Concepts of Programming Languages

3rd Week

Variables: Names, Bindings, Types & Scopes

Prof. Dr. Peter Heusch

One Name – Two Things

- Variables in imperative and object oriented languages mirror the Von-Neumann-Architecture:
  - A Processor performing all computations
  - A Memory storing all variables
  - Variable values can be accessed and altered

- Variables in functional languages are named placeholders for function results
  - Variable values can be accessed
  - Variable values cannot be altered

Imperative Variables

- Many choices to make regarding the implementation of
  - Name
  - Address
  - Value   Binding
  - Type
  - Lifetime
  - Visibility (Scope)

Names

- Every variable and every field inside a higher data type is characterized by its name
- During the last 50 years names became longer and longer:
  - Fortran I: 6 Characters
  - COBOL: 30 Characters
  - ANSI C and Fortran 90 : 31 Characters
  - C++: No specified limit, but some compilers limit the number of significand chars
  - Ada and Java: Arbitrary length and all characters are relevant

Case Sensitivy

- Many languages (e.g., Pascal, Modula, Oberon, Delphi, Ada) ignore case of names
- Other languages (C, C++, Java) treat someName, SomeName and SOMENAME as different
- During the last years the trend seems to favor case sensitive names
- C, C++, Java use conventions for case:
  - All non-constants use CamelCaseSyntax
  - Constants use UPPERCASE_WITH_UNDERSCORE

Reserved Words vs. Keywords

- Reserved Words are not allowed as names
  - Keywords are allowed as names
- An example from FORTRAN:
  - `Real SomeVar` is a type declaration, therfore `Real` is a keyword
  - `Real = 3.141` is an assignment, therefore `Real` is a variable name
- Reserving all special words enhances the readability of a programming language

Aside: Identifiers and Blanks

- Predefined Identifiers:
  - Java has predefined literals (`true`, `false`, `null`), that are not reserved but cannot be used as variable names
  - In Pascal, the names of builtin data types are not reserved and could in theory be reused
- Blanks:
  - Fortran & Elan allow for blanks in variable names, most other languages don't
  - Ignoring all blanks in identifiers introduces the need for additional delimiter chars

Imperative Variables

- Many choices to make regarding
  - Name
  - **Address**
  - **Value** Binding
  - Type
  - Lifetime
  - Visibility (Scope)

Variable Attributes: Addresses & Values

- **Addresses** of variables are non-constant values during program runtime:
  - Stack variables: different addresses for each invocation
  - Heap variables: different addresses for each allocation
- Aliasing: Two variables may assign the same address (makes programs harder to understand!)
  - Aliases can exist via pointers, references and unions
- **Value**: The contents of the location with which the variable is associated

Assigning Value: Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization
- Initialization is often done in the declaration statement,
    - in Java: `int sum = 0;`
- When initiliziation occurs automatically at program start, problems may occur:
    - Mutual import in Python
    - Static initializers in C++
    - Compiling mutually referencing classes in Java

Imperative Variables

- Many choices to make regarding
  - Name
  - Address
  - Value Binding
  - **Type**
  - Lifetime
  - Visibility (Scope)

Variable Attributes: Type

The **type** determines:

- Range of values
- Set of possible operations that are defined
- For floating point, the type also specifies the precision

- Abstract memory cell - the physical cell or collection of cells associated with a variable

Explicit and Implicit Declarations

- *Explicit* declaration: Program statement used for declaring the types of variables
- *Implicit* declaration: Default mechanism for specifying types of variables (at their first appearance in the program)
  - FORTRAN, PL/I, BASIC and Perl provide implicit declarations, usually given by prefixes or suffixes of the name
  - Advantage: writability
  - Disadvantage: reliability
  - FORTRAN can use both declarations

Imperative Variables

- Many choices to make regarding
    - Name
    - Address
    - Value
    - Type
    - Lifetime
    - Visibility (Scope)

Binding

Binding

- A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
    - Or of a variable to an address: l-value (left)
    - Or of a variable to a value: r-value (right)
- Binding time is the time at which a binding takes place.
    - Earliest at language design time
    - Latest during runtime

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 FORTRAN 77 variable to a memory cell (or a C `static` variable)
- Runtime: bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

- Bindings that are performed before runtime and that cannot change afterwards are called static:
    - Binding a Java variable to its type
    - Binding a C function to its code
- Bindings that are performed at runtime or can change at runtime are called dynamic:
    - Binding a Java variable to its address
    - Binding a Python function to its code

Dynamic Type Binding

- Assigns variable type at run time
  - Used in many interpreted languages (JavaScript, PHP, Python)
  - Advantage: flexibility, genericity
  - Disadvantage: high runtime cost, type checking by compiler is difficult
- An example from Python:

```
 list = [ 1, 2, 3, 4] # defines a list
 list = 3.141 # defines a float
 def list(): # defines a function
print(„This is a list“) # String
```

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- Type checking ensures that the operands of an operator are of compatible types
- A compatible type is one that is:
  - legal for the operator
  - OR allowed under language rules to be implicitly converted to a legal type (this is called coercion)
- A type error is the application of an operator to an operand of an inappropriate type

Coercion

- Coercion can weaken strong typing considerably!
- Example: C++ allows coercion by four rules:
    - Coercion between the builtin numeric types
    - Coercion from derived value to base value
    - Coercion by construction
    - Coercion by explicit conversion operator
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada:
    - Checking for subrange types
    - Checking for array dimensions

Type Checking (Definition)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- **A programming language is strongly typed if type errors are always detected**
- Programming languages are often categorized according to their type checking:
  - Strongly typed languages (many compiled languages are „almost", but very few are really)
  - Weakly type languages (most other languages)

Strong/Weak/Static/Dynamic

	Strong Typing	**Weak Typing**
Static Typing	Bind type at compile time Check rigorously	Bind type at compile time Check loosely
Dynamic Typing	Bind type at runtime Check vigorously	Bind type at runtime Check loosely

Strongly Typed Languages

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
  - FORTRAN 77 is not: parameters, `EQUIVALENCE`
  - Pascal is mostly: variant records
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked, pointers can be cast at random
  - Ada is, almost (`UNCHECKED CONVERSION`),
  - Java and Eiffel are almost, too (explicit casting may cause runtime errors)

Name Type Compatibility

- Two variables are said to be of the same type iff their type names are identical:
  - Easy to implement when types are simple names
  - Hard to implement when variable declarations can be structured, e.g. Pascal:
    procedure foo (index: array[1..10] of integer)... is legal but unusable
- Hard restrictions, e.g. enums and integers are mutually incompatible

Structure Type Compatibility

- Structure type compatibility means that two variables have compatible types if their types have identical structures
    - More flexible, but harder to implement

- Used in COBOL in assignments like
  `MOVE CORRESPONDING FROM a TO b`

Type Compatibility (contd.)

- Consider the problem of two structured types:
  - Are two record types compatible if they are structurally the same but use different field names?
  - Are two array types compatible if they are the same except that the subscripts are different? (e.g. [1..10] and [0..9])
  - Are two enumeration types compatible if their components are spelled differently?
  - With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Imperative Variables

- Many choices to make regarding
    - Name
    - Address
    - Value — Binding
    - Type
    - **Lifetime**
    - Visibility (Scope)

Lifetime

- The lifetime of a variable is the time during which it is bound to a particular memory cell
- Lifetime in terms of Storage Binding:
    - Allocation: Getting a cell from a pool of available cells
    - Deallocation: Putting a cell back into the pool
- Different storage binding strategies determine when allocation happens
    - Before program execution
    - At declaration execution)
- Lifetime depends on memory management
    - End can be deterministic (C++, C, PASCAL, Python)
    - End can be random (Java, Eiffel)

Imperative Variables

- Many choices to make regarding
  - Name
  - Address
  - Value   Binding
  - Type
  - Lifetime
  - **Visibility (Scope)**

Scope

- The scope of a variable is the range of statements over which it is visible
- Variables that are visible in a program unit but were not declared there are called nonlocal
- The scope rules of a language determine how references to names are associated with variables

Static Scoping

- Based on program text
- 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

Blocks

Blocks are used to introduce static scopes inside program units:

- In C, C++:

```
for (...) {
   int index;
   ...
}
```

- In Ada:

```
declare LCL : FLOAT;
begin
  ...
end
```

Hiding

- Variables can be *hidden* from a unit by having a "closer" variable with the same name
- C++, Java & Ada allow access to these "hidden" variables
    - In Ada: unit.name
    - In Java: class.name or reference.name
    - In C++: class_name::name
    - In C++ also as „implicit unhiding“ in constructors
- Java forbids hiding of local variables by other local variables

Static Scope: Example

```
function big(){

   function sub1() {
      var x = 7;
      x = x+1;
      sub2();
   }
   function sub2() {
      var y = x;
   }

   var x = 3;
   sub1();
}
```

local x hides x in big()
to any use in sub1()

x refers to x in big()
(first mention in the set
of static ancestors)

Static Scope (contd.)

- Suppose the spec is changed so that `sub2()` must now access some data in `sub1()`
- Solutions:
    - Put `sub2()` in `sub1()` (but then, `big()` can no longer call it and `sub2()` cannot access `big()`'s `x`)
    - Move the data from `sub1()` that `sub2()` needs to `big()` (but then, all procedures can access them)
- Same problem for procedure access
- Overall: static scoping often encourages many globals

Dynamic Scope

- Idea: Not the textual layout determines the data acess possibilities, but the calling order
- If a variable is not found inside the current scope, the **calling** scopes are searched in reverse order
- Needs supporting data structures to find variables at runtime
- Can be problematic if variables are referenced that do not exist in any surrounding scopes

Dynamic Scope: Example

```
function big(){

   function sub1() {
      var x = 7;
      x = x+1;
      sub2();
   }
   function sub2() {
      var y = x;
   }

   var x = 3;
   sub1();
}
```

local x hides x in big() to any use in sub1()

x refers to x in sub1() (the caller)

Referencing Environments

- The referencing environment of a statement is the collection of all visible names
- In a static-scoped language, this comprises:
  - Local variables
  - Visible variables in all of the enclosing scopes
- In a dynamic-scoped language, this comprises:
  - Local variables
  - Visible variables in all active subprograms
- An active subprogram has begun execution but has not yet terminated

Static vs. Dynamic

- Static scoping tempts programmers to use global variables and avoid structuring code by nesting
    - Reliability problems due to incorrect accessibilty
- In dynamic scoping, functions can sometimes reference different variables depending on the calling function:
    - Reliability problems due to incorrect accessibility
    - Readability problems: Calling order must be known

Perl Scoping Sample

- Three scoping types:
  - Dynamic scoping: local
  - Static scoping: my
  - Default are global variables
- Sample:

```
sub foo() {
   my $x = 1;
   local $y = 2;
  $z = 3;
  bar();
}
```

```
sub bar() {
 print "$x:$y:$z";
}

$x=11;$y=12;$z=13;

bar(); # 11:12:13
foo(); # 11:2:3
bar(); # 11:12:3
```

Scope and Lifetime

- Scope and lifetime are sometimes related, but are different concepts
- Consider a static variable in a C or C++ function:
  - The scope of the variable is the block where the variable is defined
  - The lifetime of the variable is the program lifetime
- Consider an object member in Java:
  - The scope are all classes knowing the object
  - The lifetime is between `new` and `finalize` (garbage collection)

Imperative Variables

- Many choices to make regarding
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Visibility (Scope)

Binding