CS 350: Programming Language Design

Lecture 9

Binding is an association between a name and the thing which is named.

* Between an entity and attribute
* Between a variable and its type or value
* Between an operation and a symbol

Binding Time is the time at which binding occurs.

* Language design time – bind operator symbols to their 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

* A binding is static if it first occurs before run time and remains unchanged throughout program execution.

Dynamic

* A binding is dynamic if it first occurs during execution or can change during execution of the program

Type Bindings

* Before a variable can be referenced, it must be bound to a data type
* We don’t know
  + How type is specified
  + When the binding takes place

Static Type Bindings

* If static, the type may be specified by either an explicit or an implicit declaration
  + An explicit declaration is a program statement used for declaring the types of variables ( strongly typed languages) (data Type varName)
* An implicit declaration is a default mechanism for specifying types of variables other than declaration statements
  + Strong writability, low reliability

Dynamic Type Binding

* Variable type is not specified by a declaration statement
* Variable may be changed during run-time
* A type is determined when a variable is assigned a value
  + May affect address binding
* This allows flexibility
* At the cost of memory costs, type error detection being hard, and slower implementation (because of interpreters)

Variable storage binding

* Storage bindings
  + Allocation – getting a cell from some pool of available cells
  + Deallocation – putting a used cell back into the pool
* The lifetime of a variable is the time during which it is bound to a particular memory cell
* Categories
  + Static Variables
  + Stack Dynamic
  + Heap-Dynamic (Explicit and Implicit)

Static Variables

* Bound to memory cells before execution begins and remains bound to the same memory cell until program execution terminates.
* Advantages
  + High-efficiency
  + History sensitive subprogram support
* Disadvantage
  + Low flexibility (no recursion)

Stack-Dynamic Variables

* Storage bindings are created for variables when their declaration statements are elaborated
  + A declaration is elaborated when the executable code associated with it is executed
* If scalar, all attributes except address are statically bound
  + Local variables in C and Java
* Advantage: allows recursion, conserves storage
* Disadvantages
  + Overhead of allocation and deallocation
  + Subprograms cannot be history sensitive
  + Inefficient references (indirect addressing)

Heap-dynamic variables

* What is heap?
* Allocation specified by the programmer
* Takes effect during execution
* Advantages
  + Dynamic storage management
  + Flexibility

Explicit Heap

* Allocated and deallocated by explicit directives
* Referenced only through pointers or references
* Disadvantage
  + Inefficient cost of allocations, references, and de-allocations
  + Explicit de-allocation is unreliable

Implicit Heap

* Allocation and deallocation caused by assignment statements
  + All attributes are bound every time they are assigned
* Eg
  + All strings and arrays in Perl, JavaScript, and PHP
* Advantage
  + Flexibility
* Disadvantages
  + Inefficient – introduce the run-time overhead of maintaining all the dynamic attributes
  + Loss of error detection by the compiler

Scope and Lifetime

The scope of a variable is the range of statements over which it is visible

* The local variables of a program unit are available only in the unit
* Non-local 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

The lifetime of a variables is the time during which it is bound to a memory cell

Scope and lifetime are sometimes related, but they are different concepts

* Scope is textual or spatial, whereas lifetime is temporal

Static Scope

* Scope may be statically determined prior to execution
* To connect a name reference to a variable, you must search for the declaration of the variables
* Once your program finds a name reference, the search goes as follows
  + Search declaration, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
  + Enclosing static scopes are called its static ancestors, the nearest static ancestor is called a static parent

Blocks

Some languages allow a section of code to have its own local variables whose scope is minimized.

* Defined by blocks, can be nested
* Treated like sub-programs -> variables are stack dynamic
* Storage is allocated when the block is entered and deallocated when the block is exited
* May hide another variable in a larger enclosing scope

Global Scope

* C, C++, PHP, JS, 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 functions definitions
* C and C++
  + Definitions outside functions in a file create global variables
  + A global variable in C is implicitly visible in all subsequent functions in that file, except those that include a declaration of a local variable with the same name
    - In which case it is accessible with the scope operator

Variable shadowing

* Variables can be hidden from a unit by having a closer variable with the same name
* Can be accessed with selective references
  + C++ uses the scope resolution operator ( :: )

Static Scoping works well in many situations, however it can provide too much access which is dangerous, as our program grows the base foundation breaks down and local variables may become global variables, subprograms also gravitate toward globalness rather than nestedness.

Dynamic Scoping is based on sequences of program units, but not their layout. This indicates temporal biasedness vs. spatial. References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

Ultimately, dynamic scoping is convenient but it is less reliable, hard to determine static attributes, it will take longer time to resolve and knowing sequence will become necessary for understanding a program, which causes poor readability.