



Linguaggio C: le Funzioni

Sottoprogrammi

Spesso può essere utile avere la possibilità di costruire nuove istruzioni, o nuovi operatori che risolvano parti specifiche di un problema:

Un **sottoprogramma** permette di dare un **nome** a una parte di programma, rendendola *parametrica*.

Esempio: algoritmo *naïve sort*

```
#include <stdio.h>
#define dim 10
main()
{   int V[dim], i,j, max, tmp;

    /* lettura dei dati */
    for (i=0; i<dim; i++)
    {   printf("valore n. %d: ",i);
        scanf("%d", &V[i]);
    }
    /*ordinamento */
    for(i=dim-1; i>1; i--)
    {   max=i;
        for( j=0; j<i; j++)
        if (V[j]>V[max])
            max=j;
        if (max!=i) /*scambio */
        {   tmp=V[i];
            V[i]=V[max];
            V[max]=tmp;
        }
    }
    /* stampa */
    for (i=0; i<dim; i++)
        printf("\n%d", V[i]);
}
```

Limiti di questa soluzione:

- difficile leggibilità
- funziona solo con vettori di 10 elementi
- non è riutilizzabile

Soluzione:

si può assegnare un nome ad ogni parte del programma, racchiudendone le istruzioni che la definiscono all'interno di un **componente software riutilizzabile**:

il sottoprogramma.

Esempio: algoritmo *naïve sort*

```
#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** e **stampa** sono nomi di **sottoprogrammi**, ognuno dei quali rappresenta una parte del programma (nella prima versione).

• **leggi(V, dim)** : V e dim sono parametri, e rappresentano i dati dell'algoritmo che il sottoprogramma rappresenta

Vantaggi di questa soluzione:

- leggibilità
- sintesi
- possibilità di riutilizzo delle diverse parti

Riutilizzabilita`

Mediante i sottoprogrammi e` possibile eseguire piu` volte lo stesso insieme di operazioni senza doverlo riscrivere.

Ad esempio: ordinamento di due vettori.

```
#include <stdio.h>
#define dim 10
#define dim2 25
..
main()
{ int V1[dim], V2[dim2];
  leggi(V1, dim);
  leggi(V2, dim2);
  ordina(V1, dim);
  ordina(V2, dim2);
  stampa(V1, dim);
  stampa(V2, dim2);
}
```

Sottoprogrammi: funzioni e procedure

Un sottoprogramma e` una nuova istruzione, o un nuovo operatore definito dal programmatore per sintetizzare una sequenza di istruzioni.

In particolare:

- **procedura**: e` un sottoprogramma che rappresenta un'istruzione non primitiva
- **funzione**: e` un sottoprogramma che rappresenta un operatore non primitivo.

Tutti i linguaggi di alto livello offrono la possibilita` di definire funzioni e/o procedure.

→ Il linguaggio C realizza solo il concetto di **funzione**.

Funzioni come componenti software

Una funzione è un "*componente software*" che cattura l'idea matematica di *funzione*:

- molti possibili **ingressi** (che non vengono modificati!)
- una sola uscita (il **risultato**)



- Una funzione:
 - riceve dati di ingresso attraverso i **parametri**
 - esegue una **espressione**, la cui valutazione fornisce un **risultato**
 - denota un **valore** in corrispondenza al suo *nome*

Funzioni come componenti software

Esempio: Data una funzione $f : \mathbb{R} \rightarrow \mathbb{R}$

$$f(x) = 3 * x^2 + x - 3$$

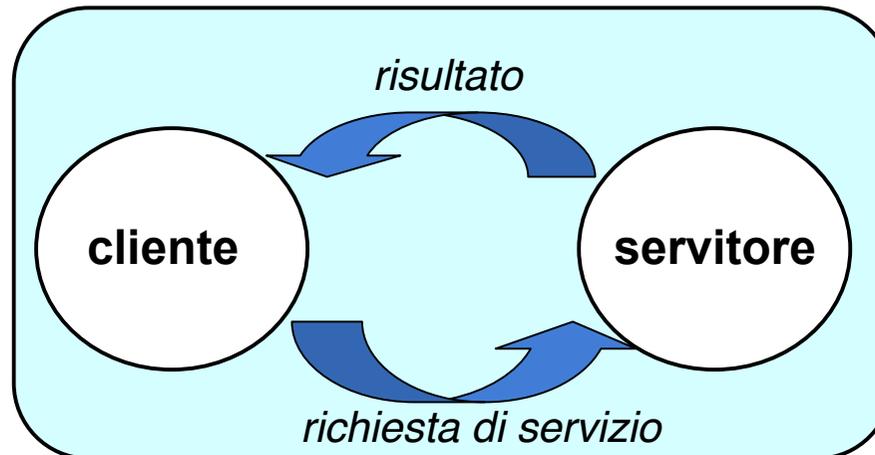
→ se x vale 1 allora $f(x)$ denota il valore 1.

Funzioni

Il meccanismo di uso di funzioni nei linguaggi di programmazione fa riferimento allo schema di interazione tra componenti software

cliente/servitore
(*client - server*)

Modello Cliente-Servitore



Servitore:

- un qualunque ente capace di nascondere la propria organizzazione interna
- presentando ai clienti una precisa *interfaccia* per lo scambio di informazioni.

Cliente:

- qualunque ente in grado di invocare uno o più servitori per ottenere servizi.

Modello Cliente-Servitore

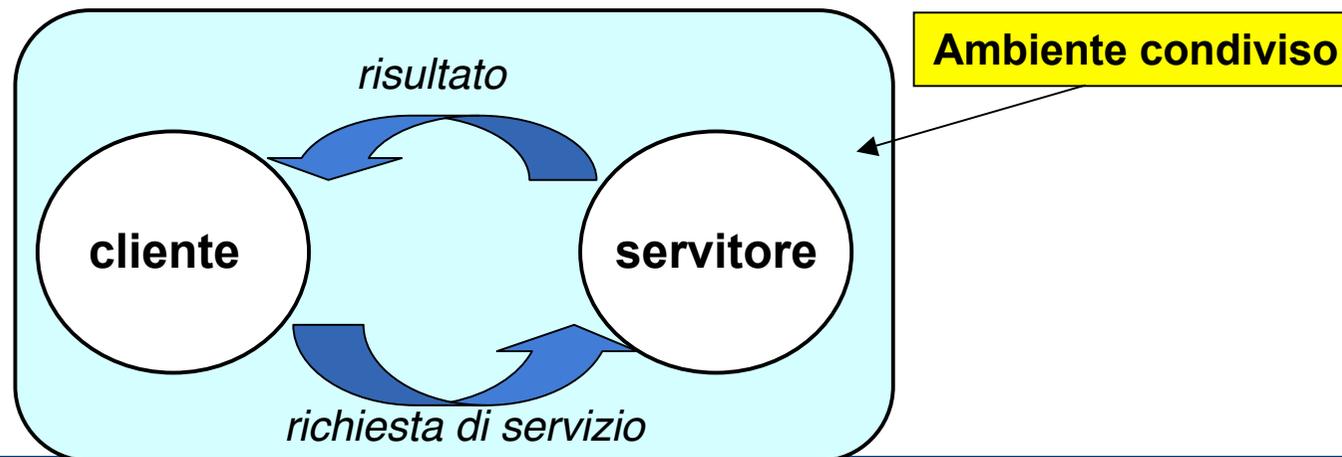
In generale, un servitore può

- essere **passivo** o **attivo**
- servire molti clienti, oppure costituire la risorsa privata di uno specifico cliente
 - in particolare: può servire un cliente alla volta, in sequenza, oppure più clienti per volta, in parallelo
- trasformarsi a sua volta in cliente, invocando altri servitori o anche se stesso.

Comunicazione Cliente/Servitore

Lo scambio di informazioni tra un cliente e un servitore può avvenire

- in modo **esplicito** tramite le interfacce stabilite dal servitore
- in modo **implicito** tramite dati accessibili ad entrambi (*l'ambiente condiviso*).



Funzione come servitore

Una funzione è un **servitore**:

- **passivo**
 - che realizza un **particolare servizio**
 - che **serve un cliente per volta**
 - che **può trasformarsi in cliente** invocando altre funzioni (o eventualmente se stessa)
-
- Il cliente *chiede* al servitore di svolgere il servizio
 - *chiamando* tale servitore (per **nome**)
 - fornendogli i dati necessari (**parametri**)
-
- ➔ Nel caso di una funzione, cliente e servitore comunicano mediante *l'interfaccia* della funzione.

Interfaccia di una funzione

L'interfaccia (o *intestazione, firma, signature*) di una funzione comprende

- **nome** della funzione
- lista dei **parametri**
- **tipo del valore** calcolato dalla funzione

→ enuncia le regole di comunicazione tra cliente servitore.

Cliente e servitore comunicano quindi mediante:

- i **parametri** trasmessi dal cliente al servitore all'atto della chiamata (direzione: **dal cliente al servitore**)
- il **valore** restituito dal servitore al cliente (direzione: dal servitore al cliente)

Interfaccia: esempio

```
int max (int x, int y )/* interfaccia */  
{  
    if (x>y) return x;  
    else return y;  
}
```

- Il simbolo **max** denota il nome della funzione
- Le variabili intere **x** e **y** sono i parametri della funzione
- Il valore restituito è un intero **int** .

Comunicazione cliente → servitore

La comunicazione **cliente è servitore** avviene mediante i **parametri**.

- **Parametri formali :**
 - sono specificati nell'interfaccia del servitore
 - indicano cosa il servitore si aspetta dal cliente
- **Parametri attuali :**
 - sono trasmessi dal cliente all'atto della chiamata
 - devono corrispondere ai parametri formali in **numero, posizione e tipo**.

Esempio

Parametri Formali

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

SERVITORE:
definizione
della
funzione

```
main () {  
    int z = 8;  
    int m;  
    m = max (z , 4);  
    ... }
```

Parametri Attuali

CLIENTE:
chiamata
della
funzione

Comunicazione cliente/servitore

L'associazione (*legame*) tra i parametri attuali e i parametri formali viene fatta *al momento della chiamata*, in modo **dinamico**.

Tale legame:

- ❑ vale solo per l'invocazione corrente
- ❑ vale solo per la durata dell'esecuzione della funzione.

ESEMPIO

Parametri Formali

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

```
main () {  
    int z = 8;  
    int m;  
    m1 = max(z, 4);  
    m2 = max(5, z);  
}
```

All'atto di questa chiamata della funzione si effettua un legame tra:

x e z

y e 4

ESEMPIO

Parametri Formali

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

```
main () {  
    int z = 8;  
    int m;  
    m1 = max (z, 4);  
    m2 = max (5, z);  
}
```

All'atto di questa
chiamata della
funzione si effettua un
legame tra

x e 5

y e z

Programmi C con funzioni

Ogni funzione viene realizzata mediante la **definizione** di una *unita` di programma* distinta dal programma principale (main).

- Generalizziamo la struttura di un programma C:
il programma e` una collezione di *unita` di programma* (tra le quali compare sempre l'*unita` main*)

Richiamiamo la regola generale sulla visibilita` degli identificatori:

"Prima di utilizzare un identificatore e` necessario che sia gia` stato definito (oppure dichiarato)."

- all'interno del file sorgente vengono specificate **prima le definizioni delle funzioni**, ed infine viene esplicitato il **main**.
- formalmente anche il main e` una funzione; essa viene invocata per prima, quando il programma viene messo in esecuzione

Esempio

```
<definizione funzione 1>  
<definizione funzione 2>  
...  
main()  
{  
...  
<chiamata di funzione 1>  
<chiamata di funzione 2>  
..  
}
```

Definizione di funzione in C

```
<definizione-di-funzione> ::=  
<tipoValore>      <nome> (<parametri-formali>)  
{  
    <corpo>;  
}
```

dove:

<parametri-formali>

- una lista (eventualmente vuota) di variabili, visibili dentro il corpo della funzione.

<tipoValore>

- deve coincidere con il tipo del valore risultato della funzione: puo` essere di tipo semplice (int, char, float o double), di tipo struct, oppure di tipo puntatore.

Definizione di funzione in C

```
<definizione-di-funzione> ::=  
<tipoValore> <nome> (<parametri-formali>)  
{  
    <corpo>;  
}
```

- Nella parte **<corpo>** possono essere presenti definizioni e/o dichiarazioni locali (*parte dichiarazioni*) e un insieme di istruzioni (*parte istruzioni*).
- I dati riferiti nel corpo possono essere **costanti**, **variabili**, oppure **parametri formali**.
- All'interno del corpo, i **parametri formali** vengono trattati come **variabili**.

Meccanismo di chiamata di funzioni

- All'atto della *chiamata*, l'esecuzione del cliente viene **sospesa** e il controllo passa al servitore.
- Il servitore "vive" solo per il tempo necessario a svolgere il servizio.
- Al termine, il servitore "muore", e l'esecuzione torna al *cliente*.

Chiamata di Funzione

La chiamata di funzione è un'espressione della forma:

```
<nomefunzione> ( <parametri-attuali> )
```

dove:

```
<parametri-attuali> ::=  
  [ <espressione> ] { , <espressione> }
```

ESEMPIO

Parametri Formali

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

SERVITORE
definizione
della
funzione

```
main () {  
    int z = 8;  
    int m;  
    m = max (z , 4);  
}
```

CLIENTE
chiamata
della
funzione

Parametri Attuali

Risultato di una funzione: return

L'istruzione

```
return <espressione>
```

provoca la terminazione dell'attivazione della funzione (*il servitore muore*) e la restituzione del controllo al cliente, unitamente al valore dell'espressione che la segue.

- Eventuali istruzioni successive alla return ***non saranno mai eseguite!***

```
int max (int x, int y ) {  
    if (x>y) return x;  
    else return y;  
    printf("ciao!"); /* mai eseguita !*/  
}
```

ESEMPIO

Parametri Formali

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

SERVITORE
definizione
della
funzione

```
main () {  
    int z = 8;  
    int m;  
    m = max (z , 4) ;  
}
```

CLIENTE
chiamata
della
funzione

Risultato

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) ;
}
```

BINDING & ENVIRONMENT

`return x;`  devo sapere cosa denota il simbolo x

- La conoscenza di cosa un simbolo denota viene espressa da una *legame* (*binding*) tra il simbolo e un valore.
- L'insieme dei *binding* validi in (un certo punto di) un programma si chiama *environment* (*ambiente*).

ESEMPIO

```
main() {  
    int z = 8;  
    int y, m;  
    y = 5  
    m = max(z, y) ; /* X */  
}
```

Consideriamo il punto X: In questo environment il simbolo `z` è legato al valore 8 tramite l'inizializzazione, mentre il simbolo `y` è legato al valore 5. Pertanto i parametri di cui la funzione `max` ha bisogno per calcolare il risultato sono noti all'atto dell'invocazione della funzione

ESEMPIO

```
main () {  
    int z = 8;  
    int y, m;  
    m = max(z, y); /* X */  
}
```

Punto X: In questo environment il simbolo `z` è legato al valore 8 tramite l'inizializzazione, mentre il simbolo `y` non è legato ad alcun valore. Pertanto i parametri di cui la funzione `max` ha bisogno per calcolare il risultato **NON** sono noti all'atto dell'invocazione della funzione e la funzione non può essere valutata correttamente

ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main(){  
    int z = 8;  
    int m;  
    m = max(2*z,13);  
}
```

ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main() {  
    int z = 8;  
    int m;  
    m = max(2*z, 13);  
}
```

*Valutazione del simbolo z
nell'environment corrente
z vale 8*



ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main() {  
    int z = 8;  
    int m;  
    m = max(2*z, 13);  
}
```

*Calcolo dell'espressione
2*z nell'environment
corrente*

*2*z vale 16*

ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main() {  
    int z = 8;  
    int m;  
    m = max(2*z, 13);  
}
```

*Invocazione della
chiamata a max con
parametri attuali 16 e 13
IL CONTROLLO PASSA
AL SERVITORE*

ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main() {  
    int z = 8;  
    int m;  
    m = max(2*z, 13);  
}
```

Viene effettuato il legame dei parametri formali x e y con quelli attuali 16 e 13.

INIZIA L'ESECUZIONE DEL SERVITORE

ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main() {  
    int z = 8;  
    int m;  
    m = max(2*z, 13);  
}
```

*Viene valutata
l'istruzione if (16 > 13)
che nell'environment
corrente e' vera.
Pertanto si sceglie la
strada*

`return x`

ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main() {  
    int z = 8;  
    int m;  
    m = max(2*z, 13);  
}
```

*Il valore 16 viene
restituito al cliente.*

**IL SERVITORE
TERMINA E IL
CONTROLLO PASSA
AL CLIENTE.**

*NOTA: i binding di x e y
vengono distrutti*

ESEMPIO

- Il servitore...

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

- ... e un possibile cliente:

```
main() {  
    int z = 8;  
    int m;  
    m = max(2*z, 13);  
}
```

*Il valore restituito (16)
viene assegnato alla
variabile m
nell'environment del
cliente.*

RIASSUMENDO...

All'atto dell'invocazione di una funzione:

- si crea una *nuova attivazione (istanza) del servitore*
- si alloca la memoria per i parametri (e le eventuali variabili locali)
- si trasferiscono i parametri al servitore
- si trasferisce il controllo al servitore
- si esegue il codice della funzione.

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 produce alcun risultato
- dall'interno del corpo della funzione `proc` non verrà restituito alcun risultato al cliente:
 - il corpo potrà contenere o meno l'istruzione `return`, eventualmente utilizzata senza argomento: `return;`

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"*/
}
```

Tecniche di legame dei parametri

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

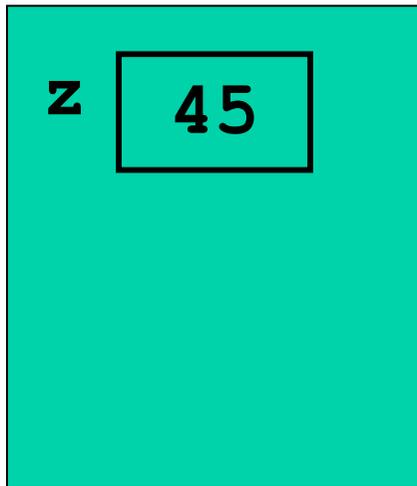
In generale, un parametro può essere trasferito (*passato*) dal cliente al servitore:

- **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

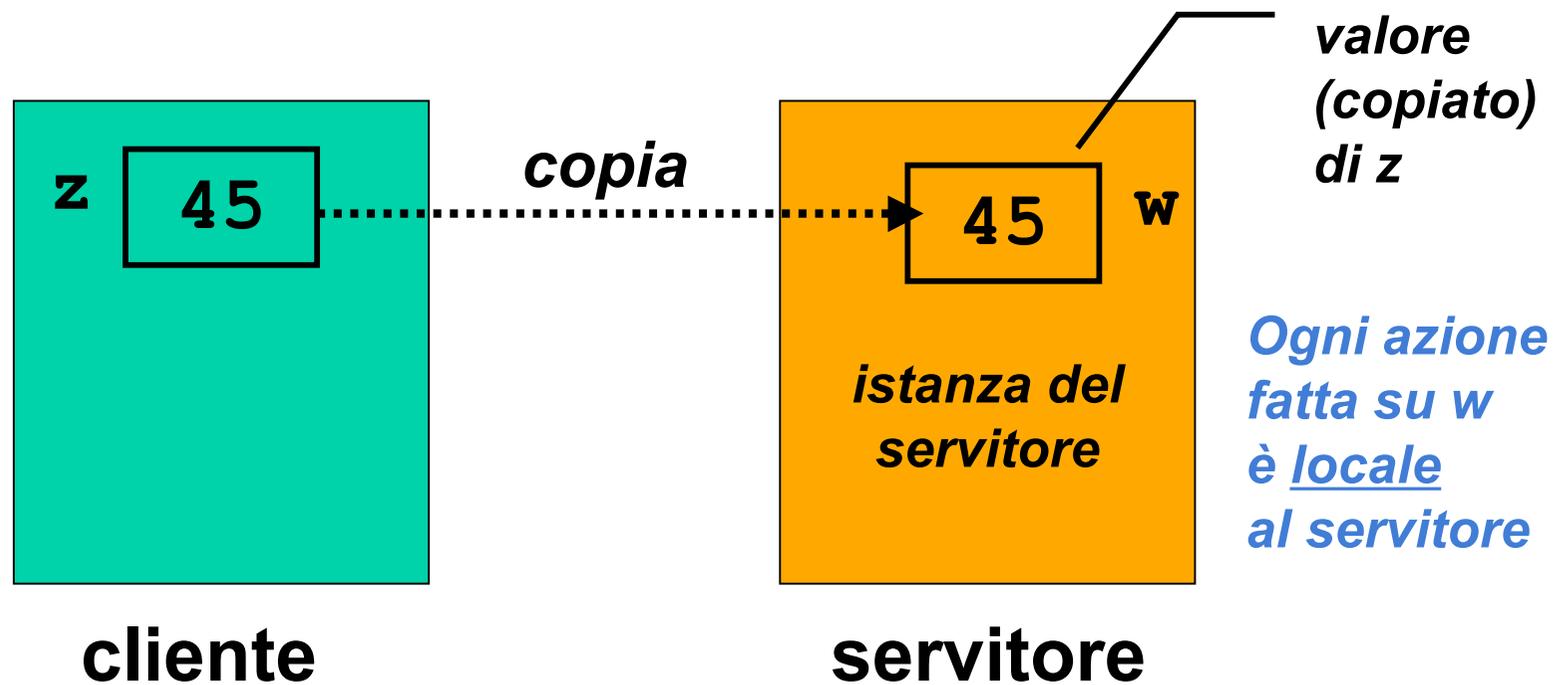
si trasferisce una copia del valore del parametro attuale...



cliente

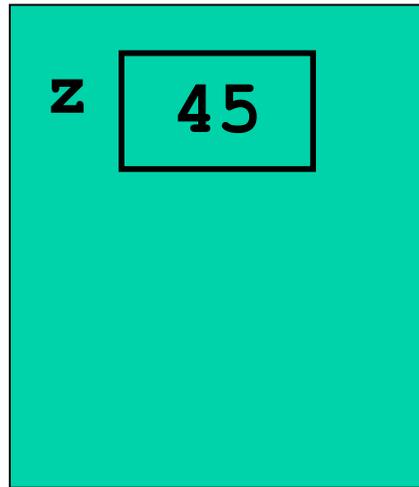
Legame per valore

si trasferisce una copia del valore del parametro attuale nel parametro formale



Legame per riferimento

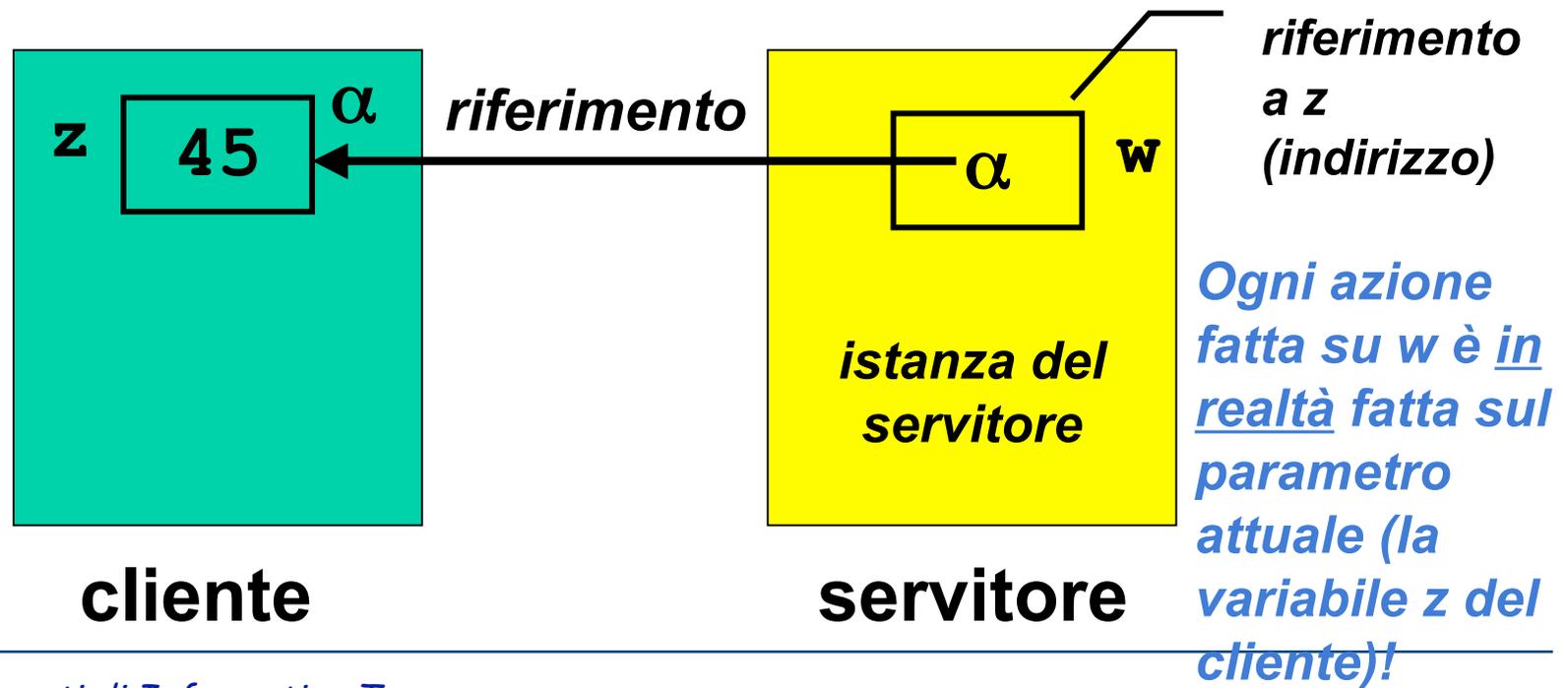
si trasferisce un riferimento al parametro attuale...



cliente

Legame per riferimento

- si trasferisce un riferimento al parametro attuale che viene associato al parametro formale



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

x 

- **Servitore: modifica** 

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

valAss restituisce il valore 87 che viene assegnato a absz

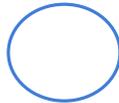
- **Cliente**

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

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);  
}
```

NOTA: IL VALORE DI z NON VIENE MODIFICATO

La printf stampa -87

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.

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 è necessariamente una variabile, ma può 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?

→ Utilizzando parametri di tipo puntatore.

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**

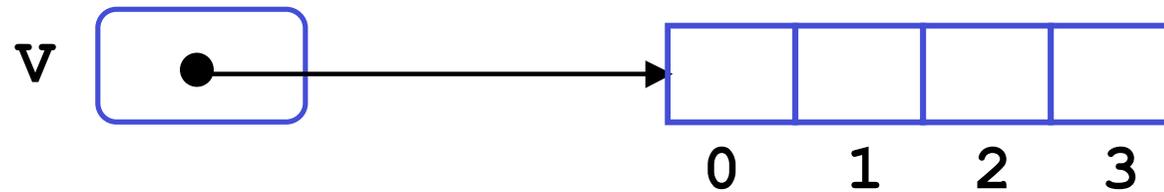
a = b **NO**

- 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
($V \equiv \&V[0]$)
-
- 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 ($v \equiv \&v[0] \equiv \alpha$) *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*.

ESEMPIO

Problema:

Scrivere *una funzione* che, dato un vettore di N interi, ne calcoli il massimo.

Definiamo la funzione:

```
int massimo(int *v, int dim){..}
```

ESEMPIO

La funzione:

```
int massimo(int *v, int dim) {  
    int i, max;  
    for (max=v[0], i=1; i<dim; i++)  
        if (v[i]>max) max=v[i];  
    return max;  
}
```

ESEMPIO

Il cliente:

```
main() {  
    int max, v[] = {43,12,7,86};  
    max = massimo(v, 4);  
}
```



Trasferire esplicitamente la dimensione del vettore è NECESSARIO, in quanto la funzione, ricevendo solo l'indirizzo iniziale, non avrebbe modo di sapere quanto è lungo il vettore !

ESERCIZIO: MAX E MIN DI UN VETTORE

```
#define DIM 15
/* definizione delle due funzioni */

int minimo (int vet[], int N)
{int i, min;
  min = vet[0];
  for (i = 1; i < N; i ++)
    if (vet[i]<min)
      min = vet[i];
  return min;
}

int massimo (int vet[], int N)
{int i, max;
  max = vet[0];
  for (i = 1; i < N; i ++)
    if (vet[i]>max)
      max=vet[i];
  return max;
}
/* continua...*/
```

ESERCIZIO: MAX E MIN DI UN VETTORE

```
/* ...continua*/

main ()
{int i, a[DIM];
 printf ("Scrivi %d numeri interi\n", DIM);
 for (i = 0; i < DIM; i++)
     scanf ("%d", &a[i]);
 printf ("L'insieme dei numeri è: ");
 for (i = 0; i<DIM; i++)
     printf(" %d",a[i]);
 printf ("Il minimo vale %d e il
         massimo è %d\n", minimo(a,DIM), massimo(a,
         DIM));
}
```

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]);
}
```

Dichiarazione di funzione

Regola Generale:

Prima di utilizzare una funzione e' necessario che sia gia' stata *definita* oppure *dichiarata*.

Funzioni C:

- **definizione**: descrive le proprieta' della funzione (tipo, nome, lista parametri formali) e la sua realizzazione (lista delle istruzioni contenute nel blocco).
- **dichiarazione** (prototipo): descrive le proprieta' della funzione senza esplicitarne la realizzazione (blocco)
 - serve per "anticipare" le caratteristiche di una funzione definita successivamente.

Dichiarazione di una funzione:

La dichiarazione di una funzione si esprime mediante l'intestazione della funzione, seguita da ";":

```
<tipo-ris> <nome> ([<lista-par-formali>]);
```

Ad esempio: Dichiarazione della funzione ordina:

```
void ordina(int vet[], int dim);
```

Dichiarazione di Funzioni

Una funzione puo` essere dichiarata piu` volte (in punti diversi del programma), ma e` definita **una sola volta**.

E` possibile inserire i prototipi delle funzioni:

- nella parte dichiarazioni globali di un programma,
- nella parte dichiarazioni del main,
- nella parte dichiarazioni delle funzioni.

Dichiarazione di funzioni

Ad esempio:

```
#include <stdio.h>
long power (int base, int n); /* dichiarazione */

main()
{ long power (int base, int n); /* dichiarazione */
  int X, exp;
  scanf("%d%d", &X, &exp);
  printf("%ld", power(X,exp));
}

long power(int B, int N)
{ int i, RIS;
  for (RIS=B, i=1; i<N; i++) RIS*=B;
  return RIS;
}
```

Dichiarazioni di funzioni di libreria

- E le dichiarazioni di `printf`, `scanf`, ecc.?

→ sono contenute nel file `stdio.h`

→ Ad esempio, la direttiva:

```
#include <stdio.h>
```

provoca l'aggiunta nel file sorgente del contenuto del file specificato, cioè delle dichiarazioni delle funzioni della libreria di I/O.

- Analogamente per le altre funzioni di libreria:

→ le dich. di `malloc` e `free` sono contenute nel file `stdlib.h`

→ le dich. di `strlen`, `strcmp`, etc. sono contenute nel file `string.h`

→ ecc.

Struttura dei Programmi C

Per migliorare la leggibilità, spesso si strutturano i programmi in modo tale che la definizione del `main` compaia prima delle definizioni delle altre funzioni.

- **Convenzione:**

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

Esempio

```
#include <stdio.h>
#define N 5
/* dichiarazioni delle funzioni: */
void leggi(int a[], int dim);
void stampa(int a[] , int dim);
void ordina (int vet[] , int dim);

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

    leggi(a, N);
    ordina(a, N);
    stampa(a, N);
}
/* continua...*/
```

```

/*... continua: 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;
       }
    }
}

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]);
} /* fine sorgente*/

```

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

Esempio: Divisione intera x/y con calcolo di quoziente e resto. Occorre calcolare *due* valori che supponiamo di mettere in due variabili globali

```
int quoziente, int resto;
```

variabili globali **quoziente** e **resto** visibili in tutti i blocchi

```
void dividi(int x, int y) {  
    resto = x % y; quoziente = x/y;  
}
```

```
main() {  
    dividi(33, 6);  
    printf("%d%d", quoziente, resto);  
}
```

Il risultato è disponibile per il cliente nelle variabili globali **quoziente** e **resto**

ESEMPIO

Esempio: Con i parametri di tipo puntatore avremmo il seguente codice

```
void dividi(int x, int y, int* quoziente, int* resto)
{
    *resto = x % y; *quoziente = x/y;
}

main() {
    int k = 33, h = 6, quoz, rest;
    int *pq = &quoz, *pr = &rest;
    dividi(33, 6, pq, pr);
    printf("%d%d", quoz, rest);
}
```

Effetti collaterali (side effects)

- Una funzione può 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.

Visibilita` degli Identificatori

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 (ad esempio, le variabili globali).
- **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?

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 3);
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 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 in P (**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);
  }
}
```

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.