CSCI 3336 Organization of Programming Languages

NAMES, BINDING, AND SCOPES

Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Checking
- Strong Typing
- Type Compatibility
- Scope and Lifetime
- Referencing Environments
- Named Constants

2

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables characterized by attributes
  - Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

3

Names

- Design issues for names:
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

4

Names (cont' d)

- 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
- Connectors
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do

5

Names (cont' d)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - 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 some other languages are not

6

Names (cont' d)

- 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

7

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

8

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
  - 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)

9

Variables Attributes (cont' d)

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

10

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.

11

Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a FORTRAN 77 variable to a memory cell (or a C `static` variable)
- Runtime -- bind a nonstatic local variable to a memory cell

12

Binding Times – example

Consider the following Java assignment:

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

- The type of count is bound at compile time
- The set of possible values of count is bound at compiler designing time
- The meaning of the operator symbol + is bound at compile time, when the type 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 execution time with this statement.

13

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

14

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

15

Explicit/Implicit Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables

  ```
  int a;
  ```

- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)

  ```
  list = [10.2, 3.5];
  ```

16

Explicit/Implicit Declaration

- For example:
  - FORTRAN, BASIC, and Perl provide implicit declarations
    - Advantage: writability
    - Disadvantage: reliability

17

Static Type Binding

For example, in C# a variable declaration of a variable must include an initial value, whose type is made the type of the variable.

```
var sum = 0;
var total = 0.0;
var name = "Fred"
```

18

Dynamic Type Binding

- 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

19

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)

20

Categories of Variables by Lifetimes (cont' d)

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- If scalar, all attributes except address 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)

21

Categories of Variables by Lifetimes (cont' d)

- *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

22

Categories of Variables by Lifetimes (cont' d)

- Explicit heap-dynamic
  - Consider the following C++ example:

```
int *intnode  // create a pointer
intnode = new int;  // create the heap-dynamic
                    // variable
.....
delete intnode;  //deallocate
```

23

Categories of Variables by Lifetimes (cont' d)

- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl and JavaScript

```
highs = [74, 50, 31];  //javascript example of an
                       // array of length 3
```

- Advantage: flexibility
- Disadvantages:
  - Run-time overhead
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

24

Variable Attributes: Scope

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

25

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

26

Scope (cont'd)

- 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`

27

Blocks

- A method of creating static scopes inside program units
  - Examples:

C and C++:
```
for (...) {
  int index;
  ...
}
```

Ada:
```
declare LCL : FLOAT;
begin
  ...
end
```

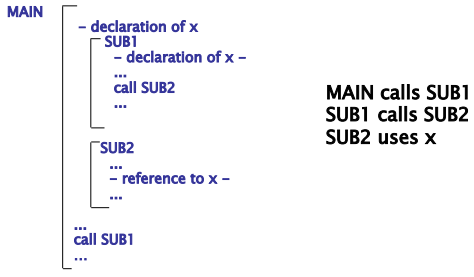
28

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

29

Scope Example



30

Scope Example (cont' d)

- Static scoping
  - Reference to x is to MAIN's x
- Dynamic scoping
  - Reference to x is to SUB1's x
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability, reliability

31

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a `static` variable in a C or C++ function

32

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
- 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

33

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;
```

34

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

35