CS2510
Modern Programming Languages

Lecture 5

Names, Bindings and Scopes

Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative languages are abstractions of the von Neumann architecture
  - Memory
  - Processor
- Variables are characterised by attributes
  - Scope, lifetime, type checking, initialisation, and type compatibility

Names

- Design issues for names:
  - Are names case-sensitive?
  - Are special words reserved words or keywords?

Names (continued)

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

Names (continued)

Special characters

- PHP:
  - variable names must begin with dollar signs
- Perl:
  - 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 (continued)

Case sensitivity

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

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
- 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 have 6 attributes:
  1. Name
  2. Address
  3. Value
  4. Type
  5. Lifetime
  6. 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 because 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;
- Floating point: type also determines the precision

Value

- Contents of location (memory cell(s)) associated with a variable
- 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

Binding

- Association between entity and attribute(s)
- Examples:
  - A variable and its type or value
  - 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 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

- *Explicit declaration*
  - Program statement declares types of variables
- *Implicit declaration*
  - Conventions to establish types of variables
  - No explicit reference to types
  - Examples: Fortran, Perl, Ruby, JavaScript, and PHP
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Explicit/Implicit Declaration (cont'd)

Some languages use *type inferencing* to determine types of variables

- Context enables compiler to work out types
- C#
  - Variables can be declared with **`var`** and an initial value
  - The initial value sets the type
- Visual BASIC 9.0+, ML, Haskell, F#, and Go
  - Use type inferencing
  - Where a variable appears determines its type

Dynamic Type Binding

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

Variable Attributes (continued)

Storage bindings and 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
- Remains bound to same memory cell throughout execution
- Examples: C and C++ `static` variables in functions
- Advantages:
  - Efficiency (direct addressing),
  - History-sensitive subprogram support
- Disadvantage:
  - Lack of flexibility (no recursion)

Categories of Variables by Lifetimes

Stack-dynamic

- Storage bindings are created for variables when their declaration statements are *elaborated*
  - A declaration is elaborated when executable code associated with it is executed
- If scalar, all attributes except address are statically bound
  - local variables in C subprograms (not declared **`static`**) and Java methods
- Advantage:
  - Allows recursion
  - Saves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Categories of Variables by Lifetimes

Explicit heap-dynamic

- Allocated/deallocated by explicit directives
- Specified by the programmer
  - Take effect during execution
- Referenced only through pointers or references
- Dynamic objects in C++ (via **`new`** and **`delete`**)
- Advantage: dynamic storage management
- Disadvantage: inefficient and unreliable

Categories of Variables by Lifetimes

Implicit heap-dynamic

- Allocation and deallocation caused by assignment statements
- Strings & arrays in Perl, JavaScript, and PHP
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Variable Attributes: Scope

- **Scope** of a variable: range of statements it can be used
- **Local variables (**of a program unit): declared in that unit
- **Nonlocal variables (**of a program unit): visible in the unit but declared elsewhere
- **Global variables**: special category of nonlocal variables
- **Scope rules** of a language determine how references to names are associated with variables

Static Scope

- Follows what is stated in the code (not in the execution)
- To connect a name reference to a variable, you/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**
  - Nearest static ancestor is called a **static parent**
- Some languages allow nested subprogram definitions
  - These create nested static scopes
  - Examples: JavaScript, Python, and others

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name
- Ada allows access to these "hidden" variables
  - E.g., `unit.name`

Blocks

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

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

- **Important**:
  - legal in C and C++, but not in Java and C# as it is 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

The LET Construct

- Most functional languages include some form of **`let`** construct
- A **`let`** construct has two parts
  - First part binds names to values
  - Second part uses the names defined in the first part
- In Scheme:

```
(LET (
   (name₁ expression₁)
   …
   (nameₙ expressionₙ)
)
```

The LET Construct (continued)

- In ML:

```
let
  val name_1 = expression_1
  …
  val name_n = expression_n
in
 expression
end;
```

- In F#:
  - First part: **`let`** left_side = expression
  - (left_side is either a name or a tuple pattern)
  - All that follows is the second part

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 program structure consisting 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)
  – A declaration outside a function definition specifies that it is defined in another file

Global Scope (continued)

PHP

- Programs embedded in HTML markup documents, in a number of fragments, statements & function definitions
- Variables (implicitly) declared in a function are local to that function
- 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

- Works well in many situations
- Problems:
  - In most cases, too much access is possible
  - As a program evolves, initial structure is destroyed and local variables often become global;
  - Subprograms tend to become global, rather than nested

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables connected to declarations
- One needs to search back through the chain of subprogram calls to the point variable is referred

Scope Example

```
function big() {
    function sub1() {
        var x = 7;
    }
    function sub2() {
        var y = x;
        var z = 3;
    }
    var x = 3;
}
```

Assume that
- big calls sub1
- sub1 calls sub2

sub2 uses x

- Static scoping
  - Reference to x in sub2 is to big's x
- Dynamic scoping
  - Reference to x in sub2 is to sub1's x

Scope Example

Evaluation of Dynamic Scoping:

- Advantage: convenience
- Disadvantages:
  - While a subprogram is executing, its variables are visible to all subprograms it calls
  - Impossible to statically type check
  - Poor readability: impossible to statically determine the type of a variable

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a `static` variable in a C or C++ function

Referencing Environments

- **Referencing environment** of a statement
  - Collection of all names visible in the statement
- In static-scoped languages:
  - Local variables plus all visible variables in all enclosing scopes
- In dynamic-scoped languages:
  - Referencing environment contains local variables plus all visible variables in all **active subprograms**
  - A subprogram is **active** if its execution has begun but has not yet terminated

Named Constants

Named constant

- A variable bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterise programs
- Binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - C++ & Java: expressions of any kind, dynamically bound
  - C# has two kinds, **`readonly`** and **`const`**
    - Values of **`const`** named constants bound at compile time
    - Values of **`readonly`** named constants dynamically bound

Summary

- Case sensitivity and relationship of names to special words represent design issues of names
- Variables characterised by name, address, value, type, lifetime, scope
- Binding: association of attributes with program entities
- Scalar variables:
  - static,
  - stack dynamic,
  - explicit heap dynamic,
  - implicit heap dynamic
- Strong typing detects all type errors