

CS 430
Spring 2020

Mike Lam, Professor

$$y = \textcircled{2x} + 5$$

Variables and Scoping

Course outline

- Syntax (modules 2-3)
- **Semantics** (modules 5-8, 10-11, and 13-14)
 - Variables and scoping
 - Types and type checking
 - Expressions and control structures
 - Parameters and subprograms
- Implementation (modules 16 and 18-19)
 - Activation and environments
 - Abstraction and OOP
 - Concurrency and Error Handling
- History (module 20)

Variables

- What is a variable?

Variables

- Variable: an abstraction of memory cells
 - Most languages have variables
 - However, they are NOT essential for computation!
- Six main attributes/properties:
 - Name
 - Address
 - Value
 - Type
 - Lifetime
 - Scope

Binding

- **Binding**: attribute/entity association
 - Bindings begin at **binding time**
 - Language design/implementation time
 - Compile time
 - Load/link time
 - Run time
 - **Static** bindings begin before the program is executed and do not change during execution
 - **Dynamic** bindings may begin or change during execution

Name

- **Name** – string of characters that serves as an identifier
 - Case sensitivity
 - Special characters with meanings (e.g., \$ and @ in Ruby)
 - Standards or conventions (e.g., camelCase vs. under_scores)
 - Semantic significance (e.g., type in FORTRAN and Prolog)
- Keyword vs. reserved word
 - **Keyword**: string of characters with special meaning
 - **Reserved word**: string of characters that cannot be used as a variable name (may or may not be a keyword)
- Name bindings are usually static
 - Often created by a **declaration**
 - Do all variables have a name?

Address

- Address: location of a variable in memory
 - Sometimes called **I-value**
- Address bindings may be static or dynamic
 - Creation of this binding is called **allocation**
- **Aliases**: multiple variables with identical addresses

Value

- Value: contents of the memory associated with a variable
 - Sometimes called **r-value**
- Value bindings are usually dynamic
 - Otherwise, they wouldn't be "variable"
 - First binding is called **initialization**
 - Important exception: **named constants**

Type

- **Type**: range of values a variable can store
 - And the operations that can be applied to it
- Common types:
 - Primitive: integer, floating-point, complex, bool, character
 - Composite: array/string, pointer, tuple, record, union, object
- **Implicit** vs. **explicit** binding
 - `int x = 5` vs. `x = 5`
- **Static** vs. **dynamic** typing
 - i.e., can a variable change type at runtime?
- **Type inference**
 - A language can be both implicitly and statically typed!

Type binding examples

- **Java** (explicit static)

```
int x = 5;  
x = "hello";    // compiler error
```

- **JavaScript** (implicit dynamic)

```
var x = 5;      // x is an int  
x = "hello";   // now it's a String
```

- **Java 10** (implicit static)

```
var x = 5;      // x is inferred to be an int  
x = "hello";   // compiler error
```

Lifetime

- **Lifetime**: duration of address/storage binding
- Common lifetimes are based on location:
 - **Static** (entire program execution)
 - **Stack-dynamic** (single function execution)
 - **Heap** (arbitrary)
 - Binding is created at **allocation**
 - Binding is destroyed at **deallocation**

Lifetime

- **Explicit heap-dynamic**: nameless memory accessed w/ pointers or references (e.g., C/C++ or Java)
 - Allocated explicitly (e.g., `malloc` in C or `new` in C++ or Java)
 - Can be deallocated explicitly (e.g., `free` in C and `delete` in C++)
 - Some languages (e.g., Rust) have delegation mechanisms
 - Can be deallocated implicitly (e.g., garbage collection in Java)
- **Implicit heap-dynamic**: allocated only when assigned a value (e.g., arrays in Javascript)
 - Reallocated when assigned a different value
 - Deallocated implicitly

We will consider instance variables to be explicit or implicit according to the object they belong to; this is somewhat ambiguous in our textbook

Scope

- **Scope**: program range where a variable is visible
 - A variable is **visible** if it can be referenced without qualification
 - Many possible ranges (e.g., **block**, **function**, **global**, **package**)
 - OOP brings even more possibilities (**public**, **private**, **protected**)
- **Local** vs. **non-local** variables
 - A variable is **local** in the scope where it is declared
 - Local variables **shadow** (hide) non-local variables w/ same name
 - Sometimes shadowed variables are still accessible w/ qualification
- Often related to lifetime
 - But not necessarily! (e.g., `static local` in C)

Scope

- **Static (lexical)** vs. **dynamic** scoping
 - Code structure vs. call structure
 - Both involve finding a variable (**name resolution**) by searching through a hierarchy of scopes
 - Static scoping: compiler can do the search
 - Dynamic scoping: search the stack at runtime
 - Dynamic scoping is rare now and usually optional
 - Example: “my” (static) vs. “local” (dynamic) in Perl

Referencing Environment

- **Referencing environment**: all variables visible at some statement **without qualification**
 - Local scope plus ancestor scopes
 - Related concept from compilers: **nested symbol tables**
 - Which variables are visible at the **blue** and **green** statements?

```
class Shadowing3 {  
    public static void main(String[] args) {  
        if (true != false) {  
            x = 6;  
        }  
        int x = 5;  
        System.out.println(x);  
    }  
}
```

Environment at **blue**: { main.args:string }

Environment at **green**: { main.args:string, main.x:int }

Static/dynamic scoping example

- For both static and dynamic scoping:
 - What is the output?
 - What are the referencing environments at location A, B, and C?

```
program p {
  int x = 5
    y = 2
  // LOCATION A
  func g() {
    int x = 12
      z = 8
    // LOCATION B
    f()
  }
  func f() {
    // LOCATION C
    println(x)
  }
  g()
}
```


Scoping nuances

- Some languages allow mixing of declarations and code (e.g., C99)
 - Scope is usually from declaration to end of program unit
- Some languages require declaration before reference
 - **Declaration order** can influence scoping
- Block-structured languages often restrict scope of declarations in a block
 - Sometimes allow duplicate names within a larger enclosing scope
- Many languages do not require explicit declarations (e.g., Ruby)
 - Scoping often defaults to function-level (why not block?)
- Scoping is usually enforced by compiler/interpreter, but not always
 - In Python, “private” class fields (starting w/ underscores) aren’t private!

Scoping nuances

- “Global” can mean different things
 - In Ruby, global variables are truly global (accessible from entire program)
 - In C, “global” variables are actually only accessible from code in the same module (extern required to access it from a different file)
 - In Python, global variables must be marked in functions that wish to use them, and must be tagged with module name outside the module

Global scoping example

- What does this Python program print?

```
x = 5
```

```
def bar():  
    print(x)
```

```
def baz():  
    x = 7  
    print(x)
```

```
def bam():  
    global x  
    x = 7
```

```
bar()  
baz()  
print(x)  
bam()  
print(x)
```

```
x = 5
```

```
def hipster():  
    print(x)  
    x = 4  
    print(x)
```

```
hipster()
```

Block scoping examples

- Java:

```
int foo() {  
    int x;  
    if (someTest()) { x = 5; }  
    else { x = 7; }  
    return x;  
}
```

```
int foo() {  
    if (someTest()) { int x = 5; }  
    else { int x = 7; }  
    return x;  
}
```

- Ruby:

```
def foo()  
    if someTest()  
        x = 5  
    else  
        x = 7  
    end  
    return x  
end
```

Case studies

- Questions
 - What is the name, address, value, type, lifetime, and scope?
 - Are the bindings static or dynamic?
- Cases
 - Java “private” class instance variable
 - What would be different in C++?
 - Java “public static final” class variable
 - C local loop index variable
 - i.e., “for (`int i = 0; i < N; i++`)”

Reminder: common lifetimes include

- Static
- Stack dynamic
- Explicit heap dynamic
- Implicit heap dynamic

Case studies

Java "private" class instance variable

name: static, bound at compile time
address: dynamic, bound on object instantiation
value: dynamic, bound on every assignment
type: static, bound at compile time
lifetime: explicit heap dynamic: object instantiation to garbage collection
scope: static, all methods in class

Java "public static final" class variable

name: static, bound at compile time
address: static, bound at compile time
value: static, bound at compile time
type: static, bound at compile time
lifetime: static, entire execution
scope: static, all code that can see the class

C local loop index variable

name: static, bound at compile time
address: dynamic, bound at function entry
value: dynamic, re-bound on every loop iteration
type: static, bound at compile time
lifetime: stack dynamic, during function execution
scope: static, loop body