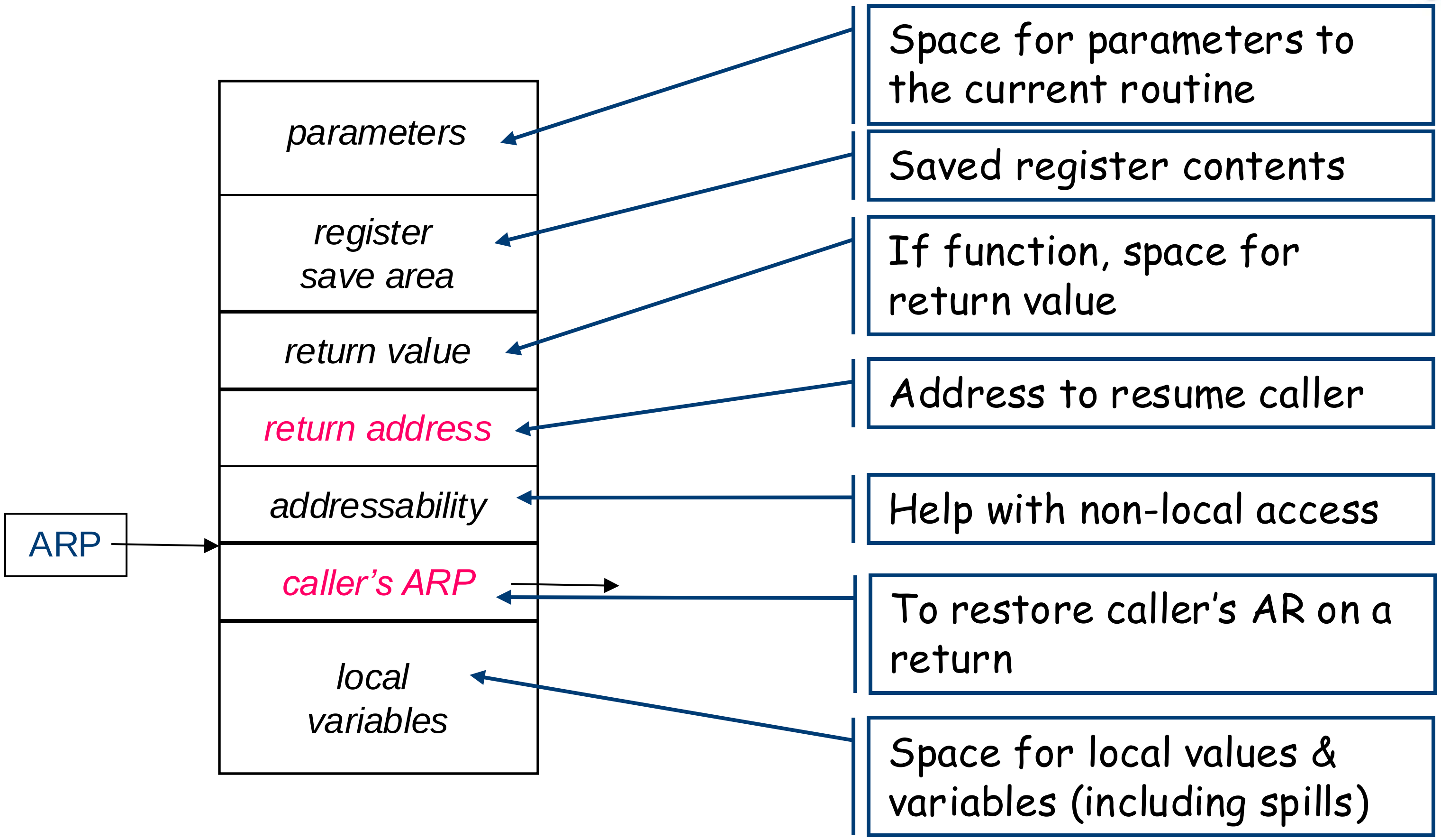
- → If size is fixed at compile time, store in fixed-length data area
- → If size is variable, store descriptor in fixed length area, with pointer to variable length area
- → Variable-length data area is assigned at the end of the fixed length area for the block in which it is allocated (including all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



Includes fixed length data for all blocks in the procedure …

Variable-length data area

Activation Record Basics



One AR for each invocation of a procedure

ARP ≈ Activation Record Pointer

Activation Record Details

How does the compiler find the variables?

- They are at known offsets from the AR pointer
- The static coordinate leads to a "loadAI" operation
  - Level specifies an ARP, offset is the constant

Variable-length data

- If AR can be extended, put it above local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

- Must generate explicit code to store the values
- Among the procedure's first actions

```
int x = 0;
```

is the same as

```
int x;
x = 0;
```

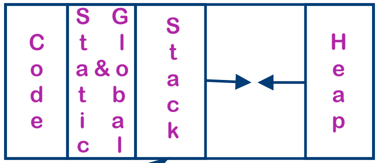
Activation Record Details

Where do activation records live?

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

⇒ Keep ARs on a <u>stack</u>

Algol-60 rules



Yes! That stack.

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

⇒ ARs <u>must</u> be kept in the heap

ML rules

- If a procedure makes no calls

⇒ AR can be allocated statically

Fortran 66 & 77

Efficiency prefers static, stack, then heap

STOP

Establishing Addressability

Must create base addresses

- Local variables
  - Convert to static data coordinate and use ARP + offset
- Global & static variables
  - Construct a label by mangling names (*i.e.*, &_fee)
- 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

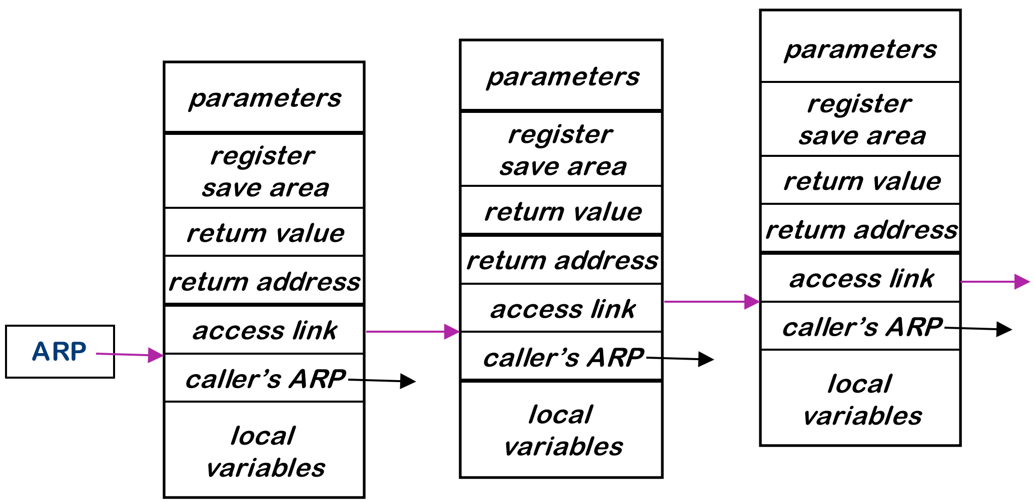
The "free variables" mentioned earlier; Lexical scoping has replaced deep binding for these variables with efficient lookups.

Establishing Addressability

Using Access Links to Find an ARP for a Non-Local Variable

- 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

Access & maintenance cost varies

All accesses are relative to ARP (r

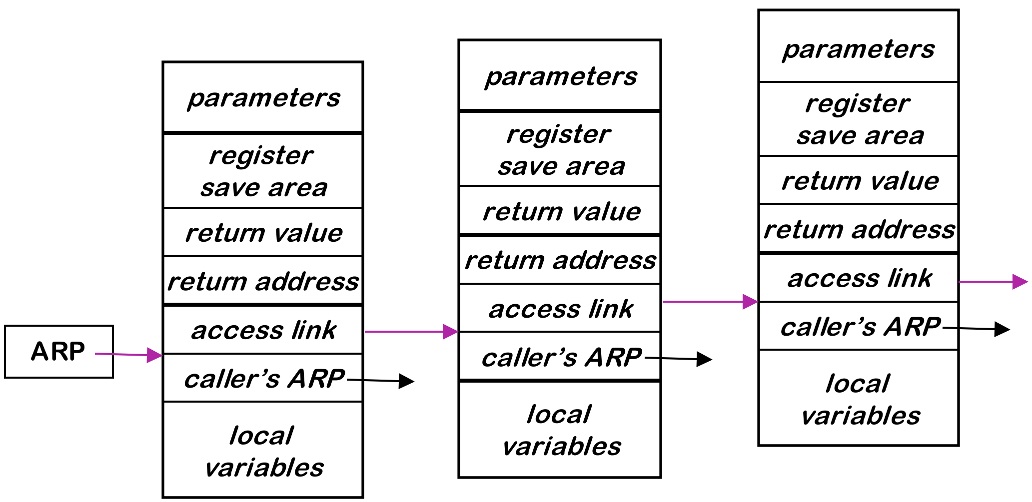
SC	Generated Code	
‹2,8›	loadAI r_0,8	$\Rightarrow r_{10}$
‹1,12›	loadAI r_0,-4	$\Rightarrow r_1$
	loadAI r_1,12	$\Rightarrow r_{10}$
‹0,16›	loadAI r_0,-4	$\Rightarrow r_1$
	loadAI r_1,-4	$\Rightarrow r_1$
	loadAI r_1,16	$\Rightarrow r_{10}$

Assume
- Current lexical level is 2
- Access link is at ARP - 4
- ARP is in r_0

Maintaining access link
- Calling level *k*+1
  → Use current ARP as link
- Calling level $j \le k$
  → Find ARP for level *j* -1
  → Use that ARP as link

Establishing Addressibility

Why does it work?

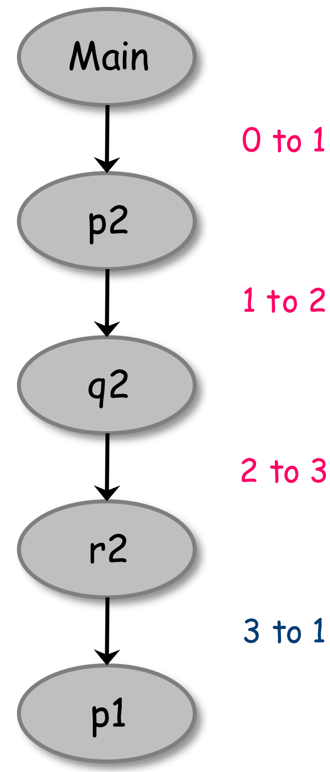


Maintaining access links
- Calling level *k*+1
→ Use current ARP as link
- Calling level *j* ≤ *k*
→ Find ARP for level *j* –1
→ Use that ARP as link

- If the call is to level k+1 the called procedure must be nested within the calling procedure
  - *Otherwise, we could not see it!*
- If the call is to level j>k, the called procedure must be nested within the containing procedure at level j-1

The Problem

```
procedure main {
    procedure p1 { … }
    procedure p2 {
        procedure q1 { … }
        procedure q2 {
            procedure r1 { … }
            procedure r2 {
                call p1; … // call up from level 3 to level 1
            }
            call r2;      // call down from level 2 to level 3
        }
        call q2;          // call down from level 1 to level 2
    }
    call p2;              // call down from level 0 to level 1
}
```



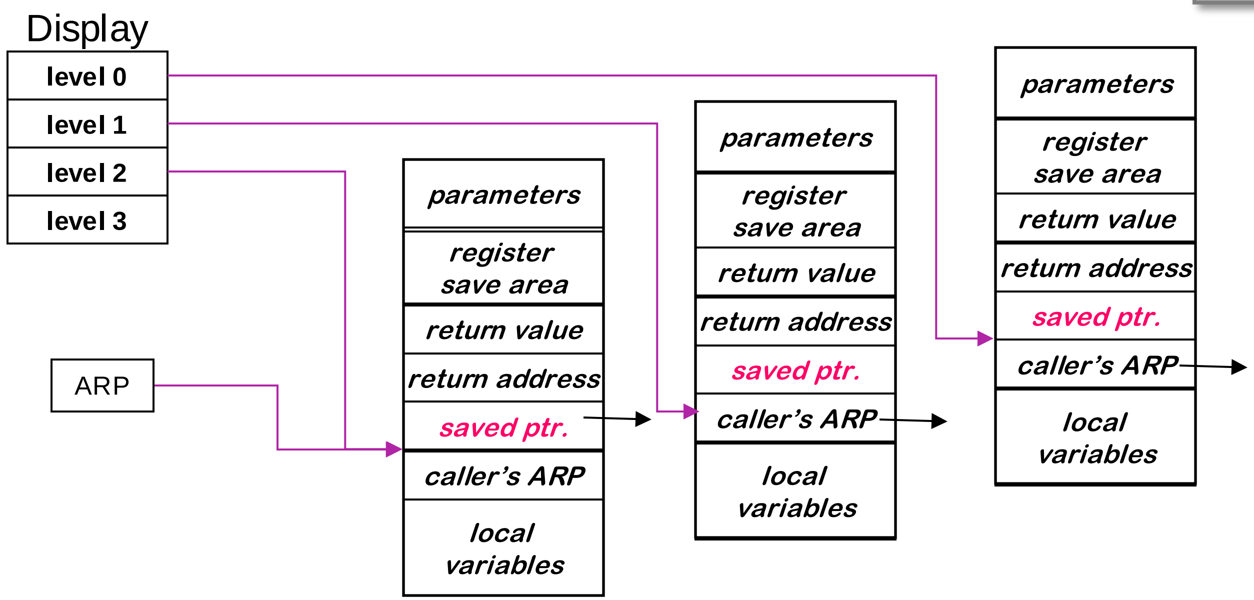
Call History

Establishing Addressability

Using a Display to Find an ARP for a Non-Local Variable

- 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	$\Rightarrow r_{10}$
<1,12>	loadI _disp	$\Rightarrow r_1$
	loadAI r_1,4	$\Rightarrow r_1$
	loadAI r_1,12	$\Rightarrow r_{10}$
<0,16>	loadI _disp	$\Rightarrow r_1$
	loadAI r_1,0	$\Rightarrow r_1$
	loadAI r_1,16	$\Rightarrow r_{10}$

Access & maintenance costs are fixed

Address of display may consume a register

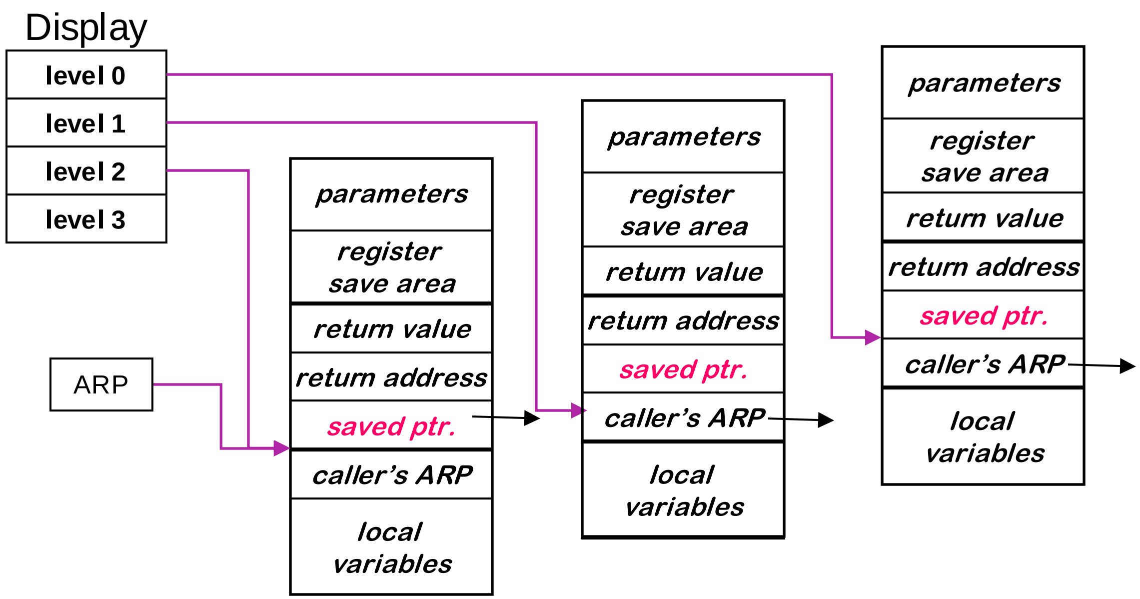
Desired AR is at _disp + 4 x *level*

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 old level *j* entry

Establishing Addressibility

Why does it work?



Maintaining access links

- 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 old level *j* entry

- If the call is from level k≥j, the display above the called procedure is the same as display[0:j-1] for the calling procedure
- If the call is from level j-1, it pays to save and restore display[j] anyway

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

Creating and Destroying Activation Records

All three parts of the procedure abstraction leave state in the activation record

Assume, for the moment, an Algol-60 environment where the activation information is dead on the return.

- How are ARs created and destroyed?
  - Procedure call must allocate & initialize *(preserve caller's world)*
  - Return must dismantle environment *(and restore caller's world)*
- Caller & callee must collaborate on the problem
  - Caller alone knows some of the necessary state
    - → Return address, parameter values, access to other scopes
  - Callee alone knows the rest
    - → Size of local data area (with spills), registers it will use

Their collaboration takes the form of a *linkage convention*

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, to allow interoperability

Saving Registers

Who saves the registers? Caller or callee?

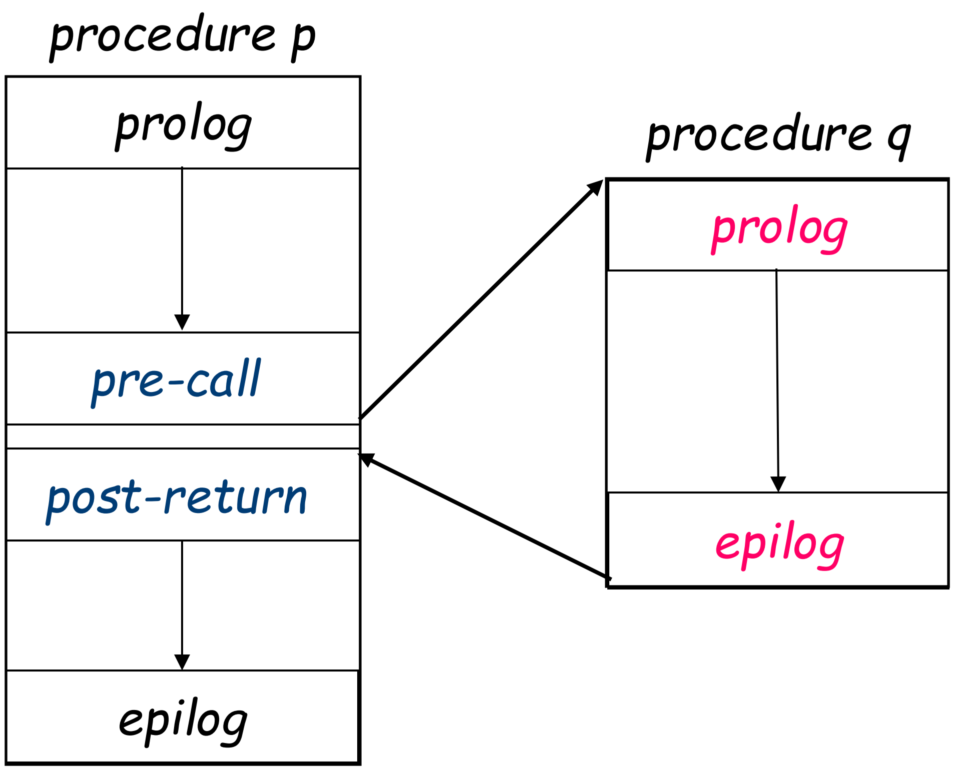
- Arguments for saving on each side of the call
  - Caller knows which values are LIVE across the call
  - Callee knows which registers it will use

- Conventional wisdom: divide registers into three sets
  - Caller saves registers
    - →Caller targets values that are not LIVE across the call
  - Callee saves registers
    - →Callee only uses these AFTER filling caller saves registers
  - Registers reserved for the linkage convention
    - →ARP, return address (if in a register), …

Where are they stored? In one of the ARs …

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 $\Rightarrow$ 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

Where do parameter values reside? (CW)
- In registers (*1st 3 or 4*)
- In callee's AR (*the rest*)

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 callee's environment
- Preserve parts of 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 a separate heap object for local variables

Procedure Linkages

Epilog Code

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

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

The Details

- Store return value?
  - Some implementations do this on the return statement
  - Others have return assign it & epilog store it into caller's AR
- 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

Back to Activation Records

If activation records are stored on the stack

Algol-60 rules

- Easy to extend — simply bump top of stack pointer
- Caller & callee share responsibility
  - Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
  - Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

ML rules

- Hard to extend
- Several options
  - Caller passes everything in registers; callee allocates & fills AR
  - Store parameters, return address, etc., in caller's AR !
  - Store callee's AR size in a defined static constant

Name mangling, again

Without recursion, activation records can be static

Fortran 66 & 77

Communicating Between Procedures

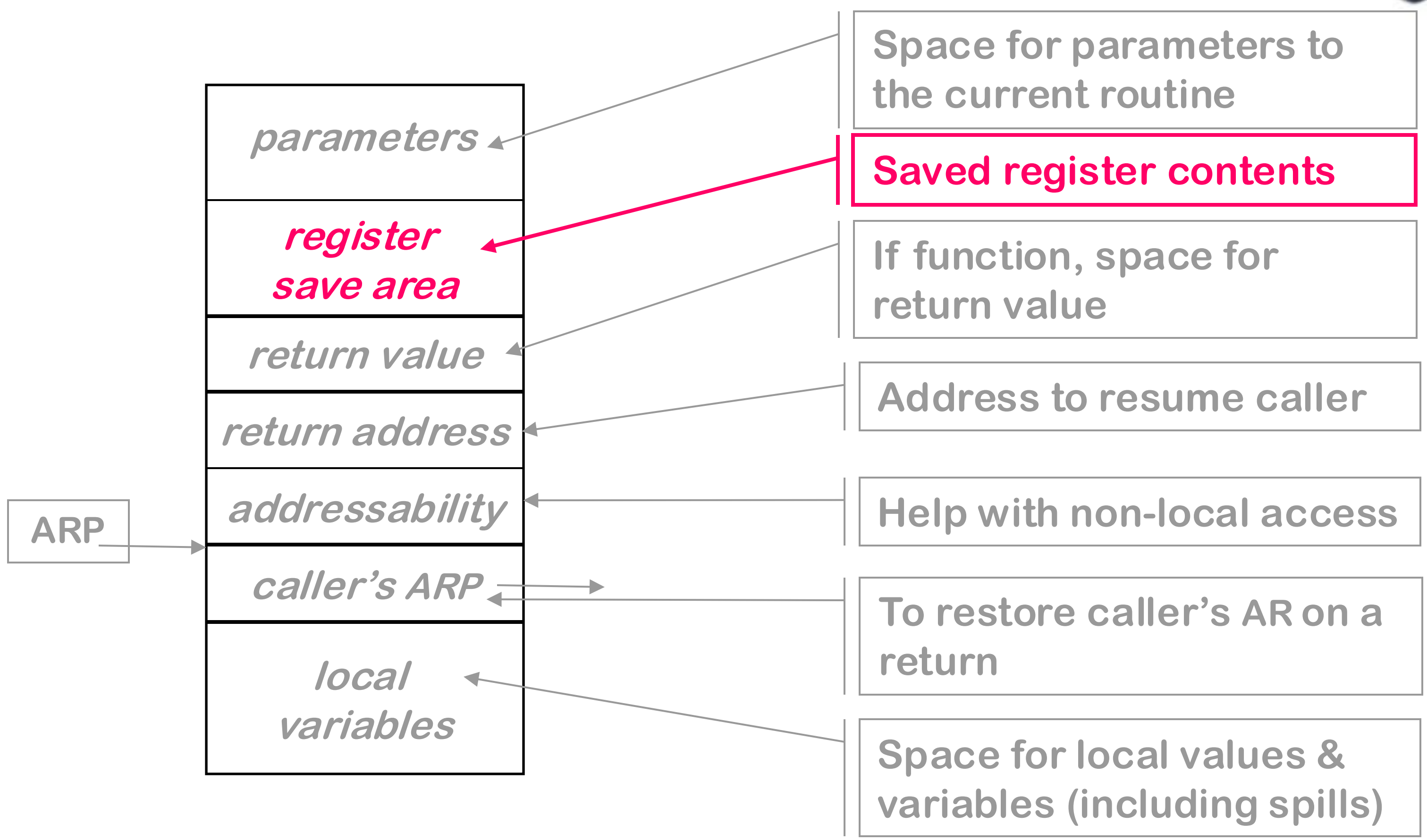
Most languages provide a parameter passing mechanism

⇒ 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 …

Remember This Drawing?



Makes sense to store *p*'s saved registers in *p*'s AR, although other conventions can work …

ARP ≈ Activation Record Pointer

More Thoughts on Register Save Code

Both memory access costs and number of registers are rising

- Cost of register save is rising (*time, code space, data space*)
- Worth exploring alternatives

Register Windows

- Register windows were a hot idea in the late 1980s
- Worked well for shallow calls with high register pressure
- Register stack overflows, leaf procedures hurt

A Software Approach

- Use library routine for save & restore
- Caller stores mask in callee's AR
- Callee stores its mask, a return address, and jumps to routine
- Saves code space & allows for customization

⇒ Store caller saves & callee saves together

Back to Activation Records

If activation records are stored on the stack

- Easy to extend — simply bump top of stack pointer
- Caller & callee share responsibility
  - Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
  - Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

- Hard to extend an allocated AR
- Either defer AR allocation into callee
  - Caller passes everything it can in registers
  - Callee allocates AR & stores register contents into it
    - → Extra parameters stored in caller's AR !
- Or, callee allocates a local data area on the heap

Requires one extra register

Static (e.g., non-recursive) is easy and inexpensive