Chapter 5 continued

Variable Attributes (continued)

- Storage Bindings & Lifetime
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool
- The lifetime of a variable is the time during which it is bound to a particular memory cell

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables in functions
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)

Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are *elaborated*.

  (A declaration is elaborated when the executable code associated with it is executed)

- If scalar, all attributes except address are statically bound
  - local variables in C subprograms (not declared **`static`**) and Java methods
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Categories of Variables by Lifetimes

- *Explicit heap-dynamic* -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references, e.g. dynamic objects in C++ (via **`new`** and **`delete`**), all objects in Java
- Advantage: provides for dynamic storage management
- Disadvantage: inefficient and unreliable

Categories of Variables by Lifetimes

- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *local variables* of a program unit are those that are declared in that unit
- The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special category of nonlocal variables
- The scope rules of a language determine how references to names are associated with variables

Static Scope

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*
- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python)

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Example in C:

```
void sub() {
  int count;
  while (...) {
    int count;
    count++;
    ...
  }
  …
}
```

- Note: legal in C and C++, but not in Java and C# - too error-prone

The **`LET`** Construct

- Most functional languages include some form of **`let`** construct
- A let construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part
- In Scheme:

```
(LET (
   (name1 expression1)
   …
   (namen expressionn)
)
```

The **`LET`** Construct (continued)

- In ML:

```
let
   val name1 = expression1
   …
   val namen = expressionn
in
  expression
end;
```

- In F#:
  - First part: **`let`** left_side = expression
  - (left_side is either a name or a tuple pattern)
  - All that follows is the second part

Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In the official documentation of C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, that is misleading, because a variable still must be declared before it can be used

Declaration Order (continued)

- In C++, Java, and C#, variables can be declared in `for` statements
  - The scope of such variables is restricted to the `for` construct

Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions
- C and C++have both declarations (just attributes) and definitions (attributes and storage)
  - A declaration outside a function definition specifies that it is defined in another file

Global Scope (continued)

- PHP
  - Programs are embedded in HTML markup documents, in any number of fragments, some statements and some function definitions
  - The scope of a variable (implicitly) declared in a function is local to the function
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`

Global Scope (continued)

- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

Evaluation of Static Scoping

- Works well in many situations
- Problems:
  - In most cases, too much access is possible
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Scope Example

```
function big() {
    function sub1()
       var x = 7;
    function sub2() {
      var y = x;
    }
   var x = 3;
}
```

```
big calls sub1
sub1 calls sub2
sub2 uses x
```

- Static scoping
  - Reference to `x` in `sub2` is to `big`'s `x`
- Dynamic scoping
  - Reference to `x` in `sub2` is to `sub1`'s `x`

Scope Example

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically type check
    3. Poor readability- it is not possible to statically determine the type of a variable

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **`static`** variable in a C or C++ function

Referencing Environments

- The *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Named Constants

- A *named constant* is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - C++ and Java: expressions of any kind, dynamically bound
  - C# has two kinds, **`readonly`** and **`const`**
    - the values of **`const`** named constants are bound at compile time
    - The values of **`readonly`** named constants are dynamically bound

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors