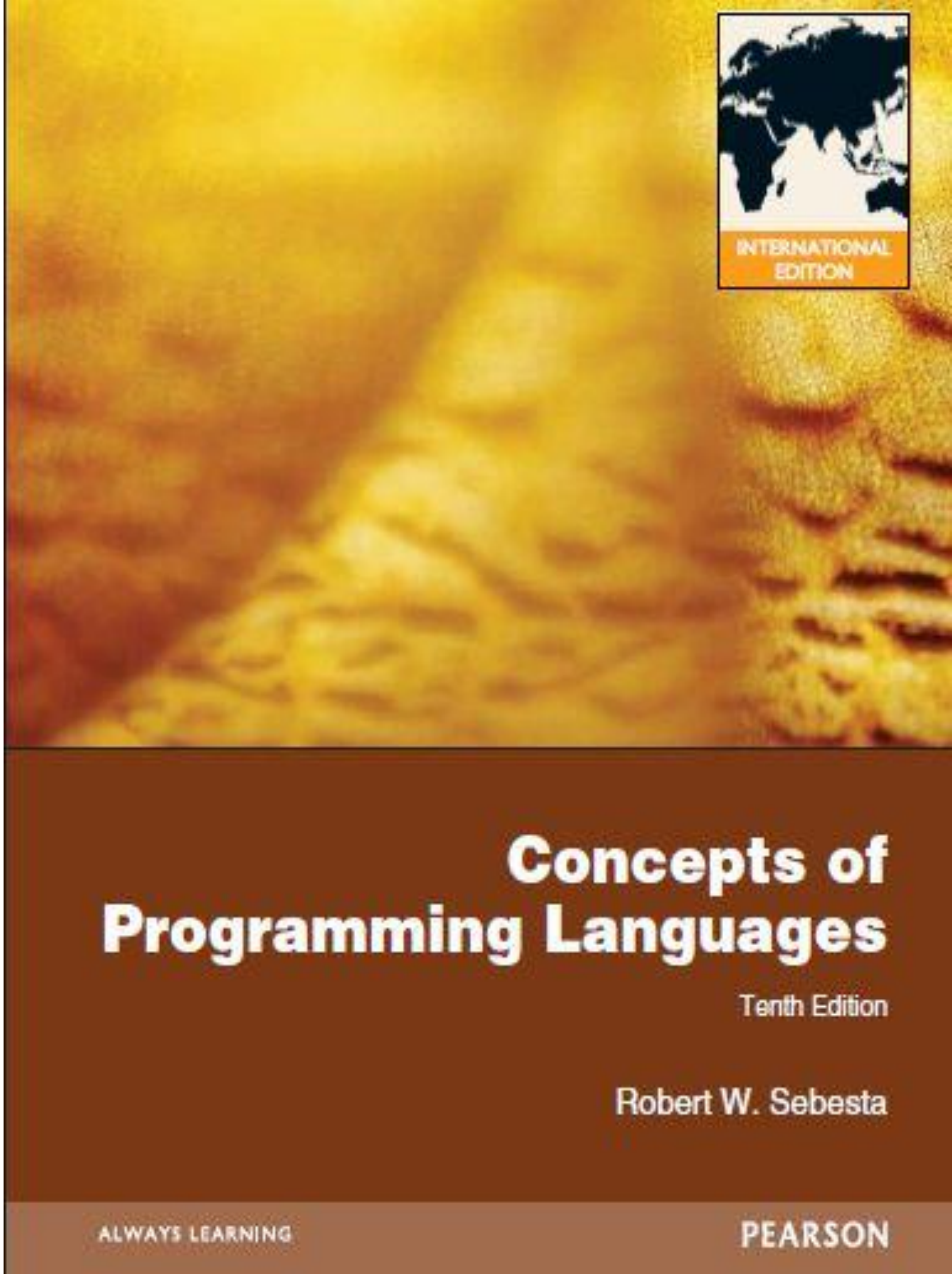
Programming Language

Instructor:
Min-Chun Hu
anita_hu@mail.ncku.edu.tw



Lecture 7
Names, Bindings, and Scopes

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

Introduction

- Imperative languages are abstractions of von Neumann computer architecture
- Composed of two primary components:
  - Memory: stores both instructions and data
  - Processor: provides operations for modifying the contents of the memory

Variables

- A variable is the abstraction of a memory cell
- A variable can be characterized by a sixtuple of attributes
  - Name: not all variables have them
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Names

- A name is a string of characters used to identify some entity in a program.
  - A letter followed by a string consisting of letters, digits, and underscore characters "_".
- Design issues for names:
  - Maximum length?
  - Are names case sensitive?
  - Are special words of the language reserved words or keywords?

Names (Cont.)

- 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 (Cont.)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN 95: maximum of 31
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C#, Ada, and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Names (Cont.)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case-sensitive
    - In C, the case-sensitive problem is avoided by the convention that variable names do not include uppercase letters
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Names (Cont.)

- Special words
  - An aid to readability; used to delimit or separate statement clauses
  - A keyword is a special word only in certain contexts
    - e.g., in Fortran:
      - `Real VarName` (`Real` *is a data type followed with a name, therefore* `Real` *is a keyword)*
      - `Real = 3.4` (*`Real` is a variable name)*
  - A *reserved word* is a special word that cannot be used as a user-defined name
  - Potential problem with reserved words:
    - If the language include a large number of reserved words, many collisions would occur (e.g., COBOL has 300 reserved words!)

Names (Cont.)

- Class Definition of This in Visual Basic.NET:

```
Public Class this
        This class does something...
End Class
```

- Using This Class in C#:

```
this x = new this();  //Won't compile!
@this x = new @this();  // Will compile!
```

Address

- The memory address that a variable is associated
  - Called *l-value* because the address is what required when the name of a variable appears in the left side of an assignment statement
  - A variable may have different addresses at different times during execution or at different places in a program
    - E.g. sum in sub1 and sub2
  - If multiple 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)

```c
struct sdata {
    int x;
    long y;
    double z;
    char *a;
};
```

```c
union udata {
    int x;
    long y;
    double z;
    char *a;
};
```

Example of struct

```
#include <iostream.h>
#pragma pack(8)
struct example1
{
short a;
long b;
};
struct example2
{
char c;
example1 struct1;
short e;
};
#pragma pack()
int main(int argc, char* argv[])
{
example2 struct2;
cout << sizeof(example1) << endl;
cout << sizeof(example2) << endl;
return 0;
}
```

Example of union

```
#include <stdio.h>

union data {
    int vi;
    double vd;
};

int main(void)
{
    union data a;
    a.vi = 11;
    printf("a = (%d, %f)\n", a.vi, a.vd);
    a.vd = 22.0;
    printf("a = (%d, %f)\n", a.vi, a.vd);
    return 0;
}
```

int : 4bytes   double: 8 bytes

data	0…0	0…0

a=(11, 0.0000000)

a=(0, 22.000000)

Value

- The contents of the location that a variable is associated
  - Called *r-value* because it is what required when the name of the variable appears in the right side of an assignment statement

Type

- Determines the range of values of variables and the set of operations that are defined for values of that type
  - E.g. `int` type in JAVA specifies a value range of −2147483648 to 2147483647, and arithmetic operations for addition, subtraction, multiplication, division, and modulus
- In the case of floating point, type also determines the precision

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.

Possible Binding Times

- Language design time
  - Bind operator symbols to operations, e.g. '*' is usually bound to the multiplication operation
- Language implementation time
  - Bind data type to a range of possible values
- Compile time
  - Bind a variable to a type in C or Java
- Load time
  - Bind a C or C++ static variable to a memory cell
- Link time
  - Bind a library subprogram to the subprogram code
- Run time
  - 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.
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Type Binding

- How is a type specified?
- When does the binding take place?
- If static, the type may be specified by either an explicit or an 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
- Fortran, BASIC, Perl, Ruby, JavaScript, and PHP provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Static Type Binding (Cont.)

- Some languages use type inferencing to determine types of variables according to context
  - C# - a variable can be declared with `var` and an initial value. The initial value sets the type
  - Visual BASIC 9.0+, ML, Haskell, F#, and Go use type inferencing. The context of the appearance of a variable determines its type

Ex:

fun circumf(r) = 3.14159 * r * r;

Function takes a real arg. and produces a real result.

The types are inferred from the type of the constant.

fun times10(x) = 10 * x;

The argument and functional value are inferred to be **int**.

Dynamic Type Binding

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

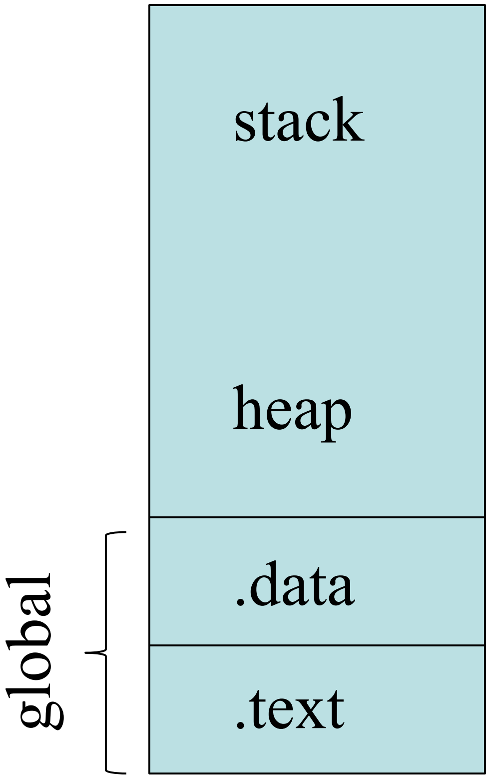
```
list = [2, 4.33, 6, 8];

list = 17.3;
```

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

Storage Binding 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
- Four categories of variables according to lifetimes:
  - Static variables
  - Stack-dynamic variables
  - Explicit heap-dynamic variables
  - Implicit heap-dynamic variables




```
void f(){
int i;\\non-static local variable (in stack)
static int l;\\static local variable (in .data)
}
```

JAVA Example

heap variable

stack variable

```
class MyClass {
  static int a;  class variable
  int b;    instance variable
  public static void myMethod(int c) { method parameter
      try {
int d;  local variable
     } catch (Exception e) {  exception-handler parameter
     }
  }
   MyClass(int f) {  constructor parameter
     int[] g = new int[100];
local variable   array component
  }
}
```

Static Variables

- 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
- Disadvantages:
  - Lack of flexibility (no recursion)
  - Storage cannot be shared among variables

Stack-Dynamic Variables

- Storage bindings are created for variables when their declaration statements are *elaborated (i.e. executed)*.
- If scalar, all attributes except address are statically bound
  - Local variables in C subprograms (not declared static) and Java methods are stack-dynamic variables
- Advantages:
  - Allows recursion
  - Conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Explicit Heap-Dynamic Variables

- Allocated and deallocated by explicit run-time instructions specified by the programmer
- Referenced only through pointers or references
- E.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java
  - **int** *intnode; // Create a pointer
    intnode = **new int**; //Create the heap-dynamic variable
    …
    delete intnode; // Deallocate the heap-dynamic variable
- Advantage:
  - Provides for dynamic storage management
- Disadvantage:
  - Inefficient and unreliable
  - Difficult to use pointer and reference variables correctly

Implicit Heap-Dynamic Variables

- Allocation and deallocation caused by assignment statements
- E.g. All variables in APL; all strings and arrays in Perl, JavaScript, and PHP
  - hights = [74, 84, 86, 90, 71]
- Advantage:
  - Flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

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 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

Static Scope

- The scope of a variable is statically determined based on program text
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- To connect a name reference to a variable, you (or the compiler) must find the declaration

Static Scope (Cont.)

- 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

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)

➔ The scope can be determined only at run time

- References to variables are connected to declarations by searching back through the chain of subprogram calls

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

LISP Example

```
;(defvar x 1)

(let (
      (x -1)
    )
    (defun f (choice)
        (let (
              (x 5)
            )
            (defun h ()
                (format standard-output "%d\n" x)
            )
            (defun g1 ()
                (let (
                      (x 31)
                    )
                    (h)
                )
            )
            (defun g2 ()
                (let (
                      (x 42)
                    )
                    (h)
                )
            )
            (if (eq choice 1) (g1) (g2))
        )
    )
    (f 1)
    (f 2)
    (format standard-output "%d\n" x)
)
```

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

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

Declaration Order

- C99, C++, Java, JavaScript, and C# allow variable declarations to appear anywhere a statement appear in a program unit
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In C# and JavaScript, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - In C#, a variable still must be declared before it can be used
    - In JavaScript, the use of a variable before declaration will result in the value `undefined`

Declaration Order (Cont.)

- 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 count=0; count<10; count++) {

    …

    }

Blocks

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

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

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

Block: 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))
   expression
)
```

Example: (a+b)/(c+d)

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

Block: The LET Construct (Cont.)

- In ML:

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

- In F#:
  - First part: `let` left_side = expression
  - (left_side is either a name or a tuple pattern)
  - All that follows is the second part

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 specify attributes) and definitions (specify attributes and cause storage allocation)
  - A global variable defined after a function can be made visible in the function by declaring it to be external:
    - **extern int** sum;

Global Scope (Cont.)

- 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 (Cont.)

- 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

Global Scope (Cont.)

- Example:

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

Output:
local day is Tuesday
global day is Monday
global month is January

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Example:

```
void printheader() {
…
}
void compute() {
  int sum;
  …
  printheader()
}
```

-- The scope of `sum` is completely contained within the `compute` function, but does not extend to the body of `printheader` function

-- The lifetime of `sum` extends over the time during which `printheader` executes

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
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - Ada, 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