Chapter 4

Names, Bindings, Type Checking, and Scopes

CHAPTER 5 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 where two primary components are
  - Memory
  - Processor
- Variables characterized by attributes
  - Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

NAMES

- Name : string of characters used to identify some entity in a program
- Common form of names : a letter followed by a string (letters, digits and underscore)
- Some languages (i.e. Fortran90) allow embedded space in names.

Names

- Design issues
  - The form of names
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

NAMES (CONTINUED)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN I: maximum 6
    - COBOL: maximum 30
    - FORTRAN 90 and ANSI C: maximum 31
    - Ada and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

NAMES (CONTINUED)

- Connectors
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do
  - Popular in 70s and 80s, replaced by so-called “camel” notation
    - Using connectors : Sum_Of_Salaries
    - “camel” notation : SumOfSalaries

NAMES (CONTINUED)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Can be avoided using naming conventions
    - worse in C++ and Java because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)
  - C, C++, and Java names are case sensitive
    - The names in other languages are not
    - Difficult to remember the case usage – resulting in difficulties in writing a program

NAMES (CONTINUED)

- 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, 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)*
  - A *reserved word* is a special word that cannot be used as a user-defined name

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

Variables Attributes

- Name - most variables have them
- Address - the memory address with which it is associated
  - 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)
  - Sometimes referred as l-**value**

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. Sometimes referred as **r-value**
- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

The Concept of Binding

- The l-value of a variable is its address
- The r-value of a variable is its value
- A *binding* is an association, such as between an attribute and an entity, 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
  - For example, the asterisk symbol (*) is usually bound to the multiplication operation at language design time.
- Language implementation time-- bind floating point type to a representation
  - A data type, such as **int** in C, is bound to a range of possible values at language implementation time.
- Compile time -- bind a variable to a type in C or Java
  - A variable in C (e.g. X) is bound to a particular data type (e.g. int)
- Load time -- bind a FORTRAN 77 variable to a memory cell (or a C `static` variable)
  - A variable may be bound to a storage cell when the program is loaded into memory.
- Runtime -- bind a nonstatic local variable to a memory cell.
  - That same binding does not happen until run time in some cases, as with variables declared in Java methods

Binding Times

- Consider the following statement:

```
count = count + 5;
```

  - The type of count – bound at compile time
  - The set of possible values of count – bound at compiler design time
  - The meaning of + operator symbol – bound at compile time
  - The internal representation of the literal 5 – bound at compiler design time
  - The value of `count` – bound at execution time

BINDING OF ATTRIBUTES TO VARIABLES

- 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

- Variable must be bound to data type before it can be referenced in a program.
- How is a type specified?
- When does the binding take place?

Static Type Binding

- If **static**, the type may be specified by either an explicit or an implicit declaration
- An *explicit declaration* is a statement in a program that lists variable names and their respective types. E.g. `int var;`
- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program). E.g. in FORTRAN, identifier beginning with I, J, K, L, M or N are implicitly declared at integer type. Otherwise real type.
  - Advantage: writability
  - Disadvantage: reliability

Dynamic Type Binding

- Dynamic Type Binding (JavaScript and PHP)
- Specified through an assignment statement.
- New assignment override the previous assignment

  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

TYPE INFERENCE

- The type of most expressions can be determined without requiring the programmer to specify the types of the variables.
- E.g. (ML programming language):

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

The type floating point (real in ML) is inferred for variable `r`.

Storage Bindings & Lifetime

- Two important processes in binding
  - Allocation - getting a memory cell from some pool of available memory
  - 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

Storage Bindings & Lifetime

Static

- Bound to memory cells before execution begins and remains bound to the same memory cell throughout execution,

  e.g. C static variables.
- Advantages: efficiency (direct addressing), history-sensitive subprogram support
- Disadvantage: lack of flexibility (no recursive subprogram) and storage cannot be shared with other variables

Storage Bindings & Lifetime

Stack-dynamic

- Stack dynamic variables are allocated from the run-time stack.
- Storage bindings are created for variables when their declaration statements are elaborated.

  E.g. local variables in C subprograms and Java methods
- All attributes except storage are statically bound
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Storage Bindings & Lifetime

Explicit heap-dynamic

- Nameless memory cells that are allocated and deallocated by explicit run-time instructions specified by the programmer, which take effect during execution, e.g.

```
int *intnode;
intnode = new int;
….
delete intnode;
```

- 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 that can grow and shrink.
- Disadvantage: difficulty of using pointers and the complexity of storage management.

Storage Bindings & Lifetime

Implicit heap-dynamic

- Bound to heap storage only when they are assigned values (dynamic type binding).
  - all variables in APL; all strings and arrays in Perl and JavaScript i.e. dynamic type binding variables

    E.g.

    ```
    height = [74, 84, 86, 90, 71]
    ```
- Advantage:
  - Highest degree of flexibility
- Disadvantages:
  - Run-time overhead in maintaining the the dynamic attributes
  - Loss of error detection

SCOPE

- The scope of a variable is the range of statements over which it is visible (a variable is visible if it can be referenced in that statement)
- The scope rules of a language determine how references to names are associated with variables
- The *nonlocal variables* of a program unit are those that are visible but not declared there

Static Scope

- Based on program text
- To connect a name reference to a variable, you (or the 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

Static Scope (continued)

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

  E.g.

```
procedure Big is
   X : Integer;
   procedure Sub1 is
      x : Integer;
      begin
      …
      end;
  procedure Sub2 is
      begin
     … X …
      end;
begin
…
end;
```

- C++ and Ada allow access to these hidden variables.
  - In Ada: **`unit.name`**
  - In C++: **`class_name::name`**

Blocks

- A method of creating static scopes inside program units e.g. compound statements E.g.

  C and C++:

```
if (list[i] < list[j]){
                      int temp;
                      temp = list[i];
                      list[i] = list[j];
                      list[j] = temp;
             }
```

```
#include <stdio.h>
int main() {
int n = 1;                          A
{                                   A
   int n = 2;                       B
   printf("%d\n", n);               B
}                                   B
printf("%d\n", n);                  A
}                                   A
```

Blocks (continued)

- The scopes created by blocks are treated exactly like those created by subprograms.
- Referenced to variables in a block that are not declared there are connected to declarations by searching enclosing scopes in order of increasing size.

  Another e.g.

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

- This code is valid in C and C++ but not in Java and C#

Global Scope

- Declared outside any function definitions and accessible throughout the program.
- In C, a local variable can be made global with the use of the reserved work **`extern`**

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 that forced execution to this point

Dynamic Scope

- Consider the following two different call sequences for `Sub2` in the previous **`procedure`** `Big` example.
  - `Big` calls `Sub1`, which calls `Sub2`. X therefore refers to the X declared in `Sub1`.
  - `Sub2` is called directly from `Big`. X therefore refers to the X declared in `Big`.

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

SCOPE

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage:
    - Can't protect local variables from accessibility
    - Inability to statically type check references to nonlocals
    - poor readability – virtually impossible for human reader
    - Longer time to access nonlocal variables compared to static scoping

Scope and Lifetime

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

  E.g.

```
void printheader(){
      …
}

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

- Consider also a `static` variable in a C or C++ function. It is statically bound to the scope of that function. But the lifetime is throughout program execution.

REFERENCING ENVIRONMENTS

- The *referencing environment* of a statement is the collection of all names that are visible in the statement, excluding variables in nonlocal scopes that are hidden by declarations in nearer procedures.
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes

REFERENCING ENVIRONMENTS

```
procedure Example is
   A, B : Integer;
   …
   procedure Sub1 is
   X, Y : Integer;
      begin
      …    <-------------   1
      end;
   procedure Sub2 is
   X : Integer;
   …
      procedure Sub3 is
         X : Integer;
         begin
         …    <----------- 2
         end;
      begin
      …    <-------------- 3
      end;
   begin
   …
   end;
```

At point 1: X and Y of Sub1, A and B of Example

At point 2: X of Sub3, (X of Sub2 is hidden), A and B of Example

At point 3: X of Sub2, A and B of Example

Referencing Environments

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

Referencing Environments

```
void sub1(){
  int a, b;
  …    <---------------1
}

void sub2(){
   int b, c;
   …    <--------------2
   sub1;
}

void main(){
   int c, d;
   …    <--------------3
   sub2();
}
```

What are the referencing environments at points 1, 2 and 3?

At point 1: `a` and `b` of `Sub1`, `c` of `Sub2`, `d` of `main`

At point 2: `b` and `c` of `Sub2`, `d` of `main`

At point 3: `c` and `d` of `main`

Named Constants

- A *named constant* is a variable that is bound to a value only once e.g. `const float pi = 3.142`
- Used to parameterize programs, e.g.

```
void example(){
  int[] intlist = new int[100];
  string[] strList = new String[100];
  …
  for(index=0; index<100; index++) {
    …
  }
…
}
```

Named Constants

- Replacing 100 with a constant `len`

```
void example(){
  final int len = 100
  int[] intlist = new int[len];
  string[] strList = new String[len];
  …
  for(index=0; index<len; index++) {
    …
  }
…
}
```

- Advantages: readability and program reliability

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 (for storage binding)