Names, Bindings, and Scopes

CSIT 313

Spring 2015

Context and Names

- Variables in imperative languages are essentially names for memory locations
  - Functional languages (Lisp, Scheme, etc.) also allow names to be identified with values of expressions
- Names
  - Case sensitive??
  - Are special words keywords or reserved?

Variables

- Best to think in terms of six attributes: (1) name, (2) address, (3) value, (4) type, (5) lifetime, (6) scope
- Name: most variables have names
- Address : address of memory cell
  - l- value – variables on left side of assignment statement
  - Aliasing: pointers & references, union types, subprogram parameters
- Type: determines range of values set of operations
- Value: contents of memory cell
  - Abstract memory cells
  - r-values – variables on right side of assignment

Binding

- A **binding** is an association between and attribute and an entity
  - Between a variable and its type or its value
  - Between and operation and a symbol
- Binding times
  - Example of different binding time: Java statement **count = count + 5** (p. 210)
  - Binding of actual parameters to formal parameters
  - Binding time of variable to storage location may affect current value of a variable
- **Static** and **dynamic** bindings

Static Type Bindings

- Explicit declaration
- Implicit declaration
  - **Implicit none** statement in Fortran
- Type inference
  - Example from C#:

    var alpha = 0;   // type of alpha is **int**
    var beta = 0.0;   // type of beta is **float**
    var gamma = “Put on a happy face”;
              // type of gamma is **string**

Dynamic Type Bindings

- Allows for greater flexibility
  - Possible to write a program “generically” so that it can process **any** type of input
  - Most languages prior to 1990 used static type binding
- Python, Ruby, JavaScript, and PHP use dynamic binding
- The keyword **dynamic** in C# after 2010
- Disadvantages of dynamic typing
  - Reduced compile-time error checking capability leads to lower reliability
  - Cost: type checking must be done at run time

Storage Bindings and Lifetime

- Allocation & deallocation
  - Lifetime of a variable
- Static variables
- Stack-dynamic variables
- Explicit heap-dynamic variables
- Implicit heap-dynamic variables

Static Variables

- Bound to memory cells before program execution
  - Globally accessible variables
  - Static local variables in subprograms
- Advantage: Efficiency
  - Direct memory access possible
  - No run-time allocation or deallocation
- Disadvantages
  - Reduced flexibility – recursive subprograms not possible with only static variables
  - Cannot share storage among variables

Stack-Dynamic Variables

- Storage bindings created when variable’s declaration statement is **elaborated.** Variable’s type is statically bound
  - **Elaboration** is the storage allocation and address binding that occurs when execution reaches code to which declaration is attached – *at run time!*
  - Allocated space on ***run-time stack***
- Advantages
- Disadvantages

Explicit Heap-Dynamic Variables

- Nameless memory cells that are allocated and deallocated using explicit (run-time) instructions
  - The **heap** is a collection of memory cells whose use and organization are not predictable
  - Pointer or reference variables
- C++ pointers: **new** and **delete** operators
- Java references: All objects explicitly heap dynamic – created using **new** operator
  - Implicit deallocation using *garbage collection*

Explicit Heap-Dynamic Variables (2)

- **Advantages:**
  - Allow construction of dynamic structures like linked lists and trees
- **Disadvantages:**
  - Difficulty of using pointer and reference variables correctly
  - Cost of reference variables
  - Complexity of required storage management

Implicit Heap-Dynamic Variables

- Bound to heap storage only when assigned values
  - Example: JavaScript statement
    testScores = [92,87,65,72,96];
  - **testScores** is now an array of five numbers – whatever it was before
  - Memory in heap allocated for array on assignment
- Advantage: flexibility allows one to write highly generic code
- Disadvantages
  - Run time overhead of maintaining all dynamic attributes
  - Loss of compile-time error detection

Scope

- The **scope** of a variable is the range of statements in which the variable is **visible** (i.e., can be used in statement)
- **Static scoping** (lexical scoping)
  - Scope can be determined at compile time
  - Effect of nested subprograms: static *parent* and *ancestors*
  - Effect of nested blocks (block-structured languages)

Static Scoping with Nested Subprograms

- Example from JavaScript
```
function outer() {
   function sub1() { var x = 7; sub2(); }
   function sub2() {var y = x; z = y; }
   var x = 3; var z = 0;
  sub1();
} // outer
```
- Hidden variables
  - Ada allows access to hidden variables from ancestor scope to be made visible with selective references (e.g. **outer.x** within **sub1**).
- **Class Exercise:** What are the values of x and z on return from outer?

Another Static Scoping Example

```
var x,y,z;
procedure sub1() {
  var x;
  function sub2() {
    var y;
    x = 20; y = 30; z = 40; // program point 1
  }
  x = 8; y = 12; z = 16; sub2();  //program point 2
}
function sub3() {
  var z;
  x = 5; y = 10; z = 16; sub1(); // program point 3
}
x = 1; y = 2; z = 3;  sub3();  // program point 4
//CLASS EXERCISE: What are the values of x, y, and z
at this point?
```

Static Scoping and Blocks

- In the C family of languages, a **block** is a list of statements enclosed in curly braces.
  - A variable declared in a block has that block as its scope.
  - Example in C:

```
void subpgm() {
   int count = 0; … // other statements
  while(…) {
    int count = 0; count++; ….
    // Which count is incremented?
  } // while block
  …
} //subpgm
```

Functional Languages: LET statements

- In most functional languages, a **let** statement binds the **value** of an expression to a **name** that can be used in another expression within scope
- **Example** (Scheme):

```
( LET  (
    ( horiz ( - x2 x1) )
    ( vert ( - y2 y1) )
    ( horiz + vert)
)
```

Declaration Order

- Declaration position and scope
  - In C99, C++, Java, scope goes from variable declaration within block of subpogram to end of block or subprogram
  - In C#, the scope is the entire block. However variables must be declared before they are used
- **for** loops in Java, C#, and C++
  - Early versions of C++: scope goes to end of smallest enclosing block
  - C#, Java, later versions of C++: scope confined to **for** construct

Global Scope

- In some languages, program structure is a sequence of function definitions, but variable definitions can occur outside the functions.
- Such variables are available in any function -- at least after they're defined – so they are called **global variables.**

Global Scope (2)

- Global variables in C and C++
- A **declaration** binds a variable to a type and other attributes but does not allocate storage
- A **definition** allocates storage
- Implicitly visible in all functions in file that occur after declaration
- May be defined in a different file
- The **extern** qualifier on a variable declaration within a function
- The scope operator (::) for hidden global variables

Global Scope (3)

PHP

- Statements may be interspersed with functions
- Any variable is implicitly defined with its use in a statement
- Variables defined outside functions are global – visible in rest of program, *except within functions*
- Global variables can be made visible in functions
  - $GLOBALS array
  - **global** declaration statement

Global Scope (4)

Python

- Variables implicitly defined when used as targets in assignment statements
- Global variables may be *referenced* within a function (e.g., used on right side of assignment)
- Global variables may be assigned to (used on left side) only if declared to be **global** within the function
- Variables used in nested function must be declared **nonlocal**

Evaluation of Static Scoping

- Works well in many situations
- Allows more access than usually needed to variables and subprograms
- Program evolution and restructuring
  - Programmers may discard original structure if it gets in the way
  - Use more global variables than necessary
- Alternative: Control access to variables using an **encapsulation** construct

Dynamic Scoping

- Based on calling sequence of subprograms
  - Scope can only be determined at run time
  - Determining meaning of a variable using dynamic scoping
- Disadvantages
  - Local variables of subprogram is visible to all subprograms that are executing at the same time (i.e., to all descendants in the call tree)
  - Impossible to type check references to non-locals
  - Decreased readability
- Advantages
  - Variables of caller implicitly visible in called subprogram, without needing to be explicitly passed in as parameters

Scope, Lifetime, and Referencing Environment

- **Lifetime** of a variable: period when it is bound to a memory location
- **Scope** of a variable: portion of program in which it is visible (i.e., can be used)
  - Note that these are not always the same
- The **referencing environment** of a statement is the collection of variables that are visible in that statement

Named Constants

- Use in programs
  - Improving readability
  - Making program modification easier
- Implementation
  - Keyword **final** in Java
  - Keyword **const** in C or C++
  - Keywords **const** and **readonly** in C#
- Initialization – binding a variable to a value at time it is bound to storage