# Chapter 5

Names, Bindings, Type Checking, and Scopes

# Chapter 5 Topics

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

# Introduction

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

# Names

- Design issues for names:
  - Maximum length?
  - 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)

- 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 most other languages 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, 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 - 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)

# 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
- *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
- 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

# Possible Binding Times - Example

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

- The type of count is bound at compile time
- The set of possible values of count is bound at design time
- The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.
- The internal representation of the literal 5 is bound at design time.
- The value of count is bound at execution time with this statement.

# 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

# Type Binding

The two important aspect of type binding are:

- 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 at compile time
- 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 (the first appearance of the variable in the program)
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability (less trouble with Perl)

# 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, interpretation, and dynamic storage allocation)
    - Type error detection by the compiler is difficult (less reliable)
    - These languages are usually implemented using pure interpretation.

# Type Inferencing

- Type Inferencing (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference AI??

# Storage Binding and Lifetime

- Storage Bindings & Lifetime
  - 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), storage can not be shared among variables.

# Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated (at run time).
- Elaboration takes place when execution reaches the code to which the declaration is attached.
- Stack-dynamic variables are allocated from the run-time stack.
- If scalar, all attributes except address are statically bound
  - local variables in C subprograms and Java methods

# Categories of Variables by Lifetimes

Stack-Dynamic Variables (continued)

- For example, the variable declaration at the beginning of a Java method are elaborated when the method is called and are deallocated when the method completes its execution.
- Advantage: allows recursion; conserves storage (subprograms share the same memory space for their locals)
- 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
- An EHD variable is bound to a type at compile time, so the binding to the type is static.
- Advantage: provides for dynamic storage management
- Disadvantage: inefficient because of the complexity of heap data structure and unreliable (difficult to use pointers and references)

# 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
- All the attributes of a variable are bound every time they are assigned!
- Advantage: flexibility
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

# Type Checking (6.10)

- 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
- Dynamic type binding requires type checking at run time which is called *dynamic type checking*.

# Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic (type checking at runtime)
- A programming language is *strongly typed* if type errors are always detected.
  - This requires that the types can be determined, either at compile time or at run time.

# 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` (allows a variable of one type to refer to a value of a different type)
  - Pascal is not: variant records
  - C and C++ are not: unions are not type checked
  - Ada is, almost (`UNCHECKED_CONVERSION` is loophole)
  - Java is similar to Ada

# Strong Typing (continued)

- 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

# Name Type Compatibility

- There are two approaches for type equivalence:
  - Name type equivalence (or compatibility)
  - Structure type equivalence
- *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

# Structure Type Compatibility

- *Structure type compatibility* means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement and disallows differentiating between types with the same structure.

# Type Compatibility (continued)

- 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])
  - With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

# 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
  - Two classes of scopes: 1) static 2) dynamic

# Static Scope

- The scope of a variable can be statically determined, prior to execution
- Based on program textual layout
- 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`**

# 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
```

# Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Example in C:

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

- Note: This code is legal in C and C++, but not in Java and C# - too error-prone

# Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, a variable still must be declared before it can be used

# Declaration Order (continued)

- In C++, Java, and C#, variables can be declared in `for` statements
  - The scope of such variables is restricted to the `for` construct

# Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions

- C and C++have both declarations (just attributes) and definitions (attributes and storage)

# Global Scope (continued)

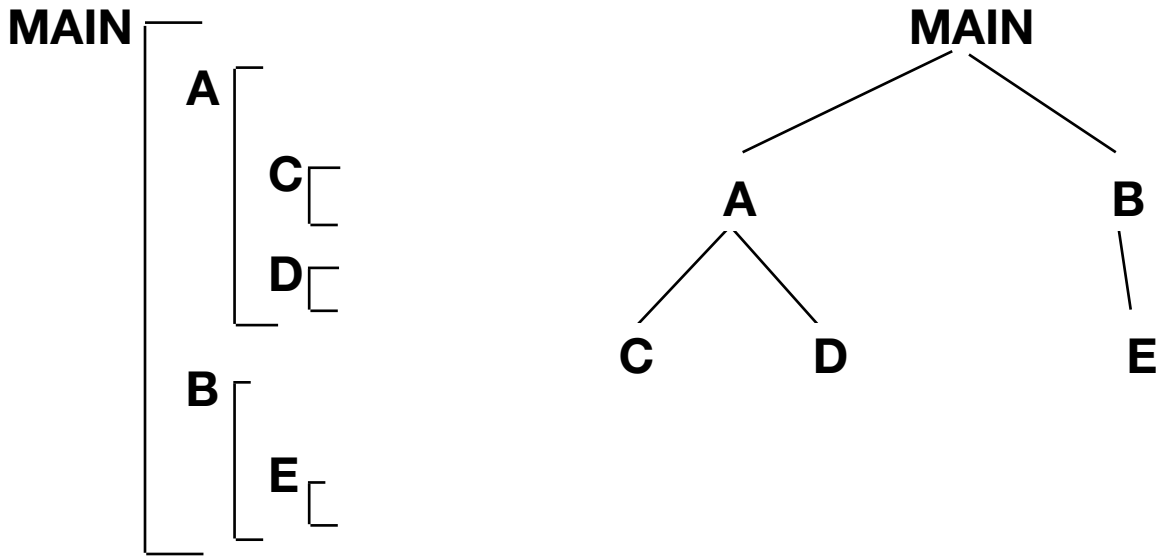
- PHP
  - The scope of a variable (implicitly) declared in a function is local to the function
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`

# Global Scope (continued)

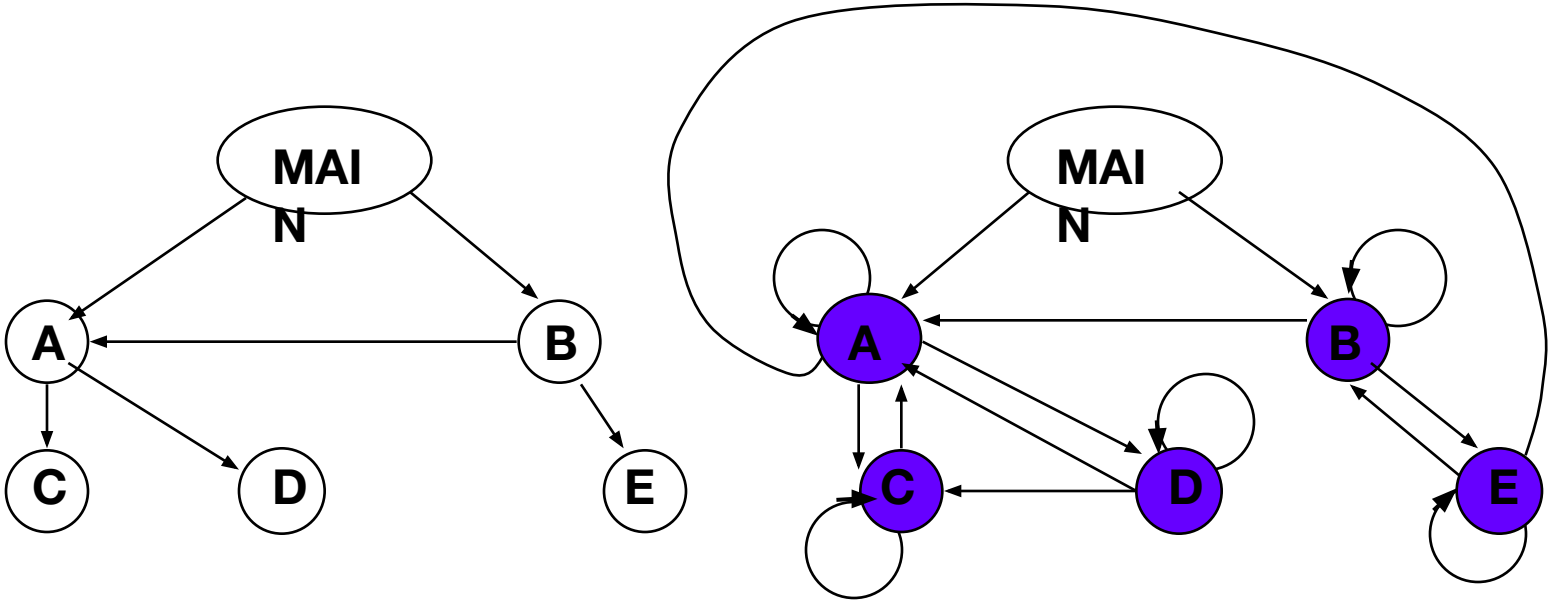
- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

# Evaluation of Static Scoping

- Assume MAIN calls A and B
  A calls C and D
  B calls E



# Static Scope Example



# Static Scope (continued)

- Suppose the specification 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 global variables.

# 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

```
MAIN
    - declaration of x
      SUB1
        - declaration of x -
        ...
        call SUB2
        ...

      SUB2
        ...
        - reference to x -
        ...

    ...
    call SUB1
    ...
```

**MAIN calls**
**SUB1**
**SUB1 calls**
**SUB2**
**SUB2 uses x**

# 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 and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a `static` variable defined in a C or C++ function  lifetime: start to the end of the whole program; Scope: just the function it is defined in

# 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

# Named Constants

- A *named constant* is a variable that is bound to a value only once
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- In static, constant expressions can contain only previously declared named constant, constant values, and operators.
- In dynamic, expressions may contain variables to be assigned to constant in the declaration.
- Languages:
  - FORTRAN 90: constant-valued expressions (only static binding of values)
  - Ada, C++, and Java: Allow dynamic binding of values to named constants.

# 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
- Variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors