

Lecture 4

Names, Bindings & Scopes

Chapter 5

First Semester
1447 - 2025

CONCEPTS OF PROGRAMMING LANGUAGES



ROBERT W. SEBESTA

12/E

ISBN 0-321-49362-1

Lecture 4 Topics:

- **Introduction**
- **Names:**
 - Design Issues
 - Name Forms
 - Special Words
- **Variables:**
 - Name
 - Address
 - Type
 - Value
- **The Concept of Binding:**
 - Binding of Attributes to Variables
 - Type Bindings:
 - Static Type Binding
 - Dynamic Type Binding
 - Storage Bindings and Lifetime:
 - Static Variables
 - Stack-Dynamic Variables
 - Explicit Heap-Dynamic Variables
 - Implicit Heap-Dynamic Variables

Lecture 4 Topics (continued):

- **Scope:**
 - Static Scope
 - Blocks
 - Declaration Order
 - Global Scope
 - Evaluation of Static Scoping
 - Dynamic Scope
 - Evaluation of Dynamic Scoping
- **Scope & Lifetime**
- **Referencing Environments**
- **Named Constants**

Introduction

- **Imperative languages** are abstractions of **von Neumann architecture**, which has two components:
 - **Memory**: stores both instructions and data.
 - **Processor**: provides operations for modifying the contents of the memory.
- The abstractions in a language for the **memory cells** of the machine are **variables**.
- For example, an **integer variable**, which is usually represented directly in one or more bytes of memory.

Introduction

- **Variables** are characterized by a collection of **attributes** (**properties**), the most important of which is **type**.
- **Data Type** is a fundamental concept in programming languages (Chapter 6).
- To design a **data type**, must consider these issues:
 - **Variable Scope**.
 - **Variable Lifetime**.
 - **Variable Type Checking**.
 - **Variable Initialization**.
 - **Variable Type Compatibility**.

Names

- **Names** are one of the fundamental **attributes** of **variables**.
- **Names** are also associated with **subprograms**, **formal parameters**, and other **program constructs**.
- The term **identifier** is often used *interchangeably* with **name**.
- **Design issues** for **names**:
 - Are **names** case sensitive?
 - Are **special words** of the language **reserved words** or **keywords**?

Names Forms

- A **name** is a string of characters used to identify some entity in a program.
- In most PLs, **names** have the **same form**:
 - A **letter** followed by a **string** consisting of letters, digits, and underscore characters.
- **Name 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 (**External names** are those defined **outside functions**, which must be handled by the **linker**).
 - **C#** and **Java**: no limit, and all are significant.
 - **C++**: no limit, but implementers often impose one.

Names Forms (continued)

- **Special Characters:**

- **PHP:** all variable names must begin with dollar signs (\$).
- **Perl:** all variable names begin with special characters (\$, @ or %), which specify the variable's type.
- **Ruby:** variable names that begin with @ are instance variables; those that begin with @@ are class variables.

Names Forms (continued)

- **Case Sensitivity:**
 - Uppercase and lowercase letters in **names** are **distinct**.
 - **Names** in the **C-based languages** are case sensitive.
 - **Names** in others are not.
- **Disadvantage:**
 - **Readability:** **names** that look alike are different, such as `rose`, `ROSE` and `Rose`
 - Worse in **C++**, **Java**, and **C#** because predefined names are mixed case (e.g., `IndexOutOfBoundsException`).

Names Forms (continued)

■ Special Words:

- An aid to readability; used to delimit or separate statement clauses (**syntactic** parts of statements and programs).
- A *keyword* is a word that is special only in certain contexts.
- 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** or a **collection of cells**.
- Variables can be characterized as a **sextuple of attributes**:
 1. **Name**
 2. **Address**
 3. **Value**
 4. **Type**
 5. **Lifetime**
 6. **Scope**

Variables Attributes

- **Name**: the most common names in programs and not all variables have them.
- **Address**: the memory address (**l-value**) with which it is associated.
 - A **variable** may have different addresses at different times during execution (like in a *subprogram calls*).
 - A **variable** may have different addresses at different places in a program (*like in dynamic-allocation*).
 - 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 types.
 - **Aliases** are harmful to **readability** (program readers must remember all of them).

Variables Attributes (continued)

- **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.
- **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 (*needs l-value to be accessed first*).
 - Abstract memory cell: the physical cell or collection of cells associated with a **variable**.

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.
- **Binding** and **Binding time** are prominent concepts in the semantics of programming languages.

Possible Binding Times

- **Language Design Time:**

- Example: bind operator symbols to operations.
 - E.g., the asterisk symbol (*) is usually bound to the multiplication operation...etc.
 - `area = length * width;`

- **Language Implementation Time:**

- Example: bind floating point data type to a representation (a range of possible values).
 - `float area = 20.2;`

- **Compile Time:**

- Example: bind a variable to a type in C or Java.
 - `int factor;`

Possible Binding Times (continued)

- **Load Time:**

- Example: bind a C or C++ `static` variable to a memory cell.
 - E.g., binding the previous “`area`” variable to the memory cell (a storage) “`0x00000000`”

- **Link Time:**

- Example: a call to a library subprogram is bound to the subprogram code at link time.
 - E.g., `int m = java.lang.Math.max(20, 22);`

- **Run Time:**

- Example: bind a non-static local variable to a memory cell.
 - E.g., variables declared in Java methods.

Binding Times: Example

- Some of the **bindings** and their **binding times** for the parts of the assignment statement below are as follows:

`count = count + 5;`

- The **type** of `count` is bound at **compile time**.
- The set of possible **values** of `count` is bound at **compiler design time**.
- The meaning of the **operator symbol** (+) is bound at **compile time**, when the **types** of its operands have been determined.
- The internal representation of the literal (5) is bound at **compiler design time**.
- The **value** of `count` is bound at **execution time** (run time) with the statement.

Binding of Attributes to Variables

- **Static** and **Dynamic** Binding:
 - A **binding** is **static** if it first (1) occurs before run time and (2) remains unchanged throughout program execution.
 - A **binding** is **dynamic** if it first (1) occurs during execution or can (2) change during execution of the program.

Type Bindings

- Before a **variable** can be referenced in a program, it must be bound to a **data type**.
- The two important aspects of this **binding** are:
 - How is a data type specified?
 - When does the binding take place?
- If **static**, the **type** may be specified by either an (1) **explicit** or an (2) **implicit** declaration.

Static Type Binding

- 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.
 - Implicit variable type binding is done by the **language processor**, either a **compiler** or an **interpreter**.
- Basic, Perl, Ruby, JavaScript, and PHP provide **implicit declarations**.
 - **Advantage**: writability (a minor convenience).
 - **Disadvantage**: reliability (less trouble with Perl).

Static Type Binding (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**. Examples:
 - `var sum = 0;`
 - `var total = 0.0;`
 - `var name = "Ali";`
 - Visual Basic 9.0+, ML, Haskell, and F# use **type inferencing**.
 - The **context** of the appearance of a variable determines its type.

Dynamic Type 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;`
- **Advantage:** flexibility (generic program units)
 - Any variable can be assigned any type value.
 - a variable's type can change any number of times during program execution.
- **Disadvantages:**
 - High cost (dynamic type checking and interpretation).
 - Type error detection by the compiler is difficult.

Storage Bindings and 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.
 - So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell.

Storage Bindings and Lifetime

- To investigate **storage bindings** of **variables**, it is convenient to separate **scalar** (unstructured) variables into **four categories**, according to their **lifetimes**.
- These categories are named:
 - **Static**.
 - **Stack-Dynamic**.
 - **Explicit Heap-Dynamic**.
 - **Implicit Heap-Dynamic**.

Static Variables

- A **static variable** is bound to memory cells (1) before execution begins and (2) 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 Variables

- **Stack-dynamic**: **storage bindings** are created for **variables** when their declaration statements are **elaborated** (A declaration is elaborated when the 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.
- **Advantage**: allows recursion; conserves storage.
- **Disadvantages**:
 - Overhead of **allocation** and **deallocation**.
 - Subprograms cannot be history sensitive.
 - Inefficient references (indirect addressing).

Explicit Heap-Dynamic Variables

- **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.
- **Code Example in C++:**

```
int *intnode;           // Create a pointer
intnode = new int;      // Create the heap-dynamic variable
. . .
delete intnode;         // Deallocate the heap-dynamic variable
                        // to which int node points
```

Explicit Heap-Dynamic Variables

- **Advantage:** provides for dynamic storage management for dynamic structures such as:
 - Linked list.
 - Trees.
 - etc.
- **Disadvantage:**
 - Inefficient
 - Unreliable.

Implicit Heap-Dynamic Variables

- **Implicit Heap-Dynamic**: allocation and deallocation caused by assignment statements.
 - All variables in APL; all strings and arrays in Perl, JavaScript, and PHP.
 - **Example (JavaScript)**:
 - `highs = [74, 84, 86, 90, 71];`
- **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.
- **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 (source code).
- To connect a name reference to a **variable**, you (or the *compiler*) must find the declaration.
- **Search process:** (1) 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 allow nested subprogram definitions, which create **nested static scopes** (e.g., Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python).

Scope (continued)

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

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

JavaScript function, `big`, in which the two functions `sub1` and `sub2` are nested.

Blocks

- A method of creating **static scopes** inside program units.
- From **ALGOL 60**.
- Example in **C**:
- **Note**: legal in **C** and **C++**, but not in **Java** and **C#**.
 - Too error-prone.

```
void sub()  
{  
    int count;  
    while (...)  
    {  
        int count;  
        count++;  
        ...  
    }  
    ...  
}
```

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**:

```
(LET (  
  (name1 expression1)  
  ...  
  (namen expressionn)  
)
```

```
(LET (  
  (top (+ a b))  
  (bottom (- c d)))  
  (/ top bottom)  
)
```

Scheme example

(a + b) / (c - d)

The **LET** Construct (continued)

- In **ML (Meta Language)**:

let

val name₁ = expression₁

...

val name_n = expression_n

in

expression

end;

```
let
  val top = a + b
  val bottom = c - d
in
  top / bottom
end;
```

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 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.

Declaration Order C# Example

```
{  
    {int x; // Illegal  
    ...  
    }  
    int x;  
}
```

C# does not allow the **declaration** of a **variable** in a **nested block** to have the same name as a variable in a **nesting scope**.

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.

```
void fun() {  
    . . .  
    for (int count = 0; count < 10; count++) {  
        . . .  
    }  
    . . .  
}
```

In later versions of **C++**, as well as in **Java** and **C#**, the **scope** of `count` is from the `for` statement to the end of its body (the right brace).

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.

```
extern int sum;           // C & C++
```

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)

■ PHP Example:

```
$day = "Monday";  
$month = "January";  
  
function calendar() {  
    $day = "Tuesday";  
    global $month;  
    print "local day is $day ";  
    $gday = $GLOBALS['day'];  
    print "global day is $gday <br \>";  
    print "global month is $month ";  
}  
  
calendar();
```

Interpretation of this code produces the following:

```
local day is Tuesday  
global day is Monday  
global month is January
```

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.

```
day = "Monday"

def tester():
    print "The global day is:", day

tester()
```

```
The global day is: Monday
```

Global Scope (continued)

■ Python Example:

```
day = "Monday"

def tester():
    print "The global day is:", day
    day = "Tuesday"
    print "The new value of day is:", day

tester()
```

UnboundLocalError

Global Scope (continued)

- **Python 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 is as follows:

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

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.
- Thus, the **scope** can be determined only at **run time**.

Scope Example

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

// 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:***
 - While a subprogram is executing, its **variables** are visible to all subprograms it calls.
 - Impossible to statically **type check**.
 - **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.

```
void printhead() {  
    . . .  
} /* end of printhead */  
void compute() {  
    int sum;  
    . . .  
    printhead();  
} /* end of compute */
```

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.

Referencing Environments

■ Python Example:

```
g = 3; # A global

def sub1():
    a = 5; # Creates a local
    b = 7; # Creates another local
    . . . <----- 1
def sub2():
    global g; # Global g is now assignable here
    c = 9; # Creates a new local
    . . . <----- 2
def sub3():
    nonlocal c: # Makes nonlocal c visible here
    g = 11; # Creates a new local
    . . . <----- 3
```

See the next slide.

Referencing Environments

■ Python Example:

The referencing environments of the indicated program points are as follows:

<i>Point</i>	<i>Referencing Environment</i>
1	local a and b (of sub1), global g for reference, but not for assignment
2	local c (of sub2), global g for both reference and for assignment
3	nonlocal c (of sub2), local g (of sub3)

Referencing Environments

- **C, C++ Example:**
- The referencing environments of the indicated program points (left) are as follows:

```
void sub1() {  
    int a, b;  
    . . . <----- 1  
} /* end of sub1 */  
void sub2() {  
    int b, c;  
    . . . . <----- 2  
    sub1();  
} /* end of sub2 */  
void main() {  
    int c, d;  
    . . . <----- 3  
    sub2();  
} /* end of main */
```

Point

Referencing Environment

1

a and b of sub1, c of sub2, d of main, (c of main and b of sub2 are hidden)

2

b and c of sub2, d of main, (c of main is hidden)

3

c and d of main

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**:
 - 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.

Named Constants Java Example

```
void example() {  
    int[] intList = new int[100];  
    String[] strList = new String[100];  
    . . .  
    for (index = 0; index < 100; index++) {  
        . . .  
    }  
    . . .  
    for (index = 0; index < 100; index++) {  
        . . .  
    }  
    . . .  
    average = sum / 100;  
    . . .  
}
```

Named Constants Java Example

```
void example() {  
    final int len = 100;  
    int[] intList = new int[len];  
    String[] strList = new String[len];  
    . . .  
    for (index = 0; index < len; index++) {  
        . . .  
    }  
    . . .  
    for (index = 0; index < len; index++) {  
        . . .  
    }  
    . . .  
    average = sum / len;  
    . . .  
}
```

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.

Any Questions?

- Please, read chapter 5.
- I hope you were taking some notes!
- To test your understanding of this lecture, have a go with the “Review Questions” in pages **227-228** of the textbook.
- We will do more exercises later on.
- Please, keep reviewing this lecture regularly.