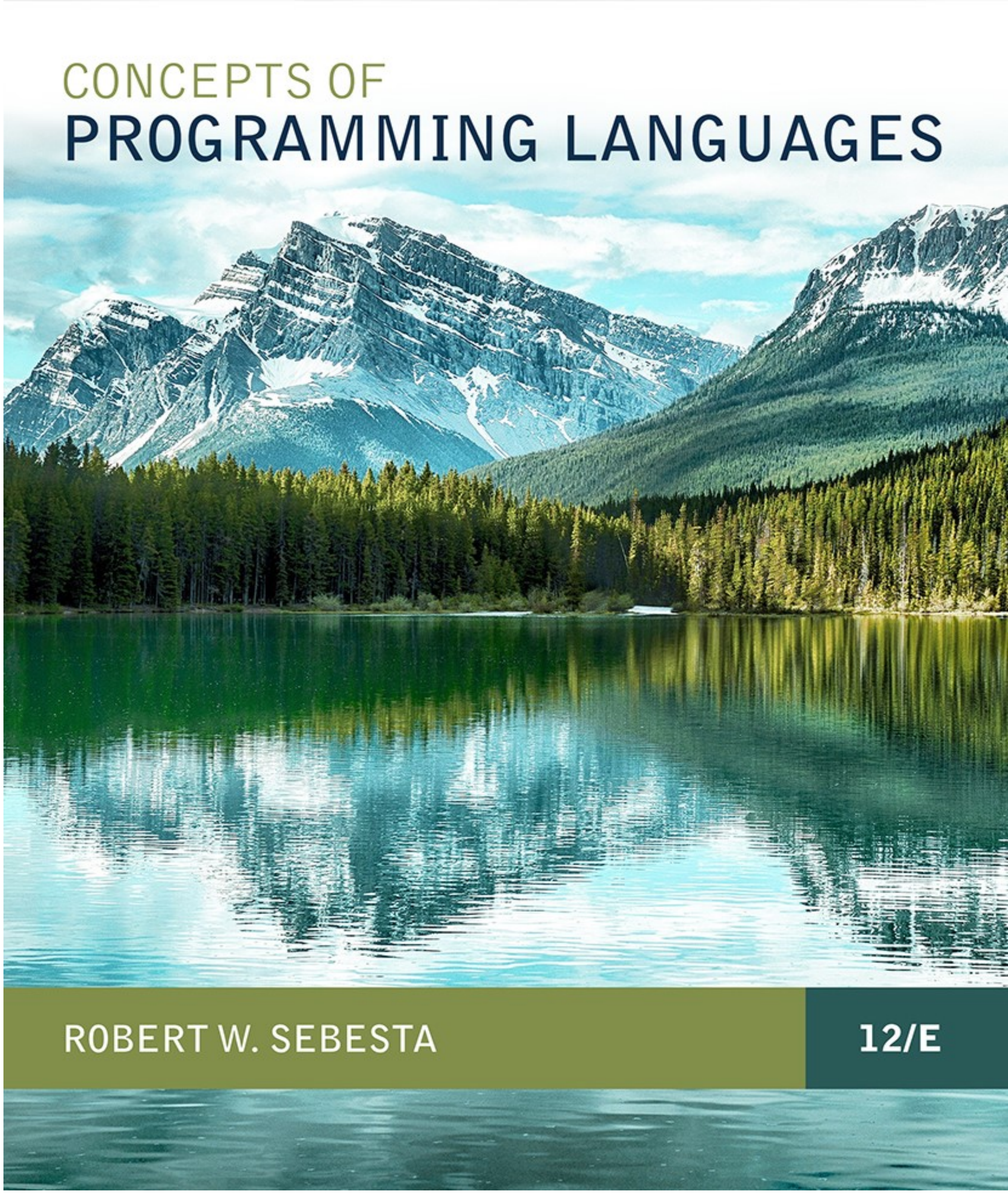
Chapter 5

Names, Bindings, and Scopes



ISBN 0-321-49362-1

Chapter 5: Name, Binding, Scope

- Introduction
- Variable
  - Name
  - Address
  - Value
  - Type
    - Binding
  - Lifetime:
    - Static, Stack-Dynamic, Explicit/Implicit Heap-Dynamic
  - Scope
    - Static, Dynamic, Global
- Referencing Environment
- Named Constant

Introduction

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

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

Variable Attribute: Name

- Design issues for names:
  - Length?
  - Special Characters?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Variable Attribute: Name

- Not all variables have a name
  - E.g., in C++: *new* int; // create a nameless heap-dynamic variable
- 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

Variable Attribute: Name

- Special characters
  - PHP: all variable names must begin with **dollar signs $**
    - E.g., $text = "Hello world!";
  - Perl: all variable names begin with special characters, which specify the variable's type—scalar ($), array (@), or hash (%)
    - E.g., $pi = 3.141592; @colors = ("red", "green"); %grades= (A=> 90, B => 80);
  - Ruby: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables
    - E.g., @inst= "csci6221.10" @@class="csci6221"

Variable Attribute: Name

- Case sensitivity
  - Disadvantage: readability--names that look alike are different
    - Names in the C-based or Java languages are **case** sensitive
    - Names in others, such as **Ada**, are **not**
    - In C++, Java, and C# : **predefined** names are **mixed** case
      - e.g. `IndexOutOfBoundsException`

Variable Attribute: Name

- 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/meaning
    - E.g., **Java**: finally, class; goto is **NOT** a keyword (i.e., no goto in Java)
  - A *reserved word* is a special word that **cannot** be used as a **user-defined** name/identifier
    - E.g., **Java**: finally, class, char, **goto**
  - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variable Attribute: Address

- *Address* – the memory address with
  - a variable may have **different addresses at different times during execution**
  - a variable may have different addresses at different places in a program
  - 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)

Variable Attribute: Type, Value

- *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—single or double
- *Value* – the contents of the location (address) with which the variable is associated.

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
- between an operation and a symbol

- *Binding time*--a binding takes place
  - Compile time -- bind a variable to a type in Java
  - Load time -- bind a C++ `static` variable to a memory cell
  - Runtime -- bind a non-static 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.
  - may be specified by either an explicit or an implicit declaration
- A binding is *dynamic* if it first occurs **during** execution or can change during execution of the program
  - may be specified through an assignment statement

Static Type Bindings

- An *explicit declaration* is a program statement used for declaring the types of variables
  - int final; // in C
- An *implicit declaration* is a **default** mechanism for specifying types of variables through *default* conventions
  - var csci6221; // in JavaScript
- Basic, Perl, Ruby, JavaScript, and PHP provide implicit declarations
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Static Type Bindings (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
    - **var** sum = 0; /// type of sum is *int*
    - **var** total = 0.0; /// type of sum is *float*
    - **var** name= “CSCI”; /// type of sum is *string*

  - 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)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)

Variable Attribute: Lifetime

- Storage Bindings
  - Allocation - getting a memory cell from some pool of available cells
  - Deallocation - putting a memory cell back into the pool
- Lifetime
  - 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)

Note: http://stackoverflow.com/questions/572547/what-does-static-mean-in-a-c-program

Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are *elaborated*.
  - E.g., when making a function call in C, stack-dynamic variables (in the parameter list OR defined in the callee function without *static*) are created in the runtime stack;
  - Lifetime of these variables ends when function call ends.
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation at the runtime
  - 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
- 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
  - 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

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is **visible**
- *Local variables* of a program unit are those that are declared in that unit
- *Nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special nonlocal variable
  - E.g., JavaScript:

```
var y = 0;     //  y:  global variable
function outer() {
      var x = 1;    // x: local to outer()
      function inner() {
            x += 1; // x: non-local to inner()
      }
}
```

Static Scope

- To connect a name **reference** to a variable, 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
- 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

Blocks

- A method of creating static scopes inside program units
- Example in C:

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

Note: legal in C, C++, but **not** in Java, C#--too error-prone

The `LET` Construct

- Most functional languages include some form of `let` construct
  - Related to *blocks* of imperative languages
- E.g., in Scheme:

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

The **LET** Construct (continued)

- E.g., in ML:

```
let
   val name₁ = expression₁
   …
   val nameₙ = expressionₙ
in
  expression
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 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

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
  - e.g.,

```
for ( int i=1; i<100;  i++ ) {
        System.out.println("I= " + i);
}
```

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

```
def func():
    global  g
    print  g
    g = "reset to default"      # reset global var
    print g

g = "set global var's val"    # update global var
func()
g = "update global var's val again"
print g
```

set global var's val
reset to default
update global var's val again

Evaluation of Static Scoping

- Works well in many situations
- Problems:
  - In most cases, too much access is possible
  - As a program evolves, if the initial structure is destroyed, 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
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Scope Example

```
function big() {
    function sub1() {
        var x = 7;
        sub2();
    }
    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 = 3**
- Dynamic scoping
  - Reference to `x` in **sub2** is to `sub1`**'s x = 7**

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
  - The scope of the variable is *static* and *local* to that function
  - The lifetime of the variable extends **over the entire execution** of the program

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
  - In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms
    - A subprogram is active if its execution has begun but has not yet terminated

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
- E.g.,
  - C++ and Java: expressions of any kind, dynamically bound
    - Java: **final** List csci6221; … csci6221= new ArrayList();
  - 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