Variable Attributes

- **Type Inferencing** (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined (by the compiler) from the context of the reference

- **Storage Bindings & Lifetime**
  - **Allocation**: getting a memory 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)

- **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
    - e.g, local variables in C subprograms and Java methods
  - **Advantage**: allows recursion; conserves storage
  - **Disadvantages**:
    - Overhead of allocation and de-allocation
    - 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`), and all objects in Java
  - **Advantage**: provides for dynamic storage management
  - **Disadvantage**: inefficient and unreliable

- **Implicit heap-dynamic**: Allocation and deallocation caused by assignment statements
  - e.g., 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

Type Checking

- **Type checking** is the activity of ensuring the operands of an operator are of compatible types
- A **compatible type** is one that is
  - either legal for the operator
  - or is allowed by coercion
- **Coercion** is automatic conversion of one type to a legal type (for a specific operand) under language rules
- **Type error** is the application of an operator to an operand of an illegal type

Variable Attributes: **Scope**

- The **scope** of a variable is the range of statements over which it is visible
- The **nonlocal variables** of a program unit are those that are visible but not declared there
- 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 (e.g., Ada, JavaScript, Fortran 2003, and PHP) allow nested subprogram definitions, which create **nested static scopes**

---

Scope (cont’d)

- Variables can be *hidden* from a unit by having a "closer" variable with the same name
- Some languages (e.g. Ada and C++) allow access to these "hidden" variables
  - E.g., `unit::name`

---

Blocks

- A method of creating static scopes inside program units—from **ALGOL 60**
- Example in C:
  ```c
  void sub() {
    int count;
    while (...) {
      int count;
      count++;
      ...
    }
    ...
  }
  ```

---

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 **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, a variable still must be declared before it can be used
Declaration Order (cont’d)

- In C++, Java, and C#, variables can be declared in for statements
  - the scope of such variables is restricted to the for construct
- In C++, Java, and C#,
  - the scope of a variable defined in a method starts at the definition
  - the scope of a variable defined in a class is the whole class

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

```plaintext
Big
  - declaration of X
    - declaration of X -
      ... call Sub2 ...
    ... Sub2 ...
      - reference to X -
        ... ...
        call Sub1 ...
...
Big calls Sub1
Sub1 calls Sub2
Sub2 uses X
```
**Scope Example**

- **Static scoping**
  - Reference to X is to Big’s X
- **Dynamic scoping**
  - Reference to X is to Sub1’s X
- **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**
  - the scope is limited to the enclosing function
  - the lifetime is not
- **Other examples?**

**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:**
  - **FORTRAN 95:** constant-valued expressions
  - **Ada, C++, and Java:** expressions of any kind
  - **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