

# Capitolo 6

I nomi e l'ambiente



# Nomi

- **nome**

- sequenza di caratteri usata per *denotare* qualche cos'altro

- `const pi = 3.14;`

- `int x;`

- `void f(){...};`

oggetto denotato:

la costante 3.14

una variabile

la definizione di f

nomi

- Nei linguaggi i nomi sono spesso **identificatori** (token alfa-numerici)
- L'**uso** di un nome serve ad indicare l'oggetto denotato
  - oggetti simbolici più facili da ricordare
  - astrazione (sia sui dati che sul controllo)

# Oggetti denotabili

- Oggetto *denotabile*
  - quando può essergli associato un nome
- Nomi definiti dall'utente
  - variabili, parametri formali, procedure (in senso lato), tipi definiti dall'utente, etichette, moduli, costanti definite dall'utente, eccezioni
- Nomi definiti dal linguaggio
  - tipi primitivi, operazioni primitive, costanti predefinite.
- Terminologia:
  - *Legame* (binding), o *associazione*, tra nome e oggetto

# Ambiente

## Ambiente:

insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione

## Dichiarazione:

meccanismo (implicito o esplicito) col quale si crea un'associazione in ambiente

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

# Ambiente, 2

Lo stesso nome può denotare oggetti distinti  
in punti diversi del programma

## Aliasing

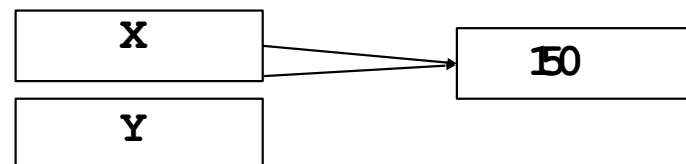
nomi diversi denotano lo stesso oggetto

passaggio per riferimento

puntatori

ecc.

```
int *X, *Y;           // X,Y puntatori a interi
X = (int *) malloc (sizeof (int));
                        // allocata la memoria puntata
*X = 5;                // * dereferenzia
Y=X;                   // Y punta alla stesso oggetto di X
*Y=10;
write(*X);
```



# Blocchi

- Nei linguaggi moderni l'ambiente è *strutturato*
- **Blocco:**
  - regione testuale del programma, identificata da un segnale di inizio ed uno di fine, che può contenere dichiarazioni *locali* a quella regione
    - `begin...end`                      Algol, Pascal
    - `{...}`                              C, Java
    - `(...)`                              Lisp
    - `let...in...end`                      ML
  - anonimo (o in-line)
  - associato ad una procedura

# Perché i blocchi

- Gestione locale dei nomi

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

- chiarezza
- ognuno può scegliere il nome che vuole

- Con un'opportuna allocazione della memoria (vedi dopo):

- ottimizzano l'occupazione di memoria
- permettono la ricorsione

# Annidamento

- Blocchi sovrapposti solo se annidati



- Regola di visibilità (preliminare)
  - Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che *nasconde*, o *maschera*, la precedente).



# Suddividiamo l'ambiente

- L'ambiente (in uno specifico blocco) può essere suddiviso in
  - **ambiente locale**: associazioni create all'ingresso nel blocco
    - variabili locali
    - parametri formali
  - **ambiente non locale** : associazioni ereditate da altri blocchi
  - **ambiente globale**: quella parte di ambiente non locale relativo alle associazioni comuni a tutti i blocchi
    - dichiarazioni esplicite di variabili globali
    - dichiarazioni del blocco più esterno
    - associazioni esportate da moduli ecc.

# Esempio

```
A: {int a = 1;

    B: {int b = 2;
        int c = 2;

        C: {int c = 3;
            int d;
            d = a+b+c;
            write(d)
        }

        D: {int e;
            e = a+b+c;
            write(e)
        }
    }
}
```

# Operazioni sull'ambiente

- **Creazione** associazione nome-oggetto denotato (naming)
  - dichiarazione locale in blocco
- **Riferimento** oggetto denotato mediante il suo nome (referencing)
  - uso di un nome
- **Disattivazione** associazione nome-oggetto denotato
  - entrata in un blocco con dichiarazione che maschera

**Riattivazione** associazione nome-oggetto denotato

  - uscita da blocco con dichiarazione che maschera
- **Distruzione** associazione nome-oggetto denotato (unnaming)
  - uscita da blocco con dichiarazione locale

# Operazioni sugli oggetti denotabili

- Creazione
  - Accesso
  - Modifica (se l'oggetto è modificabile)
  - Distruzione
- 
- Creazione e distruzione di un oggetto non coincidono con creazione e distruzione dei legami per esso

# Alcuni eventi fondamentali

1. Creazione di un oggetto
2. Creazione di un legame per l'oggetto
3. Riferimento all'oggetto, tramite il legame
4. Disattivazione di un legame
5. Riattivazione di un legame
6. Distruzione di un legame
7. Distruzione di un oggetto

Il tempo tra 1 e 7 è la **vita** (o il tempo di vita: *lifetime*)  
**dell'oggetto**

Il tempo tra 2 e 6 è la **vita dell'associazione**

# Tempo di vita

La vita di un oggetto *non* coincide con la vita dei legami per quell'oggetto

- Vita dell'oggetto più **lunga** di quella del legame: variabile passata ad una proc per riferimento (Pascal: var)

```
procedure P (var X:integer); begin... end;  
...  
var A:integer;  
...  
P(A) ;
```

Durante l'esecuzione di P esiste un legame tra X e un oggetto che esiste prima e dopo tale esecuzione.

# Tempo di vita 2

- Vita dell'oggetto più **breve** di quella del legame: area di memoria dinamica deallocata

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

Dopo la `free` non esiste più l'oggetto, ma esiste ancora un legame ("pendente") per esso (Y): *dangling reference*

# Regole di scope

- Come deve essere interpretata la regola di visibilità?

*Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che nasconde, o maschera, la precedente)*

- in presenza di procedure  
cioè di blocchi che sono eseguiti in posizioni diverse dalla loro definizione
- in presenza ambiente non locale (e non globale)



# Qual è lo scope?

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

quale x incrementa foo?

un riferimento non-locale in un blocco B può essere risolto:

nel blocco che *include sintatticamente* B

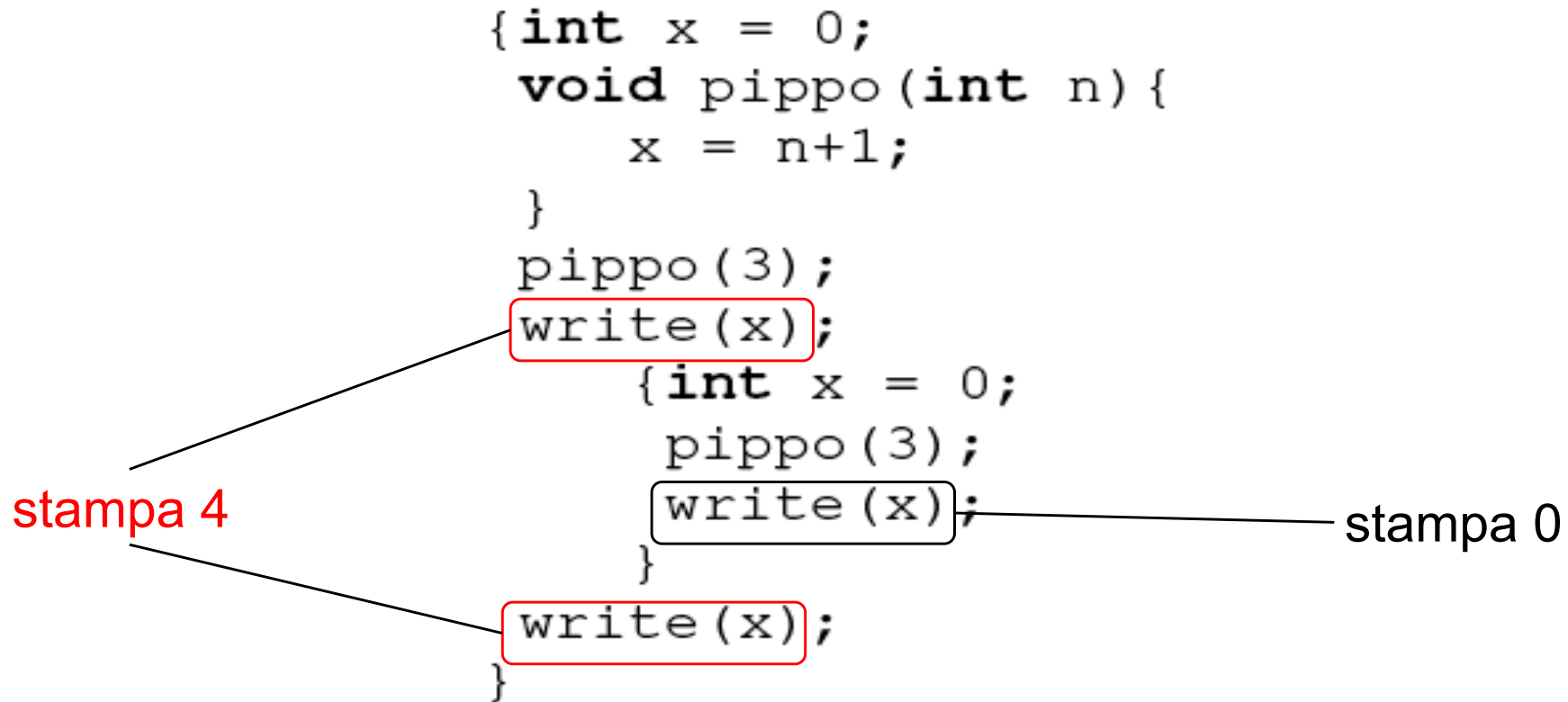
scope statico

nel blocco che è *eseguito immediatamente prima* di B

scope dinamico

# Scope statico

- Un nome non locale è risolto nel blocco che *testualmente* lo racchiude

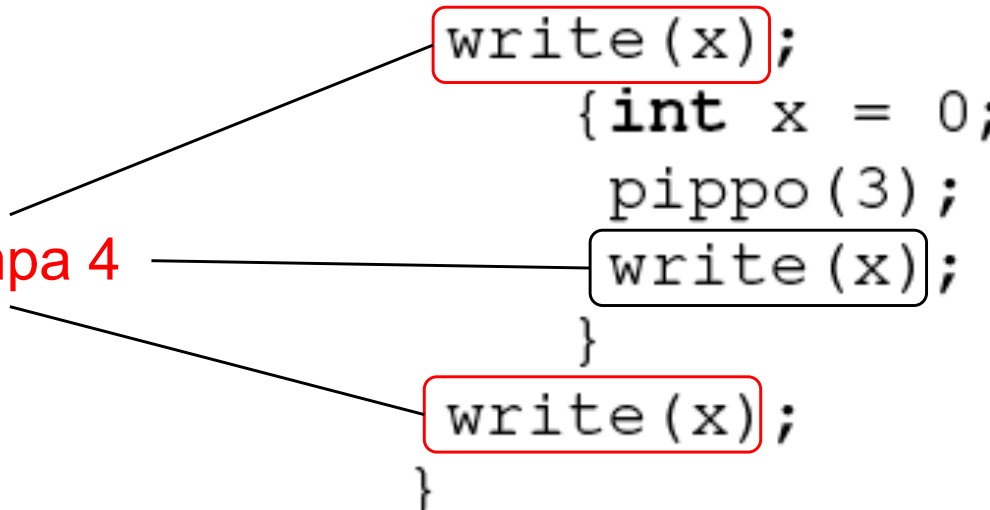


# Scope dinamico

- Un nome non locale è risolto nel blocco attivato *più di recente* e non ancora disattivato

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

stampa 4



# Scope statico: indipendenza dalla posizione

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

- il corpo di `foo` è parte dello scope della `x` più esterna
- la chiamata di `foo` è compresa nello scope della `x` più interna
- `foo` può essere chiamata in molti contesti diversi
- l'unico modo in cui `foo` può essere compilata in modo univoco è che il riferimento a `x` nelle due chiamate di `foo` sia sempre quello più esterno (scope statico)

# Scope statico: indipendenza dai nomi locali

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

la modifica del locale **x** in **y** dentro **fie**

- modifica la semantica del programma in scope **dinamico**
- non ha alcun effetto in scope **statico**

# Scope dinamico: specializzare una funzione

- `visualizza` è una procedura che rende a colore sul video una certa forma

```
...  
{var colore = rosso;  
  visualizza(testo);  
}
```

# Scope statico vs dinamico

- **Scope statico** (*scoping statico, statically scoped, lexical scope*).
  - informazione completa dal testo del programma
  - le associazioni sono note a tempo di compilazione
  - principi di indipendenza
  - più complesso da implementare ma più efficiente
  - Algol, Pascal, C, Java, ...
- **Scope dinamico** (*scoping dinamico, dynamically scoped*).
  - informazione derivata dall'esecuzione
  - spesso causa di programmi meno ``leggibili''
  - più semplice da implementare, ma meno efficiente
  - Lisp (alcune versioni), Perl
- Differiscono solo in presenza congiunta di
  - ambiente non locale e non globale
  - procedure

# Attenzione: C

- Algol, Pascal, Ada, Java permettono di annidare blocchi di sottoprogrammi  
non possibile in C:
  - funzioni definite solo nel blocco più esterno
  - dunque in una funzione l'ambiente è partizionato in locale e globale
  - non si presenta il problema dei non-locali

Questo non vuol dire che la regola di scoping (statico o dinamico) sia indifferente in C !

Significa solo che è semplice determinare dove risolvere un non-locale (cioè *nell'ambiente globale*)

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



# Determinare l'ambiente

- **L'ambiente** è dunque determinato da
  - regola di **scope** (statico o dinamico)
  - regole specifiche, p.e.
    - quando è visibile una dichiarazione nel blocco in cui compare?

discuteremo più avanti

- regole per il **passaggio dei parametri**
- regole di **binding** (shallow o deep)
  - intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X

# Alcune regole specifiche

- Dov'è visibile una dichiarazione all'interno del blocco in cui essa compare?
  - a partire dalla dichiarazione e fino alla fine del blocco
    - *Java: dichiarazione di una variabile*
  - sempre (dunque anche *prima*) della dichiarazione
    - *Java: dichiarazione di un metodo*
      - permette metodi mutuamente ricorsivi

```
{a=1;    // no!  
  int a;  
  ...  
}
```

```
{void f(){  
    g(); // si  
}  
void g(){  
    f(); // si  
}  
...  
}
```

# Mutua ricorsione

Mutua ricorsione (funzioni, tipi) in linguaggi dove un nome deve essere dichiarato prima di essere usato?

- rilasciare tale vincolo per funzioni e/o tipi
  - Java per i metodi
  - Pascal per tipi puntatore

## Pascal

```
type lista = ^elem;  
elem = record  
    • definizioni incomplete  
    info : Integer;  
    next : lista;  
end
```

## Java

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

## Ada

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

## C

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