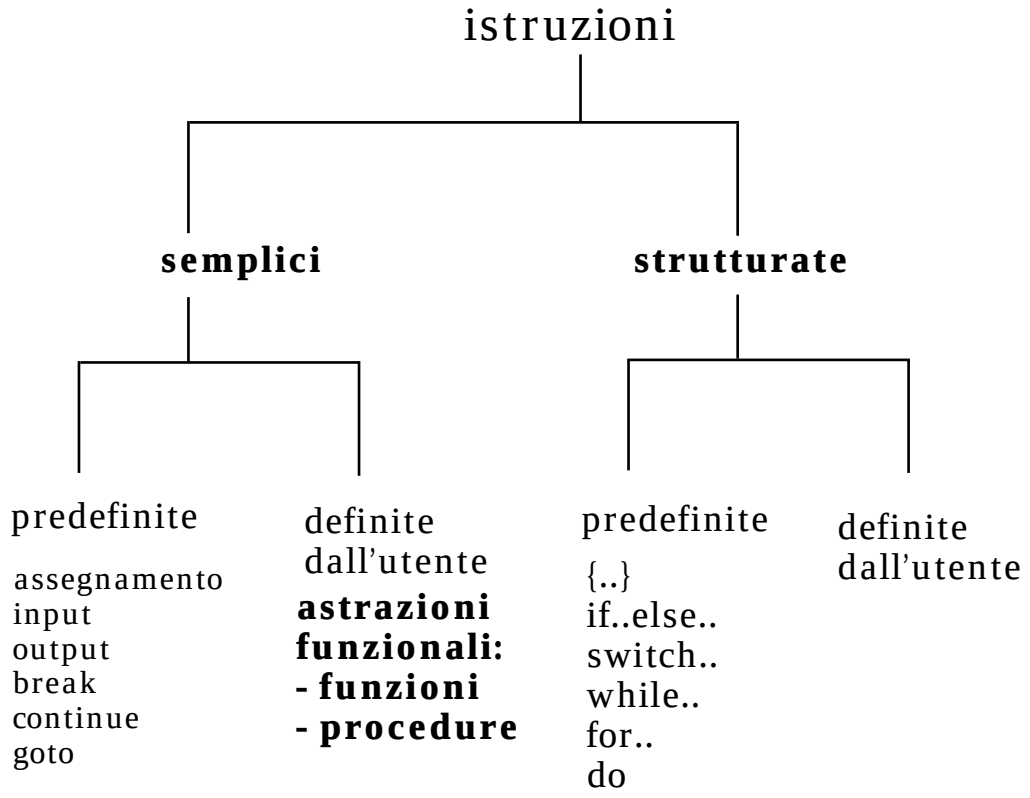
Astrazioni funzionali: Funzioni e Procedure



Spesso puo' essere utile avere la possibilita' di costruire delle nuove istruzioni che risolvano parti specifiche di un problema.

Ad esempio:

Ordinamento di un insieme:

```
#include <stdio.h>
#define dim 10

main()
{int V[dim], i,j, max, tmp, quanti;

/* lettura dei dati */
for (i=0; i<dim; i++)
  { printf("valore n. %d: ",i);
    scanf("%d", &V[i]);
   }
/*ordinamento */

for(i=0; i<dim; i++)
  {quanti=dim-i;
  max=quanti-1;
  for( j=0; j<quanti; j++)
   { if (V[j]>V[max])
        max=j;
   }
  if (max<quanti-1)
   { /*scambio */
   tmp=V[quanti-1];
   V[quanti-1]=V[max];
   V[max]=tmp;
   }
  }
/*stampa */
for(i=0; i<dim; i++)
  printf("Valore di V[%d]=%d\n", i, V[i]);
}
```

Potrebbe essere conveniente scrivere lo stesso algoritmo in modo piu' **astratto**:

```
#include <stdio.h>
#define dim 10

main()
{
int V[dim];

/* lettura dei dati */
leggi(V, dim);

/*ordinamento */
ordina(V, dim);

/*stampa */
stampa(V,dim);
}
```

☞ **`leggi(),ordina(),stampa()`** sono

astrazioni funzionali.

Astrazioni funzionali (*funzioni* e *procedure*):

- Consentono di definire nuove istruzioni che agiscono sui dati utilizzati dal programma, nascondendo la sequenza delle operazioni effettivamente eseguite dalla macchina.

- Vengono realizzate mediante la definizione di unita` di programma (***sottoprogrammi***) distinte dal programma principale (*main*) dando un nome ad un gruppo di istruzioni e stabilendo le modalita` di comunicazione tra l'unita` creata e quella in cui viene utilizzata.

Tutti i linguaggi di alto livello offrono la possibilita' di utilizzare funzioni e/o procedure:

- costrutti per la **definizione** di sottoprogrammi

- meccanismi per l'**utilizzo** di sottoprogrammi (meccanismi di ***chiamata***)

Funzioni e Procedure

Definizione:

Nella fase di **definizione** di un sottoprogramma (funzione o procedura) si stabilisce:

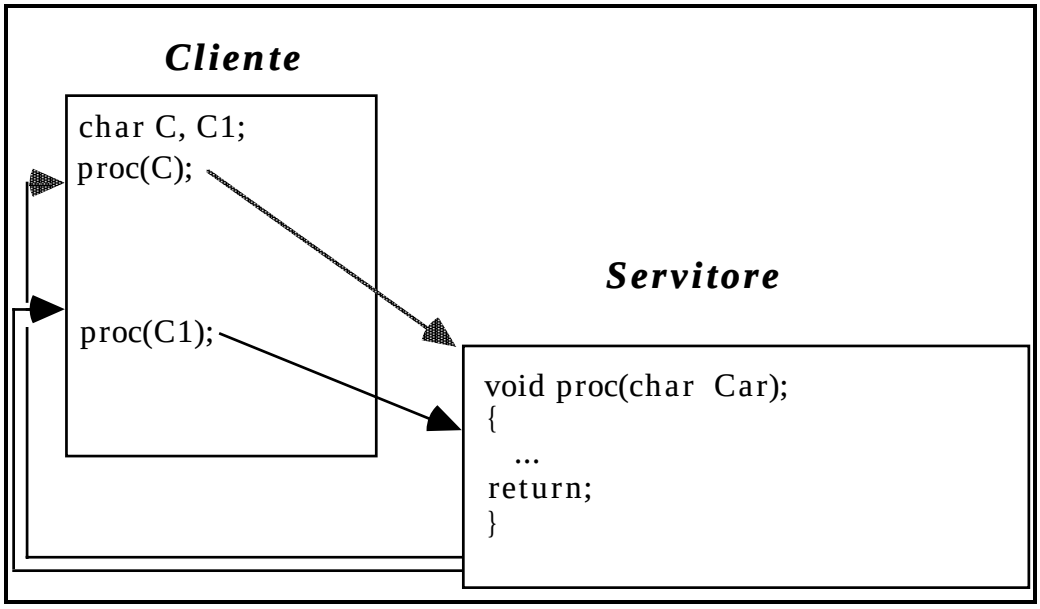
- un **identificatore** del sottoprogramma
- si esplicita il **codice** eseguito dal sottoprogramma (nel linguaggio utilizzato)
- si stabiliscono le **modalita' di comunicazione** tra l'unita` di programma che usa il sottoprogramma ed il sottoprogramma stesso (definizione dei **parametri formali**).

Utilizzo di funzioni/procedure:

- Per eseguire il gruppo di istruzioni del sottoprogramma, si utilizza l'identificatore assegnato al sottoprogramma in fase di definizione (***chiamata*** o invocazione del sottoprogramma).

- Alla chiamata viene sospesa l'esecuzione del programma (o dell'unita`) che contiene l'invocazione ed il controllo passa al sottoprogramma chiamato.

Modello ***Cliente-Servitore***



Parametri:

I ***parametri*** costituiscono il mezzo di comunicazione tra unita' chiamante ed unita' chiamata.

parametri formali: sono quelli specificati nella definizione del sottoprogramma (nell'esempio, **Car**). Sono in numero prefissato e ad ognuno di essi viene associato un tipo.

parametri attuali: sono i valori (o le variabili) effettivamente specificati all'atto della chiamata (nell'esempio, C e C1).

Parametri:

- Parametri ***attuali*** (specificati nella chiamata) e ***formali*** (specificati nella definizione) devono corrispondersi in ***numero***, ***posizione*** e ***tipo***.

- All'atto della chiamata avviene il ***legame dei parametri***, cioe' ai parametri formali vengono associati i parametri attuali. Esistono, in generale, varie forme di legame.

Esempio:

```
#include <stdio.h>

int f(int A, float B, char C)
{
/* definizione della funzione f*/
...
}


main()
{
  int p1;
  float p2;
  char p3;

  ...
  f(p1, p2, p3); /* corretto */
  f(p2, p1, p1); /* scorretto!*/
  f(p1, p2);       /* scorretto!*/
}
```

Funzioni e Procedure

Vantaggi:

- **riutilizzo di codice**: sintetizzando in un sottoprogramma un sotto-algoritmo, si ha la possibilita` di invocarlo piu` volte, sia nell'ambito dello stesso programma, che nell'ambito di programmi diversi (evitando di dover replicare ogni volta lo stesso codice).
- migliore **leggibilita'**: si ha in fatti una maggiore capacita` di astrazione
- sviluppo **top-down**: si delega a funzioni/procedure da sviluppare in una fase successiva la soluzione di sottoproblemi.

- testo del programma piu` **breve**: minore probabilita` di errori, dimensione del codice eseguibile piu` piccola.

☞ Utilizzando le astrazioni funzionali, la struttura dei programmi risulta articolata in **piu' unita'** (programma principale + sottoprogrammi).

Astrazioni funzionali

Si suddividono in procedure e funzioni:

Procedura:

E` un'astrazione della nozione di ***istruzione***. E` un'istruzione non primitiva attivabile in un qualunque punto del programma in cui puo` comparire un'istruzione.

Funzione:

E' l'astrazione del concetto di ***operatore*** (funzione o predicato) su un tipo di dato (primitivo o definito da utente). Si puo` attivare durante la valutazione di una qualunque espressione e **restituisce un valore.**

Ad esempio:

```
{
  leggi(N);/* procedura*/
  Fatt = fattoriale(N); /* funzione*/
};
```

☞ Formalmente, in C le procedure sono soltanto ***funzioni***; le procedure possono essere realizzate come funzioni che non restituiscono alcun valore (**void**).

Funzioni in C

La **definizione** di procedure e funzioni segue regole sintattiche simili.

Definizione di funzione:

<def-funzione> ::= <intestazione>
{ **<parte-dichiarazioni> <parte-istruzioni>** }

Quindi, per definire una funzione, e' necessario specificare una ***intestazione*** e un ***blocco*** {..}:

<intestazione>::=<tipo-ris> <nome> ([<lista-par-formali>])

L' intestazione contiene:

- **tipo del risultato** (***codominio***). Il tipo restituito puo` essere predefinito o definito dall'utente. Una funzione non puo` restituire valori di tipo:
  - **vettore**
  - **funzione**
- **identificatore** (*nome*) della funzione

- **lista dei parametri formali** (***dominio***). Per ciascun parametro formale viene specificato il tipo ed un identificatore che e` un nome simbolico per rappresentare il parametro all'interno della funzione (nel *blocco*). I parametri sono separati mediante virgola).

Definizione di Funzioni

Esempio:

```
int max (int a, int b) /*intestaz. */
{
  if (a>b) return a;
  else return b;
}
```

Blocco:

- Nella parte **blocco** possono essere presenti definizioni e/o dichiarazioni locali (***parte dichiarazioni***) e segue un **corpo** in cui viene specificato l'algoritmo rappresentato dalla funzione (***parte istruzioni***).
- I dati riferiti nel blocco possono essere **costanti**, **variabili**, oppure **parametri formali**.
- All'interno del blocco, i parametri formali vengono trattati come variabili.

Esempio:

```
#define N 100

typedef   char vettore[N];

int minimo (vettore vet)
{
int i, v, min;/* def. locali a minimo */
  for (min=vet[0], i=1; i<N; i++)
    {  v=vet[i];
       if (v<min) min=v;
    }
  return min;
}
```

☞ i, v, min sono ***variabili locali*** (dette variabili ***automatiche***):

- **tempo di vita**: esistono solo durante l'esecuzione della funzione minimo
- **visibilita'**: sono visibili (cioe` utilizzabili) soltanto all'interno della funzione minimo.

Istruzione return:

return [<espressione>]

restituisce il controllo al chiamante e assegna all'identificatore della funzione il valore dell'<espressione> (se specificato).

Esempio:

```
int  read_int () /* intest. */
{
 int a
 scanf("%d", &a);
 return a;
}
```

Possono essere ***piu' istruzioni return***:

```
int max (int a, int b) /*intest.*/
{
 if (a>b)    return a;
 else  return b;
}
```

o ***nessuna***:

```
int  print_int (int a) /* intestazione */
{
 printf("%d", a);
}
```

☞ In questo caso, il sottoprogramma termina in corrispondenza del simbolo **}** e il valore restituito e' ***indefinito*** .

Esempio:

```
/* funzione elevamento a potenza */

long power (int base, int n)
{
 int i;
 long p=1;

 for (i=1;i<=n;++i)
       p *= base; /* p = p*base */
 return p; /* ritorna il risultato */
}
```

Chiamata di funzioni:

All'interno di una espressione:

... NomeFunzione(Lista Parametri Attuali) ...

Esempio:

```
main()
{
int x=5;
printf("%d %d %d\n",
       x,power(x,2),power(x,3));
...
x=max(power(x,2), 30);
printf("%d\n", x);
}
```

Esempio:

codice delle funzioni leggi, ordina e stampa di pagina 3:

```
void leggi(int V[], int n);
{
 for (i=0; i<n; i++)
  { printf("valore n. %d: ",i);
    scanf("%d", &V[i]);
   }
}

void ordina(int V[], int n);
{
 for(i=0; i<n; i++)
  {quanti=n-i;
  max=quanti-1;
  for( j=0; j<quanti; j++)
     { if (V[j]>V[max])
          max=j;
     }
  if (max<quanti-1)
     { /*scambio */
     tmp=V[quanti-1];
     V[quanti-1]=V[max];
     V[max]=tmp;
     }
  }
}
```

```
void stampa(int V[], int n);
{
  for(i=0; i<n; i++)
    printf("Valore di V[%d]=%d\n", i,
  V[i]);
}
```

Realizzazione delle Procedure

Una funzione puo' anche avere nessun valore (void) come risultato.

void	insieme vuoto di valori (dominio vuoto)
void fun(...)	funzione che non restituisce alcun valore › **procedura**

Esempio:

```
void print_int (int a) /* intestazione */
{
  printf("%d", a);
}
```

☞ In questo modo si realizza il concetto di **procedura**: non e' necessario prevedere l'istruzione di return all'interno del blocco; se si utilizza, **non si deve specificare alcun argomento**:

return;

☞ Una funzione puo‘ anche non avere parametri.
Alcune versioni di C esigono che si specifichi nell’intestazione *"void"*, altre accettano una sintassi del tipo:

```
void dummy()
{
printf("Ciao!\n");
 }
```

Esempio:

```
#include <stdio.h>

int max (int a, int b) /*def. max*/
{
  if (a>b) return a;
  else return b;
}

void print_int (int a) /* def. */
{
  printf("%d\", a);
  return;
}

void dummy() /*def. dummy */
{
printf("Ciao!\n");
}

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  print_int(max(A,B));
  dummy();
}
```

Dichiarazione di funzione

Regola Generale:

Prima di utilizzare un identificatore e' necessario che sia gia' stato **almeno** dichiarato.

Funzioni C:

- **definizione**: descrive le proprieta` della funzione (tipo, nome, lista parametri formali) e la sua realizzazione (lista delle istruzioni coontenute nel blocco).
- **dichiarazione (*prototipo*)**: descrive le proprieta` della funzione senza definirne la realizzazione (***blocco***).

Dichiarazione di una funzione:

La **dichiarazione** di una funzione (***prototipo***) consiste nella sua intestazione, seguita da "**;**".

Ad esempio:

```
int max(int a, int b);
```

☞ Deve essere specificata ***prima*** della chiamata della funzione, **se la definizione segue (testualmente) la chiamata.**

Esempio:

```
#include <stdio.h>

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}

int max (int a, int b) /*intestaz. */
{
  if (a>b) return a;
  else return b;
}
```

☞ In questo caso il compilatore segnala un **errore** in corrispondenza della chiamata **max(A,B)**, perche' viene usato un identificatore che viene definito successivamente (dopo il main())

Soluzione:

```
#include <stdio.h>

int max(int a, int b);

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}

int max (int a, int b) /*intestaz. */
{
  if (a>b) return a;
  else return b;
}
```

☞ La dichiarazione anticipa le proprieta‘ di un oggetto (la funzione) che verra‘ definito successivamente.

E le dichiarazioni di printf/scanf etc. ?

☞ sono contentue nel file stdio.h:

#include <stdio.h>

provoca l’inserimento del contenuto del file specificato.

Esempio:

Calcolo della radice intera di un numero intero letto a terminale.

```
#include <stdio.h>
/* prototipi delle funzioni: */

int  RadiceInt (int par);
int  Quadrato (int par);

main(void)
{ int X;
scanf("%d", &X);
printf("Radice: %d\n", RadiceInt(X));
printf("Quadrato: %d\n", Quadrato(X));
}

/* definizione funzioni: */

int  RadiceInt (int par) /* intest.*/
{
 int cont = 0;
 while (cont*cont <= par)
  cont = cont + 1;
 return (cont-1);
}

int  Quadrato (int par)/* intest. */
{
 return (par*par);
}
```

Struttura dei Programmi C

Protocollo da utilizzare:

<lista dichiarazioni di funzioni>
<main>
<definizioni delle funzioni dichiarate>

Esempio:

```
#include <stdio.h>

int max(int a, int b);

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}

int max (int a, int b) /*intestaz. */
{
  if (a>b) return a;
  else return b;
}
```

Tecniche di legame dei parametri

Al momento della chiamata di un sottoprogramma (funzione o procedura) viene effettuata l'associazione fra parametri attuali e parametri formali: **legame** (o **passaggio**) dei parametri.

In generale, esistono vari meccanismi di passaggio dei parametri.

Meccanismi piu' comuni:

- Legame per **valore** (C, Pascal);
- Legame per **indirizzo**, o per riferimento (Pascal, Fortran;).

Tecniche di passaggio dei parametri

Hp: consideriamo un sottoprogramma **P**
con un parametro formale **pf**

Chiamata:

P(pa)

› **pa** e' il **parametro attuale**

Legame per valore

All'atto della chiamata di P:

1• viene allocata una cella di memoria associata a pf

2• viene valutato pa (puo` essere un'espressione), ed il suo valore viene **copiato** in pf

pa	**----> valore iniziale di pa**
pf	**----> valore iniziale di pa**

☞ Il parametro formale **pf** viene trattato come una **variabile locale** al sottoprogramma P: puo` essere modificato mediante assegnamento, etc.. In generale, al termine della chiamata, pf potra` assumere un valore diverso da quello iniziale.

Alla fine dell'esecuzione di P:

pa	**----> valore iniziale di pa**
pf	**----> nuovo valore**

☞ **Se pa e` una variabile**, al termine della chiamata, il suo valore rimane inalterato

☞ **Se pa e` una espressione**, il valore delle variabili che compaiono nell'espressione (nell'unita` di programma chiamante) rimane **inalterato**.

Legame per valore

Quindi:

Parametri passati per valore servono a fornire **valori in ingresso** al sottoprogramma.

☞ E' la tecnica di legame adottata normalmente dal C:

```
#include <stdio.h>


void P(int pf);

main()
{
int pa=10;

P(pa);
printf("valore finale di pa: %d\n",
    pa); /* pa vale 10 */
}

void P(int pf)
{
pf=100;
printf("valore finale di pf: %d\n",
         pf);
return;
}
```

Esempio:

Funzione che scambia due variabili X, Y (di tipo integer) se X>Y e restituisce il valore minore tra i due.

```
#include <stdio.h>
int scambia1 (int A, int B);
main()
{ int X,Y ;
  scanf("%d %d", &X, &Y);
  printf("Scambia: %d %d %d\n",
           scambia1(X,Y), X,Y);
}

int scambia1 (int A, int B)
{ int   T;
  if (A>B)
    {  T=A;
       A=B;
       B=T;
       return A; }
  else  return B;
}
```

Dati in ingresso: 33 5 qual e‘ il risultato stampato?

Alla chiamata di scambia1:

Alla fine dell'attivazione di scambia1:

Legame per indirizzo

Hp: consideriamo un sottoprogramma **P**
con un parametro formale **pf**

Chiamata:

P(pa)

› pa e' il parametro attuale

1• viene allocata una cella di memoria associata a pf
2• viene calcolato l'indirizzo di pa (**pa deve essere una variabile**),
3• l'indirizzo di pa viene assegnato a pf

pa ---->
valore di pa
pf -----> (indirizzamento indiretto)

☞ Il parametro formale pf si comporta come una **variabile locale** alla procedura P. Viene creata al momento dell'attivazione di P ed inizializzata all'**indirizzo di pa**. Durante l'esecuzione di P, il valore di pa puo` essere modificato, utilizzando come riferimento il valore di pf.

Alla fine dell'esecuzione di P:

pa ---->
valore di pa
pf -----> (indirizzamento indiretto)

☞ Il valore di pa nell'unita' di programma chiamante viene alterato.

Legame per indirizzo

Quindi:

Parametri passati per indirizzo servono per dare **valori in uscita** dall'unita` chiamata all'unita` chiamante.

NB: In **C** la tecnica di legame per indirizzo ***non e' presente*** (a parte per i vettori)

Esempio:

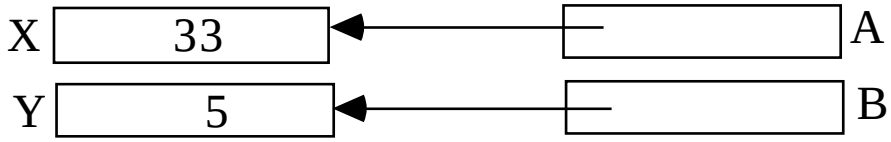
Il programma che segue e` solo a scopo esemplificativo (non c'e` il passaggio per indirizzo).

Funzione che scambia due variabili X, Y (di tipo integer) se X>Y e restituisce il valore minore tra i due.

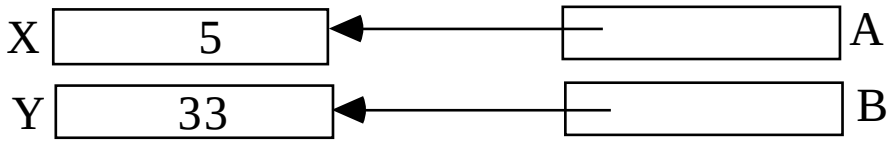
```
#include <stdio.h>
int  scambia1 (int A, int B);
main()
{ int   X,Y ;
  scanf("%d %d", &X, &Y);
  printf("\n%s %", "Scambia: ",
scambia1(&X,&Y);
  printf("\n%d \t %d %", X,Y);
}
```

```
int scambia1 (int A, int B)
/* se fosse per indirizzo */
{
  int T;
  if (A>B)
    {  T=A;
       A=B;
       B=T;
       return A; }
  else   return B;
}
```

Alla chiamata:



Alla fine dell'attivazione della procedura scambia2:



Passaggio dei parametri per indirizzo in C

In C questa tecnica di legame si puo‘ emulare utilizzando ***puntatori come parametri***.

```
#include <stdio.h>
int  scambia2 (int *A, int *B);

main()
{ int X,Y ;
  scanf("%d %d", &X, &Y);
  printf("\n Scambia: %d",
scambia2(&X,&Y));
  printf("\n%d \t %d %", X,Y);
}

int  scambia2 (int *A, int *B)
  {int   *T;
   if (*A>*B)
      {  *T=*A;
         *A=*B;
         *B=*T;
         return *A;
      }
   else return *B;
  }
```

Stampa: 5 /* scambia2 */
5 33 /* X, Y */

Esempio:

```
#include <stdio.h>
void  Fun ( ?     int  X);

main()
{int  N;
      N=3;
      printf("%d", N);          {1}
      Fun (N);
      printf("%d", N);          {3}
}

void  Fun ( ?     int  X)
      {
            X=X+1;
            printf("%d", X);                {2}
      }
```

Se il legame e' ***per valore*** abbiamo le stampe:

{1} 3
{2} 4
{3} 3

Se il legame e' ***per indirizzo*** :

{1} 3
{2} 4
{3} 4