3. VARIABLES, BINDINGS AND SCOPES

T: Chs. 5, 10.3

R2: 10.7, 10.8

Outline

- Introduction
- Names
- Variables
- Pointers and References
- The Concept of Binding
- Scope
- Referencing Environments
- Named Constants

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor

- Identifiers: Used to name things as subprograms, variables, classes, constants, etc.
  - Defined as names that start by a letter or underscore, and followed by any sequence of zero or more letters, digits or underscores.

Names

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C# and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one
- Names in the C-based language are case sensitive. Other languages are not (e.g., Fortran, Ada, PHP).
- Special words
  - An aid to readability; used to delimit or separate statement clauses
  - A *reserved word* is a special word that cannot be used as a user-defined name.
  - A *keyword* is a word that is special only in certain contexts.

Variables

- A variable is an abstraction of a memory cell.
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Variables Attributes (continued)

- *Name: Not every variable has a name*
  - Dynamic variables do not have specified names by the programmer.
- *Address:* the memory address with which it is associated
  - If two variable names can be used to access the same memory location, they are called **aliases**.
  - Aliases are created via pointers, reference variables, and C and C++ union
  - L-value: The l-value of a variable is its address
    - use of a variable in the case where you care about it's address
    - Appears on the Left side of the assignment operator
    - Is an address that is used to store a value Ex: x = …

Variables Attributes (continued): Addresses

- Memory for a running program is divided into several regions or segments.
- The operating system may control permissions on memory.
  - The runtime system is responsible for managing memory.
- Possible Regions
  - Code
  - Data
  - Heap
  - Stack
- *Abstract memory cell* - the physical cell or collection of cells associated with a non-structured (scalar) type variable

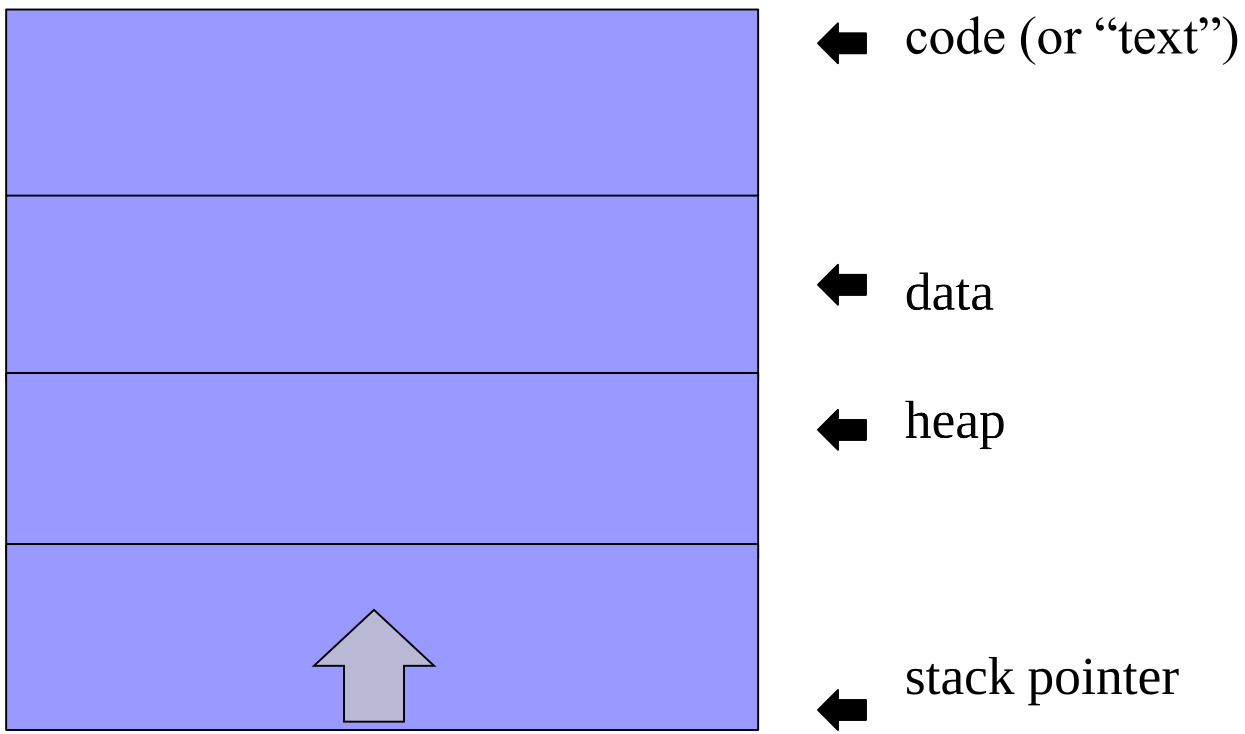
Memory Allocation to variables

- A variable needs enough memory to hold an instance of an object of that type (in other words, the type dictates how much memory is needed)
- `int x;`
  - declares that x is an integer; enough memory is allocated to hold an integer
- `Obj y;`
  - In C++ this means that y is an object of the class Obj; and enough memory is allocated to hold the y.
    - A constructor is called if one is provided to perform initialization of the object data members.
  - In Java this means that y is a reference to an object of the class Obj; and enough memory is allocated to hold a REFERENCE to that object.
    - By definition Java initializes the reference to null.
    - The reference is not the object!

The von Neumann Architecture

- **Memory Usage**
  - Programs use memory in several ways:
    - the actual executable machine instructions
    - global variables
    - constants (such as numeric numbers or strings)
    - memory that is used to support function calls
      - this is usually done in a “stack”, at runtime
    - memory that is dynamically allocated and freed as needed
      - this is usually done in a “heap”, at runtime
      - the “new” operator is one way to dynamically allocate memory

Memory Layout



Memory Allocation to variables in C++

- Where, in memory, are the variables?
  - Variables declared inside of a function, and variables for function arguments, have memory that is allocated on the stack.
  - Global variables are placed in the data segment.
  - Dynamically allocated memory is on the heap, and is assigned using “new”
    - A *heap* is a region of storage in which subblocks can be allocated and deallocated. The memory management systems keeps a record of Used and Free blocks.

Variables Attributes (continued)

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

Variables Attributes (continued):

- *Value* - the contents of the location with which the variable is associated
  - L-value/r-value of a variable refer to the meaning of a variable in the program depending on where it has been used.
  - R-value: The r-value of a variable is its value
    - use of a variable in the case where you care about the value stored in the variable
    - Appears on the Right side of the equals sign is the value stored in the variable
    - Ex: … = … x …

Pointers

- A pointer is a variable that contains a memory address.
- Pointers must be explicitly declared
  - In C/C++:
    - `int *ip; //declares ip as a pointer to an int`
    - `Obj *op; //declares op as a pointer to an Obj`
- Pointers need to be initialized: they must "point to" something:
  - Dereferencing (using the * operator on) a pointer that has not been initialized is an error.
  - For example,

```
int x;
int *xp;
*xp = 100;  // what does xp point to??
```

How is Memory used?

```
int outside; // external variable
            // put in a data segment by compiler/linker
               // exists for lifetime of the program

// the code for the function is in a code segment
void myFunction() {
// these two variables are on the stack
// and are put there at runtime by the
// call to the function
// exists for lifetime of the function call
SomeClass x;
SomeClass *p;

// the memory returned by new comes from the heap
// exists until cleaned up/freed (by user or by runtime
// environment)
p = new SomeClass();
}
```

The Concept of Binding

- Definition: Binding is the association between an attribute and an entity.
  - Binding time is the time at which a binding takes place.
- Binding of Attributes to Variables
  - Static binding: starts and remains throughout the program execution.
  - Dynamic binding: occurs during run time and may change during program execution.

The Concept of Binding

- Type Bindings
  - Static Type Binding
    - **Explicit** type declarations in C/C++:

```
int val;
double num;
int x[10]; //an array of 10 integers are allocated to x
```

    - **Implicit** type declarations using a default mechanism for specifying types of variables through default conventions, rather than declaration statements. Basic, Perl, Ruby, JavaScript, and PHP provide implicit declarations.
      - Naming conventions: Use of $ for scalars, and @ for arrays in Perl
      - inferencing deduced from the value of its initializer
        - C# use of var
        - C++ use of auto

```
int val = 5;
auto aval = val;//aval is int holding a copy of val
```

The Concept of Binding

- Dynamic Type Binding
  - JavaScript, Python, Ruby, PHP, and C# (limited))
  - Specified through an assignment statement, e.g., JavaScript

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

  - For example, a program that processes numeric data can be written as a generic program, dealing with data of any numeric type.
  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult

The Concept of Binding

- 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
  - *Static*--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, as in C and C++ `static` variables in functions

    ```
    static int xvar;
    ```

  - *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
    - Local variables and function parameters

The Concept of Binding

- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive

- *Explicit heap-dynamic* -- Allocated and deallocated by explicit directives as in C++, 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

The Concept of Binding

- Use case:

```
int *intptr ; //create a pointer variable
intptr = new int;//create a heap-dynamic variable.
. . . .
delete intptr;//deallocate heap-dynamic variable to
                  //which intptr points
```

- *Implicit-heap-dynamic* -- Bound to heap storage only when they are assigned values.
  - Examples: JavaScript, and PHP, and all strings and arrays in Perl
    - For example Javascript assignment

      high = {74, 84, 86, 90, 71};
  - Advantage: high degree of flexibility (generic code)
  - Disadvantages:
    - Inefficient run-time overhead, because all attributes are dynamic.

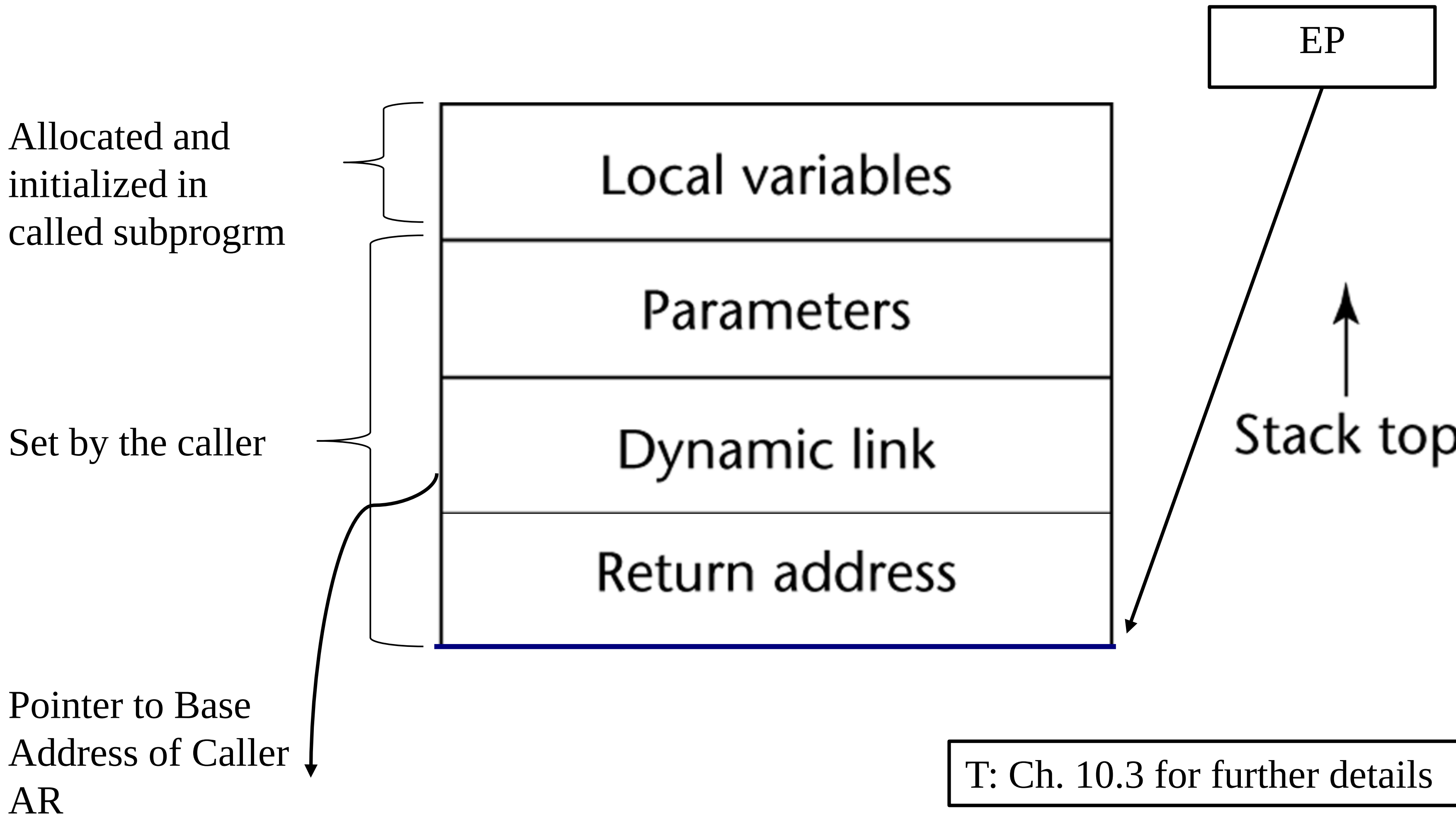
Categories of Variables by Lifetimes

```
int outside;    // external variable
                // exists for lifetime of the program
void myFunction(int var, int &refVar, int *ptr) {
  //var parameter is passed by-value
  //refVar is a reference variable.
  //Ptr pointer parameter passed by ref
  static int counter=0;//exists for the lifetime of the program
  // exists for lifetime of the function call
  int val; //stack-dynamic
  SomeClass x;
  SomeClass *p;


  Val = var + refVar +*ptr;

  // the memory returned by new comes from the heap
  // exists until cleaned up/freed (by user or by runtime
  // environment)
  p = new SomeClass(); //heap-dynamic
}
```

Typical Activation Record for a Language with Stack-Dynamic Local Variables



T: Ch. 10.3 for further details

An Example: C++ Function

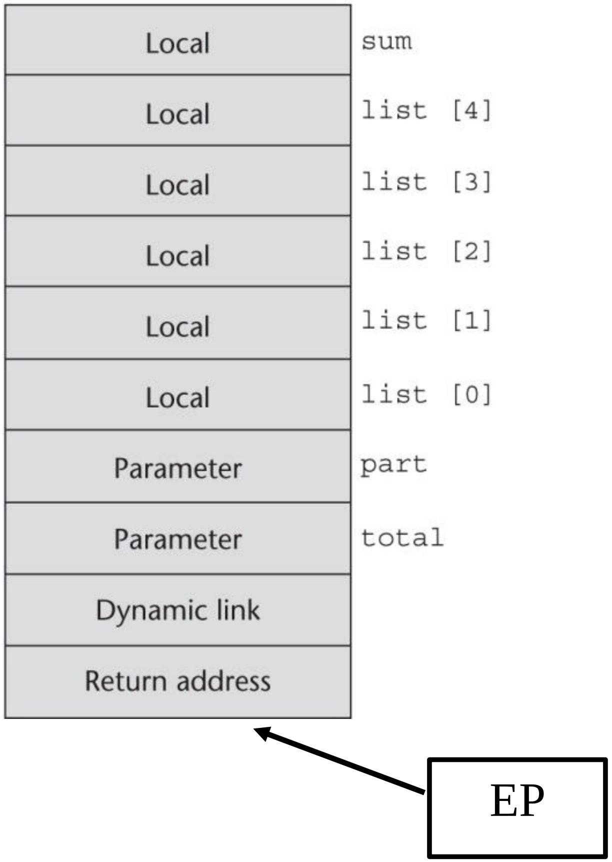
```
void sub(float total, int part)
{
   int list[5];
   float sum;
   …
   return;
}
//call from main() function
Sub(256.5, 1234);
//assume address of next inst. is 100
```

Return address = 100

Offset address of total parameter = 2

Actual address of total = EP + 2

EP is the base address of the AR in the run-time stack



Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible.
  - A variable is visible in a statement if it can be referenced or assigned in that statement.
  - 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.

Scope

- What constitutes a scope?
  - Global
    - C and C++ have global functions and data
    - In Java everything has to be in a class; there's no global functions or global data
  - File (compilation unit)
  - Certain subdivisions of the code (e.g., class)
- Possible Scopes
  - C++ Namespaces
  - Java Packages
  - Classes (which may be nested in other classes)
  - Functions
  - Blocks
  - For loops

Scope

- Static Scope
  - There are two categories of static-scoped languages:
    - Subprograms that can be *nested*
      - A reference to a name is *non local* if it occurs in a nested scope of the defining scope; otherwise, it is *local*.
      - *Examples: Javascript, Python, and Scheme*
    - Subprograms that cannot be nested (i.e., disjoint).
      - In disjoint scopes, same name can be bound to different entities without interference.
      - *Nested scopes are created only by nested class definitions and blocks*
      - *Examples: C-based languages (C++, Java)*

Scope (continued)

- In nested subprogram definitions, variables can be hidden from a unit by having a "closer" variable with the same name
  - JavaScript example of nested functions:

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

  - Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Pascal, Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python)

Static Scope

- In languages with nested subprogram definitions, when an expression refers to an identifier,
  - The compiler first checks the local declarations
  - If the identifier is not local, the compiler works outward through each level of nesting until it finds an identifier with same name having a declaration, where it stops.
  - Any identifier with the same name declared at a level further out is never reached
  - If compiler reaches global declarations and still cannot find the identifier, an error message results

Blocks

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

```
void sub() {
  int count; //count scope is sub function
  while (...) {
     int count; //scope of a new variable named count
     // in while-loop body. Hides count
      count++;
      ...
  }
  . . .
}
```

  - Note: The reuse of names in nested blocks is legal in C and C++, but it is not in Java and C# - too error-prone

Declaration Order

- C++ allows 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 C++, variables can be declared in `for` statements.
  - The scope of such variables is restricted to the `for` construct.

Scope (continued)

```
void sort (float a[ ], int size) {

for (int i= 0; i < size; i++)  // i local; size not
  for (int j = i+ 1; j < size; j++)
                                // j local; i, and size not
         if (a[j] < a[i]) {  // j local; a, i, nonlocal
             float t;
             t = a[i]; // t local; a, and i nonlocal
             a[i] = a[j];// a, i, j nonlocal
             a[j] = t;// t local; a, j nonlocal
        }
}//end of function
```

Scope (continued)

```
for (int i = 0; i < 10; i++) {
      cout << i << endl;
}

if(i == 10) {
// this is an invalid use
// of the i from the for loop:
// i only exists in the for loop, so
// i is not in scope here
}
```

Global Scope

- C, C++, Python, and Javascript 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 no storage) and definitions (attributes and storage)
  - A declaration outside a function definition specifies that the variable is defined global in another file and storage should not be reserved for it here.

```
extern int var;//declaration statement
```

  - A global variable that is defined after a function can be made visible in the function by declaring it to be external.
  - The idea of declarations and definitions carries over to functions as well in C and C++.

Global Scope

- A global variable that is hidden by a local variable with the same name can be accessed using the scope operator (::)

```
1 int count;
2 int function() {
3     int count;
4     count = 0;// the local count on line 3
5     ::count = 1;// the global count on line 1
6 }
```

Global Scope

- 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. Example:

```
day = “Monday”
def tester():
  global day
  print “The global day is : ”, day
  day = “Tuesday”
  print “The new value of day is: ”, day

tester()
```

  - The output of this script:

```
The global day is: Monday
The new value of day is: Tuesday
```

Evaluation of Static Scoping

- It provides a method for nonlocal access that works well in many situations.
- Problems:
  - In most cases, it allows too much access to variables and subprograms than is necessary.
    - Too crude method to concisely specifying such restrictions.
  - Effect of software continuous evolution:
    - Changes often result in restructuring.
    - Designers are encouraged to use global variables than is necessary.
    - 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)
  - Examples: Perl, Common Lisp, and SNOBOL4
  - References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

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

Dynamic Scope

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

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

Dynamic Scope

- 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

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.

Namespaces

- It is a mechanism by which the programmer can create a named scope.
  - For example, the standard header file **cstdlib** contains function prototypes for several library functions (e.g., abs function).

```
//in header file cstdlib:
namespace std{
  . . . .
  int abs(int);
  . . .  .
}
```

- Access to namespace identifiers is through one of the following methods:
  - Using a qualified name (e.g., `x = std::abs(val);`)
  - Using a declaration, such as:

    ```
    using std::abs;
    alpha = abs(beta);
    ```
  - Using directive locally or globally: `using namespace std;`

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.

```
public class Student {
  private String name;
  public Student (String name, ...) {
    this.name = name;
    // this.name required; String name
    // in constructor argument hides
    // String name in class definition
    ...}
}
```

Named Constants

- A *named constant* is a variable that is bound to a value only once.
  - 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