Chapter 5: Names, Bindings, and Scopes

Principles of Programming Languages

Contents

Introduction

- Imperative languages are abstractions of von Neumann architecture
  – Memory → variables
- Variables characterized by attributes, among of them is **type**
- Scope and lifetime of **variables** are important issues when designing type

Variable Attributes

- Name
- Address
- Value
- Type
- Lifetime
- Scope

Names (Identifiers)

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

Name Forms

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN I: maximum 6
    - FORTRAN 95: maximum 31
    - C89: unlimited but first 31 is significant
    - Ada, Java, C#: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Name Forms

- Most of languages: a letter followed by s string consisting of letters, digits, or underscore
- Special forms:
  - PHP: variables begin with $
  - Perl: special beginning characters denote type $, @, %
  - Ruby: @ instance variable, @@ class variable

Name Forms

- Case sensitivity
  - Many languages, including C-based languages, are case sensitive
  - Disadvantage:
    - readability (names that look alike are different)
    - writability: C++ and Java's predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Special Words

- 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
  - A *reserved word* is a special word that cannot be used as a user-defined name
  - Which one is better? …, but…

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as 6 attributes:
  - Name
  - Address
  - Type
  - Value
  - Lifetime
  - Scope

Variables Attributes

- *Name* - not all variables have them
- *Address* - the memory address with which it is associated
  - A variable may have different addresses at different times during execution
  - l-value
  - Two variable names can be used to access the same memory: aliases
    - created via pointers, reference variables, subprogram parameters
    - harmful to readability

Variables Attributes

- *Type* - determines the range of values of variables and the set of operations that are defined for values of that type
- *Value* - the contents of the memory cell(s) associated with the variable
  - Memory cell here is abstract cell, not physical cell
  - r-value

The Concept of Binding

- A *binding* is an association
  - an attribute and an entity
  - 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
- Language implementation time -- data type is bound to range of possible values
- Compile time -- bind a variable to a data type in Java
- Load time -- bind a variable to a memory cell for C `static` variable
- Link time -- call to library subprogram to its code
- Runtime -- bind a non-static local variable to a memory cell

Example

- Consider the C assignment statement

  `count = count + 5;`

  – Type of `count`

  – Set of possible values of `count`

  – Meaning of +

  – Internal representation of 5

  – Value of `count`

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

Binding

- Let's discuss two types of binding:
  - Type binding: variable to data type
  - Storage binding: variable to its address

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

Explicit/Implicit Declaration

- 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
  - Ex.
    - FORTRAN: I-N: Integer, others real
    - Perl: $a: scalar, @a: array
  - Advantage: writability
  - Disadvantage: reliability (less trouble with Perl)

Dynamic Type Binding

- JavaScript and PHP
- 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

Type Inference

- Ex. ML

```
fun circumf(r) = 3.14159 * r * r;
fun square(x) = x * x;
fun square( x : real) = x * x;
fun square(x) = (x : real) * x;
fun square(x) = x * (x : real);
fun square(x) : real = x * x;
```

Storage Binding & Lifetime

- Storage binding
  - 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

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., all FORTRAN 77 variables, C static variables
- Advantages: efficiency (direct addressing), history-sensitive subprogram support
- Disadvantage: lack of flexibility (no recursion)

Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated.
  - If scalar, all attributes except storage are statically bound
    - local variables in C subprograms and Java methods
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - 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 and JavaScript
- Advantage: flexibility
- 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
- The scope rules of a language determine how references to names are associated with variables
- Local vs. nonlocal variables

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

Scope (continued)

- 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
  - In Ada: `unit.name`
  - In C++: `class_name::name`

Example -- Ada

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin     -- of Sub1
    ...
    end;      -- of Sub1
  procedure Sub2 is
    begin     -- of Sub2
    ... X ...
    end;      -- of Sub2
begin     -- of Big
...
end;      -- of Big
```

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Examples: C or C++ (not Java or C#)

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

- Global variables are defined in outermost block

```
var A, B, C: real; //1
procedure Sub1 (A: real); //2
  var D: real;
  procedure Sub2 (C: real);
     var D: real;
     begin
     … C:= C+B; …
     end;
  begin
  … Sub2(B); …
  end;
begin
… Sub1(A); …
end.
```

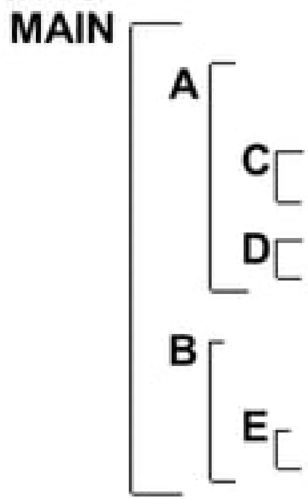
Declaration	Scope
A:real //1	Main
B:real //1	Main,Sub1,Sub2
C:real//1	Main,Sub1
A:real //2	Sub1,Sub2
…	

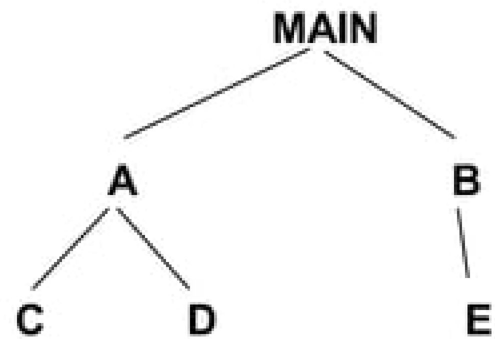
Evaluation of Static Scoping

- Assume MAIN calls A and B

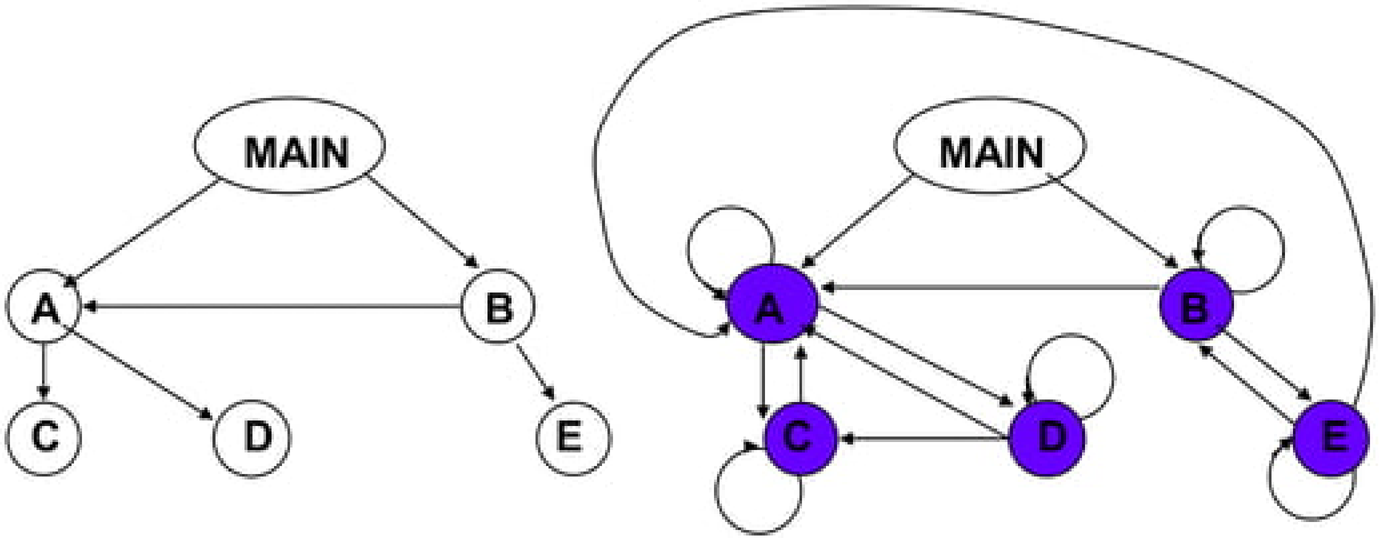
  A calls C and D

  B calls A and E





Static Scope Example



Static Scope (continued)

- Suppose the spec is changed so that D must now access some data in B
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access
- Overall: static scoping often encourages many globals

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
- APL, SNOBOL4, early LISP (Perl, Common LISP)

Example

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin     -- of Sub1
    ...
    end;      -- of Sub1
  procedure Sub2 is
    begin     -- of Sub2
    ... X ...
    end;      -- of Sub2
begin     -- of Big
...
end;      -- of Big
```

```
Big --> Sub1 --> Sub2
```

```
Big --> Sub2
```

Evaluation

- Disadvantages:
  - Local variables are not private anymore, less reliable
  - Cannot statically type check
  - Readability
  - Reading overhead
- Advantage:
  - No need to pass parameters

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Examples:
  - A variable declared in a Java method that contains no method calls
  - A variable declared in C/C++ function with `static` specifier
  - Subprogram calls

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

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin    -- of Sub1
    ...  <--------------------------- 1
    end;     -- of Sub1
  procedure Sub2 is
    X : Integer;
    ...
    procedure Sub3 is
      X: Integer;
      begin    -- of Sub3
      ...   <------------------------ 2
      end;     -- of Sub3
    begin    -- of Sub2
    ...    <------------------------- 3
    end;     -- of Sub2
  begin    -- of Example
  ...    <--------------------------- 4
  end.     -- of Example
```

Referencing Environment

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

```
main --> sub2 --> 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 */
```

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:
  - FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called *initialization*
- Initialization is often done on the declaration statement, e.g., in Java

```
int sum = 0;
```

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

Summary (continued)

- Static scoping provides a simple, reliable, and efficient method of allowing visibility of nonlocal variables in subprograms
- Dynamic scoping provides more flexibility than static scoping but at expense of readability, reliability, and efficiency
- Referencing environment of a statement is the collection of all the variables that are visible to that statement