COS 301

Programming Languages

Sebesta Chapter 5

Names, Binding and Scope

The beginning of wisdom is to call
things by their right names

Chinese Proverb

Topics

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

Abstraction and the Von Neumann Machine

- Imperative languages are abstractions of the von Neumann architecture
  - Variables are an abstraction for memory; a one-dimensional array of cells
  - Some abstractions are close to the machine, for example an n-bit integer in an n-bit machine
  - Others require a mapping function, such as two or higher dimensional arrays
- Variables are characterized by attributes
  - One of the most important is the type
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

Language groups

- We will consider a broad division of imperative and imperative/OO languages in discussing names because the treatment of names falls broadly into two classes:
  - C-Like: C, C++, Java, etc (AKA the curly-brace languages)
  - Pascal-Like: Pascal, Ada, Modula, etc.
- Other major language families
  - Fortran
  - Scripting

Names

- The terms *name* and *identifier* are roughly synonymous
  - Names are associated with variables, subprograms, modules, classes, parameters, types and other constructs
- Design issues for names:
  - Allowable length
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names - Length

- 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

Underscores

- Underscores
  - Many languages allow underscores (and possibly a few other characters)
  - Some languages allow variable names to contain only underscores
    - You can write a complete program with variables _, __, ___, _____, etc
  - Often used to separate words in meaningful variable names: ex. grand_total_score
  - The way that names are formed is often part of a language culture
    - GrandTotalScore (camel case) grand_total_score
    - GRANDTOTALSCORE GRAND_TOTAL_SCORE
- In C one convention is to use uppercase names for constants, lower or mixed case names for variables

Special Characters

- Most languages allow a few special characters in names (_,!,@,$, etc.)
- Some languages require them
  - PHP: all variable names must begin with $. Names without $ are constants
  - Perl: all variable names begin with $, @, or %, which specify the variable's type
  - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Special Characters

- Languages typically allow only a few characters other A-Z,a-z, 0-9, with syntactic restrictions
  - Cobol allows hyphens in names (what sort of parsing problem does this present?)
  - C allows unrestricted use of the underscore
  - Fortran prior to Fortran 90 allowed embedded spaces
    - Grand Total Score is the same as GrandTotalScore

Implicit Typing

- Many BASICs use type declaration chars: foo$ is a string, foo! is a single precision float
- In FORTRAN 77 variables beginning with I,J,K,L, M are implicitly integers, otherwise REAL is assumed
  - COUNT is a real
  - KOUNT is an integer
  - IMPLICIT statement allows mods to this rule
- QuickBasic has DEFINT, DEFSNG, etc. statements to define implicit typing

Character Case

- Case sensitivity
  - Languages that allow only upper-case are less readable; no longer an issue
  - Case sensitive languages have a readability issue: names that look alike are different
    - Names in the C-based languages are case sensitive
    - Names in many others are not
    - PHP has case-sensitive variable names, case-insensitive function names
  - Case sensitive languages that do not require variable declarations also have a writability issue: it's too easy to accidentally create a new variable

Keywords and Reserved Words

- Keywords
  - 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)*
      - In Fortran this is legal:

```
Real Integer
Integer Real
```

      - Pascal: `const true = false`
  - PL/I had no reserved words!

```
If if = then then else = if else if = else
```

Keywords and Reserved Words

- Reserved Words
  - 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)

The Concept of Binding

- A binding is an association between an entity (such as a variable) and a property (such as its value).
  - A binding is *static* if the association occurs before run-time.
  - A binding is *dynamic* if the association occurs at run-time.
- Static binding is used by most compiled languages
- Interpreted languages often delay name resolution until runtime

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a tuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Variable Names

- Name - not all variables have them
  - Those that do not are typically "heap-dynamic" variables
- A variable name is a binding of a name to a *memory address*

Variable Addresses

- 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 decrease readability (program readers must remember all of them)

Variable Type

- *Type* - determines the range of values of variables and the set of operations that are defined for values of that type

Variable Value

- The contents of the memory cell associated with the variable
  - Note that "memory cell" is an abstract concept
  - Double-precision floats may occupy 8 bytes of physical storage in most implementations but we think of them as occupying 1 memory cell

L-values and R-values

- A distinction in the use of variable names first introduced in Algol 68
- L-value - use of a variable name to denote its address.
- R-value - use of a variable name to denote its value.

  Ex: x = x + 1
  - Store into memory at address of x the value of x plus one.
- On the LHS x denotes an address while on the RHS it denotes a value

Explicit Dereferencing and Pointers

- Some languages support/require explicit dereferencing.

  ML: `x := !y + 1`

  C: `x = *y + 1`
- C pointers

```
int x,y;
int *p;        ; p is a pointer to int
x = *p;
*p = y;
```

- Note that C is liberal about whitespace:

```
x *y;    /* y is a pointer to type x */
x * y;   /* same as above */
```

The Concept of Binding

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 C or C++ `static` variable to a memory cell)
- Runtime -- bind a nonstatic local variable to a memory cell

Example: Java Assignment Statement

- In the simple statement

  `count = count + 1;`
- Some of the bindings and binding times are
  - Type of count is bound at compile time
  - Set of possible values is bound at design time (but in C, at implementation time)
  - Meaning of + is bound at compile time, when types of operands are known
  - Internal representation of the literal 1 is bound at compiler design time
  - Value of count is bound at runtime 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

- 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 and BASIC provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability
- Vestiges in modern languages:
  - Fortran: **`Implicit None`**
  - Visual Basic: **`Option Explicit`**

Dynamic Type Binding

- Perl symbols $, @, % divide variables into three namespaces: scalars, arrays and hashes
  - Within each namespace type binding is dynamic
- JavaScript and PHP use dynamic type binding
- 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 and interpretation)
    - Type error detection by the compiler is difficult

Cost of Dynamic Type Binding

- Type checking at run time
- Every variable needs a runtime descriptor to maintain current type
- Storage for variable is of varying size
  - Implying heap storage and runtime garbage collection
- Languages with dynamic type binding are usually implemented as pure interpreters
  - Cost of interpreter may hide cost of dynamic type checking

Type Inferencing

- In ML types are determined (by the compiler) from the context of the reference

```
fun circumference(r) = 3.14159 * r * r;
```

- Here the compiler can deduce type *real* and produce a *real* result

```
fun cube(x) = x * x * x;
```

- Here the compiler will deduce type *int*.
  - The default numeric type is int.
  - If we call cube with a real number we have a runtime error.

```
cube(3.14159);
```

- Solution

```
fun cube(x) : real = x * x * x;
```

Storage Bindings and Lifetime

- Allocation – allocating memory from some pool of available memory
- Deallocation - putting a cell back into the pool
- The lifetime of a variable begins when memory is allocated and ends when memory is deallocated

Four categories of lifetime

- For scalar variables
  - Static
  - Stack-dynamic
  - Explicit heap-dynamic
  - Implicit heap-dynamic

Static Variables

- Static variables are bound to addresses before execution begins and remain bound until program terminates
  - Most global variables are static
  - Some languages such C/C++ support local static variables
- Advantages:
  - efficiency (direct addressing), history-sensitive subprogram support
- Disadvantage: lack of flexibility
  - A language with ONLY static variables does not support recursion

The `static` Keyword

- In C and C++ a variable declared as static inside a function retains its value over function invocations:

```
int hitcount() {
  static int count  = 0;
  return ++count;
}
```

- But inside C#,C++ and Java class definitions the static modifier means that the variable is a class variable rather than an instance variable

Global Variables in Java

- Java does not allow declaration of variables outside of any scope, so C/C++ style globals cannot be used
- Solution is to use public static variables in a class

```
public class GlobalData {
  public static int usercount = 0;
  public static long hitcount = 0;
}
```

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
- For scalar variables, all attributes except address are statically bound
  - local variables in C subprograms and Java methods
  - Note that while actual memory address is not statically bound a relative address (stack offset) is statically bound
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Explicit Heap Dynamic Variables

- *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

```
int *intnode;        //c or c++
intnode = new int;
. . .
delete intnode;
```

- With explicit heap dynamic variables we do not need to incur the overhead of garbage collection

Explicit Heap Dynamic variables

- The main disadvantages are
  - Failure to deallocate storage results in memory leaks
  - Other difficulties in using variables correctly (dangling pointers, aliasing)
  - Heap fragmentation
- C# provides implicit deallocation
- C++ style pointers can be used but header of any method that defines a pointer must include the keyword **`unsafe`**

Implicit Heap Dynamic Variables

- *Implicit heap-dynamic*—bound to heap storage by by assignment statements
  - All attributes are bound by at this time
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Names are in effect just holders for pointers
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

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
- A common source of error is accidental modification of a non-local variable

Static scope

- In *static* scoping, a name is bound to a collection of statements according to its position in the source program.
- Also called *lexical* scoping – based on grammatical structure of the program
- Algol 60 introduced nested scoping, including nested functions and a begin-end block of code that could include declarations and functions
- Static scope is determined at compile time and does not vary with the execution history of a program
- Static scope is used by most modern languages

Static Scope – non local names

- 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*

Nested and Disjoint Scopes

- Name collision becomes increasingly important as programs and memories become larger
  - Names with limited scope can be reused elsewhere
- Two different scopes are either nested or disjoint.
- In disjoint scopes, same name can be bound to different entities without interference.
- Nested scopes are like nested loops: one scope is contained within another
- What constitutes a unit of scope?

Lexical Units of Scope

	Algol	C	Java	Ada
Package	n/a	n/a	yes	yes
Class	n/a	n/a	nested	yes
Function	nested	yes	yes	nested
Block	nested	nested	nested	nested
For Loop	no	no	yes	automatic

- This table is not definitive
- Note that the compilation unit always constitutes a scope in addition to the above units

Nested Subprograms

- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Fortran 2003, Pascal and PHP)
- Varying levels of support in Javascript, Actionscript, Perl, Ruby, Python
- In other languages nested scopes can still be created:
  - Java and C++ in nested classes and in blocks
  - C in blocks
- Considered a useful tool for encapsulation and isolation of concerns by limiting visibility
  - The OO paradigm provided a different mechanism for writing functions with limited visibility

Nested Subprograms (Pascal)

```
function Foo(i: integer): integer
  function Bar(x: integer): integer
    begin
        bar := i * x
    end
begin
    Foo := Bar(42)
end
```

Name hiding

- In nested scopes variables in an outer scope can be hidden from an inner by declaring a variable with the same name
- Ada allows access to these "hidden" variables
  - E.g., **`unit.name`**

Ada Example

```
procedure Main is
  x : Integer;
  procedure p1 is
    x : Float;
    procedure p2 is
      begin
        ... x ...
      end p2;
    begin
      ... x ...
    end p1;
  procedure p3 is
    begin
      ... x ...
    end p3;
Begin
 ... x ...
End Main;
```

References to x in Main and p3 refer to the integer x declared in Main

References in p1 and p2 refer to the float x declared in p1

But we could use explicit main.x in p1 or p2 to refer to the integer variable

Block Scope

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

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

- Java allows block scope but does not allow variables in the immediate enclosing scope to redeclared in the block

Declaration Order

- C99, C++, Java, VB.NET 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
- Other languages require all variables to be declared before any executable statements
  - Which is more readable? Writable?

For Loop control control variables

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

```
For (int k = 0; k < j; k++{
   . . .
}
```

- Ada provides automatic block score for loop control variables in `for` statements

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
  - Variables declared outside of any function are usually allocated in static storage
- C and C++have both declarations (just attributes) and definitions (attributes and storage)
  - A declaration using **`extern`** specifies definition in another file

```
extern int c
```

Global Scope - PHP

- Programs are embedded in XHTML markup documents, in any number of fragments, some statements and some function definitions
- 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
- PHP is unusual in that global variables are not implicitly visible to functions
  - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`
  - Because PHP does not require variable declaration, a reference to a global is normally assumed to be a hiding declaration of a new variable

PHP example

```
function DoMySQLiQuery($sql, $result_type = MYSQL_BOTH) {
    global $mysqli;
    $result = mysqli_query($sql, $mysqli);
    if(mysqli_errno($this->mLink))
       die(mysqli_error($mysqli));
    $phpResult = array();
    while ($row = mysqli_fetch_array($result, $result_type)) {
       $phpResult[] = $row;
    }
    mysqli_free_result($result);
    if(mysqli_errno($mysqli))
          die(mysqli_error($mysqli));
    return $phpResult;
}
```

Global Scope - 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 some cases, too much access is possible
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested
  - Use of global variables is often frowned upon because of unexpected side effects and changes
- But sometimes global variables are the most efficient solution to a problem (ex. Lexical analysis)

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

```
Big
    - declaration of X
    Sub1
      - declaration of X -
      ...
      call Sub2
      ...

    Sub2
      ...
      - reference to X -
      ...

    ...
    call Sub1
    ...
```

Big calls Sub1
Sub1 calls Sub2
Sub2 uses X

Scope Example

- Static scoping
  - Reference to X is to Big's X
- Dynamic scoping
  - Reference to X is to Sub1's X
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically type check
    3. Poor readability- it is not possible to statically determine the type of a variable

Scope and Lifetime

- The *lifetime* of a variable is the time interval during which the variable has been allocated a block of memory.
- Early languages Fortran and Cobol used static (compile time) allocation.
  - Memory was allocated in a global memory area
  - Use of static global memory for function parameters and return addresses means that recursive functions cannot exist
  - All variables existed for the duration of the program
- Memory management was the programmer's responsibility
  - Early machines had very limited memory space e.g., IBM 1130 32KB; IBM 360 64KB

Dynamic stack variables

- Algol introduced the notion that memory should be allocated/deallocated at scope entry/exit.
- This allows recursive functions to exist because the local memory for the function is allocated when the function is invoked
- Almost all languages use a stack for function local memory
  - Structure is often called a "stack frame"
  - Contains parameters, return addresses, local or automatic variables, pointers to stack frames for caller and/or outer scope
- For dynamic stack variables, scope == lifetime

When Scope != Lifetime

- With static allocation a variable never "forgets" its value, even variables declared within a function scope
- With dynamic allocation variables are created and destroyed as the program runs.
  - Memory allocated on function entry is returned on exit and will be overwritten
- Most languages provide mechanisms that can be used to break the *scope equals lifetime* rule.

Counting function invocations

- It is sometimes handy to know how often a particular function has been called:

```
Double func() {
 count++
. . .
}
```

- But if count is declared inside func it will be recreated with every invocation
- So we do this:

```
Double func() {
  static int count = 0;
  count++
  . . .
}
```

- Note that count is initialized to 0 during compilation (not runtime) and is never reinitialized

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

Example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- of Sub1
      ...   <---------- 1
    end -- of Sub1
  procedure Sub2 is
    X, Z : Integer;
    procedure Sub3 is
      X : Integer;
      begin -- of Sub3
        ...   <---------- 2
      end   -- of Sub3
    begin -- of Sub2
      ...   <---------- 3
    end -- of Sub2
  begin -- of Example
    ... <---------- 4
  end   -- of Example
```

- Referencing Environments
- At point 1:
  X and Y of Sub1, A and B of Example
- At point 2:
  X of Sub3 (X of Sub 2 is hidden), Z of Sub3, A and B of Example
- At point 3:
  X of Sub 2, A and B of Example
- At point 3:
  A and B of Example

Dynamic Scope Example

```
void sub1(){
  int a,b;
  ... <-----------1
} // end sub1
void sub2(){
  int b,c;
  ... <-----------2
  sub1();
} // end sub2
void main(){
  int c,d;
  ... <-----------3
  sub2();
} // end main
```

- Referencing Environments
- At point 1:
  a and b of sub1, c of sub2, d of main (c of main and b of sub2 are hidden)
- At point 2:
  b and c of sub2, d of main (c of main is hidden)
- At point 3:
  c and d of main

Named Constants

- A *named constant* is a variable that is bound to a value only when it is bound to storage
- Advantages: readability, parameterization and modifiability
  - It's more readable to write pi than 3.14159
  - It's easier to modify **`#define question_count 129`** than to change the magic number 129 wherever it occurs
  - This provides a crude form of parameterization
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic

Named Constants

- Named constants in some languages must be declared with constant valued expressions
  - Fortran 95, C, C++ (with #define)
- Storage need not be allocated for such constants
- Other languages allow dynamic binding of values to named constants

```
const int elementcount = rows * columns;
```

- Ada, C++, and Java: expressions of any kind
- C# has two kinds, `readonly` and `const`
  - the values of `const` named constants are bound at compile time
  - The values of `readonly` named constants are dynamically bound

Initialized Data

- It is often convenient to initialize variables with known values at compile-time
- Many languages allow initial values to be specified in a variable declaration:

```
int x = 0;
int  c[5] = {10,20,30,40,50}
int * foo = c;  /* foo is an alias of c */
```

- Data can only be initialized with literal values or expressions that can be evaluated before runtime
- In compile languages, initialized data becomes part of the executable image