Allocazione Dinamica della memoria

- Il C mette a disposizione delle primitive per la gestione dinamica della memoria, grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdlib.h>`
- Le primitive principali sono
  - malloc,calloc, e realloc: consentono di allocare dinamicamente memoria per una variabile di un tipo specificato o per array di dimensione non nota a priori;
  - free: consente di rilasciare dinamicamente memoria (precedentemente allocata con malloc)
- I tipi di dato sono ancora statici, ovvero hanno una dimensione fissata a priori. Le variabili di un certo tipo di dato possono invece essere create.

malloc

- La chiamata di funzione

  malloc(sizeof(TipoDato));

  crea in memoria una variabile di tipo TipoDato, e restituisce come risultato l'indirizzo della variabile creata (NULL se fallisce).
- Se p è una variabile di tipo puntatore a TipoDato, l'istruzione

  p=malloc(sizeof(TipoDato));

  assegna l'indirizzo restituito dalla funzione malloc a p che punta quindi alla nuova variabile (p già esiste).
- Una variabile creata dinamicamente è necessariamente anonima: a essa si può fare riferimento solo tramite un puntatore

  a differenza di una variabile dichiarata mediante un proprio identificatore, che può essere riferita sia direttamente sia tramite un puntatore
- Sul blocco di memoria allocato è possibile usare i puntatori con l'usuale aritmetica.

Puntatore `NULL`

- Quando fallisce la richiesta di allocazione dinamica della memoria per mancanza di spazio, la funzione restituisce un puntatore nullo, ovvero un puntatore che non punta a nulla, rappresentato da `NULL`, con valore 0, dichiarato in `stdlib.h`.
- NULL ha valore falso (vale 0), mentre ogni puntatore non nullo ha valore vero (e diverso da NULL, ovvero da 0)

```
if (p == NULL) ...
if(!p) ...
```

Allocazione dinamica di array

- Posso creare dinamicamente un array di interi la cui dimensione `N` viene immessa da tastiera? Lo si può fare nel modo seguente
-

```
ptr = malloc(N*sizeof(int));
```

- Un array allocato dinamicamente può poi essere trattato come un array creato staticamente.

calloc

- Che assunzioni possiamo fare sul contenuto della memoria appena allocata con `malloc`?
- Nessuna, come nel caso della dichiarazione di una variabile non inizializzata. Però, esiste la funzione

```
calloc(size_t num, size_t size);
```

- Alloca dinamicamente `num` elementi di dimensione `size` (in bytes) e li inizializza a 0, ad esempio con

```
int * ptr = (int*) calloc(N,sizeof(int));
```

  si allocano `N` interi e li si inizializza a `0`.
- Restituisce `NULL` in caso di fallimento dell'allocazione

realloc

- Che succede se ci si accorge che l'array allocato è troppo piccolo e lo si vuole ingrandire dinamicamente senza perdere le informazioni memorizzate?

```
realloc(void *ptr, size_t size);
```

- `ptr` è un puntatore a un'area di memoria precedentemente allocata (altrimenti il comportamento e indefinito)
- `size` è la nuova dimensione (in bytes) dell'area di memoria
- Restituisce il puntatore al primo elemento dell'array ridimensionato o `NULL`, che potrebbe essere diverso se il blocco è stato spostato.
- Attenzione: Come fa realloc a conoscere la dimensione dei dati originali? Lo sa perché malloc ha registrato questa informazione quando è stato chiamato. Il sistema deve infatti tenere traccia delle dimensioni dei blocchi di memoria allocati, per evitare di allocare due volte una particolare regione di memoria.

free

- In C, la gestione dello heap e la deallocazione dello spazio è lasciata al programmatore.
- Se p è l'indirizzo di una variabile allocata dinamicamente, la chiamata

  free(p);

  rilascia lo spazio di memoria puntato da p

  > la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.

- free deve ricevere come parametro attuale (per valore) un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (come malloc), altrimenti il comportamentoè indefinito.
- Attenzione: La chiamata `free(p)` non può modificare il valore del puntatore, che quindi non diventa `NULL`, ma continua a puntare all'indirizzo della zona precedentemente allocata. Accedere tramite `p` (il valore del puntatore p non cambia) a questa zona, adesso libera e riallocabile, è possibile ma scorretto.

Heap

- Poiché le variabili dinamiche possono essere create e distrutte in un qualsiasi punto del programma esse non possono essere allocate sullo stack.
- Vengono allocate in un'altra zona di memoria chiamata heap (mucchio). La loro gestione risulta molto più inefficiente.

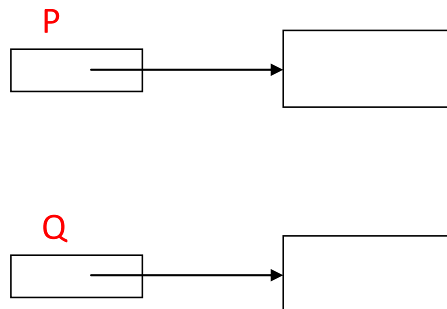
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento a essa.

  Esempio:

  ```
  P=malloc(sizeof(TipoDato));
  ...
  P=Q;
  ```

- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).

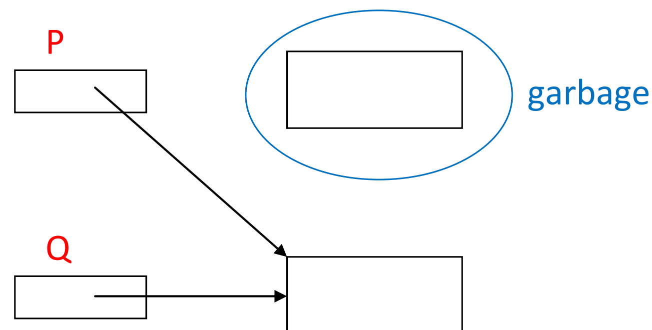


Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento a essa.

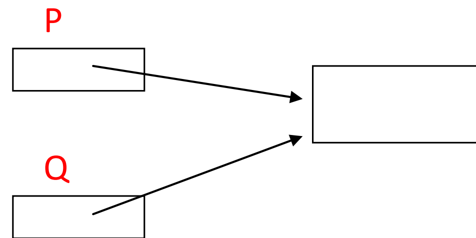
  Esempio:

  ```
  P=malloc(sizeof(TipoDato));
  ...
  P=Q;
  ```
- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



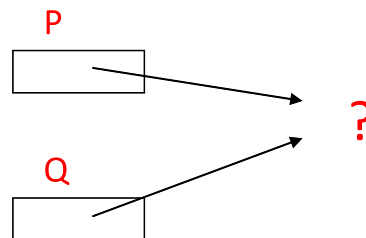
Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



Riferimenti fluttuanti (dangling references) (cont.)

- È possibile che il blocco di memoria ridimensionato da una `realloc` venga allocato in una differente posizione dello heap. Il contenuto della precedente area di memoria viene quindi spostato dalla `realloc`. Però

  Esempio:

```
 int *ptr1 = (int*) malloc(5 * sizeof(int));
int *ptr2 ;
....
ptr2 = (int*) realloc(ptr1,10*sizeof(int));
```

  Il puntatore `ptr1` che puntava all'area di memoria prima della chiamata alla `realloc` dopo potrebbe puntare a un'area non più valida.

- Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - la prima comporta spreco di memoria
  - la seconda comporta risultati imprevedibili e scorretti.
- La seconda è più pericolosa della prima e in alcuni linguaggi non è prevista l'istruzione `free`.
- Viene lasciato al supporto del linguaggio l'onere di effettuare garbage collection ("raccolta rifiuti").

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

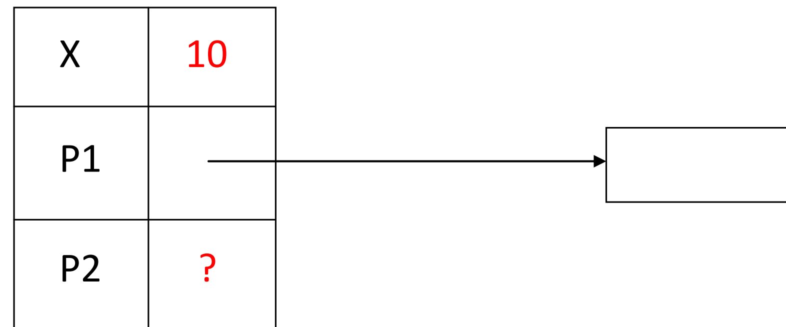
PILA | HEAP

X	10
P1	?
P2	?

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA

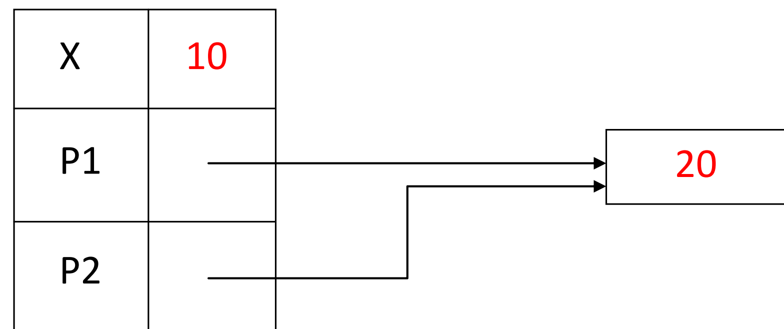
HEAP

X	10
P1	→ 20
P2	?

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

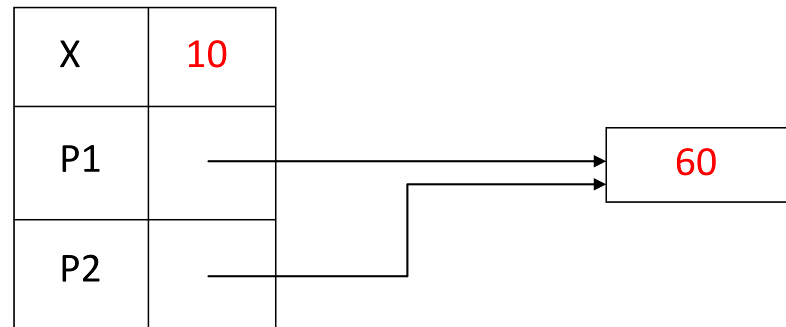
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

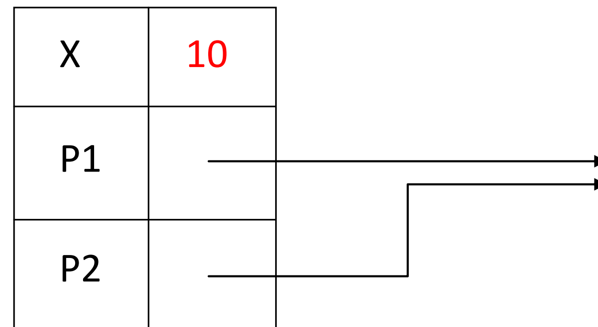
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP



Liste

- È molto comune dover rappresentare sequenze di elementi tutti dello stesso tipo e fare operazioni su di esse.

  Esempi: sequenza di interi (23 46 5 28 3)
  sequenza di caratteri ('x' 'r' 'f')
  sequenza di persone con nome e data di nascita
- Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.
- Vediamo adesso un modo basato sull'allocazione dinamica di variabili, che ci permette di realizzare liste di elementi in maniera che la memoria fisica utilizzata corrisponda meglio a quella astratta, cioè al numero di elementi della sequenza che vogliamo rappresentare.

Diversi modi di rappresentare sequenze di elementi

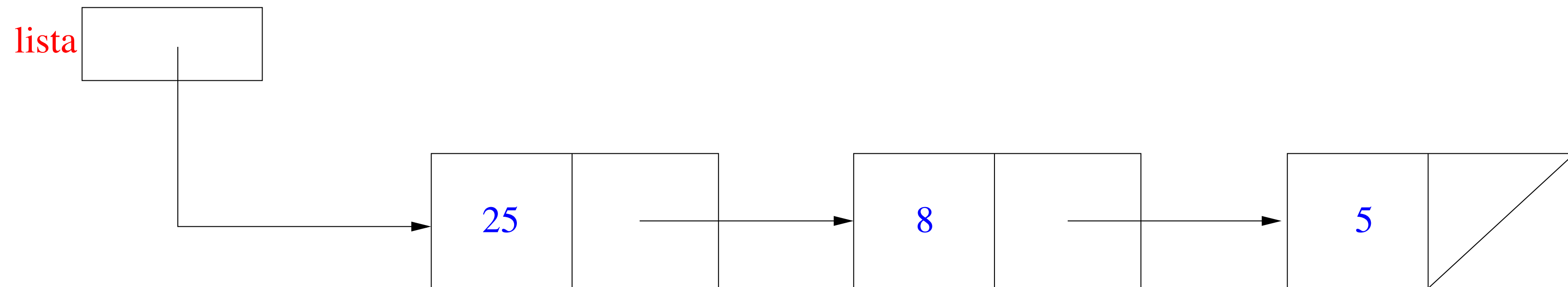
1. Rappresentazione sequenziale: tramite array
   - Vantaggi:
     - l'accesso agli elementi è diretto (tramite indice) ed efficiente
     - l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali
     - è semplice manipolare l'intera struttura (copia, ordinamento, ...)
   - Svantaggi:
     - dobbiamo avere un'idea precisa della dimensione della sequenza
     - inserire o eliminare elementi è complicato e inefficiente (comporta un numero di spostamenti che nel caso peggiore può essere dell'ordine del numero degli elementi della struttura)

2. Rappresentazione collegata
    - Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
    - Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.
    - L'accesso agli elementi è di tipo sequenziale: per accedere al generico nodo, si deve scandire la lista, dato che l'accesso ad un elemento è possibile attraverso il puntatore contenuto nell'elemento precedente.

2. Rappresentazione collegata (continua)

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite strutture e puntatori.
- Ogni elemento è rappresentato con una struttura C:
  - un campo (o più campi se necessario) per l'elemento (ad es. `int`)
  - un campo puntatore alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo identico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo
  - il campo puntatore ha valore `NULL` che assume quindi il significato di "fine lista".
- L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
  - Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di "inizio lista" (o "testa della lista") con la lista stessa.
- l'accesso a una lista avviene attraverso il puntatore al primo elemento.

Graficamente



- La variabile lista, di tipo puntatore, è utilizzata per accedere alla sequenza.

Definizione ricorsiva di lista

- Possiamo definire ricorsivamente una lista come segue.
- Una lista è una struttura definita su un insieme di elementi che:
  - non contiene nessun elemento, ovvero è una lista vuota: $[\ ]$, oppure
  - contiene un elemento EL detto *testa* (head) della lista) seguito dal resto della lista L, detta *coda*: $[EL, L]$
- La definizione usata dal C riflette proprio questa definizione. Una variabile di tipo lista può valere NULL (che rappresenta la lista vuota), oppure può essere un puntatore a una struttura che contiene un dato più un puntatore, che rappresenta un'altra lista.

Esempio: Sequenze di interi.

```
struct EL {
  int info;
  struct EL *next;
};
typedef struct EL ElementoLista;
typedef ElementoLista *ListaDiElementi;
```

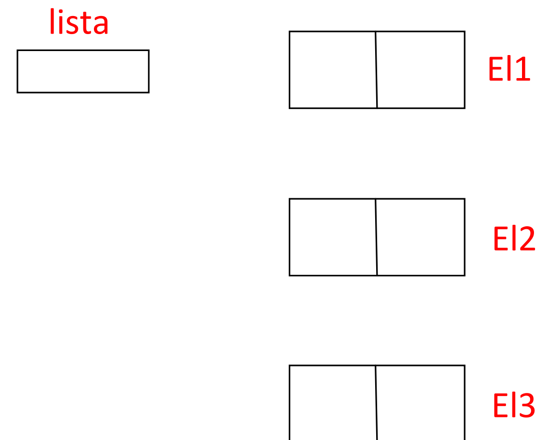
1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`, che ne diventa l'abbreviazione;
3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.

- A questo punto possiamo definire variabili di tipo lista:

```
ListaDiElementi Lista1, Lista2;
```

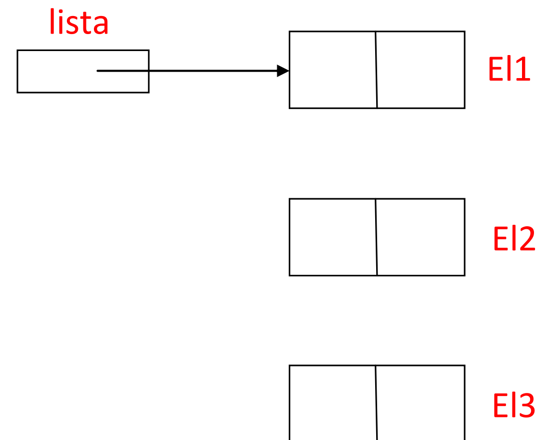
Creazione di una lista di tre interi fissati: (8, 3, 15) [allocazione statica]

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



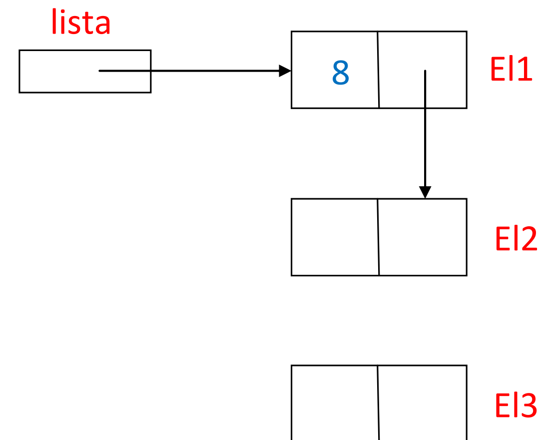
Creazione di una lista di tre interi fissati: (8, 3, 15) [allocazione statica]

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



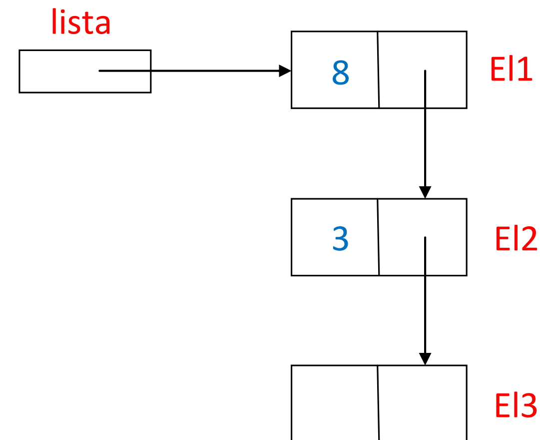
Creazione di una lista di tre interi fissati: (8, 3, 15) [allocazione statica]

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



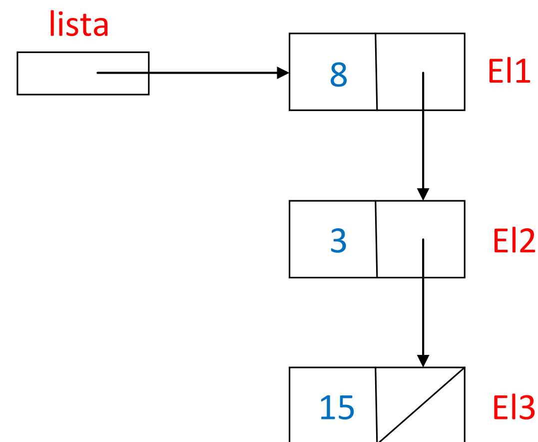
Creazione di una lista di tre interi fissati: (8, 3, 15) [allocazione statica]

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15) [allocazione statica]

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Aliasing

- Si parla di aliasing quando si utilizzano due puntatori (alias) per far riferimento allo stesso valore.
- Se si modifica il valore puntato da uno dei due, implicitamente (come effetto collaterale) si modifica anche il valore puntato dall'altro, essendo lo stesso.
- Questo è un fenomeno particolarmente rilevante quando si manipolano liste.
- Nell'esempio visto prima, se avessi:

  ```
  lista2=&El2;
  lista2 --> info = 9
  ```

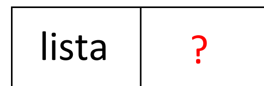
  allora avrei anche che la condizione
  `((El1-->next)-->info == 9)` sarebbe vera
  Ricordiamo che `lista2 --> info` equivale a `(*lista2).info`

- Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?
- Quello che abbiamo visto non è l'unico modo. Possiamo ricorrere all'allocazione dinamica della memoria.

Creazione di una lista di tre interi fissati (8, 3, 15) [allocazione dinamica]

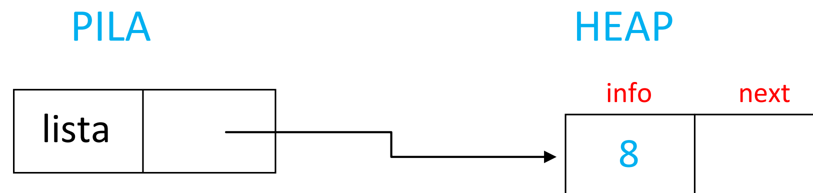
```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

PILA HEAP



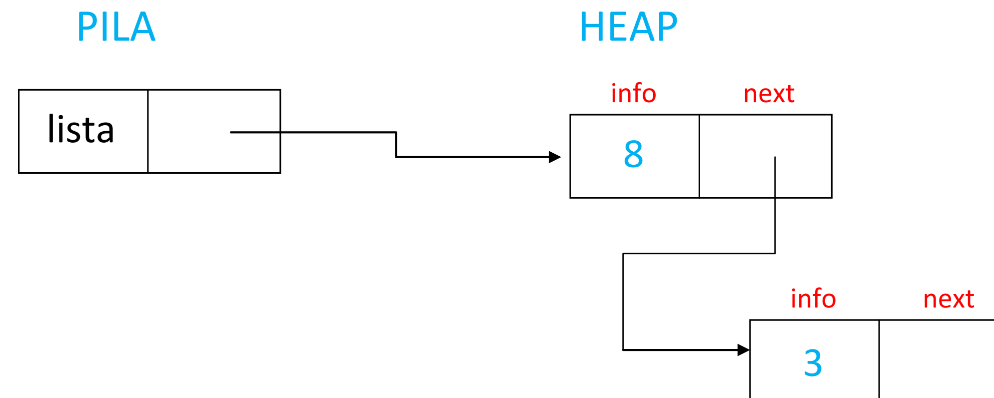
Creazione di una lista di tre interi fissati (8, 3, 15) [allocazione dinamica]

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



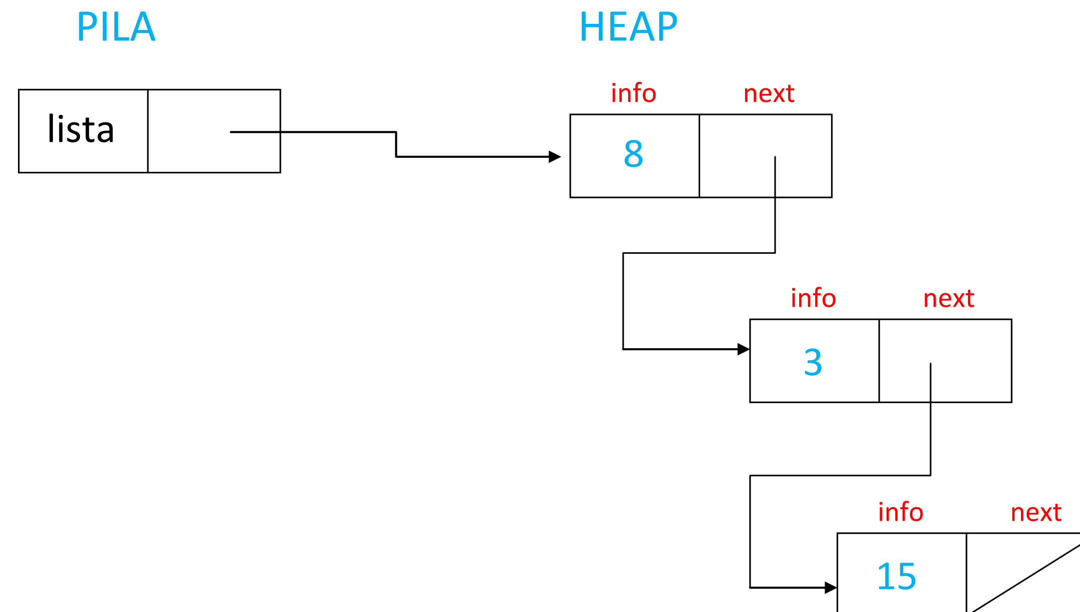
Creazione di una lista di tre interi fissati (8, 3, 15) [allocazione dinamica]

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati (8, 3, 15) [allocazione dinamica]

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Osservazioni:

```
struct EL {
  int info;
  struct EL *next;
};
typedef struct EL ElementoLista;
typedef ElementoLista *ListaDiElementi;
```

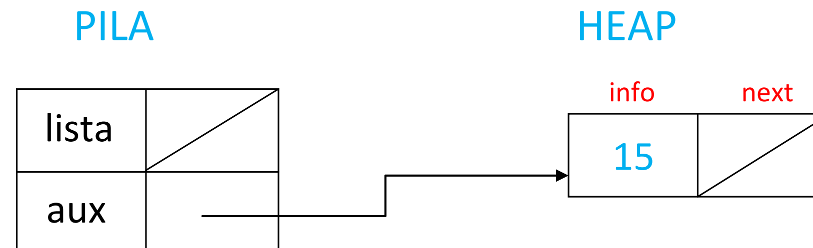
- `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e non una struttura, così come, in **int** `*p`, `p` è un puntatore a intero e non un intero.
- la zona di memoria per ogni elemento della lista (non per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- Esiste un modo più semplice di creare la lista di 3 elementi?
- Creiamo la lista a partire dal fondo!

```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

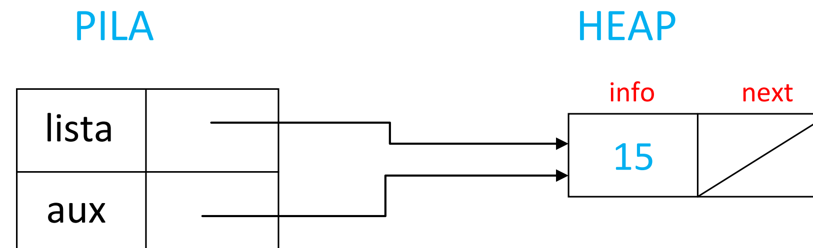
PILA HEAP

lista	/
aux	?

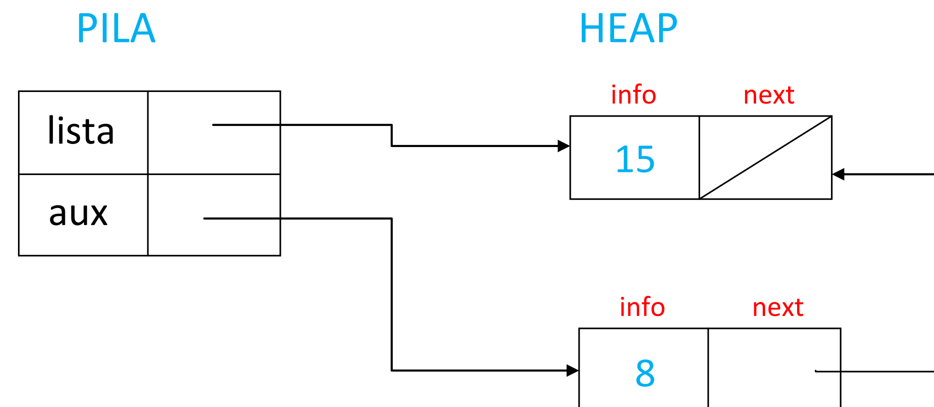
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



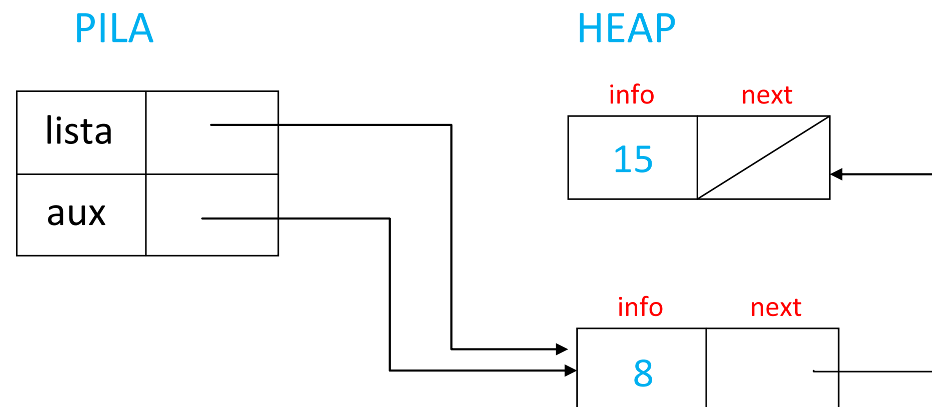
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;    aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;     aux->next = lista;
lista = aux;
```



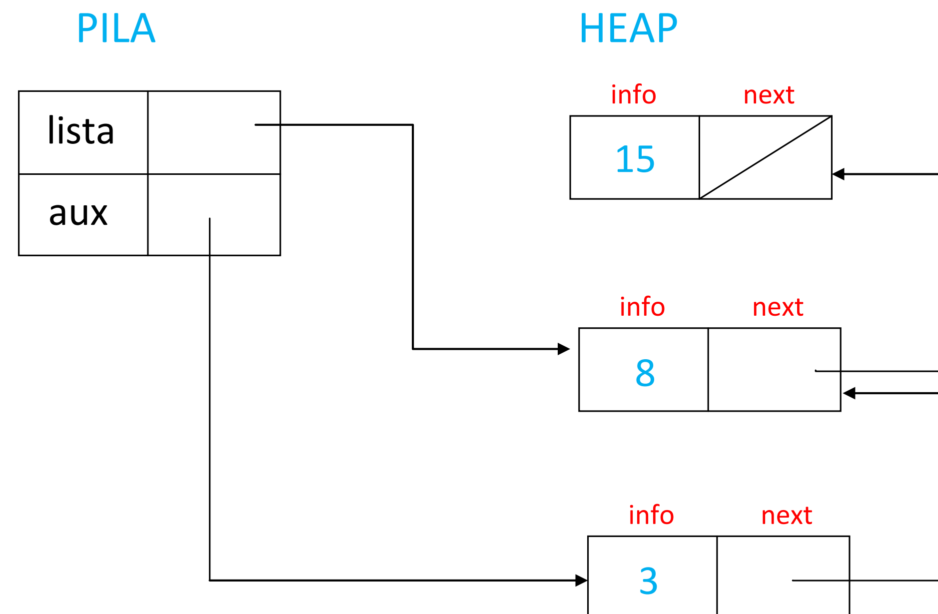
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



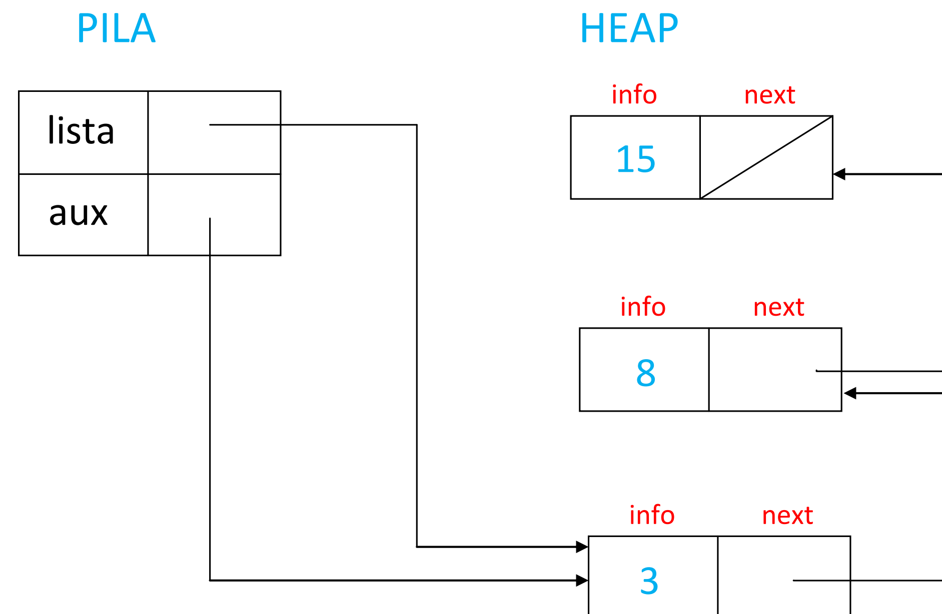
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;    aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;     aux->next = lista;
lista = aux;
```



Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.
- Usiamo liste di interi per semplicità, ma tutte le operazioni sono realizzabili in modo del tutto analogo su liste di altro tipo (salvo rare eccezioni)
- Facciamo riferimento alle dichiarazioni dei tipi `ElementoLista` e `ListaDiElementi` viste in precedenza

Inizializzazione

- Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile testa della lista.
- Tale variabile deve essere modificata e quindi passata per indirizzo.
- Ciò provoca, nell'intestazione della procedura, la presenza di un puntatore a puntatore.

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

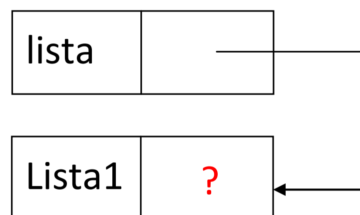
PILA

Lista1	?

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

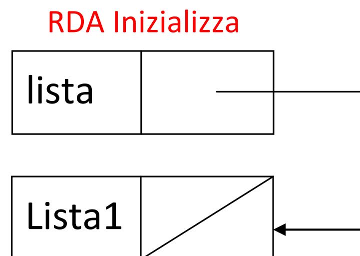
```
ListaDiElementi Lista1;

Inizializza(&Lista1);
```



```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;

Inizializza(&Lista1);
```

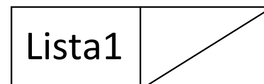


```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;

Inizializza(&Lista1);
```

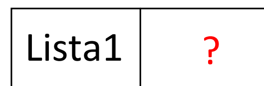
PILA



Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA



Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

RDA Inizializza

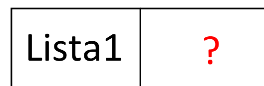
lista	?
Lista1	?

Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

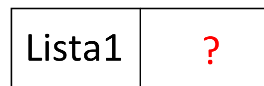
RDA Inizializza



Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA



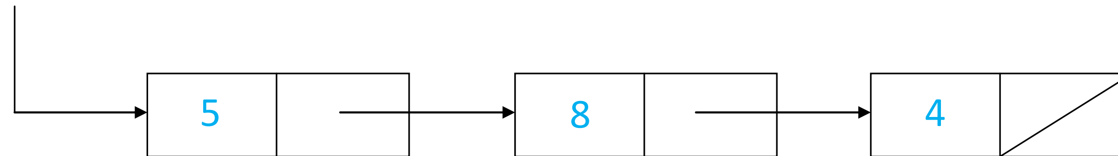
Controllo lista vuota

```
boolean ListaVuota(ListaDiElementi lista)
{
 return (lista==NULL);
}
```

A `lista` viene passato il valore contenuto nella variabile testa di lista e quindi punta al primo elemento della lista considerata.

Stampa degli elementi di una lista

- Data la lista



vogliamo che venga stampato:

5 -> 8 -> 4 -> //

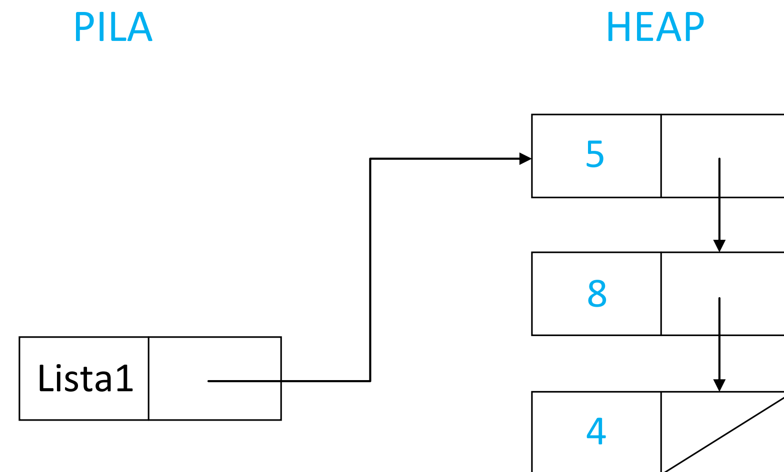
Versione iterativa:

```
void StampaLista(ListaDiElementi lis)
{
   while (lis != NULL)
    {
     printf("%d -->", lis->info);
     lis = lis->next;
    }
printf("//");
}
```

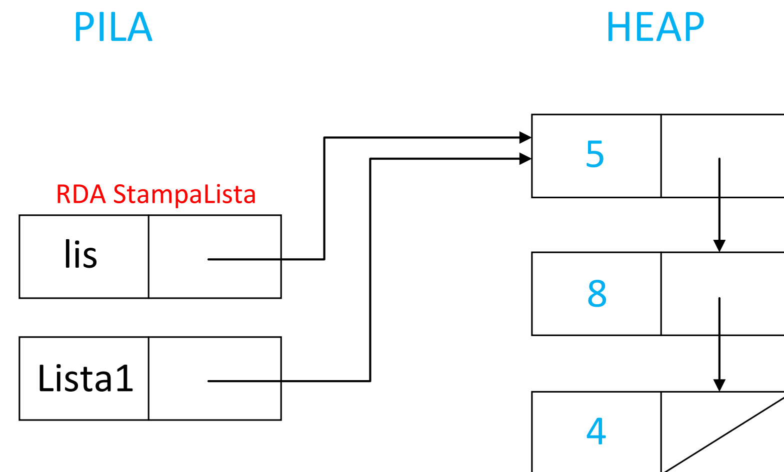
- `lis = lis->next` fa puntare `lis` all'elemento successivo della lista
- `(lis != NULL)` permette di scorrere fino alla fine della lista, di cui solitamente non sappiamo la lunghezza.
- Attenzione: Possiamo usare `lis` per scorrere la lista perché, avendo utilizzato il passaggio per valore, le modifiche a `lis` non si ripercuotono sul parametro attuale.

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```

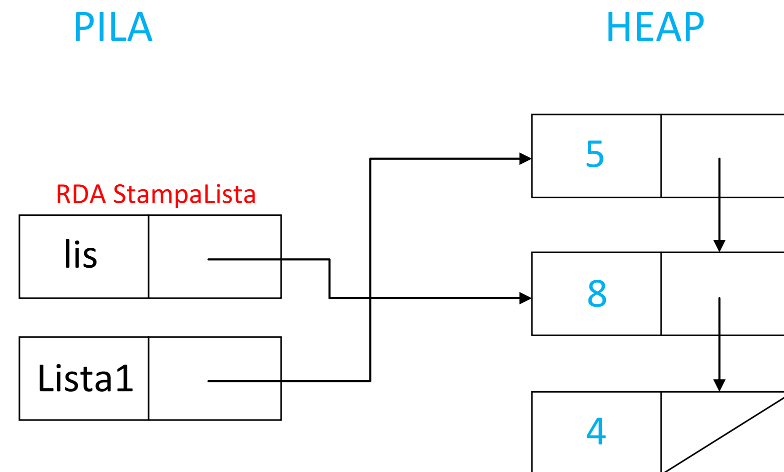
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



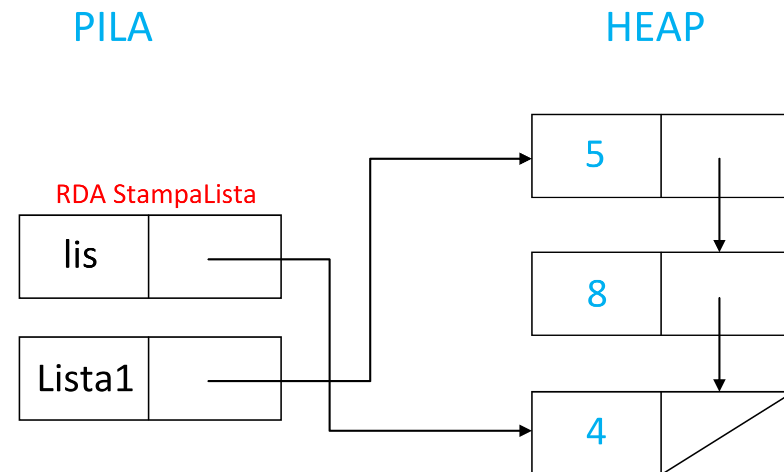
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



Output

5 -->

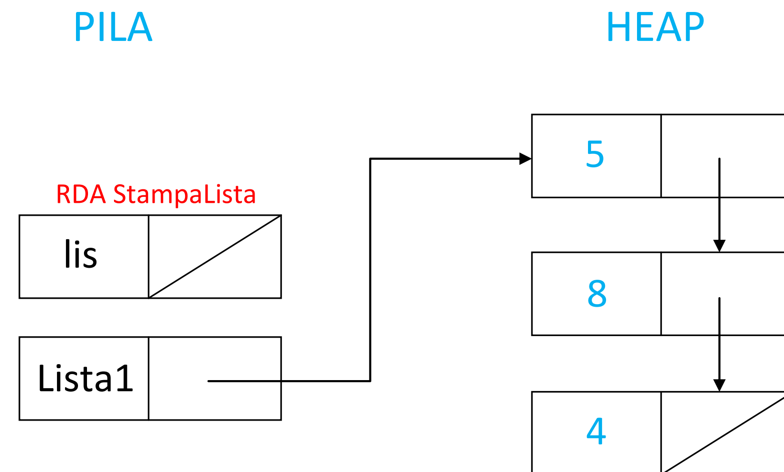
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



Output

5 --> 8 -->

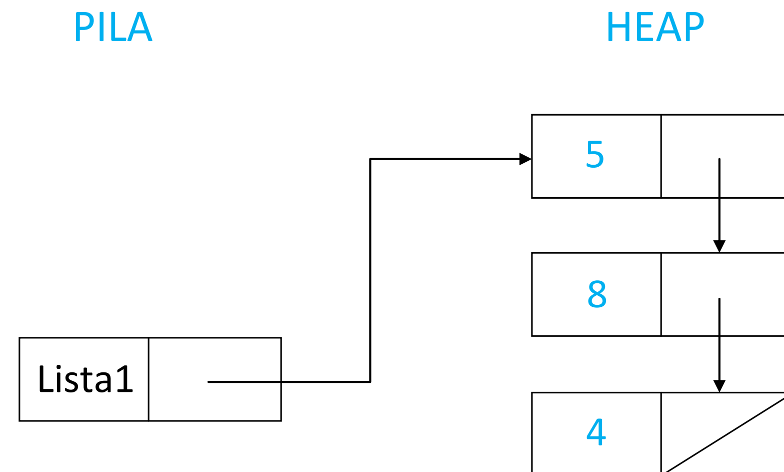
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



Output

5 --> 8 --> 4 --> //

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



Output

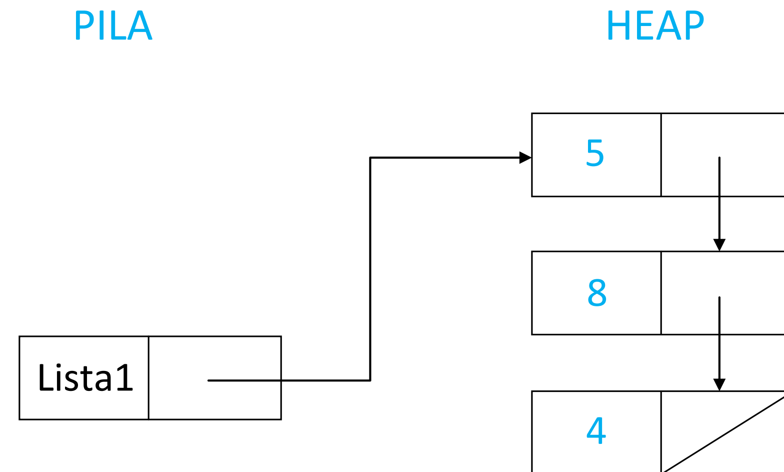
5 --> 8 --> 4 --> //

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

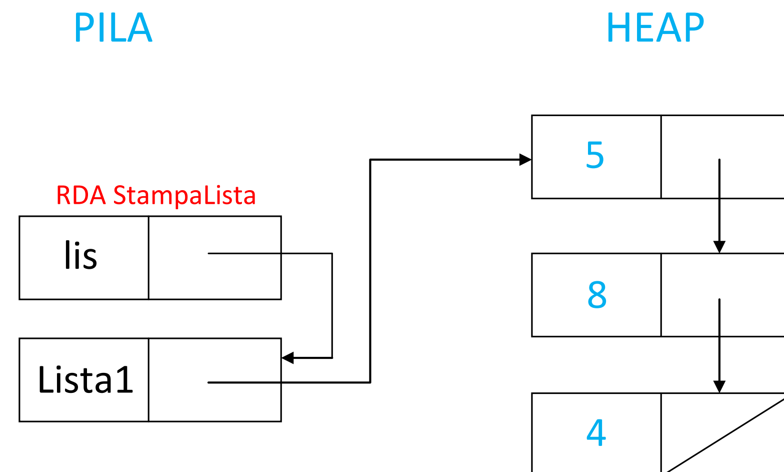
Cosa sarebbe successo passando il parametro per indirizzo?

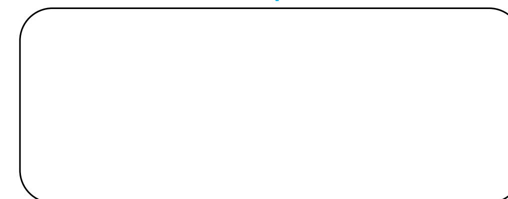
```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

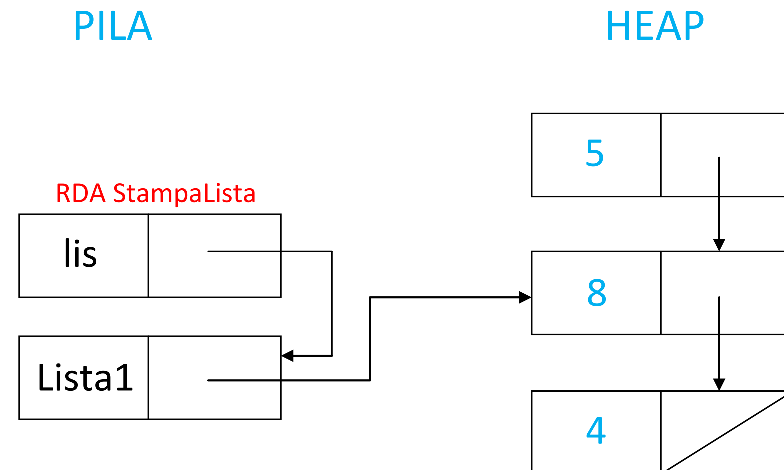
```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```





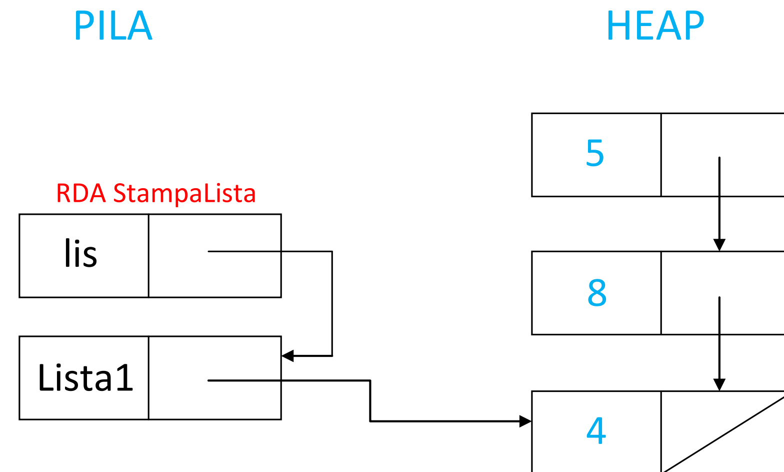
Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

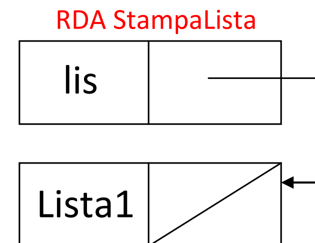


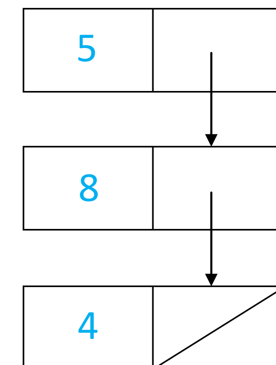
Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

PILA

HEAP





Output

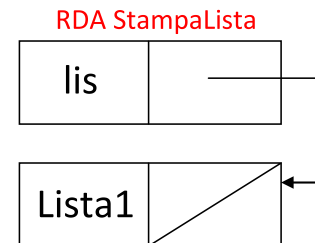
5 --> 8 --> 4 -->

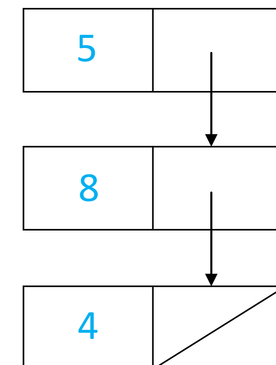
Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

PILA

HEAP





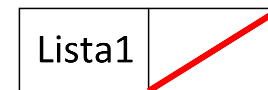
Output

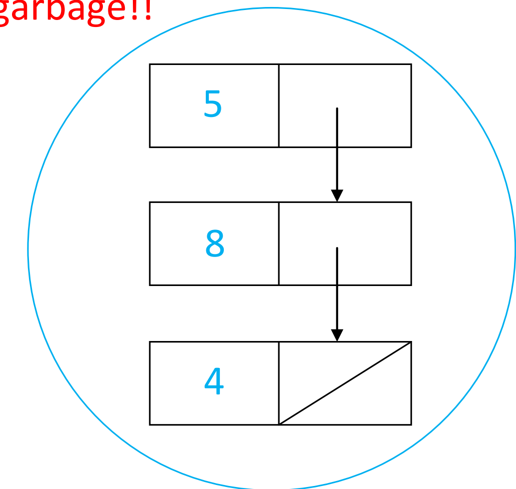
5 --> 8 --> 4 --> //

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

PILA





Output

5 --> 8 --> 4 --> //

Versione ricorsiva

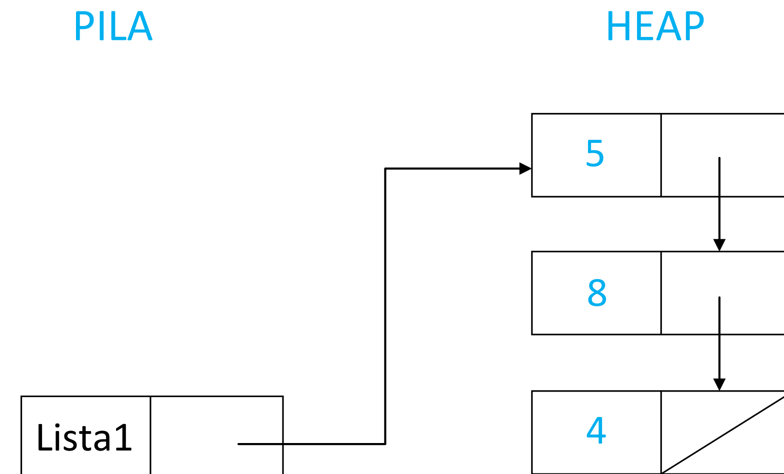
Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

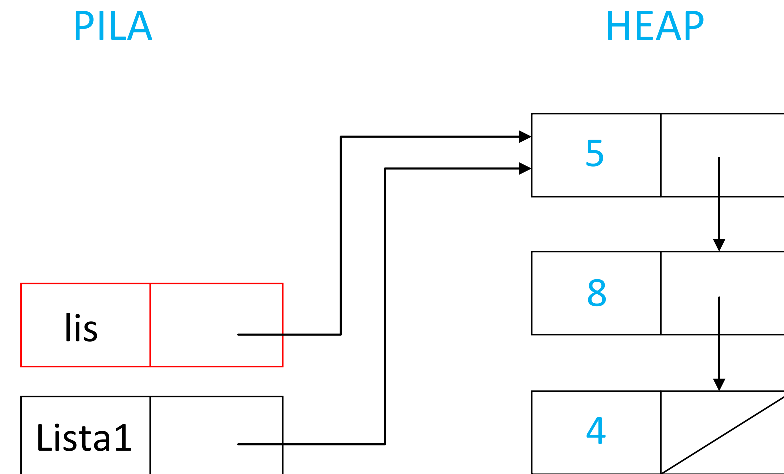
```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

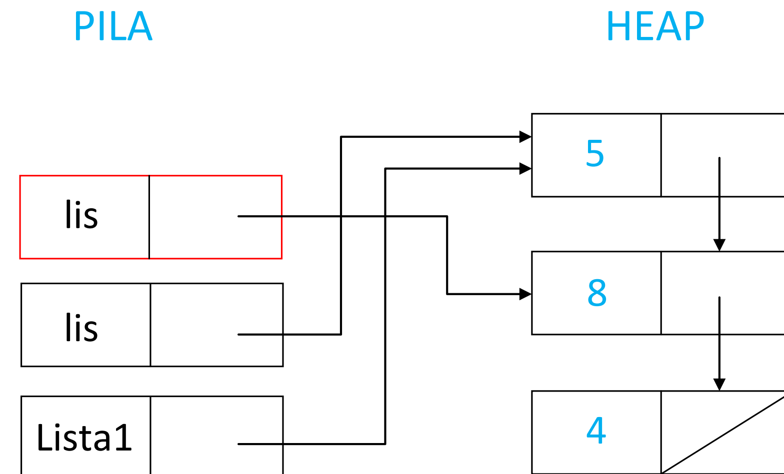
```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



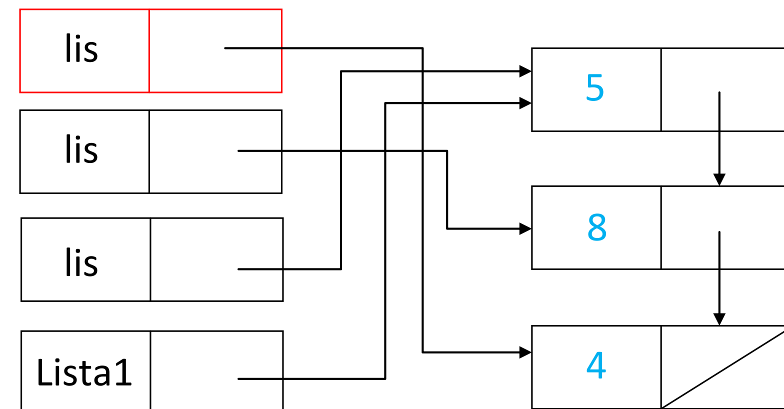
Output

5 -->

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



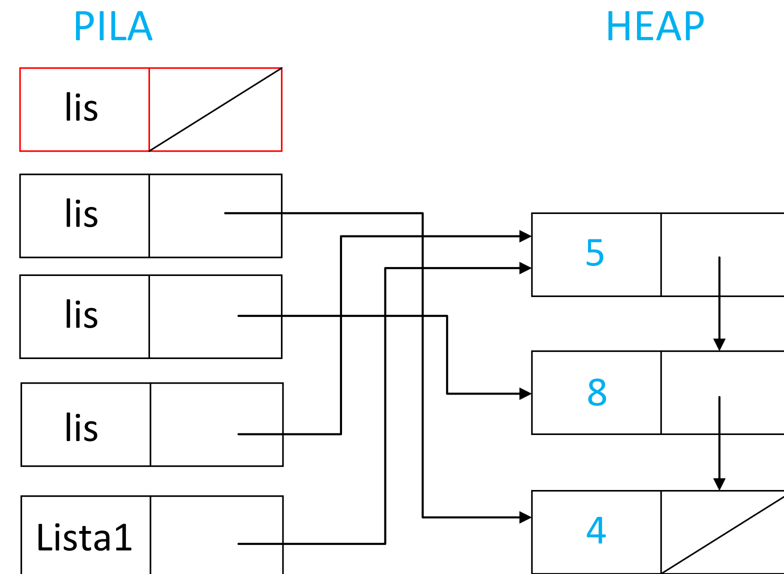
Output

5 --> 8 -->

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

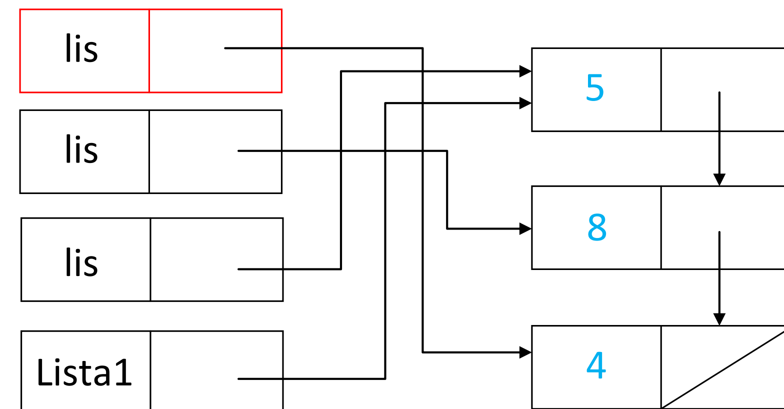
```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



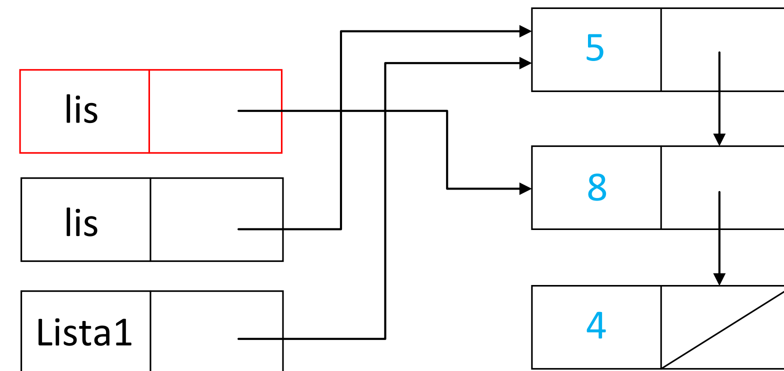
Output

5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



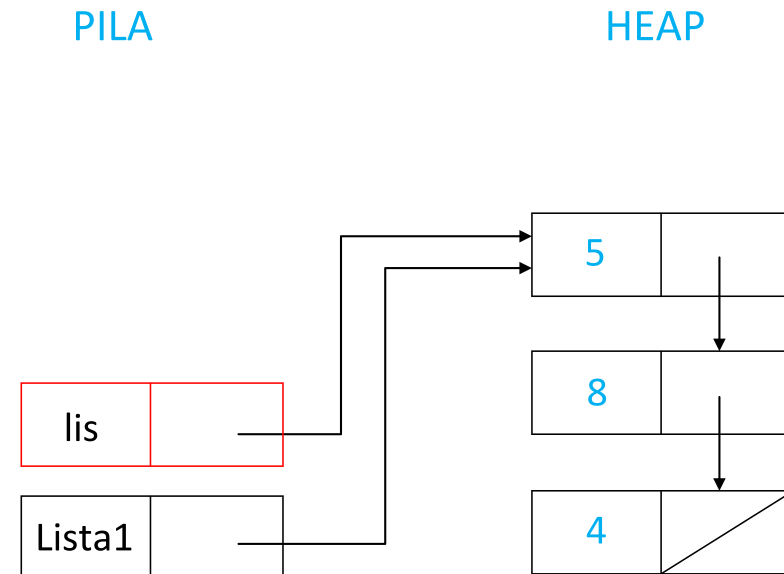
Output

5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente [passaggio per valore]

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```