Chapter 5

Names, Bindings, Type Checking, and Scopes

Chapter 5 Topics

1. Introduction
2. Names
3. Variables
4. The Concept of Binding
5. Type Checking
6. Strong Typing
7. Type Compatibility

Chapter 5 Topics (continued)

8. Scope
9. Scope and Lifetime
10. Referencing Environments
11. Named Constants
5.1 Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory => Data Abstraction
  - Processor => Control Abstraction
- Variables characterized by attributes
  - Name, type, must consider scope, lifetime, type checking, initialization, and type compatibility

5.2 Names

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

5.2 Names (cont.)

- 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
5.2 Names (cont.)

- 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 other languages are not

5.2 Names (cont.)

- Special words
  - An aid to readability; used to delimit or separate statement clauses
  - Def: A keyword is a word that is special only in certain contexts i.e. in Fortran:
    Real VarName (Real is data type followed with a name, therefore Real is a keyword)
    Real = 3.4 (Real is a variable)
  - Disadvantage: poor readability
  - Def: A reserved word is a special word that cannot be used as a user-defined name

5.3 Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  (name, address, value, type, lifetime, and scope)
- Name – not all variables have them (anonymous)
5.3 Variables (cont.)

- Address – the memory address with which it is associated (also called lvalue)
- 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 harmful to readability (program readers must remember all of them)

5.3 Variables (cont.)

- How aliases can be created:
  - Pointers, reference variables, C and C++ unions, (and through parameters – discussed in Chapter 9)
  - Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN
    - EQUIVALENCE in Fortran
  - Replace them with dynamic allocation

5.3 Variables (cont.)

- 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
5.4 The Concept of Binding

- The *l*-value of a variable is its address
- The *r*-value of a variable is its value
- Def: A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Def: Binding time is the time at which a binding takes place.

5.4 The Concept of Binding (cont.)

- Possible binding times:
  - Language design time—e.g., bind operator symbols to operations
  - Language implementation time—e.g., bind floating point type to a representation
  - Compile time—e.g., bind a variable to a type in C or Java
  - Load time—e.g., bind a FORTRAN 77 variable to a memory cell (or a C static variable)
  - Runtime—e.g., bind a nonstatic local variable to a memory cell

5.4 The Concept of Binding (cont.)

- Def: A binding is static if it first occurs before run time and remains unchanged throughout program execution.
- Def: A binding is dynamic if it first occurs during execution or can change during execution of the program.
5.4 The Concept of Binding (cont.)

- **Type Bindings**
  - 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

5.4 The Concept of Binding (cont.)

- **Def:** An **explicit declaration** is a program statement used for declaring the types of variables
- **Def:** An **implicit declaration** is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability

5.4 The Concept of Binding (cont.)

- Dynamic Type Binding (JavaScript and PHP)
- Specified through an assignment statement e.g., JavaScript
  ```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
5.4 The Concept of Binding (cont.)

- Type Inferencing (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference
- Storage Bindings & Lifetime
  - Allocation – getting a cell from some pool of available cells
  - Deallocation – putting a cell back into the pool
- Def: The lifetime of a variable is the time during which it is bound to a particular memory cell

5.4 The Concept of Binding (cont.)

- 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)

5.4 The Concept of Binding (cont.)

- Categories of variables by lifetimes
  - Stack—dynamic—Storage bindings are created for variables when their declaration statements are elaborated.
    - e.g. local variables in C subprograms and Java methods
- If scalar, all attributes except address are statically bound
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)
5.4 The Concept of Binding (cont.)

• 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

5.4 The Concept of Binding (cont.)

• Categories of variables by lifetimes
  - Implicit heap-dynamic—Allocation and deallocation caused by assignment statements e.g. 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

5.5 Type Checking

• Generalize the concept of operands and operators to include subprograms and assignments
  - Type checking is the activity of ensuring that the operands of an operator are of compatible types
    - A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type. This automatic conversion is called a coercion.
    - A type error is the application of an operator to an operand of an inappropriate type
5.5 Type Checking (cont.)

• If all type bindings are static, nearly all type checking can be static

• If type bindings are dynamic, type checking must be dynamic

• Def: A programming language is strongly typed if type errors are always detected

5.6 Strong Typing

• Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

• Language examples:
  – FORTRAN 77 is not: parameters, EQUIVALENCE
  – Pascal is not: variant records
  – C and C++ are not: parameter type checking can be avoided; unions are not type checked
  – Ada is, almost (UNCHECKED CONVERSION is loophole)
    (Java is similar)

5.6 Strong Typing (cont.)

• Coercion rules strongly affect strong typing—they can weaken it considerably (C++ versus Ada)

• Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada
5.7 Type Compatibility

- Our concern is primarily for structured types
- Def: Name type compatibility means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

5.7 Type Compatibility (cont.)

- Structure type compatibility means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
  - Consider a declaration of list node that is recursive

5.7 Type Compatibility (cont.)

- Consider the problem of two structured types:
  - Are two record types compatible if they are structurally the same but use different field names?
  - Are two array types compatible if they are the same except that the subscripts are different? (e.g. [1..10] and [0..9])
  - Are two enumeration types compatible if their components are spelled differently?
  - With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)
5.7 Type Compatibility (cont.)

- Language examples:
  - Pascal: usually structure, but in some cases name is used (formal parameters)
  - C: structure, except for records
  - Ada: restricted form of name
    - Derived types allow types with the same structure to be different
    - Anonymous types are all unique, even in:
      - A, B : array (1..10) of INTEGER
        - A and B are NOT type-compatible

5.8 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

5.8 Scope (cont.)

- 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
5.8 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
  - In Ada: unit.name
  - In C++: class_name::name

5.8 Scope (cont.)

- Blocks
  - A method of creating static scopes inside program units— from ALGOL 60
  - Examples:
    - C and C++: for {...}
      
      ```
      int index;
      ...
      ```
    - Ada: declare LCL : FLOAT;
      
      ```
      begin
      ...
      end
      ```

5.8 Scope (cont.)

- Evaluation of Static Scoping
  - Provides a method for nonlocal access

- Consider an example:
  Assume MAIN calls A and B
  A calls C and D
  B calls A and E
A Programmer could mistakenly call a noncallable subprogram in static scoping (e.g. forward calling). Caught only at run-time.

Desirable calls

Potential call graph

How to restrict access?

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
5.8 Scope (cont.)

- 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

Scope Example

- 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

Scope Example

- declaration of x
  - declaration of x -
    - call SUB2
      ...
  - SUB2
    - reference to x -
      ...
    ...
    - call SUB1
      ...

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x
5.9 Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a static variable in a C or C++ function
  - Scope is static and local to the function
  - Lifetime extends over entire execution

5.10 Referencing Environments

- Def: 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
- We know 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

5.11 Named Constants

- Def: 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 (int xyz[SIZE], a[SIZE][SIZE])
- The binding of values to named constants can be either static (called manifest constants) or dynamic
- Languages:
  - Pascal: literals only
  - FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind
Variable Initialization

• Def: 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., 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
• Strong typing means detecting all type errors