Names, environment, blocks, scope rules

Based on the slides of Maurizio Gabbrielli

Names

- Name
  - Sequence of characters used to *denote* something

    - const PI = 3.14;
    - int x;
    - void F() {...};

Denated object:
The constant 3.14
A variable
The definition of F

- In languages names are often identifiers (alphanumeric tokens)
  - But they can also be other symbols (_,+,:=,...)
- The use of a name is used to indicate the denoted object

Denotable Objects

- Denotable Object
  - When an “object” can be associated with a name
  - Names defined by the user
    - variables, formal parameters, procedures (broadly), types defined by user, labels, modules, constants defined by user, exceptions
  - Language-defined names
    - Primitive types, primitive operations, predefined constants

Binding time: association between name and object

- Static
  - Language design
    - Primitive types, names for predefined operations and constants
  - Writing the program
    - Definition of some names (variables, functions etc.) whose link will be completed later
  - Compilation (+ linking and deployment)
    - Binding of some names (global var)
- Dynamic
  - Run time
    - Definitive link of all names not yet tied (e.g., local variables in blocks)

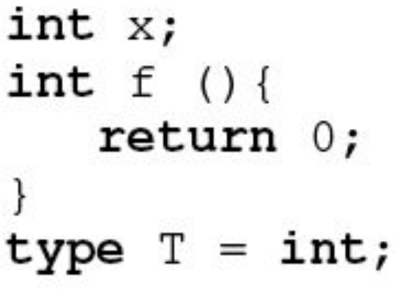
Environment

Environment:

A set of associations between names and objects that exist at run time at a specific point in the program and in a specific time of execution

Declaration:

Mechanism with which you create an association in environment



```
int x;
int f (){
   return 0;
}
type T = int;
```

Environment

The same name can denote distinct objects

- In different points of the program

Aliasing

- Different names denote the same object
  - Pointers (int *X, *Y; X = (int *) malloc (sizeof (int)); Y = X)
  - Passing by reference
  - ...

Blocks

- In modern languages the environment is **structured**
- Block:
  - Textual region of the program, identified by a beginning and an end signal, which may contain declarations local to that region
    - **`Begin... end`** Algol, Pascal
    - **`{...}`** C, Java
    - **`(...)`** Lisp
    - **`let... in... end`** ML
  - Anonymous (or in-line)
  - Associated with a procedure

Why blocks?

- Local name management

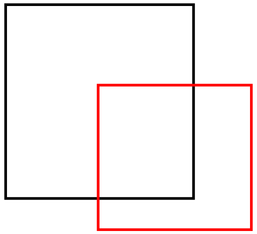
```
{int tmp = x;
 X = Y;
 y = tmp
}
```

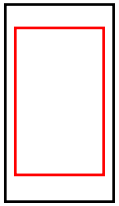
  - Everyone can choose the name he wants
  - Clarity

- With an appropriate memory allocation:
  - Optimize the memory occupation
  - Allow recursion

Nesting

- Overlapping blocks only if nested





- (Preliminary) Visibility rule
  - A local declaration to a block is visible in that block and in all the blocks nested therein, unless in the nested blocks a new declaration of the same name appears (which hides or mask the previous)

Environment Divisions

- The environment (in a specific block) can be divided into
  - **Local environment**: Bindings created at the entry of the block
    - Local variables
    - Formal parameters
  - **Non-local environment**: bindings inherited from other blocks
  - **Global Environment**: part of a non-local environment related to associations common to all blocks
    - Explicit declarations of global variables
    - Declarations of the outermost block
    - Associations exported by modules

Operations on the Environment

- Naming (Name-Object Association)
  - Block Local declaration
- Referencing (Object denoted by its name)
  - Using a name
- Disabling (Name-Object Association)
  - A declaration masks a name
- Reactivation (Name-Object Association)
  - Block exit with declaration that mask previous name
- Unnaming (Name-Object Association)
  - Block exit with local declaration

Operations on denotable objects

- Creating
- Access
- Modification (if object is editable)
- Destruction

- Creation and destruction of an object do not coincide with the creation and destruction of the bonds for it

Life

The **life of an object does not coincide with the life of the bindings** for that Object

- E.g., Object before and after of parameter

```
procedure P (var x:integer); begin … end
…
var a:integer;
…
P(a);
```

During P execution there is a link between x and an object that exists before and after that execution.

Life

- Life of Object Shorter than that of the binding
- E.g., Dynamic deallocation of memory

```
int *X, *Y;
...
X = (int *) malloc (sizeof (int));
Y = X;
...
free (X);
X = null;
```

After the `free` there is no object, but there is still a Dangling reference for it (`Y`):

Scope rules

- How should the visibility rule be interpreted?

  A local declaration to a block is visible in that block and in all the blocks nested therein, unless in the nested blocks a new declaration of the same name appears (which hides or mask the previous)

  – In the presence of procedures?

    That is, of blocks that are executed in different positions by their definition

Visibility rule & Procedures

```
{
    int x=10;
    void foo() {
        x++;
    }
    void fie() {
        int x=0;
        foo();
    }
    fie();
}
```

which x increments foo?

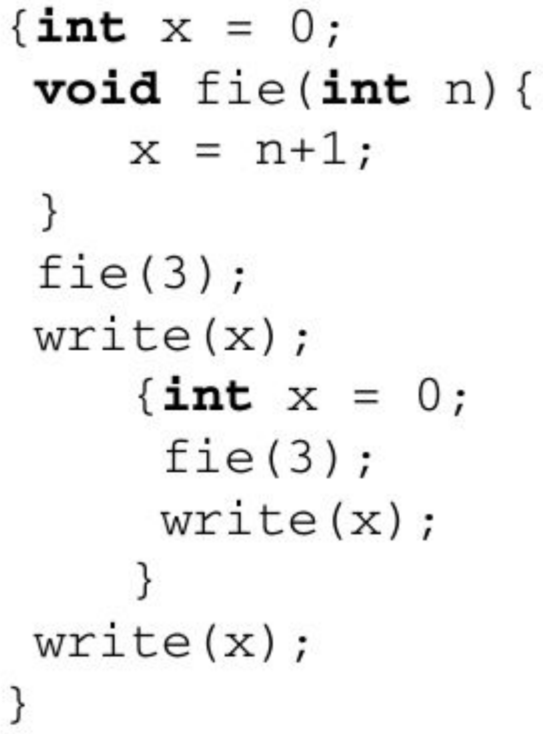
A non-local reference in a B block can be resolved:

In the block that *syntactically includes* B
In the block that is *run immediately before* by B

Static scope

Dynamic scope

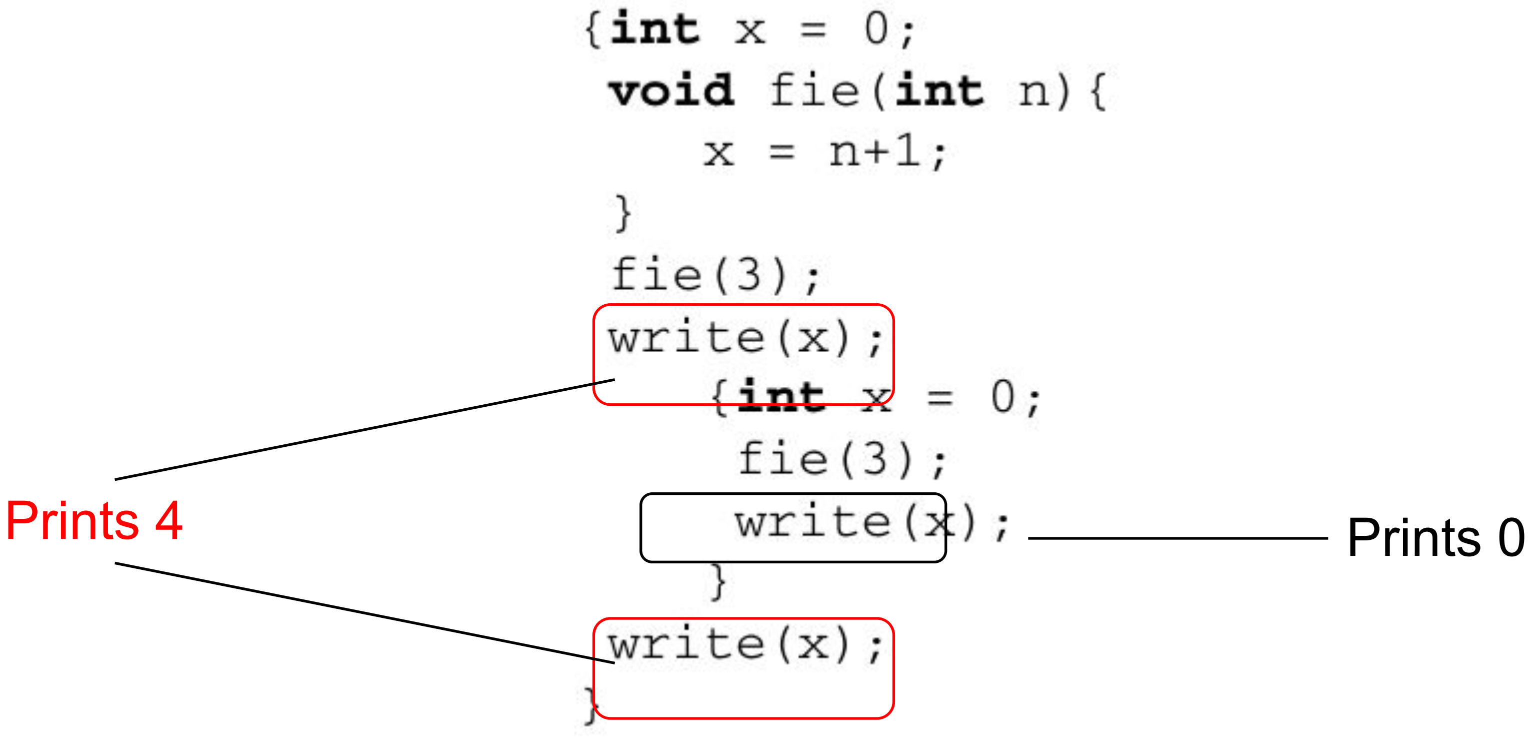
Example. Consider this program



```
{int x = 0;
 void fie(int n){
    x = n+1;
 }
 fie(3);
 write(x);
    {int x = 0;
     fie(3);
     write(x);
    }
 write(x);
}
```

Static Scope

- A non-local name is resolved in the block that encloses it



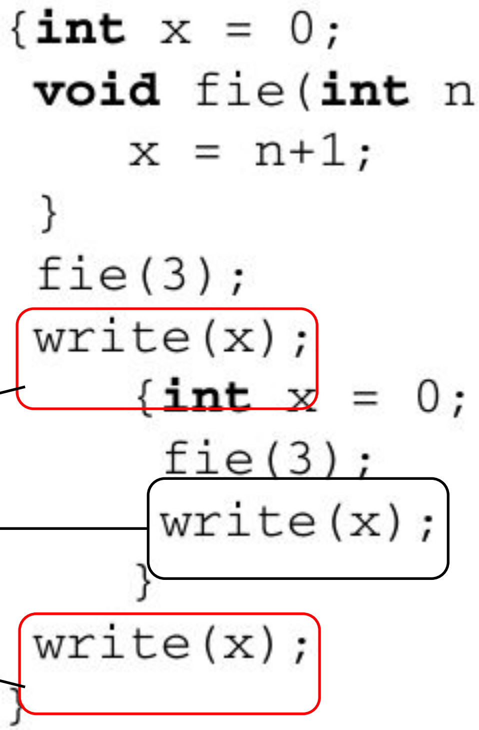
```
{int x = 0;
 void fie(int n){
    x = n+1;
 }
 fie(3);
 write(x);
    {int x = 0;
     fie(3);
     write(x);
    }
 write(x);
}
```

Prints 4

Prints 0

Dynamic Scope

- A non-local name is resolved in the block that has been activated more recently and not yet deactivated



```
{int x = 0;
 void fie(int n){
     x = n+1;
 }
 fie(3);
 write(x);
     {int x = 0;
      fie(3);
      write(x);
     }
 write(x);
}
```

Print 4

Static Scope: Independent from position

- The body of **`foo`** is part of the scope of the outermost x
- The call of **`foo`** is included in the scope of the innermost x
- **`foo`** can be called in many different contexts
- the only way in which **`foo`** can be uniquely compiled is that the reference to x is always the outermost one

```
{
  int x = 10;
  void foo () {
    x++;
  }
  void fie () {
    int x = 0;
    foo();
  }
  fie();
  foo();
}
```

The call of **`foo`** internal to **`fie`** and the one in the main access the same variable: the external x

Static Scope: independency from local names

Changing the name `y` in `x` in `fie`

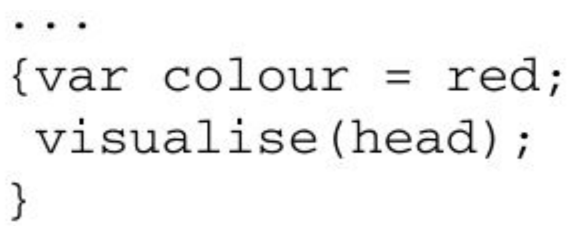
- Change the semantics of the program with dynamic scope
- Has no effect with static scope

```
{
  int x = 10;
  void foo () {
    x++;
  }
  void fie () {
    int x = 0;
    foo();
  }
  fie();
  foo();
}
```

Independence principle: consistent naming of a program's local names should not affect the semantics of the program itself

Dynamic Scope: Specializing a function

- **`visualise`** is a procedure that colors on video some object
- Color can be defined just before the call of the procedure



```
...
{var colour = red;
 visualise(head);
}
```

Static vs Dynamic Scoping

- Static Scoping
  - Complete information from the program text
  - Bindings are known at compile time
  - Principles of Independence
  - Conceptually more complex to implement but more efficient
  - Algol, Pascal, C, Java,...
- Dynamic Scoping
  - Information derived at run time
  - Often cause less readable programs
  - Conceptually simpler to implement, but less efficient
  - Lisp (some versions), Perl, Bash

Identify the Environment

- The environment is determined by
  - Rule of scoping (static or dynamic)
  - Specific rules, e.g.
    - when is a declaration visible in the block in which it appears?

Too see later

  - Rules for the parameter passing

Some specific rules

- Where is a declaration visible in the block in which it appears?
  - Starting from the declaration and until the end of the block

    Java: Declaring a variable

    ```
    {a = 1;   No!
     int a;
     … }
    ```

  - Always (so also *before*) of the declaration

    Java: Declaring a method

    - Allows mutually recursive methods

    ```
    {
        void f() {
          g(); Yes
        }
        void g() {
          f(); Yes
        }
     … }
    ```

Mutual recursion

Mutual recursion (functions, types) in languages where a name must be declared before being used?

- Release this constraint for functions and/or types
  - Java for methods
  - Pascal for pointer types

Pascal

```
type list = ^elem;
     elem = record
           info: integer;
           next: list;
          end
```

Java

```
    {
     void f() {
        g();
     }
     void g() {
        f();
     }
    }
```

- Incomplete definitions

Ada

```
type elem;
type list is access elem;
type elem is record
            info: integer;
            next: list;
      end
```

C

```
struct elem;
struct elem {
    int info;
    elem *next;
}
```

Suggested Exercises

- Chapter 4 from 1 to 5