Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables, are the abstractions for memory cells, 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 reserved words or keywords?
    - Predefined words?

Names (cont’d)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN 95: maximum of 31
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C#, Ada, and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

This lecture covers review questions: 1-12
And problems: 1-6

Dr. Amirhossein Chinaei, ECE, UPRM Spring 2010

***Some slides are adapted from the Sebesta’s textbook
Names (cont’d)

- Special characters
- Examples:
  - **PHP**: all variable names must begin with $  
  - **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

Names (cont’d)

- Case sensitivity
  - **Disadvantage**: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in most others are not
    - Worse in **C++**, **Java**, and **C#** because predefined names are mixed case
      (e.g., `IndexOutOfBoundsException`)

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
      - Potential problem with reserved words: If there are too many, many collisions occur (e.g., **COBOL** has 300 reserved words!)

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** – is the memory address with which the variable 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 (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
  - The **l-value** of a variable is its address
  - The **r-value** of a variable is its value
- **Abstract memory cell** – the physical cell or collection of cells associated with a variable

The Concept of Binding

**Definition**: 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**: 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**: bind a **C** or **C++ static variable** to a memory cell
- **Link time**: bind a library variable to a memory cell
- **Run time**: bind a non-static local variable to a memory cell
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

- 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 (the first appearance of the variable in the program).
- **FORTRAN, BASIC, and Perl** provide implicit declarations (**Fortran** has both explicit and implicit):
  - **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**
  
  ```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