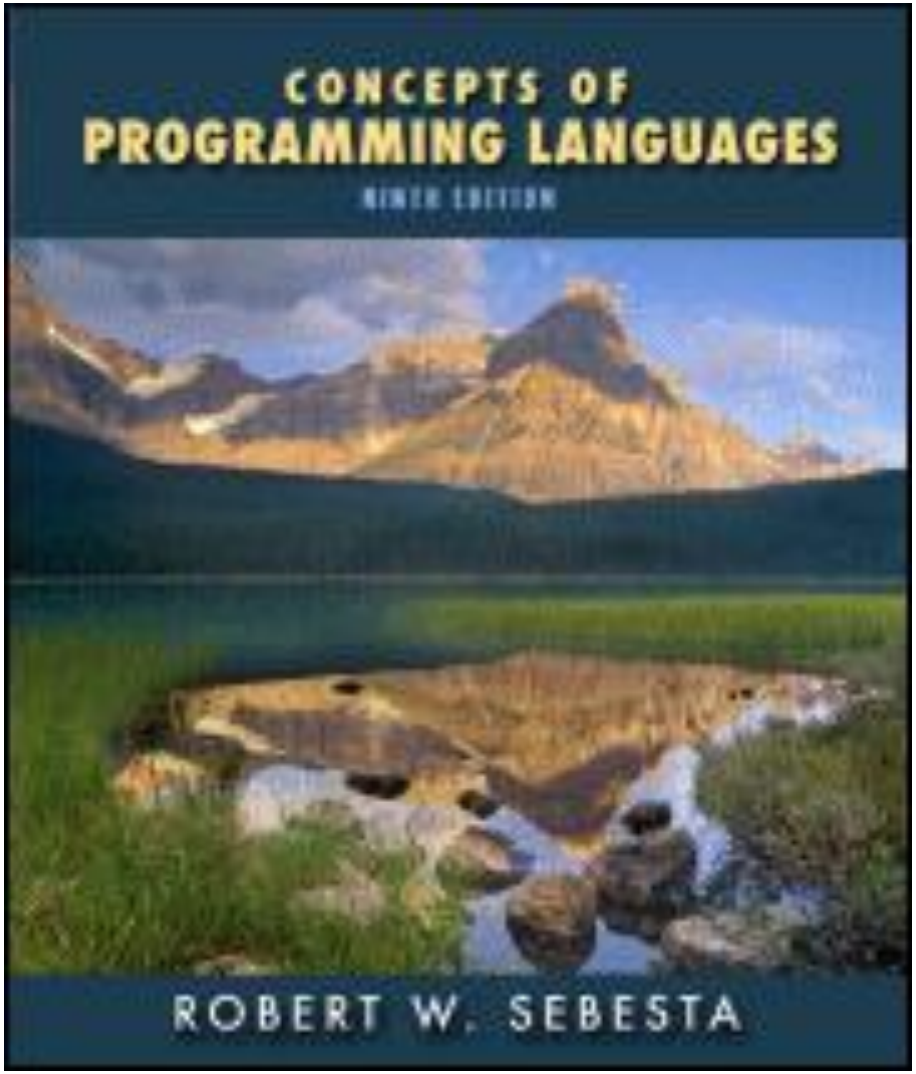
Chapter 5

Names, Bindings, and Scopes



Ch05 – Names, Bindings, Type checking, and Scopes

5.1 Introduction*

5.2 Names*

5.3 Variables*

5.4 The Concept of Binding

5.5 Scope

5.6 Scope and Lifetime*

5.7 Referencing Environments*

5.8 Named Constants*

5.4 The Concept of Binding

- Binding
  - A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Possible binding time
  - 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/C++ or Java
  - Load time -- bind a C/C++ static variable
  - Runtime -- bind a nonstatic local variable to a memory cell

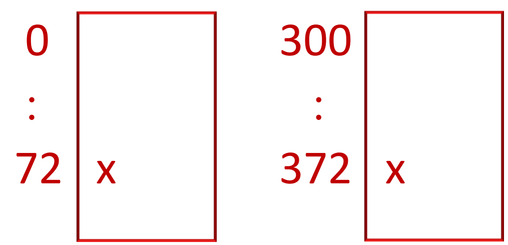
5.4 The Concept of Binding

- Static and dynamic bindings
  - 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
- Storage bindings and lifetime (5.4.3)
  - Static storage binding: static variables
  - Dynamic storage binding: stack-dynamic variables, explicit and implicit heap-dynamic variables
  - The lifetime of a variable is the time during which it is bound to a particular memory cell (i.e. from allocation to deallocation).

5.4 The Concept of Binding

- Static variables
  - Bound to memory cells before program execution begins and bound to the same memory cell throughout program execution,
    e.g. all FORTRAN 77 variables, C/C++ local static variables. C++ static data members
  - Advantages
    efficiency (direct addressing)
    history-sensitive subprogram support
  - Disadvantage
    lack of flexibility (no recursion)

memory



$x+1 \Rightarrow [72] + 1 \Rightarrow [372] + 1$

compiler loader

5.4 The Concept of Binding

- Stack-dynamic variables
  - Storage bindings are created for variables when or before their declarations are elaborated within a block. Storage is deallocated on exiting the block. e.g. C/C++ local auto variables.

```
void p(int n)
{
    int x=n;      // before
    int a[n+2];  // when
}
```

  - Advantage: allow recursion
  - Disadvantages
    Overhead of allocation and deallocation
    Subprograms cannot be history sensitive
    Inefficient references (indirect addressing)

5.4 The Concept of Binding

- Explicit heap-dynamic variables
  - Allocated and deallocated by explicit instructions specified by the programmer (or deallocated by garbage collector)
    e.g. C++ new and delete
    e.g. Java objects, new and garbage collector
  - Referenced only through pointers or references
  - Advantage
    flexibility: provide for dynamic data structures
  - Disadvantage
    inefficient, due to heap storage management
    unreliable, due to pointers

5.4 The Concept of Binding

- Implicit heap-dynamic variables
  - Bound to heap storage when they are assigned values
    Deallocated by garbage collector
    e.g. all strings and arrays in Perl and JavaScript
    @a = (1..5);
    e.g. Scheme, ML
    (define x '(a b c d e))
    val x = [2,3,4];
  - Advantage: flexibility
  - Disadvantages: Inefficient

5.4 The Concept of Binding

- Type bindings (5.4.2)
  - Static type binding

    The type of a variable is specified at compile time by:
    - explicit declaration: C, C++, Java
    - implicit declaration: Fortran, Perl
    - type inference: SML, Haskell, C++11
  - Dynamic type binding

    The type of a variable is specified at run time when it is assigned a value, e.g. JavaScript, Scheme, Perl

    (define x 1)

    (set! x "snoopy")

5.4 The Concept of Binding

Type inference (5.4.2.3, Supplementary)

- Hindley-Milner type system
  - Type inference rules

$$\frac{e1 : t1 \quad e2 : t2}{(e1,e2) : t1 * t2}$$

$$\frac{e1 : bool \quad e2 : t \quad e3 : t}{\text{if } e1 \text{ then } e2 \text{ else } e3 : t}$$

$$\frac{e1 : t1 \rightarrow t2 \quad e2 : t1}{e1\ e2 : t2}$$

$$\frac{x : t1 \quad e : t2}{\lambda x.e : t1 \rightarrow t2}$$

Lambda calculus	λx.e
SML	fn x => e
Scheme	(lambda (x) e)

5.4 The Concept of Binding

- Example - λx.x+1 ≡ λx.+(x,1)

  Type assignment { + : int*int → int, 1 : int, and so on }

$$\dfrac{+ : t2 \quad \dfrac{x : t1 \quad 1 : int}{(x,1) : t1*int}}{\dfrac{+(x,1) : t3}{\lambda x.+(x,1) : t4}}$$

$$t2 = int*int \to int$$
$$t2 = t1*int \to t3$$
$$t4 = t1 \to t3 = int \to int$$

  These equations can be solved by unification algorithm.

- Example - λf.λg.λx.f (g x)

$$\dfrac{\dfrac{f : t4 \quad \dfrac{g : t1 \quad x : t2}{g\ x : t3}}{f\ (g\ x) : t5}}{\lambda f.\lambda g.\lambda x.f\ (g\ x) : t6}$$

$$t1 = t2 \to t3$$
$$t4 = t3 \to t5$$
$$t6 = t4 \to t1 \to t2 \to t5$$
$$\therefore t6 = (t3 \to t5) \to (t2 \to t3) \to t2 \to t5$$

5.4 The Concept of Binding

- λ-bound type variables are not generic.
  - All occurrences of a λ-bound type variable must have the same type.
  - Example – λf.(f 2,f 3) is typable.

    id = λx.x : t → t

    (λf.(f 2,f 3)) id = (2,3)

    sq = λx.x*x : int → int

    (λf.(f 2,f 3)) sq = (4,9)

    In either case, both occurrences of f have the type int → int

$$\dfrac{\dfrac{\dfrac{f : t1 \quad 2 : int}{f\ 2 : t2} \qquad \dfrac{f : t1 \quad 3 : int}{f\ 3 : t3}}{(f\ 2, f\ 3) : t4}}{\lambda f.(f\ 2, f\ 3) : t5}$$

$$\begin{aligned} t1 &= int \to t2 \\ t1 &= int \to t3 \\ t4 &= t2 * t3 \\ t5 &= t1 \to t4 \\ &= (int \to t2) \to t2 * t2 \end{aligned}$$

5.4 The Concept of Binding

- Example

  $\lambda f.(f\ 2, f\ true)$ is untypable.

  $\lambda f.(f\ 2, f\ true))\ id = (2, true)$ $\qquad\qquad$ $\lambda f.(f\ 2, f\ true))\ sq$ X

  1st occurrence has the type $int \rightarrow int$

  2nd occurrence has the type $bool \rightarrow bool$

  Lesson: The type system is conservative.

$$\dfrac{\dfrac{\dfrac{f : t1 \quad 2 : int}{f\ 2 : t2} \qquad \dfrac{f : t1 \quad true : bool}{f\ true : t3}}{(f\ 2, f\ true) : t4}}{\lambda f.(f\ 2, f\ true) : t5}$$

$t1 = int \rightarrow t2$

$t1 = bool \rightarrow t3$

$t4 = t2 * t3$

$t5 = t1 \rightarrow t4$

5.4 The Concept of Binding

- Example – S ≡ λx.λy.λz.x z (y z)

$$\dfrac{\dfrac{\dfrac{x : t1 \quad z : t2}{x\ z : t4} \qquad \dfrac{y : t3 \quad z : t2}{y\ z : t5}}{x\ z\ (y\ z) : t6}}{\lambda x.\lambda y.\lambda z.x\ z\ (y\ z) : t7}$$

$$t1 = t2 \rightarrow t4$$
$$t3 = t2 \rightarrow t5$$
$$t4 = t5 \rightarrow t6$$
$$t7 = t1 \rightarrow t3 \rightarrow t2 \rightarrow t6$$

$$\therefore t7 = (t2 \rightarrow t5 \rightarrow t6) \rightarrow (t2 \rightarrow t5) \rightarrow t2 \rightarrow t6$$

- Example – Self-application λx.x x is untypable

$$\dfrac{\dfrac{x : t1 \quad x : t1}{x\ x : t2}}{\lambda x.x\ x : t3}$$

$$t1 = t1 \rightarrow t2 = (t1 \rightarrow t2) \rightarrow t2 = ((t1 \rightarrow t2) \rightarrow t2) \rightarrow t2 = \ldots$$

t1 is an infinite type

5.4 The Concept of Binding

- let-bound type variables are generic.
  - Different occurrences of a let-bound type variable may have different types, provided that it isn't also an enclosing λ-bound type variable.
  - Example

    let f = λx.x in (f 3,f true) end: int*bool

    f : t → t has two instances:

    int → int   in f 3

    bool → bool   in f true

    Note: The let expression and λf.(f 2,f true)) id have the same computation. But, the latter is ill-typed.

    Note: λg.let f = g in (f 3,f true) is untypable.

5.4 The Concept of Binding

- Example (Cont'd)

$$\dfrac{x : t1}{\lambda x.x : t2} \qquad \dfrac{\dfrac{f : t3 \quad 2 : int}{f\ 2 : t4} \qquad \dfrac{f : t5 \quad true : bool}{f\ true : t6}}{(f\ 2, f\ true) : t7}$$

$t2 = t1 \rightarrow t1$ $\qquad$ $t3 = t8 \rightarrow t8$ $\qquad$ t3 is an instance of t2

$t3 = int \rightarrow t4$

$t5 = t9 \rightarrow t9$ $\qquad$ t5 is an instance of t2

$t5 = bool \rightarrow t6$

$t7 = t4 * t6 = int * bool$

- Example

If $f = \lambda x.x : t \rightarrow t$ has been defined in top-level, the type of (f 2,f true) is inferred in exactly the same way.

5.4 The Concept of Binding

- Example – S K K 3 = K 3 (K 3) = 3

$S \equiv \lambda x.\lambda y.\lambda z.x\ z\ (y\ z) :\ (t1 \rightarrow t2 \rightarrow t3) \text{ -> } (t1 \rightarrow t2) \rightarrow t1 \rightarrow t3$

$K \equiv \lambda x.\lambda y.x :\ t1 \rightarrow t2 \rightarrow t1$

$$\dfrac{\dfrac{\dfrac{S : t1 \quad K : t2}{S\ K : t3} \qquad K : t4}{S\ K\ K : t5} \qquad 3 : int}{S\ K\ K\ 3 : t6}$$

$$t1 = t2 \rightarrow t3$$
$$t3 = t4 \rightarrow t5$$
$$t5 = int \rightarrow t6$$

$t1 = (t11 \rightarrow t12 \rightarrow t13) \rightarrow (t11 \rightarrow t12) \rightarrow t11 \rightarrow t13$

$t2 = t21 \rightarrow t22 \rightarrow t21$

$t4 = t41 \rightarrow t42 \rightarrow t41$

$t1 = t2 \rightarrow t4 \rightarrow int \rightarrow t6$

$\quad = (t21 \rightarrow t22 \rightarrow t21) \rightarrow (t41 \rightarrow t42 \rightarrow t41) \rightarrow int \rightarrow t6$

It follows that $t6 = t13 = t21 = t11 = int$

5.4 The Concept of Binding

- letrec-bound type variables
  - suppose f : t

    letrec f = λx. .... f.... in ..... f ..... end

    Like let bound variable, f : t1, where t1 is an instance of t

    f is being defined, lying in between λ-bound and let bound

    Two choices

    1. f : t   This is chosen by the Hindley-Milner type system.
    2. f : t2, where t2 is an instance of t

5.4 The Concept of Binding

- Example – f = λn.if n=0 then 1 else n*f(n-1)

  $= \;\; : int*int \rightarrow bool$

  $*,- : int*int \rightarrow int$

$$\dfrac{\dfrac{\dfrac{= : t8 \quad \dfrac{n : t1 \quad 0 : int}{(n,0) : t1*int}}{n = 0 : t9} \qquad 1 : int \qquad \dfrac{* : t6 \quad \dfrac{n : t1 \quad \dfrac{f : t3 \quad \dfrac{- : t2 \quad \dfrac{n : t1 \quad 1 : int}{(n,1) : t1*int}}{n\text{-}1 : t4}}{f(n\text{-}1) : t5}}{(n,f(n\text{-}1)) : t1*t5}}{n*f(n\text{-}1) : t7}}{\text{if } n=0 \text{ then } 1 \text{ else } n*f(n\text{-}1) : t10}}{f = \lambda n.\text{if } n=0 \text{ then } 1 \text{ else } n*f(n\text{-}1) : t3}$$

Clearly, that t1 = t4 = t5 = t7 = t10 = int and t9 = bool

So, t3 = t4 → t5 = int → int

Also, t3 = t1 → t10 = int → int

5.4 The Concept of Binding

- Typed languages
  - Languages that use static type binding
  - Also called statically or strongly typed languages
  - Variables have types
  - Perform static type checking
  - Pro

    Type safe: Type errors are all detected at compile time

    Efficiency: No type checking code at run time
  - Con

    Inflexible: Variables have fixed types

    e.g. fn x => x x is illegal in ML.

5.4 The Concept of Binding

- Untyped languages
  - Languages that use dynamic type binding
  - Also called typeless langauges/dynamically or weakly or loosely typed languages
  - Variables have no types, but values have types
  - Perform dynamic type checking
  - Con

    Type unsafe: Runtime type errors

    Inefficiency: Type checking code at run time
  - Pro

    Flexible: Variables don’t have fixed types

    e.g. (lambda (x) (x x)) is legal in Scheme, e.g. $(\lambda x.xx)\ (\lambda x.x)\ 7$

5.5 Scope

- Scope
  - The *scope* of a variable (,function, or type) is the range of program text over which it is visible.
  - Static scoping (Lexical scoping)
    - Determined by the layout of program at compile time
    - Search the "blocking" sequence until the variable is found or a compile time error occurs.
  - Dynamic scoping (Fluid scoping)
    - Determined by the calling sequence at run time.
    - Search the calling sequence until the variable is found or a run time error occurs.

5.5 Scope

- Example

```
procedure Big is
    var X
    procedure Sub1 is
        var X
    begin Sub2; end;
    procedure Sub2 is
    begin X end;
begin
    Sub1; Sub2;
end;
```

Static scoping

Sub2.X is Big.X

Dynamic scoping

Big $\Rightarrow$ Sub1 $\Rightarrow$ Sub2

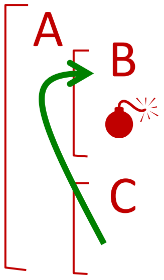
Sub2.X is Sub1.X

Big $\Rightarrow$ Sub2

Sub2.X is Big.X

5.5 Scope

- Dynamic scoping
  - Only a few languages are dynamically scoped, e.g. APL, SNOBOL4, old-style LISP.
  - Exception handling mechanisms of modern languages use dynamic scoping.
    Reason: Callers should be responsible for the exception.
  - Con
    - Hard to read
    - Must be dynamically type-checked
    - May take a longer time to search a variable
      ∵ A calling sequence may be lengthy.
    - Variables names must be stored and compared.



5.5 Scope

- Pro
  - Help communication between subprograms

    void p() { … n … }

    void q() { int n; p(); }   // n needn't be passed to p

- Dynamic scoping and dynamic type checking

  Approach 1

  Find the first matched name in the calling sequence.

  If it doesn't pass type checking or no matched name exists, error, e.g. old-style LISP

  E.g. Under this approach, the program on the next page is erroneous.

5.5 Scope

Approach 2

Find the first matched name in the calling sequence that passes type checking; if such a name does not exist, error.

```
void r() { throw 777; }    // treat it as catch(777);
void q() { r(); }
void p()
{
    try { q(); }
    catch (string s) { cout << s; }
    catch (char c) { cout << c; }
}
int main() {  try { p(); } catch (int x) { cout << x; } }
```

5.5 Scope

- Static and dynamic scoping in Perl
  - Statically scoped variables: my

```
$x=0;
sub sub1 { my $x=1; sub2(); }
sub sub2 { print $x; }              # 0 0
sub1;
sub2;
```

    - **my** creates a new, lexically-scoped variable.
    - It is invisible outside the block in which it is defined.

  Note:

  Were **my** removed, the output would be 1 1

5.5 Scope

- "Unusual" dynamically scoped variables

```
$x=0;
sub sub1 { local $x=1; sub2(); }  # localize global variable
sub sub2 { print $x; }            # 1 0
sub1;
sub2;
```

  - **local** doesn’t creates a new variable.
    Roughly, the subroutine sub1 behaves like:
    `sub sub1 { my $t=$x; $x=1; sub2(); $x=$t; }`
  - In effect, a **local** variable is invisible outside the block, but is visible in the subroutine called from the block.

5.5 Scope

- With **local**, the value of the global variable is restored even if the block exits abnormally.

```
$x=0;
sub sub1 { local $x=1; goto A;  }
sub1;
A: print $x,"\n";                    # 0
```

- Dynamic scoping for labels

```
sub sub1 { sub2(); A: print 1; }   # 1
sub sub2 { goto A; }
sub1;
A:;
```

5.5 Scope

- Static scoping
  - Classification of statically scoped languages
    - Monolithic block structure

      Only one global block, e.g. assembly languages
    - Flat block structure

      Only one or two nesting levels, e.g. Fortran, Prolog
    - Nested block structure

      Unrestricted nesting levels

      Two categories:

      Subprograms may be nested, e.g. Algol-like, Scheme, ML

      Subprograms can't be nested, e.g. C-like languages

5.5 Scope

- Classification of block
  - Subprogram (i.e. function and procedure)
  - Block statement, e.g. {…} in C/C++
  - Block expression, e.g. let expression in Schme, ML
  - Block declaration

```
local val pi=3.14 in
    fun area r = pi*r*r
end;
```

This is equivalent to

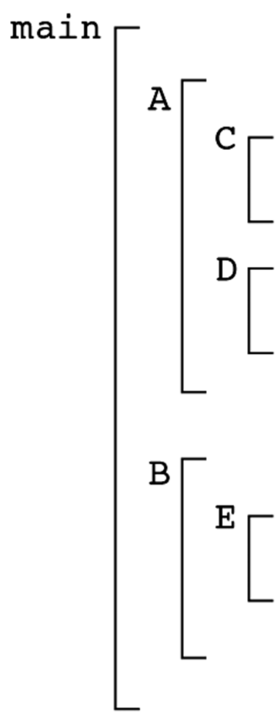
```
fun area r = let val pi=3.14 in
                 pi*r*r
             end;
```

5.5 Scope

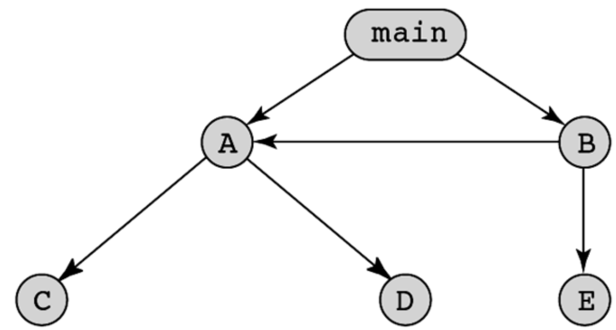
- Pro
  - Information hiding – local information is hidden
  - May be statically type-checked, e.g. ML
    (N.B. May also be dynamically type-checked, e.g. Scheme)
  - Usually take a shorter time to locate a variable
    ∵ In practice, a program has a small # of nesting levels.
  - Variable names needn't be stored and compared.
- Con
  - Usually allow too many subprogram calls and data accesses
    For example,

5.5 Scope

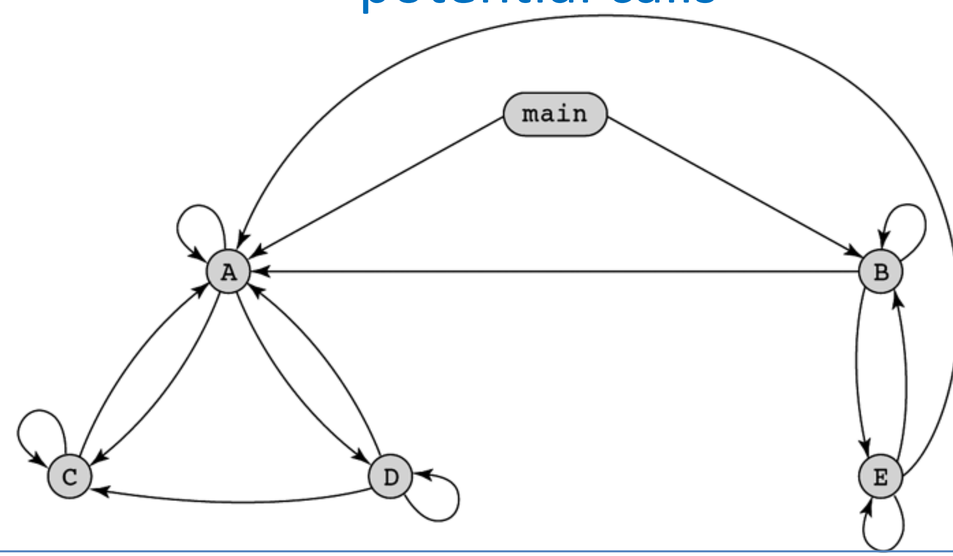
Program structure



desired calls



potential calls



5.5 Scope

- Con (Cont'd)

  Suppose the specification is changed

  D must now access some data in B

  Attempt

  - Put D in B, but then A can no longer call D
  - Let B enclose A, but then MAIN can no longer call A

  Solution

  - Move the data from B that D needs to MAIN

  But then all procedures can access them

  Overall: static scoping often encourages many globals

5.5 Scope

- Scope and name spaces (Supplementary)
  - Some languages divide names into separate name spaces.
    - Names in different name spaces don't interfere with each other.
    - The scope of names of different name spaces may be defined in a different manner.
  - Example

    Perl has 3 name spaces: $scalar, @array, and %hash

    The scope rules for these name spaces are the same.
  - Example – C++

    Category 1: Variables, functions, typedef names, and enumerators

5.5 Scope

Category 2: Class and enumeration names (ie. type names)

Both categories obey the same "local or global scope" rule.

```
typedef int a¹;
void b¹()
{
    enum a² {a¹,b¹} c¹=a¹;        // *
}
```

A category-1 name hides a category-2 name in the same scope. A hidden category-2 name can be accessed by specifying the type specifier, e.g. the starred line

```
enum a² {a¹,b¹};
enum a c=a;      // a c=a; is erroneous
```

5.5 Scope

Another example

```
void p()
{
    class a {};
    int a;
    class a b;
    a = 2;
}
```

```
class a {};
void p()
{
    int a;
    class a b;
    a = 2;
}
```

```
int a;
void p()
{
    class a {};
    a b;
    ::a = 2;
}
```

Class a and int a are in the same scope.

Class a and int a are in different scopes.

5.5 Scope

Category 3: Class members

This category obeys the "class scope" rule – the scope of a class member covers the entire class.

Pass 1: declaration
Pass 2: definition

```
enum { b=2 };
struct A {
    A(int a=b) : a(a) {}  // 2: default argument, ctor initializer
    void p() { a=8; }     // 2: function body inside the class
    void q();
    int a;                // 1: the order can't be reversed.
    enum { b=3 };
    int c[b];             // 2: region after the declaration
};
void A::q() { a=9; }      // 2: function body outside the class
```

5.5 Scope

Category 4: labels

This category obeys the "function scope" rule – the scope of a label covers the entire function.

```
int f(int n)
{
    int r=1;
    { n : if (n==0) goto r; }
    r*=n;
    n--;
    goto n;
 r : return r;
}
```