Nested Subroutines

Nesting subroutines inside each other, introduced in Algol 60, is a feature of many languages, including Ada, ML, Common Lisp, Python, Scheme, Swift. Other languages including C and its decendants, allow classes or other scopes to nest. Any **constants**, **types**, **variables**, or **subroutines** declared within a scope are not visible outside that scope in Algol-family languages. More formally, Algol-style nesting gives rise to the *closest nested scope rule* for bindings from names to objects: a name that is introduced in a declaration is known in the scope in which it is declared, and in each internally nested scope, unless it is *hidden* by another declaration of the same name in one or more nested scopes. To find the object corresponding to a given use of a name, we look for a declaration with that name in the current, innermost scope. If there is one, it defines the active binding for its name. Otherwise, we look for a declaration in the immediately surrounding scope. We continue outward, examining successively surrounding scopes, until we reach the outer nesting level of the program, where global objects are declared. If no declaration is found at any level, then the program is in error. Though they are hidden from the rest of the program, nested subroutines are able to access the parameters and local variables (and other local objects) of the surrounding scope(s).

A name-to-object binding that is hidden by a nested declaration of the same name is said to have a *hole* in its scope. In some languages, the object whose name is hidden is simply inaccessible in the nested scope (unless it has more than one name). In others, the programmer can access the outer meaning of a name by applying a *qualifier* or *scope resolution operator*.

Access to Nonlocal Objects

A compiler can arrange for a frame pointer register to point to the frame of the currently executing subroutineat run time. Using this register for **displacement** (register plus offset) addressing, target code can access objects within the current subroutine. To find objects in lexically surrounding subroutines we need a way to find the frames corresponding to those scopes at run time. Since a nesting subroutine may call a routine in an outer scope, the order of stack frames at run time may not necessarily correspond to the order of lexical nesting. We can be sure there is some frame for the surrounding scope already in the stack, since the current subroutine could not have been visible unless the surrounding scope was active. (It is possible in some languages to save a reference to a nested subroutine, and then call it when the surrounding scope is no longer active.)

The simplest way in which to find the frames of surrounding scopes is to maintain a **static link** in each frame that points to the "parent" frame: the frame of the most recent invocation of the lexically surrounding subroutine. If a subroutine is declared at the outermost nesting level of the program, then its frame will have a null static link at run time. If a subroutine is nested k levels deep, then its frame's static link, and those of its parent, grandparent, and so on, will form a static chain of length k at run time. To find a variable or parameter declared j subroutine scopes outward, target code at run time can dereference the static chain j times, and then add the appropriate offset.