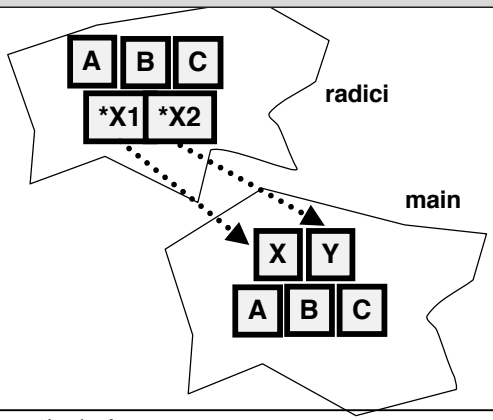
Il linguaggio C
Passaggio dei parametri



Tecniche di legame dei parametri

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

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

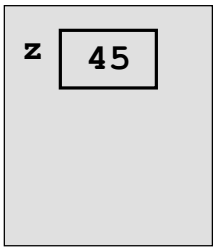
- **per valore (per copia, *by value*)**
  - **si copia *il valore* del parametro effettivo nel corrispondente parametro formale.**
- **per riferimento (per indirizzo, *by reference*)**
  - **si associa al parametro formale *un riferimento* al corrispondente parametro effettivo**

13

Legame per valore

HP: z parametro effettivo
w parametro formale

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

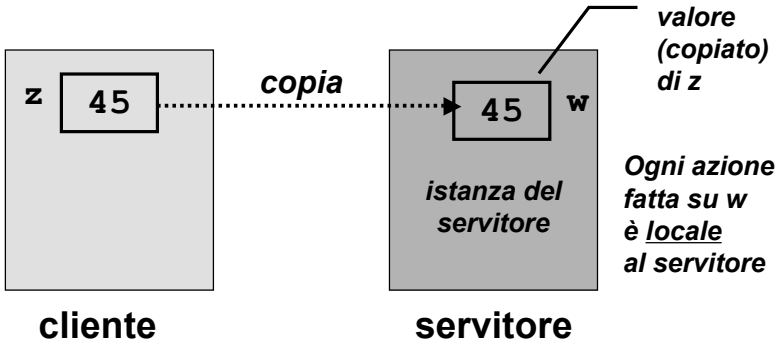


cliente

13

Legame per valore

si trasferisce *una copia del valore* del parametro attuale nel parametro effettivo



1

Legame per riferimento

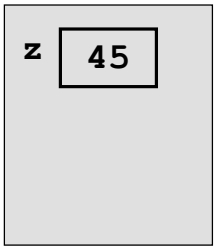
Molti linguaggi mettono a disposizione il passaggio per riferimento (*by reference*)

- ***non*** si trasferisce una copia del valore del parametro effettivo, bensì…
- *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.

13

Legame per riferimento

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

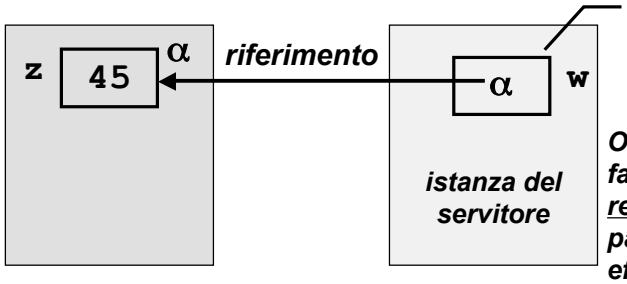


cliente

13

Legame per riferimento

si trasferisce *un riferimento* al parametro effettivo che viene associato al parametro formale



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

1

Riassumendo…

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 sia in uscita*** al/dal sottoprogramma.
- Se il passaggio avviene per riferimento, ogni parametro effettivo deve necessariamente essere una **variabile**.

3

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

1 3

Passaggio per riferimento in C

- Il C *non* fornisce *direttamente* un modo per attivare il passaggio per riferimento.
- In alcuni casi il passaggio per riferimento è ***indispensabile****:* ad esempio, nel caso di funzioni che producono più di un risultato.
- quindi, dobbiamo *costruircelo.*

È possibile costruirlo? Come?

➔ Utilizzando parametri di tipo puntatore.

REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

- In C è 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

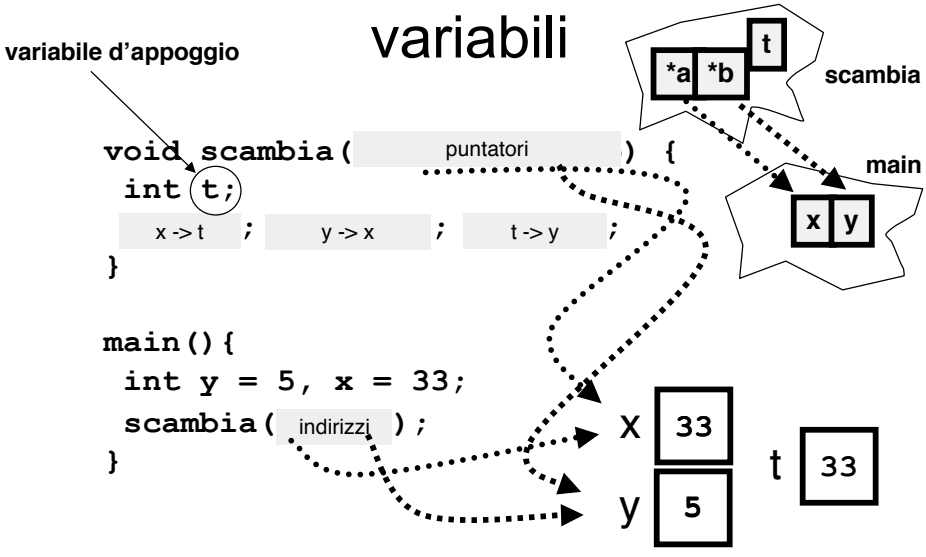
nel servitore: puntatori come parametri formali

```
#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 è: %f, il
  suo cubo è %f\n", dato, quadrato, cubo);
}
```

nel cliente: vengono passati degli indirizzi

Esempio: scambio di valori tra variabili

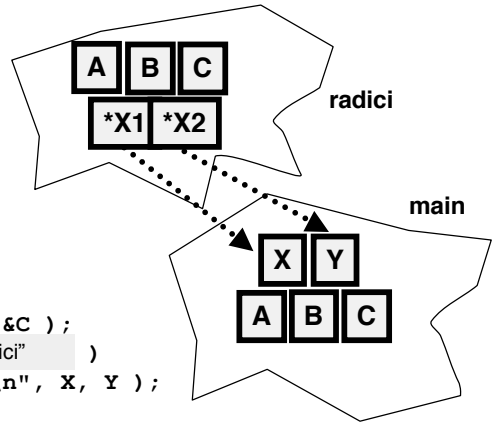


Equazione di secondo grado: $Ax^2 + Bx + C = 0$

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

int radici( A , B , C , X1 , X2 )
{  float D;
   if ( !A )  errore…
   D = B*B-4*A*C;
   if ( D<0 )  errore…
   D=sqrt(D);
          x=(-B+D)/2a
          y=(-B-D)/2a
}

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



13

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**

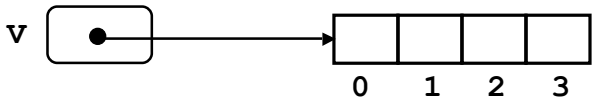
a = b **NO**

- 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.*

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 procedure:

- **`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
int  leggi(                    ) {…}; /* lettura dati */
void stampa(                   ) {…}; /* stampa */
void ordina (                  ) {…}; /* ordina */
void scambia(int* a, int* b) /* ..vista prima.. */

main () {




}
```


Esercizio: definizioni delle funzioni

```
/* legge da tastiera e scrive sul vettore "a"
   fino a un massimo di "dim" interi */
int leggi(int a[], int dim) {
 int i=0;
 while( scanf ("%d", a+i)>0 ) {
       i++;
       if( i>=dim ) break;
 }
 return i;
}

/* stampa su video i "dim" elem. di un vettore
   "a" di interi */
void stampa(int a[], int dim) {
 int i;
 printf("\nVettore:\n");
 for (i = 0; i < dim; i++)
      printf ("%d\n", a[i]);
}

/* scambia il valore di due variabili intere */
void scambia(int* a, int* b) {  /* già vista */
  int t;
  t = *a;  *a = *b;  *b = t;
}
```

Dichiarazione di funzioni (prototipi)

Regola Generale:
Prima di utilizzare una funzione è necessario che sia già stata ***definita*** oppure ***dichiarata***.

Funzioni C:

- ***definizione***: descrive le proprietà della funzione (tipo, nome, lista parametri formali) e la sua realizzazione (lista delle istruzioni contenute nel blocco).
- ***dichiarazione*** (prototipo): descrive le proprietà 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 può essere dichiarata più volte (in punti diversi del programma), ma è definita **una sola volta**.

È possibile inserire i prototipi delle funzioni:

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

NOTA: Nomi dei parametri nelle dichiarazioni di funzioni:

```
void leggi(int a[], int dim);
void leggi(int *a, int N);
void leggi(int *, int);
```

non è richiesto il nome dei parametri formali

Dichiarazione di funzioni

Ad esempio:

dichiarazioni globali

```
#include <stdio.h>
long power (int, int); /* 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) { /* definizione */
   int i, RIS;

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

dichiarazioni del main

Dichiarazioni di funzioni di libreria

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

➔ sono contentue 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 contentue nel file **`stlib.h`**

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

➔ ecc.

Esempio: max elem. in un vettore

```
#include <stdio.h>
int massimo(int *v, int dim);
```

dichiarazioni

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

 max = massimo(v, 4);

 printf("Massimo: %d\n", max);
}


int massimo(int *v, int dim) {
 int max, i;

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

definizioni

Esempio: max elem. in un vettore

```
#include <stdio.h> /* dichiarazioni di funz.,costanti,… */
int massimo(int *v, int dim); /* dichiarazioni di funz. */
```

dichiarazioni

```
main() {                  /* definizione della funzione main*/
 int max, v[] = {43,12,7,86}; /* definizione var. locali
   */

 max = massimo(v, 4); /* istruzioni che usano funzioni
                         (massimo, printf), e dati (max e v) */
 printf("Massimo: %d\n", max);
}


int massimo(int *v, int dim) { /* definizione di massimo
   */
 int max, i;          /* definizioni delle variabili locali */

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

definizioni

Il linguaggio C
Visibilità e tempo di vita delle variabili

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


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 il 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;
  dividi( 33, 6, &quoz, &rest );
  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, è possibile realizzare "funzioni" che non sono più 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.

Visibilità degli Identificatori

Dato un programma scritto in linguaggio C, è possibile distinguere:

- **Ambiente globale:**
  è costituito dalle dichiarazioni e definizioni che compaiono nella parte di dichiarazioni globali di P (ad esempio, le variabili globali).

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

Regole di visibilità degli identificatori in C

Qual è il campo di azione di un identificatore?

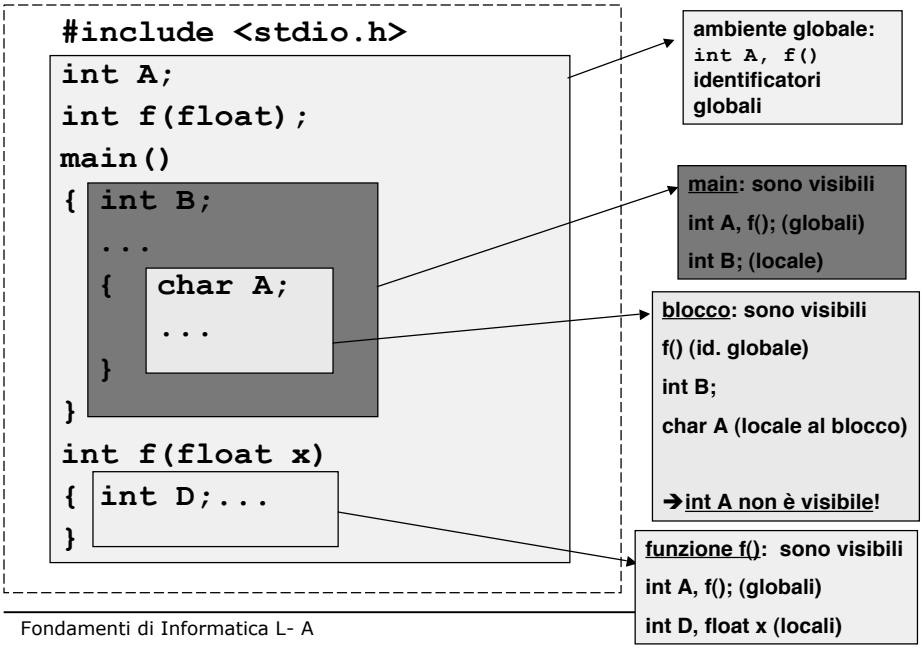
Nel linguaggio C è determinato *staticamente*, in base all'ambiente al quale l'identificatore appartiene, secondo le seguenti regole:

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** è 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 è 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***).

Visibilità degli identificatori



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("  \t  \t",i,j);
   }
   printf("  \t\n",i);
  }
}
```

Tempo di vita delle variabili

1 3

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

➔ È l'intervallo di tempo in cui la variabile ***esiste*** ed in cui, compatibilmente con le regole di visibilità, può 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 è 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 è 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.

3

Variabili `static`

- È 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`:**

- ✓ **è creata all'inizio del programma, inizializzata a 0, e deallocata alla fine dell'esecuzione;**
- ✓ **la sua visibilità è limitata al corpo della funzione f,**
- ✓ **il suo tempo di vita è pari al tempo di esecuzione dell'intero programma**
- ✓ **è 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` è 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.**