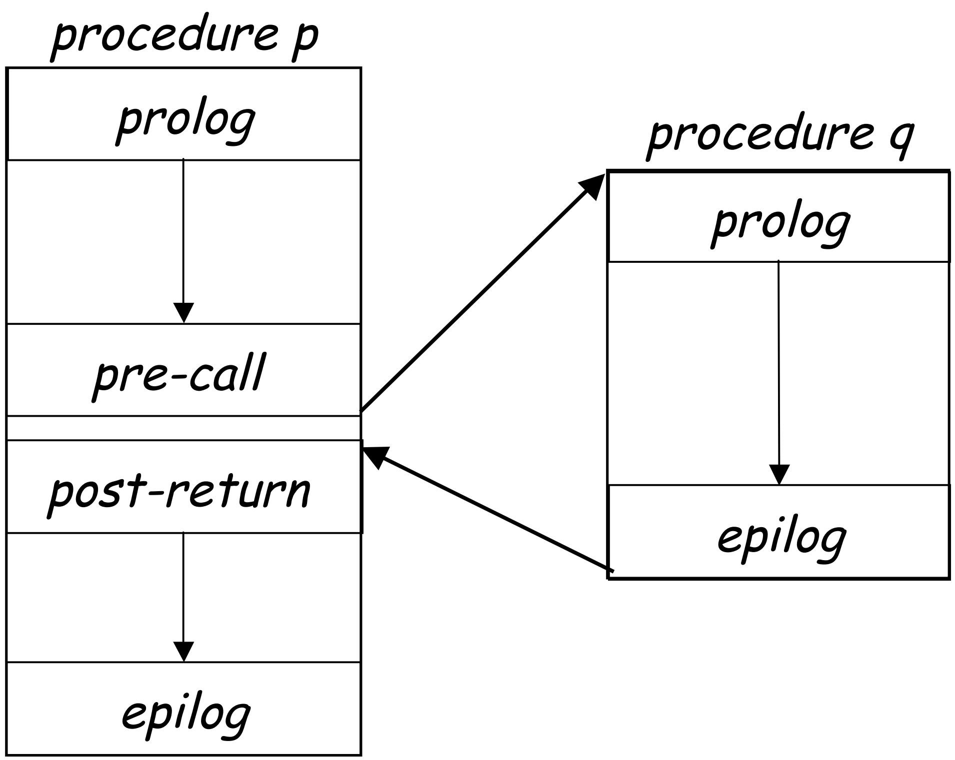
Code Shape IV
Procedure Calls & Dispatch

COMP 412
Fall 2005

Procedure Linkages

Standard procedure linkage

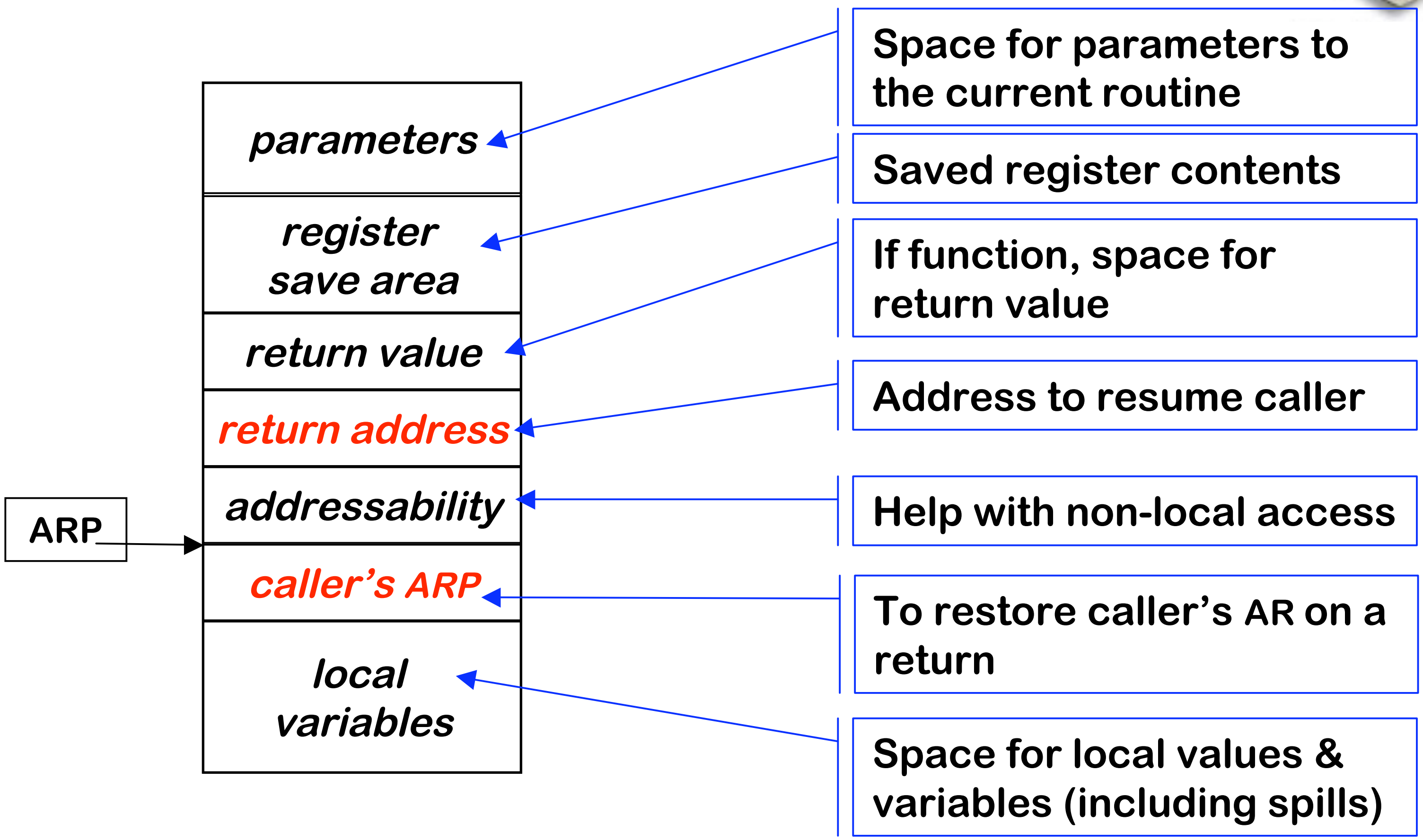


Procedure has
- standard prolog
- standard epilog

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

Activation Record Basics



One AR for each invocation of a procedure

Implementing Procedure Calls

If *p* calls *q* …

- In the code for *p*, compiler emits pre-call sequence
  - Evaluates each parameter & stores it appropriately
  - Loads the return address from a label
  - (with access links) sets up *q* 's access link
  - Branches to the entry of *q*

- In the code for *p*, compiler emits post-return sequence
  - Copy return value into appropriate location
  - Free *q* 's AR, if needed
  - Resume *p* 's execution

Invariant parts of pre-call sequence might be moved into the prolog

Implementing Procedure Calls

If *p* calls *q* …

- In the prolog, *q* must
  - Set up its execution environment
  - (with display) update the display entry for its lexical level
  - Allocate space for its (**AR** &) local variables & initialize them
  - If *q* calls other procedures, save the return address
  - Establish addressability for static data area(s)
- In the epilog, *q* must
  - Store return value (unless "return" statement already did so)
  - (with display) restore the display entry for its lexical level
  - Restore the return address (*if saved*)
  - Begin restoring *p*'s environment
  - Load return address and branch to it

Implementing Procedure Calls

If *p* calls *q*, one of them must

- Preserve register values (*caller-saves versus callee saves*)
  - Caller-saves registers stored/restored by *p* in *p*'s AR
  - Callee-saves registers stored/restored by *q* in *q*'s AR
- Allocate the AR
  - Heap allocation ⇒ callee allocates its own AR
  - Stack allocation ⇒ caller & callee cooperate to allocate AR

Space tradeoff

- Pre-call & post-return occur on every call
- Prolog & epilog occur once per procedure
- More calls than procedures
  - Moving operations into prolog/epilog saves space

Implementing Procedure Calls

If *p* calls *q*, one of them must

- Preserve register values (caller-saves versus callee saves)

If space is an issue

- Moving code to prolog & epilog saves space
- As register sets grow, save/restore code does, too { 32, 64, 128, 256
  - Each saved register costs 2 operations
  - Can use a library routine to save/restore
    - Pass it a mask to determine actions & pointer to space
    - Hardware support for save/restore or storeM/loadM

Can decouple who saves from what is saved

Implementing Procedure Calls

If *p* calls *q*, one of them must

- Preserve register values (caller-saves versus callee saves)

If code space is an issue

- All saves in prolog, all restores in epilog
  - Caller provides a bit mask for caller-saves registers
  - Callee provides a bit mask for callee-saves registers
  - Store all of them in same AR (***either caller or callee***)
  - Efficient use of time and code space
  - May waste some register save space in the AR
- Caller-save & callee-save assign responsibility not work

Implementing Procedure Calls

Evaluating parameters

- Call by reference ⇒ evaluate parameter to an lvalue
- Call by value ⇒ evaluate parameter to an rvalue & store it

Aggregates, arrays, & strings are usually c-b-r

- Language definition issues
- Alternative is copying them at each procedure call **($$)**
  - Small structures can be passed in registers *(in & out)*
  - Can pass large c-b-v objects c-b-r and copy on modification

AIX does this for C

Implementing Procedure Calls

Evaluating parameters

- Call by reference ⇒ evaluate parameter to an lvalue
- Call by value ⇒ evaluate parameter to an rvalue & store it

Procedure-valued parameters

- Must pass starting address of procedure
- With access links, need the lexical level as well
  - Procedure value is a static coordinate < ***level,address*** >
    - May also need shared data areas (*file-level scopes*)
    - In-file & out-of-file calls have (*slightly*) different costs
  - This lets the caller set up the appropriate access link

Implementing Procedure Calls

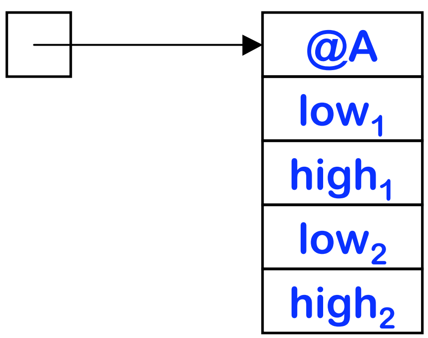
What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters

- Callee needs dimension information ⇒ build a *dope vector*
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference

Some improvement is possible

- Save len_i and low_i rather than low_i and $high_i$
- Pre-compute the fixed terms in prologue sequence



What about call-by-value?

- Most c-b-v languages pass arrays by reference
- This is a language design issue

Implementing Procedure Calls

What about A[12] as an actual parameter?

If corresponding parameter is a scalar, it's easy

- Pass the address or value, as needed
- Must know about both formal & actual parameter
- Language definition must force this interpretation

What is corresponding parameter is an array?

- Must know about both formal & actual parameter
- Meaning must be well-defined and understood
- Cross-procedural checking of conformability

⇒ Again, we're treading on language design issues

Fortran 77 lets amazing things happen in this case…

An Aside That Doesn't Fit Well Anywhere …

What about code for access to variable-sized arrays?

Local arrays dimensioned by actual parameters

- Same set of problems as parameter arrays
- Requires dope vectors (or equivalent)
  - Place dope vector at fixed offset in activation record

⇒ Different access costs for textually similar references

This presents lots of opportunities for a good optimizer

- Common subexpressions in the address polynomial
- Contents of dope vector are fixed during each activation
- Should be able to recover much of the lost ground

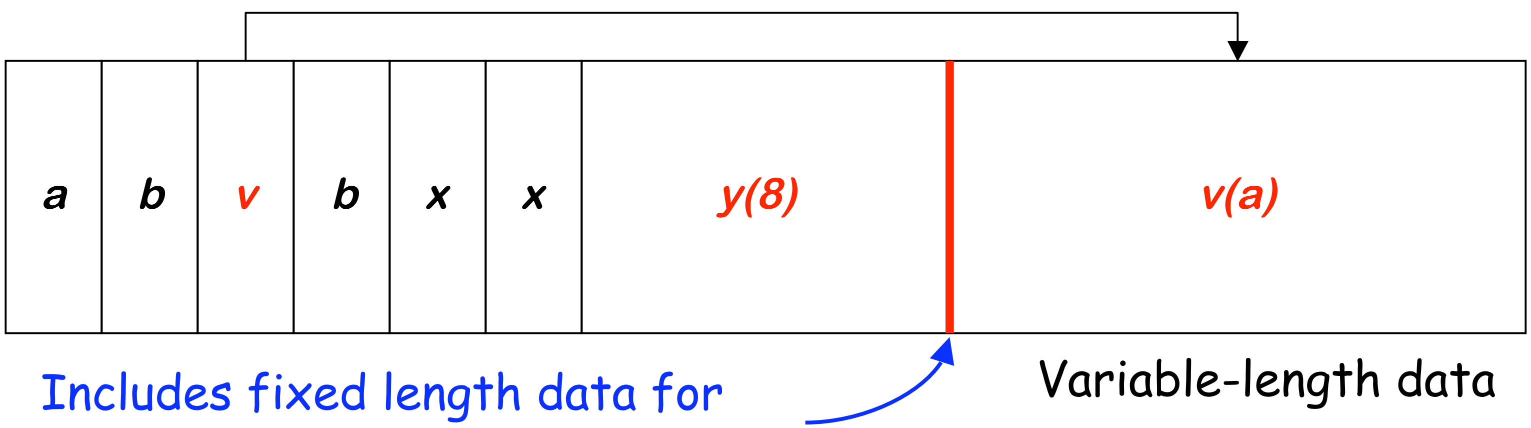
⇒ Handle them like parameter arrays

Variable-length Data

```
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 block in which it is allocated



Implementing Procedure Calls

What about a string-valued argument?

- Call by reference ⇒ pass a pointer to the start of the string
  - Works with either length/contents or null-terminated string
- Call by value ⇒ copy the string & pass it
  - Can store it in caller's AR or callee's AR
  - Callee's AR works well with stack-allocated ARs
  - Can pass by reference & have callee copy it if necessary …

Pointer functions as a "descriptor" for the string, stored in the appropriate location (register or slot in the AR)

Implementing Procedure Calls

What about a structure-valued parameter?

- Again, pass a descriptor
- Call by reference ⇒ descriptor (pointer) refers to original
- Call by value ⇒ create copy & pass its descriptor
  - Can allocate it in either caller's AR or callee's AR
  - Callee's AR works well with stack-allocated ARs
  - Can pass by reference & have callee copy it if necessary …

If it is actually an array of structures, then use a dope vector

If it is an element of an array of structures, then …

What About Calls in an OOL (Dispatch)?

In an OOL, most calls are indirect calls

- Compiled code does not contain address of callee
  - Finds it by indirection through class' method table
  - Required to make subclass calls find right methods
  - Code compiled in class *C* cannot know of subclass methods that override methods in *C* and *C*'s superclasses
- In the general case, need dynamic dispatch
  - Map method name to a search key
  - Perform a run-time search through hierarchy
    - Start with object's class, search for 1st occurrence of key
    - This can be expensive
  - Use a method cache to speed search
    - Cache holds < *key,class,method pointer* >

How big?
Bigger ⇒ more hits & longer search
Smaller ⇒ fewer hits, faster search

What About Calls in an OOL (Dispatch)?

Improvements are possible in special cases

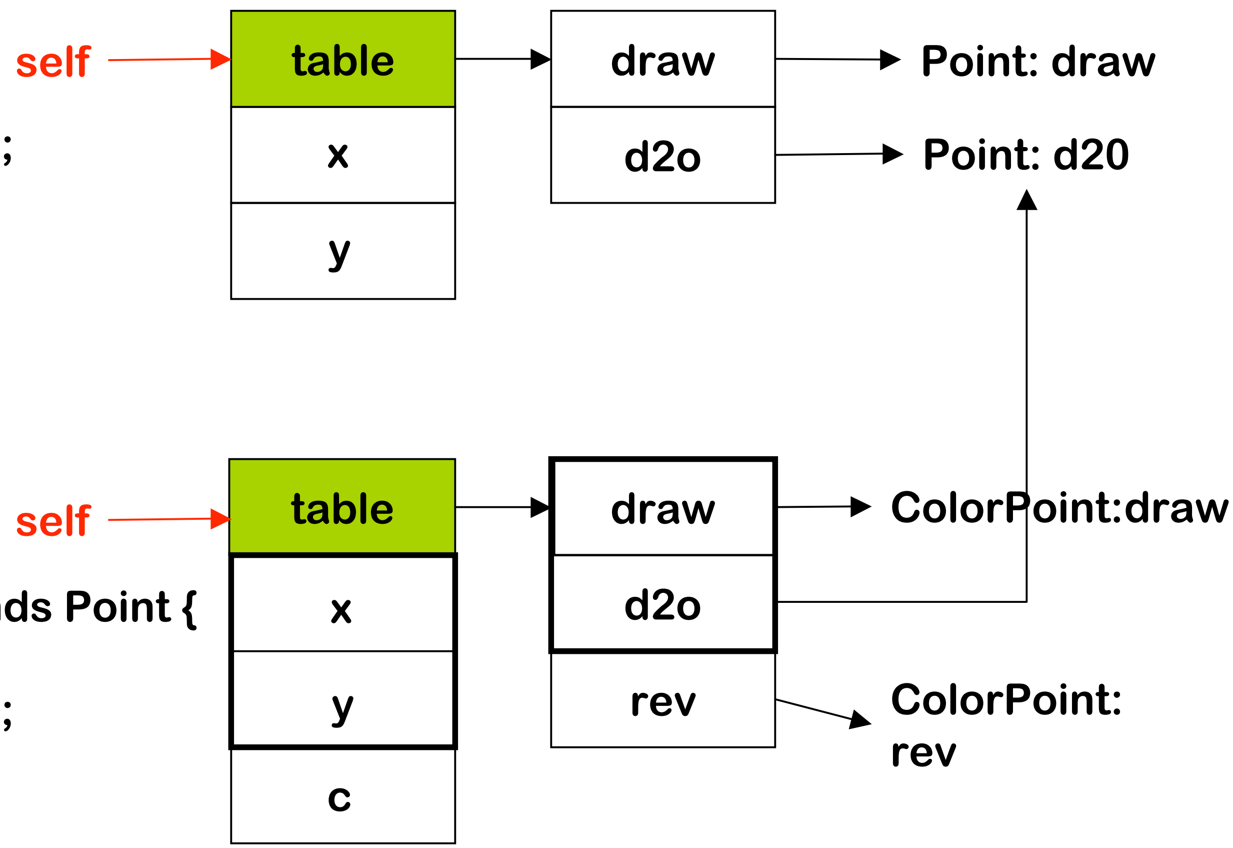
- If class has no subclasses, can generate direct call
  - Class structure must be static or class must be **FINAL**
- If class structure is static (*language design issue*)
  - Can generate complete method table for each class
  - Single indirection through class pointer (***1 or 2 operations***)
  - Keeps overhead at a low level
- If class structure changes infrequently (*behavioral issue*)
  - Build complete method tables at run time
  - Initialization & any time class structure changes
- If running program can create new classes, … (*design, again*)
  - Well, not all things can be done quickly

Single Inheritance and Dynamic Dispatch

- Use prefixing of tables

```
Class Point {
        int x, y;
        public void draw();
        public void d2o();
}
```

```
Class ColorPoint extends Point {
        Color c;
        public void draw();
        public void rev();
}
```



What About Calls in an OOL (Dispatch)?

Unusual issues in OOL call

- Need to pass receiver's object record as (1st) parameter
  - Becomes self or this
- Typical OOL has lexical scoping in method
  - Limited to block-style scoping ⇒ no need for access links
  - Can overlay successive block scopes in same method (***reuse***)
- Method needs access to its class
  - Object record has static pointer to class, to superclass, and …
  - Class pointers don't need updating like access-links
- Method is a full-fledged procedure
  - It still needs an AR …
  - Can often stack allocate them (***HotSpot does …***)

What About setjmp() and longjmp() ?

Unix system calls to implement abnormal returns

- Setjmp() stores a descriptor, *d*, for use with longjmp()
- Invoking longjump(*d*) causes execution to continue at the point after the setjump() call that created *d*

How can we implement setjmp() & longjmp() ?

- Setjmp() must store ARP and return address in descriptor
  - What about values of registers and variables?
  - If they are to be preserved, must compute a closure
    - Stack-allocated ARs ⇒ copy the stack
    - Heap-allocated ARs ⇒ keep a pointer & don't free the AR
- Longjmp() must restore environment at setjmp()
  - Restore ARP & discard ARs created since setjmp()
    - Cheap with stack-allocated ARs, might cost more with heap

Representing and Manipulating Strings

Character strings differ from scalars, arrays, & structures

- Fundamental unit is a character
  - Typical sizes are one or two bytes
  - Target ISA may (or may not) support character-size operations
- Set of supported operations on strings is limited
  - Assignment, length, concatenation, translation (?)
- Efficient string operations are complex on most RISC ISAs
  - Ties into representation, linkage convention, & source language

§ 7.6 in EaC

Representing and Manipulating Strings

Two common representations

- Explicit length field

8	a	b̸	s	t	r	i	n	g

↑ @b

Length field may take more space than terminator

- Null termination

a	b̸	s	t	r	i	n	g	\0

↑ @b

- Language design issue
  - Fixed-length versus varying-length strings *(1 or 2 length fields)*

Representing and Manipulating Strings

Each representation as advantages and disadvantages

Operation	Explicit Length	Null Termination
Assignment	Straightforward	Straightforward
Checked Assignment	Checking is easy	Must count length
Length	O(1)	O(n)
Concatenation	Must copy data	Length + copy data

Unfortunately, null termination is almost considered normal

- Hangover from design of C
- Embedded in OS and API designs

Manipulating Strings

Single character assignment

- With character operations
  - Compute address of rhs, load character
  - Compute address of lhs, store character
- With only word operations *(>1 char per word)*
  - Compute address of word containing rhs & load it
  - Move character to destination position within word
  - Compute address of word containing lhs & load it
  - Mask out current character & mask in new character
  - Store lhs word back into place

Manipulating Strings

Multiple character assignment

Two strategies

1. Wrap a loop around the single character code, or
2. Work up to a word-aligned case, repeat whole word moves, and handle any partial-word end case

- With character operations
  1. Easy to generate; inefficient use of resources
  2. Harder to generate; better use of resources
- With only word operations
  1. Lots of complication to generate; inefficient at runtime, too
  2. Fold complications into end case; reasonable efficiency

Requires explicit code to check for buffer overflow (⇒ length)

Source & destination aligned differently
⇒ much harder cases for word operations

Manipulating Strings

Concatenation

- String concatenation is a length computation followed by a pair of whole-string assignments
  - Touches every character
- Exposes representation issues
  - Is string a descriptor that points to text?
  - Is string a buffer that holds the text?
  - Consider a ← b || c
    - Compute b || c and assign descriptor to a?
    - Compute b || c into a temporary & copy it into a?
    - Compute b || c directly into a?
- What about call fee(b || c) ?

Manipulating Strings

Length Computation

- Representation determines cost
  - Explicit length turns length(b) into a memory reference
  - Null termination turns length(b) into a loop of memory references and arithmetic operations
- Length computation arises in other contexts
  - Whole-string or substring assignment
  - Checked assignment (buffer overflow)
  - Concatenation
  - Evaluating call-by-value actual parameter or concatenation as an actual parameter