# Lezione 22 - ADT LISTA

## Rappresentazione concreta del Tipo di Dati Astratto LISTA.

- **Cenni sulla rappresentazione mediante array** (puah!)
- **Rappresentazione concreta mediante struct e puntatori.**
  - **concetto della rappresentazione e tipi C**
  - **lista vuota**
  - **inizializzazione**
  - **allocazione di nodi in lista**
  - **chiusura lista (?!)**
  - **alcune funzioni: testListaVuota(), initLista(), testaLista()**
  - **implementazione di cdr() con un *approccio distruttivo*: cancellaPrimo()**
- **scansione di una list**
  - **stampa**
  - **ricerca in lista**
- **inserimento in lista**
  - **inserimento in testa**
  - **costruzione di una lista di n nodi con ins. in testa**
  - **tecnica di aggiunta di un nodo in coda alla lista**
  - **costruzione di una lista di n nodi con ins. in coda**
- **gestione di una lista con la tecnica del *record generatore***
  - **costruzione di una lista con ins. in coda...**

# What's LISTE?

Una lista è una

**SEQUENZA** di **ELEMENTI** appartenenti ad un insieme E
(l'ordine della sequenza è "significativo", cioè esiste una relazione di *predecessore / successore* tra elementi vicini nella sequenza)

es. $\lambda = (4, 9, 1)$

4 è il predecessore (o *predecessore diretto*) di 9;
1 è il successore (… *diretto*) di 9

a volte si usa una nozione estesa di precedenza: 4 precede1 … si intende *precedenza anche non diretta* … e analogamente per la proprietà di essere un successore)

**ADT**

E = dominio degli elementi di lista (persone, voli, interi, lettere ...)
L = {λ / λ è una lista di elementi in E}
Bool = {TRUE, FALSE}

Un operatore tradizionale su liste …

$$\text{cons: } L \times E \longrightarrow L$$
$$(\ldots), e \longmapsto (e, \ldots)$$

"…"= 0 o lista con + el.

**ES.**

`cons(λ, 2)` è `(2, 4, 9, 1)`
(un'altra lista)

# ADT LISTA

Una lista è una **SEQUENZA** di **ELEMENTI** appartenenti ad un insieme
(l'ordine della sequenza è "significativo", cioè esiste una relazione di *predecessore / successore* tra elementi

es. λ = (4, 9, 1)

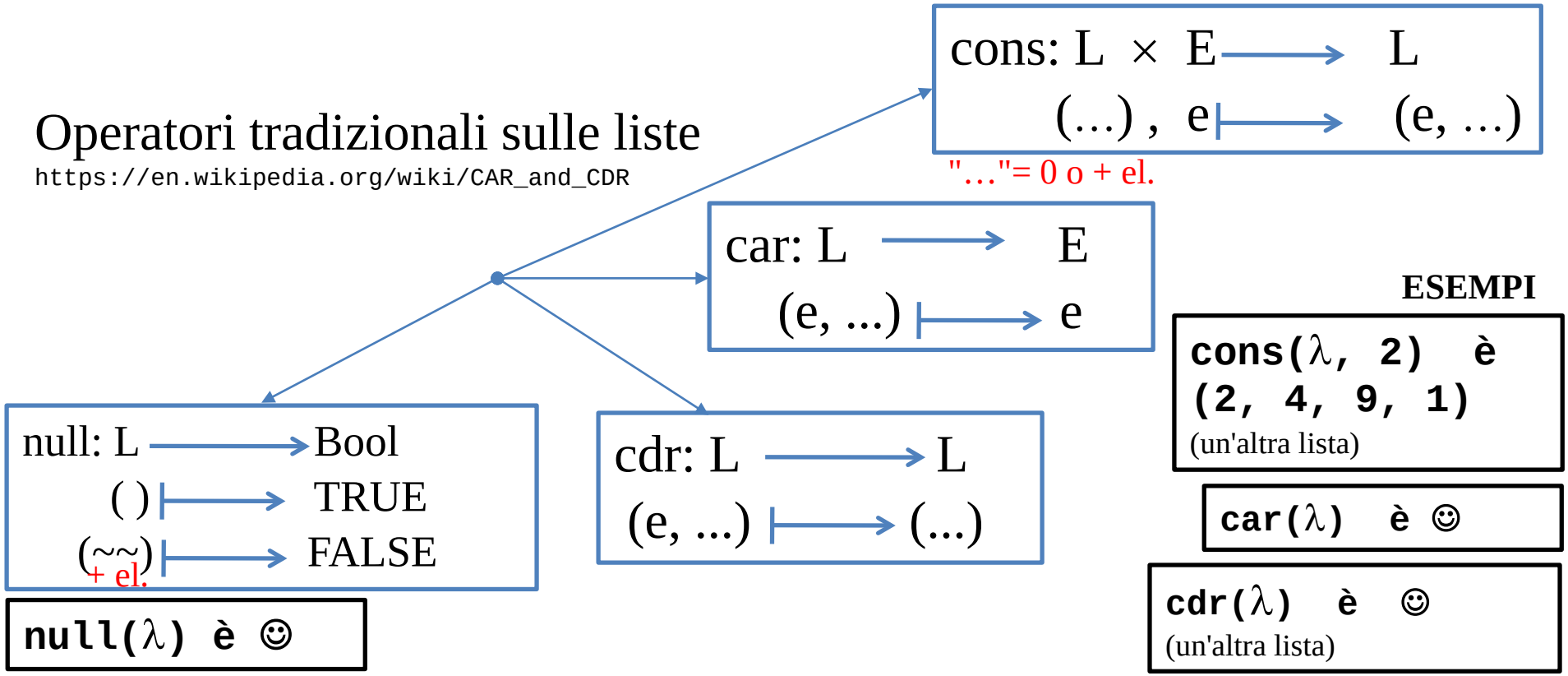
E = dominio degli elementi di lista (persone, voli, interi, lettere ...)

L = {λ / λ è una lista di elementi in E}

Bool = {TRUE, FALSE}

## Operatori tradizionali sulle liste

https://en.wikipedia.org/wiki/CAR_and_CDR




**ESEMPI**

**cons(λ, 2) è (2, 4, 9, 1)**
(un'altra lista)

**car(λ) è ☺**

**cdr(λ) è ☺**
(un'altra lista)

**null(λ) è ☺**

# ADT LISTA

Una lista è una **SEQUENZA** di **ELEMENTI** appartenenti ad un insieme
(l'ordine della sequenza è "significativo", cioè esiste una relazione di *predecessore / successore* tra elementi

es. λ = (4, 9, 1)

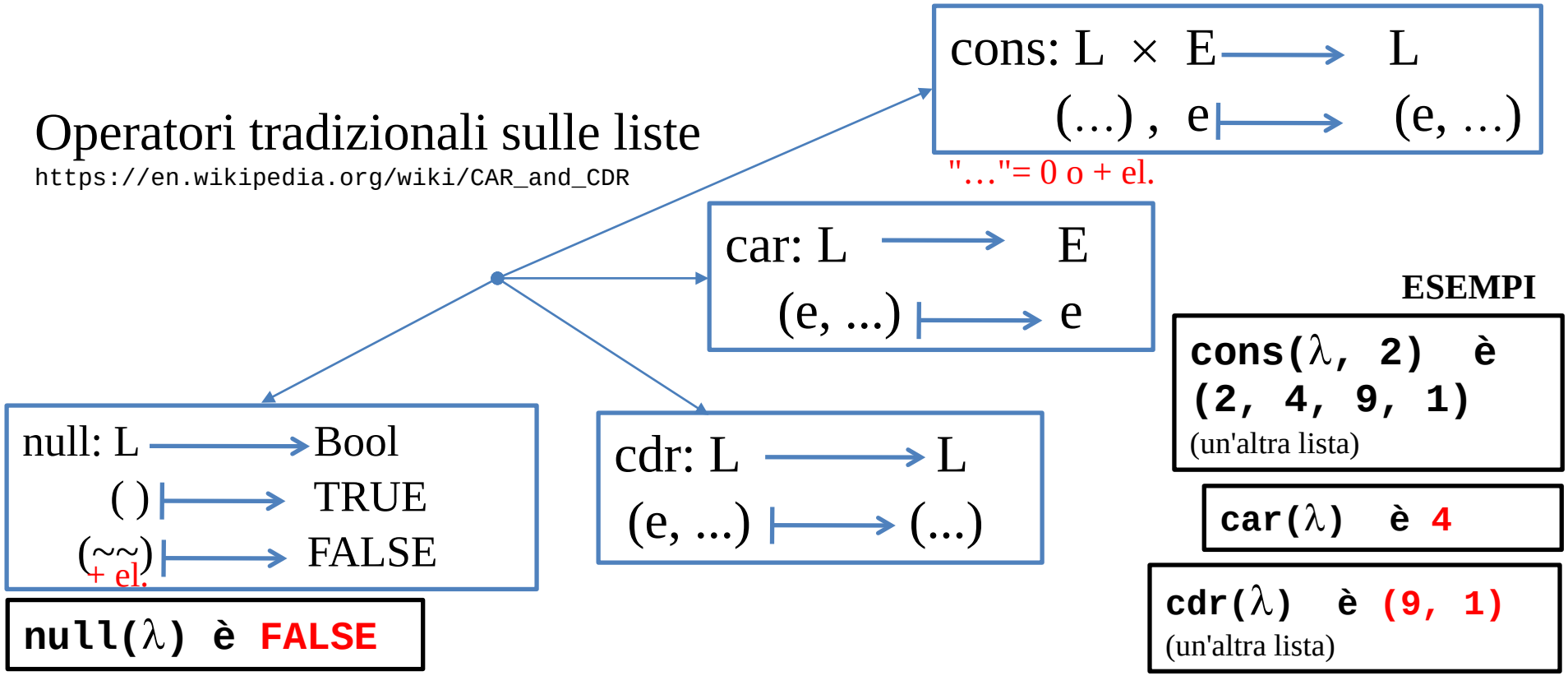
E = dominio degli elementi di lista (persone, voli, interi, lettere ...)

L = {λ / λ è una lista di elementi in E}

Bool = {TRUE, FALSE}

## Operatori tradizionali sulle liste

https://en.wikipedia.org/wiki/CAR_and_CDR




**ESEMPI**

**cons(λ, 2) è (2, 4, 9, 1)**
(un'altra lista)

**car(λ) è 4**

**cdr(λ) è (9, 1)**
(un'altra lista)

**null(λ) è FALSE**

# Rappresentazione concreta di LISTE

DOMINI ---> tipi concreti;

L ---> `TipoLista`

E ---> `TipoElem`
(TipoVolo,
int,
TipoPersona,
…)

Bool ---> `Bool`
(approccio libro:
il tipo Bool è `int` con
`#define TRUE 1`
`#define FALSE 0`)

**OPERAZIONI** ---> funzioni

```
void insLista (TipoLista *plis,
               TipoElem e)
```

```
void testaLista (TipoLista lis,
                 TipoElem *pelem)
```

```
Bool testListaVuota (TipoLista
                                lis)
```

**NB** **insLista realizza cons "distruttivamente",**

**cioè MODIFICA la lista passata come parametro (tramite indirizzo)**

**NB** **testaLista realizza car e restituisce il primo elemento tramite parametro di output**

**NB** **testListaVuota realizza null**

**NB** **cdr non lo realizziamo esplicitamente (perché … in seguito)**

# Rappresentazione concreta di LISTE

DOMINI ---> tipi concreti;

L ---> `TipoLista`

E ---> `TipoElem`
(TipoVolo,
int,
TipoPersona,
…)

Bool ---> `Bool`
(approccio libro:
il tipo Bool è `int` con
`#define TRUE 1`
`#define FALSE 0`)

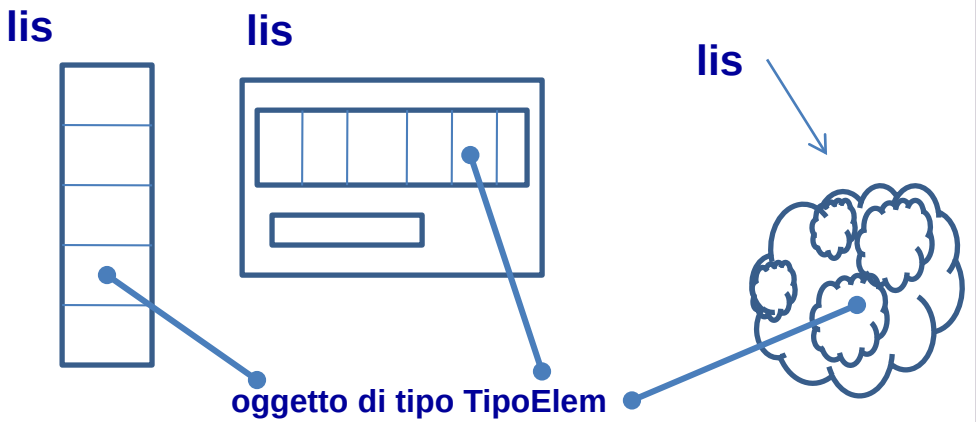
**OPERAZIONI** ---> funzioni

```
void insLista (TipoLista *plis,
               TipoElem e)
```

```
void testaLista (TipoLista lis,
                 TipoElem *pelem)
```

```
Bool testListaVuota (TipoLista
                                lis)
```

**RAPPRESENTAZIONE** in memoria
(valori di tipo TipoLista, cioe` *come e` fatta una variabile* ***`TipoLista lis`*** )



# Rappresentazione concreta di LISTE (rappresentazione mediante array)

DOMINI ---> tipi concreti;

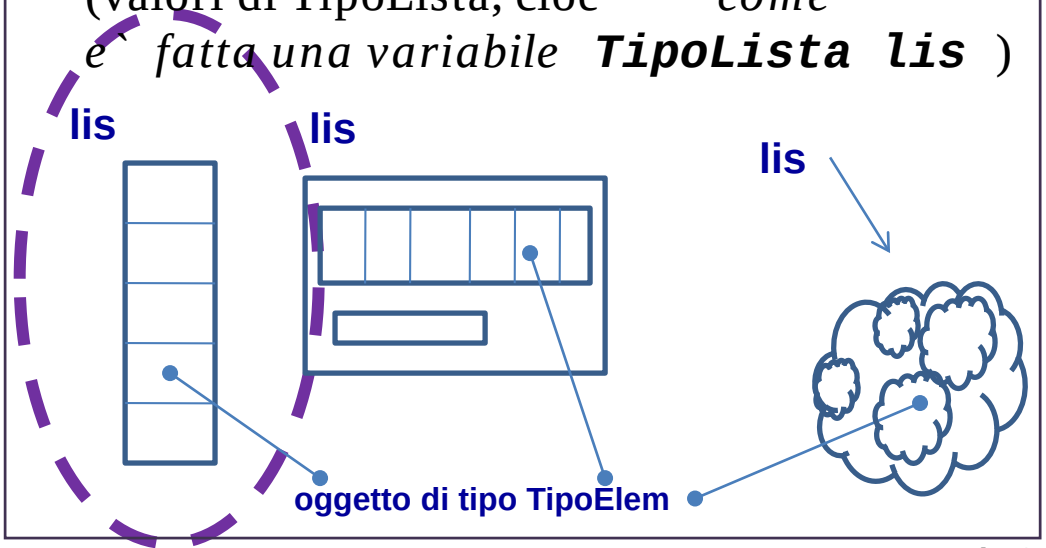
L ---> `TipoLista`

E ---> `TipoElem`
(TipoVolo,
int,
TipoPersona,
…)

Bool ---> `Bool`
(approccio libro:
il tipo Bool è`int` con
`#define TRUE 1`
`#define FALSE 0`)

```
TipoLista è
      TipoELem []

variabile lista
      TipoLista lis
```

**RAPPRESENTAZIONE** in memoria
(valori di TipoLista, cioe` *come e` fatta una variabile* ***TipoLista lis*** )



# Rappresentazione concreta di LISTE (rappresentazione mediante array: come tabella ...)

DOMINI ---> tipi concreti;

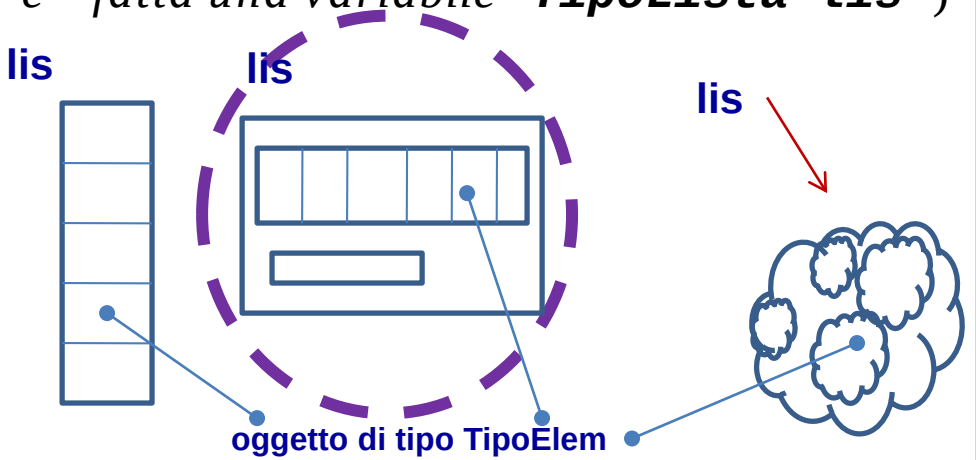
L ---> `TipoLista`

E ---> `TipoElem`
(TipoVolo,
int,
TipoPersona,
...)

Bool ---> `Bool`
(approccio libro:
il tipo Bool è`int` con
`#define TRUE 1`
`#define FALSE 0`)

```
TipoLista è
struct {
    TipoELem sostLista []
    int quantiElem;
}

variabile lista
     TipoLista lis
```

**RAPPRESENTAZIONE** in memoria
(valori di TipoLista, cioe` *come e` fatta una variabile* ***TipoLista lis*** )



# Rappresentazione di LISTA mediante array

supponendo di aver già definito **`typedef ... TipoElem`**

**`TipoLista`** va definito come un tipo "`array-di-TipoElem`"

e se definiamo una variabile **`TipoLista lis;`** questa sarà un array allocato staticamente o dinamicamente a seconda dei dettagli della definizione di **`TipoLista`**

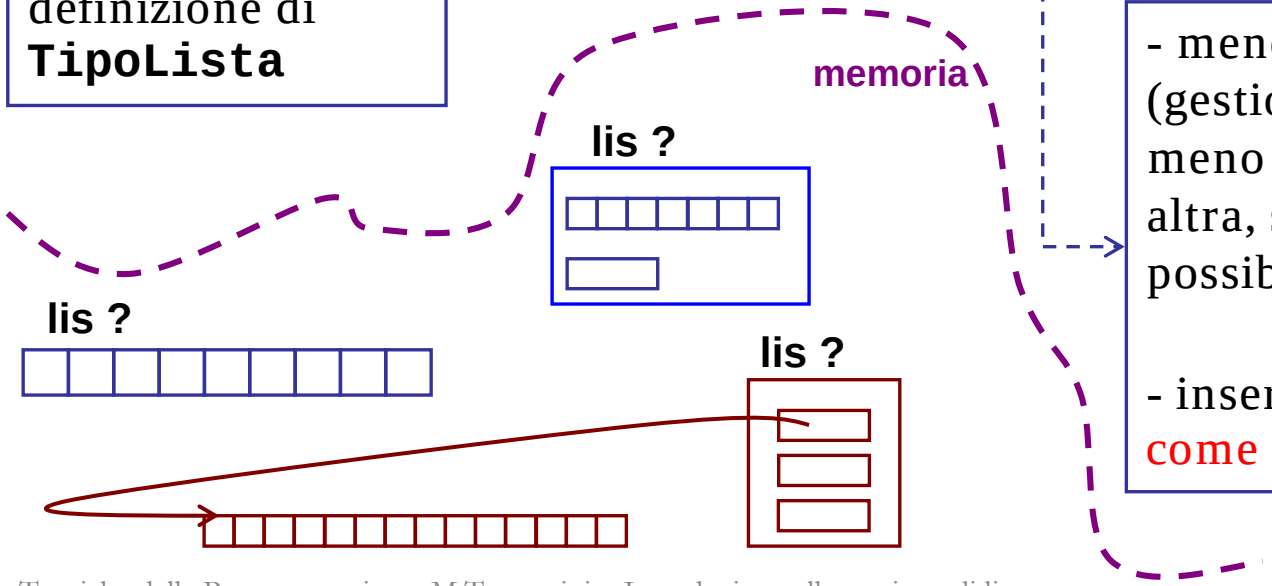
**Allocazione STATICA**

- memoria fissata in partenza (ad esempio con una costante MAXELEM che dice quanti elementi al massimo possono esistere contemporan. in lista)
- (inserimenti ed) eliminazioni con spostamenti (l'ordine conta ...)

**Allocazione DINAMICA**

- meno limitazioni sulla memoria (gestione più efficiente: si spreca meno memoria e quando ne serve altra, se ne alloca altra - se possibile)

- inserimenti ed eliminazioni come sopra



# Rappresentazione concreta di LISTE (terzo modo ...)

DOMINI ---> tipi concreti;

L ---> `TipoLista`

E ---> `TipoElem`
(TipoVolo,
int,
TipoPersona,
...)

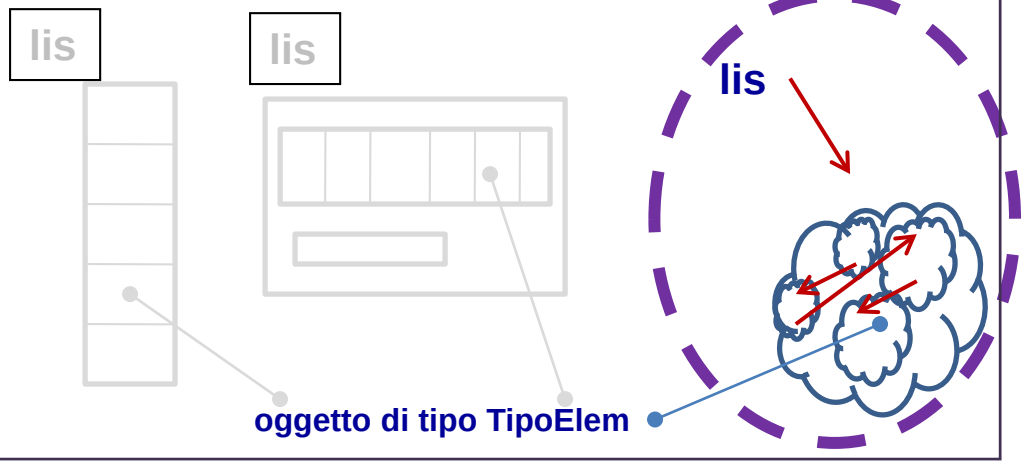
Bool ---> `Bool`
(approccio libro:
il tipo Bool è `int` con
`#define TRUE 1`
`#define FALSE 0`)

**`TipoElem è`**
**`+/- come prima …`**

**`TipoLista è`**
**`?`**

**`variabile lista`**
**`?`**

**RAPPRESENTAZIONE** in memoria
(valori di TipoLista, cioe` *come e` fatta una variabile* ***`TipoLista lis`*** )



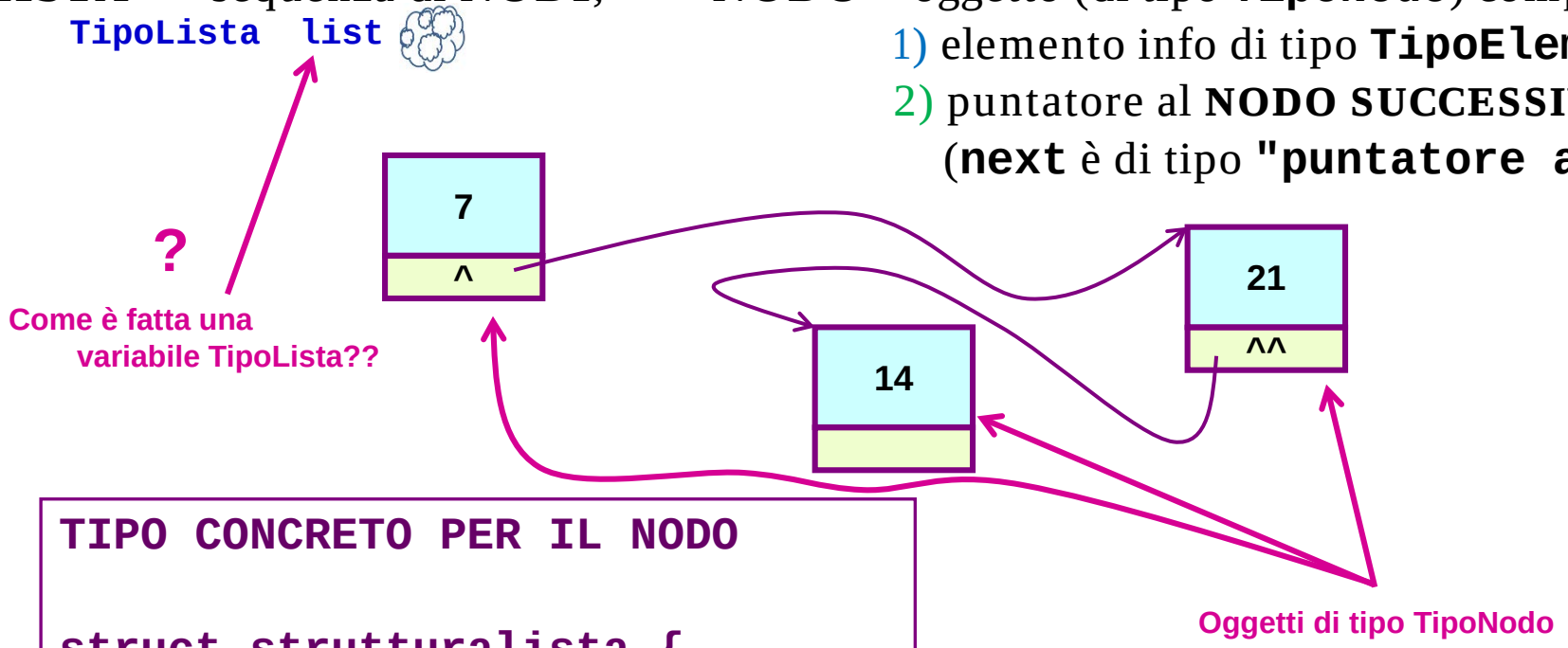
# Rappresentazione di LISTA mediante struct e puntatori

supponendo di aver già definito **`typedef ... TipoElem`**

**LISTA =** sequenza di **NODI**;

**`TipoLista list`**

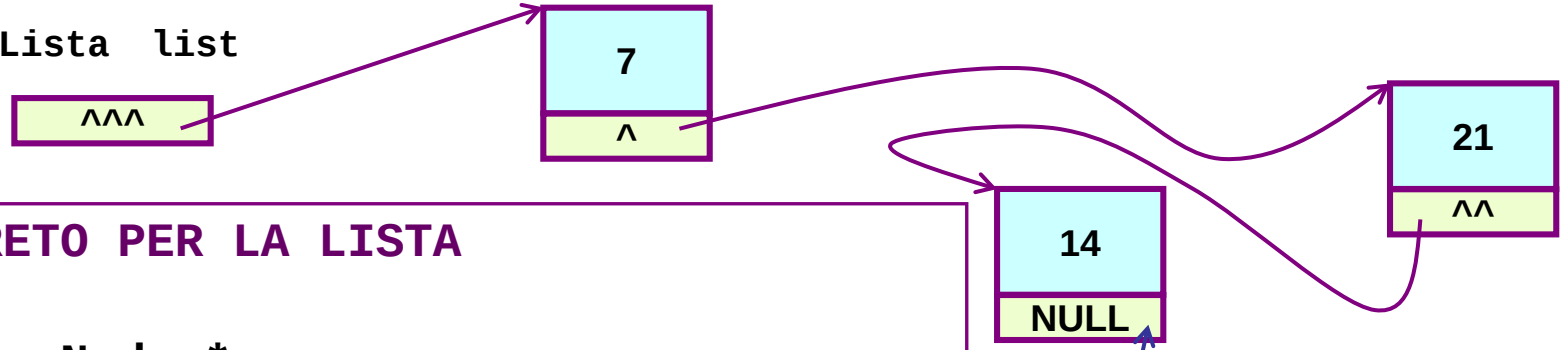
**NODO** = oggetto (di tipo **`TipoNodo`**) composto da
1) elemento info di tipo **`TipoElem`**
2) puntatore al **NODO SUCCESSIVO `next`**
(**`next`** è di tipo **`"puntatore a NODO"`**)




```
TIPO CONCRETO PER IL NODO

struct strutturalista {
  TipoElem info;
  struct strutturalista * next;
};

typedef struct strutturalista
        TipoNodo
```

**NB** nella definizione di
**`struct strutturalista`**
si usa **`struct strutturalista`**

# Rappresentazione di LISTA mediante struct e puntatori: TIPOLISTA




**TIPO CONCRETO PER LA LISTA**

```
typedef TipoNodo *
        TipoLista
```

Una variabile di tipo TipoLista è un puntatore che punta al primo nodo di una sequenza di nodi.

chiusura della lista (non c'è il nodo successivo)

Nel costruire la lista, i nodi vengono allocati dinamicamente, man mano che serve:

- quando c'è una nuova informazione da aggiungere in lista, si alloca un nuovo nodo (aggiunta o *inserimento in lista*);
- quando un elemento della lista non serve più, si dealloca il nodo corrispondente (*eliminazione da lista*) ... con attenzione ...

una lista di interi ...

```
typedef int TipoElem
…
TipoLista list
```

lista vuota

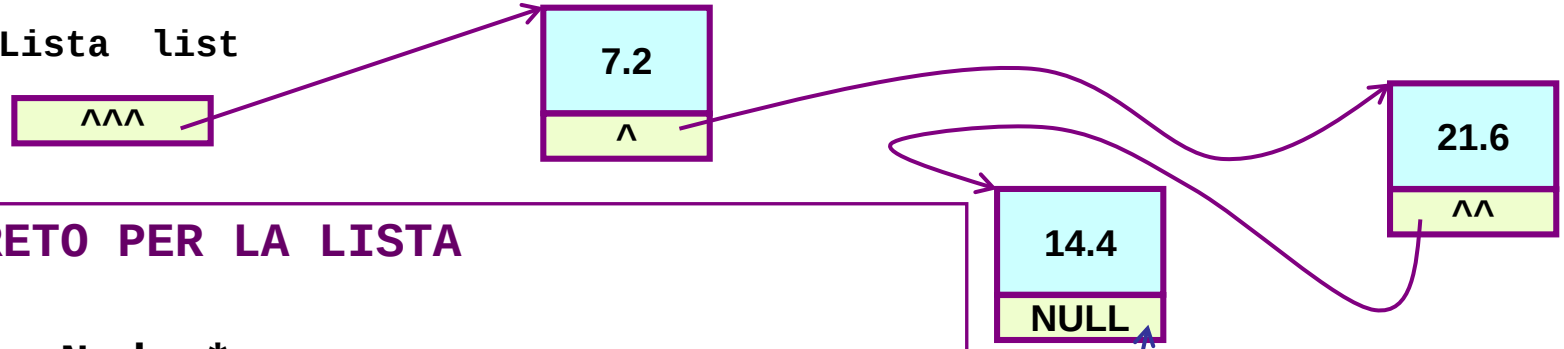
```
TipoLista list
```

NULL

inizializzazione di una lista

```
TipoLista lis
lis = NULL;
```

# Rappresentazione di LISTA mediante struct e puntatori: TIPOLISTA




**TIPO CONCRETO PER LA LISTA**

```
typedef TipoNodo *
        TipoLista
```

Una variabile di tipo TipoLista è un puntatore che punta al primo nodo di una sequenza di nodi.

chiusura della lista (non c'è il nodo successivo)

Nel costruire la li … che serve:

- quando c'è un … n nuovo nodo (aggiunta o *in* …
- quando un ele … ispondente (*eliminazione* …

Cambiando il tipo dell'elemento della lista, TipoElem, si cambia il tipo di lista rappresentatato …
prima era una lista di interi, adesso è una lista di double …

una lista di double ...

```
typedef double TipoElem
…
TipoLista list
```

lista vuota

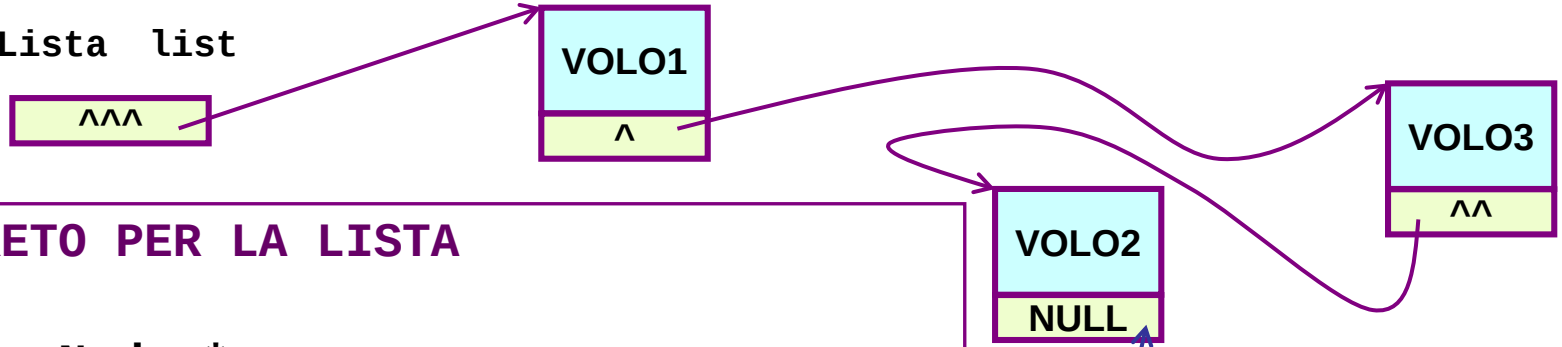
```
TipoLista list
```

NULL

inizializzazione di una lista

```
TipoLista lis
lis = NULL;
```

# Rappresentazione di LISTA mediante struct e puntatori: TIPOLISTA




**TIPO CONCRETO PER LA LISTA**

```
typedef TipoNodo *
        TipoLista
```

Una variabile di tipo TipoLista è un puntatore che punta al primo nodo di una sequenza di nodi.

chiusura della lista (non c'è il nodo successivo)

Nel costruire la li … che serve:

- quando c'è un … n nuovo nodo (aggiunta o *in*
- quando un ele … ispondente (*eliminazione*

…

una lista di voli ...

```
typedef TipoVolo TipoElem
…
TipoLista list
```

lista vuota

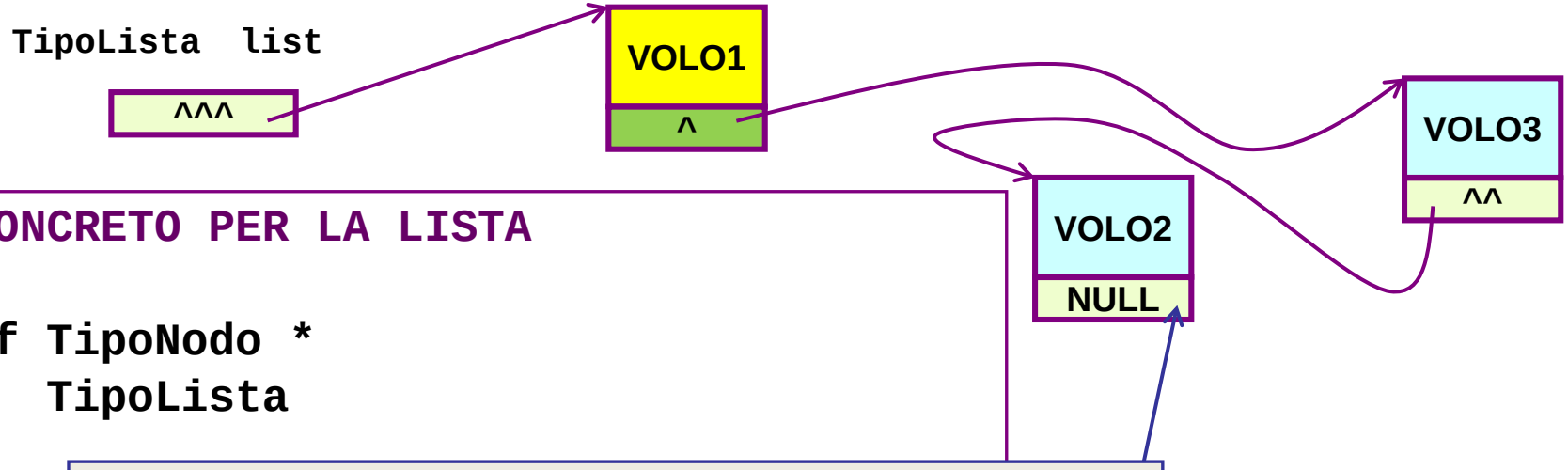
```
TipoLista list
```

NULL

inizializzazione di una lista

```
TipoLista lis
lis = NULL;
```

# Rappresentazione di LISTA mediante struct e puntatori: TIPOLISTA




**TIPO CONCRETO PER LA LISTA**

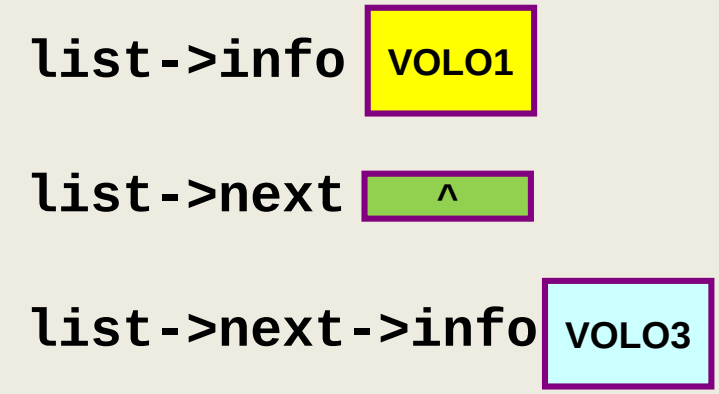
```
typedef TipoNodo *
        TipoLista
```

Una variabi… …niusura della lista (non primo nodo … …è il nodo successivo)

Nel costruire … …nano che serve:

- quando c… …oca un nuovo nodo (aggiunta…
- quando u… …corrispondente (*eliminaz…*

**NB**




una lista di voli …

```
typedef TipoVolo TipoElem
…
TipoLista list
```

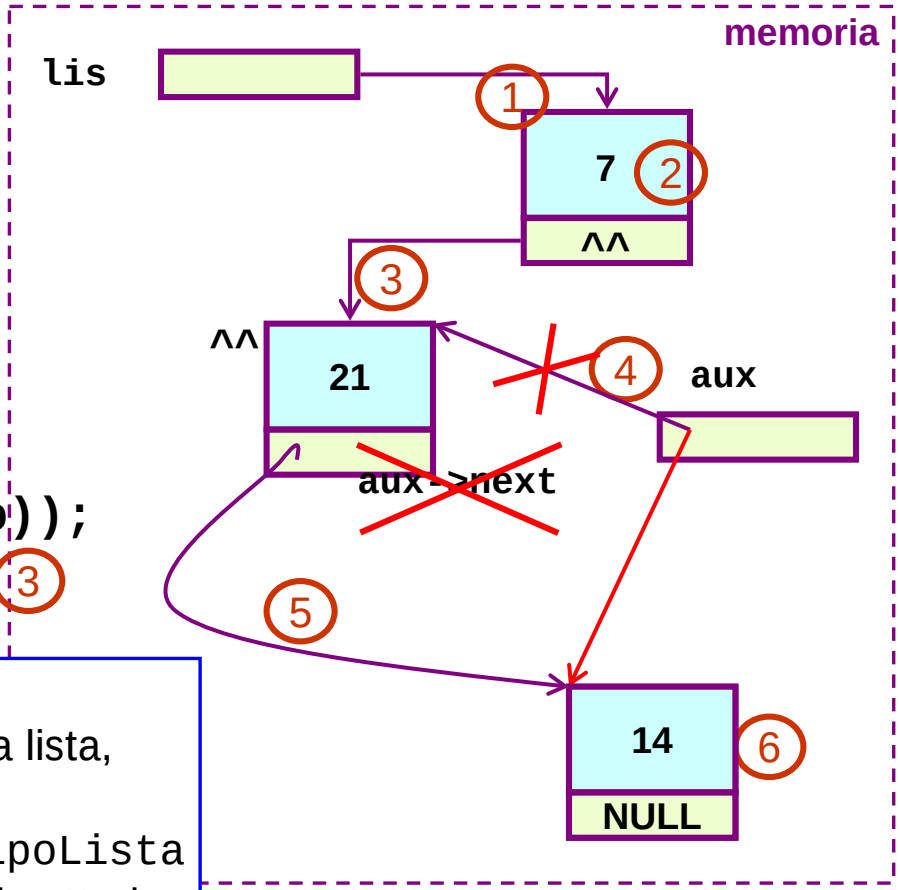
lista vuota


inizializzazione di una lista

```
TipoLista lis
lis = NULL;
```

# LISTA "struct e puntatori": allocazione di nodi in lista

```
#include <stdio.h> e <stdlib.h>
        def. TipoElem (int),
         TipoNodo e TipoLista
TipoLista lis

lis = malloc(sizeof(TipoNodo));   1

lis->info = 7;                    2

lis->next = malloc(sizeof(TipoNodo));
lis->next->info = 21              3
```




`TipoNodo *aux;`

puntatore ausiliario - non rappresenta una lista, ma serve solo a puntare nodi durante il lavoro quindi non viene definito di tipo `TipoLista` ma di tipo `puntatore a TipoNodo`

```
aux = lis->next;                  4
aux->next = malloc(sizeof(TipoNodo));   5
aux = aux->next;
aux->info=14;                     6
aux->next=NULL;
```

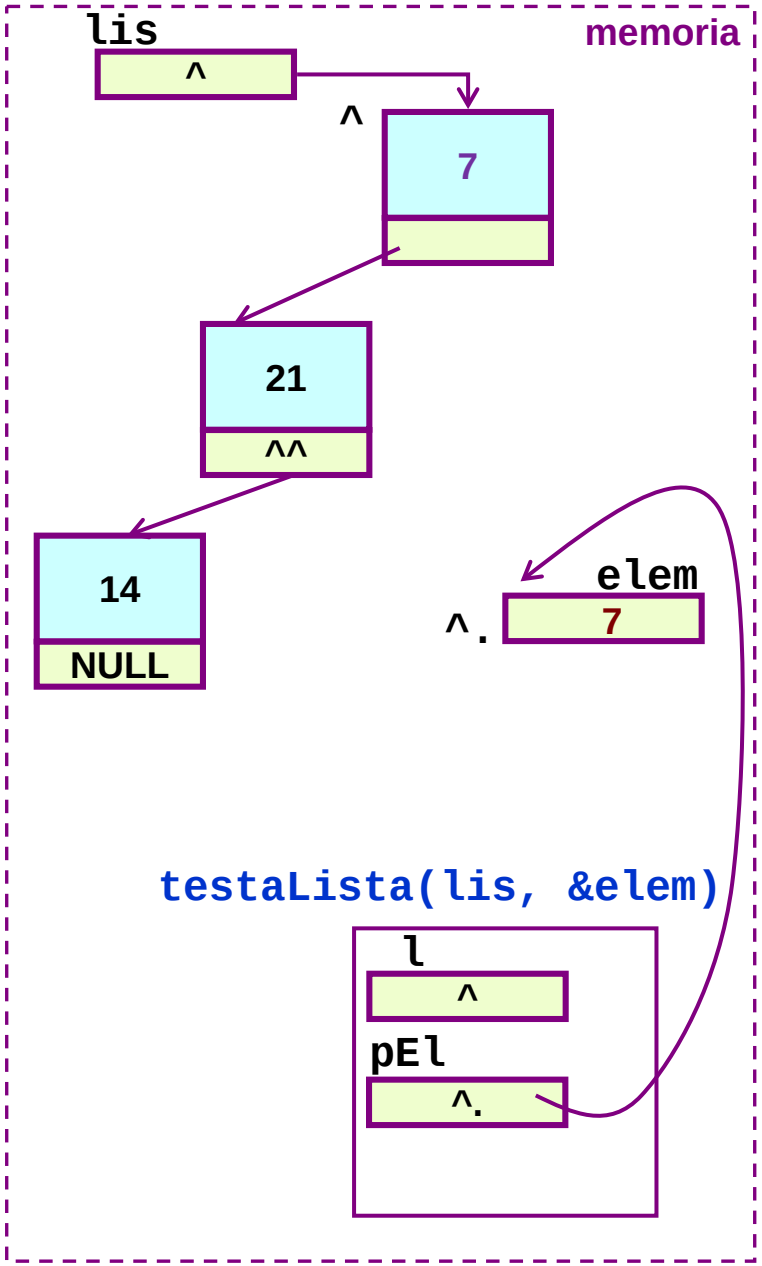
perche' usare aux, e non, ad esempio, lis?
perche' se avessimo spostato lis sul secondo nodo nessuno più avrebbe puntato sul primo
(e lis "*è una lista*" perche' punta al primo nodo di una sequenza ...)

# LISTA "struct e puntatori": alcune funzioni

```
Bool testListaVuota (TipoLista l) {
   return (l == NULL);
}
```

```
void testaLista
     (TipoLista l, TipoElem *pEl) {

  if (testListaVuota(l))
    printf("\nlista vuota!\n");
  else *pEl = l->info;
return;
}
```

```
/* esempio ... */
int main() {
 TipoElem elem; TipoLista lis;
  ...
  elem = testaLista(lis, &elem);
  ...
```



# LISTA "struct e puntatori": initLista( )

```
Bool testListaVuota (TipoLista l) {
   return (l == NULL);
}
```

Espressione booleana che valuta a **0** se l è diverso da NULL (cioè se l è non vuota) e produce il valore **1** se l è uguale a NULL, cioè è vuota

```
void initLista(TipoLista *pLis) {
  *pLis = NULL;
return;
}
```

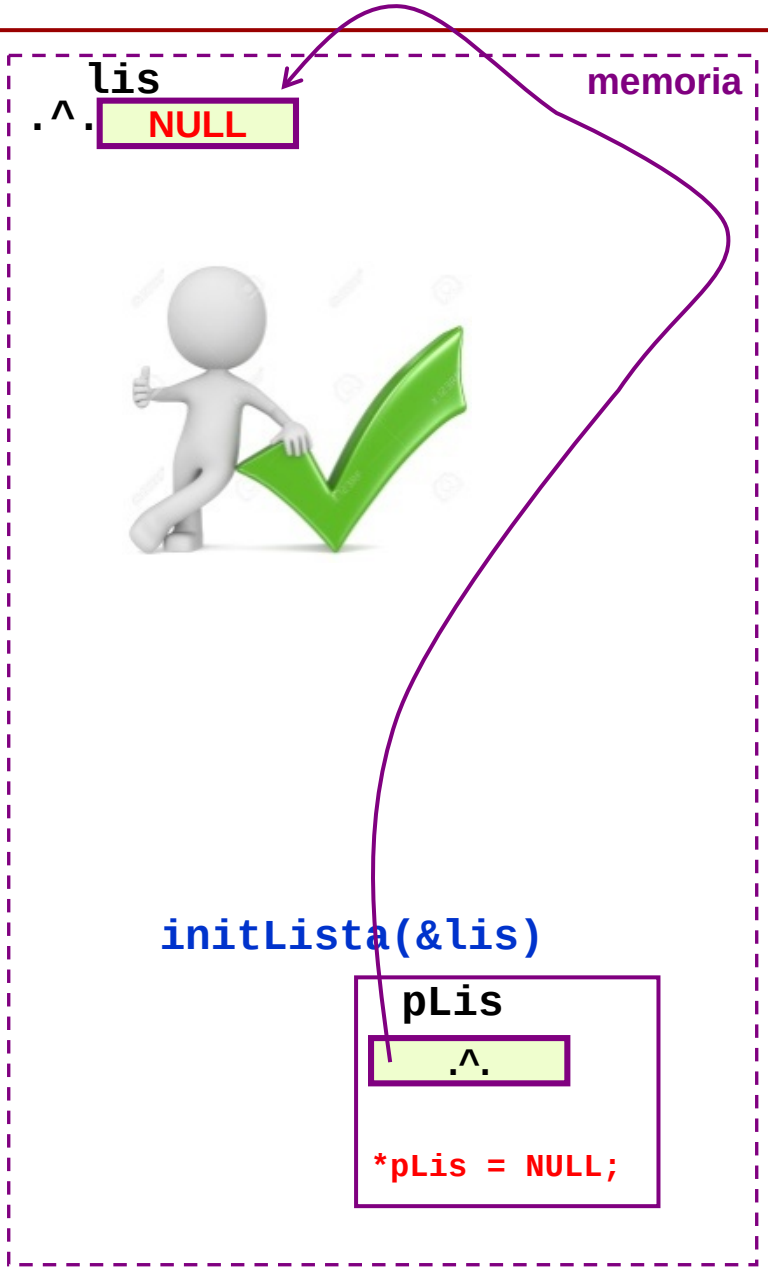
Esempio di chiamata della funzione sulla lista in figura (vuota … o svuotata, sennò non vale ... perdiamo nodi esistenti)

```
initLista (&lis);
```

La funzione produce un effetto collaterale su lis, per cui lis assume il valore NULL.

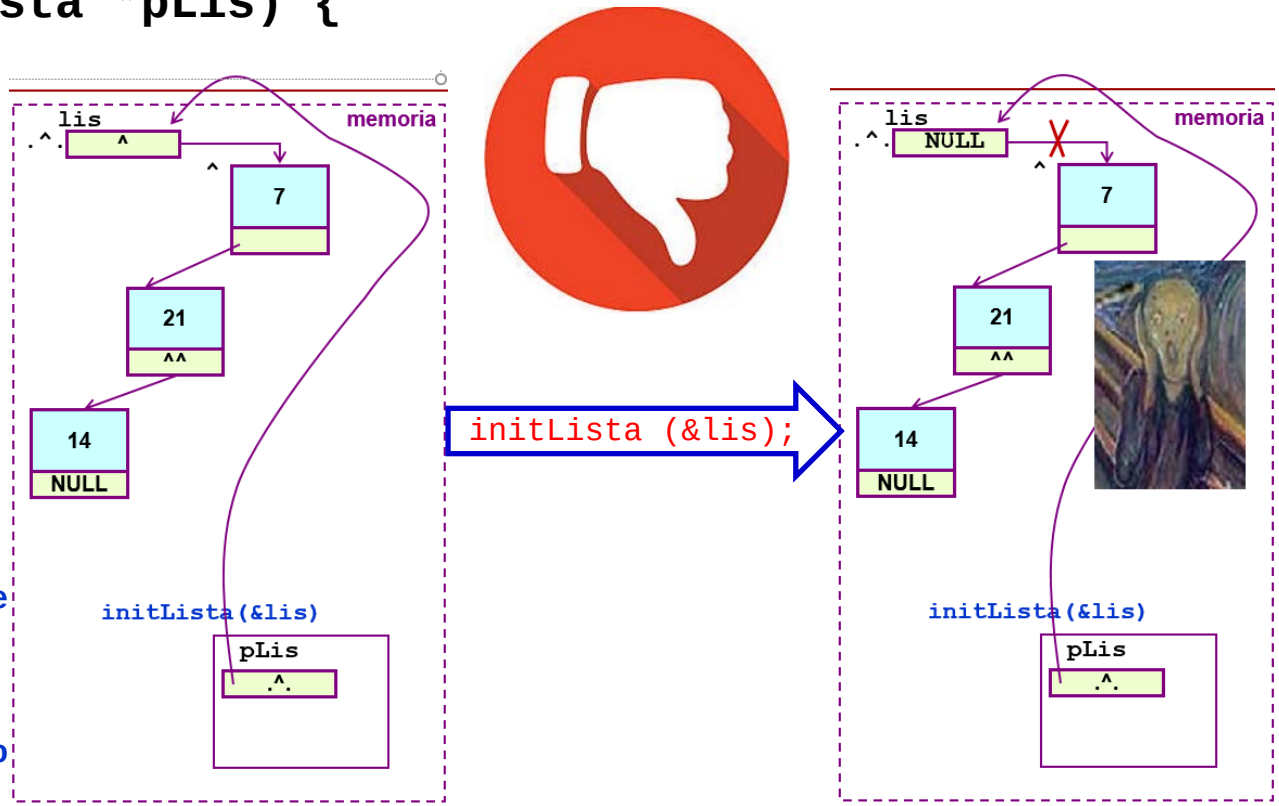
Ora lis è pronta per essere usata nella gestione di una nuova lista

```
/* esempio ... */
int main() {
 TipoLista lis;
  ...
  initLista((&lis);
  ...
```



# LISTA "struct e puntatori": `initLista()`, non cosi`

```
void initLista(TipoLista *pLis) {
  *pLis = NULL;
return;
}
```




initLista (&lis);
In questo caso la funzione metterebbe NULL in lis ...
e non va bene

**Se chiamiamo la funzione mentre lis punta ad una lista esistente (i.e. con almeno un nodo), allora fissiamoci allo specchio con disapprovazione: ora i nodi di quella lista sono ancora in memoria, ma non ne abbiamo piu` l'indirizzo e quindi sono irraggiungibili.**

**Viceversa se lis non era inizializzata, oppure se la lista cui puntava era stata deallocata, e quindi ora lis era disponibile per gestire un'altra lista, allora la chiamata di initLista() correttamente inizializza lis, con il che da ora possiamo usare lis con tranquillita` (mettete il mattino di Grieg mentre programmate).**

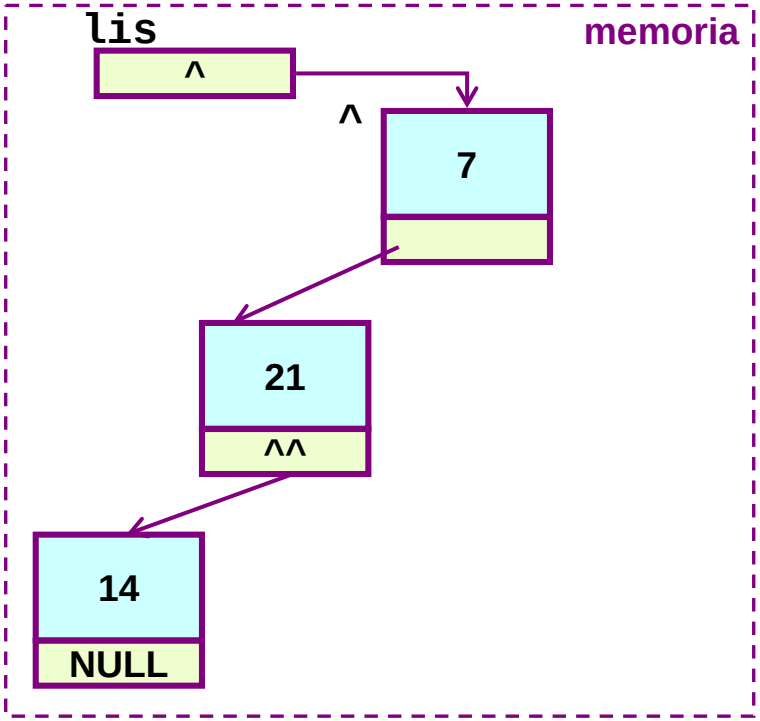
# LISTA "struct e puntatori": eliminazione primo elemento (cdr)

```
TipoLista lis;
```

**come facciamo a deallocare il primo elemento della lista?**

~~`free(lis);`~~

**No, non è questa la soluzione**



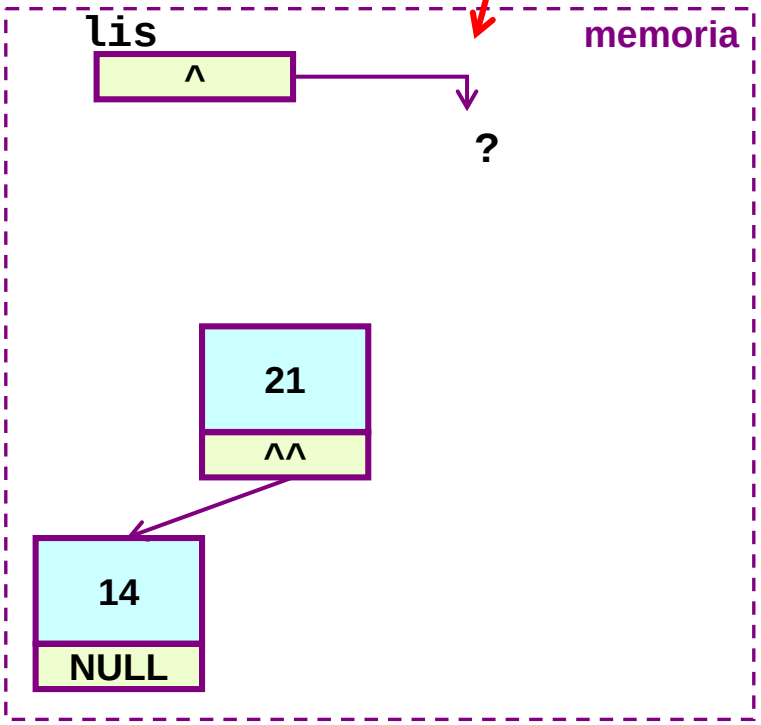

**perche'?**

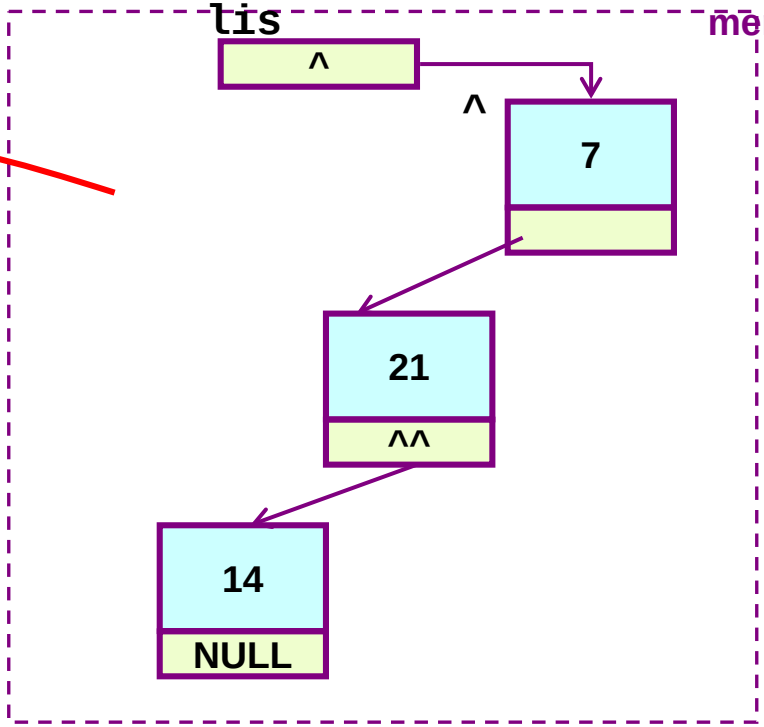
# LISTA "struct e puntatori": eliminazione primo elemento (cdr) - NON COSì

```
TipoLista lis;
```

come facciamo a deallocare il primo elemento della lista?

```
free(lis);
```




No, non è questa la soluzione

perche' viene deallocato il nodo iniziale ma la variabile lis

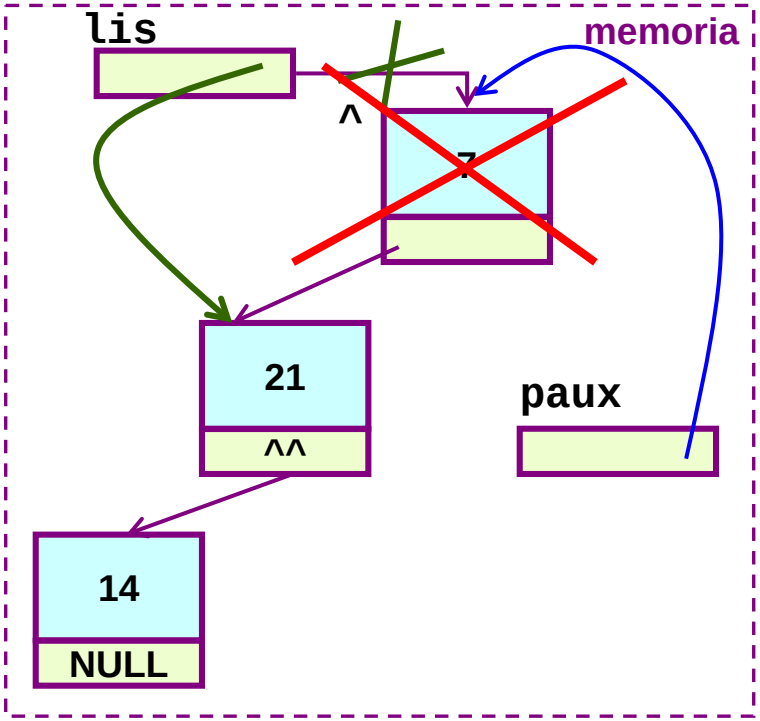
non punta su quello che dovrebbe essere ora il primo nodo;

... !! e in effetti nessuno punta su quello che dovrebbe essere il primo nodo!! eeek!!!

la lista è perduta nei meandri della memoria ...

... a meno che non ci siamo attrezzati per evitarlo ...

# LISTA "struct e puntatori": eliminazione primo elemento (cdr) - Così sì




```
TipoLista lis;

                TipoNodo * paux;

                /* puntatore ausiliario
                per puntare il nodo da
                eliminare, prima di
                cambiare lis */

 paux = lis;

  lis = lis->next;

  free(paux);

  /* paux = NULL */ se serve ...
```

**Sì, è questa la soluzione!**

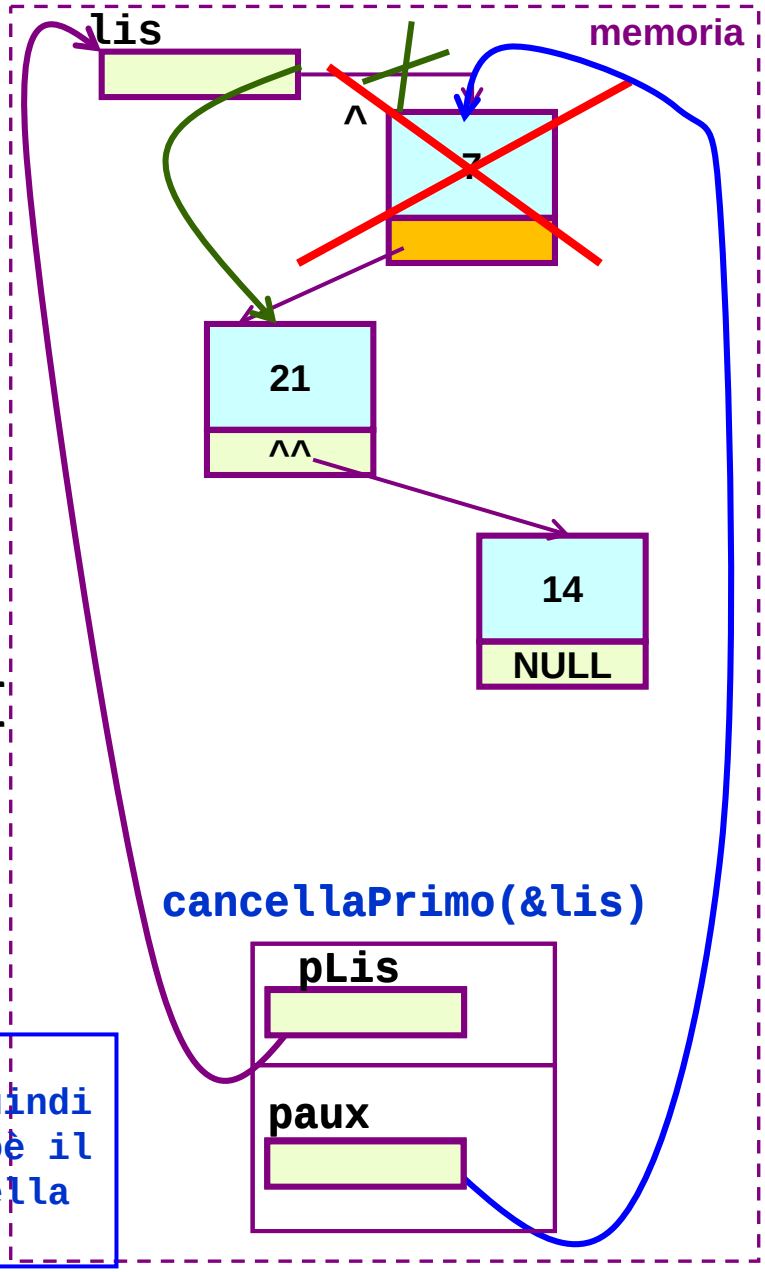
# LISTA "struct e puntatori": eliminazione primo elemento (cdr)

```
TipoLista lis;

                    TipoNodo * paux;

  paux = lis;
  lis = lis->next;
  free(paux);
  /* paux = NULL */ se serve ...
```

**E ora … funzione che elimina il primo elemento di una lista**

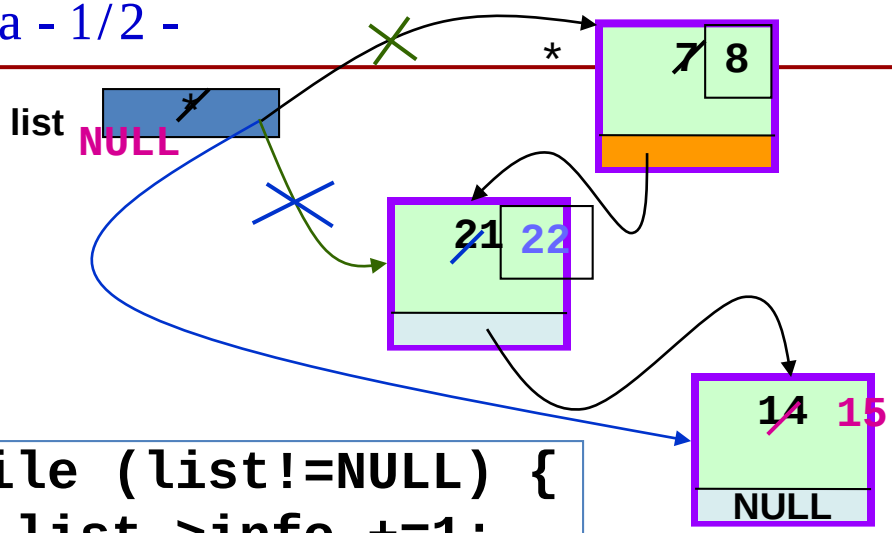
```
 void cancellaPrimo (Tipolista *pLis) {
  TipoNodo * paux;
   paux = *pLis;
   *pLis = (*pLis)->next;
   free(paux);
 return;
 }
```

scrivere "*pLis" nel RDA è come scrivere "lis" fuori di esso; quindi (*pLis)->next è lis->next, cioè il puntatore al secondo elemento della lista



# LISTA "struct e puntatori": **scansione** lista - 1/2 -

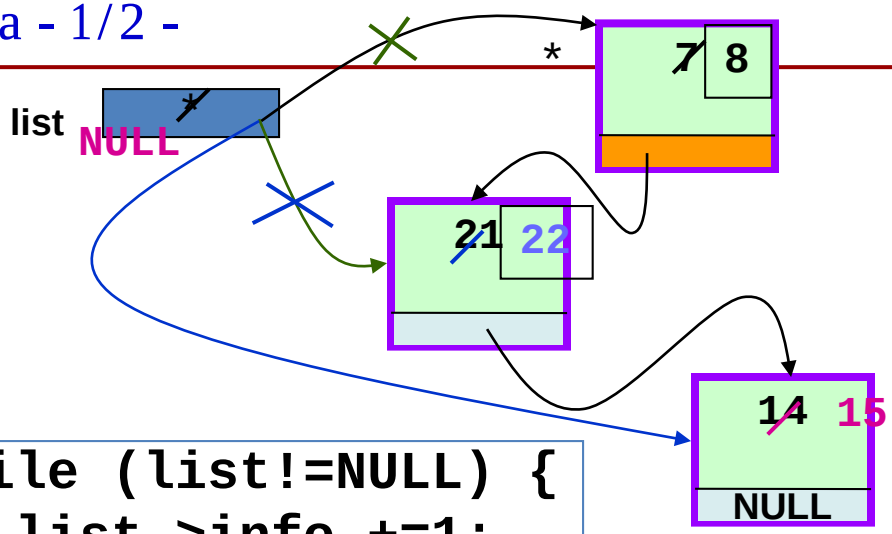
**Scansione di una lista =**
**si esegue** (almeno) **un'operazione su ciascun nodo della lista** *(es. aggiungere 1)*




```
list->info +=1;
list= list->next;
list->info +=1;
list=list->next;
list->info +=1;
list=list->next;
BASTA (list==NULL)
}
```

```
while (list!=NULL) {
    list->info +=1;
    list=list->next;
}
```

**Scansione di una lista =**
**si esegue** (almeno) **un'operazione su ciascun nodo della lista** *(es. aggiungere 1)*

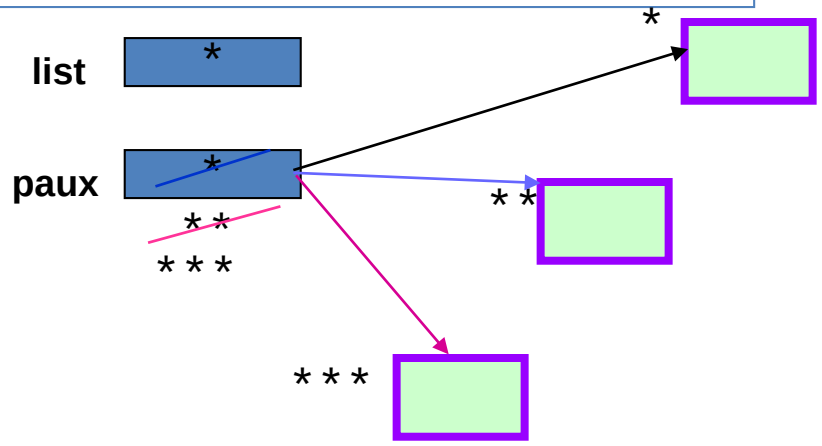



```
list->info +=1;
list= list->next;
list->info +=1;
list=list->next;
list->info +=1;
list=list->next;
BASTA (list==NULL)
}
```

```
while (list!=NULL) {
    list->info +=1;
    list=list->next;
}
```

**ARGH!!!!!!! ma adesso list==NULL**
**nessuno punta all'inizio della lista**
**soluzione: puntatore ausiliario**

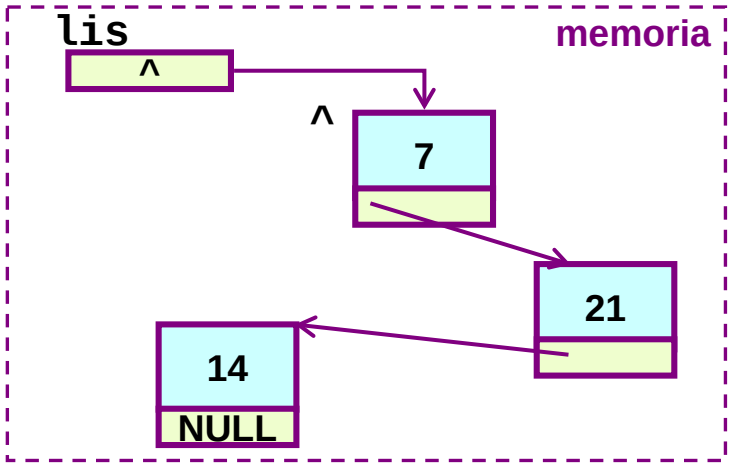
```
TipoNodo * paux;

 paux=list;
 while (paux!=NULL) {
    paux->info +=1;
    paux=paux->next;
}
```



## esempio: aggiungere 1 a tutti i nodi

```
...
int main() {
 TipoLista lis;
 TipoNodo * paux;
 ...
 paux = lis;
 while (paux) {
   paux->info +=1;
   paux = paux->next;
 }
...}
```

così lis rimane sul primo



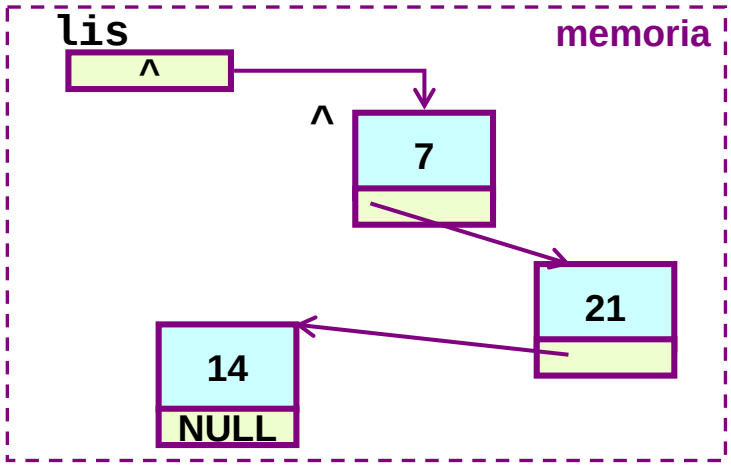

## esempio: stampa

```
...
void stampaLista (TipoLista l) {
 TipoNodo * paux = l;
 while (paux) {
   stampaElem(paux->info);
   paux = paux->next;
 }
return;
}
```

## esempio: aggiungere 1 a tutti i nodi

```
...
int main() {
 TipoLista lis;
 TipoNodo * paux;
 ...
 paux = lis;
 while (paux) {
   paux->info +=1;
   paux = paux->next;
 }
...}
```

così lis rimane sul primo




## esempio: stampa

```
...
void stampaLista (TipoLista l) {
 TipoNodo * paux = l;
 while (paux) {
   stampaElem(paux->info);
   paux = paux->next;
 }
return;
}
```

**due osservazioni**

1)

dopo rivedremo questa soluzione per stampalista e per certi altri casi di funzioni che hanno un parametro lista, passato per valore

2) usiamo questa funzione accessoria

```
void stampaElem (TipoElem e) {
 /* supponendo interi, come negli
      esempi */
 printf(" %d", e);
return;
}
```

**Gran parte delle funzioni che gestiscono TipoLista sono tali da poter e dover rimanere uguali al cambiare di TipoELem.**
**Ad esempio se TipoElem fosse una lista di VOLI, provvisto che stampaElem() avesse un'implementazione consona, come**

```
void stampElem(TipoElem el) {
   stampaVolo(el);
return;
}
```

**la funzione stampalista potrebbe rimanere uguale.**

```
...
void stampaLista (TipoLista l) {
 TipoNodo * paux = l;
 while (paux) {
   stampaElem(paux->info);
   paux = paux->next;
 }
return;
}
```

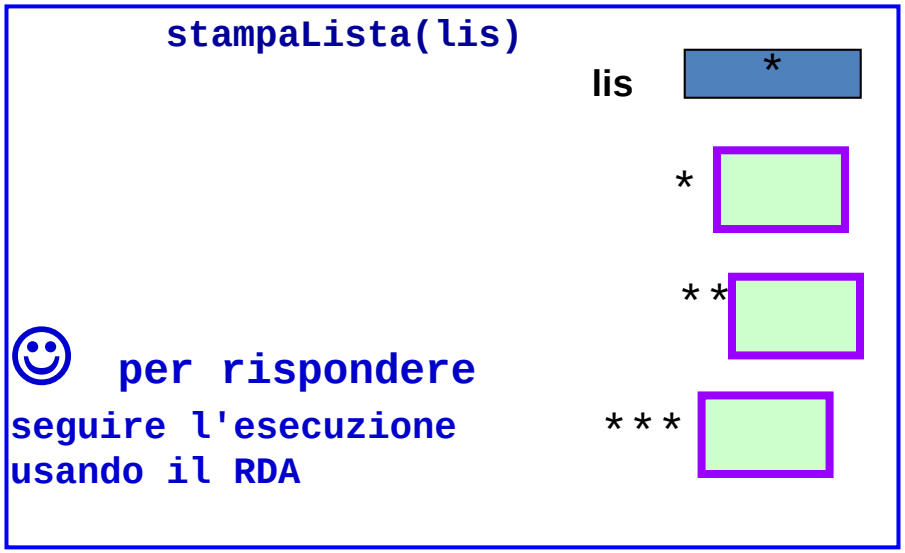
**il concetto è quello di "riuso delle funzioni che implementano algoritmi di scansione di lista" senza cambiarle: cambiando invece le funzioni accessorie ..**

se l è una lista di voli, vengono stampati i voli …
se l è una lista di persone, vengono stampate le persone …
se l è una lista di cavalieri catafratti, vengono stampati i cavalieri catafratti …
…

# LISTA "struct e puntatori": a proposito di stampaLista

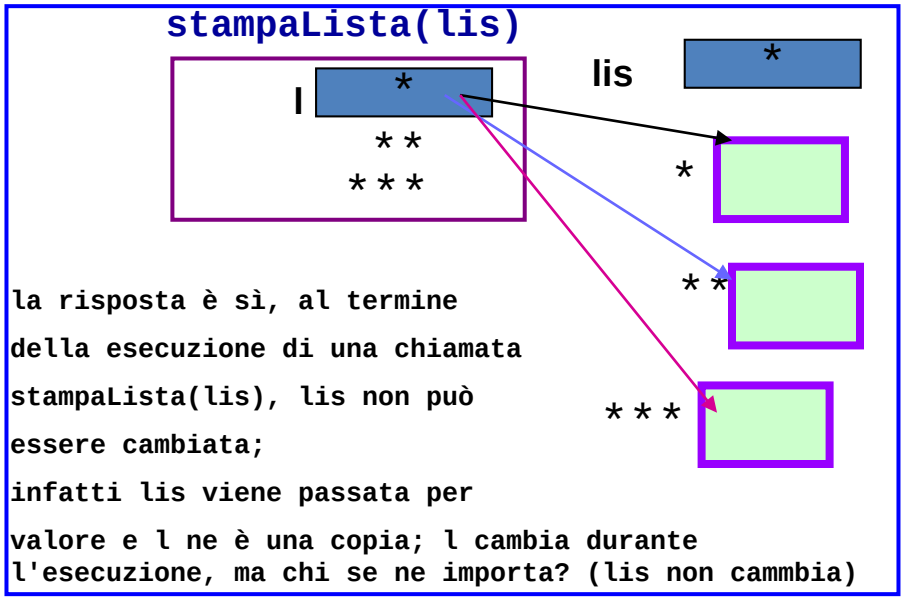
**domanda: va bene la seguente implementazione di stampaLista()?**

```
...
void stampaLista (TipoLista l) {
  while (l) {
   stampaElem(l->info);
   l = l->next;
 }
return;
}
```



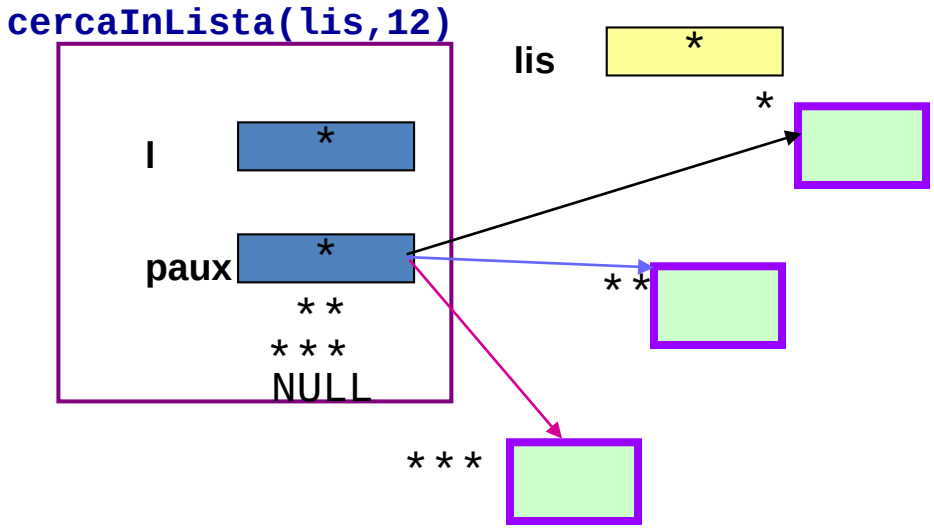
**inoltre: ?va bene la seguente implementazione di stampaLista()?**

```
...
void stampaLista (TipoLista l) {
  while (l) {
    stampaElem(l->info);
    l = l->next;
 }
return;
}
```



# LISTA "struct e puntatori": scansione per ricerca in lista

```
TipoNodo * cercaInLista (TipoLista l, TipoElem el) {

/* cerca el in l e restituisce il puntatore al nodo in cui el si
                                                    trova (oppure NULL)
ALGORITMO
-    scansione con paux: mentre paux punta su un nodo (!= NULL) …
-    (*paux è il nodo) {se paux->info==el si assegna trovato=1}
*/
  TipoNodo * paux=l;
    int trovato = 0;
  while(paux && !trovato)
    if (uguali(paux->info, el))
      trovato = 1;
    else paux = paux->next;

 return paux;
 }
```



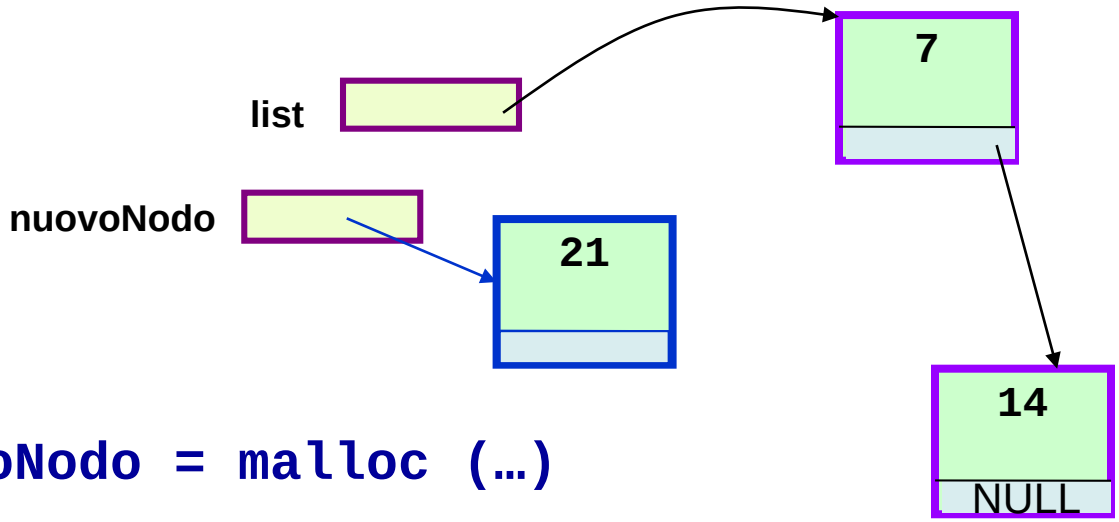

al termine del ciclo,

- o trovato è ancora 0 (siamo usciti dal ciclo con paux==NULL, senza trovare el);
- oppure trovato è cambiato (è diventato 1); in questo caso siamo usciti dal ciclo con paux che punta al nodo contenente el;

in entrambi i casi paux contiene il risultato giusto da restituire (o NULL o qualcosa)

# LISTA "struct e puntatori": inserimento in lista, *in testa* - algoritmo

```
TipoLista list;
TipoNodo *nuovoNodo;
```




1) allocazione nuovoNodo

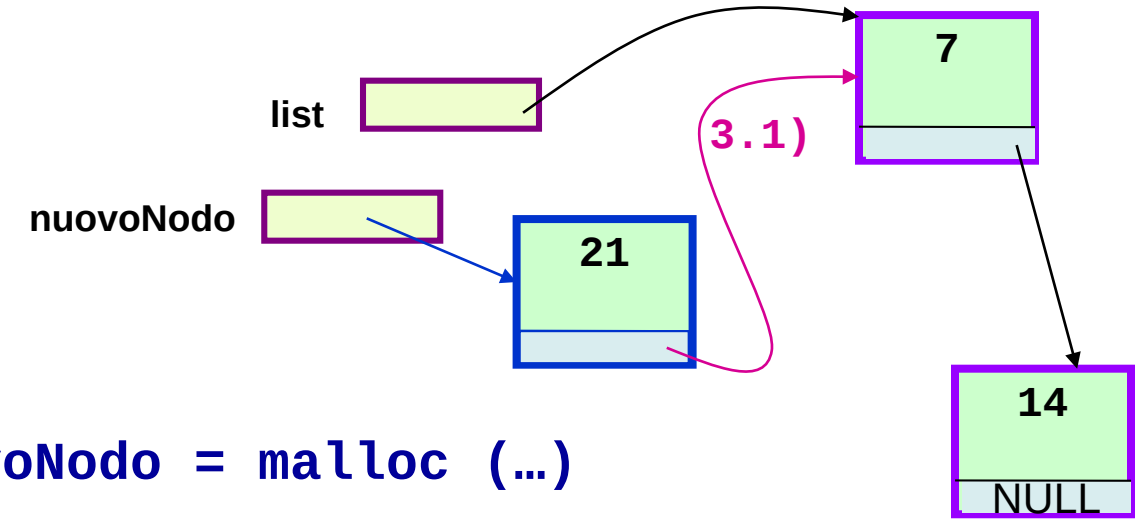
```
nuovoNodo = malloc (…)
```

2) assegnazione

```
nuovoNodo->info = 21
```

# LISTA "struct e puntatori": inserimento in lista, *in testa* - algoritmo

```
TipoLista list;
TipoNodo *nuovoNodo;
```




1) allocazione nuovoNodo

```
nuovoNodo = malloc (…)
```

2) assegnazione

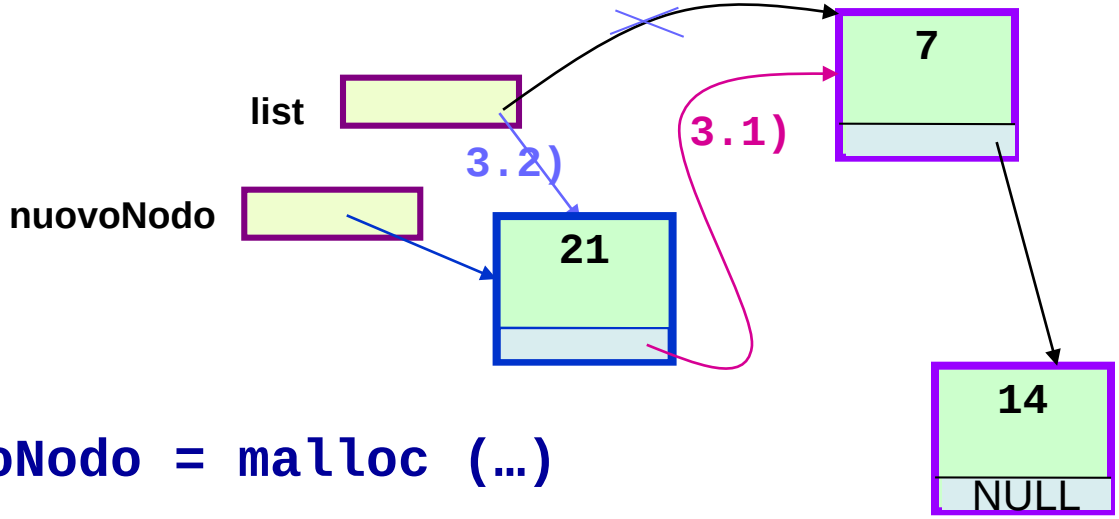
```
nuovoNodo->info = 21
```

3.1) fare in modo che il primo nodo diventi successore del nuovoNodo

```
nuovoNodo->next = list
```

# LISTA "struct e puntatori": inserimento in lista, *in testa* - algoritmo

```
TipoLista list;
TipoNodo *nuovoNodo;
```




1) allocazione nuovoNodo

```
nuovoNodo = malloc (…)
```

2) assegnazione

```
nuovoNodo->info = 21
```

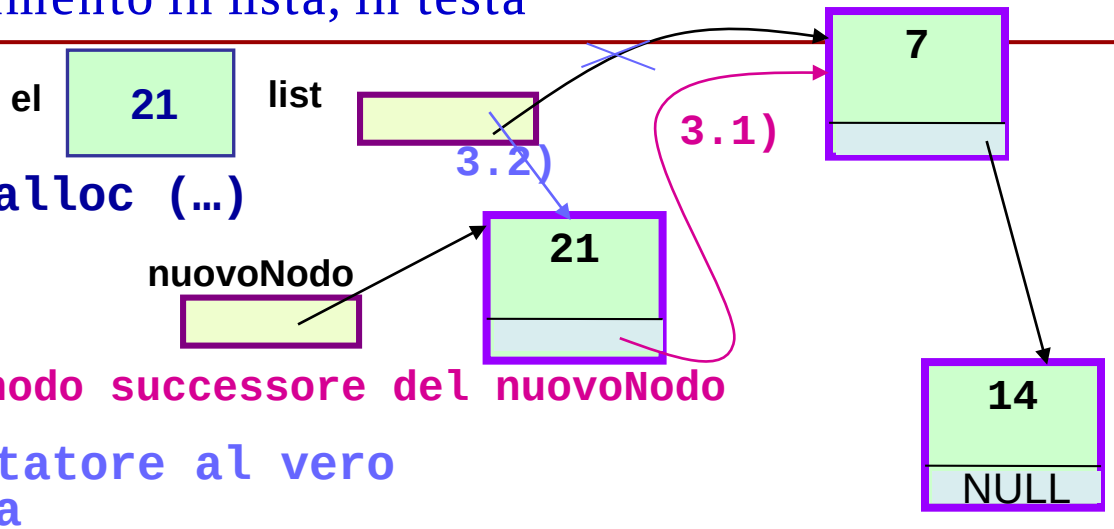
3.1) fare in modo che il primo nodo diventi successore del nuovoNodo

```
nuovoNodo->next = list
```

3.2) far diventare list puntatore al nuovo nodo, che deve ora essere i primo nodo della lista

```
list = nuovoNodo
```

# LISTA "struct e puntatori": inserimento in lista, in testa




1) allocazione nuovoNodo
   TipoLista nuovoNodo = malloc (…)

2) assegnazione
   nuovoNodo->info = 21

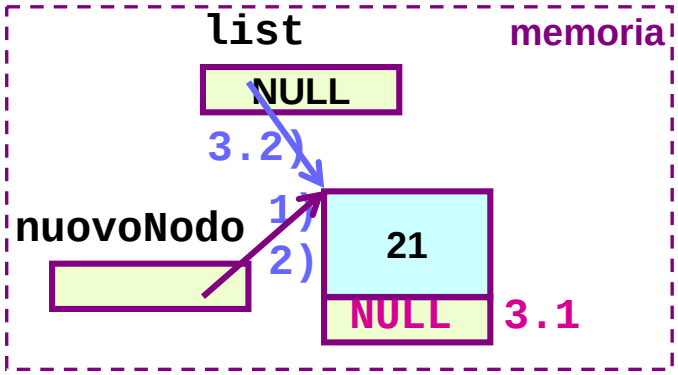
3.1) far diventare il primo nodo successore del nuovoNodo

3.2) far diventare lis puntatore al vero
     primo nodo della lista

```
int main() {                    TipoLista list;   TipoElem el;
                                TipoNodo * nuovoNodo;
...
...
  printf ("nuovo elem: ");
  leggiElem(&el);   /* o magari scanf("%d", &el) se intero */
  nuovoNodo = malloc (sizeof(TipoNodo));     /* 1) */
  nuovoNodo->info = el;              /* 2)  */
  nuovoNodo->next = list;            /* 3.1)*/
  list = nuovoNodo;                  /* 3.2)*/
... ... ...                                    return 0;}
```

# LISTA "struct e puntatori": inserimento in lista, in testa - CASO LISTA VUOTA

**CASO LISTA VUOTA**




l'istr. /* 3.2 */ fa sì che list punti al nuovo primo nodo della lista e non contenga più NULL;

precedentemente l'istruzionee /* 3.1 */ ha assegnato NULL al next del nuovo nodo; così questa lista di un nodo è chiusa per bene.

Quindi nel caso "LISTA VUOTA" l'algoritmo funziona bene.

Allora una lista si potrebbe costruire da zero, mediante una serie di inserimenti in testa

```
int main() {                    TipoLista list;   TipoElem el;
                                TipoNodo * nuovoNodo;

...
...
printf ("nuovo elem: ");
leggiElem(&el);  /* o magari scanf("%d", &el) se intero */
nuovoNodo = malloc (sizeof(TipoNodo));     /* 1) */
nuovoNodo->info = el;                      /* 2) */
nuovoNodo->next = list;                    /* 3.1)*/
list = nuovoNodo;                          /* 3.2)*/
… … … return 0;
}
```

0.1) resLista = NULL (init lista risultato)

0.2) lettura dato

1) costruzione nuovo nodo

2) assegnazione nuovo->info=dato

3) inserimento in testa del nuovo nodo

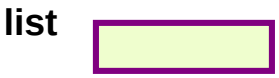
} n volte

```
TipoLista costruisciLista (int n) {
  TipoLista resLista = NULL;                    /* 0.1) */
  int i;      TipoElem dato;       TipoNodo * nuovo;

 for (i=1; i<=n; i++) {
   printf ("nuovo elem (intero): ");
   leggiElem( &dato );                          /* 0.2) */
   nuovo = malloc (sizeof(TipoNodo));           /* 1) */
   nuovo->info = dato;                          /* 2) */
   nuovo->next = resLista;                      /* 3.1)*/
   resLista = nuovo;                            /* 3.2)*/
 }
return resLista; }
```

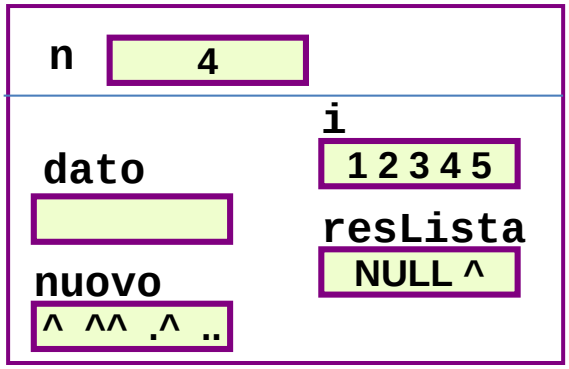
# LISTA "struct e puntatori": costruzione di una lista di n nodi mediante inserimento in testa

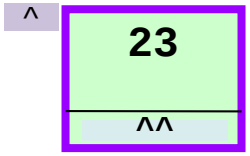
```
int main() { TipoLista list;
in quanti=4;
...
...
list = costruisciLista(quanti);
...
...
return 0;
}
```

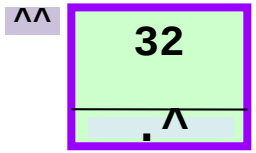
```
TipoLista costruisciLista (int n) {
  TipoLista resLista = NULL;                    /* 0.1) */
  int i;      TipoElem dato;       TipoNodo * nuovo;

 for (i=1; i<=n; i++) {
   printf ("nuovo elem (intero): ");
   leggiElem( &dato );                          /* 0.2) */
   nuovo = malloc (sizeof(TipoNodo));           /* 1) */
   nuovo->info = dato;                          /* 2) */
   nuovo->next = resLista;                      /* 3.1)*/
   resLista = nuovo;                            /* 3.2)*/
 }
return resLista; }
```

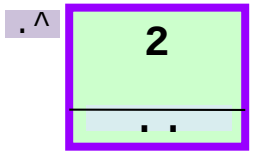


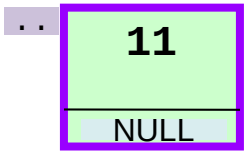

costruisciLista(quanti)



# LISTA "struct e puntatori": **FUNZIONE** di inserimento in lista, in testa

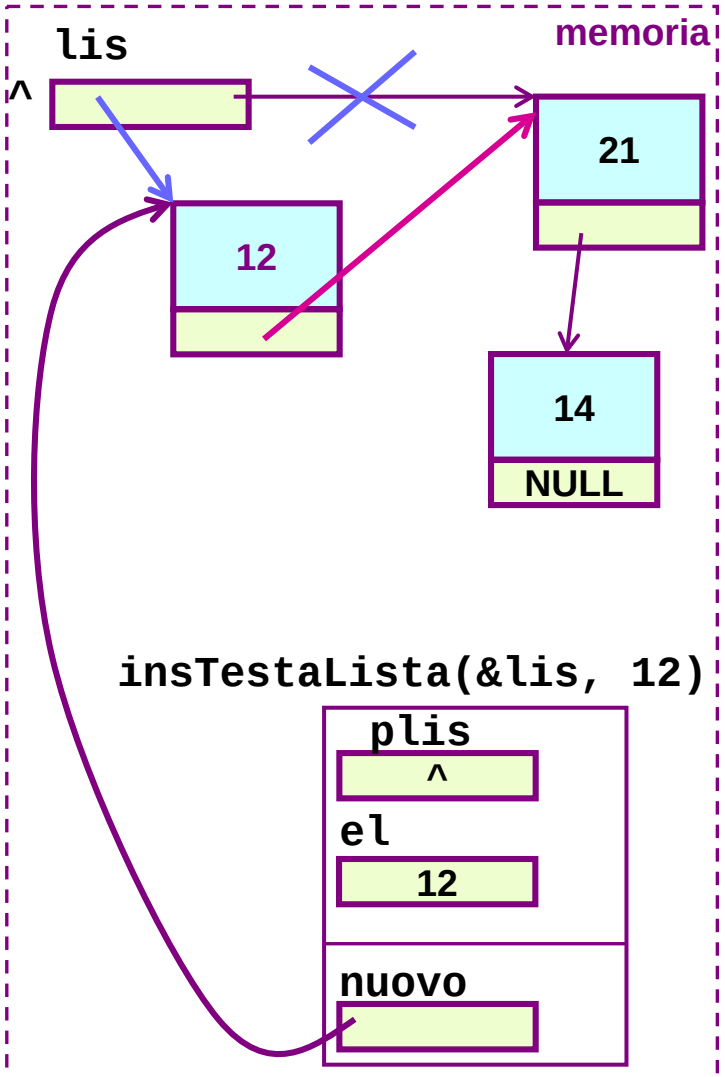
**L'idea è che con la chiamata**

```
insTestaLista(&lis, 12)
```

**la lista (21, 14) diventi (12, 21, 14)**

**in linea di principio, la lista subisce un effetto collaterale (perche'?) e quindi "la passiamo" tramite indirizzo**

```
void insTestaLista(TipoLista *plis,
                   TipoElem el)  {
   TipoNodo * nuovo;

 nuovo = malloc(sizeof(TipoNodo));
 nuovo->info = el;
 nuovo->next = *plis;     /* 3.1 */
 *plis = nuovo;           /* 3.2 */
return;
}
```




**NB1 In pratica lis – la variabile puntatore – deve poter cambiare con l'esecuzione della funzione quindi deve essere passata tramite indirizzo**

**NB2 se l'inserimento in testa ha successo, insTestaLista(&lis, 12) cambia la locazione lis («side effect»): dopo l'esecuzione della chiamata, lis punta al nuovo nodo.**

# LISTA "struct e puntatori": costruzione di una lista, usando insTestaLista()

0) resLista = NULL (init lista risultato)

1) lettura dato

2) inserimento del dato in testa a resLista

} n volte

```
TipoLista costrLista (int n) {
  TipoLista resLista = NULL;                    /* 0) */
  int i;    TipoElem dato;

 for (i=1; i<=n; i++) {
   printf ("nuovo elem (intero): ");
   leggiElem(&dato);
      /* o magari scanf("%d", &dato); se sono interi */     /* 1) */
   insTestaLista(&resLista, dato);                 /* 2) */
 }
return resLista;
}
```

supponendo che la chiamata sia lis = costrLista(3)

e la successione dei dati sia 7,14,21,

che lista viene? (disegnarla! Non dirla o pensarla, ok?)

# LISTA "struct e puntatori": costruzione di una lista, usando insTestaLista()

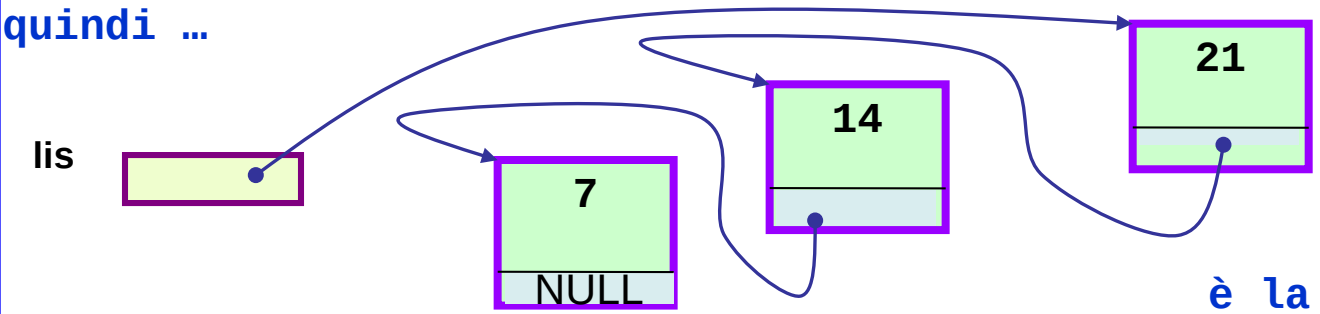
**0) resLista = NULL (init lista risultato)**

**1) lettura dato**

**2) inserimento del dato in testa a resLista** } **n volte**

```
TipoLista costrLista (int n) {
  TipoLista resLista = NULL;                    /* 0) */
  int i;    TipoElem dato;

 for (i=1; i<=n; i++) {
   printf ("nuovo elem (intero): ");
   leggiElem(&dato);
      /* o magari scanf("%d", &dato); se sono interi */    /* 1) */
   insTestaLista(&resLista, dato);               /* 2) */
 }
return resLista;
}
```

**Prima viene inserito 7, poi, in testa, viene inserito 14, e poi 21, quindi …**




**è la lista {21, 14, 7}**

# Lezione 22 - ADT LISTA - Esercizio

Scrivere un programma in cui

- viene definito il tipo TipoLista corrispondente e a "lista di numeri reali"
- usando il codice contenuto in questa lezione, senza fare uso di funzioni, che non siano di libreria, viene costruita e stampata una lista di 5 numeri reali
- usando il codice contenuto in questa lezione, facendo uso di una funzione di inserimento in testa, viene costruita e stampata una lista di 5 numeri reali