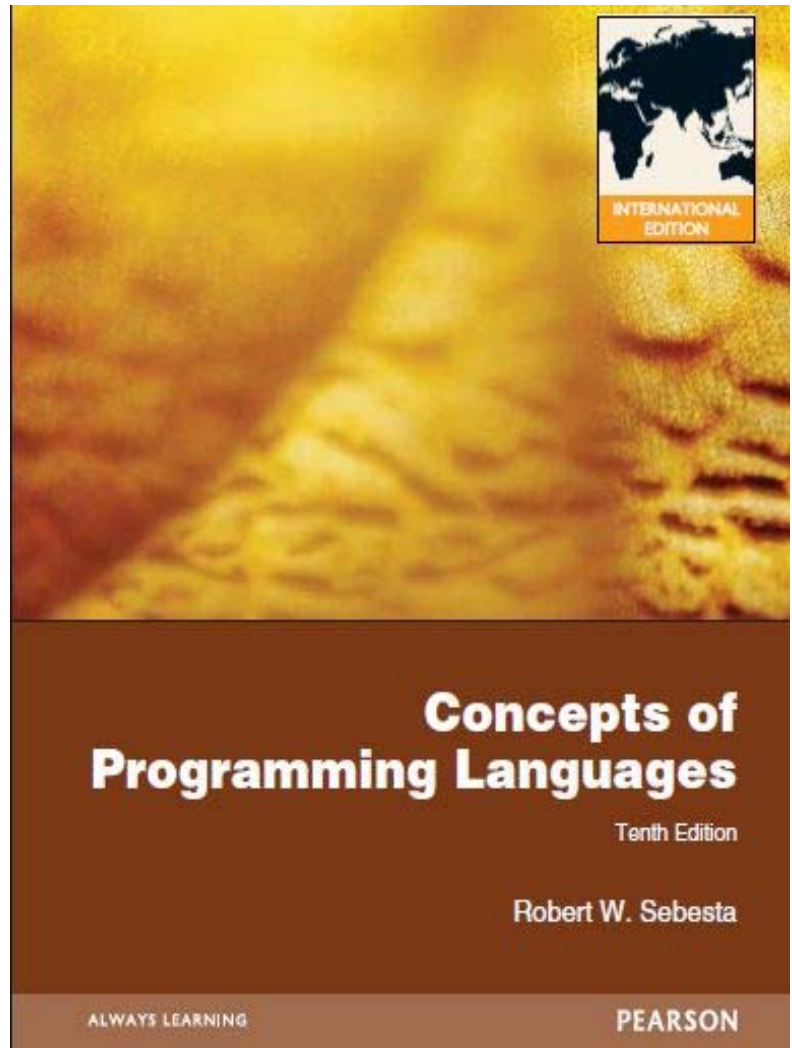


Programming Language

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

Stored programming：程式執一定要將欲執行的指令跟資料放入記憶體方可執行

Variables

- A variable is the **abstraction** of a memory cell
- A variable can be characterized by a sextuple of attributes
 - Name: **not all variables have them** new
 - Address
 - Value
 - Type
 - Lifetime 從執行到結束，變數存在的時間
 - Scope 空間

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:
 - ▣ Are names case sensitive? 避免取同樣名稱
 - ▣ Are special words of the language reserved words or keywords? 不能作為名稱使用

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.)

- 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.)

- 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 程式本身用到的syntax，已被賦與意義 可當變數名稱使用
 - 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 不能當變數名稱使用
- 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!)

Address

- The memory address that a variable is associated
 - Called ^{位置}*l-value* because the address is what required when the name of a variable appears in the left side of an assignment statement *r-value*: 要傳入的值
 - A variable may have different addresses at different times during execution or at different places in a program
 - 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)

Union Example

Union是一種將不同data types儲存在同一個記憶體空間的特殊自訂型別，union一次只會儲存一個變數資料，所有在union宣告的變數會共享同一個記憶體空間，而且會已宣告的變數型態size最大的變數空間作為記憶體空間。

```
#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, 22.0000000)

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
- *Abstract memory cell* – the physical cell or collection of cells associated with a variable

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. 把所有attribute和variable 連結起來

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. 在程式執行前
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program
在程式執時才發生

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.)

自動推測是屬於何種type

- 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

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

真正開始佔用記憶體空間的時候

- 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 global variable
- Stack-dynamic variables local variable 可以預期variable
- Explicit heap-dynamic variables 不可預期
- Implicit heap-dynamic variables

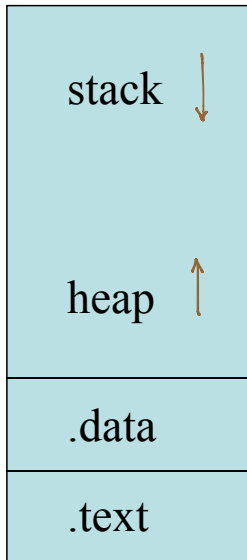
ps. Java有garbage collector所以不用去delete new出來的物件

ps.當資訊為動態配置產生時，系統會存放在Heap（與資料結構中的heap無關）

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

high address



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

Global : global variable 、static variable

Stack（堆疊）中存放的資料：local variable 、function / method parameter 、function / method return address，由系統管理，必須在編譯時期為已知。這些變數的回收發生在從stack pop出去（因為個數、大小都為已知，所以系統知道怎麼進行配置與回收）

Heap（堆積）中存放的資料：new產生的物件、malloc array，這裡的記憶體由使用者負責進行回收，使用這裡的記憶體主要是用在編譯時期還不知道大小或個數的變數

Example

heap variable

stack variable

```
class MyClass {  
    static int a; class variable 只有相同class中，才能使用static 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  
    }  
}
```


Stack-Dynamic Variables

- 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
 - ▣ 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
 - `heights = [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 靜態領域

- 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**

在區塊（block）中未定義的變數，根據程式的結構，在程式中包含其區塊的上一層區塊處尋找此變數是否在此區塊內定義，若找到變數的宣告則變數即是在此區塊內定義，若未找到則繼續往外層尋找直到變數第一次宣告處為止。若搜尋完整個程式仍未找到變數之宣告，則該變數即為未定義之變數

Static Scope (Cont.) 往外層找

- 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 動態領域

- 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

動態領域法是指在區塊中未定義的變數的型態會依副程式被呼叫的反順序來尋找。

Example

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

big **calls** sub1
sub1 **calls** sub2
sub2 **uses** x

□ Static scoping $y = 3$

➤ Reference to x in sub2 is to big's x

□ Dynamic scoping $y = 7$

➤ Reference to x in sub2 is to sub1's x

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: 較簡單
 - 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

程式增長，local variable變成global variable

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++) {`

~~`int count`~~
dynamic

`... for迴圈外不能呼叫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**: 使用不同file的variable
 - `extern int sum;` 使用extern不能去重新assign值

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
```

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

Scope and Lifetime

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

- Example:

```
void printhead() {  
    ...  
}
```

```
void compute() {  
    int sum;  
    ...  
    printhead()  
}
```

scope (範圍)

Lifetime(時間)

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

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

Referencing Environments

- The *referencing environment* of a statement is the collection of all names that are visible in the statement 與static scoping或dynamic scoping有關
- 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