Linguaggio C:
Breve riepilogo sulle funzioni

Esempio completo

```
#include <stdio.h>

int max (int x, int y )
{       if (x>y) return x;
                else return y;
}

main()
{
        int z = 8;
        int m;
        m = max(z , 4);
        printf("Risultato: %d\n", m);
}
```

Definizione

Chiamata

Funzioni: esecuzione

- **L'esecuzione comincia sempre dal main**
- **Quando avviene una chiamata a funzione:**
  - Si crea una nuova attivazione (istanza) del servitore
  - Si alloca memoria per i parametri formali e le variabili locali
  - Si trasferiscono i parametri al servitore (binding dei parametri formali al valore di quelli attuali)
  - Si trasferisce il controllo al servitore e si esegue il codice della funzione
- **Al termine dell'esecuzione della funzione il controllo ritorna al cliente** e "l'espressione funzione" denota il valore restituito con "return"

Funzioni: visibilità e ambiente

- Si chiama ambiente l'insieme degli identificatori (es. nomi di variabile) definiti in un determinato punto del programma
- Funzione e programma chiamante hanno **due ambienti distinti**
- Conseguenza importante: possono contenere identificatori con lo stesso nome, ma di fatto **diversi** (maggiori dettagli nella lezione di oggi)

Funzioni: dichiarazione e definizione

- Per poter utilizzare una funzione, il compilatore deve sapere della sua esistenza e conoscerne l'interfaccia
  - È necessario **dichiarare** una funzione prima del suo utilizzo

- Per poter eseguire un programma che utilizza una funzione, bisogna eseguirne il codice
  - È necessario **definire** una funzione, prima o dopo il suo utilizzo

Linguaggio C:
Regole di visibilità e procedure

Regole di visibilita` degli identificatori in C

Qual'e` il campo di azione di un identificatore?

Ossia: a quali unità di programma risulta visibile? Nel linguaggio C e` determinato ***staticamente***, in base al punto del programma in cui compare:

1. Al di fuori di qualunque funzione (incluso il "main")
2. Nella sezione delle dichiarazioni una funzione (incluso il "main")
3. Tra i parametri formali di una funzione
4. Nella sezioni di dichiarazioni di un ***blocco di istruzioni***

Sono identificatori i nomi di variabili e funzioni

Esempio (variabili)

```
#include <stdio.h>

int a;

int f(int b){
    int c;
}

main() {
    int d;
    {
        int e;
    }

    while (...) {
        int f;
    }
}
```

Al di fuori di qualunque funzione

Tra i parametri formali di una funzione

Nella sezione "dichiarazioni" di una funzione

Nella sezione "dichiarazioni" di un blocco di istruzioni

Visibilita` degli Identificatori

In generale, dato un programma scritto in linguaggio C, e` possibile distinguere:

- **Ambiente globale:**
  e` costituito dalle dichiarazioni e definizioni che compaiono nella parte di dichiarazioni globali di P (al di fuori di qualunque funzione).

- **Ambiente locale:**
  - **a una funzione**: e` l'insieme delle dichiarazioni e definizioni che compaiono nella parte dichiarazioni della funzione, piu` i suoi parametri formali.
  - **a un blocco**: e` l'insieme delle dichiarazioni e definizioni che compaiono all'interno del blocco.

Regole di visibilita` degli identificatori in C

Qual'e` il campo di azione di un identificatore?

precisamente: nel linguaggio C e` determinato ***staticamente***, in base all'ambiente al quale l'identificatore appartiene, secondo le seguenti regole:

1. il campo di azione di un identificatore **globale** va dal punto in cui si trova la sua dichiarazione (o definizione) fino alla fine del file sorgente (a meno della regola 4);
2. il campo di azione della dichiarazione (o definizione) di un identificatore **locale** e` il blocco (o la funzione) in cui essa compare e tutti i blocchi in esso contenuti (a meno della regola 3);
3. quando un identificatore K dichiarato in un blocco P, e` ridichiarato (o ridefinito) in un blocco Q racchiuso da P, allora il blocco Q, e tutti i blocchi innestati in Q, sono esclusi dal campo di azione della dichiarazione dell'identificatore K in P (***overriding***).
4. quando un identificatore K globale e` ridichiarato (o ridefinito) in un blocco P, allora il blocco P, e tutti i blocchi innestati in P, sono esclusi dal campo di azione della dichiarazione dell'identificatore globale K (***overriding***).

Visibilita` degli Identificatori

```
#include <stdio.h>
int A;
int f(float x);
main()
{  int B;
   ...
   {    char A;
        ...
   }
}
int f(float x)
{  int D;...
}
```

ambiente globale:
`int A, f`
identificatori globali

main: sono visibili

int A, f; (globali)

int B; (locale)

blocco: sono visibili

f (id. globale)

int B;

char A (locale al blocco).

int A non e` visibile!

funzione f: sono visibili

int A, f; (globali)

int D, float x (locali)

Esempio

```
#include <stdio.h>
main(){
     int i=0;
     while (i<=3){ /* BLOCCO 1 */
          int j=4; /* def. locale al blocco 1*/
          j=j+i;
          i++;

          { /* BLOCCO 2: interno al blocco 1*/
               float i=j; /*locale al blocco 2*/
               printf("%f\t%d\t",i,j);
          }
          printf("%d\t\n",i);
     }
}
```

Esempio

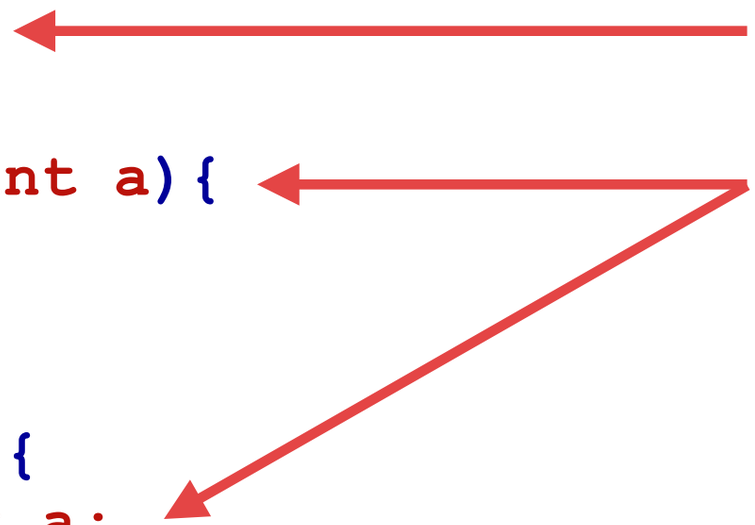
```
#include <stdio.h>

int a;

int f(int a){
     ...
}

main() {
     int a;
     {
          int a;
     }

     while (...) {
          int a;
     }
}
```



Identificatore globale

L'identificatore locale "oscura" quello globale

L'identificatore locale "oscura" quello globale e quello locale a "main"

Tempo di Vita delle variabili

E` l'intervallo di tempo che intercorre tra l'istante della creazione (allocazione) della variabile e l'istante della sua distruzione (deallocazione).

- E` l'intervallo di tempo in cui la variabile ***esiste*** ed in cui, compatibilmente con le regole di visibilita`, puo` essere utilizzata.

Nel linguaggio C si distingue tra:

- **Variabili Automatiche**:
  - **Variabili globali** sono allocate all'inizio del programma e vengono distrutte quando il programma termina: il tempo di vita e` pari al **tempo di esecuzione del programma**.
  - **Variabili locali** (e parametri formali) alle funzioni sono allocati ogni volta che si invoca la funzione e distrutti al termine dell'attivazione: il tempo di vita e` pari alla **durata dell'attivazione** della funzione in cui compare la definizione della variabile.

- **Variabili Dinamiche:**

  hanno un tempo di vita pari alla durata dell'intervallo di tempo che intercorre tra la **`malloc`** che le alloca e la **`free`** che le dealloca.

Variabili static

- E` possibile imporre che una variabile locale a una funzione abbia un tempo di vita pari al tempo di esecuzione  dell'intero programma, utilizzando il qualificatore **`static`**:

```
void f(){
    static int cont=0;
    ...
}
```

- **la variabile `static int cont`:**
  - **e` creata all'inizio del programma, inizializzata a 0, e deallocata alla fine dell'esecuzione;**
  - **la sua visibilita` e` limitata al corpo della funzione f,**
  - **il suo tempo di vita e` pari al tempo di esecuzione dell'intero programma**
  - **e` allocata nell'area dati globale (*data segment*)**

Esempio

```
#include <stdio.h>
int f()
{ static int cont=0;
        cont++;
        return cont;
}
main()
{       printf("%d\n", f());
        printf("%d\n", f());
}
```

- la variabile `static int cont`, e` allocata all'inizio del programma e deallocata alla fine dell'esecuzione; essa persiste tra una attivazione di f e la successiva: la prima printf stampa 1, la seconda printf stampa 2.

Procedure in C

Formalmente in C esiste solo il concetto di funzione:

e le *procedure* ?

E` possibile costruire delle particolari funzioni che non restituiscono alcun valore:

```
void proc (int P){..}
```

e` la definizione di una funzione (**`proc`**) che non restituisce alcun valore:

- **`void`** e` un identificatore di tipo per classificare dati il cui dominio e` l'insieme vuoto.

Procedure in C

La definizione di funzione:

```
void proc (int P){..}
```

realizza una procedura.

Osservazioni:

- la chiamata di **`proc`** non denota alcun risultato
- dall'interno del corpo della funzione **`proc`** non verra` restituito alcun risultato al cliente:
  - il corpo potra` contenere o meno l'istruzione **`return`**, eventualmente utilizzata senza argomento: **`return;`**
- Domanda legittima: ma allora a cosa servono le procedure? Un esempio nella prossima slide...

Procedure in C

Esempio:

```
#include <stdio.h>
void stampafloat(float P) /* "procedura" */
{ printf("%f\n", P);
  return; /* termina l'attivazione*/
}

float quadrato(float X) /* funzione*/
{return X*X;}

main()
{ float V;
  scanf("%f", &V);
  V=quadrato(V);
  stampafloat(V); /* chiamata di "procedura"*/
}
```

Per esempio a stampare un valore! Ma tipicamente una procedura ha lo scopo di modificare i parametri che le vengono passati

Tecniche di legame dei parametri

La tecnica di legame (o *passaggio*) dei parametri stabilisce come avviene l'associazione tra parametri attuali e formali.

In generale, un parametro può essere trasferito (*passato*) dal cliente al servitore:

- **per valore (per copia, *by value*)**
  - **si copia *il valore* del parametro attuale nel corrispondente parametro formale.**

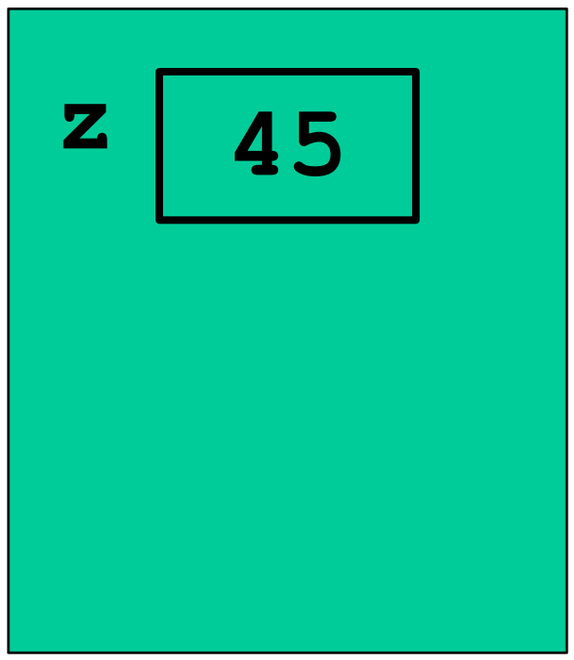
- **per riferimento (per indirizzo, *by reference*)**
  - **si associa al parametro formale *un riferimento* al corrispondente parametro attuale**

Legame per valore

HP: **z parametro attuale**

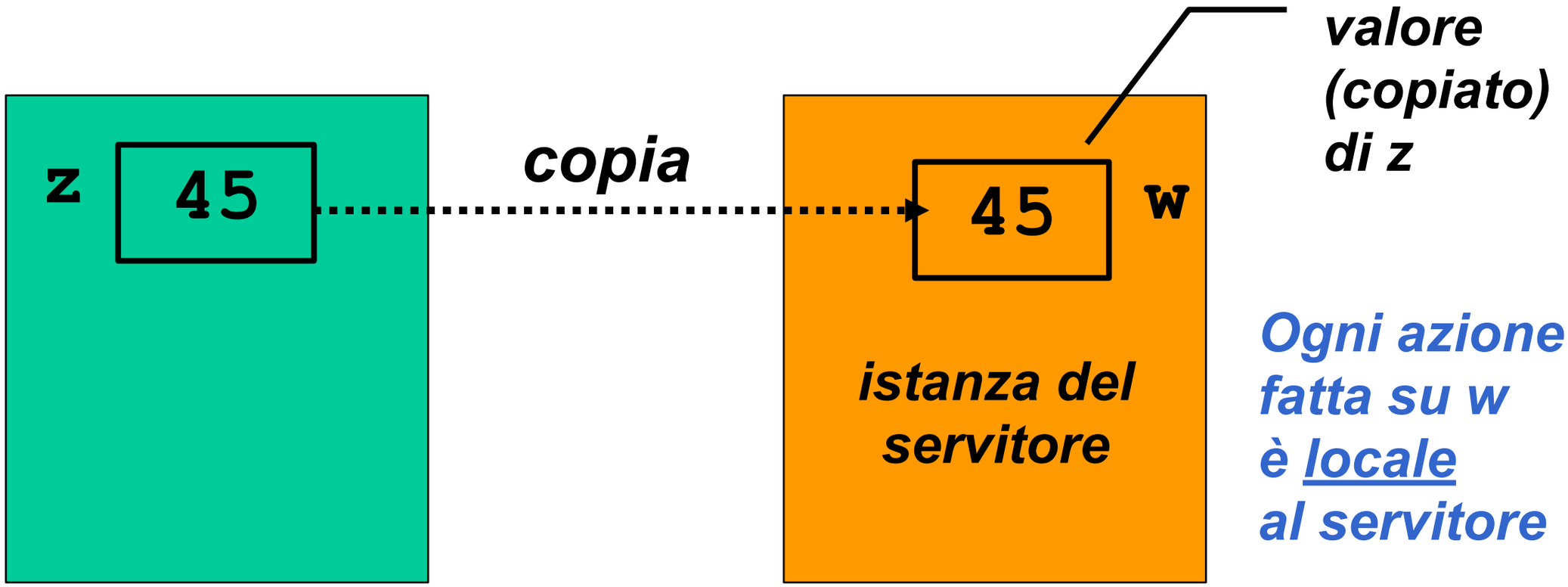
w parametro formale

si trasferisce *una copia del valore* del parametro attuale...



Legame per valore

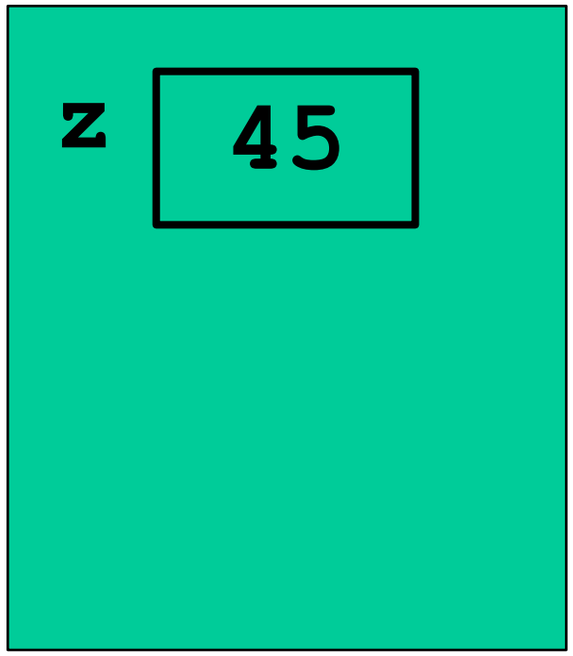
si trasferisce *<u>una copia del valore</u>* del parametro attuale nel parametro formale



Ogni azione fatta su w è <u>locale</u> al servitore

Legame per riferimento

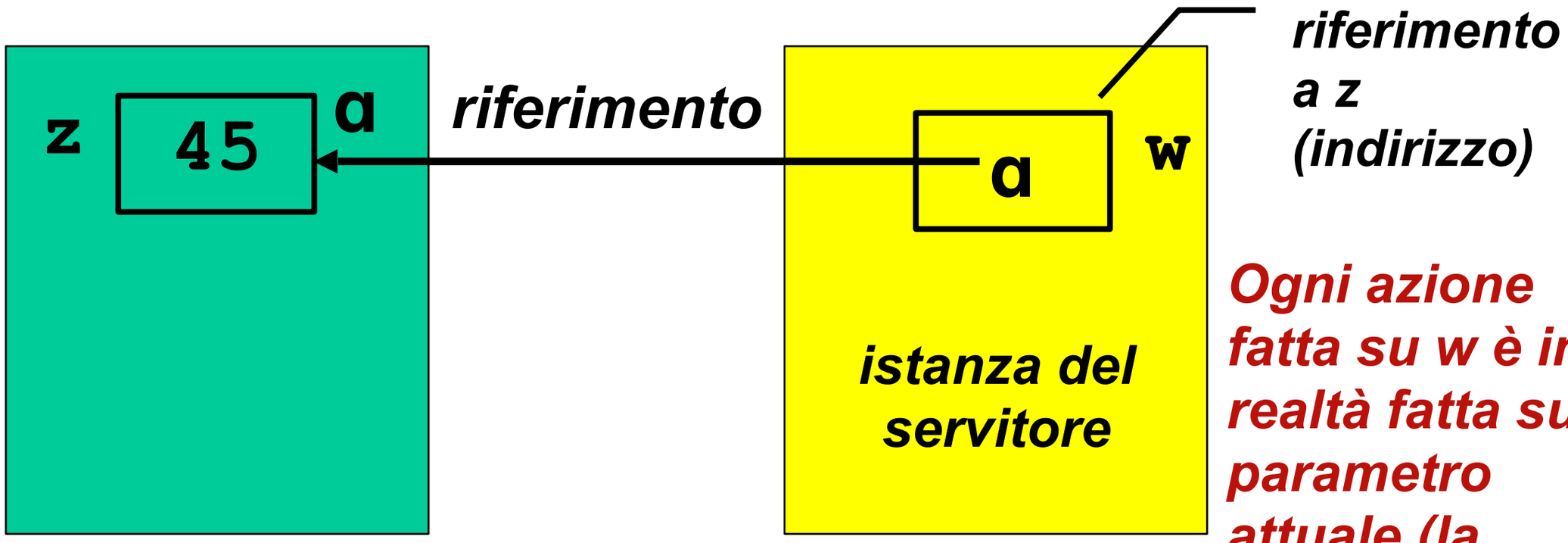
si trasferisce *un riferimento* al parametro attuale...



cliente

Legame per riferimento

- **si trasferisce *<u>un riferimento</u>* al parametro attuale che viene associato al parametro formale**



Ogni azione fatta su w è in realtà fatta sul parametro attuale (la variabile z del cliente)!

PASSAGGIO DEI PARAMETRI IN C

In C, i parametri sono trasferiti sempre e solo per valore (*by value*)

- **si trasferisce *una copia* del parametro attuale, *non l'originale*!**
- **tale copia è *strettamente privata e locale a quel servitore***
- **il servitore potrebbe quindi alterare il valore ricevuto, *senza che ciò abbia alcun impatto sul cliente***

Conseguenza: è impossibile usare un parametro per *trasferire informazioni dal servitore verso il cliente*

- **per trasferire un'informazione al cliente si sfrutta il *valore di ritorno* della funzione**

Esempio: valore assoluto

- **Definizione formale:** $|x|: \mathbb{Z} \rightarrow \mathbb{N}$

$$\begin{cases} |x| \text{ vale } x & \text{se } x \geq 0 \\ |x| \text{ vale } -x & \text{se } x < 0 \end{cases}$$

- **Codifica sotto forma di funzione C:**

```
int valAss(int x) {
  if (x < 0) return -x;
  else return x;
}
```

ESEMPIO: VALORE ASSOLUTO

- **Servitore**

```
int valAss(int x) {
      if (x<0) return -x;
      else return x;
  }
```

- **Cliente**

```
main(){
        int absz, z = -87;
        absz = valAss(z);
        printf("%d", z);
        }
```

ESEMPIO: VALORE ASSOLUTO

- **Servitore**

```
int valAss(int x) {
      if (x<0) return -x;
      else return x;
}
```

- **Cliente**

```
main(){
      int absz, z = -87;
      absz = valAss(z);
      printf("%d", z);
}
```

Quando valAss(z) viene chiamata, il valore attuale di z, valutato nell'environment corrente (-87), viene copiato e passato a valAss.

ESEMPIO: VALORE ASSOLUTO

- **Servitore**

```
int valAss(int x) {
      if (x<0) return -x;
      else return x;
}
```

- **Cliente**

```
main(){
      int absz, z = -87;
      absz = valAss(z);
      printf("%d", z);
}
```

valAss riceve quindi una copia del valore -87 e la lega al simbolo x. Poi si valuta l'istruzione condizionale, e si restituisce il valore 87.

ESEMPIO: VALORE ASSOLUTO

- **Servitore**

```
int valAss(int x) {
      if (x<0) return -x;
      else return x;
}
```

- **Cliente**

```
main(){
      int absz, z = -87;
      absz = valAss(z);
      printf("%d", z);
}
```

Il valore restituito viene assegnato a absz

ESEMPIO: VALORE ASSOLUTO

- **Servitore: modifica**

```
int valAss(int x) {
      if (x<0) x = -x;
      return x;
}
```

- **Cliente**

```
main(){
       int absz, z = -87;
       absz = valAss(z);
       printf("%d", z);
}
```

Se x e' negativo viene MODIFICATO il suo valore nella controparte positiva. Poi la funzione torna x

ESEMPIO: VALORE ASSOLUTO

x | -87

- **Servitore: modifica**

```
int valAss(int x) {
      if (x<0) x = -x;
      return x;
}
```

- **Cliente**

```
main(){
      int absz, z = -87;
      absz = valAss(z);
      printf("%d", z);
}
```

Quando valAss(z) viene chiamata, il valore attuale di z, valutato nell'environment corrente (-87), viene copiato e passato a valAss. Quindi x vale -87

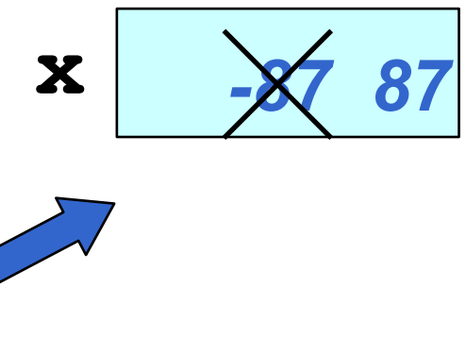
ESEMPIO: VALORE ASSOLUTO

- **Servitore: modifica**

```
int valAss(int x) {
      if (x<0) x = -x;
      return x;
}
```

- **Cliente**

```
main(){
      int absz, z = -87;
      absz = valAss(z);
      printf("%d", z);
}
```



valAss restituisce il valore 87 che viene assegnato a absz

NOTA: IL VALORE DI z NON VIENE MODIFICATO

ESEMPIO: VALORE ASSOLUTO

- **Servitore: modifica**

```
int valAss(int x) {
      if (x<0) x = -x;
      return x;
}
```

- **Cliente**

```
main(){
      int absz, z = -87;
      absz = valAss(z);
      printf("%d", z);
}
```

La printf stampa -87

NOTA: IL VALORE DI z NON VIENE MODIFICATO

Esempio completo

```
#include <stdio.h>
int valAss(int x)
{       if (x<0) x = -x;
        return x;
}
main()
{       int absz, z = -87;
        absz = valAss(z);
        printf("%d", z);
}
```

PASSAGGIO DEI PARAMETRI

Molti linguaggi mettono a disposizione il ***passaggio per riferimento*** (*by reference*)

- non si trasferisce una copia del valore del parametro attuale
- *si trasferisce un riferimento al parametro*, in modo da dare al servitore *accesso diretto* al parametro in possesso del cliente
  - il servitore *accede e modifica direttamente* il dato del cliente.

Il passaggio per riferimento è la chiave per produrre procedure con qualche utilità pratica

Passaggio dei parametri: osservazioni riassuntive

Legame per valore:

- Parametri ***passati per valore*** servono soltanto a comunicare ***valori in ingresso*** al sotto-programma.
- Se il passaggio avviene per valore, ogni parametro attuale non e` necessariamente una variabile, ma puo` essere, in generale, una **espressione**.

Legame per riferimento:

- Parametri ***passati per riferimento*** servono a comunicare ***valori sia in ingresso che in uscita*** dal sottoprogramma.
- Se il passaggio avviene per riferimento, ogni parametro attuale deve necessariamente essere una **variabile**.

Passaggio per riferimento in C

Il C *non* supporta *direttamente* il passaggio per riferimento:

- In alcuni casi il passaggio per riferimento e' ***indispensabile:*** ad esempio, nel caso di funzioni che producono piu` di un risultato.
- quindi, dobbiamo *costruircelo.*

È possibile costruirlo? Come?

1. **Utilizzando variabili globali (soluzione "sporca")**

Comunicazione cliente/servitore mediante l'ambiente condiviso

Una procedura/funzione può anche comunicare con il suo cliente **mediante aree dati globali***:* un esempio sono le ***variabili globali*** *del C.*

- Le ***variabili globali*** in C:
  - sono definite fuori da ogni funzione
  - sono allocate nell'area dati globale (accessibile al main e a tutte le funzioni)

Esempio

```
#include <stdio.h>

int x, y;

void scambia() {
     int t = x;
     x = y;
     y = t;
}

main(){
     x = 5;
     y = 7;
     printf("x = %d, y = %d", x, y);
     scambia();
     printf("x = %d, y = %d", x, y);
}
```

Comunicazione cliente/servitore mediante l'ambiente condiviso

- La procedura "scambia" ha effettivamente modificato la variabili x ed y.
- Ma ***non c'è stato alcuno scambio di parametri***!
- E se volessi scambiare delle altre variabili? ***La soluzione non è portabile***!
- Per queste ed altre ragioni, la comunicazione cliente servitore tramite ambiente condiviso va in generale evitata.

Passaggio per riferimento in C

Il C *non* supporta *direttamente* il passaggio per riferimento:

- In alcuni casi il passaggio per riferimento e' ***indispensabile:*** ad esempio, nel caso di funzioni che producono piu` di un risultato.
- quindi, dobbiamo *costruircelo.*

È possibile costruirlo? Come?

1. **Utilizzando variabili globali (soluzione "sporca")**
2. **Utilizzando i puntatori**

REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

- In C e` possibile provocare gli stessi effetti del passaggio per riferimento utilizzando parametri di tipo ***puntatore***.

- in questo caso, a differenza dei linguaggi che prevedono il legame per riferimento:

il programmatore deve gestire esplicitamente degli indirizzi (trasferiti ***per valore*** alla funzione) che verranno ***esplicitamente dereferenziati*** (nel corpo della funzione).

REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

In C per realizzare il passaggio per riferimento:

- il cliente deve *passare* ***esplicitamente*** *gli indirizzi*
- il servitore deve *prevedere* ***esplicitamente*** *dei puntatori come parametri formali*

Esempio

```
#include <stdio.h>

void quadratoEcubo(float X, float *Q, float *C {
    *Q=X*X;  /* dereferencing di Q*/
    *C=X*X*X; /* dereferencing di C*/
    return;
}
```

```
main() {
    float dato, cubo, quadrato;
    scanf("%f", &dato);
    quadratoEcubo(dato, &quadrato, &cubo);
    printf("\nDato %f, il suo quadrato e`: %f, il suo
  cubo e` %f\n", dato, quadrato, cubo);
}
```

Esempio: scambio di valori tra variabili

```
void scambia(int* a, int* b) {
 int t;
 t = *a;  *a = *b;  *b = t;
}
```

```
main(){
 int y = 5, x = 33;
 scambia(&x, &y);
}
```

Esempio: soluzione di una equazione di secondo grado: $Ax^2 + Bx + C = 0$

```
#include <stdio.h>
#include <math.h>
int  radici(float A, float B, float C,  float *X1, float *X2)
{ float D;
  D= B*B-4*A*C;
  if (D<0) return 0;
  else
        {  D=sqrt(D);
           *X1 = (-B+D)/(2*A);
           *X2=  (-B-D)/(2*A);
           return 1;
        }
}

main()
{ float    A,B,C,X,Y;
  scanf("%f%f%f",&A,&B,&C);
  if ( radici(A,B,C,&X,&Y) )
             printf("%f%f\n",X,Y);
}
```

Osservazioni

- Quando un puntatore è usato per realizzare il passaggio per riferimento, la funzione *non dovrebbe mai alterare il valore del puntatore.*
- Quindi, se **`a`** e **`b`** sono due parametri formali di tipo puntatore:

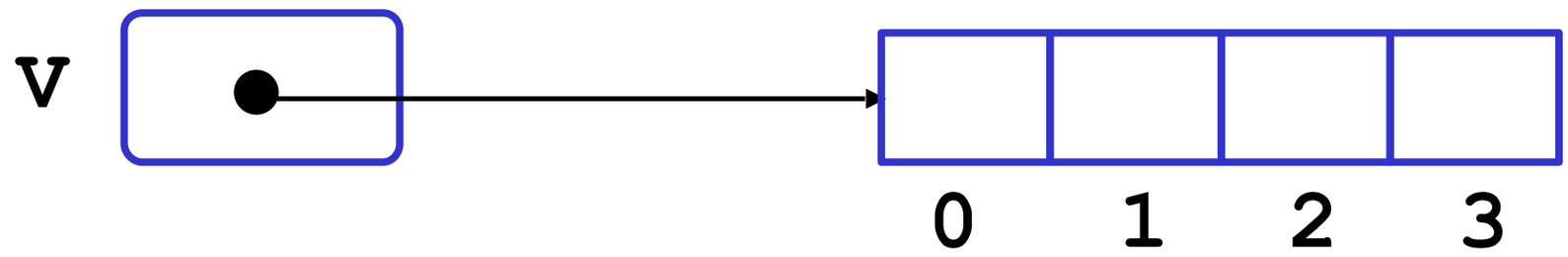
```
*a = *b        SI
a = b          NO
```

- In generale una funzione può modificare un puntatore, ma *non è opportuno che lo faccia se esso realizza un passaggio per riferimento*

Vettori come parametri di funzioni

Ricordiamo che il nome di un vettore denota il ***puntatore al suo primo elemento***.

```
int V[4];
```



Quindi, *passando un vettore a una funzione:*

- *non si passa l'intero vettore !!*
- si passa solo (***per valore!***) il suo indirizzo iniziale (`V ≡ &V[0]`)

- *agli occhi dell'utente,* ***sembra*** che il vettore ***sia passato per riferimento!***

Conclusione

A livello *concreto*:

- il C passa i parametri *sempre e solo per valore*
- nel caso di un vettore, si passa il suo indirizzo iniziale (`v` ≡ `&v[0]` ≡ α) *perché tale è il significato del nome del vettore*

A livello *concettuale*:

- il C passa *per valore* tutto tranne i vettori, che vengono trasferiti *per riferimento.*

ESERCIZIO: Ordinamento di un vettore

Torniamo al problema dell'***ordinamento di un vettore***:

Dati n valori interi forniti in ordine qualunque, stampare in uscita l'elenco dei valori dati in ordine crescente.

Soluzione, mediante le tre funzioni:

- **leggi**: inizializza il vettore con i valori dati da input;
- **ordina:** applicando il metodo naïve sort, ordina in modo crescente gli elementi del vettore,
- **stampa:** scrive sullo standard output il contenuto del vettore ordinato.

ESERCIZIO: Ordinamento di un vettore

```
#include <stdio.h>
#define N 5
void leggi(int a[], int dim){..} /*lettura dati */
void stampa(int a[] , int dim) {..} /*stampa*/
void ordina (int vet[] , int dim) {..} /* ordina /*

main (){
    int i, a[N];

    leggi(a, N);
    ordina(a, N);
    stampa(a, N);
}
```

Esercizio: definizioni delle funzioni

```
void  ordina (int vet[], int dim){
    int  j, i, min, temp;
    for  (j=0; j < dim; j++){
         min=j;
        for (i=j+1;i<dim; i++)
             if (vet[i]<vet[min]) min=i;
        if (min != j)   /*scambio */
        {
             temp=vet[min];
             vet[min]=vet[j];
             vet[j]= temp;
        }
    }
}
```

Esercizio: definizioni delle funzioni

```
void leggi(int a[], int dim){
    int i;

    printf ("Scrivi %d interi\n", dim);
    for (i = 0; i < dim; i++)
        scanf ("%d", &a[i]);
}
```

```
void stampa(int a[], int dim){
    int i;

    printf ("Vettore:\n");
    for (i = 0; i < dim; i++)
        printf ("%d\t", a[i]);
}
```

Effetti collaterali (side effects)

In C la distinzione tra funzione e procedura non è, come abbiamo visto, nitida: dipende tutto dal modo in cui sono passati i parametri

- Una funzione puo` provocare un ***effetto collaterale (side effect)*** se la sua attivazione modifica una qualunque tra le variabili definite all'esterno di essa.
- Si possono verificare effetti collaterali nei seguenti casi:
  - parametri di tipo ***puntatore***;
  - assegnamento a ***variabili globali***.

Effetti collaterali

Se presenti, e` possibile realizzare "funzioni" che non sono piu` funzioni in senso matematico.

Esempio:

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

main()
{ B=1;
  printf("%d\n",2*f(&B));   /* (1) */
  B=1;
  printf("%d\n",f(&B)+f(&B)); /* (2) */
}

int f (int * A)
{ *A=2*(*A);
  return *A;
}
```

→Fornisce valori diversi, pur essendo attivata con lo stesso parametro attuale.

L'istruzione (1) stampa 4 mentre l'istruzione (2) stampa 6.