The Procedure Abstraction
Part II: Symbol Tables and Activation Records
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?

```plaintext
procedure p {
  int a, b, c
  ....
  {
    int v, b, x, w
    ....
  }
}
```
The Procedure as a Name Space

The Problem

• At point \( X \) in the execution of the program, which declaration of “\( b \)” is current?
• At run-time, where is “\( b \)” found?
• How does compiler delete “\( b \)” going in & out of scopes?

The Answer

• The compiler must model the name space
• Lexically scoped symbol tables

```plaintext
procedure p {
    int \( a, b, c \)
    ....
    {
        int \( v, b, x, w \)
        ....
    }
}
```
Lexically-scoped Symbol Tables

The problem

- The compiler needs a distinct record for each declaration
- Nested lexical scopes allow 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`

§ 5.7 in EaC
High-level idea

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

B0: procedure b {
    int x, y, z
B1: {
    int v, b, x, w
B2: {
    int a, b, c
        ....
    }
B3: {
    int x, a, v
        ...
    }
        ...
    }
    ...
}
The Procedure as an External Interface

OS needs a way to start the program’s execution

All function code here

Activation records for executing functions “foo” and hello.txt here

0 high
The Procedure as an External Interface

- Local variables go in function’s activation record
- Code & Stack
- Heap
- main function

- Stack on stack of ARs
- AR of fopen
- AR of main

- fopen
- main

0 high
Where Do All These Variables Go?

Local

- Keep them in the procedure activation record or in a register
- Automatic ⇒ lifetime matches procedure’s lifetime
Where Do All These Variables Go?

Local

- Keep them in the procedure activation record or in a register
- Automatic 🔄 lifetime matches procedure’s lifetime

![Diagram showing stack and heap organization with ARs for fopen and main functions]
Where Do All These Variables Go?

Static (e.g., in C language)

- File scope \(\Rightarrow\) storage area affixed with file name
- Lifetime is entire execution

![Diagram showing the allocation of static variables in C language]

- Code Static Static
- Heap
- Stack
- On stack of ARs
- \(fopen\)
- \(main\)
- AR of fopen
- AR of main
Where Do All These Variables Go?

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

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Static</th>
<th>Global</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- AR of fopen
- AR of main

- Stack
- Heap
- Code
- Static

on stack of ARs
Placing Run-time Data Structures

Classic Organization

- Code, static, & global data have known size
- Heap & stack both grow & shrink over time
- This is a virtual address space
How Does This Really Work?

The Big Picture

Compiler's view

OS's view

Hardware's view

virtual address spaces

Physical address space

Hardware's view
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 store control info there!
Activation Records

Top of Stack

- ARP+n
  - Local-Data Area
    - Caller’s ARP
    - Addressability
    - Return Address
    - Return Value
  - Register-Save Area
  - Parameters

ARP

ARP-m

Callee’s AR

Local-Data Area
- Caller’s ARP
- Addressability
- Return Address
- Return Value

Register-Save Area
- Parameters

Caller’s AR

...
Local-Data Area

BO: {
    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 ...
Activation Record Basics

<table>
<thead>
<tr>
<th>TOS</th>
<th>Space for parameters to the current routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>Saved register contents</td>
</tr>
<tr>
<td>register</td>
<td>If function, space for return value</td>
</tr>
<tr>
<td>save area</td>
<td>Address to resume caller</td>
</tr>
<tr>
<td>return value</td>
<td>Space for local values &amp; variables (including spills)</td>
</tr>
<tr>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>local variables</td>
<td></td>
</tr>
</tbody>
</table>

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

Standard procedure linkage

**procedure p**

- **prolog**
- **pre-call**
- **post-return**
- **epilog**

**procedure q**

- **prolog**
- **epilog**

**callee**

**caller**

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
Pre-call Sequence

• Sets up callee’s basic AR
• Helps preserve its own environment

The Details

• Allocate space for the callee’s AR
• Evaluates each parameter & stores value or address
• Saves return address, caller’s ARP 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

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
**Post-return Sequence**

- Restores caller’s environment

**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

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
**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
- 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
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
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
• Load return address from AR
• Restore caller’s ARP
• Jump to the return address