Software II: Principles of Programming Languages

Lecture 5 – Names, Bindings, and Scopes

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables are characterized by *attributes*
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility
Names

• Design issues for names:
  – Are names case sensitive?
  – Are special words reserved words or keywords?

Names (continued)

• Length
  – If too short, they cannot be connotative
  – Language examples:
    • FORTRAN 95: maximum of 31 (only 6 in FORTRAN IV)
    • C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31 (only 8 are significant K&R C )
    • C#, Ada, and Java: no limit, and all are significant
    • C++: no limit, but implementers often impose one
Names (continued)

- Special characters
  - PHP: all variable names must begin with dollar signs
  - Perl: all variable names begin with special characters, which specify the variable’s type
  - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. IndexOutOfBoundsException)
Names (continued)

• Special words
  – An aid to readability; used to delimit or separate statement clauses
    • A keyword is a word that is special only in certain contexts, e.g., in Fortran
      – Real VarName (Real is a data type followed with a name, therefore Real is a keyword)
      – Real = 3.4 (Real is a variable)
  – A reserved word is a special word that cannot be used as a user-defined name
  – Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables

• A variable is an abstraction of a memory cell
• Variables can be characterized as 6 attributes:
  – Name
  – Address
  – Value
  – Type
  – Lifetime
  – Scope
Variables Attributes

- **Name** - not all variables have them
- **Address** - the memory address with which it is associated
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program

Aliases

- If two variable names can be used to access the same memory location, they are called *aliases*
- Aliases are created via pointers, reference variables, C and C++ unions
- Aliases are harmful to readability (program readers must remember all of them)
Value

- **Value** - the contents of the location with which the variable is associated
  - The *l-value* of a variable is its address
  - The *r-value* of a variable is its value

Type

- **Type** - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
The Concept of Binding

- A *binding* is an association between an entity and an attribute, such as between a variable and its type or value, or between an operation and a symbol.
- *Binding time* is the time at which a binding takes place.

Possible Binding Times

- **Language design time** - bind operator symbols to operations.
- **Language implementation time** - bind floating point type to a representation.
- **Compile time** - bind a variable to a type in C or Java.
- **Load time** - bind a C or C++ static variable to a memory cell.
- **Runtime** - bind a nonstatic local variable to a memory cell.
Static and Dynamic Binding

• A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
• A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Type Binding

• How is a type specified?
• When does the binding take place?
• If static, the type may be specified by either an explicit or an implicit declaration
Explicit/Implicit Declaration

- An explicit declaration is a program statement used for declaring the types of variables
- An implicit declaration is a default mechanism for specifying types of variables through default conventions, rather than declaration statements
- Fortran, BASIC, Perl, Ruby, JavaScript, and PHP provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Explicit/Implicit Declaration (continued)

- Some languages use type inferencing to determine types of variables (context)
  - C# - a variable can be declared with var and an initial value. The initial value sets the type
  - Visual BASIC 9.0+, ML, Haskell, F#, and Go use type inferencing. The context of the appearance of a variable determines its type
Type Inferencing – An Example

- ML does not require type declarations if the interpretation is unambiguous and can be inferred from other information.
- Example:
  \[
  \text{fun area(length:int, width:int):int = length * width}
  \]
  can also be written
  \[
  \text{fun area(length, width):int = length * width}
  \]
  \[
  \text{fun area(length:int, width): = length * width}
  \]
  \[
  \text{fun area(length, width:int): = length * width}
  \]
  but not
  \[
  \text{fun area(length, width) = length * width}
  \]

Dynamic Type Binding

- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited))
- Specified through an assignment statement e.g., JavaScript
  \[
  \text{list = [2, 4.33, 6, 8];}
  \]
  \[
  \text{list = 17.3;}
  \]
  – Advantage: flexibility (generic program units)
  – Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult
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

Stack-Dynamic Variables

• 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

Explicit Heap-Dynamic Variables

• 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
- Examples
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP

Implicit Heap-Dynamic Variables

- 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.

Variable Attributes: Scope (continued)

- *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

Static Scope (continued)

- 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)
Static Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name
- Ada allows 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++;
      ...
    }
    ...
  }
  ```

Legal in C and C++
Not legal in Java and C# because it’s too error-prone
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

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:
  
  ```scheme
  (let (  
    (name_1 expression_1)  
    ...  
    (name_n expression_n)  
  )
  )
  ```
The **LET** Construct (continued)

- In ML:
  ```ml
  let
      val name_1 = expression_1
      ...
      val name_n = expression_n
  in
      expression
  end;
  ```

The **LET** Construct (continued)

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

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

Named Constants (continued)

- **Languages**:
  - Ada, 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