Name, Scope and Binding

In Text: Chapter 5

Outline

- Names
- Variable
- Binding
  - Type bindings
    - Type Checking, type conversion
  - Storage bindings and lifetime
- Scope
- Lifetime vs. Scope
- Referencing Environments

Introduction

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

Names

- Design issues for names:
  - Are names case sensitive?
  - Are special words of the language reserved words or keywords?
- Length
  - If too short, they cannot be connotative
  - Language examples:
    - Fortran I: up to six characters
    - C# and Java: no limit

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not

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
  - A *reserved word* is a special word that cannot be used as a user-defined name
  - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Names (continued)

- 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: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Variable

- A **program variable** is an abstraction of a memory cell or a collection of cells
- It has several attributes
  - Name: A mnemonic character string
  - Address
    - Points to location memory
    - May vary dynamically
  - Type
    - Range of values + legal operations
    - E.g., int type in Java specifies a value range of -2147483648 to 21473647, and arithmetic operations for +, -, *, /, %

Variable

- Scope
  - Range over which the variable is visible
  - Static/dynamic
- Lifetime
  - Time during which the variable is bound to a specific location

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

- Aliases are created via pointers, reference variables, C and C++ unions
- Aliases are harmful to readability (program readers must remember all of them)

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

Variables Attributes (continued)

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

Binding

- A **binding** is an association between two things, such as a name and the thing it names
- **Binding time** is the time at which a binding takes place

Possible Binding Time

- 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 variable to a memory cell (C static variable)
- Runtime
  - Bind a nonstatic local variable to a memory cell (method variables)

An Example

```
count = count + 5
```

- `count` is a local variable
  - When is the type of `count` bound?
  - When is + bound to addition?
  - When the value of `count` is bound?

Static and Dynamic Binding

- A binding is **static** if it occurs before run time and remains unchanged throughout program execution
- A binding is **dynamic** if it occurs during execution or can change during execution of the program

An Example of Dynamic Binding

- In JavaScript and PHP,

```
list = [10.2, 3.5];
… …
list = 47;
```

Static and Dynamic Binding

- As binding time gets earlier:
  - execution efficiency goes up
  - safety goes up
  - flexibility goes down
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later bindings

ONE CANNOT OVERSTATE THE IMPORTANCE OF BINDING TIMES IN PROGRAMMING LANGUAGES

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

- An **explicit declaration** is a program statement that lists variable names and specifies their types
  - var x: int
  - Advantage: safer, cheaper
  - Disadvantage: less flexible

Static Type Binding

- An **implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements
  - First use of variable: X := 1.2;
    - X is a float and will not change afterwards
    - In C# or Swift, a var declaration of a variable must include an initial value, whose type is taken as a type of the variable

Static Type Binding

- Default rules
  - In Fortran, if an undeclared identifier begins with one of the letters I, J, K, L, M, or N, or their lower case versions, it is implicitly declared to be Integer type
  - C# - a variable can be declared with var and an initial value. The initial value sets the type
  - Visual Basic 9.0+, ML, Haskell, and F# use type inferencing. The context of the appearance of a variable determines its type
- Advantage: convenience
- Disadvantage: reliability

Dynamic Type Binding

- The type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name
  - JavaScript, Python, Ruby, PHP, and C# (limited)
- Specified through an assignment statement
  - E.g., list = [10. 2, 3.5]; (JavaScript)
  - Regardless of its previous type, list has the new type of single-dimension array of length 2

Dynamic Type Binding (continued)

- Advantage
  - flexibility (can change type dynamically)
- Disadvantage
  - Type error detection by the compiler is difficult
  - High cost
    - Type checking must be done at runtime
    - Every variable must have a runtime descriptor to maintain the current type
    - The storage used for the value of a variable must be of varying size

Type Checking

- Type checking is the activity of ensuring that the operands of an operator are of compatible types
  - The definition can be generalized to include
    - Subprograms (argument types, return type), and
    - Assignments

Type Checking

- A compatible type is one that
  - is legal for the operator, or
  - is allowed under language rules to be implicitly converted to a legal type
    - The automatic conversion is called (implicit) coercion
    - Mixed mode arithmetic (2 + 3.5)

Type Error

- A type error is the application of an operator to an operand of an inappropriate type
  - 1.5 + "Just say NO! to UVA"

Type Checking

- If all bindings of variables to types are static in a language, then type checking can nearly always be done statically
- Dynamic type binding requires type checking at runtime, which is called **dynamic type checking**
  - Dynamic type binding only allows dynamic type checking

Type Checking

- Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution
  - E.g., C and C++ unions
  - In such cases, type checking must be dynamic
- Even though all variables are statically bound to types, not all type errors can be detected by static type checking

Type Checking

- It is better to detect errors at compile time than at runtime
  - The earlier correction is usually less costly
- Penalty for static checking
  - Reduced programmer flexibility
  - Fewer shortcuts and tricks are possible

Strong Typing

- A programming language is **strongly typed** if type errors are always detected
- Advantages of strong typing
  - Ability to detect all misuses of variables that result in type errors

Language Comparison for Strong Typing

- FORTRAN 95 is not strongly typed
  - the use of Equivalence between variables of different types allows a variable of one type to refer to a value of a different type
- C and C++ are not strongly typed
  - Both include union types, which are not type checked

Language Comparison for Strong Typing

- Ada, Java, and C# are almost strongly typed
  - It allows programmers to breach the type-checking rules by specially requesting that type checking be suspended for a particular type conversion
- ML is strongly typed

Coercion Rules

- Coercion rules can weaken strong typing
  - E.g., int a = 3, b = 5;
float d = 4.5;
  - If a developer meant to type a + b, but mistakenly typed a + d, the error would not be detected by the compiler due to coercion
- Languages with more coercion are less reliable than those with little coercion
  - Reliability comparison
    - Fortran/C/C++ < Ada
    - C++ < Java/C#

Type Compatibility

- The rules dictate the type of operands that are acceptable for each operator and thereby specify the possible type errors of the language
- Type rules are called compatibility because in some cases, the type of an operand can be implicitly converted by the compiler or runtime system to make it acceptable to the operator

Type Equivalence

- A strict form of type compatibility—compatibility without coercion
- Two approaches to defining type equivalence
  – Name type equivalence (Type equivalence by name)
  – Structure type equivalence (Type equivalence by structure)

Name Type Equivalence

- Two variables have equivalent types if they are defined in the same declaration or in declarations using the same type name
  – Ex. 1, int a, b;
  – Ex. 2, int a; int b;

Name Type Equivalence

- Easy to implement but is more restrictive
  – In Ada

```
type Indextype is 1..100;
count : Integer;
index: IndexType;
```

  - The type of count is a subrange of the integers, which is not equivalent to the integer type
  - The two variables cannot be assigned to each other

Name Type Equivalence

– In Pascal

```
Type X: array[1..5] of integer
Y: X;
Procedure K(J: array[1..5] of integer
…
K(Y)   /* Y incompatible with J */
```

  - Although J and X have the same type structure, they are considered as two types
  - Y cannot be passed as a valid parameter to call K

Structure Type Equivalence

- Two variables have equivalent types if their types have identical structures
  – Ex 1., type celsius = float;
  fahrenheit = float;
  – The two types are considered equivalent

Structure Type Equivalence

- More flexible, but harder to implement
  - The entire structures of two types must be compared
- Developers are not allowed to differentiate between types with the same structure