Lecture 3: Names, Bindings, and Scope

Chapter 5

Lecture 3 Topics

- Part A: Names and Variables
  - Introduction
  - Names
  - Variables
  - The Concept of Binding
- Part B: Scope and Lifetime
  - Scope
  - Scope and Lifetime
  - Referencing Environments
  - Named Constants

Part A: Names and Variables

- Introduction
  - Imperative languages are abstractions of von Neumann architecture, also called stored program concept
    - 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:
  - Should names be case sensitive?
    - Advantage and disadvantage
  - Are special words reserved words or keywords?
  - Should there be a limit on the length?
    - Language Examples
      - C# and Java: no limit, and all are significant
      - C++: no limit, but implementers often impose one
    - Problem with too short names?
  - Should special characters be used?
    - PHP: all variable names must begin with dollar signs
    - Ruby, Perl??
  - Any additional design issues?

Do you prefer case sensitive or case insensitive identifiers?

- Answer A: Case sensitive
- Answer B: Case insensitive
- Answer C: No preference
- Answer D: None of above

What do you call some special words in Java (e.g. class, int, public, …)

- Answer A: Keywords
- Answer B: Reserved words
- Answer C: Reserved keywords
- Answer D: None of above

(A–C are all acceptable answers, explanation next)

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,

  i.e. a word could be used for special meaning or a regular name, e.g. in early FORTRAN

  IF IF .EQ. THEN THEN THEN=IF
- A *reserved word* is a special word that cannot be used as a user-defined name
  - Popularly used in modern language
  - New term (e.g. Java), *reserved keyword*, simply *keyword*
  - How many reserved keywords in Java (or C++, etc.)?
    - What if too many reserved words in a language?
    - e.g., COBOL has 300 reserved words!

Should special characters used in variables? For example, PHP variables starts with $ as in the example below

```
$txt = "Hello world!";
$x = 5;
$y = 10.5;
```

Answer A: Yes, I like it;
Answer B: No, I don’t like it;
Answer C: Oh, I don’t care
Answer D: Other

Variables

- A variable is an abstraction of a memory cell
  - Architectural influence
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime (part B)
  - Scope (part B)

Variables Attributes

- Name – not all variables have them
  - Identifiers rule in Java (C++, etc.) ?
  - Give an example of anonymous variable?
- Address – the memory address with which it is associated
  - A variable may have different addresses at different times during execution (example?)
  - A variable may have different addresses at different places in a program (example?)
  - If two variable names can be used to access the same memory location, they are called aliases
    - Aliases are created via pointers, reference variables, etc.
    - Aliases are harmful to readability
      - To be discussed in Lecture 6 Data Types

Variables Attributes (continued)

- Type – determines the range of values of variables and the set of operations that are defined for values of that type
  - Chapter 6
- 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
- *Abstract memory cell*
  - A memory location or a collection of locations associated with a variable
  - Abstraction of physical memory

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.
  - Language design time
    - e.g. bind operator symbols to operations
  - Language implementation time
    - e.g. bind floating point type to a representation
  - Compile time
    - e.g. bind a variable to a type in C or Java
  - Load time
    - e.g. bind a C or C++ `static` variable to a memory cell
  - Runtime
    - e.g. bind a non-static local variable to a memory cell

- What is the binding time for each of the following?
  - In C++, only the first 12 characters of an identifier are significant
  - In Python, ** refers to the exponent operation
  - In Java, String s1 = new String ("hi"); s1 String reference type
  - In Java, String s1 = new String ("hi"); s1 reference to a specific memory location storing "hi"

Static and Dynamic Binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
  - e.g. static variables
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program
  - e.g. a local variable defined inside a function

Type Binding

- Design Issues
  - How is a type (for a variable) specified?
    - Should we mandate *type declaration* in languages?
  - When does the binding take place?
    - Static vs. dynamic
  - If static, the type may be specified by either an explicit or an implicit declaration
    - What is an implicit type declaration?

Explicit/Implicit Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables.

```
int sum;          //C++
```

- An *implicit declaration* is a default mechanism for specifying types of variables through default conventions, rather than declaration statements.

```
sum = 30       #Python
```

  - Basic, Perl, Ruby, JavaScript, and PHP
- Pros & Cons? Readability vs. Writability?
- Some languages use type inferencing to determine types of variables (context).

```
var sum = 30 //C#
```

  - Visual Basic 9.0+, ML, Haskell, and F#

Dynamic Type Binding

- How to perform type checking with respect to implicit type declaration?
- Dynamic Type Binding
  - JavaScript, Python, Ruby, PHP, and C# (limited)
  - Specified through an assignment statement

```
list = [2, 4.33, 6, 8];        //JavaScript
list = 17.3;
```

  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult

Summary – Part A

- 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
  - Static binding and dynamic binding
  - Binding types to variables

Part B: Lifetime and Scope

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

      (Memory organization: a quick review)

- The *scope* of a variable is the range of statements over which it is visible

- Lifetime is the *dynamic* feature of a variable while scope describe the *static* span of a variable.

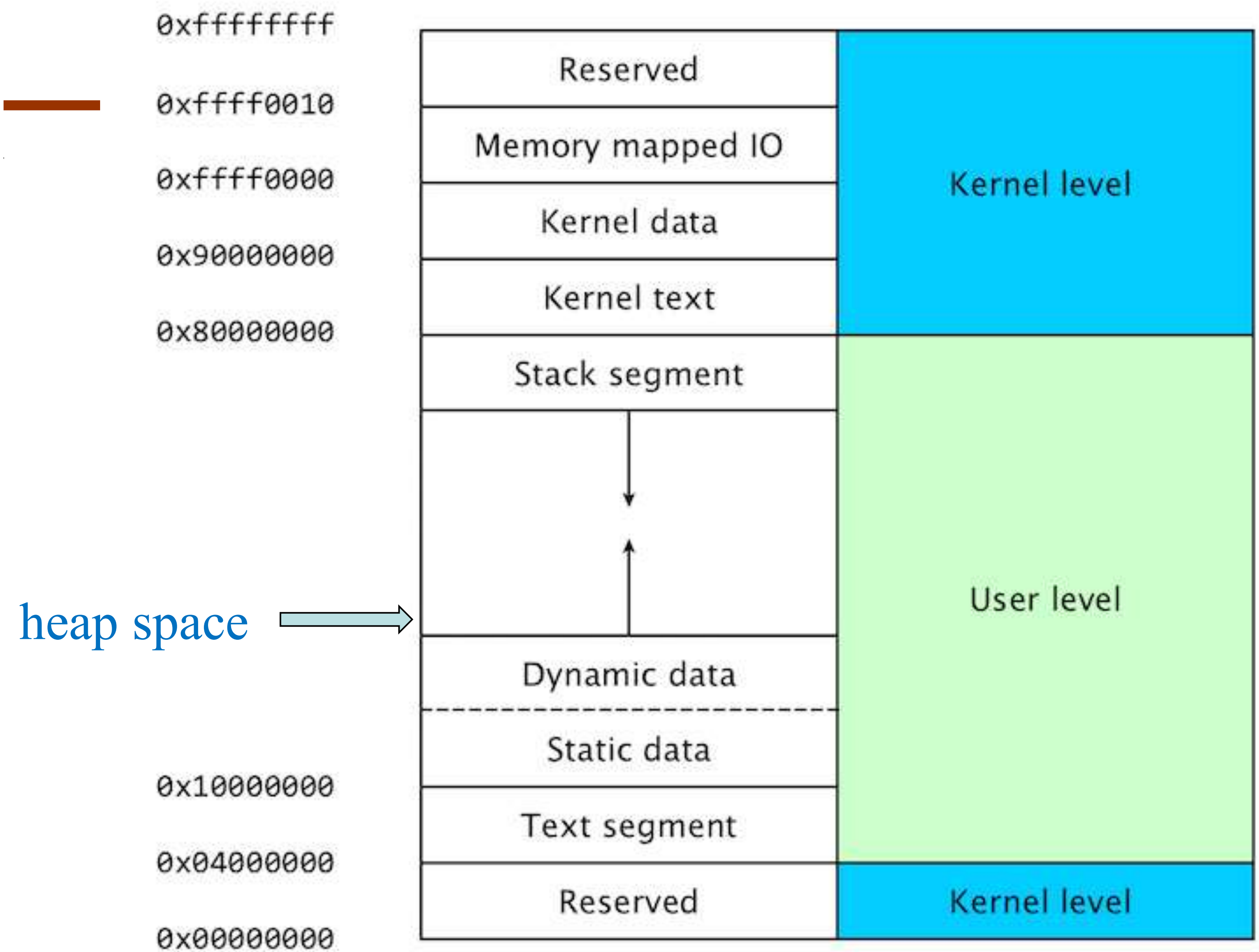
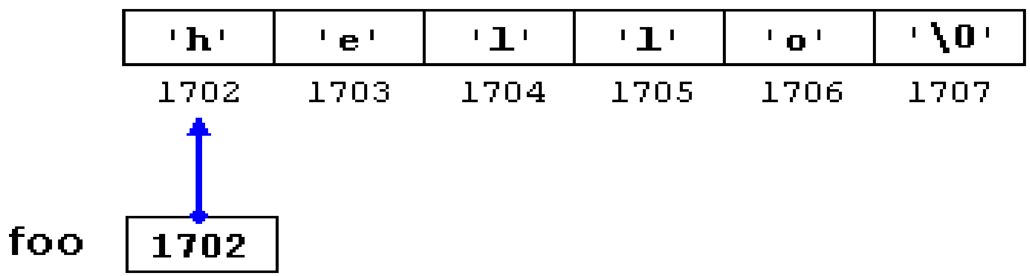


Figure source: it.uu.se

- References to objects

  String foo = new String (“hello”)



Variable 1: foo (name), string ref (type), 0100 (address)
Variable 2: anonymous, string (type), 1702 (address)

C++ or Java-like object; illustration only; details may vary.

	Memory
	…
0100	1702
	…
1702	'h’
1703	‘e’
	…

descriptor
…
(“foo”, String ref, addr=0100)
…

- References to objects

```
String foo;
foo = new String ("hello");
...
foo = new String ("bye");  //p1
        //any lifetime change??
        //if yes, which variable's lifetime got changed?
```

Categories of Variables by Lifetimes

- Four categories
  - static
  - stack-dynamic
  - explicit heap dynamic
  - implicit heap dynamic
- *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

    static int count = 0;
  - 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
    - e.g. local variables (not declared **`static`**) in C subprograms and Java methods
  - Advantage: allows recursion; conserves storage
  - Disadvantages: overhead of allocation and deallocation
- *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`**), all objects in Java
  - Advantage: provides for dynamic storage management
  - Disadvantage: inefficient and unreliable

Examples

- Stack dynamic

```
void f(int x) { double sum; …}
```

- Explicit heap dynamic

```
Square s = new Square (); //Java-like
Square *s = new Square(); //C++ like
```

The heap-dynamic variable/object is referenced via a pointer or a reference (i.e. s)

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
  - Advantage: flexibility (generic code)
  - Disadvantages:
    - Inefficient, because all attributes are dynamic
    - Loss of error detection

```
//JavaScript
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Lemon");    // adds a new element (Lemon) to fruits
```

Example -- identify variables' lifetime

```
class VariableDemo
{
   static int count=0;
   public void increment()
   {
      count++;
   }
   public static void main(String args[])
   {
      VariableDemo obj1=new VariableDemo();
      VariableDemo obj2=new VariableDemo();
      obj1.increment();
      obj2.increment();
      System.out.println("Obj1: count is="+obj1.count);
      System.out.println("Obj2: count is="+obj2.count);
   }
}
```

Discussion Question: what category a Java ArrayList belong to?

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
  - Scope and lifetime are sometimes closely related, but are different concepts, e.g. a static variable in C++
- 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

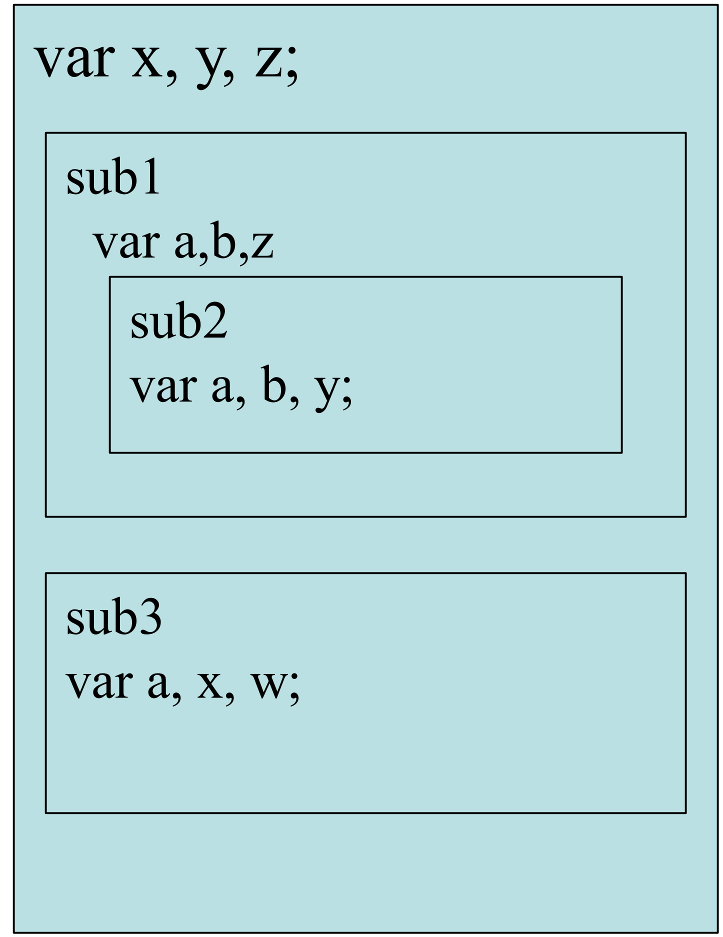
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name

- Some languages allow nested subprogram definitions, which create nested static scopes
  - Sample languages: Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python

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

Example: JavaScript Code

```
var x, y, z;          //line 1
function sub1() {
    var a, b, z;
    function sub2() {
        var a, b, y;
        . . .
        document.write(a,b,…);  //p1
    }  //end of sub2
    . . .
    document.write(a,b,…);  //p2
} //end sub1
function sub3() {
    var a, x, w;
    . . .
    document.write(a,b,…);  //p3
} //end of sub3
```



Scope: Java/C++ vs. JavaScript

Global or extern variables

f()

g()

main ()

var x, y, z;

sub1

var a,b,z

sub2

var a, b, y;

sub3

var a, x, w;

Example: JavaScript Code

```
var x, y, z;          //line 1
function sub1() {
    var a, b, z;
    function sub2() {
        var a, b, y;
        . . .
        document.write(a,b,...);  //p1
    }  //end of sub2
    . . .
    document.write(a,b,...);  //p2
} //end sub1
function sub3() {
    var a, x, w;
    . . .
    document.write(a,b,...);  //p3
} //end of sub3
```

Q1: What is the scope of y defined in line 1?
A: From line 1 to the end of the code except sub2.

Q2: At points p1, p2, p3 respectively, are *variables (a, b, w, x, y, z) visible* ? If multiple definition indicate which specific one.

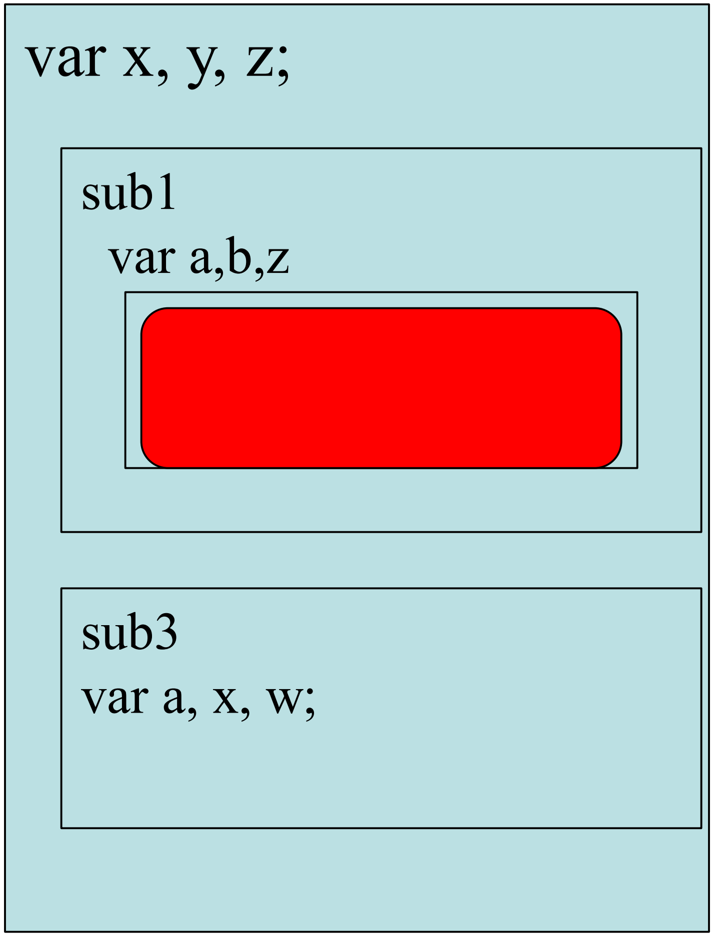
P1: a (sub2), b(sub2),
    w(not visible), x(global),
    y(sub2), z(sub1)

P2: ??

P3: ??

Example: JavaScript Code

```
var x, y, z;          //line 1
function sub1() {
    var a, b, z;
    function sub2() {
        var a, b, y;
        . . .
        document.write(a,b,...);  //p1
    }  //end of sub2
    . . .
    document.write(a,b,...);  //p2
} //end sub1
function sub3() {
    var a, x, w;
    . . .
    document.write(a,b,...);  //p3
} //end of sub3
```



Scope of y in line 1: from line 1 to the end of code except inside sub2

Blocks

- A method of creating static scopes inside program units--from ALGOL 60

Example in C:

```
void sub() {
    int count;  //outer block
    while (...) {
      int count;   //inner block
      count++;
      ...
    }
    …
  }
```

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

Declaration Order

- C++, Java, C# etc. allow variable declarations to appear anywhere a statement can appear
  - In 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
- In C++, Java, and C#, variables can be declared in for statements
  - The scope of such variables is restricted to the for construct

```
for (int j=0; j<100; j++) { sum += …;  }
System.out.println(j);
```

Examples

```
//C++
double totalCost (double  items[], int count ) {               //line1
        double margin = 20;                                     //line2
        double total = 0;                                       //line3
        for (int ct = 0; ct < count; ct++) {                    //line4
                double temp = items[ct] + margin;               //line5
                total += temp;                                  //line6
        }                                                       //line7
        double taxRate = 0.095;                                 //line8
        total = total * (1+taxRate);                            //line9
        return total;                                           //line10
}                                                               //line11
```

Q: Identify the scope of each variable defined above?

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
- PHP: Global variables can be accessed in a function through the *$GLOBALS* array or by declaring it *global*
- 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

Dynamic Scope

- Dynamic scope is 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

Example: Static vs. Dynamic Scoping

```
function big() {
    var x;
    function sub1() {
        var x = 7;
        sub2();    //sub1 calls sub2
                //assume call before function definition allowed
    } //end sub1

    function sub2() {
        print(x);        //uses x
    } //end sub2()

    x = 3;
    sub1();        //big calls sub1
} //end big
```

big calls sub1
sub1 calls sub2
sub2 uses x

- Static scoping
  - Reference to x in sub2 is to big's x (x=3)
- Dynamic scoping
  - Reference to x in sub2 is to sub1's x (x=7)

Evaluation: Static vs. Dynamic Scoping

- Static scoping
  - Works well in nested (subprogram) structure
  - However, as programming languages evolve, more popular in using C++/Java like “parallel” structures
- Dynamic Scoping:
  - Flexible (and history sensitive – able to access most recent value of a variable.)
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically determine the type of a variable and type check
    3. Poor readability

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
- 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

```
#define KILLBONUS 5000  //C
const int KILL_BONUS = 5000; //C++
public final int KILL_BONUS = 5000; //Java
```

Summary – Part B

- Scalar variables are categorized, per lifetime, as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Language designers/implementers may choose to implement static scoping and dynamic scoping
- Global variables and named constants are commonly used in programming languages, however binding and scope may vary.