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 reserved words or keywords?
* Length
  + If too short they cannot be connotative
  + Language examples:
    - C99: No limit but only the first 63 are significant; also external names are limited to a length of 31
* 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
* Case sensitivity
  + Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
* 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

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
* Name – not all variables have them
  + pointers
* Address – the memory address with which it is associated
  + A variable may have different addresses at different times during execution
  + 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 be able to tell the difference between aliases)
* 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
* Type – Determines the range of values of variables and the set of operations that are available for values of that type; in the case of floating point types, type also determines the precision
  + Integer: 4 bytes, 32 bits
    - Range is -2^31 to 2^31 – 1

The Concept of Binding

* A *binding* is an association between an entity and an attribute, such as between a variable ad its type or value, 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 C or C++ static variable to a memory cell
  + **Runtime** – 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 (the type) remains unchanged throughout program execution
  + Declaring something in a language like C, C++, or Java
* 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 implicit declaration.

Explicit / Implicit Declaration

* An **explicit declaration** is a program statement used for declaring the types of variables
  + C, C++, Java
* An **implicit declaration** is a default mechanism for specifying types of variables through default conventions, rather than declaration statements
  + Fortran (integers start with i, j, k, l, m, n)
* Some languages use **type inferencing** to determine types of variables based on context
  + 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.

Dynamic Type Binding

* **Dynamic Type Binding** (JavaScript, Python, Ruby, PHP, and C# (limited))
* Specified through an assignment statement

Storage Bindings & Lifetime

* Allocation – Getting a cell from a 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
* Four categories of variable by lifetime
  + Static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic

Categories of Variables by Lifetimes

* **Static** – Bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, eg. C and C++ static variables
* **Stack-Dynamic** – Storage bindings are created for variables when their declaration statements are *elaborated*
  + If scalar, all attributes except address are statically bound
  + Advantage – Allows recursion, conserves storage
  + Disadvantage – Overhead of allocation and deallocation, subprograms cannot be history sensitive, inefficient references (indirect addressing)
* **Explicit Heap-Dynamic** – Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
  + Referenced only through pointers or references
  + Advantage: Provides for dynamic storage management
  + Disadvantage: Inefficient and unreliable
* **Implicit Heap-Dynamic** – Allocation and deallocation caused by assignment statements
  + Advantage: Flexibility (generic code)
  + Disadvantage: Inefficient because all attributes are dynamic, loss of error detection

Scope

* The **scope** is the range of statements over which it is visible
* The *local variables* of a program unit are those that are declared in that unit
* The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
* *Global variables* are a special category of nonlocal variables
* The scope rules of a language determine how references to names are associated with variables
* **Static** – Based on what you see
* **Dynamic** – Based on execution

Static Scope

* Based on program text
* To connect a name reference to a variable, you (or the compiler) must find the declaration