CSC 218 Foundation of Sequential Program
Lecture Note 4
(Block Structured Languages and Parameter Passing Mechanisms)

Block-structured languages

A class of high-level languages in which a program is made up of *blocks* – which may include *nested blocks* as components, such nesting being repeated to any depth. A block consists of a sequence of statements and/or blocks, preceded by declarations of variables. Variables declared at the head of a block are visible throughout the block and any nested blocks, unless a variable of the same name is declared at the head of an inner block. In this case the new declaration is effective throughout the inner block, and the outer declaration becomes effective again at the end of the inner block. Variables are said to have nested scopes.

The concept of block structure was introduced in the Algol family of languages, and block-structured languages are sometimes described as *Algol-like*. The concept of nested scopes implicit in block structure contrasts with Fortran, where variables are either local to a program unit (subroutine) or global to several program units if declared to be COMMON. Both of these contrast with Cobol, where all data items are visible throughout the entire program.

Properties of an identifier (and the object it represents) may be set at

• Compile-time: These are static properties as they do not change during execution. Examples include the type of a variable, the value of a constant, the initial value of a variable, or the body of a function.

• Run-time: These are dynamic properties. Examples include the value of a variable, the lifetime of a heap object, the value of a function's parameter, the number of times a while loop iterates, etc.

Example: In Fortran

• The scope of an identifier is the whole program or subprogram.

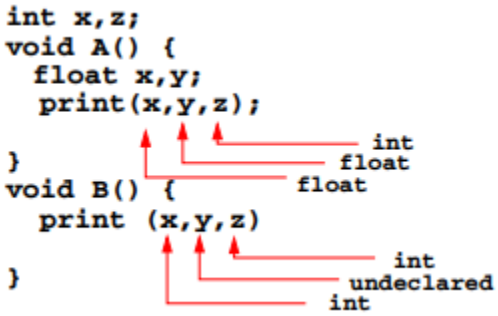
• Each identifier may be declared only once.

• Variable declarations may be implicit. (Using an identifier implicitly declares it as a variable.)

• The lifetime of data objects is the whole program.

Block Structured Languages include Algol 60, Pascal, C and Java.
• Identifiers may have a non-global scope. Declarations may be local to a class, subprogram or block.
• Scopes may nest, with declarations propagating to inner (contained) scopes.
• The lexically nearest declaration of an identifier is bound to uses of that identifier.

Binding of an identifier to its corresponding declaration is usually static (also called lexical), though dynamic binding is also possible. Static binding is done prior to execution—at compile-time. Example (drawn from C):



Parameter Passing Mechanism

In computer programming, a **parameter** or a *formal argument* is a special kind of variable used in a subroutine to refer to one of the pieces of data provided as input to the subroutine. These pieces of data are the valuesof the **arguments** (often called *actual arguments* or *actual parameters*) with which the subroutine is going to be called/invoked. An ordered list of parameters is usually included in the definition of a subroutine, so that, each time the subroutine is called, its arguments for that call are evaluated, and the resulting values can be assigned to the corresponding parameters.

Parameter passing allows the values of local variables within a main **program** to be accessed, updated and used within multiple sub-programs without the need to create or use global variables.

The mechanism used to pass parameters to a procedure (subroutine) or function. The most common methods are to pass the value of the actual parameter (*call by value*), or to pass the address of the memory location where the actual parameter is stored (*call by reference*). The latter method allows the procedure to change the value of the parameter, whereas the former method guarantees that the procedure will not change the value of the parameter. Other more complicated parameter-passing methods have been devised, notably *call by name* in Algol 60, where the actual parameter is re-evaluated each time it is required during execution of the procedure.

Parameter-passing techniques may be broken down as follows:

- Eager evaluation (applicative order) techniques. What these methods have in common is that the arguments passed in for a function's parameters are evaluated before the function is called.
    - Call-by-value
    - Call-by-reference
    - Call-by-copy-restore (also known as value-result or copy-in-copy-out)
- Lazy evaluation (normal order) techniques. What these methods have in common is that the evaluation of the arguments passed in for a function's parameters is delayed until the argument is actually used in the execution of the function.
    - Macro expansion
    - Call-by-name
    - Call-by-need

The difference between call-by-value and call-by-reference is exemplified by the difference between denoted values in our interpreters for SLang 1 and SLang 2. That is, in call-by-value, the argument for a function parameter is a copy of the value of the argument whereas, in call-by-reference, the function is given the address of the argument. Given the address, the function has the capability of modifying the argument.

Techniques used for argument passing:

- call by value
- call by result
- call by value-result
- call by reference
- call by name

(and call-by-constraint in constraint languages)

call by value: copy going into the procedure
call by result: copy going out of the procedure
call by value result: copy going in, and again going out
call by reference: pass a pointer to the actual parameter, and indirect through the pointer
call by name: re-evaluate the actual parameter on every use. For actual parameters that are simple variables, this is the same as call by reference. For actual parameters that are expressions, the expression is re-evaluated on each access. It should be a runtime error to assign into a formal parameter passed by name, if the actual parameter is an expression. Implementation: use anonymous function ("thunk") for call by name expressions