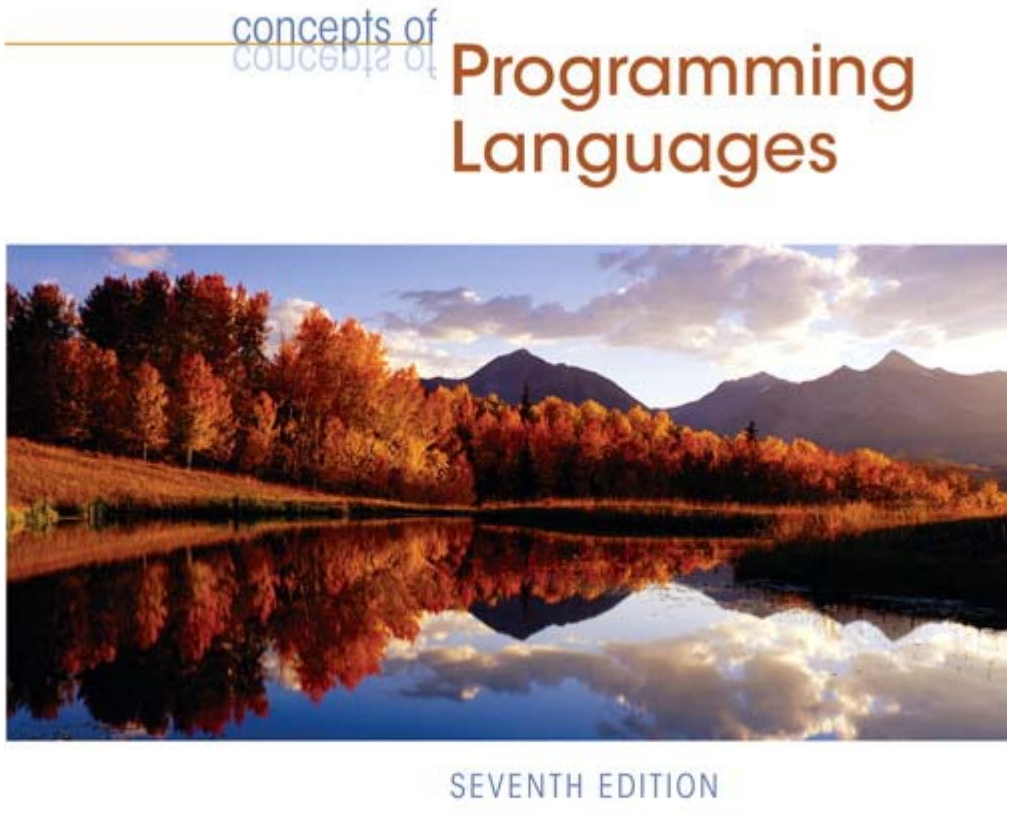
Chapter 5

Names, Bindings, Type Checking, and Scopes



ISBN 0-321-33025-0

Chapter 5 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Checking
- Strong Typing
- Type Compatibility
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables characterized by attributes
  - Name, type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

- Design issues for names:
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names (continued)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN I: maximum 6
    - COBOL: maximum 30
    - FORTRAN 90 and ANSI C: maximum 31
    - Ada and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Names (continued)

- Connectors
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - worse in C++ and Java because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)
  - C, C++, and Java names are case sensitive
    - The names in other languages are not

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
    - `Integer Real` can be used in Fortran, misleading

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - 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 (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; in the case of floating point, type also determines the precision
- *Value* – the contents of the location with which the variable is associated
- *Abstract memory cell* – the physical cell or collection of cells associated with a variable

The Concept of Binding

- The l-value of a variable is its address
- The r-value of a variable is its value
- 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 FORTRAN 77 variable to a memory cell (or a C `static` variable)
- Runtime -- bind a nonstatic local variable to a memory cell

`cnt = cnt + 5;// C statement`

- The type of `cnt` is bound at compile time
- The set of possible values of `cnt` is bound at compiler design time
- The meaning of the operator + is bound at compile time, when the types of its operands have been determined
- The internal representation of `6` is bound at compiler design time
- The value of `cnt` is bound at execution time with this statement.

Binding of Attributes to Variables: 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 (i,j,k,l,m,n -> integer) , PL/I, BASIC, and Perl ($:scalar, @: array, %:hash)
    - Advantage: writability
    - Disadvantage: reliability (less trouble with Perl)

Dynamic Type Binding

- Dynamic Type Binding (JavaScript and PHP)
- 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

Variable Attributes (continued)

- Type Inferencing (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference
- Example: ML is a functional and imperative programming language
  - `fun circumf(r) = 3.14 * r * r;`
    - **`real`** in ML
  - `fun times10(r) = 10 * r;`
    - **`int`** in ML
  - `fun square(r) = r * r;`
    - `square(1.5)` returns with an error message

Storage Bindings & 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 variables--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., all FORTRAN 77 variables, C/C++ `static` variables
  - Advantages: efficiency (direct addressing) – global variables, history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)

Categories of Variables by Lifetimes

- **Stack-dynamic variables**--Storage bindings are created for variables when their declaration statements are elaborated.
  - Allocated from run time stack
- If scalar, all attributes except address are statically bound
  - local variables in C subprograms and Java methods
- Advantage: allows recursion; conserves 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 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
- Advantage: provides for dynamic storage management
- Disadvantage: inefficient and unreliable

Categories of Variables by Lifetimes

- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl and JavaScript
- Advantage: flexibility
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *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, by compiler- generated code, to a legal type
  - This automatic conversion is called a **coercion**.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- 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

Strong Typing

- 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 not: variant records
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked
  - Ada is, almost (`UNCHECKED CONVERSION` is loophole)

    (Java, C# is similar)

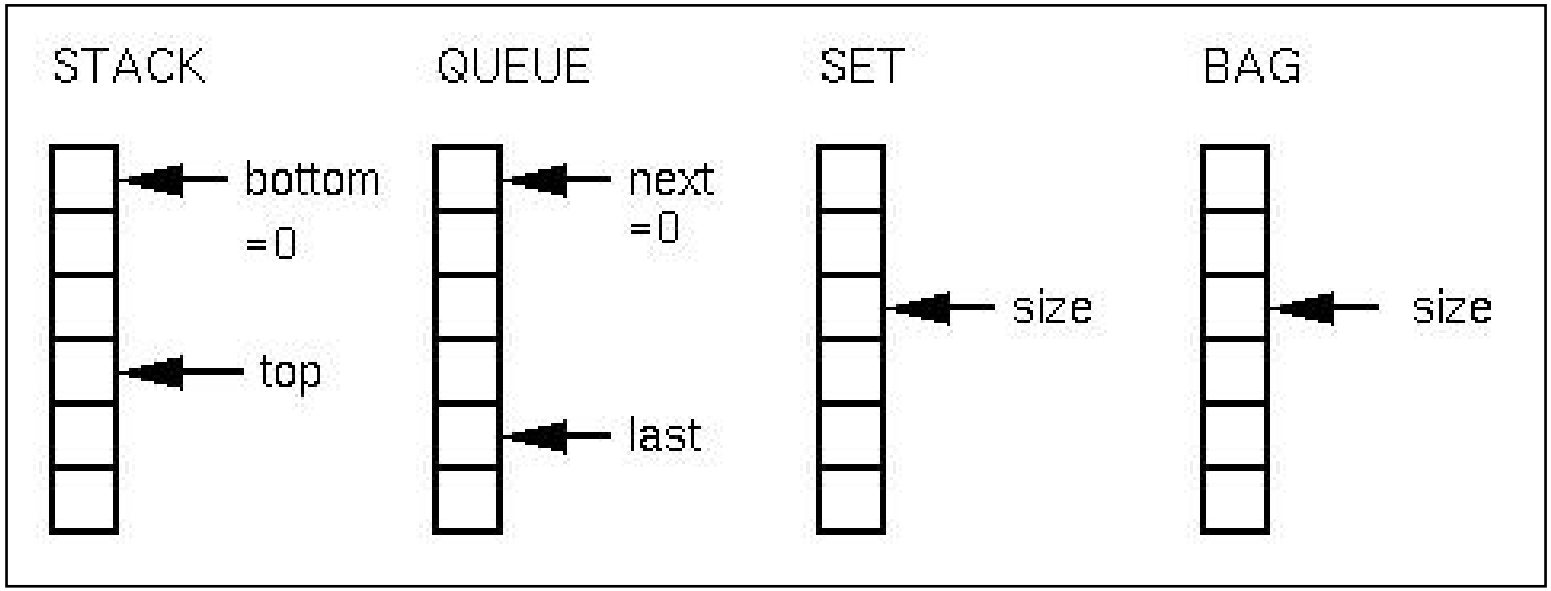
Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Equality

Consider the following 4 data types and their representations, all can be defined as:

```
struct {  int size;
          int storage[100];}
```



Meaning of A is equal to B

What does A=B mean for each data type?

stack: A.size = B.size

  For all i, A.storage[i]=B.storage[i]

queue: A.size = B.size

  For all i, A.storage[i]=B.storage[i]

set: A.size = B.size

  All A.storage[i] differ

  For all i, some j, A.storage[i]=B.storage[j]

  (i.e., B.storage is a permutation of A.storage)

bag: A.size = B.size

  B.storage is a permutation of A.storage

So equality of user types is a complex issue that cannot be built into the operator by the language definition (effectively).

Name Type Compatibility

- *Name type compatibility* means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not compatible with integer types

  ADA Examples:

```
type weekday is 1..7;
```

X

```
subtype weekday is Integer range 1..7;
```

✓

```
day : Integer;
monday : weekday;
day = monday
```

  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

Name Type Compatibility II

- Disallows differentiating between types with the same structure

```
type celsius = Float;          OK
     fahrenheit = Float;

type celsius is new Float;  NOT ALLOWED
type fahrenheit is new Float;
```

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
- ADA example

```
type Vector is array (Integer range <>) of Integer
Vector 1: Vector (1..10);
Vector 2: Vector (11..20);
```

Structure Type Compatibility (cont.)

- 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)

Type Compatibility – Cases

- C
  - Uses structure type compatibility for all types except structures and unions
    - Each declaration of `struct` and `union` creates a new type
    - `typedef` in C/C++ does not introduce a new type, any type defined by `typedef` is equivalent to the parent type
- C++
  - Uses name equivalence
- Fortran, COBOL does not allow users to define and name their types.

Implementation of subprogram storage

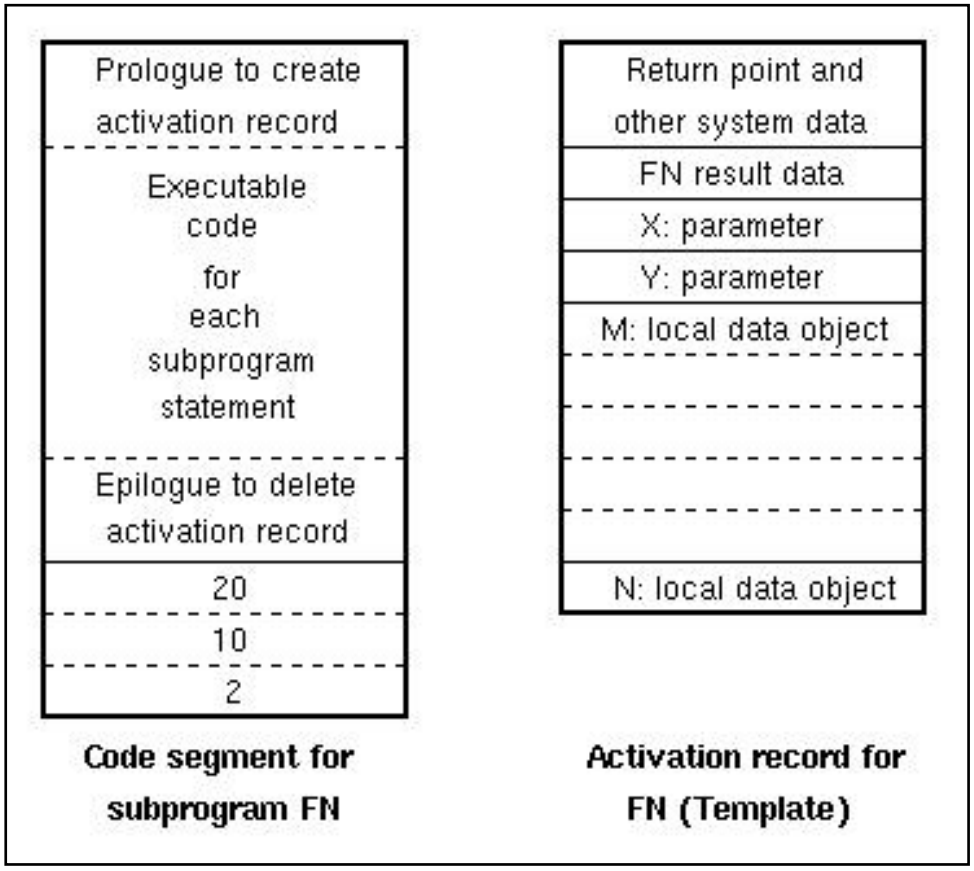
- Before looking at ADTs in detail, we first need to understand the usual method for subprograms to create local objects.
- Each subprogram has a block of storage containing such information, called an activation record.
- Consider the following C subprogram:

```
float FN( float X, int Y)
    const initval=2;
    #define finalval  10
    float M(10); int N;
    N = initval;
    if(N<finalval){ ... }
    return (20 * X + M(N)); }
```

- Information about procedure FN is contained in its activation record.

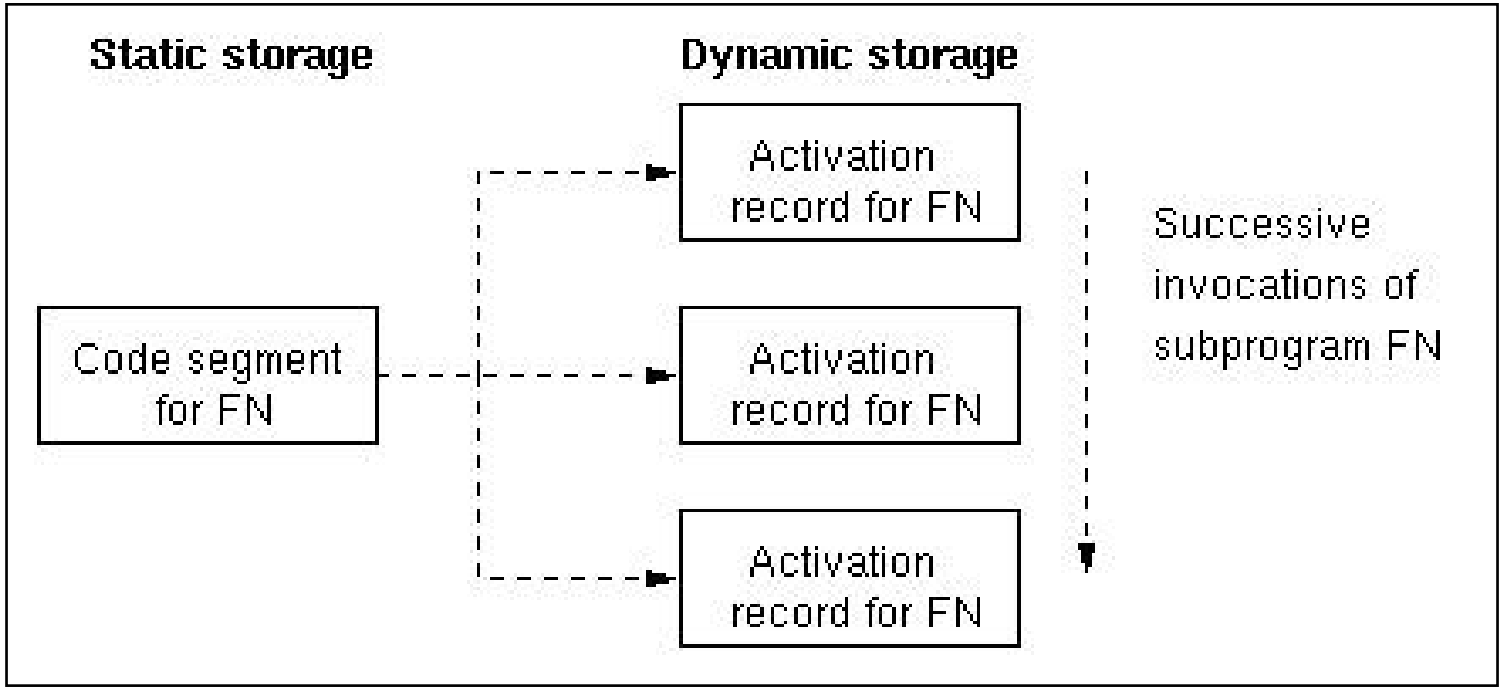
Activation records

- Above left: Code produced by compiler for the execution of procedure FN.
- Above right: Data storage needed by procedure FN during execution. Compiler binds each identifier in FN with a storage location in activation record.



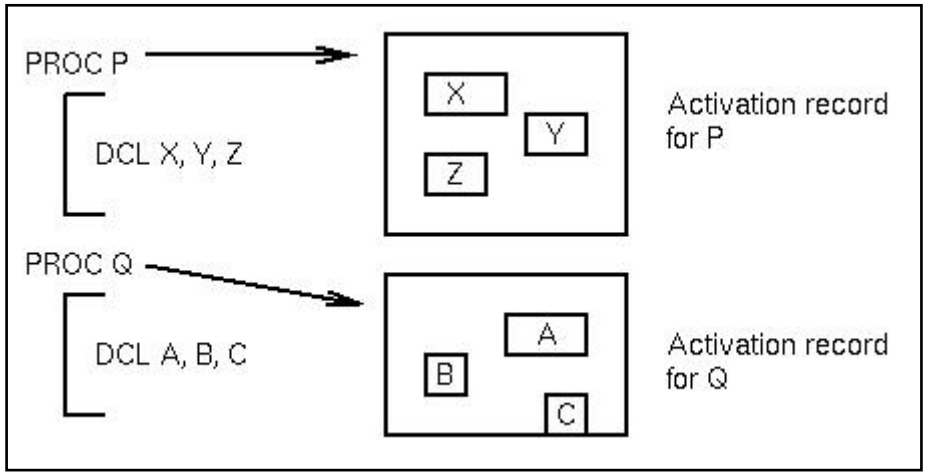
Dynamic nature of activation records

- Each invocation of FN causes a new activation record to be created. Thus the static code generated by the compiler for FN will be associated with a new activation record, each time FN is called. As we will see later, a stack structure is used for activation record storage.



Storage management

- Local data is stored in an area called an activation record (also called a stack frame).



- Address data is a two step process:

1. Get a pointer to the relevant activation record containing the declarations.
2. Find the offset in that activation record for the data item.

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

Static Scope

- ALGOL 60 introduced 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

Scope (continued)

- 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`**

Ada Example

```
procedure Big is
  X : Integer
  procedure Sub1 is
      begin –- of Sub1
       ... X ...
      end – of Sub1
 procedure Sub2 is
      X : Integer
      begin –- of Sub2
      ...
      end; -- of Sub2
 begin –- of Big
  ...
 end – of Big
```

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Examples:

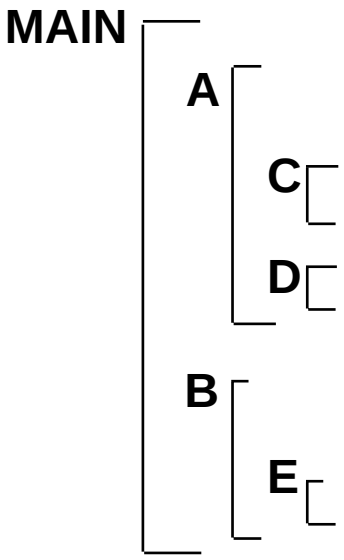
```
C and C++: for (...) {
               int index;
               ...
           }
Ada:  declare LCL : FLOAT;
      begin
           ...
      end
```

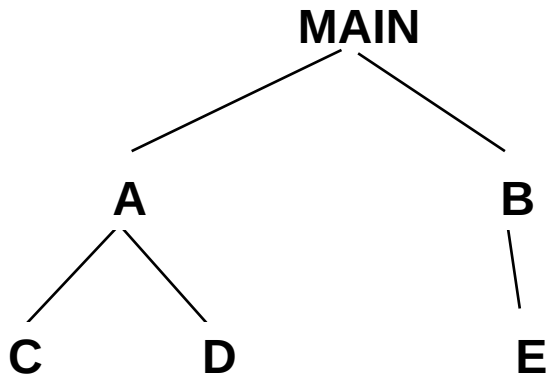
C++ Example

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

Evaluation of Static Scoping

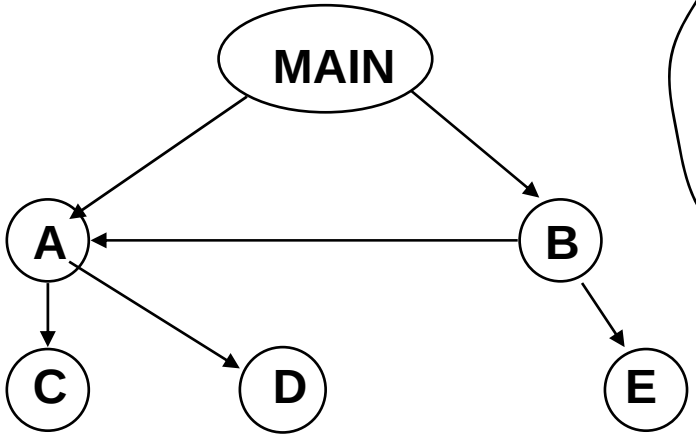
- Assume in an application APP
  MAIN calls A and B
  A calls C and D
  B calls A and E



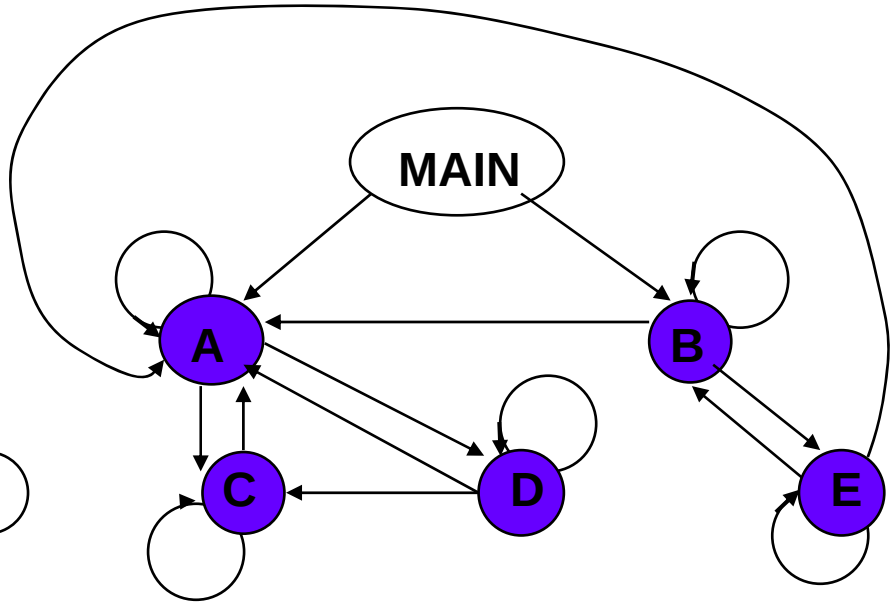


Static Scope Example

desirable calls specific
to the application APP



all possible calls



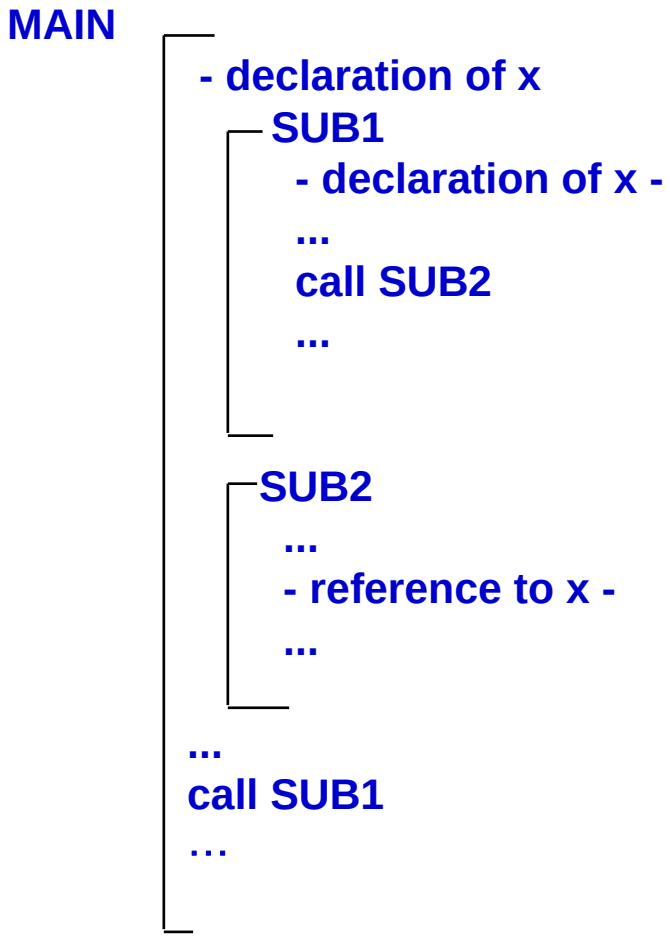
Static Scope (continued)

- Suppose the spec 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
- Overall: static scoping often encourages many globals

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



MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Scope Example

- Static scoping
  - Reference to x is to MAIN's x
- Dynamic scoping
  - Reference to x is to SUB1's x
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability

Scope and Lifetime

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

Lifetime

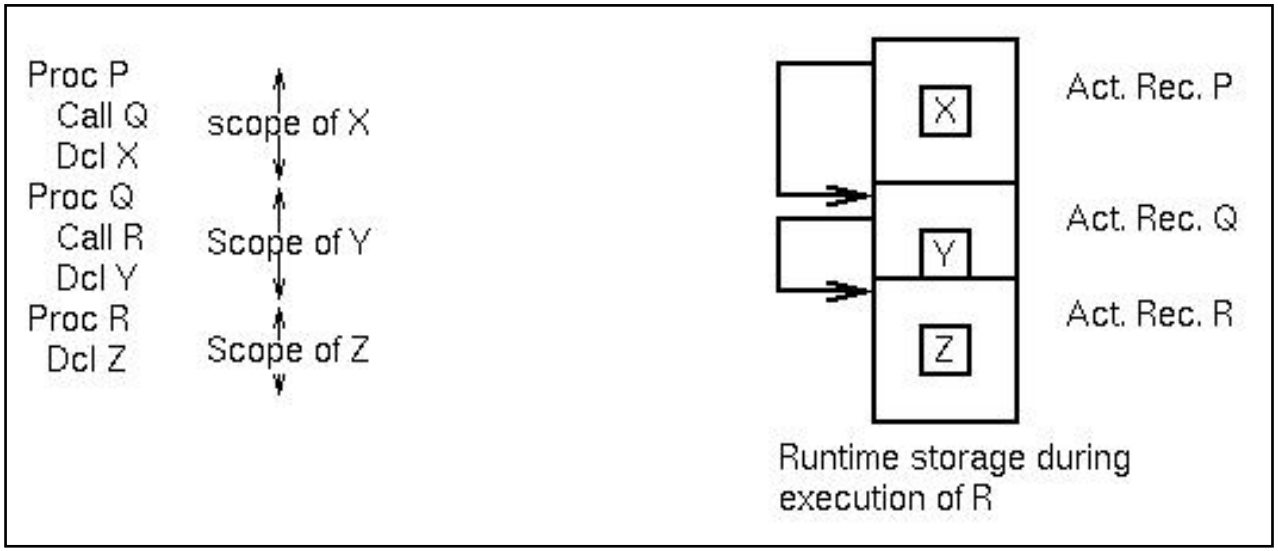
IMPORTANT:

1. The activation record pointer is determined at during program execution. It occurs frequently, so it should be an efficient calculation.
2. The offset within an activation record is fixed and determined during compilation. It should require no – or minimal – calculations.

- The lifetime of a variable is that period during program execution when the storage for the declared variable exists in some activation record (i.e., from the time the activation record for the declaring procedure is created until that activation record is destroyed).
- The scope of a variable is the portion of the source program where the data item can be accessed (i.e., the name of the item is a legal reference at that point in the program).

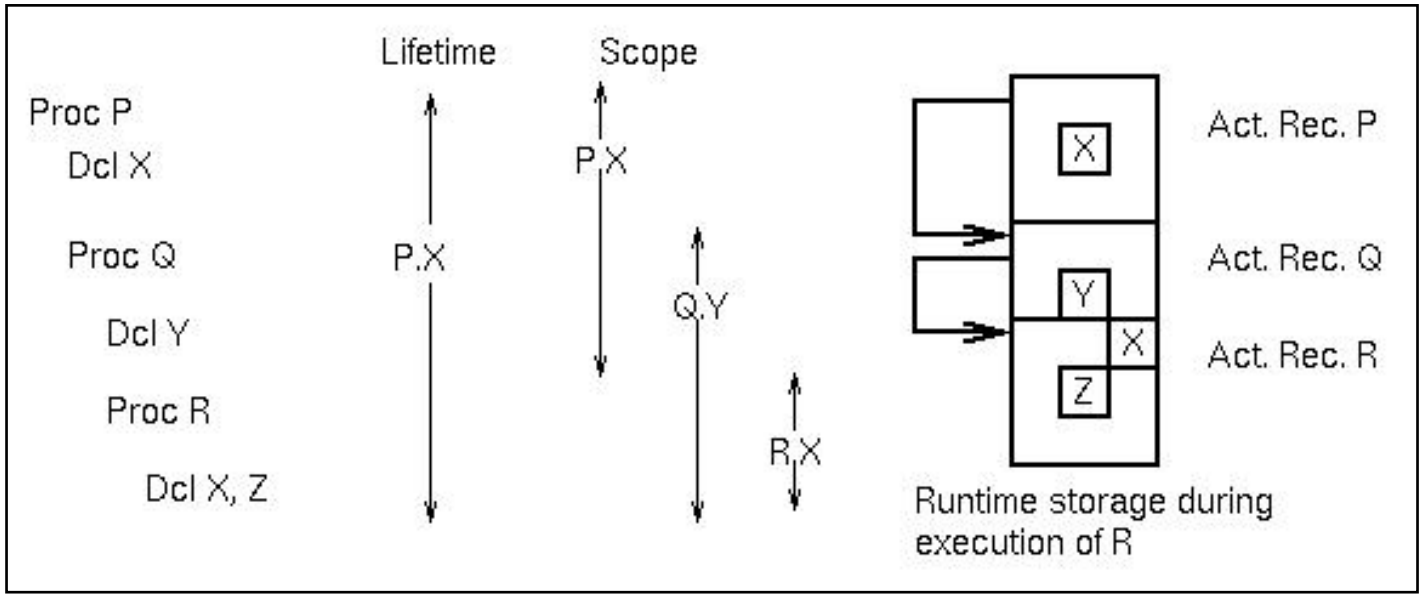
Lifetimes (continued)

- Two models of data lifetimes:
- Static: Based upon the static structure of a program
- Dynamic: Based upon the execution of a program
- In general, static lifetimes are implemented in stacks; dynamic lifetimes implemented in heaps.



Lifetimes of variables

- Lifetime of variable X is the period when the activation record for P exists.
- Note that we can change the scope, but the not lifetime of a variable by nesting procedures:



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

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 (called *manifest constants*) or dynamic
- Languages:
  - FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called *initialization*
- Initialization is often done on the declaration statement, e.g., in Java

```
int sum = 0;
```

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors