Chapter 5

Names, Bindings, and Scopes

Topics

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

Introduction

- Imperative languages are abstractions of von Neumann architecture.
  - Memory
  - Processor
- Variable
  - an abstraction of a memory cell (see chap1-#20)
  - characterized as a sextuple of attributes:

    name, address, value, ***type***, lifetime, scope.
- To design a type, the issues to consider:
  - scope,
  - lifetime,
  - type checking,
  - initialization, and
  - type compatibility

Name (= identifier)

- The fundamental attribute of a variable.
- Names are associated with subprograms, formal parameters, and other program constructs.
- Design issues for names:
  - Maximum length? Case sensitive?
  - Are special words reserved words or keywords?
- Length
  - If too short, they cannot be connotative.
  - Examples:
    - FORTRAN 90, ANSI C: maximum 31
    - 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 (cont.)

- Form:
  - A letter followed by a string consisting of letters, digits, _. (chap3-#4)
  - Camel notation or underscore separation: `myStack` or `my_stack`
- Special characters
  - PHP: all variable names must begin with dollar signs ($)
  - Perl: all variable names begin with special characters, which specify the variable's type ($, @, %)
  - Ruby: instance variable names begin with @;
    class variable names begin with @@.
- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - C-based languages, Java, Modula-2: case-sensitive.
    - In C: no uppercase letters in the name.
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `parseInt:` conversion of string to integer)

Names (cont.)

- Special words
  - An aid to readability; used to delimit or separate statement clauses.
  - *keyword*: a word that is special only in a certain context, so it can be redefined – in Fortran.
  - *reserved word*: a special word that cannot be used as a user-defined name; so it can't be redefined.
  - 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 (collection of) memory cell(s). – chap1-#19-#20
- Variables can be characterized as a sextuple of attributes:
  – Name
  – Address
  – Value
  – Type
  – Lifetime
  – Scope

Variables Attributes

- *Name*: Most variables have names, not all.
- *Address (L-value)*: the memory address with which it is associated.
  - A variable may have different addresses *at different times* during execution.
  - *Aliases*:
    - Multiple variable names can be used to access the same memory location.
    - How aliases can be created: via pointers, reference variables, unions (C and C++).
    - Aliases are harmful to readability (program readers must remember all of them).

Variables Attributes (cont.)

- *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 $\in[-2^{31}, 2^{31}-1]$.
  - in floating point, type also determines the precision.
- *Value (R-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*.
  - The value of each simple non-structured type is considered to occupy a *single abstract* cell. Thus, the term memory cell will mean *abstract memory cell*.

The Concept of Binding

- *Binding:*
  - An association

    between an entity and an attribute, such as
    between a *variable* and its *type* or *value*, or
    between an *operator* and a *symbol*.

  - *Binding Time:*

    the time at which a binding takes place.

Possible Binding Times

- Language *Design* time
  - bind operator symbols to operations; e.g.) * (multiplication)
- Language *Implementation* time
  - bind **`int`** type to a range of values $\in [-2^{31}, 2^{31}-1]$.
- *Compile* time
  - bind a variable to a particular type in C or Java
- *Load* time
  - bind a *static variable* to a memory cell in C or C++.
- *Runtime*
  - bind a *non-static local variable* to a memory cell in Java.
- *Link time*
  - bind a *library* subprogram to the subprogram code.

Possible Binding Times: Example

In C++: int count

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 *run* time with this statement.

Binding of Attributes to Variables: 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 the execution of the program.

(Static/Dynamic) 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: Explicit/Implicit Declaration

- An *explicit declaration* is a program statement used for *declaring* the types of variables.
  - E.g.) int count
  - Early PLs (Fortran, Pascal, Ada, C, etc.), Visual Basic, ML, C#, Swift
- An *implicit declaration* is a *default* mechanism for specifying types of variables through *default naming conventions*, rather than declaration statements.
  - Done either by a *compiler* or an interpreter.
  - Basic, Perl, Ruby, JavaScript, PHP
    - E.g.) Perl: $name – a scalar that can store either a string or a numeric value vs. @name – array vs. %name – a hash structure
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)
- Both *explicit*/implicit declarations create *static bindings to types.*

Explicit/Implicit Declaration (cont.)

- *Type inference*:
  - An *implicit type declaration* using *context* to determine types of variables.
  - C# : a variable can be declared with `var` and an *initial value*. The initial value sets the type.
    - E.g.) `var total = 0.0;  var name = “Fred”;`
  - Visual Basic 9.0+, ML, Haskell, and F# use type inference.
    - The *context* of the appearance of a variable determines its type.
    - ML, Haskell, F# $\in$ Functional Prog. Lang.

Dynamic Type Binding

- Dynamic Type Binding:
  - JavaScript, Python, Ruby, PHP, and C# (limited)
  - Specified through an *assignment statement*
    - Neither by a type declaration nor by the naming convention.
    - E.g) In JavaScript:

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

    - In C#: **`dyamic`** reserved word in the declaration e.g.) **`dynamic`** `any;`
  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult.
      - E.g.) In JavaScript, *i*, *x* – scalar numeric variable, *y* – array

        `i = y;` no error detection – type of `i` is changed to an array.

Variable Attributes (cont. from #7)

- 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:
  - the time during which it is bound to a particular memory cell.

Categories of Variables by *Lifetimes*

- Static variable:
  - bound to memory cells *before execution* begins and *remains* bound to the *same memory cell* throughout execution.
  - e.g.) C and C++ **`static`** specifier on a variable in functions:
    - `static int count=0;`
  - Advantages:
    - efficiency (direct addressing),
    - history-sensitive subprogram support with a local variable,
    - no run-time overhead for allocation/deallocation.
  - Disadvantage: lack of flexibility
    - no recursion
    - No sharing of storage among variables.

Categories of Variables by *Lifetimes* (cont.)

- *Stack-Dynamic* variable:

  Storage bindings are created when the variable *declaration statements are elaborated* but whose *types* are *statically bounded*.

  - The *storage allocation* & binding process indicated by the declaration during the *run time*.
  - Stack-dynamic variables are allocated from the run-time *stack*.
  - E.g.) variable declaration at the beginning of a Java method is elaborated when the *method is called* (i.e. at the beginning of execution) and the defined variables are deallocated when the method completes its execution.
  - In Java, C++, C#: default for the defined variables in methods.
  - Its type binding is static.

Categories of Variables by *Lifetimes* (cont.)

- *Stack-Dynamic* variable (cont.):
  - In C++, Java: variable declaration occurs *anywhere*.
  - If not declared at the beginning, the storage binding of *all* of the stack-dynamic variables (excluding those declared in nested blocks) can occur when the function/method *begins execution*.
- Advantage:
  - allows recursion;
  - In non-recursion, all subprograms share the same memory space for their locals, conserving storage.
- Disadvantages:
  - Overhead of allocation and deallocation
  - Slower access due to the indirect addressing.

Categories of Variables by *Lifetimes* (cont.)

- *Explicit Heap-Dynamic* variable:

  Nameless (abstract) memory cells that are allocated/deallocated to/from a heap by *explicit instruction*, specified by the programmer, which takes effect during *execution*.
  - Heap: a collection of storage cells whose organization is highly *disorganized* due to the unpredictability of its use. – a free-list
- Referenced only through *pointers* or *references*,

  e.g.) dynamic objects in C++ (via new `data-type` and delete),

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

- *Type binding is static* since it's done at *compile* time.
- Advantage: provides for *dynamic storage management*
- Disadvantage: inefficient and unreliable

Categories of Variables by *Lifetimes* (cont.)

- *Implicit Heap-Dynamic* variable*:*

  Allocation and deallocation to/from heap storage caused by *assignment statements*.

  - all variables in APL;
  - all strings and arrays in Perl,
  - JavaScript, and PHP
  - E.g.) highs = [74, 84, 86]; regardless the previous use of highs,
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Variable Attributes: Scope

- <u>Definition</u>:

  The *scope* of a variable is the range of statements over which it is *visible*.

- <u>Definition</u>:
  - The *local variables* of a program unit:
    - those that are declared *in that unit.*
  - The *nonlocal variables* of a program unit:
    - those that are visible in the unit but not declared there.
  - *Global variables:* 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 can be *statically determined based on program context*, i.e. before execution.
- To connect a name reference to a variable, the compiler must find the declaration.
- *Search declaration of variable:*
  - Search it locally → then, in increasingly larger enclosing scopes, until one is found for the given name.
- *Static ancestors*: enclosing static scopes (to a specific scope)
- *Static parent*: the nearest static ancestor.
- In the *nested subprogram definitions* of some PLs, it creates *nested static scopes*.
  - e.g.) Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python.

Static Scope (cont.)

- Variables can be hidden from a unit by having a "closer" variable with the *same name*.
- C++ and Ada allow access to these hidden variables.

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

- x is local in sub1.
- x is hidden in sub2 but declared in big.

  – Static ancestor of sub2: big, main
  – Static parent of sub2 is big while sub1 is not in its static ancestry of sub2.

Blocks

- A method of creating *static scopes* inside program unit – from ALGOL 60.
- A section of code is allowed to have *own local variables* whose *scope is minimized within a block*.
- Example in C, C++: in Ada:
  - ```
    void sub() {
      int count;
      while (...) {
       int count;
         count++;
         ...
      }
      …
    }
    ```

```
declare LCL: FLOAT;
    begin

    end
```

- Note: legal in C and C++,
  not legal in Java and C# - too error-prone
```

The **LET** Construct

- Most *functional languages* include some form of `let` construct.
- Two parts in a `let` construct:
  - The 1st part: *binds* names to values
  - The 2nd part: *uses* the names defined in the 1st part
- In Scheme:

```
(LET (
  (name_1 expression_1)
  …
  (name_n expression_n)
   expression
)
```

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

Note:
programs in FL are comprised of expressions, rather than statements.

The **LET** Construct (cont.)

- In ML:

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

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

- The *scope* of a name defined with `let` inside a function definition is from the end of the defining expression to the end of the function. The scope of let can be limited by indenting the following code, which creates a new local scope.

The **LET** Construct (cont.)

- In F#: https://www.tutorialspoint.com/fsharp
  - First part: `let` left_side = expression
    where left_side is either a name
    or a tuple pattern.
  - All that follows is the second part

```
let n1 =
    let n2 = 7
    let n3 = n2 + 3
    n3;;
let n4 = n3 + n1;;
```

Error!

```
let n1 =
  let n2 = 7
  let n3 = n2 + 3
  n3
  printfn "n3: %i" n3;;
```

Output:

```
n3: 10
```

```
let n1 =
  let n2 = 7
  let n3 = n2 + 3
  n3;;

let n4 = n1 + 1;;

printfn "n1: %i" n1;;
```

Output:

```
n1: 10
```

```
let n1 =
  let n2 = 7
  let n3 = n2 + 3

  n3
  n2;;

let n4 = n1 + 1;;

printfn "n1: %i" n1;;
```

Output:

```
n1: 7
```

```
let n1 =
  let n2 = 7
  let n3 = n2 + 3
  n3;;

let n4 = n3 + 1;;

printfn "n4: %i" n4;;
```

Output:

```
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License


/HelloWorld.fs(7,10): error FS0039: The value or constructor 'n3' is not defined
```

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#: (cf. slide-#27)
    - the variable can't be used above its declaration because a variable still must be declared before it can be used.
    - the declaration of the same named variable in a nested block is NOT allowed as a variable in a nesting scope.

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

- In C++, Java, and C#: loop 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++){
    . . .
  }
  . . .
}
```

Global Scope

- In C, C++, PHP, and Python:
  - Variable declarations to appear *outside function definitions are allowed,* except those that include a declaration of a local variable with the same name.
  - E.g.) g is a global variable (in C)

```c
/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

Global Scope (cont.)

- C and C++ have both *declarations* (just attributes) and definitions (attributes and storage) of global data.
  - A declaration outside a function definition specifies that it is defined in a different file.
    - E.g.) **`extern`** `int sum;` (C99)
  - C++: if a global variable is hidden by a local with the same name, it can be accessed using the scope operator (::).
    - E.g.) `::x`, where x is a hidden global var in a function by a local named x.

```
// Function declaration
int myFunction(int, int);

// The main method
int main() {
  int result = myFunction(5, 3); //
  printf("Result is = %d", result);

  return 0;
}

// Function definition
int myFunction(int x, int y) {
  return x + y;
}
```

Global Scope (cont.)

- PHP
  - Variables are implicitly declared when they appear.
  - 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, so implicitly invisible in any function.
    - Global variables can be accessed in a function through the `$GLOBALS` array, using the name of the global as a string literal subscript
    - or by declaring it **`global`**

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

```
$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();
```

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.
  - E.g.) UnboundLocalError

```
''' PL: pg.219 - Global Scope '''

day = "Monday"

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


tester()
----
   print("The global day is:", day)
UnboundLocalError: local variable 'day' referenced before assignment
```

```
''' PL: pg.219 - Global Scope '''

day = "Monday"

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


tester()
```

Cf) **`nonlocal`**: a keyword used to work with variables inside nested functions, where the variable should not belong to the inner function.

```
def myfunc1():
  x = "John"
  def myfunc2():
    nonlocal x
    x = "hello"
  myfunc2()
  return x
x2 = myfunc1()
print("The value of returned x:", x2)
```

```
The value of returned x: Hello
```

Evaluation of Static Scoping

- Static scoping provides a method of *nonlocal access* that 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).
- So, the scope can be determined only at *run time*.
  - APL, SNOBOL4, early Lisp by default.
  - Perl, Common Lisp allow dynamic scoping though its default scoping is static.

- References to variables are connected to declarations by searching back through the *chain of subprogram calls* that forced execution to this point.

Dynamic Scope: Example

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

```
big calls sub2
sub2 uses x
```

- The search process begins with static scoping in the local declaration. If it fails, search the *dynamic parent* or *calling* f^n, until a declaration for x is found.
- `sub2` is called from `big`. – Dynamic parent of `sub2` is `big`.
- Static scoping
  - Reference to x in `sub2` is to `big`'s x
- Dynamic scoping – x in sub2 is dynamic.
  - Reference to x is from either declaration depending on calling sequence

Scope Example: cf.)

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

```
big calls sub1
sub1 calls sub2
sub2 uses x
```

- The search proceeds from sub2 to sub1 where x is found.
  - Dynamic parent of `sub2` is `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`

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 (#19 - #23) are sometimes closely related, but are different concepts.
- Consider a `static` variable in a C or C++ function.
  - A variable is statically bound to the scope of that function and is also statically bound to storage.
  - So, its scope is static and local to the function.
  - But, its lifetime extends over the *entire execution of the program* of which it is a part.
  - E.g.) a variable `sum`
    - the scope: local to `compute`
    - the lifetime: the time during `printheader` executes.

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

Referencing Environments (RE)

- The *referencing environment* of a statement is the *collection of all names* that are *visible* in the statement.
- In a *static-scoped language*, the RE is the *local variables* plus *all visible variables* in *all of its ancestor scopes*.
  - the RE is needed while that statement is being *compiled*, so code and data structures can be created to allow references to variables from other scopes during run time.
- A subprogram is *active* if its execution has *begun* but has *not* yet been *terminated*.
- In a *dynamic-scoped language*, the RE is the *local variables* plus *all visible variables* in *all active subprograms.*

Referencing Environments: Example

- In static-scoped language (Python):

```
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
```

Point	Referencing Environment
1	Local: a, b (of sub1), Global: *g* for reference, but not for assignment.
2	Local: c (of sub2), Global: *g* both for reference and assignment
3	Nonlocal: c (of sub2) Local: *g* (of sub3)

Referencing Environments: Example

- In dynamic-scoped language (C):

```
main calls sub2
sub2 calls sub1
```

```
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, b (of sub1); c (of sub2); d (of main) Hidden: c (of main), b (of sub2)
2	b, c (of sub2); d (of main) Hidden: c (of main)
3	c, d (of main)

Named Constants

- <u>Definition:</u> A *named constant* is a variable that is bound to a value *only once*.
- Advantages: readability and modifiability.
- It's used to parameterize programs.
  - E.g.) In Java: `final int len=100;`

```
int[] intList =  new int[len];
String[] strList =  new  String[len];
```

- 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 statically bound at compile time.

      e.g.) `const int result = 2 * width + 1;`
    - The values of `readonly` named constants are dynamically bound.

Variable Initialization

- <u>Definition</u>: The binding of a variable to a *value* at the time it is bound *to storage* is called *initialization*.
- Often done on the *declaration* statement that creates it.
- If the variable is *statically* bound to storage:
  - binding and initialization occur before run time.
  - the initial value must be specified as a literal or an expression whose only nonliteral operands are named *constants* that have already been defined.
- If the storage binding is dynamic: initialization is also dynamic and the initial values can be any expression.
- e.g.) In Ada: `SUM : FLOAT := 0.0;`
- e.g.) In C++:

```
int sum = 0;
int* ptrSum = &sum;
char name[] = "George Washington Carver";
```

`ptrSum` a pointer variable (*) that stores the address (&) of `sum`.

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.