Static Chain Maintenance

- The static chain must be modified for each subroutine call and return
- The return part is trivial
  - When a subroutine terminates, its activation record is simply removed
- The call part is more complex
  - When a subroutine is called, its activation record needs to be built
  - Two methods to construct static links

Static Link Construction 1

- When a subroutine is called, search the dynamic chain until the first one of the parent scope is found
- However, this search can be avoided by treating subroutine declarations and calls as variable definitions and references
Static Link Construction 2

• At compile time,
  – When the compiler encounters a subroutine call A() in subroutine C, it determines the subroutine B which declares A
  – It then computes the nesting_depth between C and B
  – The information is stored and can be accessed by subroutine call during execution
  – When A is called, the static link to B is determined by moving down the static chain of C() nesting_depth hops

Issues with Static Chains

• A nonlocal reference is slow if the nesting depth is large
  – In practice, references to distant nonlocal variables are rare
• Time-critical code is difficult
  – Costs of nonlocal references are difficult to determine
  – Code modifications can change nesting depth, and therefore the cost
Display

- An alternative to static chains to solve the problems
- Static links are stored in an auxiliary data structure called a display
- The content of the display is a list of addresses of accessible activation record instances
- However, it has not been found to be superior to the static-chain method

Blocks

- **Blocks** are user-specified local scopes for variables
- An example in C
  ```c
  { int temp;
    temp = list[upper];
    list[upper] = list[lower];
    list[lower] = temp;
  }
  ```
- The life time of the variable `temp` begins when control enters the block, and ends when control exits it
Advantage

• The local variables inside blocks cannot interfere with any other variable with the same name but declared elsewhere in the program

Implementing Blocks

• Two methods to implement block local variables
  – Treat blocks as parameter-less subroutines
  – Treat block variables as plain local variables
Method 1

• Treat blocks as parameter-less subroutines that are always called from the same location
  – Every block has an activation record
  – An instance is created every time the block is executed
  – However, blocks can be implemented in a simpler and more efficient way

Method 2

• Insight
  – The maximum amount of storage required for block variables can be statically determined, because blocks are entered and exited in strictly textual order
• The block variables are allocated after local variables in the activation record
• Offset for all block variables can be statically computed, so block variables can be addressed exactly as if they were local variables
An Example

```c
void main() {
    int x, y, z;
    while (…) {
        int a, b, c;
        while (…) {
            int d, e;
        }
    }
    while (…) {
        int f, g;
    }
}
```

Implementing Dynamic Scoping

- Two possible ways to implement local and nonlocal variables in a dynamic-scoped language
  - Deep access
  - Shallow access
- These are different from deep and shallow binding (different semantics)
- The semantics of dynamic scoping are unaltered by the access method
Deep Access

- Nonlocal references are found by searching the activation record instances on the dynamic chain
  - Length of the chain cannot be statically determined
  - Every activation record instance must have variable names

An Example

```c
void sub3() {
    int x, z;
    x = u + v;
    ...
}  
void sub2() {
    int w, x;
    ...
}  
void sub1() {
    int v, w;
    ...
}  
void main() {
    int v, u;
    ...
}
```

Where:
- main calls sub1
- sub1 calls sub1
- sub1 calls sub2
- sub2 calls sub3

How are the definitions of u and v found?
Shallow Access

- **Key insight**
  - With dynamic scoping, there is at most one visible version of a variable of any specific name at a given time

- **Have a separate stack for each variable name in a program**
  - When a variable is created, it is given a cell at the top of the stack for its name
  - Every reference to the name is to the variable on top of the stack
  - When the subroutine terminates, all variables it declares are popped from stacks

---

Revisit the Example

```c
void sub3() {
    int x, z;
    x = u + v;
    ...
}
void sub2() {
    int w, x;
    ...
}
void sub1() {
    int v, w;
    ...
}
void main() {
    int v, w;
    ...
}
```

(The names in the stack cells indicate the program units of the variable declaration.)
Another way to implement shallow access

• Use a central table that has a location for each different variable name in a program
• Along with each entry, a bit called active is maintained that indicates whether it has a current binding or variable association
• Any access to any variable can then be to an offset into the central table
• The offset can be static, so the access can be fast

Central Table Maintenance

• When a subroutine is called, all of its local variables are logically placed in the central table
  – If the position of the new variable is already active, the original value must be saved somewhere
  – When a variable begins its lifetime, the corresponding active bit must be set
How to save values somewhere?

• Have a “hidden” stack on which all saved objects are stored
  – Since subroutines are called and then return, the lifetimes of local variables are nested, so this works

• All saved variables are stored in the activation record of the subroutine that created the replacement variable
  – The overhead is smaller because no extra stack is used