10 Bindings and scope

- Bindings and environments
- Scope and block structure
- Declarations

© 2012 David A Watt, University of Glasgow

Bindings and environments *(1)*

- The meaning of an expression/command depends on the declarations of any **identifiers** used by the expression/command.
- A **binding** is a *fixed* association between an identifier and an entity (such as a value, variable, or procedure).
- An **environment** (or **name-space**) is a set of bindings.

Bindings and environments *(2)*

- Each *declaration* produces some bindings, which are added to the surrounding environment.
- Each *expression/command* is interpreted in a particular environment. Every identifier used in the expression/command must have a binding in that environment.

Example: environments in a C program

- C program outline, showing environments:

```
extern int z;
extern const float c = 3.0e6;

void f () {
    …
}
```

{ c → the FLOAT value 3.0×10^6,
f → a VOID→VOID function,
g → a FLOAT→VOID function,
z → an INT global variable }

```
void g (float x) {
    char c;
    int i;
    …
}
```

{ c → a CHAR local variable,
f → a VOID→VOID function,
g → a FLOAT→VOID function,
i → an INT local variable,
x → a FLOAT local variable,
z → an INT global variable }

Scope

- The **scope** of a declaration (or of a binding) is the portion of the program text over which it has effect.
- In some early PLs (such as Cobol), the scope of every declaration was the whole program.
- In modern PLs, the scope of each declaration is controlled by the program's *block structure*.

Blocks

- A **block** is a program construct that delimits the scope of any declarations within it.
- Each PL has its own forms of blocks:
  - **C**: block commands ("`{ … }`"), function bodies, compilation-units.
  - **Java**: block commands ("`{ … }`"), method bodies, class declarations.
  - **Haskell**: block expressions ("**`let`** … **`in`** …"), function bodies, modules.
- A PL's **block structure** is the way in which blocks are arranged in the program text.

Monolithic block structure

- Some PLs (such as Cobol) have **monolithic block structure**: the whole program is a single block. The scope of every declaration is the whole program.

declaration of `x`

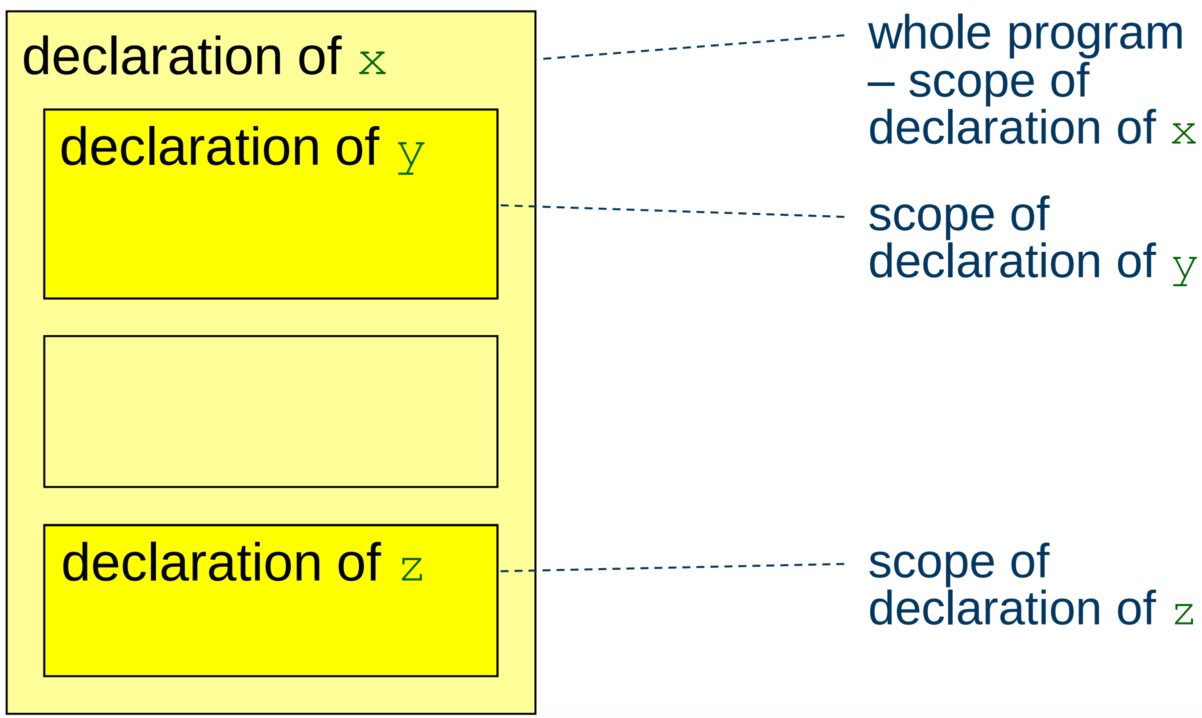
declaration of `y`

declaration of `z`

whole program – scope of declarations of `x`, `y`, `z`

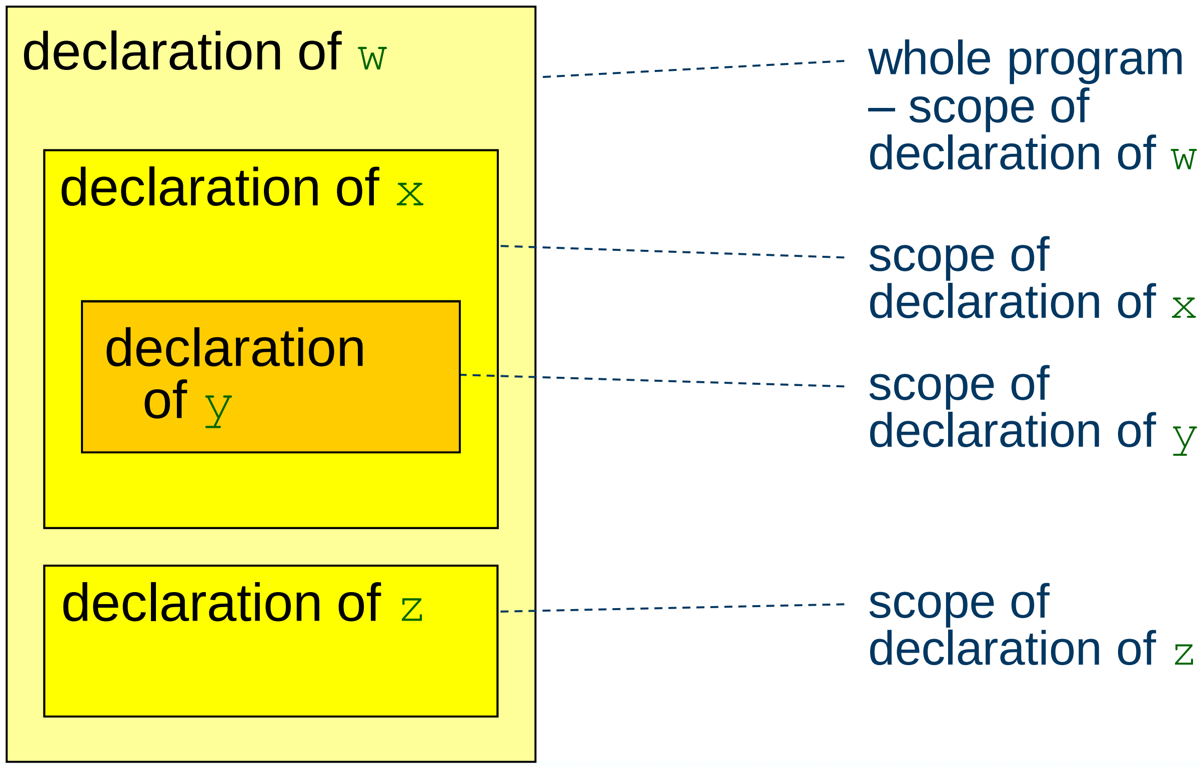
Flat block structure

- Some PLs (such as Fortran) have **flat block structure**: the program is partitioned into blocks, but these blocks may not contain inner blocks.



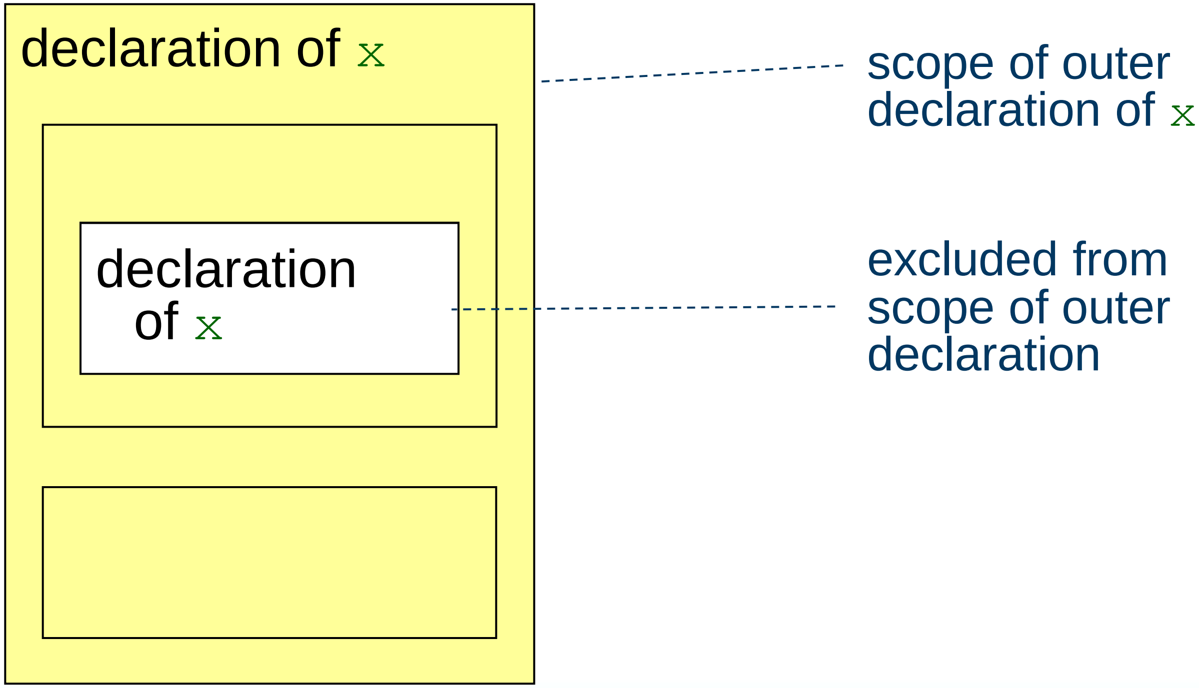
Nested block structure *(1)*

- Modern PLs have **nested block structure**: blocks may be nested freely within other blocks.



Nested block structure *(2)*

- With nested block structure, the scope of a declaration excludes any inner block where the same identifier is declared:



Example: C block structure

- C has flat block structure for functions, but nested block structure for variables:

```
extern int x1, x2;

void main () {
    int m1; float m2;
    … f(); …
}

void f () {
    float f1;
    while (…) {
        int f2;
        …
    }
    …
}
```

Binding and applied occurrences

- A **binding occurrence** of identifier `I` is an occurrence of `I` where `I` is bound to some entity *e*.
- An **applied occurrence** of identifier `I` is an occurrence of `I` where use is made of the entity *e* to which `I` is bound.
- If the PL is statically scoped *(see later)*, every applied occurrence of `I` should correspond to exactly one binding occurrence of `I`.

Example: binding and applied occurrences

- C program outline, showing binding occurrences and applied occurrences:

```
extern int z;
extern const float c = 3.0e6;

void f () {
    … c … z …
}

void g (float x) {
    int i;
    char c;
    … c … i … x … z …
}
```

Static *vs* dynamic scoping *(1)*

- A PL is **statically scoped** if the body of a procedure is executed in the environment of the procedure *definition*.
  - Then we can decide at compile-time which binding occurrence of an identifier corresponds to a given applied occurrence.
- A PL is **dynamically scoped** if the body of a procedure is executed in the environment of the procedure *call site*.
  - Then we cannot decide until run-time which binding occurrence of an identifier corresponds to a given applied occurrence, since the environment may vary from one call site to another.

Example: static scoping

- Program in a *statically* scoped PL (C):

```
const int s = 2;

int f (int x) {
    return x * s;
}

void g (int y) {
    print (f (y));
}

void h (int z) {
    const int s = 3;
    print (f (z));
}
```

The value of s here is always 2.

prints $2 \times$ y

prints $2 \times$ z

Example: dynamic scoping

- Similar program in a hypothetical *dynamically* scoped PL:

```
const int s = 2;

int f (int x) {
    return x * s;
}

void g (int y) {
    print (f (y));
}

void h (int z) {
    const int s = 3;
    print (f (z));
}
```

The value of s here depends on the call site.

prints 2 × y

prints 3 × z

Static *vs* dynamic scoping *(2)*

- Dynamic scoping fits badly with static typing.
  - In the previous slide, what if the two declarations of `s` had different types?
- Nearly all PLs (including Pascal, Ada, C, Java, Haskell) are statically scoped.
- Only a few PLs (such as Smalltalk and Lisp) are dynamically scoped.

Declarations *(1)*

- A **declaration** is a program construct that will be **elaborated** to produce binding(s).
  - A declaration may also have side effects (such as creating a variable).
- A **definition** is a declaration whose *only* effect is to produce binding(s).
  - A definition has no side effects.

Declarations *(2)*

- Simple declarations:
  - A **type declaration** binds an identifier to an existing or new type.
  - A **constant definition** binds an identifier to a value (possibly computed).
  - A **variable declaration** binds an identifier to a newly-created variable.
  - A **procedure definition** binds an identifier to a procedure.
  - And similarly for other entities (depending on the PL).

Declarations *(3)*

- Compound declarations:
  - A **sequential declaration** combines several sub-declarations, such that the later sub-declarations can use bindings produced by the earlier sub-declarations.
  - A **recursive declaration** is one that uses the bindings it produces itself.

Recursive declarations

- A recursive declaration is one that uses the bindings it produces itself.
- In almost all PLs, recursion is restricted to:
  - type (or class) declarations
  - procedure (or method) definitions.

Example: Java recursive declarations

- Java classes may be recursive.
- Java method definitions may be recursive.

```
class IntList {
    int head;
    IntList tail;

    static int length (IntList list) {
        if (list == null)
            return 0;
        else
            return 1 + length (list.tail);
    }
}
```

Example: C recursive declarations

- C struct type declarations may be recursive (but only via pointers).
- C function definitions may be recursive.

```
struct IntList {
    int head;
    struct IntList * tail;
}

int length (IntList * list) {
    if (list == NULL)
        return 0;
    else
        return 1 + length(list->tail);
}
```