

Lingaggi di Programmazione

Corso di Laurea in "ITPS"

I nomi e l'ambiente

Valeria Carofiglio

(Questo materiale è una rivisitazione del materiale
prodotto da Nicola Fanizzi)

Argomenti

- Introduzione
- Nomi
- Ambienti di riferimento
- Variabili
- Costanti con nome

Nomi-Astrazione

**Sequenza di caratteri usata per denotare
una componente del programma**

permette di astrarre

aspetti relativi ai dati - es: locazione di memoria \leftrightarrow nome
aspetti relativi al controllo - es: insieme di comandi \leftrightarrow nome

```
int pippo;
```

```
int pluto() {  
    Pippo = 1;  
}
```

Per Rappresentare oggetti e
riferirsi a questi (variabile,
funzione,...)

Ambiente
Sua strutturazione
Regole di SCOPE

Necessità di

REGOLE SEMANTICHE PRECISE
MECCANISMI IMPLEMENTATIVI ADEGUATI

Nomi

sequenza di caratteri usata per denotare oggetti:

Nei LdP i nomi possono essere

Identifieri: Token alfanumerici

Operatori simbolici (primitivi): +,-,...

Nomi

Nome ≠ oggetto da esso denotato

- **Aliasing**

- stesso oggetto con molti nomi (alias)
 - ... ma è anche possibile riutilizzare gli stessi nomi in parti diverse del programma per componenti diverse:
 - stesso nome per più oggetti

Uso dei Nomi e

astrazione elementare

- **Astrazione sui dati**
 - es. usare il nome di una variabile permette di astrarre dalla locazione di memoria, indirizzo, ecc.
 - $\text{area} = (\text{b} * \text{h}) / 2$
 - **ambiente:** implementazione del meccanismo che determina l'associazione nome-locazione
- **Astrazione sul controllo (es. procedure)**
 - un nome associato ad un insieme di comandi
 - regole per determinare la visibilità e l'utilizzo del sottoprogramma (interfaccia = nome + parametri)

Oggetti denotabili..

(lista non esauriva di oggetti cui puo' essere dato un nome)

- ..**definiti dall'utente**
 - variabili, parametri formali, sottoprogrammi, tipi, etichette, moduli, costanti, eccezioni, ...
- ..**definiti dal linguaggio**
 - tipi primitivi, operazioni primitive, costanti predefinite, parole riservate, ...

Nome ≠ oggetto da esso denotato



Associazioni nome-oggetto denotate(binding)

Il concetto di binding (associazione nome-oggetto denotato)

- Un **binding** è un'associazione, come quella tra un attributo ed un'entità o tra un'operazione ed un simbolo (operatore)
- **binding-time:**
 - momento in cui avviene il binding

NOMI e ASSOCIAZIONI :

Dalla creazione del linguaggio all'esecuzione di un programma

- Progettazione & implementazione
del linguaggio:
 - NOMI
 - Costanti primitive
 - Tipi primitivi
 - Operatori primitivi

Progettazione del linguaggio

- **Progettazione dei nomi da utilizzare:**
 - Lunghezza massima?
 - Sono permessi caratteri di **connessione** (es. "**_**")?
 - Differenza maiuscole/minuscole (**case-sensitive**)?
 - Le parole **speciali** sono considerate parole riservate o parole-chiave?

Progettazione del linguaggio

Lunghezza dei Nomi

- Se la loro lunghezza massima è troppo corta, i nomi difficilmente possono essere significativi (ossia possono suggerire un significato)
- Esempi in vari linguaggi:
 - FORTRAN I: max 6
 - COBOL: max 30
 - FORTRAN 90 e ANSI C: max 31
 - Ada e Java: nessun limite, e tutti sono significativi
 - C++: nessun limite, ma gli implementatori possono imporre un limite

Progettazione del linguaggio

I connettori nei nomi

- Esempi in vari linguaggi
 - Pascal, Modula-2, e FORTRAN 77 non li permettono
 - Altri lo fanno

Progettazione del linguaggio

Case sensitivity nei nomi

- *Svantaggio per la leggibilità:*
nomi che appaiono simili sono diversi
 - Peggiore ancora in C++ e Java poiché i nomi predefiniti mescolano maiuscole o minuscole (es. `IndexOutOfBoundsException`)
- In C, C++, e Java i nomi sono case-sensitive
 - Non succede in altri linguaggi

Progettazione del linguaggio

Parole speciali

- Aiuto alla leggibilità
- Usate per delimitare o separare l'espressione delle istruzioni
 - Una **parola chiave** è una parola che è speciale solo in certi contesti, es., in Fortran
 - **Real VarName**
Real è un tipo di dato seguito da un nome pertanto si tratta dell'uso come parola chiave
 - **Real = 3.4**
Real è una variabile in questo caso
- Una **parola riservata** è una parola speciale che non può essere usata per un nome definito dall'utente

NOMI e ASSOCIAZIONI :

Dalla creazione del linguaggio all'esecuzione di un programma

- Progettazione & implementazione del linguaggio:
 - Scrittura del programma
 - Definizione delle NOMI Def.Dall'Utente associazioni
 - Inizio
 - NOMI
 - Costanti primitive
 - Tipi primitivi
 - Operatori primitivi

NOMI e ASSOCIAZIONI :

Dalla creazione del linguaggio all'esecuzione di un programma

- Progettazione & implementazione del linguaggio:
 - Scrittura del programma
 - Inizio delle associazioni
 - Definizione NOMI Def.Dall'Utente
 - Traduzione di costrutti in linguaggio macchina e allocazione di memoria per strutture dati gestibili staticamente
 - Compilazione
 - Es: var globali: id \leftrightarrow locazione

NOMI e ASSOCIAZIONI :

Dalla creazione del linguaggio all'esecuzione di un programma

- Progettazione & implementazione del linguaggio:
 - Scrittura del programma
 - Inizio delle associazioni
 - Definizione NOMI Def.Dall'Utente
 - Compilazione
 - Traduzione di costrutti in linguaggio macchina e allocazione di memoria per strutture dati gestibili staticamente
- Esecuzione
 - Tutte le altre associazioni non ancora avvenute
 - Es:
 - var locali di procedure ricorsive
 - Variabili puntatore

NOMI e ASSOCIAZIONI :

Fasi trascurate

- **Linking-time:** Collegamento del programma
 - associare un nome ad una variabile esterna
- **Loading-time:** Caricamento del programma
 - associare una variabile ad una cella di memoria
 - . es in FORTRAN 77 o ad una variabile static (in C)

Associazione (Binding) statica e dinamica

- Un'associazione è statica se avviene prima dell'esecuzione (run time) e resta immutata per tutta l'esecuzione del programma

- Un'associazione è dinamica se avviene per la prima volta durante l'esecuzione e può essere cambiata nel corso dell'esecuzione del programma

COME?

I concetto di
AMBIENTE

Ambiente

(non esiste a livello di macchina fisica → caratteristica dei LdP ad alto livello:
occorre simularlo nella implementazione del linguaggio)

insieme delle associazioni nomi-oggetti denotabili esistenti a run-time

in uno specifico punto del programma ed
in uno specifico momento della sua esecuzione

usualmente si escludono quelle definite dal linguaggio

Componente della macchina astratta:

per ogni sezione del codice determina l'associazione
corretta per ogni nome - oggetto denotato

Introdurre l'associazione nell'ambiente: **dichiarazioni**

costrutto che permette di introdurre una nuova associazione nell'ambiente corrente

- **Esplicie**

Dichiarazione di variabile
Dichiarazione di funzione

```
int pippo;  
int pluto () {  
    return 0;  
}
```

type T = int;

Dichiarazione di tipo

- **implicite**

- inferenza del tipo dal primo uso dell'oggetto associato al nome (dal nome/dal contesto d'uso)

Ambiente: Molteplici associazioni

```
{ int pippo;  
  pippo = 2;  
  { char pippo;  
    pippo = a  
  }  
}
```

stesso nome oggetti diversi

(il nome pippo denota due variabili diverse)

- può dipendere anche dal momento in cui il flusso raggiunge una sezione di codice
 - caso dei parametri o nomi locali nei sottoprogrammi ricorsivi

Ambiente: Molteplici associazioni

```
int y=0;  
void foo(reference int x) {  
    x = x+1;  
}  
...  
foo(y);  
//qui y vale 1
```

Aliasing: passaggio parametri per riferimento

- × visibile nel programma chiamante (attraverso il suo nome)
- × visibile nel corpo della procedura attraverso il nome del parametro formale

Ambiente: Molteplici associazioni

```
int *X, *Y;  
X = (int *) malloc(sizeof (int));  
*X = 2;  
Y = X;  
*Y = 10;  
printf("%d", *X);
```

aliasing: diversi nomi per lo stesso oggetto

(uso di puntatori: riferimenti molteplici ad una stessa var. Dinamica
nello stesso ambiente)

Organizzare l'ambiente:

Blocchi

Regione di codice identificata da segnalatori di inizio e fine
che possa contenere dichiarazioni locali a tale regione

a partire da ALGOL 60

Algol e derivati: **begin** . . . **End**

C e derivati: { . . . }

Lisp: **let** . . . **in** . . .

- blocco **inline** (anonimo): può comparire dovunque possa comparire un comando semplice
- blocco associato ad un **sottoprogramma**: corpo del sottoprogramma, esteso con le dichiarazioni dei parametri formali

Ambiente di un Blocco

E' l'ambiente esistente nel momento in cui il blocco viene eseguito

(include le associazioni - binding - per i nomi dichiarati localmente al blocco)

L'ambiente cambia con l'entrata e l'uscita dai blocchi
Un blocco è il costrutto di granularità minima cui è associabile un ambiente costante

```
for (...) {  
    int index;  
    ...  
}
```

```
declare LCL :  
FLOAT;  
begin  
    ...  
end
```

ADA

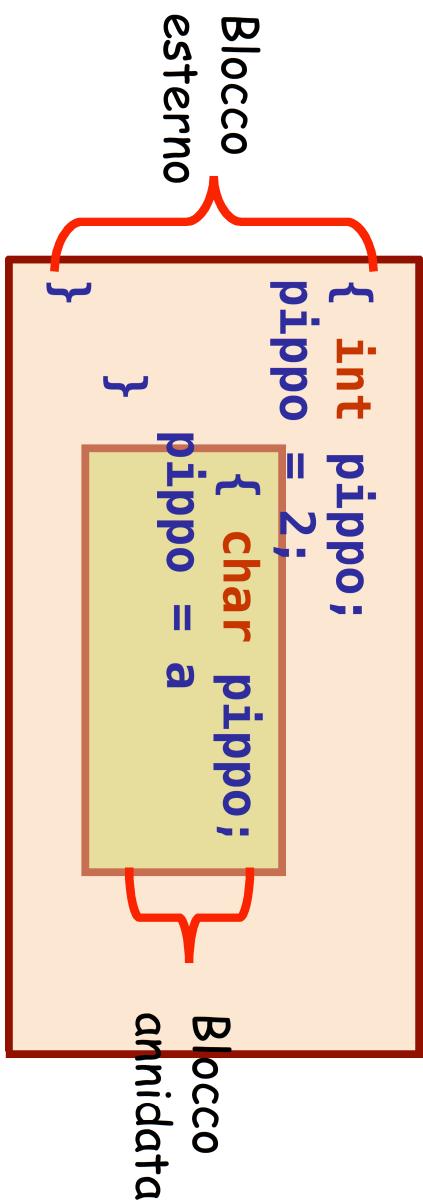
ambiti statici in unità di programma

26

Annidamento di Blocchi

strumento per organizzare l'ambiente

la definizione di un blocco inclusa in quella di un altro



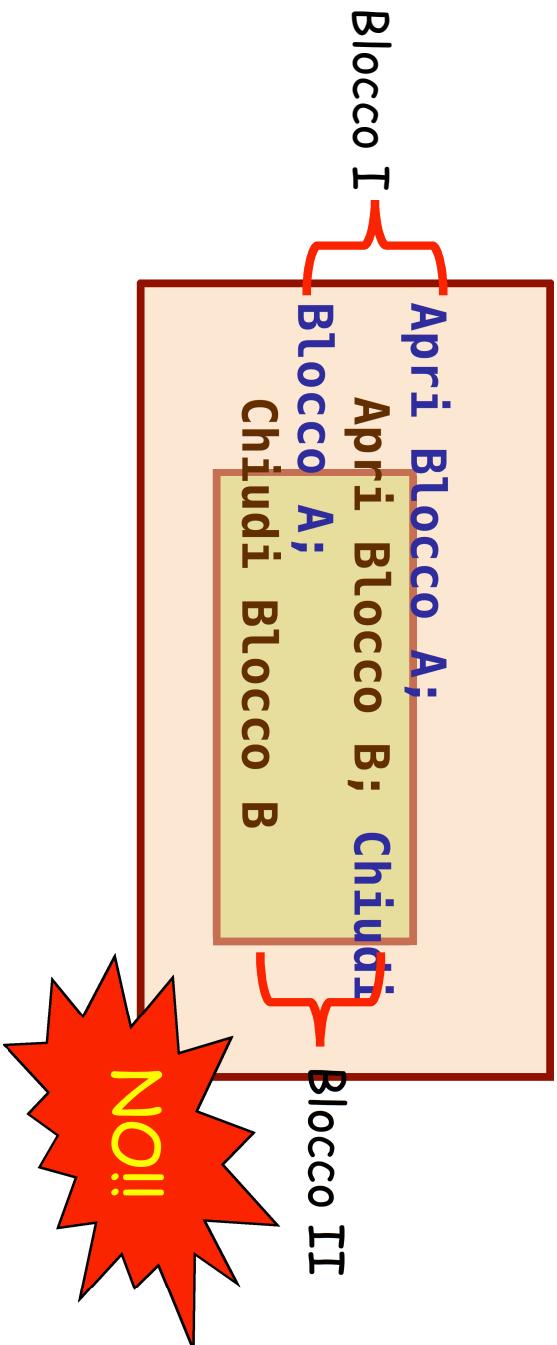
Differenze nel tipo di annidamento concesso per i LdP

- in C: i blocchi associati a sottoprogrammi non possono essere annidati (no procedure in def di procedure)
- in Pascal e Ada è possibile avere blocchi di sottoprogrammi all'interno di altri sottoprogrammi

Sovrapposizione di blocchi

Mai sovrapponibili!!!!

vale pressoché per tutti i linguaggi che prevedono i blocchi



l'ultimo blocco aperto deve essere il primo ad essere chiuso

Visibilità & Regola Canonica

come e quando una dichiarazione è visibile

Una dichiarazione d'locale ad un blocco A è **visibile**

in un altro blocco B

se il binding creato dalla dichiarazione d' in A
vale anche per l'ambiente del blocco B

Apri Blocco A;
dichiarazione d

Apri Blocco B;

Chiudi Blocco B;

Apri Blocco A;
dichiarazione d

Apri Blocco B;
dichiarazione d

Chiudi Blocco B;

Apri Blocco A;
dichiarazione d

Apri Blocco B;
dichiarazione d

Chiudi Blocco B;

d dichiarata in A
è visibile in B

d dichiarata in A
NON è visibile in B

Le due dichiarazioni sono
indipendenti

Regola

dichiarazione locale ad un blocco

visibile nel blocco ed in tutti i blocchi annidati

tranne ove compaia una dichiarazione per quel nome

Tipi d'ambiente

- L'ambiente associato ad un blocco composto da:
 - **ambiente locale:** dei binding per nomi dich. localmente; per i blocchi di sottoprogrammi contiene anche i binding per i parametri formali
 - Per individuare questo ambiente:
 - si considerano solo binding locali
 - **ambiente non-locale:** dei binding per nomi visibili nel blocco ma non dichiarati localmente
 - Per individuare questo ambiente:
 - si considerano i binding esterni
 - **ambiente globale:** dei binding creati all'inizio dell'esecuzione del programma principale; nomi usabili in tutti i blocchi
 - fa parte dell'ambiente non-locale

Tipi d'ambiente: esempio

```
A: { int a = 1;
```

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

```
int c = 2;
```

```
C: { int c = 3;
```

```
int d;
```

```
d = a+b+c;
```

```
printf("%d", d);
```

```
}
```

```
D: { int e;
```

```
e = a+b+c;
```

```
printf("%d", e);
```

```
}
```

```
}
```

Sia A il programma principale

Determinare l'Ambiente locale:

Guardare le dichiarazioni presenti nel blocco

l'Ambiente NON - locale:

Guardare le dichiarazioni esterne al blocco

Tipi d'ambiente: esempio

```
A: { int a = 1;
```

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

```
C: { int c = 3;
```

```
int d;
```

```
d = a+b+c;  
printf("%d",d);
```

```
}
```

```
D: { int e;  
e = a+b+c;  
printf("%d",e);
```

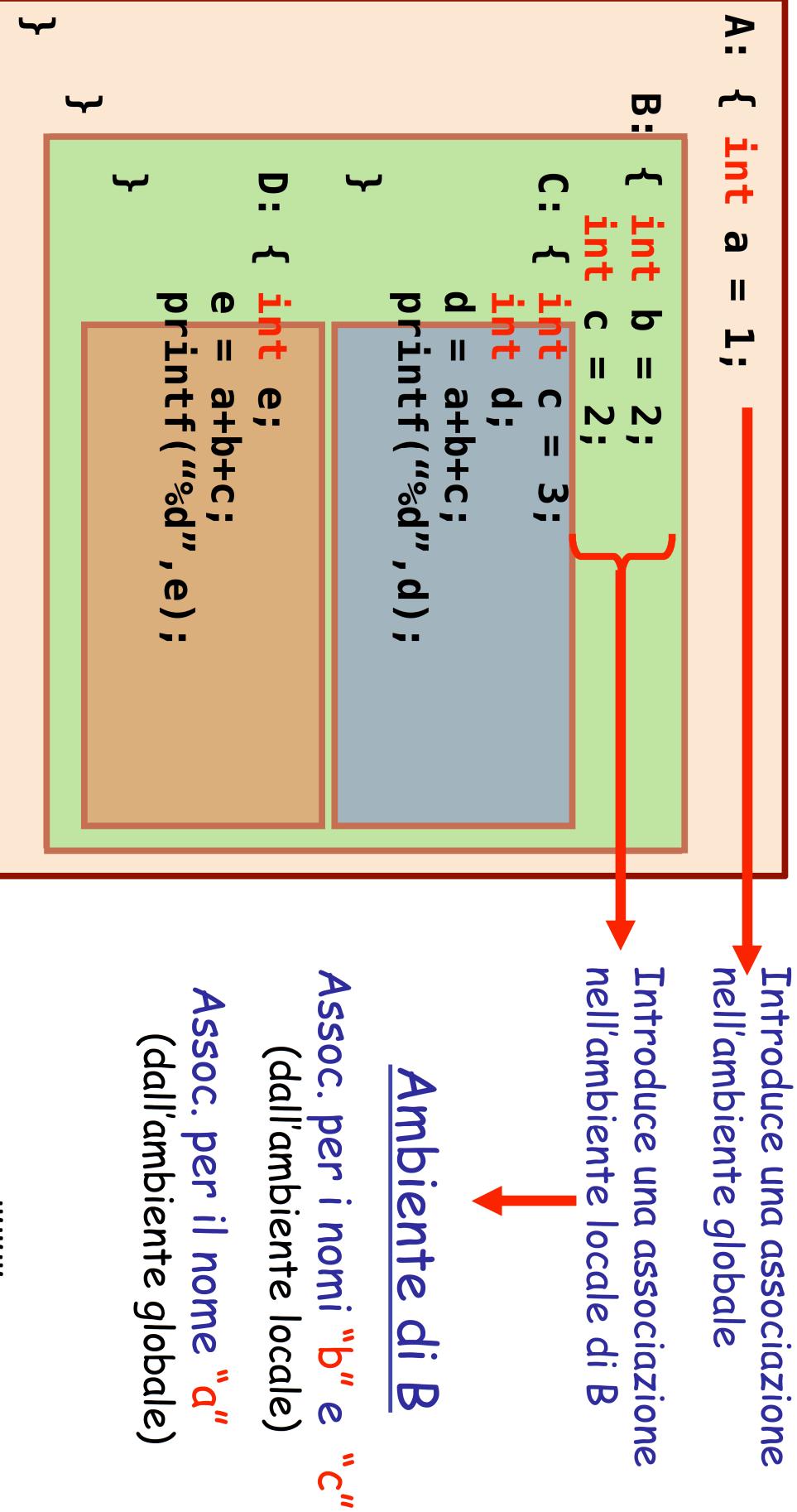
```
}
```

```
}
```

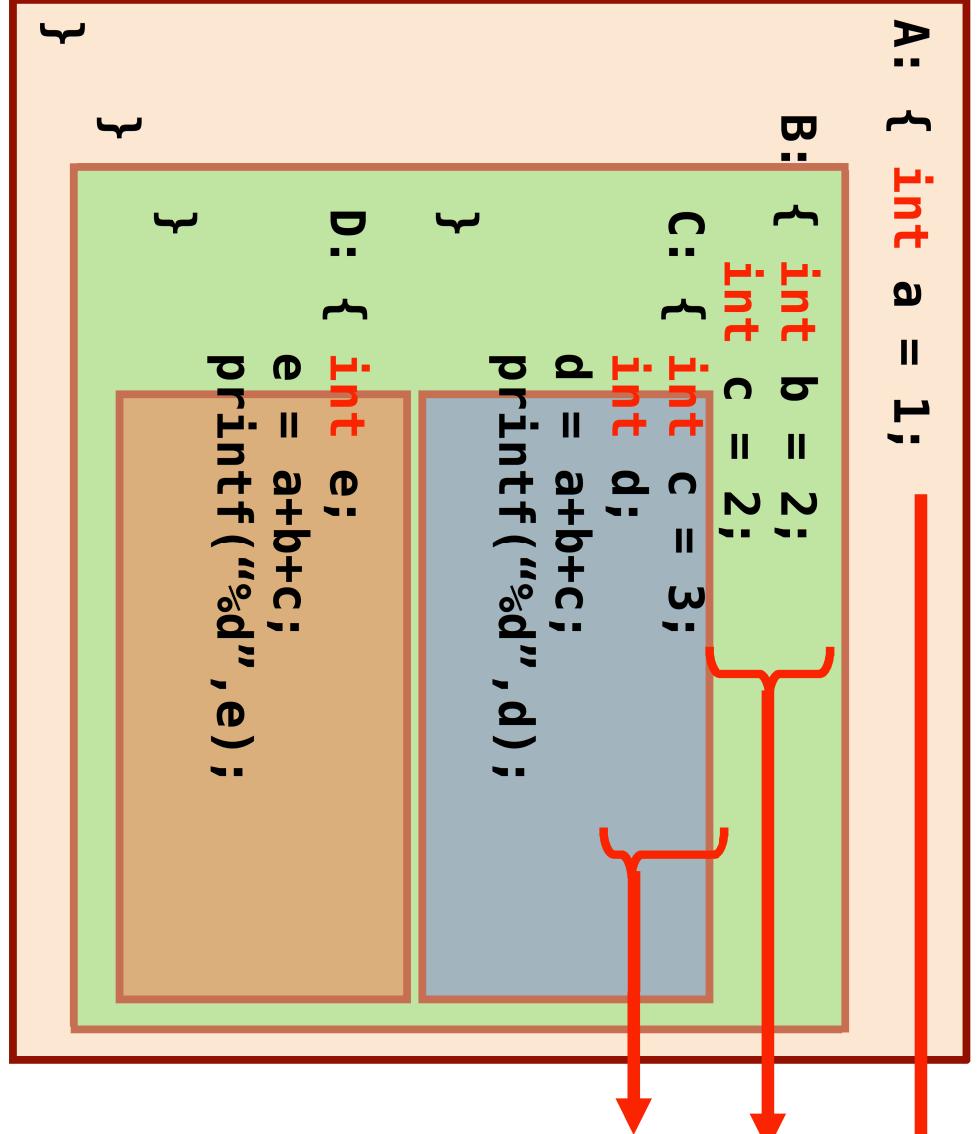
Introduce una associazione
nell'ambiente globale

Sia A il programma principale

Tipi d'ambiente: esempio



Tipi d'ambiente: esempio



Sia A il programma principale
(dall'ambiente non locale che include l'ambiente globale)

Tipi d'ambiente: esempio (cont.)

- A: blocco che introduce l'ambiente globale
 - contiene il binding per a
- B: ambiente locale
 - contiene due var. locali b e c
 - e il binding per a dall'ambiente globale
 - C: ambiente locale annidato in quello di B
 - contiene due var. locali c (che nasconde l'altra) e d
 - eredita il binding per b dall'ambiente non-locale ed a dall'ambiente globale di A
 - D: ambiente locale annidato in quello di B
 - contiene la var. locale e
 - eredita i binding per b,c dall'ambiente non-locale di B ed a dall'ambiente globale di A

Operazioni sull'ambiente

(durante l'esecuzione di un programma)

- **entrata in un blocco**

1) create associazioni tra nomi dichiarati localmente al blocco e oggetti denotati

2) disattivate le associazioni tra nomi e oggetti denotati che hanno **binding non locali preesistenti** (nomi già esistenti all'esterno del blocco e ridefiniti all'interno)

- **uscita da un blocco**

1) distruzione delle associazioni per nomi locali

2) riattivate le associazioni per nomi oscurati

Operazioni sull'ambiente: esempio

```
A: { int a = 1;
```

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

```
int c = 2;
```

```
C: { int c = 3;
```

```
int d;
```

```
d = a+b+c;
```

```
printf("%d", d);
```

```
}
```

```
D: { int e;
```

```
e = a+b+c;
```

```
printf("%d", e);
```

```
}
```

```
}
```

```
}
```

Sia A il programma principale

All'entrata del blocco C:

1) Create assoc. per i nomi "c" e "d"
(dall'ambiente locale)

2) Disattivate assoc. per il nome "c"
già esistente all'esterno
del blocco e ridefinito
all'interno (binding non-
locale preesistente)

Rimangono attive:

Assoc. per il nome "b"

Assoc. per il nome "a"

(dall'ambiente non locale che
include l'ambiente globale)

Operazioni sull'ambiente: esempio

(cont.)

```
A: { int a = 1;
```

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

```
int c = 2;
```

```
C: { int c = 3;
```

```
int d;
```

```
d = a+b+c;
```

```
printf("%d", d);
```

```
}
```

```
D: { int e;
```

```
e = a+b+c;
```

```
printf("%d", e);
```

```
}
```

```
}
```

```
}
```

Sia A il programma principale

All'uscita del blocco C:

1) Distrutte assoc. per i nomi "c" e "d"
(dall'ambiente locale)

2) Riattivata assoc. per il nome "c"

già esistente

all'esterno del blocco

(oscurate dalla ridefinizione locale di c)

Rimangono attive:

Assoc. per il nome "b"

Assoc. per il nome "a"

(dall'ambiente non locale che include l'ambiente globale)

Operazioni su nomi ed ambiente

(più in generale)

- **creazione associazione nome dichiarato-oggetto denotato (naming): elaborazione dichiarazione all'entrata di un blocco con dichiarazioni locali**
- **riferimento ad oggetto mediante il nome (referencing): uso del nome in un comando/una espressione**
- **disattivazione associazione nome dichiarato-oggetto denotato (entrrata blocco - nuova associazione per stesso nome)**
- **riattivazione associazione nome dichiarato-oggetto denotato (uscita blocco)**
- **distruzione associazione nome dichiarato-oggetto denotato (unnaming)**

Operazioni su oggetti denotabili

- **Creazione di un oggetto**
 - allocazione di memoria necessaria (possibile inizializzazione)
- **Accesso all'oggetto**
 - nome binding locazione valore (accesso a var)
 - la regola canonica rende l'associazione nome-oggetto univoca
- **Modifica dell'oggetto**
 - nome binding locazione
 - e modifica del valore nella locazione
- **Distruzione dell'oggetto**
 - deallocazione delle risorse

Operazioni su oggetti denotabili

Creazione dell'associazione nome-oggetto e Creazione dell'oggetto

sono spesso contemporanee

ESEMPIO

dichiarazioni di variabili

```
Int x;
```

Nome: **x**
oggetto: **variabile intera**
creazione oggetto: **allocazione di memoria**

Operazioni su oggetti denotabili

*Creazione dell'associazione nome-oggetto e
Creazione dell'oggetto*

sono spesso contemporanee

ESEMPIO

dichiarazioni di variabili



ciò non vale sempre!

`lifetime(oggetto) ≠ lifetime(associazione nome-oggetto)`

(es.: > passaggio per riferimento nelle procedure

<: riferimento "dangling")

Scope (visibilità)

**La visibilità di un'associazione è il segmento
di istruzioni per il quale essa vale**

- Gli ambienti possono cambiare in base alle operazioni definite su di essi
 - Op: All'entrata.... All'uscita dei blocchi
 - ambienti non locali più difficili da trattare

Visibilità e Regola canonica

Quale valore viene stampato?

```
A: {     int x = 0;  
        void procedura()  
        {  
            x = 1;  
        }  
  
B: {     int x;  
        procedura();  
    }  
  
printf("%d",x);  
}
```

Visibilità e Regola canonica

Quale valore viene stampato?

```
A: {     int x = 0;  
        void procedura()  
        {  
            x = 1;  
        }  
  
B: {     int x;  
        procedura();  
    }  
  
printf("%d", x);
```

A quale dichiarazione di
x fa riferimento la x
nella printf()?
45

Visibilità e Regola canonica

Quale valore viene stampato? (due viste)

```
A: {  
    int x = 0;  
  
    void procedura()  
    {  
        x = 1;  
    }  
  
    B: {  
        int x;  
        procedura();  
    }  
  
    printf("%d",x);  
}
```

Procedura() definita in A
x di Procedura è la stessa
dell'ambiente locale di A →
1 (valore stampato)

Visibilità e Regola canonica

Quale valore viene stampato? (due viste)

```
A: { int x = 0;
```

```
void procedura()
```

```
{  
    x = 1;  
}
```

```
B: { int x;  
procedura();  
}
```

```
printf("%d",x);  
}
```

```
A: { int x = 0;
```

```
void procedura()
```

```
{  
    x = 1;  
}
```

```
B: { int x;  
procedura();  
}
```

```
printf("%d",x);  
}
```

Procedura() definita in A
x di Procedura è la stessa
dell'ambiente locale di A →
1 (valore stampato)

Chiamata a procedura() in B →
x dell'ambiente locale di B →
regola: all'uscita di B x locale distrutta →
470 (valore stampato)

Visibilità e Regola canonica

Quale valore viene stampato? (due viste)

```
A: { int x = 0;
```

```
void pr{
```

```
}
```

```
B: { }
```

```
printf(
```

Dipende dalle regole
di scope adottate!!!
(scelta di progetto del LdP)

```
x = 1;
```

```
int x;  
procedura();
```

```
f ("%d",x);
```

```
}
```

Procedura() definita in A
x di Procedura è la stessa
dell'ambiente locale di A →
1 (valore stampato)

mata a procedura() in B →
nell'ambiente locale di B →
regola: all'uscita di B x locale distrutta →
480 (valore stampato)

Regole di visibilità

Determinano la validità delle associazioni nel passaggio da un ambiente all'altro

"una dichiarazione locale ad un blocco è visibile in esso ed in tutti i blocchi annidati"....

Si possono intendere

...sul testo del programma : staticamente

(la regola canonica già vista, ma più completa)

...Sul flusso di esecuzione: dinamicamente

Scope statico (o annidamento piu' vicino)

Basato sulla struttura sintattica del sorgente:
l'associazione è stabilita dal compilatore

(varie per i LdP)

A: { int a = 1;

B: { int b = 2;

int c = 2;

C: { int c = 3;

int d;

d = a+b+c;
printf("%d",d);

}

D: { int e;

e = a+b+c;

printf("%d",e);

}

}

PUNTO 1

Le dichiarazioni locali
(non quelle in blocchi annidati)
definiscono l'ambiente locale



Ambiente locale di B

Assoc. Per i nomi "b" e "c"
(dall'ambiente locale)

Scope statico (o annidamento piu' vicino)

**Basato sulla struttura sintattica del sorgente:
l'associazione è stabilita dal compilatore**

(varie per i LdP)

PUNTO 2

A: { int a = 1;

B: { int a = 1;

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

D: { int c = 3;
int d;
d = a+b+c;
printf("%d",d);
}

E: { int e;
e = a+b+c;
printf("%d",e);
}

L'uso di un nome in un blocco
comporta l'utilizzo delle
associazioni dell'ambiente
locale, se esiste;

ALTRIMENTI

risalire all'ambiente dei blocchi
contenitori (**1 livello alla volta**)
fino a trovare una associazione
valida

(ambiente più esterno all'ultimo: parole
predefinite del linguaggio)

Uso di a in C



Assoc. Per il nome "a" in A

Scope statico (o annidamento piu' vicino)

**Basato sulla struttura sintattica del sorgente:
l'associazione è stabilita dal compilatore**

(varie per i LdP)

PUNTO 3

```
A: { int a = 1;
```

```
    int procedura(int n){
```

```
        a = n+1;
```

```
    procedura(2);
```

il nome di un blocco
fa parte dell'ambiente
immediatamente esterno
al blocco



Ambiente del nome

Procedura() È una dichiarazione in A →

procedura():

introduce una associazione per il nome
nell'ambiente locale di A

Il nome procedura() fa
parte dell'ambiente
locale di A

E' un blocco annidato in A → l'associazione
è visibile nel corpo di procedura()
(procedure ricorsivemesse)

I suoi parametri No!!

Scope statico: esempio

Quali valori vengono stampati?

```
{  
    int x = 0;  
  
    void incrementa(int n)  
    {  
        x = n+1;  
    }  
  
    incrementa(3);  
    printf("%d",x);  
}  
  
{  
    int x = 0;  
    incrementa(3);  
    printf("%d",x);  
}  
printf("%d",x);  
}
```

Scope statico: esempio

Quali valori vengono stampati?

```
{  
    int x = 0;  
  
    void incrementa(int n)  
    {  
        x = n+1;  
    }  
  
    incrementa(3);  
    printf("%d",x);  
{  
    int x = 0;  
    incrementa(3);  
    printf("%d",x);  
}  
printf("%d",x);  
}
```

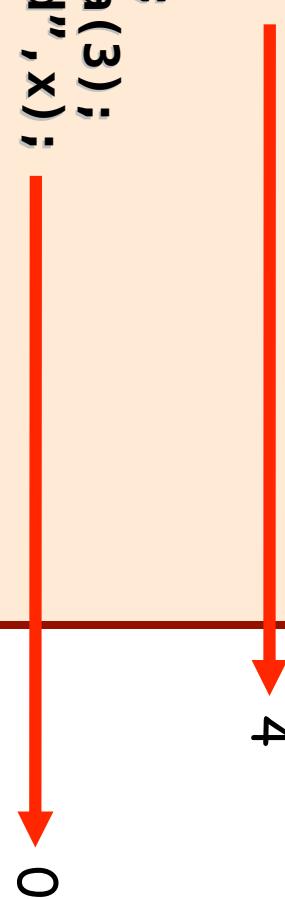


4

Scope statico: esempio

Quali valori vengono stampati?

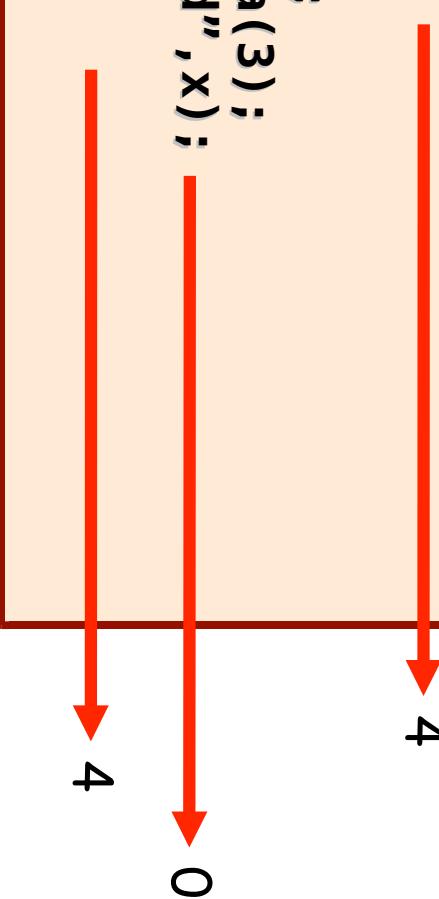
```
{  
    int x = 0;  
  
    void incrementa(int n)  
    {  
        x = n+1;  
    }  
  
    incrementa(3);  
    printf("%d",x);  
{  
    int x = 0;  
    incrementa(3);  
    printf("%d",x);  
}  
printf("%d",x);
```



Scope statico: esempio

Quali valori vengono stampati?

```
{  
    int x = 0;  
  
    void incrementa(int n)  
    {  
        x = n+1;  
    }  
  
    incrementa(3);  
    printf("%d",x);  
    {  
        int x = 0;  
        incrementa(3);  
        printf("%d",x);  
    }  
    printf("%d",x);  
}
```



Scope statico

- **vantaggi:**
 - leggibilità e comprensibilità
 - basta scorrere le dichiarazioni nel sorgente
 - ottimizzazione
 - il compilatore può decidere (quasi) tutto
 - usata dalla maggior parte dei linguaggi
 - Algol, Pascal, C, C++, Ada, Java, Scheme, ...
- **svantaggi**
 - relativamente complicata gestione a run-time
 - gerarchia degli ambienti locali e non
 - < >
 - gerarchia delle attivazione dei blocchi

Scope dinamico: esempio

Quale associazione per la `x` nel blocco di `stampa1()` ?

```
{  
    const x = 0;  
  
    void stampa1()  
    {  
        printf("%d",x);  
    }  
  
    void stampa2()  
    {  
        const x = 1;  
        stampa1();  
    }  
  
    stampa2();  
}
```

Scope dinamico: esempio

Quale associazione per la `x` nel blocco di `stampa1()` ?

```
{  
    const x = 0;  
  
    void stampa1()  
    {  
        printf("%d",x);  
    }  
  
    void stampa2()  
    {  
        const x = 1;  
        stampa1();  
    }  
  
    stampa2();  
}
```



Scope dinamico

Basato sulle sequenze di chiamata delle unità di programma,
e non su come appaiono nel sorgente
(criterio temporale anziché spaziale)

- usata in
alcuni dialetti di Lisp,
APL,
Snobol,
Perl

Scope dinamico regola di visibilità

- per l'ambiente locale la regola è identica al caso dello scope statico
- L'associazione per un nome non locale X
in un punto P del programma
è la più recente tra quelle attive, create per X
(attiva quando il flusso di controllo raggiunge P)

Scope dinamico: esempio

Quale associazione per la **x** nel blocco di **stampa1()** ?

```
{  
    const x = 0;  
  
    void stampa1()  
    {  
        printf("%"d",x);  
    }  
  
void stampa2()  
{  
    const x = 1;  
    { const x = 2; }  
    stampa1();  
}  
stampa2();
```

Statico e dinamico
differiscono solo
per la
determinazione di
ambiente non
locale e non
globale

LdP con scope dinamico → valore stampato = 1

Scope dinamico
utile per
modificare il
comportamento di
sottoprogrammi
senza dover usare
parametri espliciti.

Scope dinamico

- vantaggi:
 - flessibilità
 - stesso sottoprogramma si comporta diversamente a seconda del momento in cui viene invocato
- svantaggi
 - minore leggibilità e comprensibilità
 - difficile ricostruire le dichiarazioni dal sorgente
 - Scarsa efficienza della gestione a run-time
 - poco usata nei linguaggi *general-purpose*

Problemi di scope (statico)

Differenze tra i vari LdP

- DOVE possono essere introdotte dichiarazioni
- QUAL è l'esatta visibilità delle variabili locali
 - **regole restrittive in Pascal**
 - i. Dichiarazioni: ad inizio blocco e PRIMA dell'uso
 - ii. associazioni valevoli da inizio a fine blocco (possibili "buchi")
 - iii. Dichiarazioni di nomi non predefiniti: PRIMA dell'uso
 - **più rilassate in C e Ada**
 - Associazioni valevoli dalla dichiarazione a fine blocco
 - Eccezioni (alla iii.) tipi ricorsivi
 - sottoprogrammi forward Pascal e nomi funzioni C
- **Java: disciplina mista**
 - dichiarazioni "ovunque" e valide fino a fine blocco
 - dichiarazione membro vale per tutta la classe

Problemi di scope (statico)

Regole Restrittive del Pascal

```
Begin  
const pippo = valore;  
const valore = 0;  
End
```

Errato

```
Begin  
valore = 1;  
procedure pluto;  
begin  
const pippo = valore;  
const valore = 0;  
end  
end
```

Errore di semantica
statica segnalato dal
compilatore

pippo = 1

Contro le regole di
visibilità 65

Problemi di scope (statico)

Differenze tra i vari LdP

- DOVE possono essere introdotte dichiarazioni
- QUAL è l'esatta visibilità delle variabili locali
 - **regole restrittive in Pascal**
 - i. Dichiarazioni: ad inizio blocco e PRIMA dell'uso
 - ii. associazioni valevoli da inizio a fine blocco (possibili "buchi")
 - iii. Dichiarazioni di nomi non predefiniti: PRIMA dell'uso
 - **più rilassate in C e Ada**
 - Associazioni valevoli dalla dichiarazione a fine blocco
 - Eccezioni (alla iii.) tipi ricorsivi, nomi funzioni C
 - **Java: disciplina mista**
 - dichiarazioni "ovunque" e valide fino a fine blocco
 - dichiarazione membro vale per tutta la classe

Problemi di scope (statico)

C - ADA

```
Begin  
  const pippo = valore;  
  const valore = 0;  
End
```

Errato

```
Begin  
  valore = 1;  
  procedure pluto;  
    begin  
      const pippo = valore;  
      const valore = 0;  
    end  
end
```

Associazioni valevoli
dalla dichiarazione a
fine blocco