

*The Procedure Abstraction:
Part II
Comp 412*

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Review



From last lecture

The Procedure serves as

- A control abstraction
- A naming abstraction
- An external interface

{ Access to system services,
libraries, code from others ...

We covered the control abstraction last lecture.

Today, we will focus on **naming**.

The Procedure as a Name Space



Each procedure creates its own name space

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the procedure
 - Nested procedures are “inside” by definition
- We call this set of rules & conventions “lexical scoping”

The Java twist: allow fully qualified names to reach around scope rules

Examples

- C has global, static, local, and *block* scopes (Fortran-like)
 - Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes (*let*)
 - Procedure scope (typically) contains formal parameters

The Procedure as a Name Space



Why introduce lexical scoping?

- Provides a compile-time mechanism for binding “free” variables
- Simplifies rules for naming & resolves conflicts
- Lets the programmer introduce “local” names with impunity

How can the compiler keep track of all those names?

The Problem

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

The Answer

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



Do People Use This Stuff ?

C macro from the MSCP compiler

```

#define fix_inequality(oper, new_opcode) \
  if (value0 < value1) \
  { \
    Unsigned_Int temp = value0; \
    value0 = value1; \
    value1 = temp; \
    opcode_name = new_opcode; \
    temp = oper->arguments[0]; \
    oper->arguments[0] = oper->arguments[1]; \
    oper->arguments[1] = temp; \
    oper->opcode = new_opcode; \
  }

```

Even in C, a language not known for abstraction, people do!

Declares a new name

§ 5.7 in EaC



Lexically-scoped Symbol Tables

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*

Many implementation schemes have been proposed (see § B.4)

- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & (yes) fun
 - Good alternatives exist (*multiset discrimination, acyclic DFAs*)

Symbol tables are compile-time structures that the compiler uses to resolve references to names. We'll see the corresponding run-time structures that are used to establish addressability later.



Example

```

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
      ...
    }
    ...
  }
}

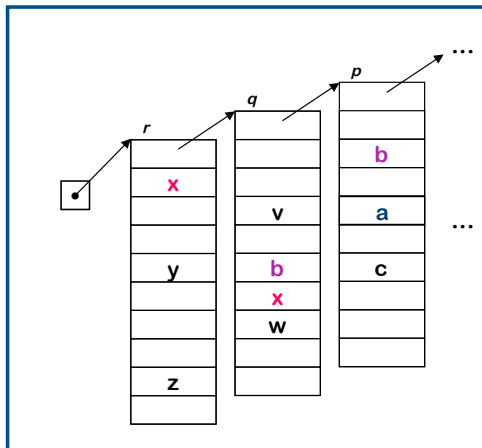
```



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 level *p* table if it is top table in the chain

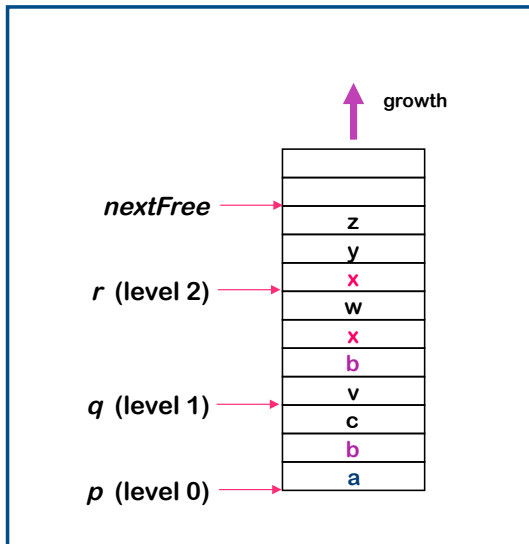
If the compiler must preserve the table (*for, say, the debugger*), this idea is actually practical.

Individual tables are hash tables.

Implementing Lexically Scoped Symbol Tables



Stack organization for the individual tables



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 much less space

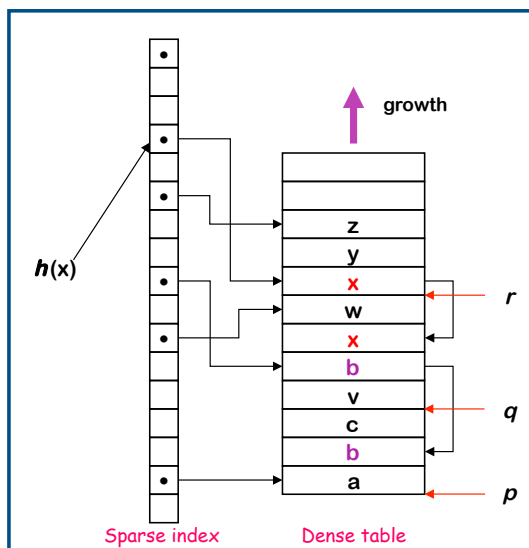
Disadvantage

- Lookups can be expensive

Implementing Lexically Scoped Symbol Tables



Threaded stack organization for the individual tables



Implementation

- *insert()* puts new entry at the head of the list for the name
- *lookup()* goes direct to location
- *delete()* processes each element in level being deleted to remove from head of list
- use sparse index for speed
- use dense table to limit space

Advantage

- lookup is fast

Disadvantage

- delete takes time proportional to number of declared variables in level

The Procedure as an External Interface



Naming plays a critical role in our ability to use procedure calls as a general interface

OS needs a way to start the program's execution

- Programmer needs a way to indicate where it begins
 - The procedure "main" in most languages
- When user invokes "grep" at a command line
 - OS finds the executable
 - OS creates a process and arranges for it to run "grep"
 - Conceptually, it does a fork() and an exec() of the executable "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 specific discussion

Where Do All These Variables Go?



Automatic & Local

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

Static

- Procedure scope ⇒ storage area affixed with procedure name
- File scope ⇒ storage area affixed with file name
 - Label them with some mangled name, such as &p.x
- Lifetime is entire execution

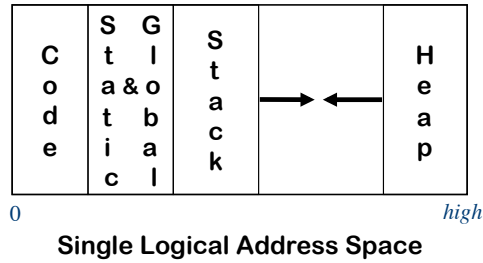
Global

- One or more named global data areas
 - One per variable, or per file, or per program, ...
 - Label either variable or data area with some mangled name
- Lifetime is entire execution



Placing Run-time Data Structures

Classic Organization



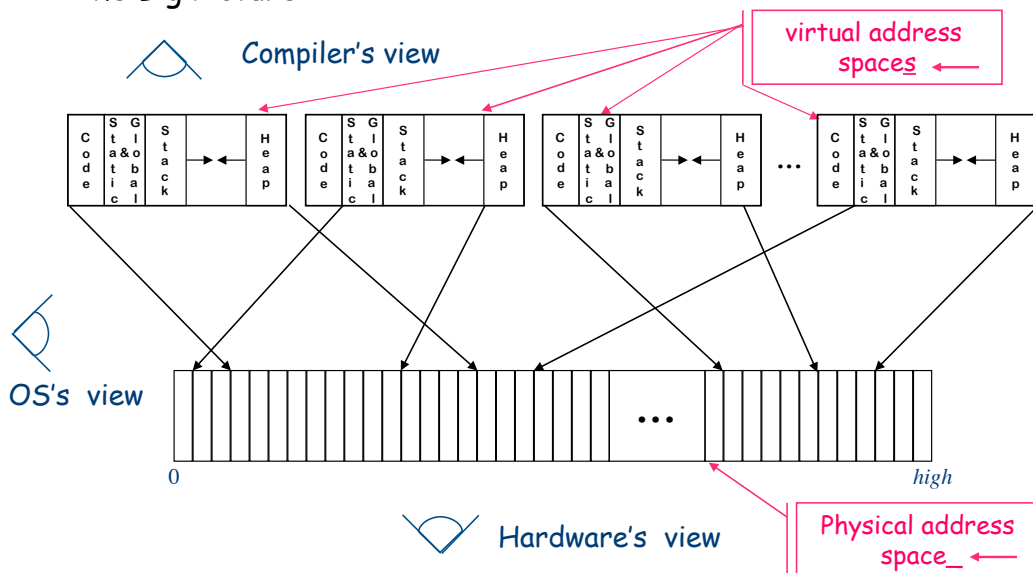
- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved
- Uses address space, not allocated memory

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



How Does This Really Work?

The Big Picture

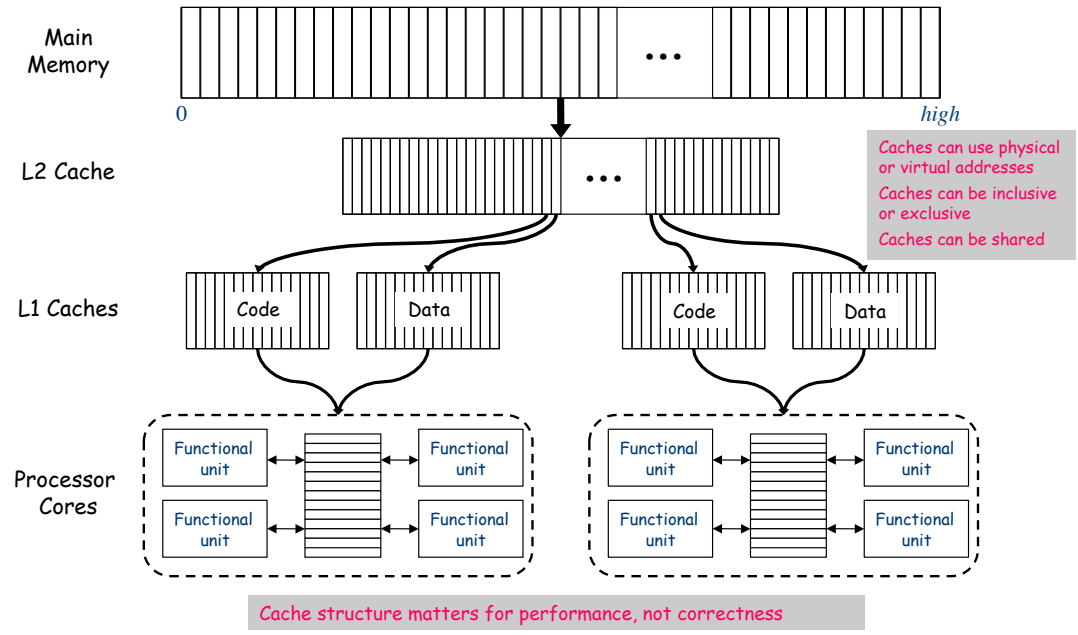


How Does This Really Work?

Some systems include L3 caches



Of course, it is no longer that simple



Where Do Local Variables Live?



A Simplistic 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 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

Recall that "static" means knowable at **compile time** (and, therefore, fixed).



Translating Local Names

How does the compiler represent a specific instance of x ?

- Name is translated into a *static coordinate*
 - $\langle \text{level}, \text{offset} \rangle$ 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
    ...
    }
    ...
  }
  ...
}

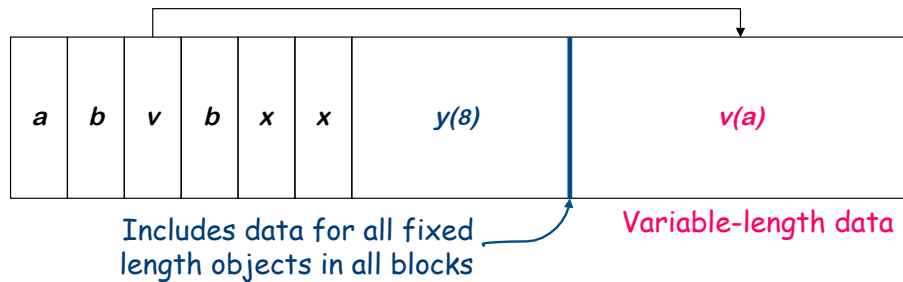
```

- Fixed length data can always be at a constant offset from the beginning of a procedure
 - 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 the second block at **level 2**?



Variable-length Data

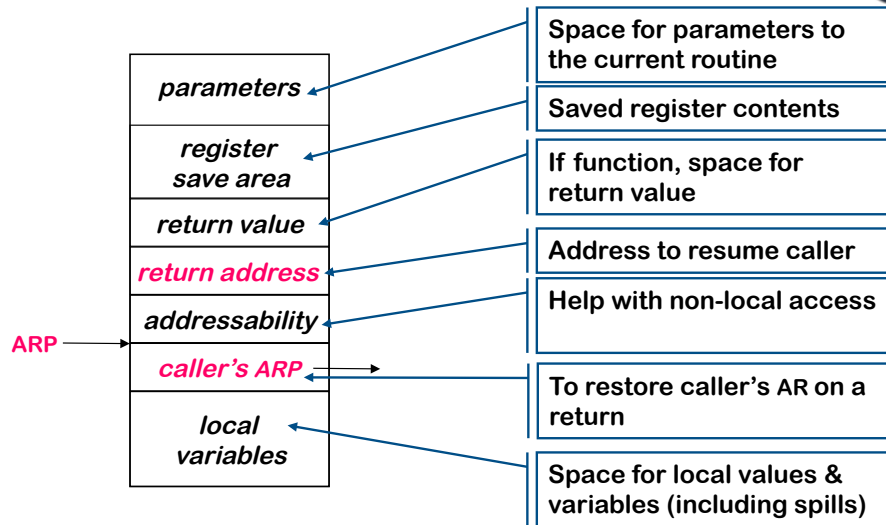
<pre> B0: { int a, b ... assign value to a B1: { int v(a), b, x ... B2: { int x, y(8) ... } } } </pre>	<p>Arrays</p> <ul style="list-style-type: none"> → If size is fixed at compile time, store in fixed-length data area → If size is variable, store descriptor in fixed length area, with pointer to variable 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)
--	--



18



Activation Record Basics



One AR for each invocation of a procedure



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

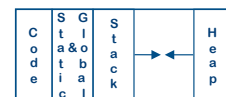


Activation Record Details

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