

Subroutines and Control Abstraction

8.2.1 Displays

EXAMPLE 8.63

Nonlocal access using a display

As noted in the main text, a display is an embedding of the static chain into an array. The j th element of the display contains a reference to the frame of the most recently active subroutine at lexical nesting level j . The first element of the display is thus a reference to the frame of some subroutine S nested directly inside the main program; the second element is a reference to the frame of a routine that is nested inside of S , and so forth, until we reach the currently active routine. Figure ©8.9 contains an example. ■

If the display is stored in memory, then a nonlocal object can be loaded into a register with two memory accesses: one to load the display element into a register, the second to load the object. On a machine with a large number of registers, one might be tempted to reduce the overhead to only one memory access by keeping the entire display in registers, but that would probably be a bad idea: display elements tend to be accessed much less frequently than other things (e.g., local variables) that might be kept in the registers instead.

Maintaining the Display

Maintenance of a display is slightly more complicated than maintenance of a static chain, but not by much. Perhaps the most obvious approach would be to maintain the static chain as usual, and simply fill the display at procedure entry and exit, by walking down the chain. In most cases, however, the following (much faster) scheme suffices: when calling a subroutine at lexical nesting level j , the callee saves the current value of the j th display element into the stack, and then replaces that element with a copy of its own (newly created) frame pointer. Before returning, it restores the old element. Why does this mechanism work? As with static chains, there are two cases to consider:

1. The callee is nested (directly) inside the caller. In this case the caller and the callee share all display elements up to the current level. Putting the callee's frame pointer into the display simply extends the current level by one. It is conceivable that the old value needn't be saved, but in general there is no way

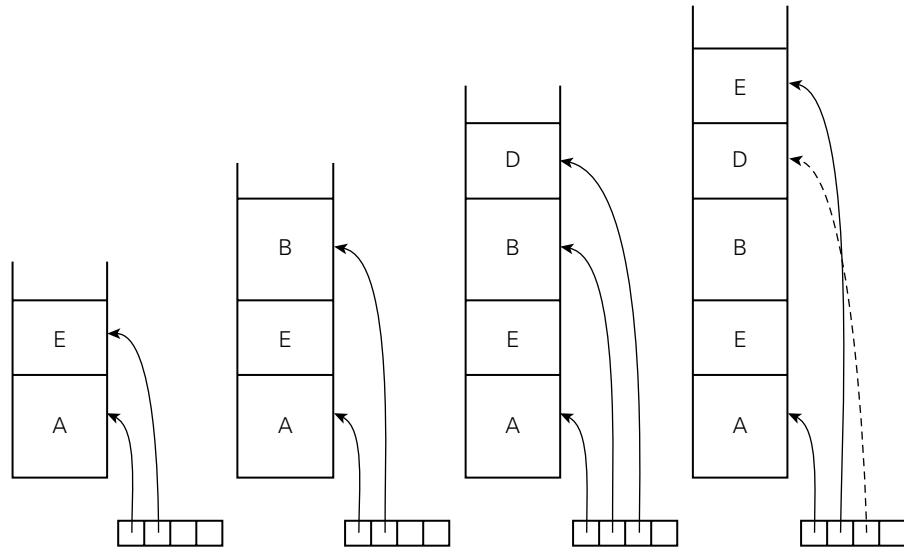


Figure 8.9 Nonlocal access using a display. The stack configurations, from left to right, illustrate the contents of the display (at bottom) for a sequence of subroutine calls, assuming the lexical nesting of Figure 8.1. Display elements beyond that of the currently executing subroutine are not used.

to tell. The caller itself might have been called by code that is very deeply nested, and that is counting on the integrity of a very deep display, in which case the old display element *will* be needed. A smart compiler may be able to avoid the save in certain circumstances.

2. The callee is at lexical nesting level j , $k \geq 0$ levels out from the caller. In this case the caller and callee share all display elements up through $j - 1$. The caller's entry at level j is different from the callee's, so the callee must save it before

DESIGN & IMPLEMENTATION

Lexical nesting and displays

Because the display is a fixed-size array, compilers that use a display to implement access to nonlocal objects generally impose a limit (the size of the display) on the maximum depth to which subroutines may be nested. If this limit is larger than, say, five or six, it is unlikely that any programmer will ever wish for more. Note that the display does not eliminate the need for a frame pointer. Because local variables are accessed so often, it is important to have the address of the current frame in a register, where it can be used for displacement-mode addressing. Similarly, on a RISC processor, where a 32-bit address will not fit in one instruction, it is important to maintain a base register for the most commonly accessed global variables as well.

storing its own frame pointer. If the callee in turn calls a routine at level $j + 1$, that routine will change another element of the display, but all old elements will be restored before they are needed again.

If the callee is a leaf routine then the display can be left intact; no one will use the element corresponding to the callee's nesting level before control returns to the caller.

Closures

A subroutine that is passed as a parameter, stored in a variable, or returned from a function must be called through some sort of *closure* (Section 3.6) that captures the referencing environment. In a language implementation based on static chains, a closure can be represented as a $\langle \text{code address, static link} \rangle$ pair. Displays are not as simple. A standard technique is to create two “entry points”—starting addresses—for every subroutine. One of these is for “normal” calls, the other for calls through closures. When a closure is created, it contains the address of the alternative entry point. The code at that entry point saves elements 1 through j of the display into the stack (it will have to create a larger-than-normal stack frame in order to do this), and then replaces those elements with values taken from (or calculated from) the closure. The alternative entry then makes a nested call to the main body of the subroutine (it skips the code immediately following the normal entry—the code that creates the normal stack frame and updates the display). When the subroutine returns, it comes back to the code of the alternative entry, which restores the old value of the display before returning to the actual caller.

More space-conserving implementations of display-based closures are possible (see Exercise ©8.35), but with higher run-time overhead.

Comparison to Static Chains

In general, maintaining a display is slightly more expensive than maintaining a static chain, though the comparison is not absolute. In the usual case, passing a static link to a called routine requires $k \geq 0$ load instructions in the caller, followed by one store instruction in the callee (to place the static link at the appropriate offset in the stack frame). The store may be skipped in leaf routines, assuming that a register is available to hold the link as long as it is needed. No overhead is required to maintain the static chain when returning from a subroutine. With a display, a nonleaf callee requires two loads and three stores ($1 + 2$ in the prologue and $1 + 1$ epilogue) to save and restore display elements. Because the callee does all the work, displays may save a little bit on code size, compared to static chains. As noted above, displays significantly complicate the creation and use of closures.

The original advantage of displays—reduced cost for access to objects in outer scopes—seems less clear today than once it did. In fact, while displays were popular in the CISC compilers of the 1970s and 1980s, they are less common in recent compilers. Most programs don't nest subroutines more than two or three levels deep, so static chains are seldom very long, and variables in surrounding scopes

tend not to be accessed very often. If they *are* accessed often, common subexpression optimizations (to be discussed in Chapter 16) are likely to ensure that a pointer to the appropriate frame remains in a register.

Some language designers have argued that the development of object-oriented programming (the subject of Chapter 9) has eliminated the need for nested subroutines [Han81]. Others might even say that the success of C has shown such routines to be unneeded. Without nested subroutines, of course, the choice between static chains and displays is moot.

✓ CHECK YOUR UNDERSTANDING

50. Describe how we access an object at lexical nesting level k in a language implementation based on displays.
 51. Why isn't the display typically kept in registers?
 52. Explain how to maintain the display during subroutine calls.
 53. What special concerns arise when creating closures in a language implementation that uses displays?
 54. Summarize the tradeoffs between displays and static chains. Describe a program for which displays will result in faster code. Describe another for which static chains will be faster.
-