

# Organization of Programming Languages

## CS320/520N

---

### Lecture 05

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

*[bunescu@ohio.edu](mailto:bunescu@ohio.edu)*

# Imperative Programming

---

- **Imperative Programming** is about writing *statements* that perform some *action* that changes *state* in order to perform a well defined computation.
- **Imperative Languages** are abstractions of von Neumann architecture:
  - Memory.
  - Processor.
- A **variable** is an abstraction of a memory cell.
- The **assignment** statement is how state is changed.
  - An abstraction of the piping of data from memory to processor in the von Neumann architecture.

# Variables & Attributes

---

- Variables can be characterized as a sextuple of attributes:
  - Name.
  - Address.
  - Value.
  - Type.
  - Scope.
  - Lifetime.
- **Name:**
  - Most variables have names.
  - Exception: *explicit heap-dynamic variables*.

# The Address

---

- **Address** – the memory address with which it is associated.
- A variable may have different addresses at different times during execution:
  - Ex: local variables allocated on the run-time stack.
- 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 (program readers must remember all of them).

# The Value

---

- **Value** – the contents of the location with which the variable is associated.
- Alternatively:
  - The **l-value** of a variable is its address.
  - The **r-value** of a variable is its value.
- Assignment statement:  **$x = x + 1$** 
  - If a variable appears on the RHS, its r-value is used.
  - If a variable appears on the LHS, its l-value is used.

# L-Values vs. R-Values

---

```
int x = 0;    // initialization at the point of declaration.  
  
...  
x = x + 1;   // get the r-value of x, add 1,  
              // and store the result into the l-value of x.
```

In imperative languages:

1. Literal constants only have r-values:
  - $\text{rval}(1) = 1$ , whereas  $\text{lval}(1)$  is undefined.
2. Variables have both r-values and l-values:
  - $x = x * y$  is interpreted by compilers as  $\text{lval}(x) \leftarrow \text{rval}(x) * \text{rval}(y)$ .

# L-Values vs. R-Values

---

3. Variables of pointer type have an r-value that is the l-value of some other variable:

- In C/C++, two special unary operators override the default action of the compiler:
  - the unary ‘&’ operator (*address of operator*) overrides the r-value computation and instead returns the l-value.
  - the unary ‘\*’ operator (*pointer dereference*) returns the r-value when applied to a pointer variable, i.e. the l-value of another variable.

```
int *p = &x; // rval(p) == lval(x)
```

```
*p = 2 * x; // rval(p) ← rval(2) * rval(x)
```

# L-Values vs. R-Values

---

## 4. Declared functions have l-values but no r-values:

```
int f(int y);    // lval(f) is some global address.
```

```
typedef int (*IFP)(int); // pointer to an int function that  
                        // takes an int argument.
```

```
IFP g = &f;    // lval(g) ← lval(f);
```

```
(*g)(5);      // because rval(g) == lval(f), this means (*g)(5)  
                // invokes f with argument rval(5).
```

# The Type

---

- The type of a variable determines:
  - The set of values that the variable can store.
  - The set of operations that are defined on these values.
- For example, the **int** primitive type in Java:
  - specifies the range  $[-2147483648, 2147483648]$ .
  - operations such as addition, subtraction, multiplication, division and modulo.

# Bindings

---

- A **binding** is an association, such as:
  - between an attribute and an entity (e.g. variable);
  - between an operation and a symbol.
- **Binding time** is the time at which a binding takes place:
  - 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: `count = count + 1;`

# Bindings: Static vs. Dynamic

---

- There are 2 types of bindings of attributes (e.g. type & address) to variables: **static** and **dynamic**.
- 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 the execution of the program.

# Static Type Binding

---

- **Explicit declaration** – a program statement used for declaring the types of variables
  - Most programming languages (C/C++, Java, ...)
  - In C/C++:
    - declarations only specify types and other attributes;
    - definitions specify attributes and cause storage allocation.
- **Implicit declaration** – a default mechanism for specifying types of variables
  - In PERL, prefixes define types: any name beginning with \$ is a scalar (numeric or string), @ is an array, % is a hash structure.
  - In ML, types are implicitly associated using type inference.

# Dynamic Type Binding

---

- The variable is associated with a type every time it is assigned a value through an assignment statement.

- Examples (Javascript):

```
list = [2, 4.33, 6, 8];
```

```
list = 17.3;
```

- Advantage: flexibility (generic program units).
- Disadvantages:
  - Usually purely interpreted  $\Rightarrow$  slow execution.
  - Costly implementation of dynamic type checking.

# Storage Binding & Lifetime

---

- **Storage Binding:**
  - **Allocation** = getting a memory cell from some pool of available cells, in order to bind it to a variable.
  - **Deallocation** = putting a memory cell (unbound from a variable) back into the pool.
- The **lifetime** of a variable is the time during which it is bound to a particular memory cell. Depending on their lifetime, 4 categories of variables:
  1. Static.
  2. Stack-Dynamic.
  3. Explicit Heap-Dynamic.
  4. Implicit Heap-Dynamic.

# Static Variables

---

- A **static** variable is bound to a memory cell before execution begins and remains bound to the same memory cell throughout execution.
  - Example: C and C++ `static` variables.

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

# Static Variables

---

- Advantages:
  - efficiency: direct addressing, no run-time overhead for allocation & deallocation.
  - history-sensitive : maintain values between successive function calls.
- Disadvantages:
  - lack of flexibility (no recursion).
  - storage cannot be shared among variables.

# Stack-Dynamic Variables

---

- **Stack-dynamic** = 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 (with some exceptions).
  - Storage is allocated from the run-time stack.
  - If scalar, all attributes except address are statically bound:
    - Example: local variables in C subprograms and Java methods.

```
int factorial(int n) {  
    int result = 1;  
    for (int i = 2; i ≤ n; i++)  
        result *= i;  
    return result;  
}
```

# Stack-Dynamic Variables

---

- Advantages:
  - 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** variables are allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution:
  - Storage is allocated from the heap.
  - The actual variables are nameless.
  - Referenced only through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java.

```
int *intNode;           // create the pointer, stack-dynamic.
...
intNode = new int;     // create the heap-dynamic variable.
...
delete intNode;       // deallocate the heap-dynamic variable.
```

# Explicit Heap-Dynamic Variables

---

- Advantages:
  - Enable the specification and construction of dynamic structures (linked lists & trees) that grow and shrink during the execution.
- Disadvantages:
  - Unreliable: difficult to use pointers & references correctly.
  - Inefficient: heap management is costly and complicated.

# Implicit Heap-Dynamic Variables

---

- **Implicit heap-dynamic** variables – allocation and deallocation caused by assignment statements:
  - All their attributes (e.g. type) are bound every time they are assigned.
  - Examples: strings and arrays in Perl, variables in JavaScript & PHP.

```
list = [2, 4.33, 6, 8];
```

```
list = 17.3;
```

- Advantages: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic.
  - Loss of error detection by compiler.

# Type Checking

---

- Preliminary step: generalize the concept of operands and operators to include:
  - subprograms as operators, and parameters as operands;
  - assignments as operators, and LHS & RHS as operands.
- **Type checking** is the activity of ensuring that the operands of an operator are of *compatible types*.
- A **compatible type** is one that is either legal for the operator, or is allowed under language rules to be implicitly converted to a legal type:
  - This automatic conversion , by compiler- generated code, is called a *coercion*.

# Type Checking

---

- A **type error** results from the application of an operator to an operand of an inappropriate type.
- **Static type checking:** if all type bindings are static, nearly all type checking can be done statically (Ada, C/C++, Java).
- **Dynamic type checking:** if type bindings are dynamic, type checking must be dynamic (Javascript, PHP).
- **Strong typing:** a programming language is strongly typed if type errors are always detected.
  - Done either at compile time or run time.
  - Advantages: allows the detection of the misuses of variables that result in type errors.

# Strong Typing: Language Examples

---

- C and C++ are not strongly typed:
  - parameter type checking can be avoided;
  - unions are not type checked.
- Ada is nearly strongly typed:
  - only exception: the UNCHECKED\_CONVERSION generic function extracts the value of a variable of one type and using it as if it were of a different type.
- Java and C# are strongly typed in the same sense as Ada:
  - types can be explicitly cast  $\Rightarrow$  may get type errors at run time.

# Strong Typing & Type Coercion

---

- Coercion rules can weaken the strong typing considerably i.e. loss in error detection capability:
  - C++'s strong typing less effective compared to Ada's.
- Although Java has just half the assignment coercions of C++:
  - its strong typing is more effective than that of C++.
  - its strong typing is still far less effective than that of Ada.

# Variable Attributes: Scope

---

- The **scope** of a variable is the range of statements over which it is *visible*
  - Variable  $v$  is visible in statement  $s$  if  $v$  can be referenced in  $s$ .
- The scope rules of a language determine how occurrences of names are associated with variables:
  - **static scoping.**
  - **dynamic scoping.**
- Two types of variables:
  - **local** variables: declared inside the program unit/block.
  - **nonlocal** variable: visible, but declared outside the program unit.

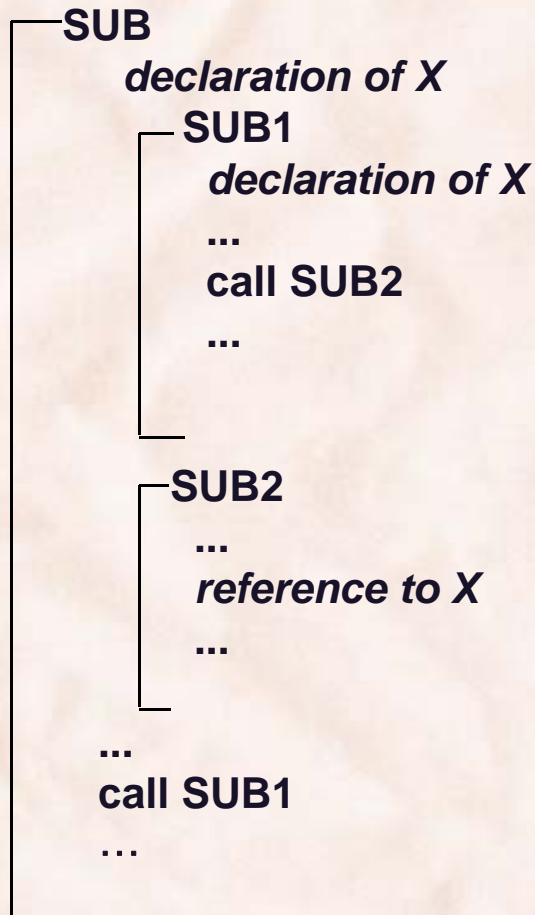
# Static Scope

---

- Introduced in ALGOL 60 as a method of binding names to nonlocal variables:
  - 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.
- Two ways of creating nested static scopes:
  - Nested subprogram definitions (e.g., Ada, JavaScript, and PHP).
  - Nested blocks.
- Given a specific scope:
  - Enclosing static scopes are called its static ancestors;
  - The nearest static ancestor is called its static parent.

# Static Scope Example

---



Sub calls Sub1  
Sub1 calls Sub2  
Sub2 uses X

# Static Scope

---

- Variables can be hidden from a unit by having a "closer" variable with the same name.
- C++ and Ada allow access to these "hidden" variables
  - In Ada: `unit.name`
  - In C++: `class_name::name`

# Static Scope: Blocks

---

- Blocks – a method of creating (nested) static scopes inside program units (introduced in ALGOL 60)
- Examples:
  - C-based languages:

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

- Ada:

```
declare Temp : Float;  
begin  
    ...  
end
```

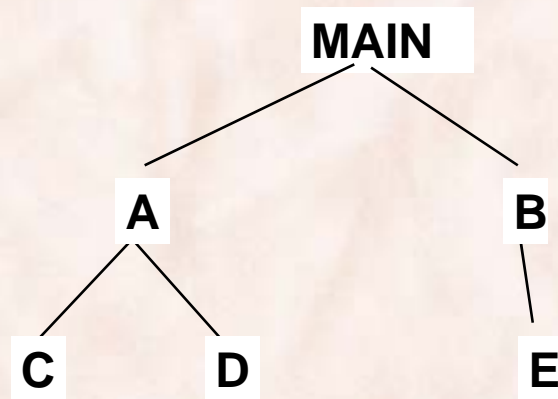
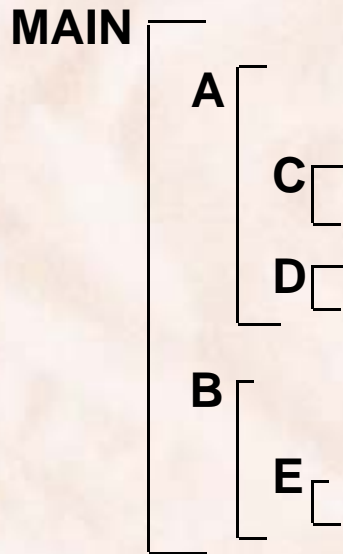
# Static Scope: Evaluation

---

MAIN calls A and B

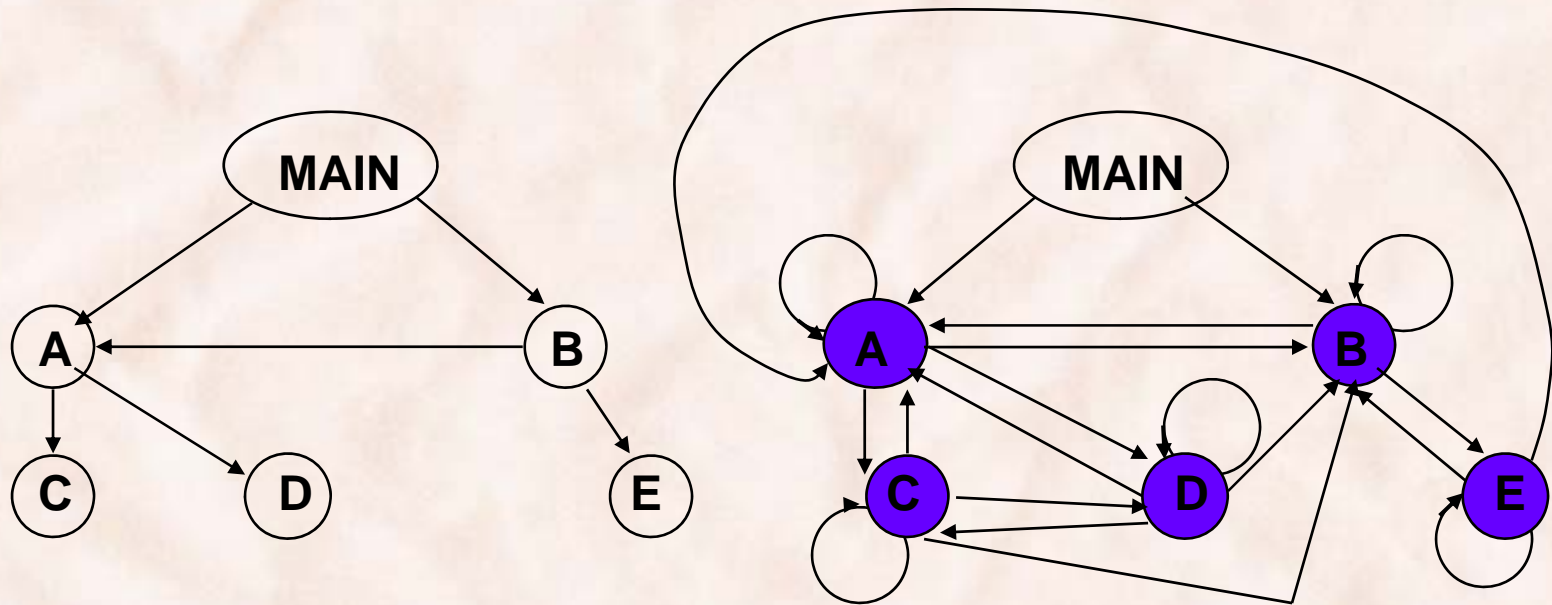
A calls C and D

B calls A and E



# Static Scope: Evaluation

---



# Static Scope: Evaluation

---

- Suppose the specification is changed so that D must now access some data in B.
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables).
  - Move the data from B that D needs to MAIN (but then all procedures can access them).
- Same problem for procedure access as for data access.
- Overall: static scoping often encourages many globals.

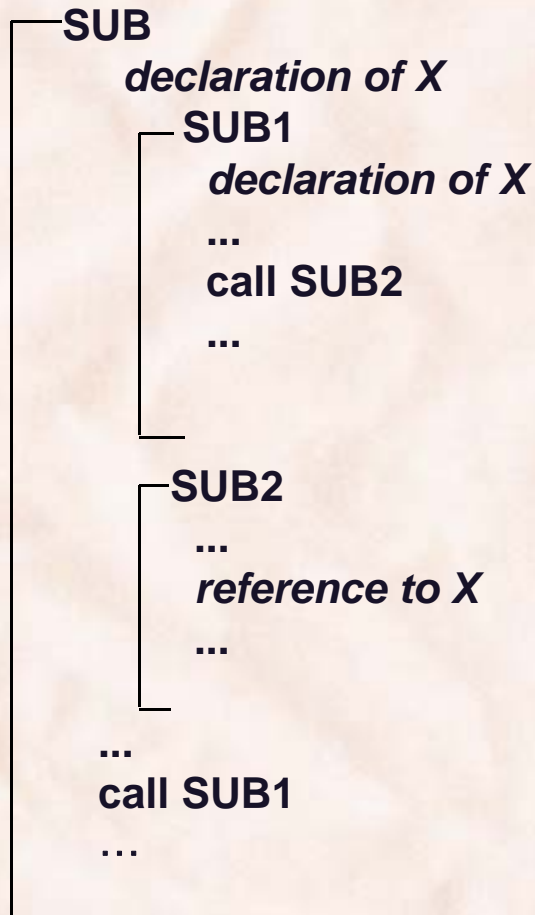
# Dynamic Scope

---

- **Static Scope:** names are associated to variables based on their textual layout (spatial).
- **Dynamic Scope:** names are associated to variables based on calling sequences of program units (temporal).
  - References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

# Dynamic Scope: Example

---



Sub calls Sub1  
Sub1 calls Sub2  
Sub2 uses X

# Dynamic Scope: Evaluation

---

- Advantages:
  - convenience: called subprogram is executed in the context of the caller  $\Rightarrow$  no need to pass variables in the caller as parameters.
- Disadvantages:
  - poor readability
    - virtually impossible for a human reader to determine the meaning of references to nonlocal variables.
  - less reliable programs than with static scoping.
  - execution is slower than with static scoping.

# Scope vs. Lifetime

---

- Sometimes scope and lifetime appear to be related:
  - Ex: a local variable in a Java method without method calls.
    - **scope**: from declaration to the end of the method (*spatial*).
    - **lifetime**: begins when the method is entered, and ends when the execution of the method terminates (*temporal*).
- Sometimes scope and lifetime are unrelated:
  - Ex: a static variable inside a C/C++ function:
    - **scope**: the scope of that function (statically bound)
    - **lifetime**: the entire execution of the program (statically bound).
  - Ex: a local variable inside a C++ function containing a function call.

# 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.
  - In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all **active** subprograms:
    - A subprogram is **active** if its execution has begun but has not yet terminated.
  - Variables in *enclosing scopes/active subprograms* can be hidden by variables with same name.
- See examples in Chapter 5.10.

# Named Constants

---

- A **named constant** is a variable that is bound to a value only when it is bound to storage.
- Advantages:
  - Readability and modifiability.
  - Used to parameterize programs.
- The binding of values to named constants can be either:
  - **Static** (constant-valued expressions):
    - FORTRAN 95:.
    - C# `const` named constants.
  - **Dynamic** (expressions of any kind):
    - Ada, C++, and Java.
    - C# `readonly` named constants.

# Variable Initialization

---

- **Initialization** = the binding of a variable to a value at the time it is bound to storage.
- Static storage binding:
  - ⇒ initialization occurs before run time
  - ⇒ initial value must be a constant expression (combination of constant literals and named constants).
- Dynamic storage binding:
  - ⇒ initialization occurs at run time.
  - ⇒ initial value can be an expression of any kind.

# Summary

---

- Variables: name, address, value, type, lifetime, scope.
- Binding = association of attributes with program entities:
  - Static vs. Dynamic.
- Type Binding:
  - Static vs. Dynamic
- Variables based on storage binding and lifetime:
  - static vs. stack dynamic vs. explicit/implicit heap dynamic
- Type Checking:
  - Static vs. Dynamic, Strong Typing & Type Coercion.
- Scope, Referencing Environment.