Liste collegate

Una lista è una sequenza ordinata di elementi dello stesso tipo. Per esempio, `(1 23 -1 2 -1)` è una sequenza di numeri interi, mentre `('a' 'r' 'l' 'P')` è una sequenza di caratteri. La notazione che usiamo per rappresentare le sequenze è quella di mettere gli elementi fra parentesi tonde, separati da spazi.

Il modo più semplice di rappresentare delle liste in C è quello di usare gli array. In queste pagine vediamo una rappresentazione alternativa, che fa uso delle cosidette *strutture collegate*, in cui i vari elementi che compongono un insieme composto di dati sono rappresentati in zone di memoria che possono anche essere distanti fra loro (al contrario degli array, in cui gli elementi sono consecutivi), e sono legati fra di loro mediante puntatori.

Rappresentare sequenze in memoria

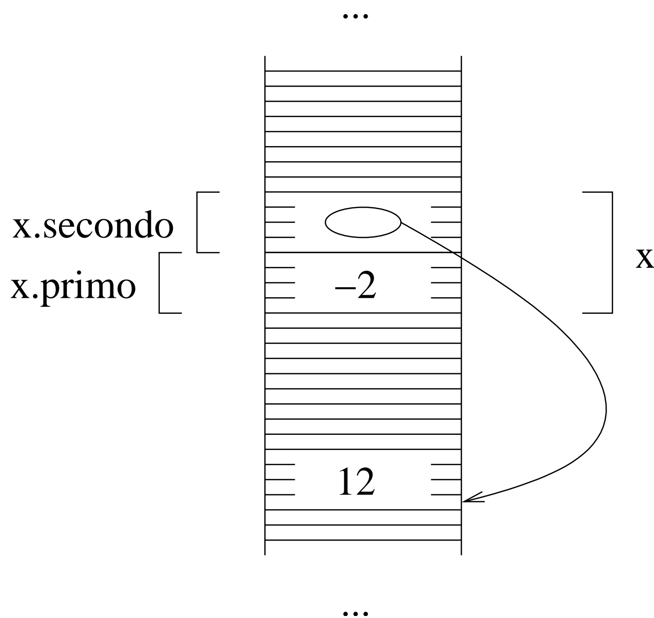
Supponiamo di voler rappresentare una specifica sequenza di interi in memoria, per esempio la sequenza `(-2 12)`, che è composta da due soli interi. Possiamo usare per esempio un array statico di due soli elementi, mettendo `-2` nella prima posizione e `12` nella seconda, oppure un array dinamico con la stessa disposizione. Se sappiamo che la sequenza da rappresentare sarà sempre composta da due elementi, possiamo anche usare una struttura:

```
struct SeqDue {
  int primo;
  int secondo;
};
```

Una rappresentazione alternativa, cha usa ancora le strutture, è la seguente:

```
struct SeqDue {
  int primo;
  int *secondo;
};
```

Questo metodo di rappresentazione può, a prima vista, sembrare illogico: perchè usare un puntatore a intero per il secondo elemento, invece che un intero? Vedremo poi che questa seconda rappresentazione si può estendere per rappresentare sequenze di lunghezza generica. Se la variabile `x` viene dichiarata di tipo `struct SeqDue`, allora la sua rappresentazione in memoria sarà la seguente:



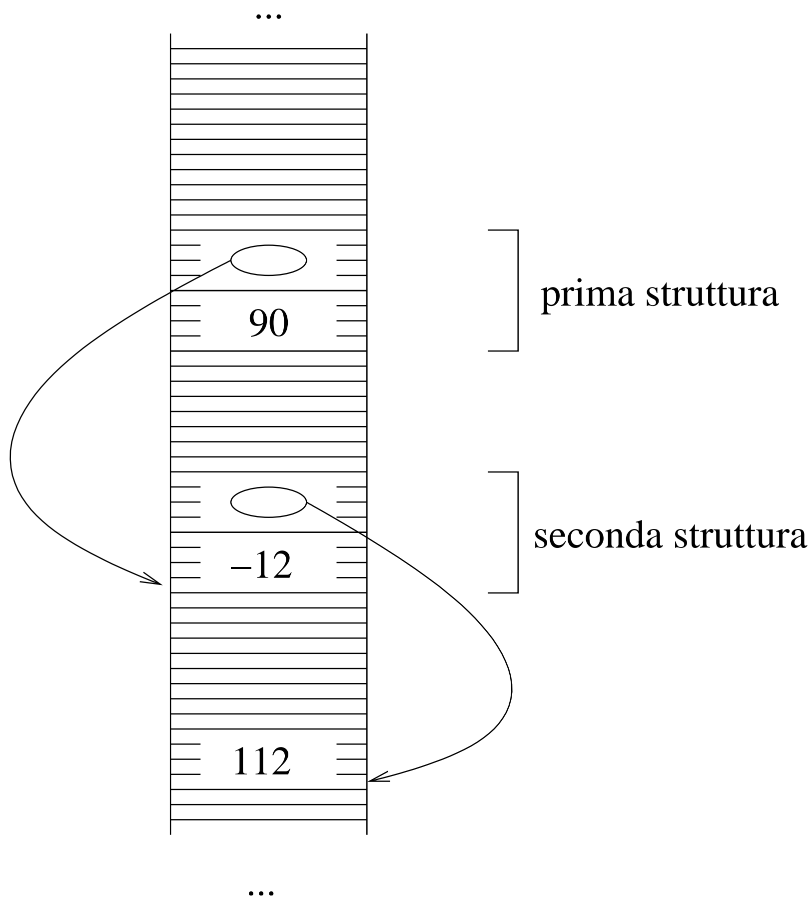
In altre parole, la variabile `x` è una struttura con due campi: è composta da una variabile intera `x.primo`, e da un puntatore `x.secondo`. Ricordiamo che un puntatore è semplicemente un numero, che rappresenta un indirizzo di memoria. In questo caso, la zona di memoria indicata da questo puntatore va interpretata come un intero.

Naturalmente, non esiste nessuna garanzia, a priori, che questa zona contenga effettivamente il secondo numero della sequenza. Allocare la memoria e metterci il secondo intero è compito del programmatore.

Il punto fondamentale da tenere in considerazione è che vogliamo che la sequenza sia rappresentabile come una unica variabile. Nel caso di sopra, è vero che la variabile `x` non contiene il secondo intero, però è anche vero che da `x` siamo comunque in grado di risalire sia al primo che al secondo elemento della sequenza. Possiamo quindi, per esempio, passare `x` a una funzione, e questa funzione è in grado di capire quali sono gli elementi della sequenza. Ad esempio, possiamo scrivere una funzione che riceve la sola variabile `x` e stampa gli elementi della sequenza. In generale, la regola che dobbiamo rispettare nel definire un meccanismo di rappresentazione delle liste è:

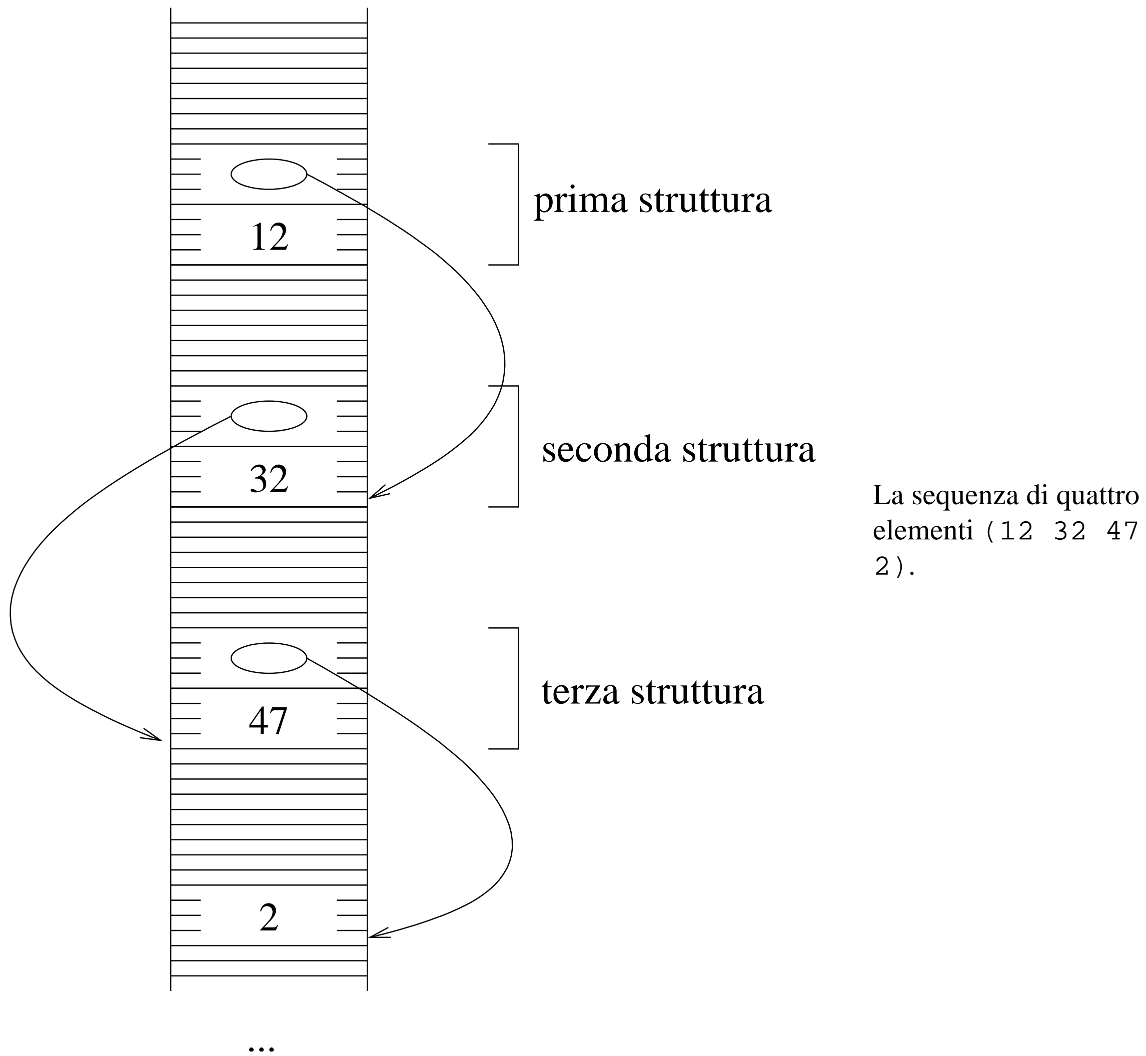
> ogni lista deve essere rappresentabile usando una sola variabile, ossia, dato il valore di questa variabile deve essere possibile capire quali sono tutti gli elementi della sequenza.

Possiamo pensare di estendere questo meccanismo su una sequenza di tre elementi. Vediamo un modo possibile di rappresentare in memoria una sequenza di tre elementi (non ci preoccupiamo, per ora, della definizione dei tipi). Possiamo organizzare i dati in questo modo: usiamo una struttura il cui primo campo è il primo elemento della sequenza, mentre il secondo campo è un puntatore, che contiene quindi un indirizzo. In particolare, questo indirizzo indica la posizione iniziale in cui è memorizzata una seconda struttura, il cui primo campo è il secondo elemento della sequenza, mentre il secondo campo è un puntatore al terzo elemento della sequenza. Si veda di seguito la rappresentazione grafica.



Questo è il modo in cui viene rappresentata la sequenza `(90 -12 112)`: il primo numero viene messo come primo campo della prima struttura, ecc. Ogni struttura contiene come secondo campo l'indirizzo della seguente.

Per sequenze di quattro elementi possiamo usare la seguente diposizione.



La sequenza di quattro elementi `(12 32 47 2)`.

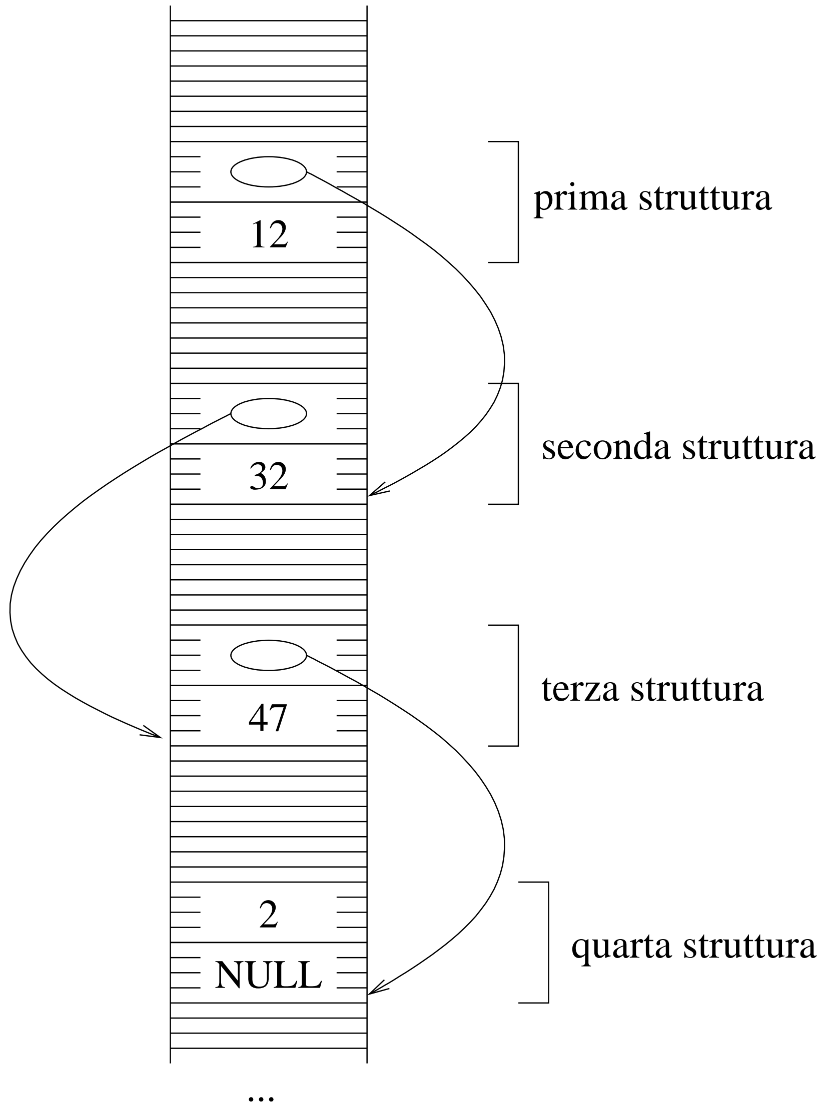
Le figure di sopra danno una possibile rappresentazione in memoria di alcune liste. Per poter realizzare queste disposizioni in memoria, occorre dare delle definizioni di tipo. Diamo per esempio la definizione per la sequenza di quattro elementi:

```
struct Terza {
  int terzoelem;
  int *quartoelem;
};

struct Seconda {
  int secondoelem;
  struct Terza *terzoelem;
};

struct SeqQuattro {
  int primoelem;
  struct Seconda *secondoelem;
};
```

Quello che è importante capire è che

1. la definizione di tipo richiede una definizione per ogni elemento della sequenza;
2. le definizioni sono quasi tutte uguali: ogni struttura è composta da un intero e da un puntatore.

Consideriamo ora la seguente variante della rappresentazione di sopra: l'ultimo elemento della lista, invece di essere un intero, è una struttura composta da un intero e da un puntatore di valore `NULL`. Questa è una variante lecita:



Osserviamo ora che ogni struttura è composta da un intero e da un puntatore, e questo vale ora anche per l'ultima struttura, che differisce dalle altre solo per il fatto che il valore del puntatore è `NULL`. Quindi, ognuna delle strutture è composta da un intero e da un puntatore a qualcosa. In particolare, osserviamo che la zona di memoria che inizia dall'indirizzo scritto nel puntatore è ancora una struttura identica alla precedente. Possiamo quindi definire tutte le strutture nello stesso identico modo:

```
struct Sequenza {
  int valore;
  struct Sequenza *next;
};
```

In altre parole, sappiamo che tutte le strutture devono essere composte da un intero e da un puntatore, e questo giustifica il fatto che la definizione di tipo per la struttura è composta da due campi, il primo dei quali è un intero, e il secondo è un puntatore. Per capire di che tipo deve essere il puntatore, dobbiamo semplicemente vedere cosa contiene la zona di memoria che inizia dall'indirizzo che il puntatore contiene. In questo caso, la zona contiene ancora una struttura di tipo `struct Sequenza`, quindi dichiariamo che il puntatore è un puntatore a una struttura `struct Sequenza`.

Lista vuota

Questa definizione di tipo va bene per sequenze di un qualsiasi numero di elementi, ma c'è un caso su cui non funziona: la lista vuota. Infatti, una lista è una sequenza di interi, che può anche essere composta da nessun numero. Una sequenza che non contiene nessun elemento si denota con `( )`. Un esempio del perchè questo caso è cosí importante: supponiamo di voler memorizzare dei numeri che vengono forniti dall'utente in una lista: all'inizio dell'esecuzione, quando l'utente non ha ancora inserito nessun numero, la lista deve risultare vuota.

Esiste un modo molto semplice per estendere la definizione di sopra in modo tale che comprende anche il caso di lista vuota: definiamo una lista non come una struttura a due campi, ma come un *puntatore alla struttura*. In altre parole, diamo un nome di comodo alla struttura, e poi definiamo la lista come un puntatore alla struttura:

```
struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;
```

Per prima cosa, verifichiamo facilmente che, in questo modo, possiamo rappresentare sia liste vuote che liste con elementi. Se dichiariamo `l` come una variabile di tipo `TipoLista`, allora questa variabile è di tipo puntatore, ossia contiene l'indirizzo di una zona di memoria che contiene una struttura. Rappresentiamo la lista vuota mettendo il valore `NULL` nella variabile `l`. Se la lista non è vuota, allora la rappresentiamo usando la sequenza di strutture collegate, la cui prima struttura è quella all'indirizzo contenuto in `l`.

Resterebbe ora da dire perchè usiamo questo particolare meccanismo, e non altri, per rappresentare la lista vuota. Il motivo è che, in questo modo, si ha una certa coerenza di rappresentazione. In particolare, la lista intera è rappresentata da `l`, mentre la lista composta da tutti gli elementi tranne il primo è `(*l).next`. Questa uniformità semplifica molto il lavoro di programmazione, e permette la creazione di funzioni ricorsive sulle liste.

Tutto questo discorso spiega la definizione di tipo della lista. È però anche possibile partire dalla definizione di tipo, e vedere a cosa corrisponde in memoria. Nella prossima pagina, facciamo finta di non sapere come si è arrivati a questa definizione di tipo, e facciamo vedere come sia possibile rappresentare liste di interi.

Riepilogo:

1. possiamo rappresentare liste usando strutture composte da due campi, uno dei queli è un puntatore a un'altra struttura dello stesso tipo;
2. usiamo una struttura anche per rappresentare l'ultimo elemento della sequenza, anche se questo non ha una struttura che segue (per cui il puntatore non sarebbe necessario): in questo modo, si

riesce a dare una definizione di tipo abbastanza semplice;
3. per riuscire a rappresentare anche la lista vuota, usiamo un puntatore a struttura per rappresentare una lista (cioè non rappresentiamo una lista con una variabile struttura ma con una variabile puntatore a struttura).

Si vede chiaramente che questo meccanismo di rappresentazione soddisfa la regola che la lista deve essere rappresentata da un'unica variabile. Infatti, se passiamo il valore di `l` (il puntatore alla prima struttura) a una funzione, questa è per esempio in grado di capire quali sono tutti gli elementi della lista. Vedremo più avanti una funzione che stampa una lista a partire dal solo puntatore iniziale.

Dalla definizione di tipo alla rappresentazione in memoria

In questa pagina vediamo come si possano rappresentare delle sequenze di elementi usando la definizione di struttura data nella pagina precedente, ossia:

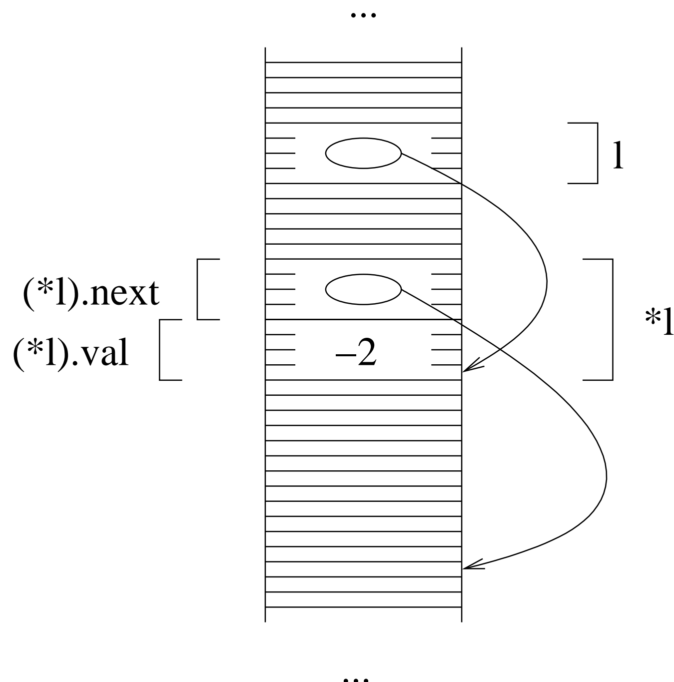
```
struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;
```

Quale è il significato di questa definizione? In C, le definizioni (come tutto il resto) vanno interpretate in senso letterale. Quando si dichiara una variabile di tipo `TipoLista`, per esempio

```
  TipoLista l;
```

questo equivale a dichiarare una variabile che contiene un puntatore, ossia un indirizzo di memoria. In particolare, la zona puntata (ossia quella che comincia a questo indirizzo) deve contenere una struttura di tipo `struct NodoLista`. Questa struttura è identificata, come al solito, da `*l`.

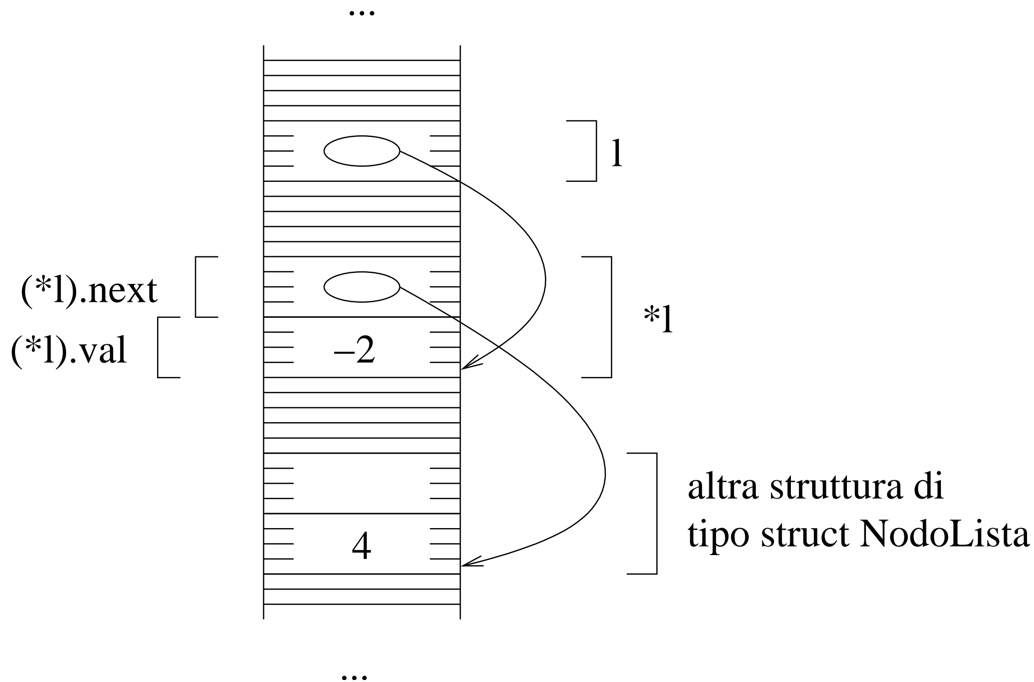
In base alla definizione di tipo, ogni struttura di tipo `struct NodoLista` è composta da due campi. Nel nostro caso, i due campi sono `(*l).val` e `(*l).next`. La prima è semplicemente una variabile intera, e quindi non ha nulla di complicato. La seconda variabile è un puntatore. In memoria, quello che succede è che la struttura occupa 4+4=8 byte, nell'ipotesi che sia interi che puntatori occupino quattro byte ciascuno.



Questa figura si ottiene semplicemente interpretando in modo letterale le definizioni di tipo. Il campo puntatore della struttura è semplicemente un numero, per cui sappiamo quanti byte occupa in memoria (anche senza sapere cosa c'è a partire dall'indirizzo scritto dentro). D'altra parte, la definizione di tipo ci dice anche che questo puntatore `(*l).next` è di tipo `struct NodoLista *`, e quindi la zona di memoria che inizia dall'indirizzo in `(*l).next` contiene una struttura di tipo `struct NodoLista`. In altre parole, `(*l).next` individua una zona di memoria in cui si trova una struttura analoga a `*l`.

Si noti che è compito del programmatore allocare le zone di memoria, ossia riservarle per contenere delle strutture. In altre parole, quando si definisce una variabile di tipo lista con `TipoLista l`, l'unica zona di memoria che viene allocata è quella per la variabile `l`. È poi compito del programmatore allocare anche la memoria per la struttura. Quello che il linguaggio garantisce è solo che questa zona verrà interpretata come una struttura di tipo `struct NodoLista`.

Lo stesso discorso vale per la zona di memoria puntata da `(*l).next`. Se questa zona di memoria è stata allocata correttamente, allora è una struttura di tipo `struct NodoLista`. Quindi, abbiamo in memoria:



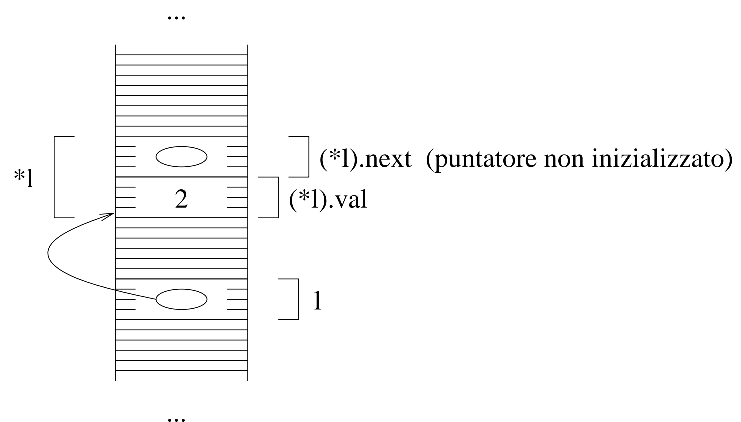
La struttura è fatta nello stesso modo della precedente, per cui è composta da un intero e da un puntatore. Questo puntatore indica una zona di memoria in cui inizia una struttura dello stesso tipo. D'altra parte, il puntatore può anche valere `NULL`.

Questo meccanismo consente di rappresentare sequenze di interi di lunghezza generica. Infatti, mettiamo il primo intero nella prima struttura, il secondo nella seconda struttura, ecc. Ogni struttura contiene anche un puntatore, che dà l'indirizzo della successiva struttura. Quando arriviamo all'ultimo elemento, lo mettiamo nell'ultima struttura, e mettiamo `NULL` come puntatore.

Supponiamo di voler per esempio rappresentare la sequenza `(2 -9 1)`. Per prima cosa, allochiamo la prima struttura, e ci mettiamo il primo elemento. Per allocare una struttura, usiamo come al solito la funzione `malloc`. Per poter poi accedere alla struttura (per esempio, per scriverci dentro il primo elemento della lista) ci serve sapere il suo indirizzo. QUesto è dato dal valore di ritorno della funzione `malloc`. Dato che `l` deve contenere questo indirizzo, facciamo `l=malloc(...)` che alloca la prima struttura e mette il suo indirizzo in `l`.

Codice

```
TipoLista l;

l=malloc(sizeof(struct NodoLista));
(*l).val=2;
```



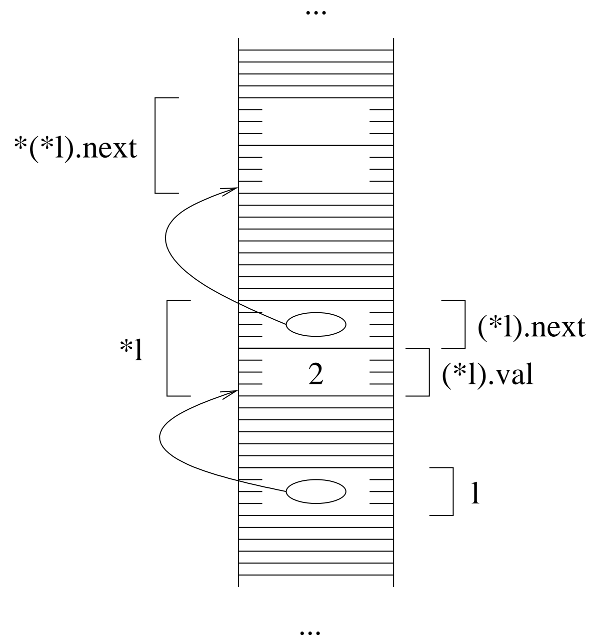
Nella rappresentazione con frecciae, dato che ho messo in `l` l'indirizzo della struttura allocata, ho una freccia da `l` alla struttura. Le parentesi di `(*l).val` sono necessarie perchè, secondo le regole di precedenza degli operatori del C, la espressione `*l.next` viene interpretata come `*(l.next)`. Questo produce un errore perchè `l` non è una struttura, per cui `l.next` non è una espressione valida.

Dal momento che dobbiamo ancora memorizzare degli elementi, ci serve una seconda struttura, il cui indirizzo va messo in `(*l).next`. La funzione `malloc` alloca una struttura e restituisce il suo indirizzo, che va messo in `(*l).next`.

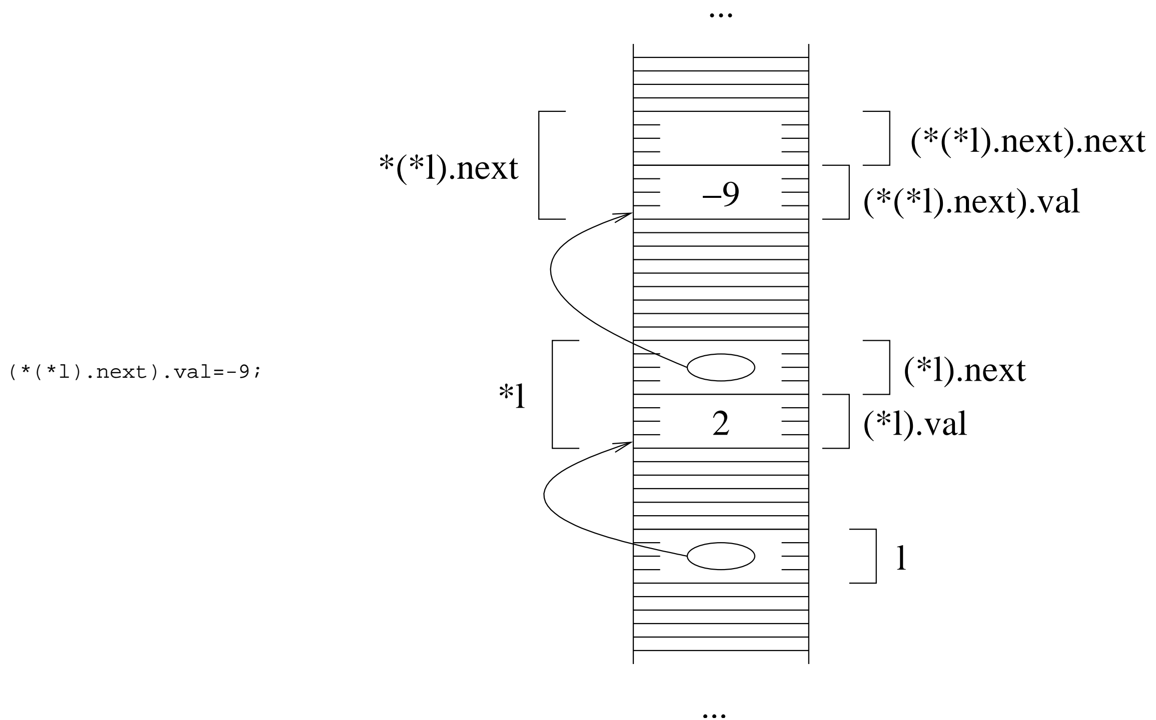
Codice

Memoria

```
(*l).next=malloc(sizeof(struct Sequenza));
```



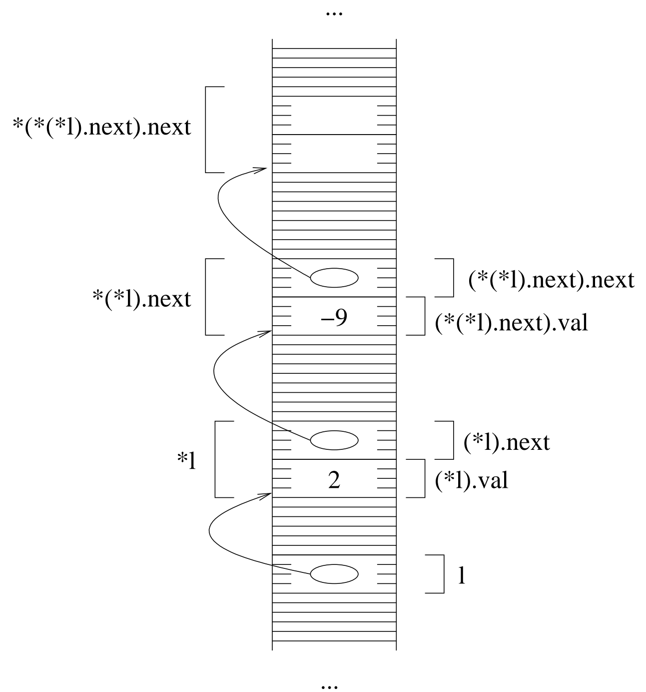
A questo punto, abbiamo due strutture in memorie, e in particolare `(*l).next` contiene l'indirizzo della seconda. Da questo segue che `*(*l).next` è esattamente la seconda struttura. Quindi, per mettere il secondo intero nella seconda struttura, facciamo:



Questo significa: dato che `(*l).next` è un puntatore a struttura, allora `*(*l).next` è una struttura, quindi è composta da due variabili, la prima della quali è l'intero `(*(*l).next).val` (le parentesi sono necessarie per una questione di precedenza degli operatori); la seconda variabile di questo struttura, ossia `(*(*l).next).next` è un puntatore. In particolare, è l'indirizzo iniziale di una struttura di tipo `struct NodoLista`. Per essere più precisi, è l'indirizzo della terza struttura, quella che contiene il terzo elemento della sequenza. Dal momento che la lista ha tre elementi, questa terza struttura deve esistere, e deve contenere il valore `1` nel campo intero. Per prima cosa, questa struttura va allocata:

Codice

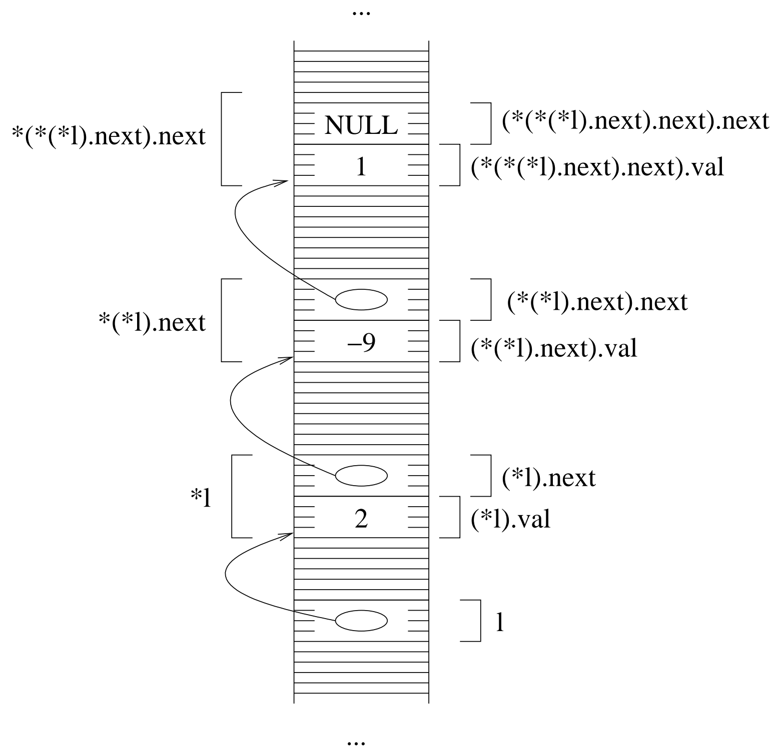
```
(*(*l).next).next=malloc(sizeof(struct Sequenza));
```



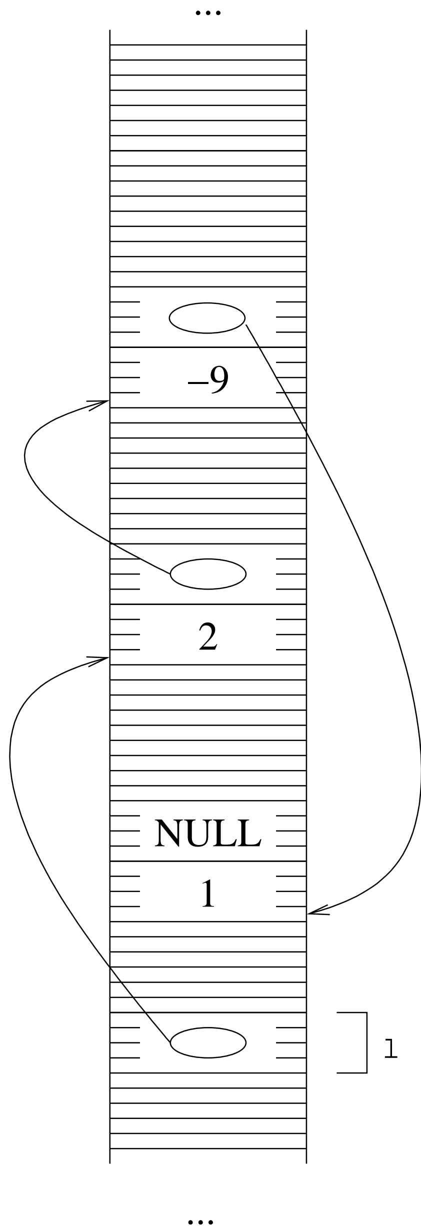
Ora possiamo dire che `*(*(*l).next).next` è una struttura, e in particolare è la terza struttura. Quindi, il numero intero che contiene deve valere `-1`; dal momento che non ci sono altri elementi nella lista, mettiamo a `NULL` il valore puntatore di questa struttura per indicare che la sequenza finisce qui.

Codice

```
(*(*(*l).next).next).val=-1;
(*(*(*l).next).next).next=NULL;
```



Nota: nelle figure di sopra le strutture sono state disegnate l’una sopra l’altra. In realtà non esiste nessuna garanzia che la allocazione di memoria faccia una cosa del genere. Quello che si potrebbe avere allla fine della costruzione è anche una cosa del genere:



Scansione di una lista

Per scansione di una lista si intende la ripetizione di una stessa operazione su tutti gli elementi della lista, a partire dal primo e andando avanti in ordine. Vediamo ora un semplice caso di scansione: quello della stampa di una lista. Dal momento che vogliamo ripetere la stampa su tutti gli elementi della lista, in ordine, si tratta appunto di fare una scansione della lista.

Vediamo prima un modo che si ottiene facilmente dalla definizione. Supponiamo quindi che `l` sia una variabile di tipo `TipoLista`. Per definizione, si tratta di un puntatore a una struttura `*l` a due campi: il primo elemento della lista è il campo `val` di questa struttura, ossia `(*l).val`. Il secondo elemento della sequenza si ottiene considerando che la seconda struttura è `*(*l).next`, e quindi il secondo elemento è `(*(*l).next).val`. In modo simile, il terzo elemento è `(*(*(*l).next).next).val`. Possiamo quindi stampare i primi tre elementi della sequenza con il programma listano.c riportato qui sotto.

```
/*
  Stampa dei primi tre elementi di una lista.
*/

#include<stdlib.h>

/* definizione di tipo */

struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;


/*
  main
*/

int main () {
  TipoLista l;


                /* creazione lista (2 -9 1) */

  l=malloc(sizeof(struct NodoLista));

  (*l).val=2;
  (*l).next=malloc(sizeof(struct NodoLista));

  (*(*l).next).val=-9;
  (*(*l).next).next=malloc(sizeof(struct NodoLista));

  (*(*(*l).next).next).val=1;
  (*(*(*l).next).next).next=NULL;


                /* stampa della lista l */

  printf("%d ", (*l).val);
  printf("%d ", (*(*l).next).val);
  printf("%d ", (*(*(*l).next).next).val);

  printf("\n");

  return 0;
}
```

Stampa liste di lunghezza generica

Il programma di sopra va bene se la lista è composta da tre elementi. Per stampare una lista più lunga servirebbe un'altra chiamata alla funzione `printf` per stampare il quarto elemento, ecc. Da notare che le espressioni che danno l'elemento della lista a partire dal puntatore iniziale diventano sempre più complicate mano a mano che si avanza nella lista (si provi a scrivere la espressione che dà il decimo elemento).

Vediamo ora un meccanismo più semplice per stampare tutti gli elementi in ordine. L'idea di base è che, dato che tutte le strutture della lista sono analoghe, si può realizzare un ciclo. Per capire quando uscire da questo ciclo, sfruttiamo il fatto che l'ultima struttura ha `NULL` nel campo puntatore.

Cominciamo a modificare il codice del programma di sopra per evitare di ripetere le espressioni lunghe. Dato che la espressione `(*l).next` compare molto spesso, la memorizziamo in una variabile. Il tipo deve coincidere con il tipo di `(*l).next`, per cui questa nuova variabile deve essere di tipo puntatore a struttura `struct NodoLista`, oppure di tipo `TipoLista`, che è eqivalente. Abbiamo quindi una dichiarazione `TipoLista s;` e poi la istruzione:

```
printf("%d ", (*l).val);
s=(*l).next;
printf("%d ", (*s).val);
```

Questo codice si ottiene semplicemente inserendo l'istruzione di assegnazione su `s`, e sostituendo a `(*l).next` la variabile `s` nella seconda istruzione si stampa. In altre parole, dal momento che questa variabile `s` contiene una copia dell'indirizzo della seconda struttura della sequenza, il secondo elemento della lista è `(*s).val`.

La successiva struttura della sequenza (la terza) ha indirizzo `(*s).next`. Dato che la seconda struttura non ci serve più (è stata già stampata), possiamo direttamente memorizzare l'indirizzo della terza struttura nella variabile `s`, e da questa stampare il campo `val` della struttura:

```
s=(*s).next;
printf("%d ", (*s).val);
```

A questo punto, possiamo di nuovo stampare `(*s).val`, e passare alla successiva struttura con `s=(*s).next;`. Questo si conclude quando si arriva a una struttura il cui campo `next` vale `NULL`.

Il punto fondamentale è che possiamo usare una variabile puntatore `s` per memorizzare l'indirizzo della struttura che stiamo guardando, e ogni volta facciamo gli stessi passi:

1. se `s` vale `NULL`, allora la lista è finita;
2. stampa `(*s).val`
3. copiamo in `s` il valore di `(*s).next`

Possiamo seguire questa strategia anche per la stampa del primo elemento della lista: basta che inizialmente copiamo l'indirizzo della prima struttura in `s`. Il ciclo di scansione della sequenza diventa quindi:

```
s=l;
while(s!=NULL) {
  printf("%d ", (*s).val);
  s=(*s).next;
}

printf("\n");
```

Riportiamo qui sotto il codice completo del programma listamain.c che crea un sequenza di tre elementi e li stampa. Da notare come la parte di stampa sia totalmente indipendente dalla lunghezza della sequenza. Inoltre, rimpiazzando la istruzione di stampa `printf("%d ", (*s).val)` con qualche altra istruzione, è possibile fare una qualsiasi operazione sugli elementi della sequenza.

```
/*
  Definizione di lista, creazione, stampa.
*/

#include<stdlib.h>

struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;

/*
  main
*/

int main() {
  TipoLista l;
  TipoLista s;


                /* creazione della lista (2 -9 1) */

  l=malloc(sizeof(struct NodoLista));

  (*l).val=2;
  (*l).next=malloc(sizeof(struct NodoLista));

  (*(*l).next).val=-9;
  (*(*l).next).next=malloc(sizeof(struct NodoLista));

  (*(*(*l).next).next).val=1;
  (*(*(*l).next).next).next=NULL;


                /* stampa la lista */

  s=l;
  while(s!=NULL) {
    printf("%d ", (*s).val);
    s=(*s).next;
  }

  printf("\n");

  return 0;
}
```

Stampa sequenza usando una funzione

Il codice di stampa della sequenza, come abbiamo detto, funziona con liste di qualsiasi lunghezza. È quindi utile realizzare una funzione che stampa gli elementi di una lista. Il programma seguente lista.c definisce e usa una funzione di stampa di una sequenza di interi.

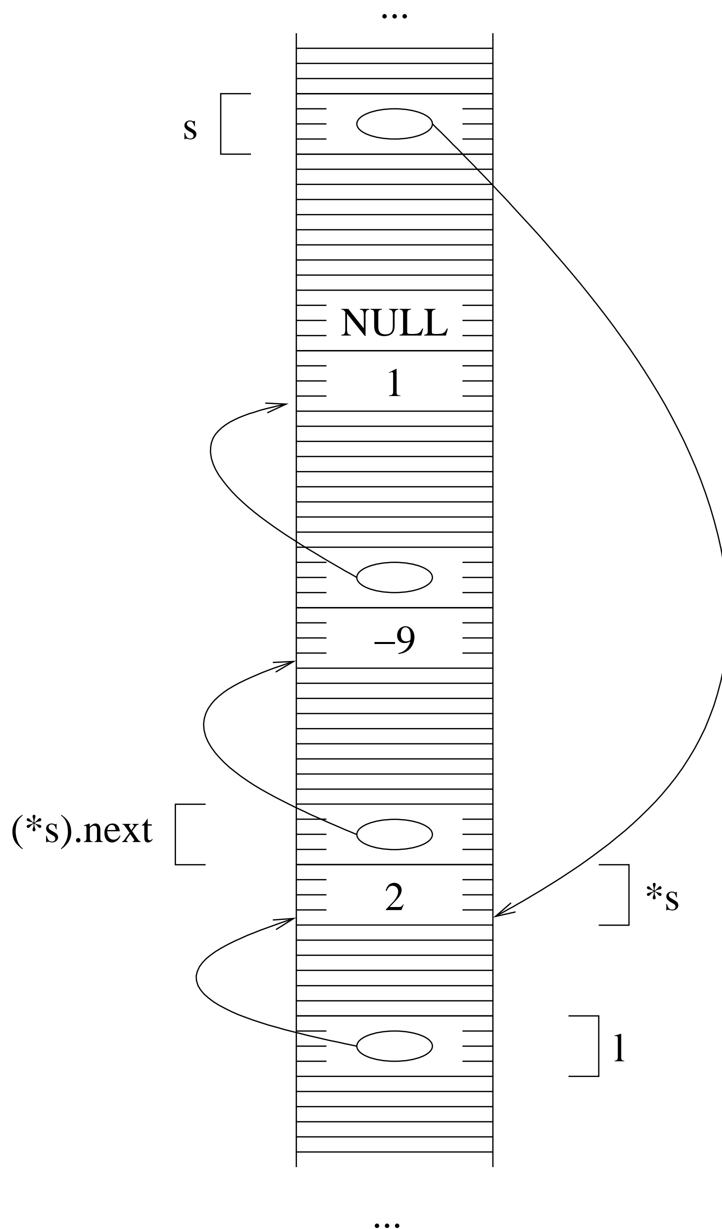
```
/*
  Definizione di lista, creazione, stampa.
*/

#include<stdlib.h>
```

```
struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;

/*
  stampa di una lista di lunghezza generica
*/

void StampaLista(TipoLista l) {
  TipoLista s;

  s=l;
  while(s!=NULL) {
    printf("%d ", (*s).val);
    s=(*s).next;
  }

  printf("\n");
}

/*
  main
*/

int main() {
  TipoLista l;


                /* creazione della lista (2 -9 1) */

  l=malloc(sizeof(struct NodoLista));

  (*l).val=2;
  (*l).next=malloc(sizeof(struct NodoLista));

  (*(*l).next).val=-9;
  (*(*l).next).next=malloc(sizeof(struct NodoLista));

  (*(*(*l).next).next).val=1;
  (*(*(*l).next).next).next=NULL;


                /* stampa la lista */

  StampaLista(l);

  return 0;
}
```

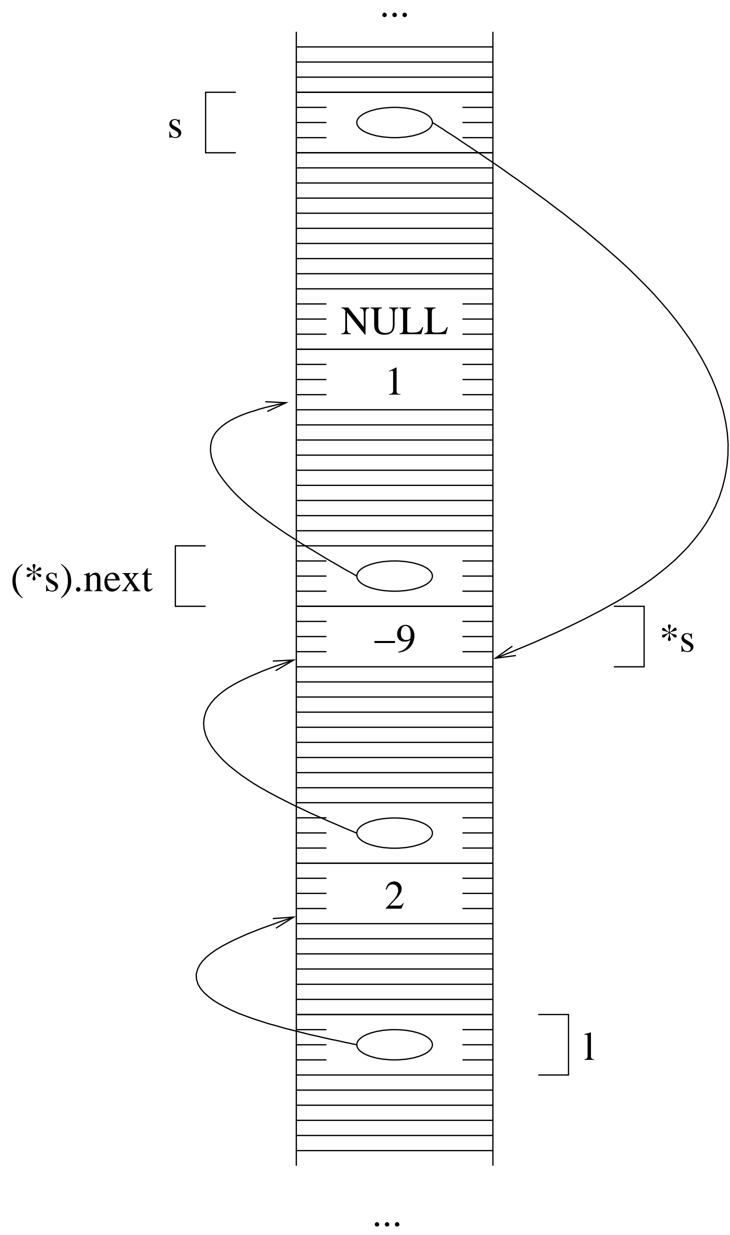
Una osservazione importante da fare è che la funzione riceve come parametro il puntatore alla prima struttura e basta. A partire dalla prima struttura è possibile capire quali sono i successivi elementi. Questo indica che è corretto dire che una variabile di tipo `TipoLista` rappresenta una lista di interi: la sequenza definita da una variabile di questo tipo è data dalla sequenza di interi che si incontrano seguendo i puntatori da una struttura all'altra.

Vediamo di seguito la evoluzione della memoria quando si esegue il programma di sopra. Quando si assegna a `s` il valore di `l` quello che succede è che queste il valore di `l` viene copiato in `s`. Quindi, l'indirizzo che è individuato da `l` ora è anche l'indirizzo di memoria individuato da `s`.

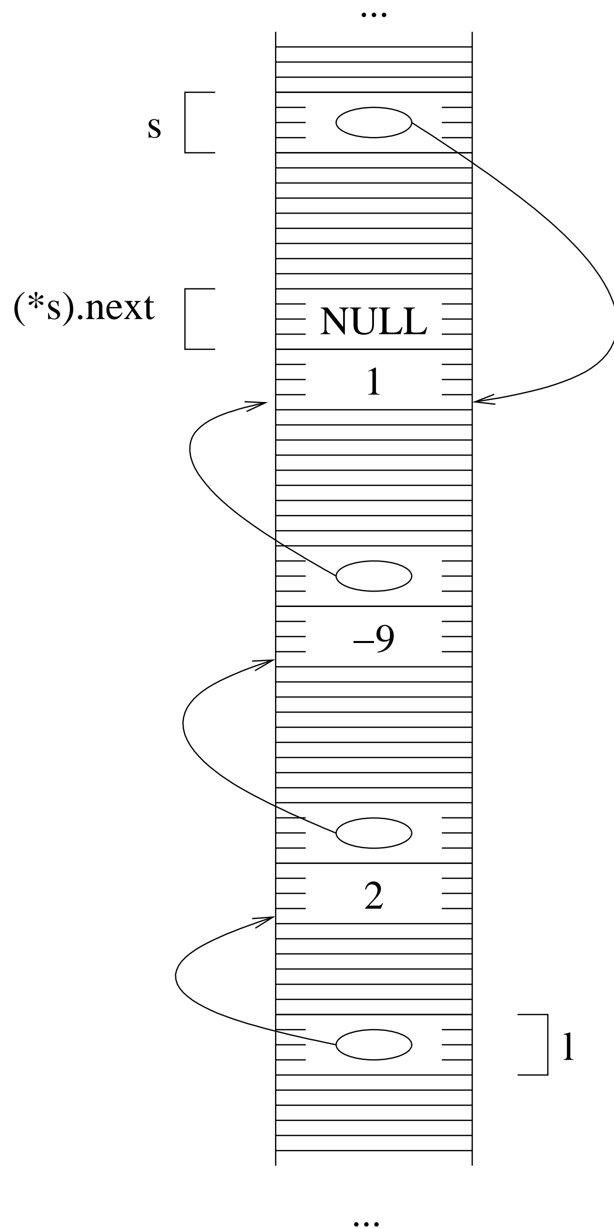
Nella rappresentazione con freccie, possiamo dire che *la freccia di* `s`*, dopo l'assegnamento* `s=l`*, ha la punta nella stessa posizione delle punta della freccia di* `l`. Se quindi eseguiamo la prima istruzione `s=l`, quello che otteniamo è che la freccia di `s` va alla locazione iniziale della prima struttura.



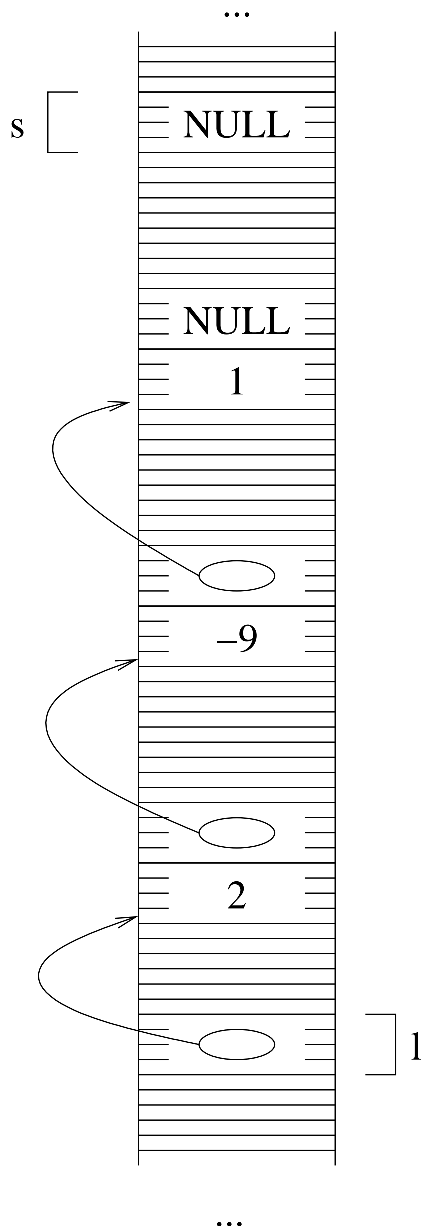
La stampa di `(*s).val` produce effettivamente la stampa del primo elemento della lista, che è `2`. Quando si esegue l'istruzione `s=(*s).next`, quello che succede è che la freccia di `s` deve puntare alla stessa locazione della freccia di `(*s).next`, quindi, guardando il disegno di sopra, è chiaro che si arriva alla situazione qui sotto:



Si stampa `(*s).val`, che è `-9`, ossia effettivamente il secondo elemento della lista. Si ripete ancora la assegnazione `s=(*s).next`. Si noti che la struttura `*s`, e quindi la variabile `(*s).next`, è diversa da quello che era in precedenza. Quello che si ottiene è che le due freccie puntano allo stesso indirizzo.



Si stampa `(*s).val`, che vale `-1`, e che è il terzo elemento della lista. Si esegue nuovamente `s=(*s).next`, ma in questo momento `(*s).next` vale `NULL`. Quindi, nella variabile `s` si mette `NULL`.



A questo punto la condizione del ciclo while è falsa, dal momento che `s` non è più diverso da `NULL`. Quindi, si esce dal ciclo. Se si va a vedere le stampe che sono state fatte, si nota come in effetti siano stati stampati gli elementi della lista, in ordine. È chiaro che questa funzione va bene per liste di lunghezza generica.

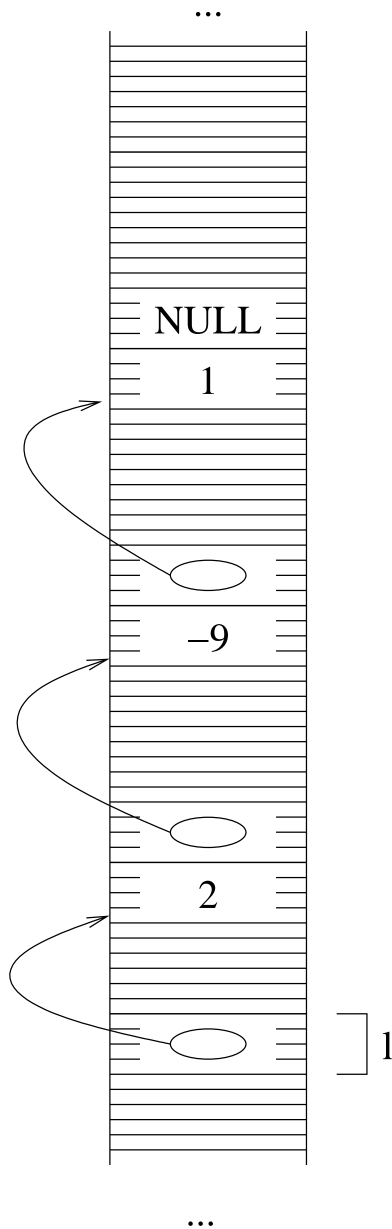
Costruzione liste dal fondo

Il metodo che abbiamo usato nelle pagine precedenti per costruire una lista è basato sull'uso di sequenze di tipo `*(*(*l).next).next).next`. Questo chiaramente si può fare solo se il numero di elementi di cui è fatta una lista è fissato a priori, e soprattutto se è un numero piccolo.

La costruzione della lista fatta in questo modo avviene in modo sequenziale, ossia allocando prima la prima struttura, poi la seconda, ecc. In questa pagina vediamo come sia molto più facile costruire una lista partendo dal fondo, ossia iniziando ad allocare l'ultima struttura, poi la penultima, ecc. Questo ci consente di realizzare dei cicli per costruire delle liste (si possono fare cicli anche per la costruzione in

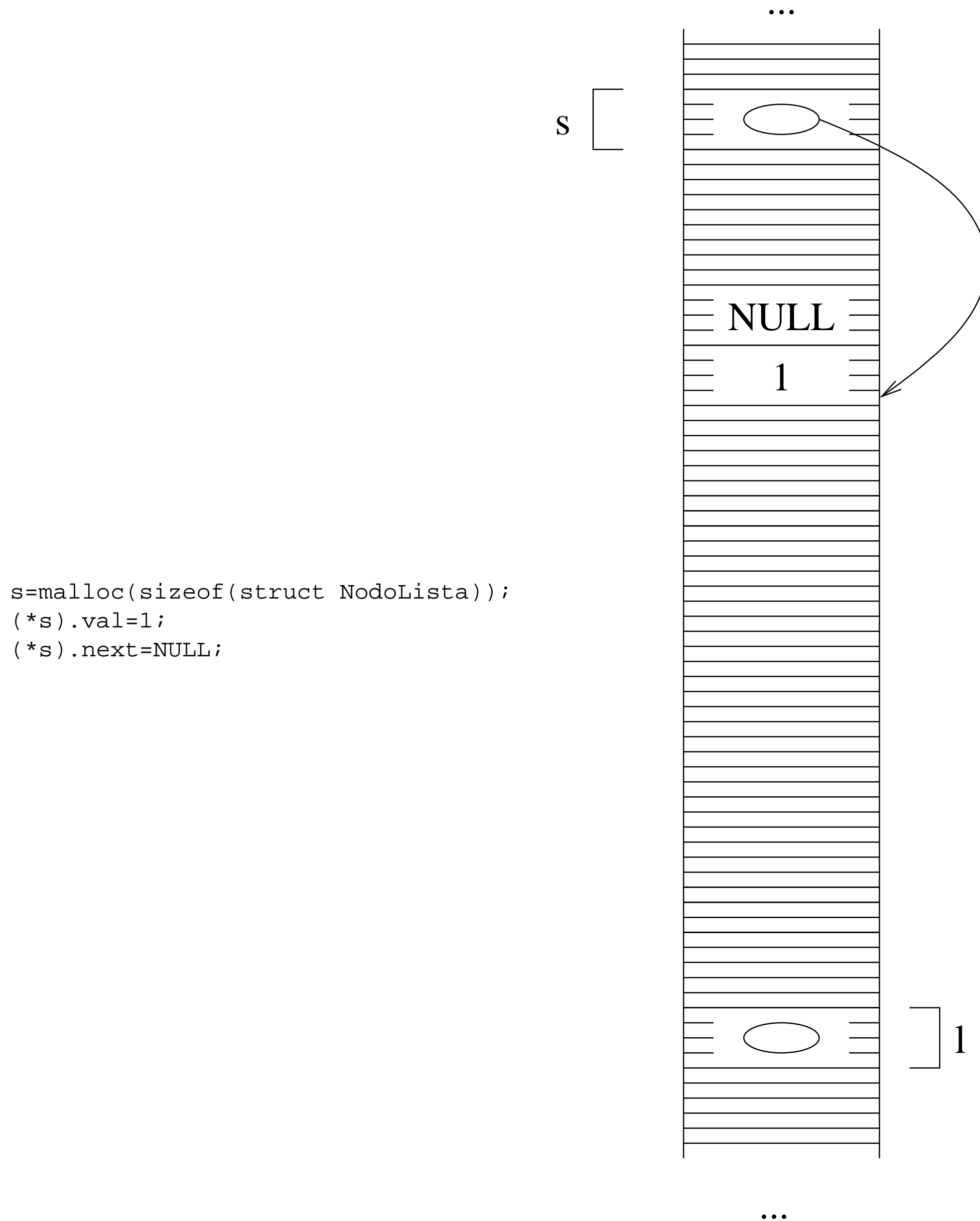
avanti, ma è più complicato).

Vediamo ora quali sono i passi da compiere per costruire una lista a partire dall'ultimo elemento. Consideriamo la lista `(2 -9 1)`. La situazione finale, a cui vogliamo arrivare, è qualcosa del genere:



Come al solito, non esiste nessuna garanzia che le strutture siano messe in questo ordine nella memoria.

Abbiamo detto che vogliamo iniziare dall'ultima struttura della lista, ossia quella che contiene `1`. Quello che dobbiamo fare è allocare la struttura e metterci dentro `1` e `NULL`. La allocazione di fa con `malloc`. Per usare la struttura allocata, occorre sapere il suo indirizzo, che il valore di ritorno della `malloc`. Mettiamo questo indirizzo in una variabile:

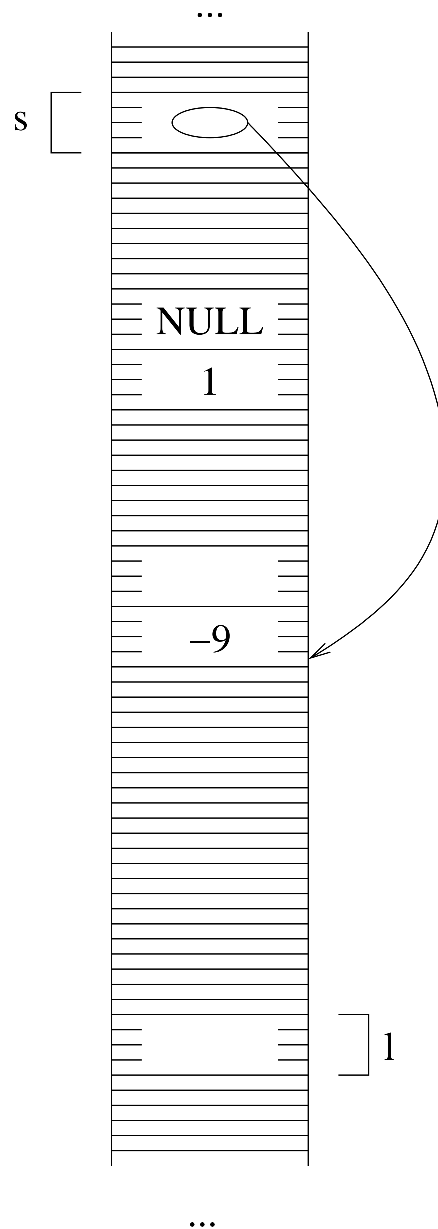



```
s=malloc(sizeof(struct NodoLista));
(*s).val=1;
(*s).next=NULL;
```

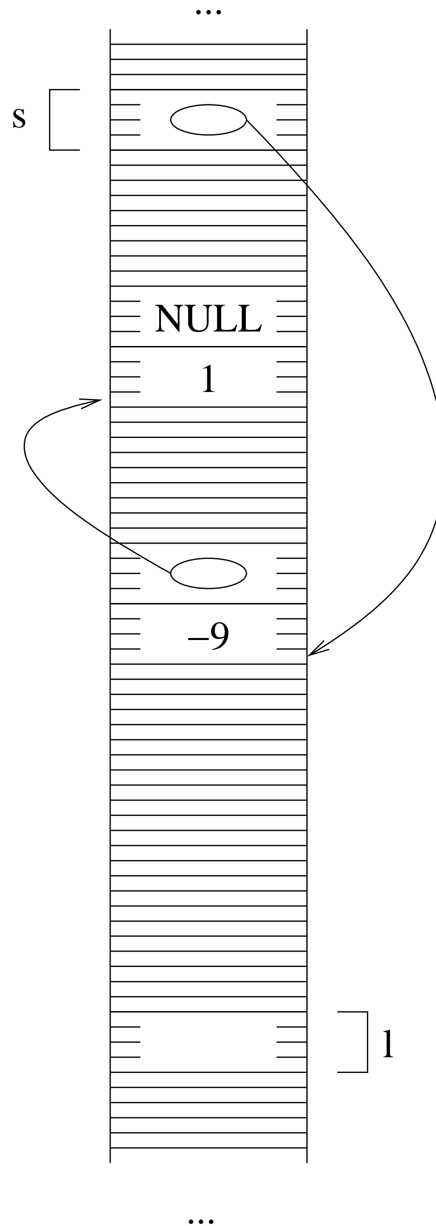
Questa struttura deve essere l'ultima struttura della lista, con i campi riempiti con i valori che deve avere (l'ultimo elemento della lista e `NULL`). Occorre ora allocare la penultima struttura della lista. Questo si può ancora fare con una istruzione `malloc`. Occorre però fare attenzione a dove si memorizza il valore di ritorno, ossia l'indirizzo della nuova struttura.

Supponiamo infatti di fare la stessa cosa di prima, ossia `s=malloc(sizeof(struct NodoLista));` e poi `(*s).val=-9;`. A questo punto, abbiamo il problema di cosa mettere nel campo `next` della nuova struttura. Questo è il campo `next` della penultima struttura, e quindi deve contenere l'indirizzo dell'ultima struttura. D'altra parte, l'indirizzo dell'ultima struttura stava in `s`, ma ora non c'è più: il suo valore è stato sovrascritto dall'indirizzo della penultima struttura quando abbiamo fatto l'ultima chiamata a `malloc`.

```
/* codice che NON FUNZIONA */
s=malloc(sizeof(struct NodoLista));
(*s).val=-9;
(*s).next=...?;
```



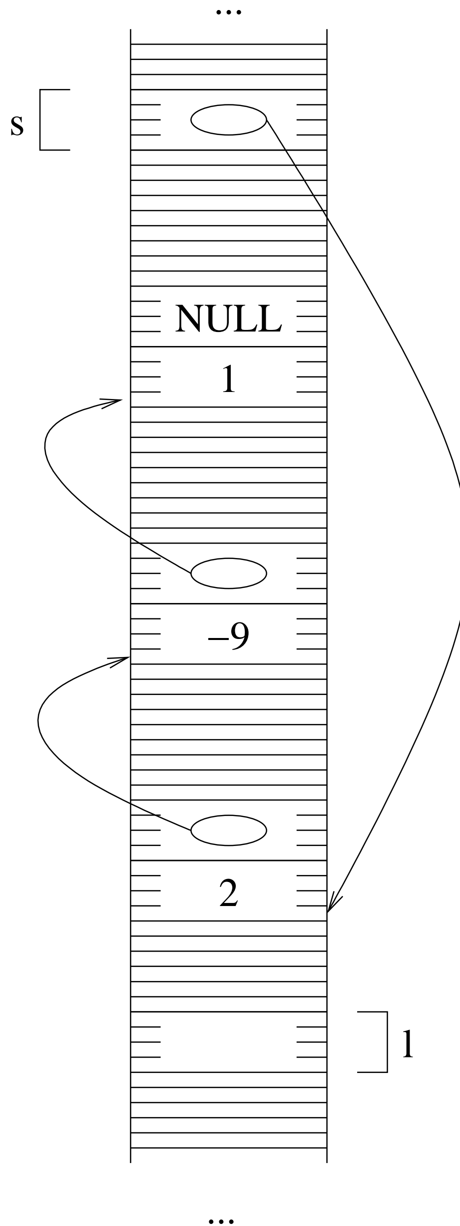
Tutto questo dice che, prima di fare `s=malloc(sizeof(struct NodoLista));` dobbiamo salvare il valore che stava prima in `s`, ossia l'indirizzo dell'ultima struttura, perchè questo valore poi ci occorre. Quindi facciamo:




```
t=s;
s=malloc(sizeof(struct NodoLista));
(*s).val=-9;
(*s).next=t;
```

In altre parole, in `t` (non disegnato) mettiamo il valore che aveva prima `s`, ossia l'indirizzo dell'ultima struttura. Poi facciamo la allocazione, mettendo in `s` l'indirizzo della struttura allocata, che deve diventare la penultima della lista. Possiamo ora riempire questa struttura: nel campo `val` ci mettiamo il valore del penultimo elemento della lista, ossia -9, e nel campo `next` ci mettiamo l'indirizzo dell'ultima struttura della lista. Questo indirizzo lo conosciamo, perchè lo abbiamo memorizzato in `t`.

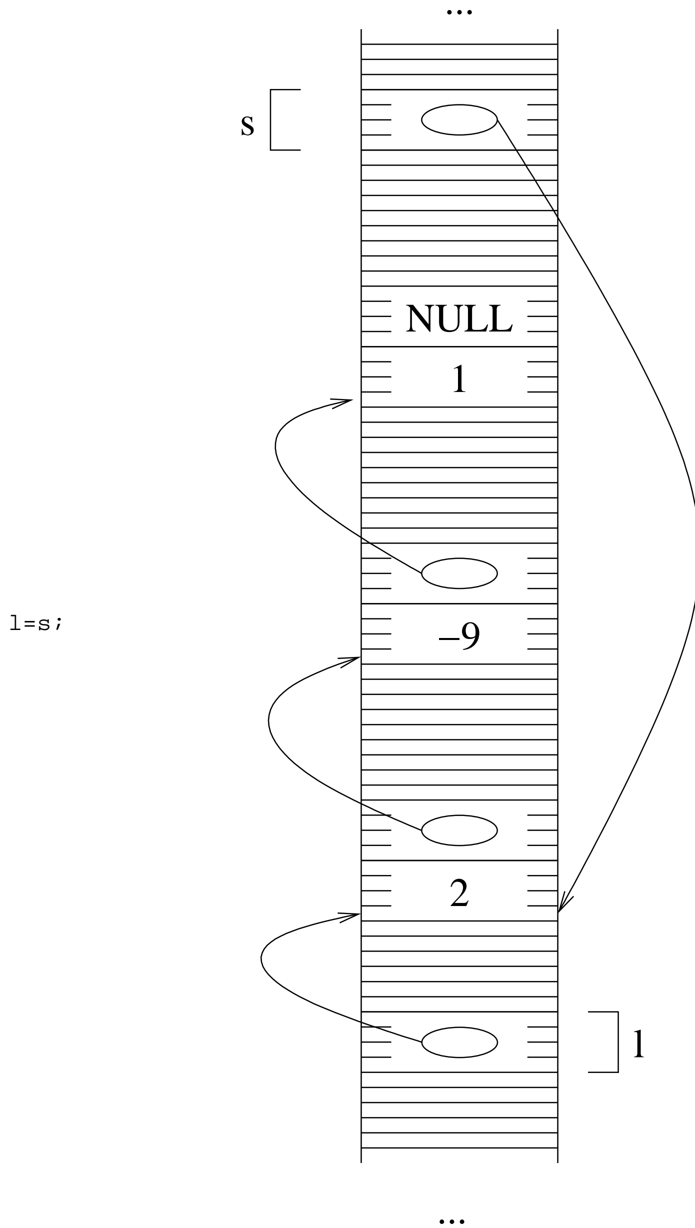
Dobbiamo ora fare l'allocazione della terz'ultima struttura della lista (nel nostro caso è anche la prima, dato che la lista ha solo tre elementi). Facciamo l'allocazione mettendo l'indirizzo della struttura creata in `s`, ma prima occorre salvare da qualche parte il valore di `s`, dal momento che l'indirizzo della penultima struttura ci serve per poter collegare ad essa la terz'ultima. Quindi, prima salviamo il valore di `s` in una variabile, poi facciamo la allocazione e riempiamo i campi.




```
t=s;
s=malloc(sizeof(struct NodoLista));
(*s).val=2;
(*s).next=t;
```

Si tratta di fare esattamente come prima: si salva il valore di `s`, che è l’indirizzo della struttura che è l’ultima a essere stata costruita, e in `s` ci si mette l’indirizzo ritornato dalla funzione `malloc`, ossia quello di una nuova struttura. Si mette nel campo `val` di questa struttura il valore opportuno, mentre nel campo puntatore ci si mette il valore di `t`, ossia l’indirizzo della struttura che era stata creata prima di questa.

L’ultima cosa che manca da fare è mettere in `l` l’indirizzo della prima struttura della sequenza, ossia di quella che contiene il primo elemento della lista. Questo indirizzo si trova già in `s`, quindi possiamo semplicemente fare:



Il codice completo è:

```
s=malloc(sizeof(struct NodoLista));
(*s).val=1;
(*s).next=NULL;

t=s;
s=malloc(sizeof(struct NodoLista));
(*s).val=-9;
(*s).next=t;

t=s;
s=malloc(sizeof(struct NodoLista));
(*s).val=2;
(*s).next=t;
```

Si tratta di ripetere sempre lo stesso blocco di istruzioni. In particolare, possiamo fare una modifica a questo codice per semplificarlo ulteriormente: mettiamo inizialmente in `s` il valore `NULL`

```
s=NULL;

t=s;
s=malloc(sizeof(struct NodoLista));
(*s).val=1;
(*s).next=t;

t=s;
s=malloc(sizeof(struct NodoLista));
(*s).val=-9;
(*s).next=t;

t=s;
s=malloc(sizeof(struct NodoLista));
(*s).val=2;
(*s).next=t;
```

Questo codice è esattamente uguale al precedente: `s` inizialmente contiene `NULL`, valore che viene poi copiato in `t`. La prima istruzione `(*s).next=t` è quindi equivalente a `(*s).next=NULL`, che è l'istruzione che stava nel codice precedente.

A questo punto è chiaro che si tratta di ripetere lo stesso blocco di quattro istruzioni per ogni struttura da creare. Ogni blocco crea una struttura, e la prima struttura creata è quella che contiene l'ultimo elemento della lista.

Vediamo ora un esempio in cui si può effettivamente realizzare una lista usando un ciclo. Supponiamo di volere la seguente lista:

```
(1 2 3 4 5 6 7 8 9 10 11 12 ... 1200)
```

Crearla scrivendo espressioni del tipo `(*(*(*(*l).next).next).next).next` è chiaramente impensabile. Possiamo però sfruttare il fatto che sappiamo perfettamente quanto vale l'ultimo elemento della lista, il penultimo, ecc. Possiamo quindi costruire prima la struttura che contiene 1200, poi quella che contiene 1199, ecc fino a quella che contiene il primo elemento della lista, ossia 1.

Si tratta quindi di fare un ciclo. Si noti che *il ciclo non va da 1 a 1200*. Infatti, fino a questo momento sappiamo solo come costruire la lista a partire dall'ultimo elemento. In altre parole, visto che dobbiamo prima creare la struttura che contiene 1200, poi quella che contiene 1199, poi 1198, ecc, è chiaro che dobbiamo *partire dal 1200 e arrivare a 1*. Questo è dovuto al fatto che costruiamo la lista nell'ordine opposto, dall'ultimo elemento al primo.

Possiamo quindi realizzare un ciclo che parte dall'ultimo elemento e arriva al primo. Ad ogni passo creaiamo una struttura e la colleghiamo al pezzo di lista che abbiamo costruito.

```
s=NULL;

for(i=1200; i>=1; i--) {
  t=s;
  s=malloc(sizeof(struct NodoLista));
  (*s).val=i;
  (*s).next=t;
}
l=s;
```

Nel caso ci fossero ancora dubbi, si provi a simulare su carta l'evoluzione della memoria nel caso in cui la lista da costruire abbia quattro elementi invece che milleduecento. Alla fine si vedrà che `l` contiene l'indirizzo di una struttura il cui campo `val` è 1, e seguendo i puntatori si leggono gli altri valori 2, 3 e 4.

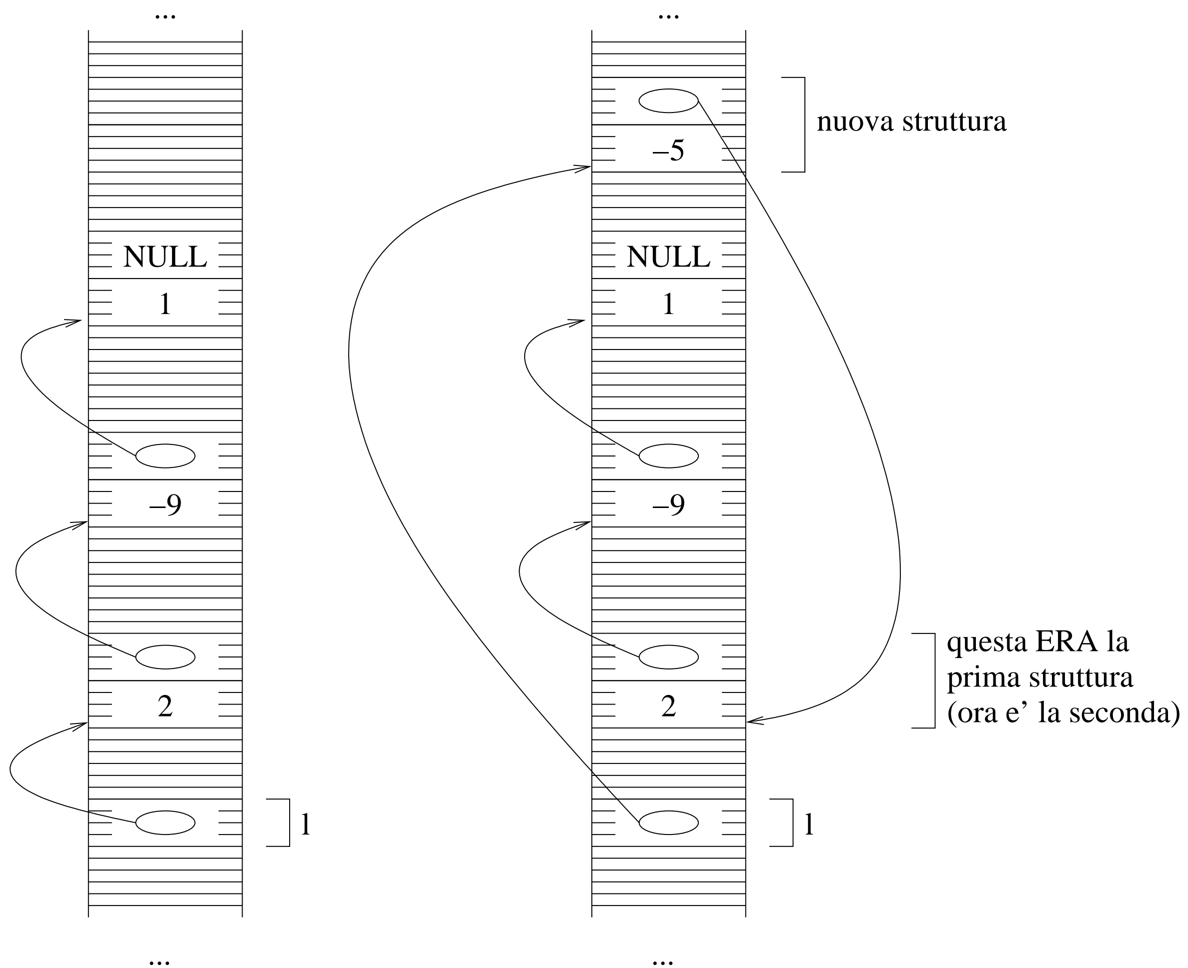
Il programma completo indietro.c che costruisce questa lista e la stampa, è qui sotto.

```
/*
  Crea una lista composta dai
numeri (1 2 3 4 5 ... 1200)
*/

#include<stdlib.h>
#include<stdio.h>

struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;

int main() {
  TipoLista l;
  struct NodoLista *s, *t;
  int i;
  struct NodoLista *q;

                        /* costruzione dal fondo: si parte allocando
                        l'ultima struttura, e poi si risale fino alla
                        prima, collegando ogni struttura con la successiva */

  s=NULL;

  for(i=1200; i>=1; i--) {
    t=s;
    s=malloc(sizeof(struct NodoLista));
    (*s).val=i;
    (*s).next=t;
  }
  l=s;


                        /* stampa la lista */

  q=l;
  while(q!=NULL) {
    printf("%d ", (*q).val);
    q=(*q).next;
  }

  return 0;
}
```

Aggiunta di un elemento in testa a una lista

Vediamo ora in che modo si può aggiungere un elemento in testa a una lista. La aggiunta in testa è la creazione di un nuovo elemento nella lista, in prima posizione. Nell'esempio della lista `(2 -9 1)`, se si aggiunge il numero `-5` in testa, la lista deve diventare `(-5 2 -9 1)`, cioè il nuovo elemento deve diventare il primo, e di seguito ci sono i vecchi elementi della lista.

In che modo si può eseguire questa modifica? Partiamo dallo stato della memoria: alla fine della aggiunta dobbiamo arrivare a una situazione in cui c'è una nuova struttura, il cui campo `next` punta al primo elemento della vecchia lista. Possiamo quindi fare questo paragone fra lo stato della memoria prima della aggiunta, e quello che deve essere dopo la aggiunta.



Cominciamo con la creazione della nuova struttura. A questo scopo, definiamo un puntatore e facciamo la allocazione della struttura puntata:

```
TipoLista t;

t=malloc(sizeof(struct NodoLista));
(*t).val=-5;
```

Questo crea la nuova struttura e ci mette il valore dell'intero. La struttura va ora collegata al resto della lista. Nella figura che dice come deve essere fatta la memoria alla fine, la freccia punta alla prima struttura della vecchia lista. Questa è la struttura a cui punta la freccia che parte da `l`. Dal momento

che assegnare un puntatore equivale a spostare la punta della sua freccia nella locazione dove si trova la punta dell'altra, possiamo ottenere questo risultato facendo:

```
(*t).next=l;
```

Resta ora soltanto una cosa da sistemare: la freccia che parte da `l` deve puntare alla nuova struttura. Dal momento che l'indirizzo di questa struttura si trova in `t`, possiamo semplicemente fare:

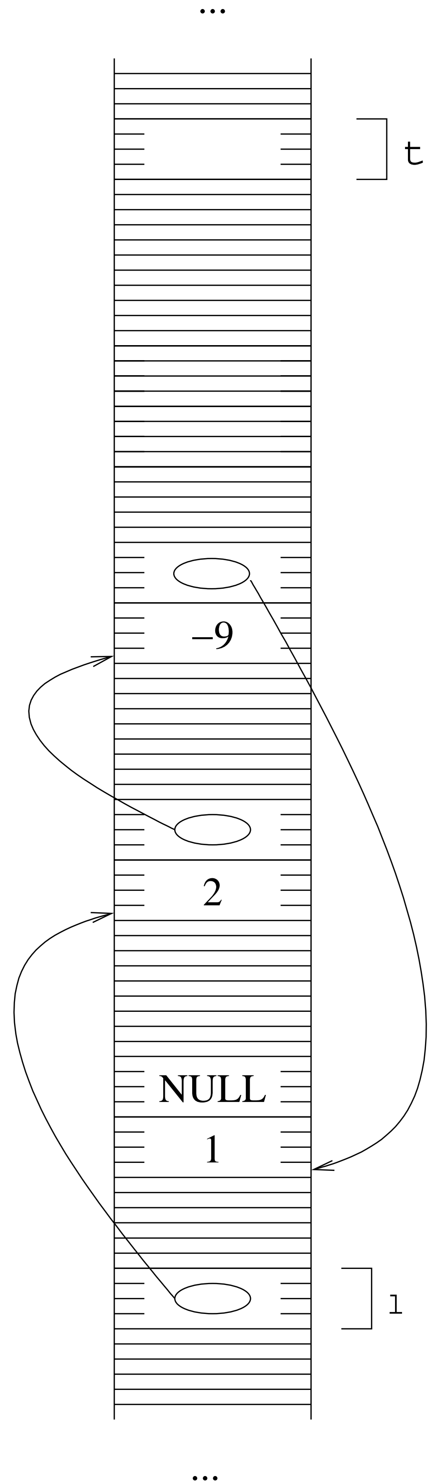
```
l=t;
```

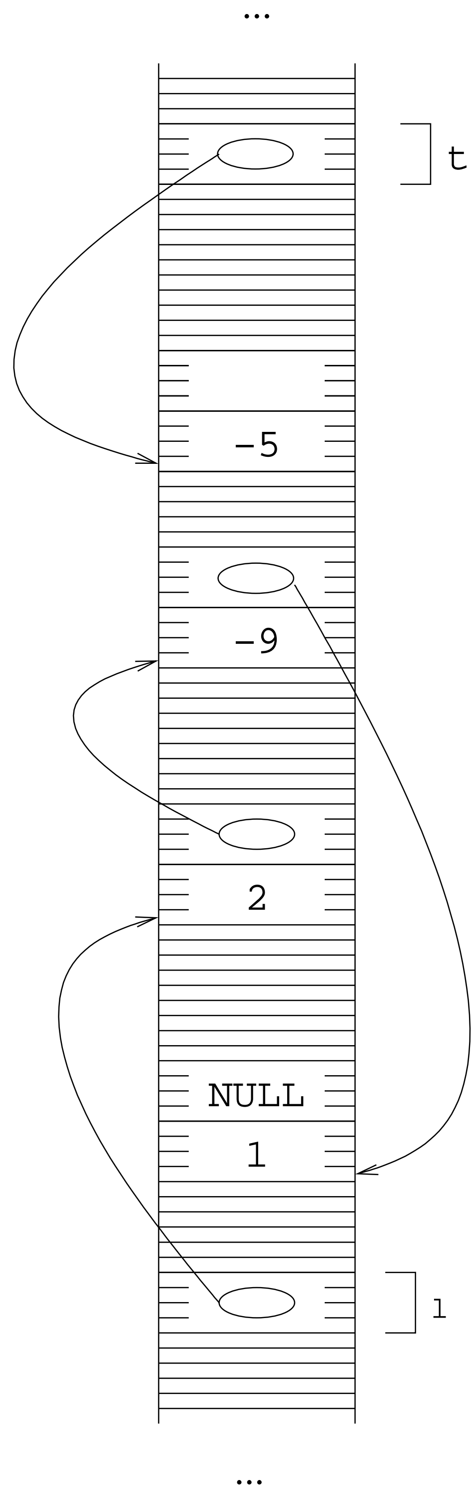
Si riporta qui sotto il programma aggiunta.c che crea una lista nel modo visto in precedenza, ci aggiunge due elementi in testa, e stampa tutta la lista. Si noti che definire una variabile come puntatore a struttura `struct NodoLista`, oppure come variabile di tipo `TipoLista` sia la stessa cosa. Infatti, la definizione del tipo `TipoLista` dice che le variabili di questo tipo sono effettivamente puntatori a strutture di tipo `struct NodoLista`.

```
/*
  Aggiunta di un elemento in testa a una lista.
*/

#include<stdlib.h>

struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;


/*
  stampa di una lista di lunghezza generica
*/

void StampaLista(TipoLista l) {
  TipoLista s;

  s=l;
  while(s!=NULL) {
    printf("%d ", (*s).val);
    s=(*s).next;
  }

  printf("\n");
}


/*
  main
*/

int main() {
  TipoLista l;
  TipoLista s, t;

                /* creazione della lista (2 -9 1) */

  s=NULL;

  t=s;
```

```
  s=malloc(sizeof(struct NodoLista));
  (*s).val=1;
  (*s).next=t;

  t=s;
  s=malloc(sizeof(struct NodoLista));
  (*s).val=-9;
  (*s).next=t;

  t=s;
  s=malloc(sizeof(struct NodoLista));
  (*s).val=2;
  (*s).next=t;

  l=s;

                /* aggiunge l'elemento -5 in testa alla lista */

  t=malloc(sizeof(struct NodoLista));

  (*t).val=-5;
  (*t).next=l;

  l=t;


                /* aggiunge l'elemento 13 in testa alla lista */

  t=malloc(sizeof(struct NodoLista));

  (*t).val=13;
  (*t).next=l;

  l=t;


                /* stampa la lista */

  StampaLista(l);

  return 0;
}
```

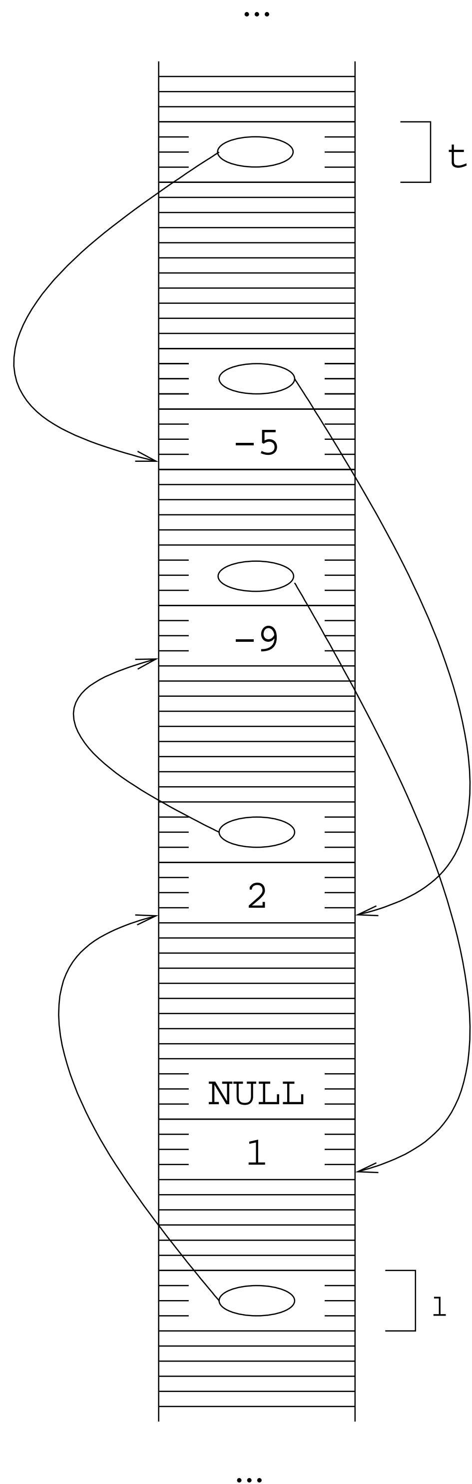
Vediamo ora cosa succede nella memoria quando si esegue la prima aggiunta in testa. Supponiamo che la memoria, prima della aggiunta, abbia il contenuto riportato qui a fianco (questa volta usiamo un esempio in cui le strutture non sono memorizzate in ordine).

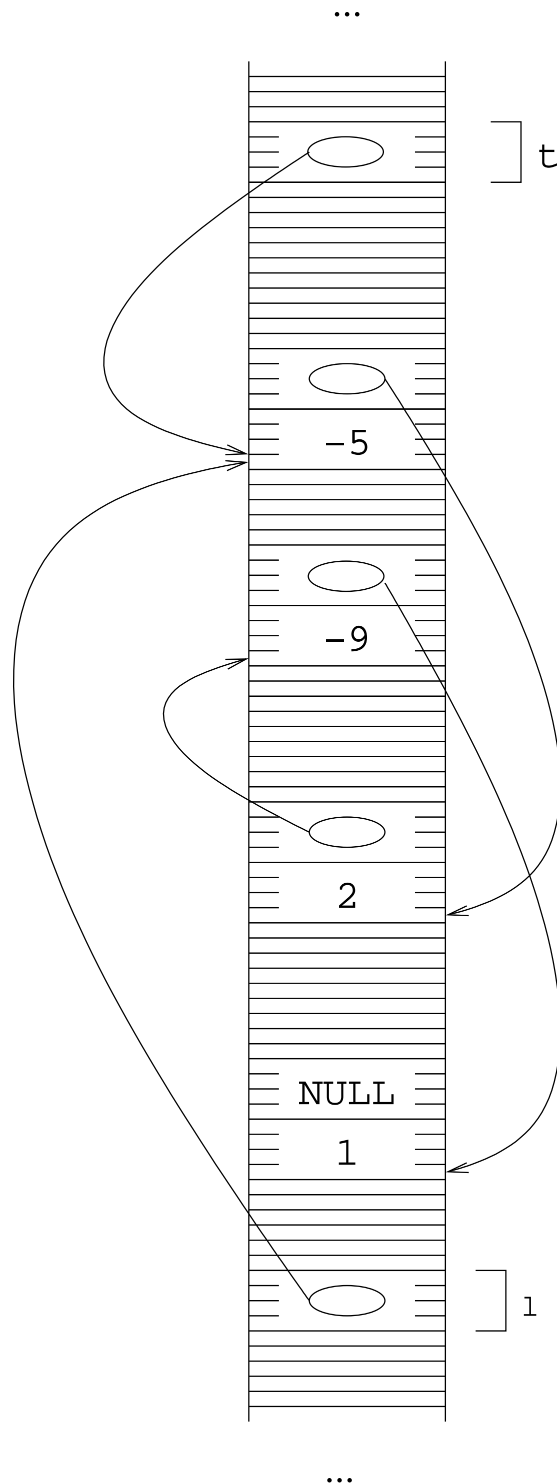




Quando si esegue la allocazione, si alloca la zona di memoria per una struttura, e il suo indirizzo va in `t`. Eseguendo l'istruzione `(*t).val=-5`, viene messo il valore `-5` nel primo campo di questa struttura.

Ora si esegue l'istruzione `(*t).next=l`. L'effetto è quello di copiare il numero memorizzato in `l` nella variabile `(*t).next`. Dopo questa istruzione, nel campo `next` della struttura puntata da `t` c'è il numero che stava in `l`, che è l'indirizzo della prima struttura della vecchia lista.





L’ultima istruzione è la copia del contenuto di `t` nella variabile `l`. Dal momento che `t` contiene l’indirizzo della nuova struttura, questo fa sí che ora `l` contenga l’indirizzo della nuova struttura, ossia che punti alla nuova struttura.

Il fatto che `l` ora rappresenta la lista `(-5 2 9 -1)` può non essere chiaro, a prima vista. Per capire qual’è la lista indicata da un puntatore, basta seguire le freccie: l’elemento puntato da `l` è la struttura che contiene `-5`, e quindi questo è il primo elemento della lista. Seguendo il puntatore di questa struttura, si arriva alla struttura il cui primo campo è `2`, poi ancora a `-9` e a `1`. Quindi la lista rappresentata è effettivamente `(-5 2 9 -1)`, ossia la lista di partenza a cui è stato aggiunto un nuovo elemento.

Costruzione con aggiunte successive

Il metodo di costruzione delle liste basato sull'uso di sequenze di tipo
`*(*(*l).next).next).next`, è inattuabile in caso di liste appena più lunghe di tre elementi. Abbiamo visto come si possano costruire le liste partendo dal fondo. Vediamo ora un altro metodo, che è completamente equivalente.

Questo metodo è basato sulla ripetizione della aggiunta di un elemento in testa alla lista. Per prima cosa, possiamo realizzare una funzione che aggiunge un elemento in testa a una lista:

```
void InserisciTestaLista(TipoLista *pl, int e) {
  TipoLista t;

  t=malloc(sizeof(struct NodoLista));
  (*t).val=e;
  (*t).next=*pl;

  *pl=t;
}
```

Questa procedura fa esattamente quello che faceva il frammento di codice che inseriva in testa. L'unico punto a cui bisogna fare attenzione è il fatto che, dal momento che il valore della variabile puntatore `l` viene modificato, va passato per riferimento. Questo punto potrebbe creare qualche difficoltà. In realtà è molto semplice: consideriamo la chiamata con la lista `(2 -9 1)` e il nuovo elemento `-5` (vedi figure nella pagina precedente). Quando la procedura viene chiamata, per esempio con `InserisciTestaLista(&l, -5)`, il *valore* di `l` è l'indirizzo della struttura che contiene `2`, mentre alla fine dell'esecuzione deve essere l'indirizzo della struttura che contiene `-5`. Quindi, il *valore* di `l` viene modificato. Inoltre, questo cambiamento deve essere visibile nel programma chiamante (se cosí non fosse, il programma chiamante continuerebbe a vedere la lista `(2 -9 1)` anche dopo la esecuzione della funzione). Quindi, la variabile `l`, dato che il suo valore viene modificato e la modifica deve essere visibile al chiamante, va passata per indirizzo.

La seconda cosa da notare è che la funzione va bene anche nel caso in cui la lista di partenza è vuota. Se eseguiamo `InserisciTestaLista(&l, 12)` e la variabile `l` vale `NULL`, quello che si ottiene è che `l` viene modificato in modo da puntare a una struttura che contiene `12` e `NULL`, cosa che si può verificare graficamente. Questa è esattamente la lista `(12)`, che è quello che ci si aspetta aggiungendo `12` in testa alla lista vuota `()`.

Possiamo ora passare alla costruzione della lista. Supponiamo di voler creare la lista `(90 32 -22)`. Potremmo procedere in questo modo: partiamo dalla lista vuota, e a questa aggiungiamo `-22` in testa. Come si è detto, si ottiene la lista `(-22)`. Se si aggiunge anche `32`, sempre in testa a questa nuova lista, si ottiene `(32 -22)`. Se aggiungo anche `90` ottengo esattamente la lista che serviva, ossia `(90 32 -22)`.

lista	cosa faccio
`()`	aggiungo `-22` in testa
`(-22)`	aggiungo `32` in testa
`(32 -22)`	aggiungo `90` in testa
`(90 32 -22)`	

Detto in un altro modo, se aggiungo una sequenza di elementi in ordine, ottengo che gli elementi si trovano nella lista in ordine inverso rispetto all'ordine in cui sono stati inseriti: se inserisco prima `-22`, poi `32` e alla fine `90`, quello che ottengo è la lista in cui questi elementi si trovano in ordine inverso, ossia `(90 32 -22)`.

Il seguente programma costruzione.c costruisce la lista `(13 -5 2 -9 1)` partendo dalla lista vuota e inserendo gli elementi in testa, cominciando dall'ultimo (prima `1`, poi `-9`, poi `2`, ecc).

```
/*
  Costruzione di una lista dal fondo.
*/

#include<stdlib.h>

struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;


/*
  stampa di una lista di lunghezza generica
*/

void StampaLista(TipoLista l) {
  TipoLista s;

  s=l;
  while(s!=NULL) {
    printf("%d ", (*s).val);
    s=(*s).next;
  }

  printf("\n");
}


/*
  aggiunta di un elemento in testa alla lista
*/

void InserisciTestaLista(TipoLista *pl, int e) {
  TipoLista t;

  t=malloc(sizeof(struct NodoLista));
  (*t).val=e;
  (*t).next=*pl;

  *pl=t;
}


/*
  main
*/

int main() {
  TipoLista l;
```

```
                /* creazione della lista (13 -5 2 -9 1) */

  l=NULL;       /* lista vuota */
  InserisciTestaLista(&l, 1);
  InserisciTestaLista(&l, -9);
  InserisciTestaLista(&l, 2);
  InserisciTestaLista(&l, -5);
  InserisciTestaLista(&l, 13);


                /* stampa la lista */

  StampaLista(l);

  return 0;
}
```

Lettura di una lista da file

Possiamo effettuare la lettura di una lista da file semplicemente partendo dalla lista vuota, e aggiungendo ogni volta elementi in testa alla lista. Occorre però tenere presente che la lista contiene gli elementi del file *in ordine inverso*.

Il seguente programma listafile.c usa la funzione di inserimento in testa per leggere una lista da file. Si tratta di un ciclo: se il file contiene ancora un intero (ossia, se `fscanf` ritorna `1`) lo si mette in testa alla lista; altrimenti, si esce dal ciclo.

```
/*
  Lettura di una lista da file
(in ordine inverso).
*/

#include<stdlib.h>
#include<stdio.h>


/* il tipo lista */

struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista *TipoLista;


/*
  stampa di una lista di lunghezza generica
*/

void StampaLista(TipoLista l) {
  TipoLista s;

  s=l;
  while(s!=NULL) {
    printf("%d ", (*s).val);
    s=(*s).next;
  }
```

```
  printf("\n");
}


/*
  aggiunta di un elemento in testa alla lista
*/

void InserisciTestaLista(TipoLista *pl, int e) {
  TipoLista t;

  t=malloc(sizeof(struct NodoLista));
  (*t).val=e;
  (*t).next=*pl;

  *pl=t;
}


/*
  main
*/

int main() {
  FILE *fd;
  int res;
  int x;

  TipoLista l;


                /* apre il file */
  fd=fopen("lista.txt", "r");
  if (fd==NULL) {
    perror("Errore in apertura del file");
    exit(1);
  }


                /* inizializza la lista */
  l=NULL;


                /* legge fino alla fine del file */
  while(1) {
    res=fscanf(fd, "%d", &x);
    if( res!=1 )
      break;

    InserisciTestaLista(&l, x);
  }

  fclose(fd);


                /* stampa la lista */

  StampaLista(l);

  return 0;
}
```

L'unica differenza fra questo programma e quello di lettura di un vettore è che, in questo caso, dopo che si è letto un elemento, lo mettiamo in testa a una lista invece di metterlo nel vettore. Non è più necessaria la variabile per memorizzare l'indice corrente del vettore.

L'operatore ->

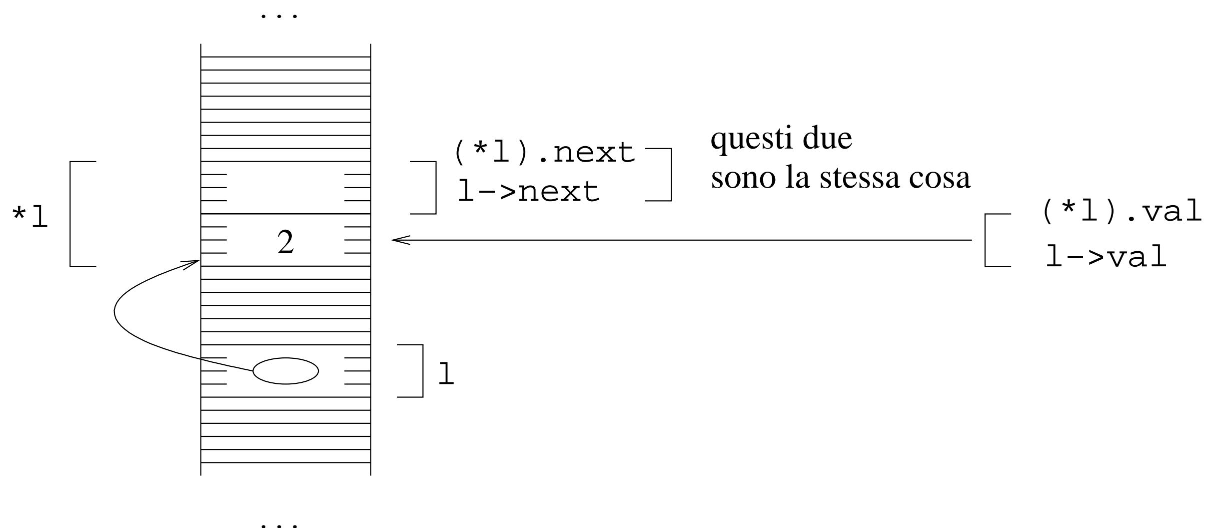
Già dai pochi esempi di programmi sulle liste visti fino a questo momento risulta chiaro che appaiono molto spesso espressioni del tipo

```
(*l).next
```

In cui `l` è un puntatore a una struttura. Il significato è chiaro: prendi la struttura puntata da `l`, e di questa prendi il campo `next`. Dal momento che questa espressione si ripete molto spesso, il C ne fornisce una forma abbreviata:

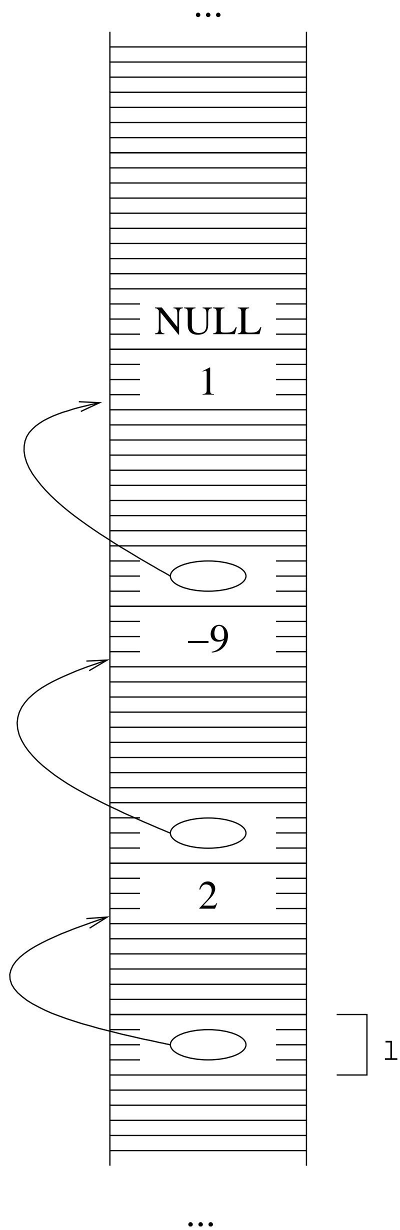
```
l->next
```

Questa espressione ha esattamente lo stesso significato della precedente. A sinistra della freccia deve apparire un puntatore a una struttura, e a destra il nome di uno dei campi di questa struttura. Questa espressione individua la sottovariabile della struttura `l` il cui nome è `next`. Questa espressione si può chiaramente usare anche al di fuori dell'ambito delle liste.

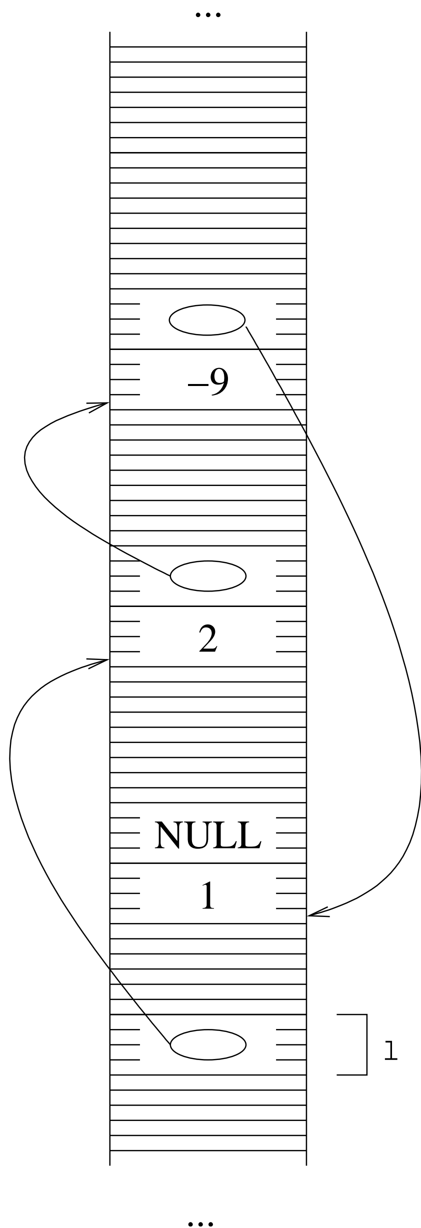


La notazione grafica

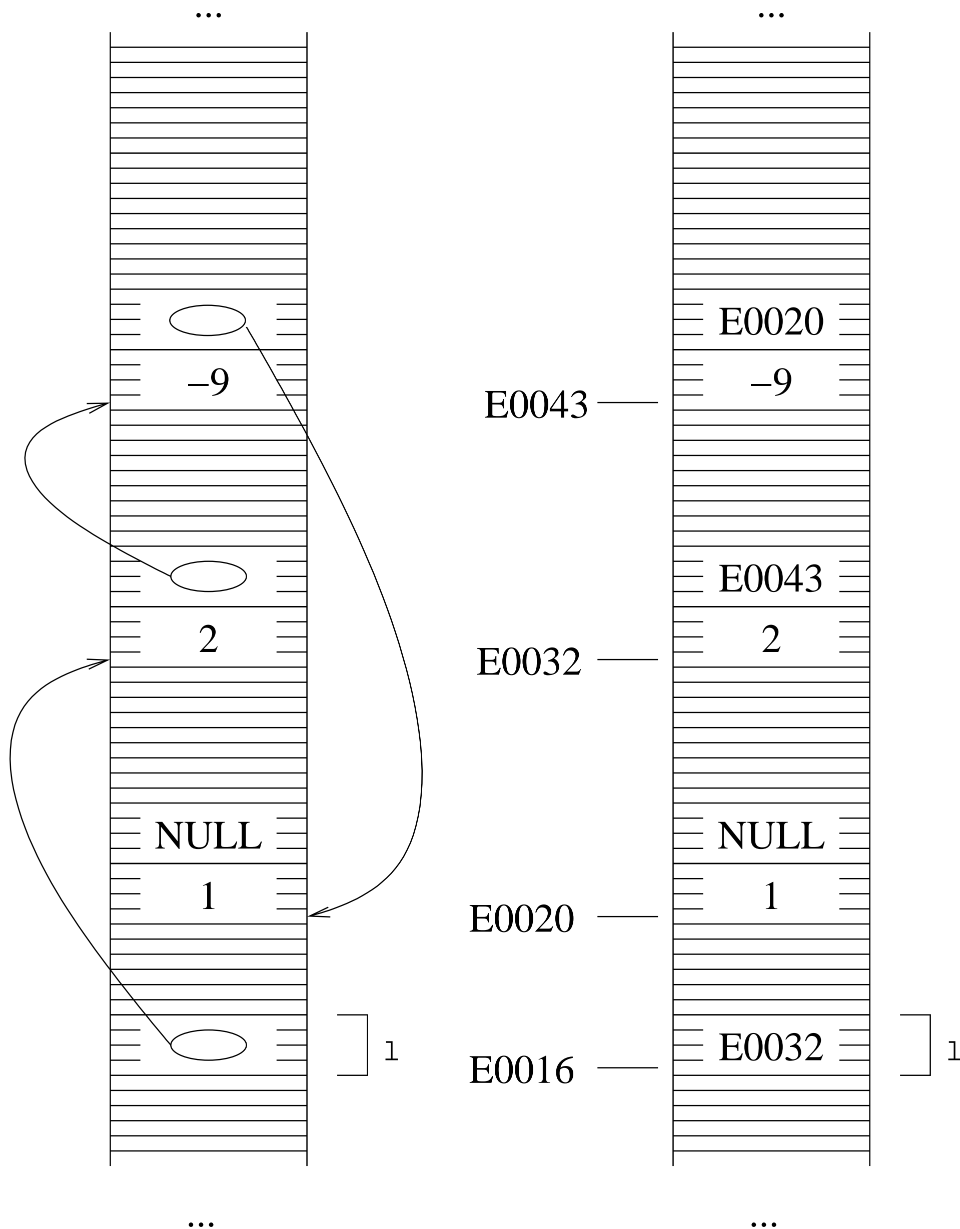
La notazione usata fino a questo momento per visualizzare il contenuto della memoria è quella in cui la memoria viene vista come una serie di rettangoli (uno per ogni byte) l'uno sopra l'altro, con le frecce per indicare che un puntatore contiene un certo indirizzo. Per le liste, questo metodo di visualizzazione può risultare scomodo.



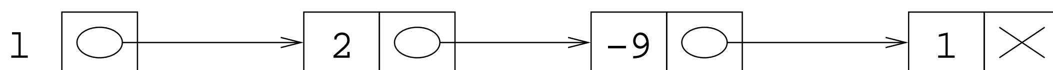
La figura di sopra può sembrare abbastanza chiara. Occorre però tenere presente che non c’è nessuna garanzia della posizione relativa fra gli elementi: potrebbero anche essere messi cosí:



Ripetiamo, ancora una volta, che le freccie sono una notazione: in realtà, nei campi puntatore ci sono degli indirizzi, ossia dei numeri. La freccia indica solo dove si trova l'indirizzo contenuto in un puntatore. Per esempio, si può confrontare l'esempio di sopra usando la freccia e dei numeri di esempio. La freccia è più chiara, ma non va mai dimenticato che in effetti i puntatori sono numeri.



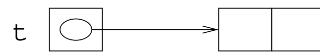
Introduciamo ora una nuova notazione grafica per rappresentare lo stato della memoria quando sono presenti liste e strutture. Il metodo dovrebbe essere già chiaro dall'esempio qui sotto.



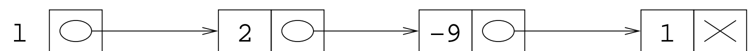
Ogni variabile si rappresenta con un rettangolo, e ogni struttura è rappresentata disegnando le sue componenti attaccate. La freccia indica come al solito che un puntatore contiene l'indirizzo in cui si trova un'altra variabile (che può essere di qualsiasi tipo, ad esempio di tipo scalare, oppure struttura). Usiamo la croce per indicare il valore `NULL`.

Si tenga ancora presente che si tratta di un modo per visualizzare quello che c'è nella memoria. In effetti, ogni rettangolo è un blocco di byte, e le freccie indicano la locazione scritta in un puntatore.

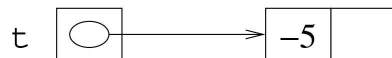
Facciamo vedere come questa notazione sia molto più comoda della precedente per rappresentare operazioni su liste. Consideriamo come esempio l'aggiunta di un elemento in testa, con questa nuova notazione. La prima cosa è la allocazione di una nuova struttura. Questo corrisponde a disegnare una nuova coppia di rettangoli, che rappresentano la nuova struttura.



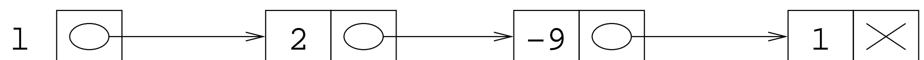

```
t=malloc(sizeof(struct NodoLista));
```



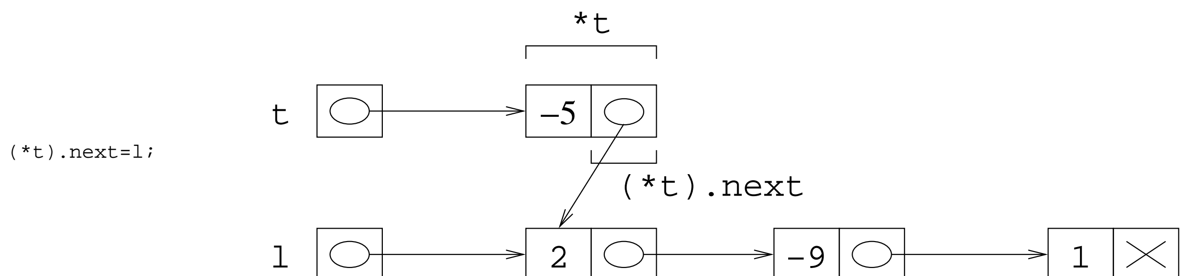
Viene poi scritto il valore `-5` nel campo `val` della struttura puntata da `t`:



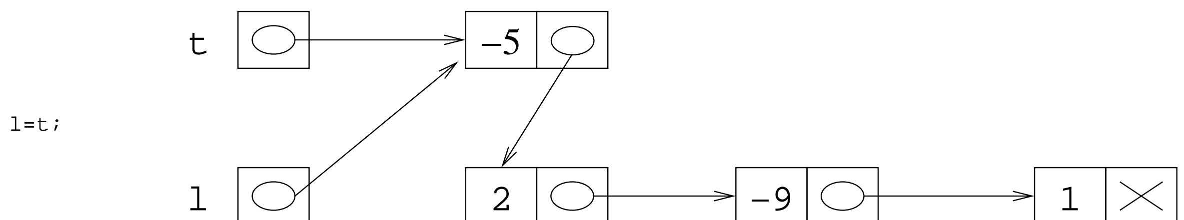

```
(*t).val=-5;
```



Occorre fare attenzione alla assegnazione `(*t).next=l`. Questo significa che i due puntatori contengono lo stesso indirizzo. Quindi, le freccie che usiamo per indicare gli indirizzi devono convergere sulla stessa locazione, che è quella di arrivo della freccia di `l`. Quello che risulta in questo caso è:



Per lo stesso motivo, quando si esegue `l=t` le freccie che partono da `l` e da `t` devono andare nello stesso punto. In particolare, la freccia di `l` deve andare verso il punto indicato dalla freccia di `t`.



Questa è la lista `(-5 2 -9 1)`: per capirlo basta seguire i puntatori (infatti, la lista si ottiene seguendo le freccie da struttura a struttura, mentre la posizione nella figura non conta).

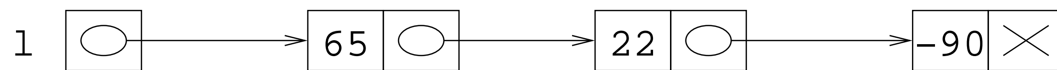
Questa nuova notazione permette di disegnare liste collegate in modo più facile. Va comunque tenuto presente che le freccie sono comunque puntatori, ossia numeri, e che la freccia è solo un modo comodo per indicare la locazione il cui indirizzo è memorizzato in un puntatore.

trasformare un array in una lista

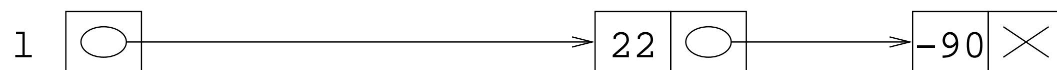
Eliminazione del primo elemento di una lista

Finora abbiamo visto come si crea una lista, come si aggiungono elementi, e come si scandisce (abbiamo visto solo la stampa). Ora vediamo come si fa a cancellare gli elementi di una lista. Il caso più semplice, che è quello che vediamo in questa pagina, è la cancellazione del primo elemento di una lista.

Iniziamo a vedere come è fatta le memoria prima della cancellazione, e come deve essere fatta dopo.



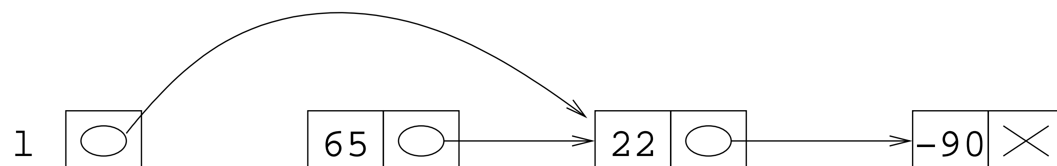
prima della cancellazione



dopo la cancellazione

Si noti che la prima struttura della lista non è più necessaria, e quindi va deallocata usando la funzione `free`.

Si vede chiaramente dalla figura che la posizione della freccia che parte da `l` deve essere modificata. Infatti, non deve più puntare alla prima struttura della lista, ma alla seconda. Spostare la freccia sul grafico equivale a cambiare il valore di `l`. In particolare, dentro `l` dobbiamo mettere l'indirizzo della seconda struttura della lista. Dal primo grafico risulta anche evidente che il campo `next` della prima struttura della lista contiene l'indirizzo della seconda struttura. Quindi, per ottenere lo spostamento della freccia, occorre copiare il valore di `l->next` in `l`. Il meccanismo più semplice per eliminare il primo elemento è quello di fare `l=l->next`. Questo modifica la situazione iniziale in questo modo:



la lista, dopo aver eserguito `l=l->next`

Questa figura si ottiene dalla considerazione che fare l'assegnamento fra puntatori significa copiare l'indirizzo scritto in un puntatore nell'altro, e questo equivale a modificare la freccia in modo che abbia la punta nella posizione in cui si trova la punta dell'altra.

Da questa figura si capisce chiaramente che:

1. la lista rappresentata da `l` è ora `(22 -90)`. Infatti, il primo elemento della lista è quello che si trova nella struttura puntata da `l`, e questa struttura contiene `22` e un puntatore che punta a un'altra struttura che contiene `-90` e `NULL`;
2. la struttura che contiene il valore `65` non è più necessaria. Infatti, partendo da `l` e seguendo i puntatori, non si arriva mai a questa struttura.

Si ricordi che la disposizione grafica delle strutture non ha importanza: la lista è quella che si ottiene seguendo i puntatori, ossia le freccie. Eventuali strutture che si trovano al di fuori di questo percorso non fanno comunque parte della lista.

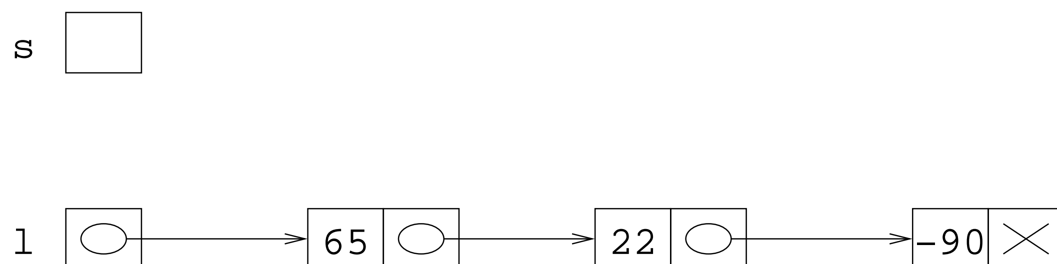
La struttura che contiene `65` non viene più usata, per cui occupa memoria senza motivo. È quindi bene deallocare la memoria. Per deallocare la memoria è necessario passare alla funzione `free` l'indirizzo iniziale della zona, in questo caso l'indirizzo della struttura. Il problema è che, avendo solo il valore di `l`, non riusciamo a risalire a questo indirizzo. Se per esempio la funzione di cancellazione del primo elemento è stata realizzata con una funzione, a questa passiamo solo il valore di `l`. Una volta che abbiamo fatto `l=l->next` non abbiamo nessun modo per risalire all'indirizzo della prima struttura (basta guardare la figura: possiamo capire gli indirizzi della varie strutture solo seguendo le freccie a partire da `l`, ma in questo modo non arriviamo mai alla prima struttura).

Se però guardiamo la prima figura, quella che dice lo stato della memoria prima della cancellazione, vediamo che l'indirizzo della prima struttura è scritto in `l`. È stata l'istruzione `l=l->next` che ci ha fatto perdere questa informazione. Dal momento che l'indirizzo che ci serve per fare la `free` è scritto nella variabile `l` all'inizio, possiamo copiare questo valore in un'altra variabile prima di fare `l=l->next`.

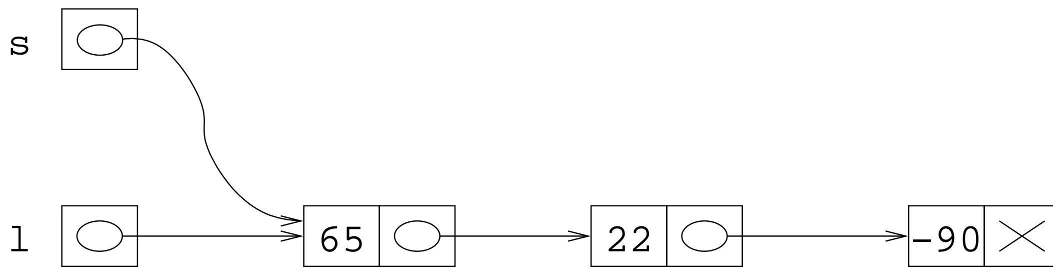
Facciamo quindi una sequenza di questo tipo: primo, memorizziamo il valore di `l` in una variabile temporanea `s`; secondo, mettiamo in `l` l'indirizzo della seconda struttura con `l=l->next`; terzo, liberiamo la memoria occupata dalla prima struttura: dato che il suo indirizzo sta scritto in `s`, questo si può fare con `free(s)`.

```
s=l;

l=l->next;

free(s);
```

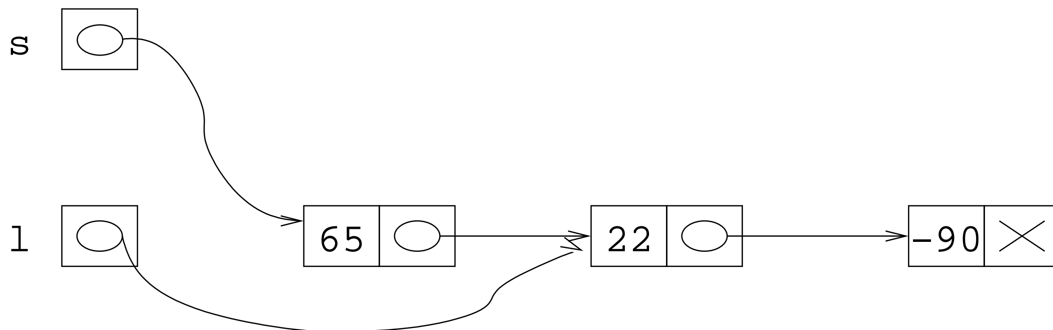
L'esecuzione di queste tre istruzioni produce la seguente evoluzione della memoria.



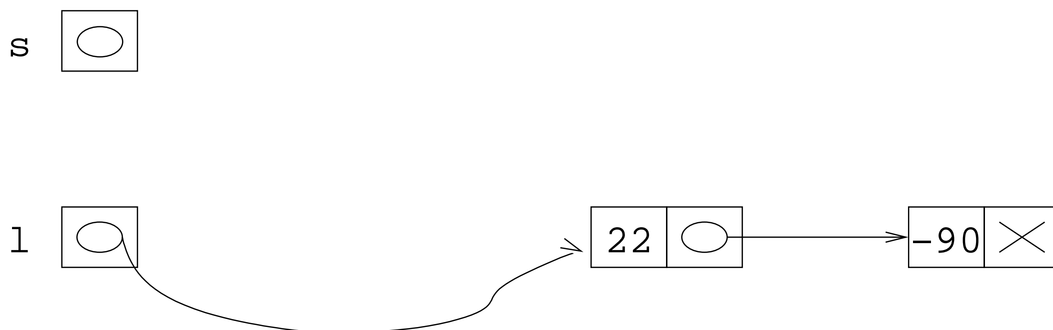
situazione iniziale



situazione dopo `s=l`



situazione dopo `l=l->next`



situazione dopo `free(s)`
la struttura precedentemente puntata da `s`
e' stata deallocata

A prima vista, può sembrare che questo codice si possa semplificare usando la seguente soluzione:

```
                /* CODICE ERRATO */
free(l);
l=l->next;
```

Questa soluzione è **sbagliata**. Infatti, una volta che la zona di memoria puntata da `l` è stata deallocata con `free(l)`, il suo contenuto è indefinito. Per quello che è possibile sapere, la zona di memoria potrebbe anche essere stata riempita di zeri. In questo caso, l'operazione di andare a prendere il secondo campo `l->next` è un errore di programmazione, perchè si sta accedendo a una zona di memoria il cui contenuto non è garantito. Quindi, dopo `l=l->next` il contenuto di `l` (ossia l'indirizzo contenuto in `l`) è indefinito.

In alcuni casi, questo frammento di codice può anche funzionare. Questo però è un problema più che un vantaggio, perchè diventa molto difficile capire perchè il programma in alcuni casi non funziona.

Scriviamo ora una funzione che realizza la cancellazione del primo elemento di una lista. Questa funzione riceve come parametro il puntatore alla prima struttura della lista. Dai grafici di sopra è abbastanza evidente che il puntatore al primo elemento della struttura deve essere modificato (dopo la esecuzione della funzione deve contenere l'indirizzo della seconda struttura e non della prima). Quindi, va passato per riferimento. Questo si fa come al solito: al posto di `l` si mette `*pl` nella intestazione e nel corpo della funzione, e il programma chiamante usa `&l` come parametro.

Per il resto, il corpo della procedura è esattamente identico al frammento di codice che elimina il primo elemento di una lista:

```
void EliminaTestaLista(TipoLista *pl) {
  TipoLista s;

  s=*pl;

  *pl=(*pl)->next;

  free(s);
}
```

Il programma delprimo1.c è un esempio di uso di questa funzione: si legge una lista da file, e si cancellano i primi due elementi.

La funzione definita sopra funziona, se la lista non è vuota. Nel caso in cui la lista sia inizialmente vuota, si genera un errore in esecuzione. Infatti, l'istruzione `*pl=(*pl)->next` tenta di accedere a una struttura il cui indirizzo si trova in `*pl`. Nel caso in cui la lista è vuota, `*pl` contiene `NULL`, e tentare di accedere alla memoria puntata da `NULL` genera un errore.

Per evitare questi problemi, possiamo aggiungere alla funzione un test per verificare se la lista è vuota. Nel caso in cui la lista è vuota possiamo comporarci in diversi modi: stampare un messaggio di errore ed eventualmente uscire, oppure semplicemente assumere che eliminare il primo elemento da una lista vuota genera la lista vuota. Nel programma delprimo2.c usiamo la funzione in cui è stata adottata la prima scelta.

```
void EliminaTestaLista(TipoLista *pl) {
  TipoLista s;

  if(*pl==NULL) {
    printf("Tentativo di eliminazione testa da lista vuota\n");
    exit(1);
  }

  s=*pl;

  *pl=(*pl)->next;

  free(s);
}
```

Eliminazione ultimo elemento di una lista

La eliminazione dell'ultimo elemento di una lista può sembrare, a prima vista, un problema abbastanza semplice. Invece, presenta un serie di trabocchetti.

La prima cosa da fare è chiaramente quella di seguire i puntatori fino ad arrivare all'ultima struttura della lista, che è quella che va deallocata. Nel programma delultimo1.c usiamo un ciclo di scansione che termina solo quando il puntatore di scansione vale `NULL`. La lista viene passata per riferimento. Questo può non essere a prima vista chiaro, dal momento che la il puntatore deve comunque contenere l'indirizzo della prima struttura, sia prima che dopo la cancellazione. Esiste infatti un caso in cui questo non vale: se la lista ha un unico elemento, la cancellazione dell'ultimo equivale a mettere `NULL` nel puntatore. Quindi, il puntatore che prima conteneva l'indirizzo della prima struttura ora contiene `NULL`, per cui è stato modificato, e quindi va passato per riferimento.

```
/* prima versione: ERRATA */

void EliminaUltimoLista(TipoLista *pl) {
  TipoLista r;

  r=*pl;

  while(r!=NULL)
    r=r->next;

  free(r);
  r=NULL;
}
```

Il problema di questa funzione è evidente, se si pensa che dopo un ciclo la condizione del ciclo deve essere necessariamente falsa (se non ci sono `break` dentro il ciclo). Nel nostro caso, la condizione è `r!=NULL`; se questa è falsa allora `r` vale `NULL`. In altre parole, si esce dal ciclo solo quando `r` vale `NULL`. Si può quindi dire che il ciclo `while` equivale alla assegnazione `r=NULL`. È quindi chiaro che l'istruzione `free(r)` è equivalente a `free(NULL)`, e che l'ultima istruzione della funzione, ossia `r=NULL`, non ha nessun effetto. Questa funzione lascia la lista inalterata.

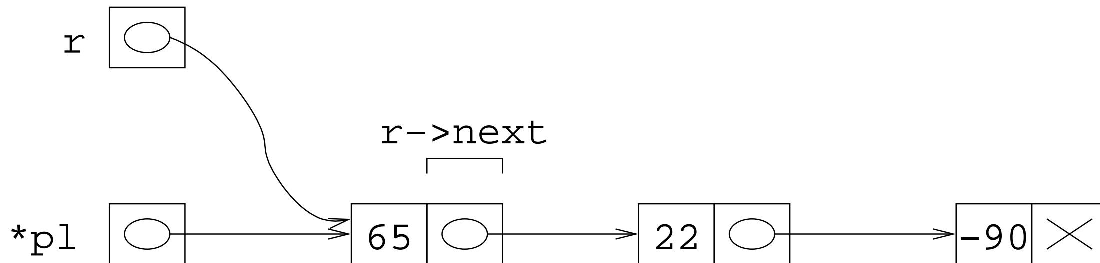
Quello che ci serve non è ovviamente un puntatore con dentro `NULL`. Ci serve l'indirizzo dell'ultima struttura della lista, in modo da poterla deallocare. Se si va a vedere il ciclo di scansione della lista della funzione precedente, ci si accorge subito che esiste un momento in cui effettivamente `r` punta all'ultima struttura della lista, ossia contiene il suo indirizzo. Il problema del programma di sopra è che viene poi eseguita una ulteriore iterazione del ciclo, per cui questo valore che ci serve viene rimpiazzato da `r->next`, che vale `NULL`.

Occorre quindi fare in modo che non si vada avanti fino a che `r` vale `NULL`, ma occorre fermarsi quando `r` punta all'ultima struttura della lista. Per definizione, l'ultima struttura della lista è quella che contiene `NULL` nel campo `next`. Quindi, `r` punta all'ultima struttura della lista quando `r->next` vale `NULL`. Possiamo quindi modificare il ciclo, come viene fatto nel programma delultimo2.c, la cui funzione di eliminazione dell'ultimo elemento è riportata qui sotto.

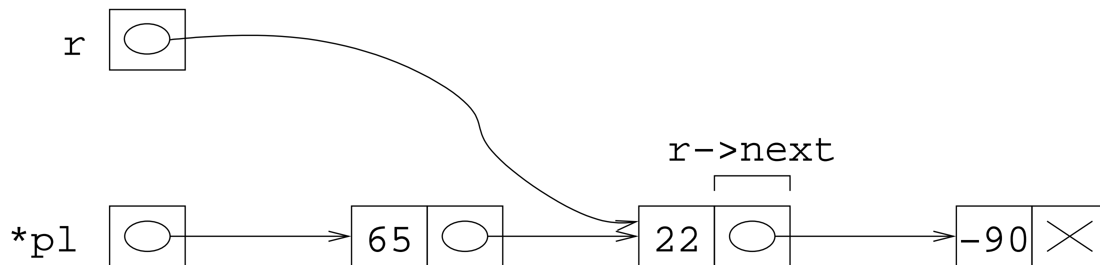
```
/* seconda versione: ERRATA */

void EliminaUltimoLista(TipoLista *pl) {
  TipoLista r;

  r=*pl;
```

```
  while(r->next!=NULL)
    r=r->next;

  free(r);
  r=NULL;
}
```

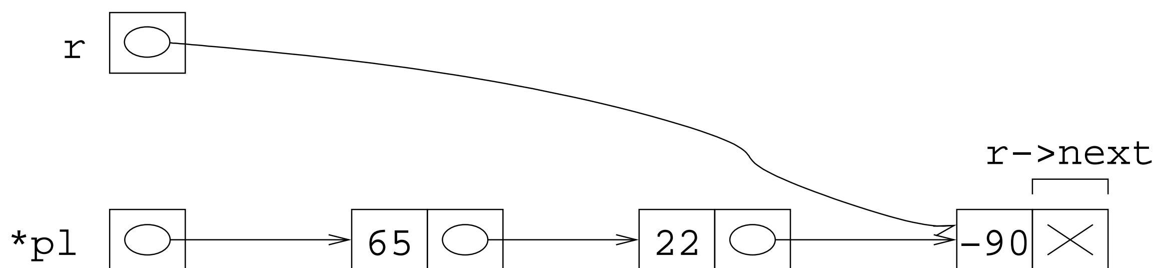
Questa funzione realizza in parte quello che era richiesto, ma non è completamente corretta. Infatti, l'ultima struttura viene effettivamente deallocata, ma l'ultimo elemento non è stato realmente tolto dalla lista. Consideriamo infatti la evoluzione della memoria per una semplice lista di tre elementi.



dopo l'esecuzione di `r=*l;`

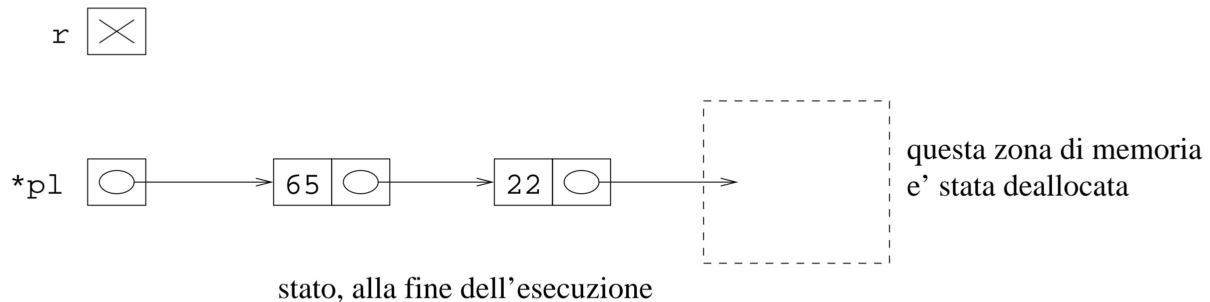


dopo la prima esecuzione di `r=r->next;`



dopo la seconda esecuzione di `r=r->next;`

A questo punto `r->next` vale effettivamente `NULL`, per cui si esce dal ciclo (si vede chiaramente dalle figure che non si esce prima, dato che prima `r->next` contiene indirizzi di altre strutture e non `NULL`). A questo punto viene liberata la memoria puntata da `r`, e in `r` si mette il valore `NULL`, per cui si ottiene una situazione di questo tipo:



stato, alla fine dell'esecuzione

Questa figura si ottiene semplicemente operando le due istruzioni `free(r)` e `r=NULL` sulla figura di sopra: dato che `r` punta all'ultima struttura, questa viene deallocata. Si mette poi `NULL` in `r`, e questo viene rappresentato graficamente con una croce.

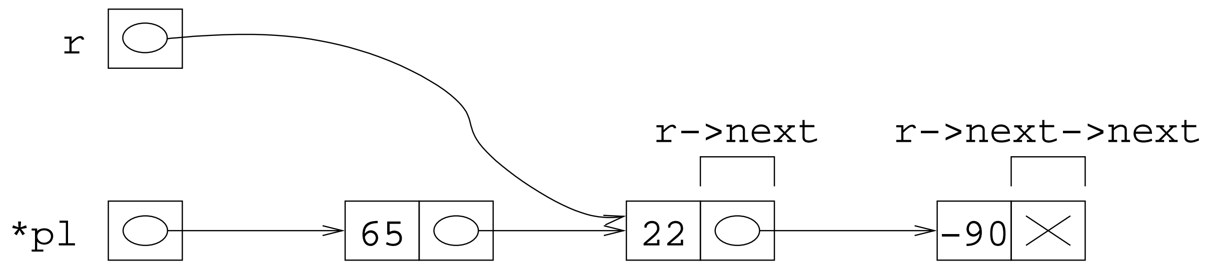
È evidente che la lista finale non è fatta nel modo in cui ci si aspetta. Infatti, l'ultima struttura (quella che contiene `22`) non ha `NULL` nel campo `next`. Al contrario, contiene l'indirizzo di una zona di memoria che è stata deallocata, per cui non sappiamo neanche con certezza cosa c'è in questa zona. Si noti che l'istruzione `r=NULL` mette il valore `NULL` nella variabile `r` e basta. Come per tutte le assegnazioni in C, quando si assegna un valore a una variabile, le altre non vengono infuenzate. Nel nostro caso, mettere `NULL` in `r` non lo mette anche nel campo `next` della penultima struttura.

Il problema è quindi quello di mettere il valore `NULL` nel campo `next` della struttura che rappresenta il penultimo elemento della lista (e che ora deve diventare l'ultimo). Per poter fare questa operazione, ci serve l'indirizzo della penultima struttura della lista. Infatti, dato questo indirizzo, possiamo usare `(indirizzo penultima struttura)->next=NULL` per chiudere in fondo la lista.

Riguardando ancora l'evoluzione della memoria durante la scansione, si vede chiaramente che esiste un momento in cui `r` contiene effettivamente questo indirizzo. Questo avviene esattamente alla fine della penultima iterazione del ciclo. Possiamo quindi dire che si esegue una iterazione di troppo. In particolare, `r` contiene l'indirizzo della penultima struttura solo quando `r->next` contiene l'indirizzo dell'ultima struttura. A sua volta, `r->next` è l'indirizzo dell'ultima struttura solo se `r->next->next` vale `NULL`. Possiamo quindi far terminare il ciclo non appena `r->next->next` vale `NULL`. A questo punto: possiamo deallocare l'ultima struttura, dato che conosciamo il suo indirizzo, che è `r->next`; possiamo mettere a `NULL` il campo `next` della penultima struttura, dato che l'indirizzo della penultima struttura è `r`, e quindi possiamo fare `r->next=NULL`. In altre parole, definiamo la funzione nel seguente modo. Il programma completo in cui questa funzione è inserita è delultimo3.c

```
void EliminaUltimoLista(TipoLista *pl) {
  TipoLista r;

  r=*pl;

  while(r->next->next!=NULL)
    r=r->next;

  free(r->next);
  r->next=NULL;
}
```

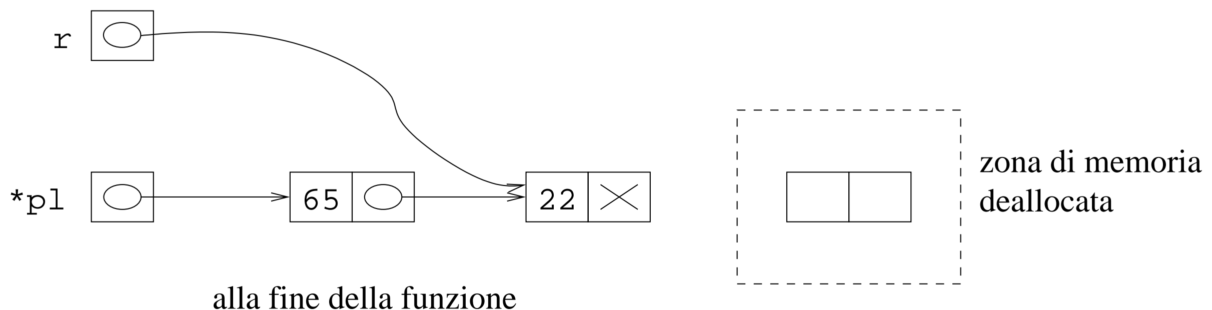
Rivediamo ora nel dettaglio la situazione della memoria dopo che `r=r->next` è stato eseguito per la prima volta:



dopo la prima esecuzione di r=r->next;
r->next->next vale NULL

A questo punto, `r->next->next` vale `NULL`, per cui si esce dal ciclo. Ora `r` punta alla penultima struttura, mentre `r->next` è l'indirizzo dell'ultima struttura. Quindi, possiamo fare la deallocazione dell'ultima struttura passando `r->next` alla funzione `free`. Per mettere `NULL` nel campo `next` della penultima struttura, basta fare `r->next=NULL`.

Quello che si ottiene è la seguente situazione (la zona di memoria deallocata ha un contenuto indefinito).

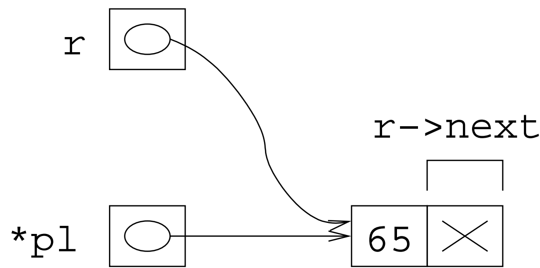


alla fine della funzione

La lista rappresentata da `*pl` si ottiene semplicemente seguendo i puntatori, per cui il primo elemento è `65`, mentre il secondo è 22, e non ci sono altri elementi (si arriva al puntatore `NULL`). Quindi, la lista che si ottiene è `(65 22)`, che è effettivamente la lista originaria `(65 22 -90)` a cui è stato tolto l'ultimo elemento.

Tutto questo funziona bene se la lista ha tre elementi. È facile verificare che va ancora tutto bene se la lista contiene più elementi, oppure nel caso in cui ne contiene solo due (basta simulare il comportamento della funzione usando la notazione grafica). D'altra parte, la funzione va in errore nel caso in cui la lista contiene un elemento, oppure nessuno.

Supponiamo infatti che la lista contenga un solo elemento. In questo caso, abbiamo una situazione del genere:



la lista di un solo elemento,
dopo l'esecuzione di r=*l;

In altre parole, r->next vale NULL, per cui r->next->next equivale a NULL->next, ossia (*NULL).next. Come è noto, l'operatore * produce un errore quando viene applicato a NULL. Il caso in cui la lista è vuota è simile, soltanto che r=*pl fa sí che in r venga messo il valore NULL.

La soluzione del problema è comunque abbastanza semplice. Infatti, solo due casi particolari (lista vuota e lista di un elemento) danno dei problemi. Quindi, possiamo semplicemente controllare questi due casi e agire di conseguenza. Nel caso di lista vuota (ossia *pl==NULL), possiamo semplicemente stampare un messaggio di errore. Nel caso di lista di un elemento (ossia (*pl)->next==NULL), dobbiamo deallocare questo elemento e mettere NULL in *pl e uscire dalla funzione.

Il programma finale delultimo4.c contiene quindi la seguente funzione.

```
void EliminaUltimoLista(TipoLista *pl) {
  TipoLista r;

  if(*pl==NULL) {
    printf("Tentativo di eliminazione da lista vuota\n");
    exit(1);
  }

  if((*pl)->next==NULL) {
    free(*pl);
    *pl=NULL;
    return;
  }

  r=*pl;

  while(r->next->next!=NULL)
    r=r->next;

  free(r->next);
  r->next=NULL;
}
```

Si noti che il controllo *pl==NULL viene fatto *prima* del controllo (*pl)->next==NULL. Questo è necessario. Se si prova a mettere prima il controllo (*pl)->next==NULL, e la lista è vuota, questo equivale a verificare NULL->next==NULL, ossia è un accesso a *NULL, che genera un errore. Seguendo invece l'ordine della funzione sopra, nel caso in cui la lista è vuota *pl vale NULL, per cui si esegue la prima istruzione condizionale e non si arriva alla seconda.

Liste passate per valore

In questa pagina vediamo cosa succede se una funzione modifica una lista. Consideriamo il seguente programma di esempio sommauno.c che somma 1 a tutti gli elementi di una lista di interi.

La funzione di somma a tutti gli elementi della lista non presenta nessuna difficoltà: si tratta infatti di effettuare una scansione della lista, aggiungendo uno a tutti gli elementi che si incontrano. La cosa interessante del programma sommauno.c è che la lista viene passata per valore invece che per riferimento.

```
void SommaUno(TipoLista l) {
  TipoLista s;

  s=l;

  while(s!=NULL) {
    s->val=s->val+1;
    s=s->next;
  }
}
```

A prima vista può sembrare che la funzione non vada bene, se è scritta in questo modo: infatti, il passaggio per valore dovrebbe copiare la lista, e quindi le modifiche fatte dalla funzione dovrebbero avere effetto solo sulla copia locale. invece, il programma funziona perfettamente: il programma principale, alla fine della esecuzione della funzione, vede la lista modificata. Stampando la lista prima e dopo la chiamata di funzione, si vede che effettivamente la lista è stata modificata, e che le modifiche sono visibili dal programma principale: se per esempio si stampa la lista prima e dopo la chiamata di funzione, si vede come la lista dopo la chiamata di funzione è diversa da quella di prima.

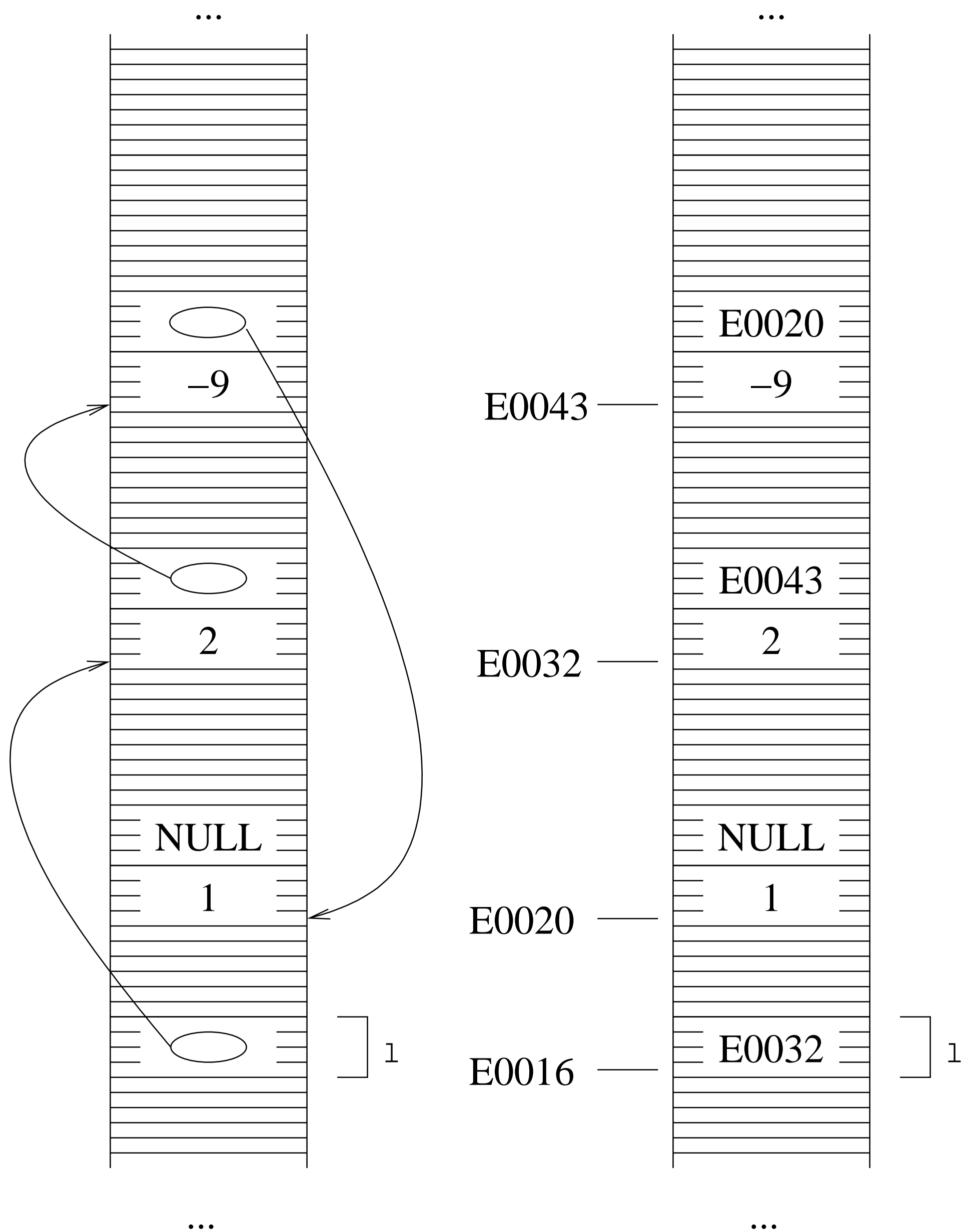
Per capire questo comportamento, occorre dare una interpretazione letterale al concetto di ''passaggio di parametro''. La regola del C sulle variabili passate a una funzione è:

> quando si passa un valore a una funzione, viene creata una nuova variabile, locale alla funzione, in cui viene copiato questo valore.

Il meccanismo del passaggio dei parametri va interpretato letteralmente. Una variabile di tipo `TipoLista` è un puntatore, ossia contiene un numero (che rappresenta un indirizzo di memoria). Quando viene chiamata la funzione usando `SommaUno(l)` quello che succede è che si crea una variabile locale alla funzione, in cui viene copiato *il valore passato alla funzione*, ossia il numero che sta scritto nella variabile `l`. Sui puntatori, la regola di sopra si può specializzare come segue (si noti che non è una nuova regola, ma solo un caso particolare di quella di sopra):

> quando si passa un puntatore per valore, viene creata una copia locale del solo puntatore: gli oggetti da esso puntati non sono copiati, per cui i due puntatori (quello del programma chiamante e quello locale) contengono lo stesso indirizzo, ossia puntano allo stesso identico oggetto.

Vediamo quindi in che modo evolve la memoria. Supponiamo che la lista `l` sia costituita dalla sequenza di valori `(2 -9 1)`. Una possibile disposizione in memoria della strutture si trova qui sotto. Usiamo sia la rappresentazione con freccie che quella con numeri.



La funzione `SommaUno` ha un unico parametro, di tipo `TipoLista`, ossia un puntatore a struttura. Quando la funzione viene chiamata, si crea una variabile locale di questo tipo, e in essa viene messo il valore che è stato passato. Nell'esempio di sopra, il valore passato è `E0032`, e questo è quindi il valore che viene messo nella variabile locale della funzione.

...

E0032 — l (della funzione)

E0020
–9 — E0043

E0043
2 — E0032

NULL
1 — E0020

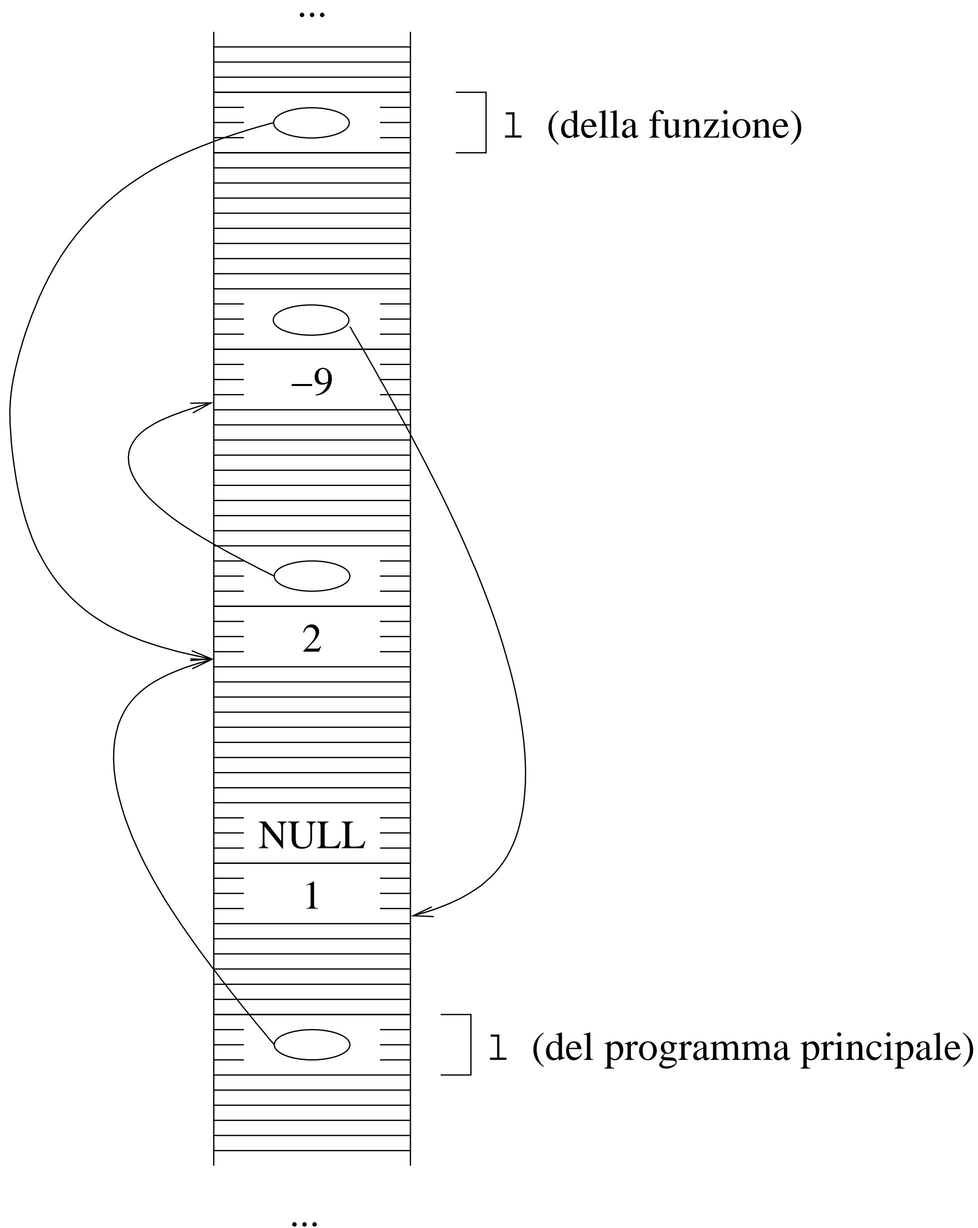
E0032 — E0016 — l (del programma principale)

...

Da notare che questo è semplicemente quello che si ottiene applicando due regole note del linguaggio C, ossia:

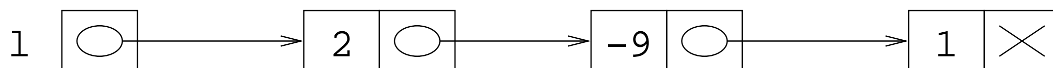
1. passare un valore significa creare una variabile locale in cui viene scritto il valore;
2. una variabile puntatore contiene un numero.

Proviamo ora a vedere cosa succede traducendo il diagramma di sopra nella notazione con freccie. Tutto quello che serve è considerare che, se un puntatore contiene un numero, allora si mette una freccia con la punta verso l'indirizzo identificato da quel numero.



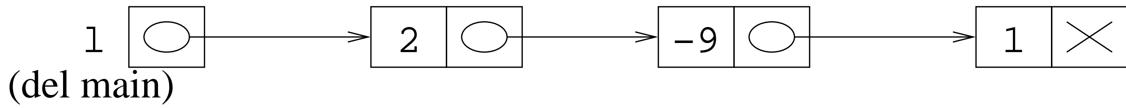
Lo stesso risultato si ottiene considerando che la funzione contiene una variabile locale `l`, e che in questa viene messo il valore del parametro passato. Dal momento che sia la variabile locale che il parametro passato sono puntatori, questo equivale alla copia fra puntatori, che nella notazione grafica equivale a mettere nella variabile in cui si copia una freccia con la punta nella stessa locazione della variabile da cui si copia.

Possiamo anche usare l'ultima notazione grafica, quella introdotta appositamente per le liste. La situazione iniziale è la seguente:



Quando si chiama una funzione, si creano tutte le sue variabili locali. In particolare, si crea anche una variabile locale per ognuno dei parametri passati. Nel nostro caso, viene creata una variabile locale `l` di tipo `TipoLista`.

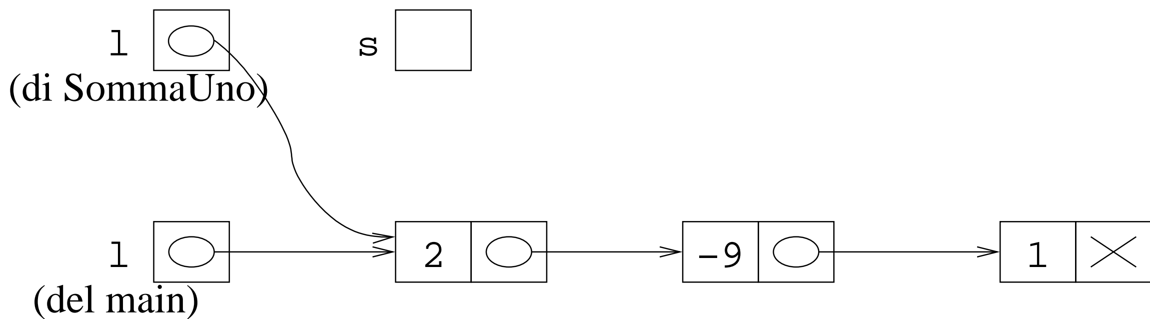
l
(di SommaUno)



Chiamare una funzione equivale a creare una variabile locale `l`, in cui viene copiato il valore del parametro. Questo equivale a fare una cosa del tipo:

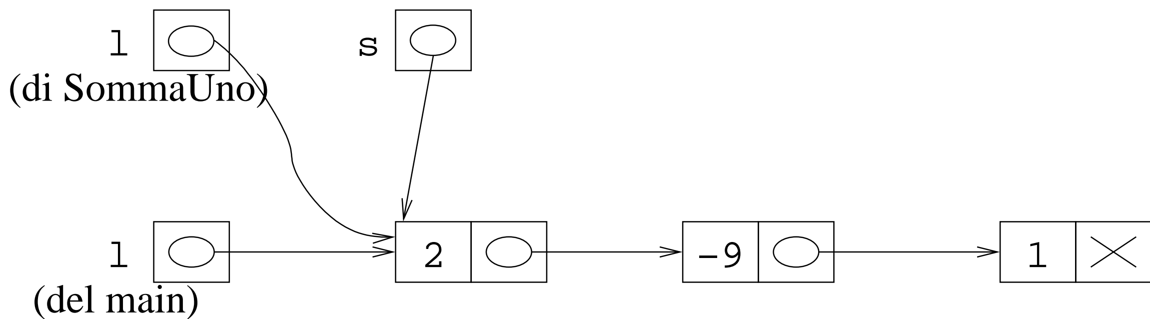
```
l (locale della funzione) = l (del programma principale)
```

Come si è già detto, assegnare un puntatore equivale a copiare il numero. Questo equivale a dire che la `l` della funzione alla fine contiene il numero che stava nella `l` del programma principale. Dal momento che questo numero è l'indirizzo della prima struttura, la variabile locale `l` contiene l'indirizzo della prima struttura, che si rappresenta mettendo una freccia da essa alla prima struttura della lista.

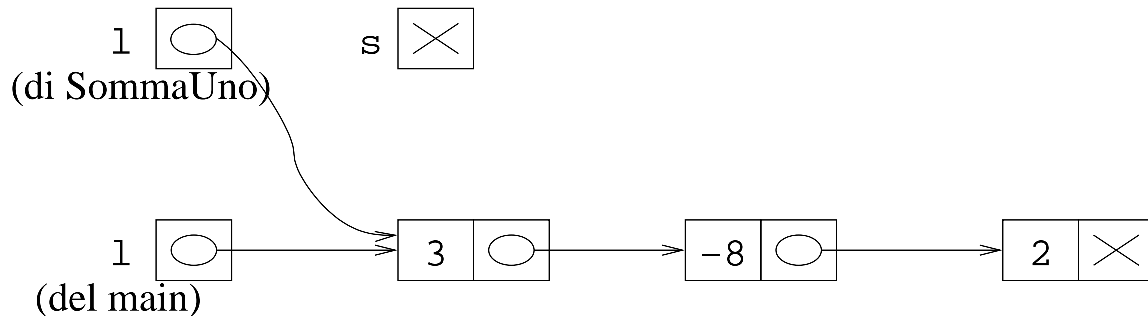


Graficamente, assegnare i puntatori equivale a mettere in quello a sinistra di = una freccia con la punta nella stessa posizione del puntatore a destra del simbolo =. Questo è quindi lo stato della memoria quando la funzione inizia.

Il resto della funzione non presenta difficoltà. Consideriamo per esempio il primo passo: si assegna ad `s` il valore di `l`, e si esegue il ciclo. Al primo passo si incrementa di uno il valore di `s->val`, ossia si aumenta di uno il campo `val` della prima struttura della lista.



Si noti che l'incemento viene fatto sulla struttura puntata da `s`, che è la stessa puntata da `l` (sia quella dal `main` che quella della funzione). Lo stesso incremento viene poi fatto per tutte le altre strutture. Quando la funzione termina, lo stato della memoria è quindi il seguente.



La funzione, a questo punto, termina. Questo equivale a cancellare le variabili locali, ossia la variabile `s` e la variabile `l` locale della funzione. Quello che rimane sono la variabile `l` del programma principale, e le strutture allocate dinamicamente.

Come è fatta la lista prima/dopo la chiamata di funzione? Prima della chiamata la lista è `(2 -9 1)`, come abbiamo assunto all'inizio. Per vedere come è fatta la lista dopo la chiamata, occorre partire da `l` e vedere le strutture che si incontrano seguendo i puntatori. Dal diagramma di sopra appare abbastanza evidente che il primo elemento è `3`, seguito da `-8` seguito da `2`. La lista dopo la chiamata è quindi `(3 -8 2)`.

Riassumendo, la lista è stata cambiata dalla funzione, anche se la variabile `l` è stata passata per valore. Questo avviene per i seguenti motivi.

1. una variabile di tipo lista è un puntatore, per cui contiene semplicemente un numero (che è un indirizzo di memoria);
2. quando si passa una variabile a una funzione, viene creata una variabile locale in cui viene copiato il valore passato;
3. quindi, passare un puntatore equivale a creare una variabile puntatore locale che punta allo stesso oggetto;
4. la lista si ottiene guardando gli oggetti puntati.

In altre parole, dato che la lista è rappresentata dalle strutture puntate da `l`, e passare un puntatore equivale a creare una nuova variabile che punta allo stesso oggetto, le modifiche alle strutture puntate sono modifiche alle strutture puntate nel programma principale (questo avviene perchè sono le stesse strutture).

La regola per decidere se passare una lista per valore oppure per indirizzo non dipende dalla possibilità per la funzione di fare modifiche sulla lista. Infatti, la lista può venire modificata anche se viene passata per valore. La regola è invece la seguente:

> una lista va passata per indirizzo se il puntatore iniziale viene modificato, e questo modifica deve essere visibile dal programma principale.

Notiamo ancora una volta che questo *non è una nuova regola*, ma semplicemente quello che si ottiene applicando letteralmente le regole sul passaggio dei parametri in C. Infatti, passare un puntatore equivale a copiare il suo valore. Se il *valore* del puntatore (ossia l'indirizzo a cui punta) deve venire alterato dalla funzione, allora occorre fare un passaggio per indirizzo. Se invece il puntatore continua

ad avere la punta della freccia nello stesso punto, allora il valore del puntatore non viene cambiato, per cui si può fare un passaggio per valore anche se la lista va cambiata.

Una altro caso interessante è quello della aggiunta di un elemento in testa alla lista. In questo caso, il puntatore che rappresenta la lista inizialmente punta alla prima struttura della vecchia lista. Quando si inserisce la nuova struttura, il puntatore contiene l'indirizzo della nuova struttura, che è diverso dall'indirizzo della vecchia.

Nel caso il concetto illustrato sopra non fosse stato chiarito, si suggerisce di leggere il capitolo su memoria e puntatori. Vediamo una serie di esempi. Partiamo da una situazione ovvia: cosa succede se una funzione riceve un puntatore a intero?

```
void Esempio(int *a) {
  *a=12;
}

int main() {
  int a=3;
  Esempio(&a);
  ...
```

Questo cambia il valore di `a`? Chiaramente sí. Si tratta infatti del passaggio della variabile `a` per indirizzo. Ora proviamo a passare, invece che un intero, una struttura.

```
struct abcd{
  int cmp;
};

void Esempio(struct abcd *a) {
  (*a).cmp=12;
}

int main() {
  struct abcd a;
  a.cmp=12;
  Esempio(a);
  ...
```

Non cambia niente: si tratta del passaggio di una struttura per indirizzo, per cui le modifiche fatte nella funzione sono modifiche che vengono fatte direttamente sulla struttura dichiarata nel programma principale.

Una variabile di tipo `TipoLista` è un puntatore a una struttura. In particolare, la struttura contiene a sua volta un campo puntatore, ma questo, dal punto di vista del linguaggio, è irrilevante. La cosa fondamentale è che passare una lista a una funzione equivale a passare l'indirizzo della prima struttura. Quindi, la funzione opera non su una copia della struttura, ma sulla struttura stessa, nello stesso modo in cui passando l'indirizzo di una variabile intera, la funzione modifica la variabile stessa e non una sua copia.

Volendo riassumere: le liste sono puntatori a struttura, per cui seguono semplicemente le regole dei puntatori, sia per quello che riguarda l'assegnazione che il passaggio dei parametri a funzione. In particolare, il fatto che una varabile di tipo `TipoLista` rappresenta una sequenza dipende semplicemente dalla interpretazione che diamo alle strutture puntate. In altre parole, per il linguaggio una variabile di tipo lista non è altro che un puntatore: siamo noi che diciamo che rappresenta la lista

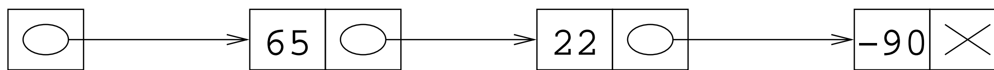
ottenuta seguendo i vari puntatori.

Per concludere: si faccia attenzione al fatto che il passaggio per valore può dare luogo a modifiche nella lista passata può creare degli errori, nel caso in cui si facciano modifiche alla lista puntata e ci si aspetta che non abbiano effetti sulla lista originaria.

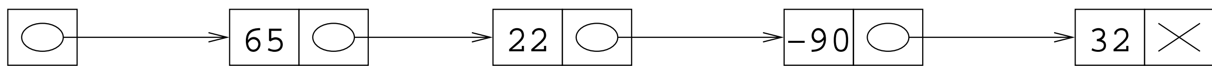
Aggiunta di un elemento in coda a una lista

Risolvere il seguente esercizio: si scriva una funzione che riceve come parametri una lista (ossia una variabile di tipo `TipoLista`) di interi e un intero; questa funzione modifica la lista aggiungendo l'intero come ultimo elemento.

Per risolvere questo esercizio, consideriamo la rappresentazione grafica della memoria prima e dopo l'aggiunta. Usiamo la lista `(65 22 -90)` come esempio di lista, e il valore `32` come valore da aggiungere in coda.



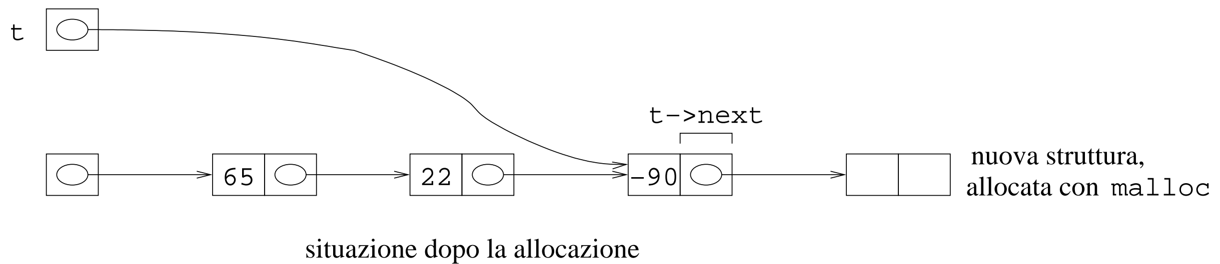
situazione iniziale



situazione finale

Dal momento che la lista è rappresentata usando delle strutture, e la lista va modificata, alcune di queste strutture devono essere modificate. Per modificare una struttura occorre sapere il suo indirizzo. Cerchiamo quindi di capire quali indirizzi ci servono. Dato che dobbiamo modificare l'ultima struttura della lista, ci serve il suo indirizzo. Facciamo quindi una scansione della lista, fino ad arrivare ad avere l'indirizzo dell'ultima struttura. Come al solito, se usiamo un puntatore `t` per scandire la lista, questo punta all'ultimo elemento quando `t->next` vale `NULL`.

Facciamo quindi una scansione della lista, portando avanti il puntatore `t` fino a che `t->next` vale `NULL`. A questo punto, `t` è l'indirizzo dell'ultima struttura della lista. Possiamo quindi aggiungere il nuovo elemento: `t->next` deve diventare l'indirizzo della prossima struttura della lista, ossia di una struttura nuova che abbiamo creato appositamente. Quindi, allochiamo una struttura con la funzione `malloc`, e mettiamo in `t->next` il valore di ritorno della funzione, che è l'indirizzo della nuova struttura.



situazione dopo la allocazione

A questo punto, possiamo mettere i valori nella struttura. Si può fare in due modi:

```
/* primo modo */
t->next->val=32;
t->next->next=NULL;

/* secondo modo */
t=t->next;
t->val=32;
t->next=NULL;
```

È abbastanza evidente che sono due modi equivalenti.

La funzione ora è quasi completa. Mancano solo alcune cose:

1. la lista è passata per valore o riferimento?
2. la funzione va bene anche per casi limite (come per esempio la lista vuota)?

In effetti, si tratta dello stesso problema: nel caso in cui la lista è vuota, allora il puntatore al primo elemento della lista va modificato. Infatti, quando la lista è vuota vale `NULL`, quando si aggiunge un elemento in coda alla lista vuota deve puntare al primo (ed unico) elemento della lista, quindi non vale più `NULL`. Dato che questa variabile va modificata, occorre passarla per riferimento.

Si ricorda che la regola sul passaggio di liste a una funzione non è: "se la lista viene modificata, allora occorre passarla per riferimento". Questa regola non è esatta: vedere la pagina sul passaggio di liste per valore nel caso ci siano dei dubbi. La regola, al contrario, è: *se il puntatore iniziale della lista viene modificato (ossia viene fatto puntare a un altro indirizzo) allora va passato per indirizzo*. Quando si aggiunge un elemento a una lista non vuota, allora il puntatore iniziale contiene sempre l'indirizzo della prima struttura della lista, per cui non viene modificato. Quindi, se si sa con certezza che la lista non è vuota, si può anche passare per valore.

D'altra parte, se la lista è inizialmente vuota, il puntatore `l` contiene `NULL` prima della chiamata, mentre dopo la chiamata contiene l'indirizzo della prima struttura della lista. Quindi, è stato modificato. Dato che la modifica deve essere visibile dal programma principale, occorre passare la lista per indirizzo.

Se poi controlliamo il comportamento della funzione nel caso di lista vuota, ci aggorgiamo che non possiamo semplicemente fare una scansione. Infatti, una volta fatto `t=*pl` per iniziare la scansione, il valore di `*pl` non viene più modificato. Mettiamo quindi una istruzione condizionale per verificare se `*pl` vale `NULL`. In questo caso, la aggiunta di un elemento consiste nella allocazione di una struttura, il cui indirizzo iniziale va in `*pl`.

Si può facilmente verificare che, togliendo questo controllo alla funzione riportata qui sotto (contenuta nel programma incoda.c), si verifica un errore quando si tenta di aggiungere un elemento in coda a una lista vuota.

```
void InserisciCodaLista(TipoLista *pl, int e) {
  TipoLista t;

                        /* caso lista inizialmente vuota */
  if(*pl==NULL) {
    *pl=malloc(sizeof(struct NodoLista));
    (*pl)->val=e;
    (*pl)->next=NULL;
    return;
  }

                        /* caso lista con almeno un elemento */
  t=*pl;

        /* vado avanti fino alla fine della lista */
  while(t->next!=NULL)
    t=t->next;

        /* qui t punta all'ultima struttura della lista: ne
        creo una nuova e sposto il puntatore in avanti */
  t->next=malloc(sizeof(struct NodoLista));
  t=t->next;

        /* metto i valori nell'ultima struttura */
  t->val=e;
  t->next=NULL;
}
```

Lettura lista da file, in ordine

Un meccanismo possibile per leggere una lista da file, rispettando l'ordine degli elementi sul file, è quello di iniziare con una lista vuota, e poi aggiungere gli elementi letti da file sempre in coda alla lista. Possiamo quindi usare la funzione di aggiunta in coda.

Qui sotto riportiamo la sola funzione `main` del programma leggiavanti.c. La funzione `InserisciCodaLista` è descritta nel dettaglio nella pagina precedente (aggiunta di elementi in coda a una lista). Più avanti vedremo come questo metodo di lettura da file ha degli svantaggi, e analizzeremo una soluzione alternativa.

```
int main() {
  FILE *fd;
  int res;
  int x;

  TipoLista l;


                /* apre il file */
  fd=fopen("lista.txt", "r");
  if (fd==NULL) {
    perror("Errore in apertura del file");
    exit(1);
  }


                /* inizializza la lista */
  l=NULL;
```

```
                /* legge fino alla fine del file */
  while(1) {
                /* legge un elemento, esce se non c'e' */
    res=fscanf(fd, "%d", &x);
    if( res!=1 )
      break;

                /* l'elemento letto lo metto in coda */
    InserisciCodaLista(&l, x);
  }

  fclose(fd);


                /* stampa la lista */

  StampaLista(l);


  return 0;
}
```

Lettura lineare di lista da file

Il metodo di lettura in ordine visto nella pagine precedente funziona perfettamente, ma non è efficiente. Il problema è il seguente: ogni volta che si legge un nuovo elemento da file, questo viene aggiunto in coda alla lista. L'operazione di aggiunta in coda, cosí come la abbiamo implementata, richiede la scansione di tutta la lista.

Supponiamo per esempio che il file da cui la lista va letta contiene 25 elementi. L'inserimento del primo non richiede scansione, dal momento che si tratta di inserire l'elemento nella lista vuota. D'altra parte, quando siamo arrivati, per esempio, al ventesimo elemento, per inserirlo in coda occorre prima fare la scansione di tutti i diciannove elementi precedentemente inseriti, solo per arrivare a conoscere l'indirizzo dell'ultima struttura della lista. Per inserire il ventunesimo occorre scandire da capo tutti i precedenti venti elementi che già stanno nella lista, ecc.

Supponiamo quindi di chiederci: quante operazioni vengono eseguite? Quando si inserisce l'i-esimo elemento vengono effettuate $c \times i$ operazioni, dove c è una costante (ossia: il numero di operazioni eseguite è proporzionale a i). Per inserire n elementi, occorre inserire il primo, poi il secondo, il terzo, ecc, quindi complessivamente servono:

```
costo inserimento   costo inserimento   ...
primo elemento      secondo elemento

c × 1            +  c × 2            + c × 3 + ... + c × n
```

Sfruttando una semplice regola algebrica, si può vedere come il valore di questa espressione sia proporzionale a n^2. Quindi, per inserire n elementi abbiamo eseguito un numero quadratico di operazioni. Si noti che, al contrario, la costruzione della lista in ordine inverso (partendo cioè dall'ultimo elemento) richiedeva solo un numero lineare di operazioni. Questo è dovuto al fatto che l'elemento da inserire veniva messo in testa alla lista, per cui non era necessaria nessuna scansione: per ogni elemento da inserire si eseguivano un numero lineare di operazioni.

Vediamo ora un metodo di lettura della lista da file che richiede solo un numero lineare di operazioni. L'idea di base è la seguente: per aggiungere un elemento in coda a una lista occorre avere l'indirizzo dell'ultima struttura della lista; possiamo quindi memorizzare questo indirizzo in una variabile puntatore `s`, invece di fare ogni volta la scansione per arrivare all'ultima struttura. Occorre fare in modo che `s` punti sempre all'ultima struttura della lista anche dopo aver inserito un elemento in coda alla lista.

Mettiamo quindi un ciclo, in cui a ogni iterazione leggiamo un elemento e lo aggiungiamo in coda. L'aggiunta in coda si fa cosí: se la lista è vuota, allora coincide con l'aggiunta in testa; se la lista non è vuota, allora `s` punta all'ultima struttura, per cui si crea una nuova struttura il cui indirizzo viene messo in `s->next` (che prima conteneva, per definizione, `NULL`). In altre parole, si fa `s->next=malloc(sizeof(struct NodoLista))`. Sia che la lista fosse vuota oppure no, `s` deve alla fine puntare all'ultima struttura della nuova lista (questo è necessario per poter inserire il prossimo elemento). Se la lista era vuota, l'ultima struttura della nuova lista è la prima, per cui si può fare `s=l`. Se la lista non era vuota, `s` era il puntatore all'ultima struttura, che ora è la penultima. Per farlo puntare all'ultima, basta portare il puntatore avanti, con `s=s->next`.

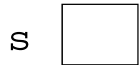
A questo punto, `s` punta all'ultima struttura della lista. Possiamo quindi mettere il valore letto da file in `s->val`, e poi in `s->next` ci va il valore `NULL` che indica che la lista è finita.

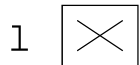
Di seguito riportiamo la funzione `main` del programma leggivero.c che legge una lista da file in ordine.

```
int main (int argn, char *argv[]) {
  FILE *fd;
  int res;
  int x;

  TipoLista l, s;

              /* controllo argomenti */
  if( argn-1!=1 ) {
    printf("Dare solo il nome del file come argomento\n");
    exit(1);
  }


              /* apre il file */
  fd=fopen(argv[1], "r");
  if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
  }


              /* inizializza la lista */
  l=NULL;

  while(1) {
                        /* tento di leggere un elemento */
    res=fscanf(fd, "%d", &x);
    if(res!=1)
      break;

                        /* se ci riesco, creo una struttura */

    if(l==NULL) {       /* la lista era vuota */
```

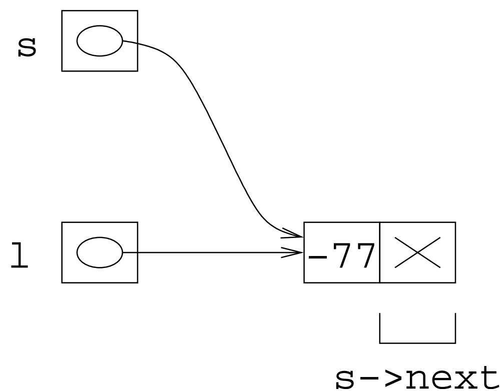
```
      l=malloc(sizeof(struct NodoLista));
      s=l;
    }
    else {              /* la lista contiene gia' elementi */
      s->next=malloc(sizeof(struct NodoLista));
      s=s->next;
    }

                        /* ora s punta all'ultima struttura della lista */
    s->val=x;
    s->next=NULL;
  }


                /* chiude il file */
  fclose(fd);


                /* stampa la lista */
  StampaLista(l);

  return 0;
}
```

Proviamo ora a simulare l'evoluzione della memoria. Consideriamo la situazione in cui il file contiene solo i tre interi `-77, 12, -34`. All'inizio, `l` contiene il valore `NULL`, mentre `s` non è stato ancora assegnato (ha un valore indefinito).



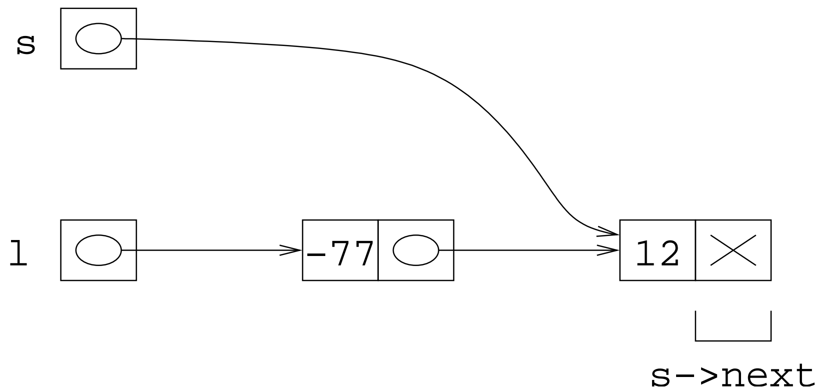


Quando si legge il primo intero, viene creata una nuova struttura, il cui indirizzo viene messo in `l`, e viene poi copiato in `s`. A questo punto, si riempiono i campi `val` e `next` della struttura puntata da `s`.

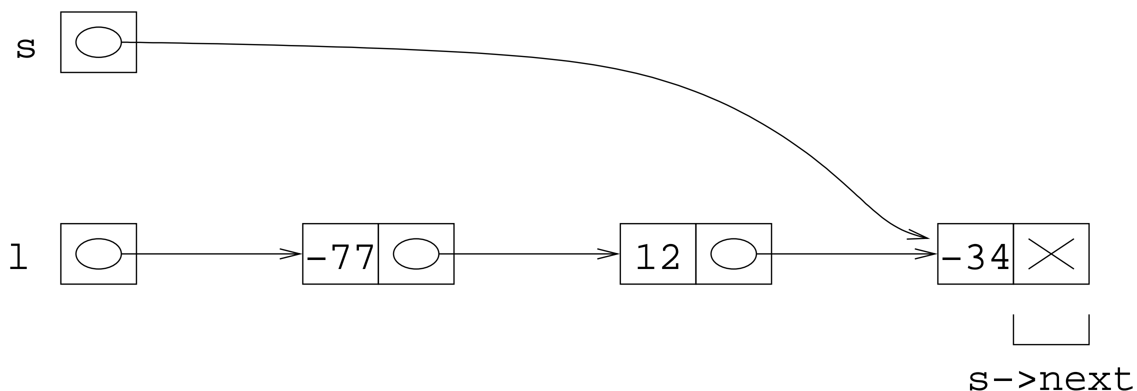


Si noti che `s` punta all'ultima struttura della lista (che è anche la prima, visto che la lista ha un solo elemento).

Dopo aver letto il secondo intero, si crea un nuova struttura, il cui indirizzo viene messo in `s->next` (la lista, a questo punto, non è vuota). Questo indirizzo va quindi a sovrascrivere il valore precedente, che era `NULL`. Si esegue poi la istruzione `s=s->next`, per cui `s` ora punta alla struttura che è appena stata creata. Si riempiono i campi `val` e `next` di questa struttura. Si noti che, alla fine della iterazione del ciclo, `s` contiene ancora l'indirizzo dell'ultima struttura della lista.



Si legge ora un altro elemento da file. Si crea una nuova struttura, il cui indirizzo va in `s->next`. Si porta ancora avanti il puntatore, per cui `s` punta all'ultima struttura della lista (quella appena creata), che viene riempita.



Il file, a questo punto è finito. Quando si esegue la istruzione `fscanf`, il valore di ritorno in `res` non vale `1`, per cui si esce dal ciclo. A questo punto, la lista contiene già i valori letti da file, nell'ordine.

È importante notare che, alla fine di ogni iterazione del ciclo, la variabile `s` deve sempre puntare all'ultima struttura della lista: questo è necessario per poter poi inserire il prossimo elemento nella successiva iterazione del ciclo.

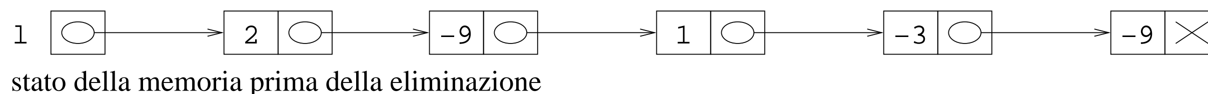
Proviamo ora a calcolare quante operazioni vengono fatte per leggere una lista di `n` elementi. Per ogni elemento, si crea una struttura, si porta avanti il puntatore, e si riempie la struttura. Quindi, abbiamo quattro istruzioni per ogni elemento della lista. Per leggere una lista di `n` elementi, abbiamo quindi eseguito `c × n` operazioni, dove `c` è una costante. Il metodo visto in questa pagina consente quindi di leggere una lista, in ordine, con un numero di operazioni proporzionali al numero di elementi della lista.

Eliminazione elementi in mezzo

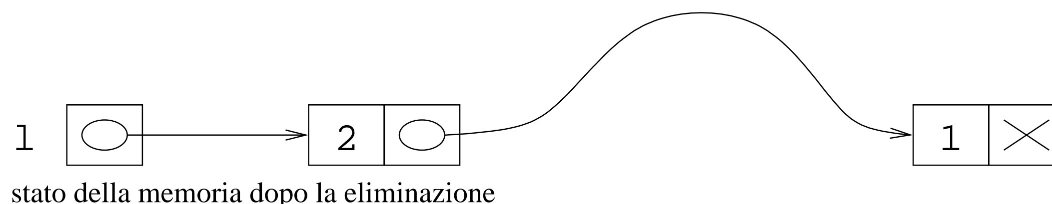
Questo esercizio consiste nella scrittura di una funzione che elimina, da una lista di interi, tutti i numeri negativi. Si noti che la eliminazione deve avvenire anche se ci sono elementi negativi all'inizio e alla fine della lista. Di seguito si riportano alcuni possibili file che contengono liste di esempio. Il programma di eliminazione dovrebbe leggere questi file in una lista, stampare la lista, eliminare i negativi e poi stampare di nuovo. Il confronto fra la stampa prima e dopo permette di capire se il programma funziona in modo corretto.

1. lista1.txt (lista vuota)
2. lista2.txt
3. lista3.txt
4. lista4.txt
5. lista5.txt
6. lista6.txt
7. lista7.txt
8. lista8.txt

Come al solito, confrontiamo lo stato della memoria prima e dopo l'eliminazione. Usiamo come lista di esempio `(2 -9 1 -3 -9)`. Nel caso in cui si devono eliminare elementi in mezzo a una lista, conviene usare un esempio in cui si siano almeno due elementi consecutivi da eliminare, perchè questo è un caso su cui è facile sbagliare.



stato della memoria prima della eliminazione



stato della memoria dopo la eliminazione

Dal confronto risulta abbastanza evidente che, se un elemento è negativo, allora serve conoscere l'indirizzo della struttura che lo contiene, per poterla deallocare. Questo indirizzo sta scritto nella struttura precedente. Dato che la struttura precedente va modificata (per ''saltare'' la struttura che ha l'elemento negativo) ci serve in effetti l'indirizzo della struttura che precede quella in cui si trova il valore negativo. Facciamo quindi una scansione della lista usando una variabile puntatore `s`. Non possiamo fare l'eliminazione quando `s->val` è negativo, dal momento che, in questo modo, `s` punta alla struttura che contiene il negativo, e non alla struttura precedente. Facciamo invece il controllo su `s->next->val`. Se questo numero è negativo, l'indirizzo della struttura che lo contiene è `s->next`.

L'eliminazione avviene mettendo in `s->next` l'indirizzo della struttura che *segue* quella che ha un numero negativo dentro. Questa struttura è ovviamente `s->next->next`. L'istruzione di eliminazione dalla lista è quindi `s->next=s->next->next`. Dal momento che la struttura va poi deallocata, occorre salvare il suo indirizzo in una variabile `r`, e poi fare la `free` una volta che è stata eliminata dalla sequenza di strutture che rappresentano la lista.

Questa è la struttura generale dell'algoritmo: si fa una scansione della lista, eliminando la struttura per la quale `s->next->val` è negativo. Occorre ora considerare alcuni casi particolari. Quando si fanno operazioni di cancellazione di elementi in mezzo a una lista occorre tenere presente cosa succede quando:

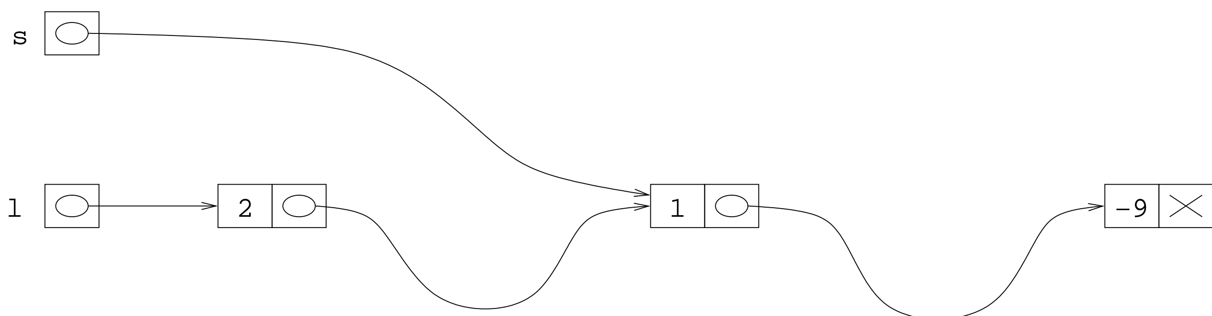
1. la lista è vuota
2. la lista contiene un unico elemento da eliminare
3. la lista inizia o finisce con un elemento da eliminare;
4. ci sono più elementi consecutivi da eliminare;

È abbastanza chiaro che la procedura di sopra non funziona se la lista è vuota: infatti, il primo accesso a `s->next->val`, quando `s` è `NULL`, causa un errore. Quindi, il caso di lista vuota va tenuto in considerazione separatamente. In particolare, se la lista è vuota non va ovviamente modificata (intuitivamente, la lista vuota a cui tolgo tutti gli elementi negativi, rimane vuota).

Il caso in cui la lista inizia con un elemento negativo (sia esso l'unico o no) è un altro caso in cui l'algoritmo di sopra non funziona. Infatti, al primo passo si controlla `s->next->val`, per cui il primo intero `s->val` non viene controllato affatto. Per eliminare il primo elemento della lista, basta controllare se `l->val` è negativo. Se lo è, si tratta di eliminare il primo elemento della lista.

Il caso in cui la lista termina con un elemento negativo, d'altra parte, non crea difficoltà: l'algoritmo funziona anche in questo caso.

Passiamo ora al caso in cui si sono più elementi negativi consecutivi. Supponiamo quindi di aver eliminato il primo elemento. Ora `s` punta alla struttura precedente a quello eliminato. Questa struttura ha un puntatore al successivo, ossia quello che prima seguiva l'elemento ora eliminato. Per poter eliminare anche questo elemento, ci serve che `s` punti alla struttura precedente. Ma `s` punta già alla struttura precedente! Quello che occorre quindi fare è controllare di nuovo `s->next->val`, ed eliminare la struttura successiva se questo valore è negativo. In altre parole, nella lista di esempio, dopo aver eliminato il quarto elemento (-4), ci troviamo nella seguente situazione:



Le linee curve indicano dove abbiamo eliminato delle strutture. Come si vede, subito dopo l'eliminazione della struttura che contiene `-3`, la struttura successiva a quella puntata da `s` contiene ancora un elemento da eliminare.

Per chiarire questo fatto, consideriamo l'assunzione che abbiamo fatto nel definire il ciclo: dato che facciamo il controllo su `s->next->val`, e tutte le strutture vanno controllate, è necessario che `s->next` assuma l'indirizzo di *tutte* le strutture che compongono la lista.

Una volta fatta una eliminazione, la struttura che va analizzata è quella che segue quella eliminata. Ma l'indirizzo di questa struttura si trova ora in `s->next`. Quindi, non è necessario, in questo caso, fare `s=s->next`, dato che `s` contiene già l'indirizzo della struttura precedente a quella da analizzare.

Il codice che risulta da queste considerazioni è il seguente:

```
              /* eliminazione elementi negativi */

if(l!=NULL) {

              /* eliminazione negativi dal secondo in poi */
  if(l->next!=NULL) {
    s=l;
    while(s->next!=NULL) {
      if(s->next->val < 0 ) {
        r=s->next;
        s->next=s->next->next;
        free(r);
      }
      else
        s=s->next;
    }
  }

              /* elimina primo elemento se negativo */
  if(l->val<0) {
    r=l;
    l=l->next;
    free(r);
  }
}
              /* fine eliminazione elementi negativi */
```

Nel caso in cui la lista è vuota, non si fa nulla. Nel caso in cui la lista ha almeno un elemento, facciamo la scansione. Ogni volta, `s` punta a una struttura della lista, e analizziamo (controlliamo se il campo `val` è negativo) la struttura successiva. Dal momento che occorre analizzare tutte le strutture, è necessario portare ogni volta avanti il puntatore; nel caso in cui una struttura viene eliminata, `s->next` già contiene l'indirizzo di una struttura che ancora va analizzata: in questo caso, il puntatore non va portato avanti.

Il ciclo `while` elimina dalla lista tutti gli elementi negativi che contiene, ma solo se non si trovano in prima posizione. In altre parole, se il primo elemento della lista è negativo, non viene eliminato. Mettiamo quindi un controllo sul primo elemento della lista.

Concludiamo notando come il controllo sul primo elemento va messo *dopo* il ciclo. Se infatti lo mettessimo prima, il programma non funzionerebbe sempre: se la lista iniziasse con due elementi negativi, eliminare solo il primo produrrebbe una lista che ha ancora un primo elemento negativo, il quale non verrebbe eliminato dal successivo ciclo `while`.

In effetti è anche possibile eliminare prima tutti gli eventuali elementi negativi con cui la lista inizia, e poi tutti gli altri. Dato che non si sa con quanti elementi negativi la lista potrebbe iniziare, occorrerebbe un ciclo in cui, a ogni iterazione, si elimina il primo elemento della lista se negativo, e si esce altrimenti.

Riportiamo qui sotto una parte del programma delmezzo.c in cui l'eliminazione degli elementi negativi è realizzata con una funzione. Dal momento che la lista potrebbe iniziare con un elemento negativo (che va quindi cancellato), il puntatore iniziale della lista potrebbe venire alterato dalla funzione, e questa modifica deve essere visibile al programma principale. La lista va quindi passata per indirizzo.

```
void EliminaNegativi(TipoLista *pl) {
  TipoLista s, r;


              /* lista vuota: niente da eliminare */
  if(*pl==NULL)
    return;

              /* eliminazione negativi dal secondo in poi */
  if((*pl)->next!=NULL) {
    s=*pl;
    while(s->next!=NULL) {
      if(s->next->val < 0 ) {
        r=s->next;
        s->next=s->next->next;
        free(r);
      }
      else
        s=s->next;
    }
  }

              /* elimina primo elemento se negativo */
  if((*pl)->val<0) {
    r=*pl;
    *pl=(*pl)->next;
    free(r);
  }

}
```

Inserimento in lista ordinata

Si risolva il seguente problema: sia data una lista di interi, ordinata in ordine crescente (prima il più piccolo, poi quelli più grandi); è dato un intero, che va inserito nella lista in modo tale che questa resti ordinata. Di seguito si danno alcuni file che contengono liste ordinate in questo modo: si provi a inserire l'elemento 2 in ciascuna di esse.

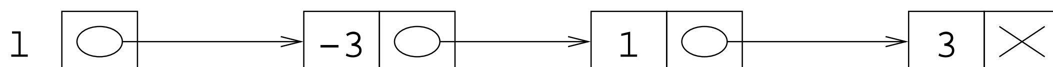
1. listaord1.txt
2. listaord2.txt
3. listaord3.txt
4. listaord4.txt
5. listaord5.txt
6. listaord6.txt
7. listaord7.txt (lista vuota)
8. listaord8.txt

Il problema consiste in un inserimento in mezzo a una lista. Si tratta infatti di trovare il primo elemento della lista che è superiore a quello dato, e inserire l'elemento prima di esso. Va anche tenuto conto che la lista può inizialmente essere vuota, e che la posizione in cui l'elemento va inserito può

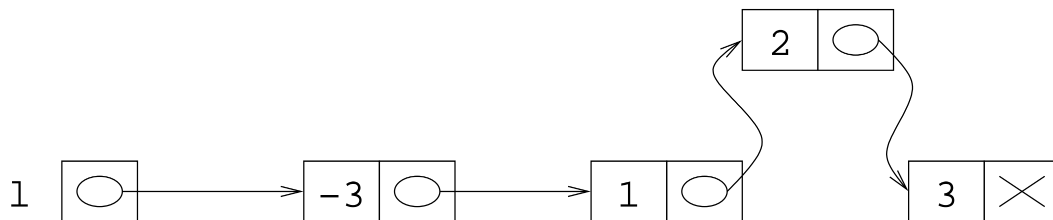
essere sia la prima che l'ultima.

Consideriamo inizialmente il caso in cui l'elemento va inserito in mezzo. Come si è detto, l'elemento va inserito subito prima del primo elemento della lista che è superiore ad esso. Questo significa che dobbiamo fare una scansione della lista, confrontando ogni elemento con quello da inserire. Va tenuto conto che, una volta trovata la posizione in cui l'elemento va inserito, questo va inserito prima. In altre parole, la situazione è la seguente:

2 (elemento da inserire)

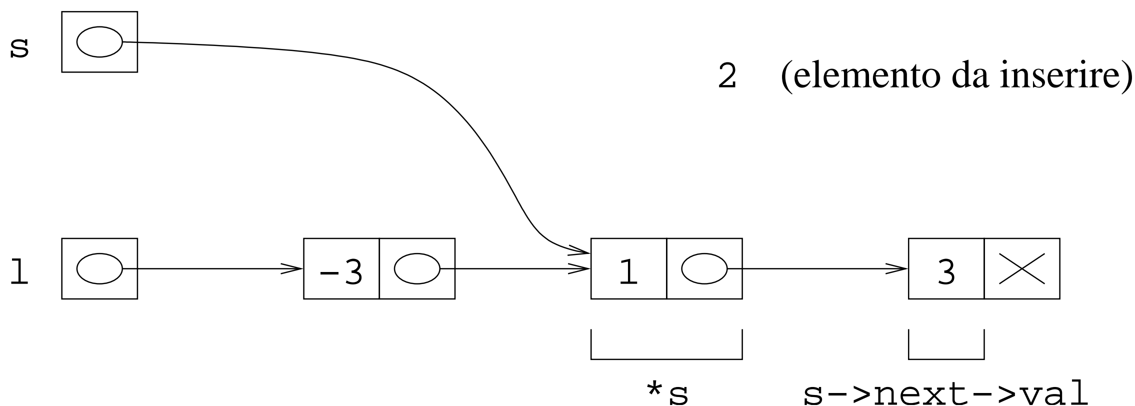


Dopo l'inserimento, la situazione deve essere invece la seguente:



In altre parole, ci serve confrontare l'elemento da inserire con ogni elemento della lista. Una volta trovato l'elemento maggiore, la struttura che viene modificata è la precedente. Dalla figura si vede chiaramente come la freccia della struttura che precede quella che contiene `3` è stata spostata, ossia il valore del puntatore è cambiato.

Quello che ci serve fare è semplicemente usare un puntatore `s` per fare la scansione della lista. Una volta trovato l'elemento superiore a quello da inserire, ci serve sapere l'indirizzo della struttura precedente. Per questo motivo, facciamo il confronto con `s->next->val` invece che con `s->val`. In questo modo, se il valore è superiore a quello dell'elemento, allora l'indirizzo della struttura da modificare è scritto in `s`.

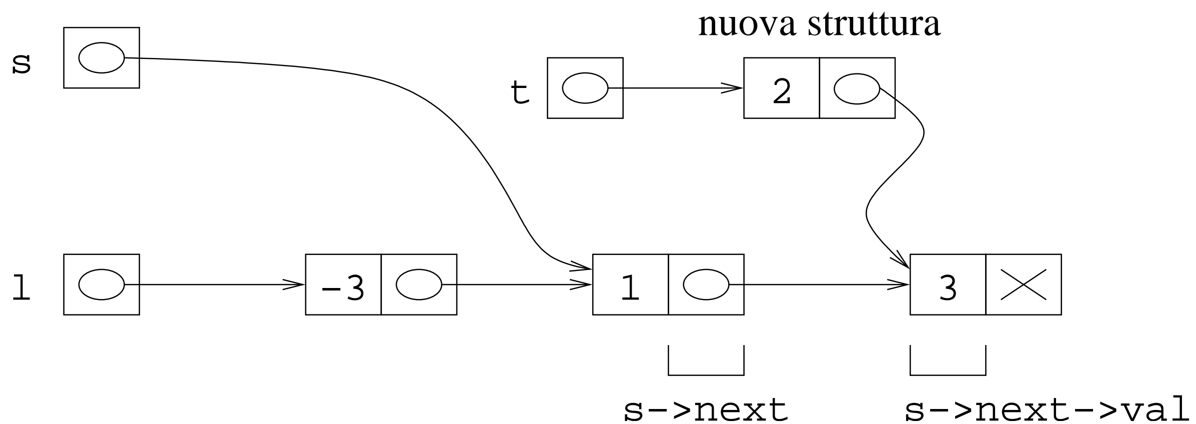


In altre parole, facendo la scansione fino al punto in cui `s->next->val` è maggiore dell'elemento da inserire, ci troviamo nella situazione di poter modificare la struttura precedente, perchè l'indirizzo di questa è scritto in `s`.

Per inserire l'elemento in mezzo, occorre allocare una nuova struttura, e metterci dentro il valore da inserire. Il campo `next` di questa nuova struttura deve contenere l'indirizzo della struttura che contiene l'elemento superiore. L'indirizzo di questa struttura si trova in `s->next`. Quindi, facciamo:

```
t=malloc(sizeof(struct NodoLista));
t->val=e;
t->next=s->next;
```

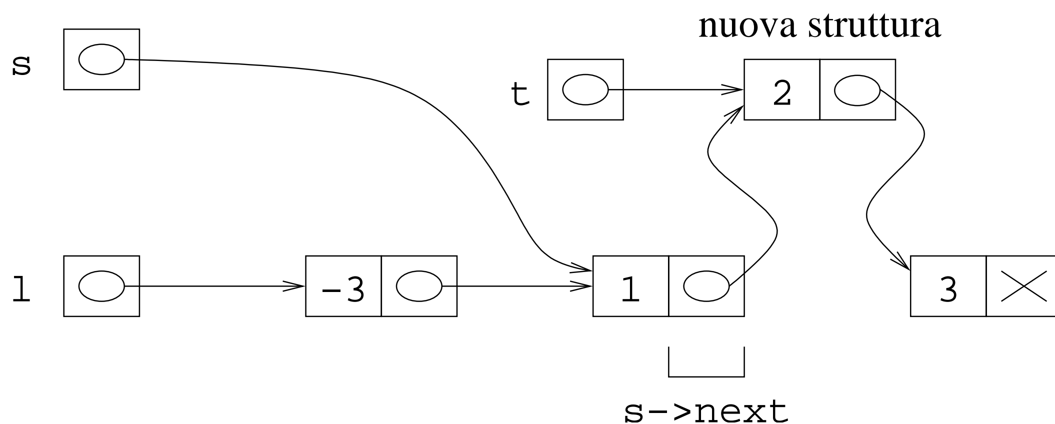
Questo produce la situazione seguente:



Ora, l'indirizzo della nuova struttura va messo in `s->next`. Abbiamo usato una variabile temporanea `t` per memorizzare questo indirizzo, quindi possiamo eseguire l'ultimo collegamento facendo:

```
s->next=t;
```

Questo produce la situazione seguente.



A questo punto la modifica alla lista è ultimata: a parte le due variabili temporanee `s` e `t`, la lista contiene effettivamente tutti gli elementi di prima più quello nuovo, e questi elementi si trovano in ordine crescente (a partire dal più piccolo).

Vediamo ora i casi particolari. La prima cosa da tenere presente è che la lista può essere vuota. In questo caso, l'algoritmo di sopra non funziona, dal momento che si deve fare un ciclo in cui si controlla `s->next->val`, e `s` vale inizialmente `NULL`. Il caso in cui la lista è vuota va quindi trattato separatamente. D'altra parte, questo caso è facile: basta allocare una struttura, il cui indirizzo va messo in `l`.

Il caso in cui l'elemento va iserito in prima posizione richiede anche esso un trattamento a sè. Infatti, alla prima iterazione del ciclo il confronto viene fatto fra l'elemento da inserire e il secondo, e comunque l'elemento viene inserito fra il primo e il secondo. Quindi, se la lista non è vuota, controlliamo se l'elemento va inserito in prima posizione, ed eventualmente lo inseriamo in testa.

L'ultimo caso da analizzare è quello in cui l'elemento va inserito in coda alla lista. Nel ciclo di scansione della lista, dobbiamo terminare se troviamo un elemento superiore a quello da inserire. Nel caso in cui si arriva alla fine della lista, l'elemento va inserito in coda.

Il programma completo inmezzo.c contiene la seguente funzione di inserimento in lista ordinata.

```
void InserisciListaOrdinata(TipoLista *pl, int e) {
  TipoLista s, r;

              /* la lista non contiene elementi */
  if(*pl==NULL) {
    *pl=malloc(sizeof(struct NodoLista));
    (*pl)->val=e;
    (*pl)->next=NULL;
    return;
  }


              /* l'elemento va inserito in prima posizione */
  if((*pl)->val>=e) {
    s=*pl;
    *pl=malloc(sizeof(struct NodoLista));
    (*pl)->val=e;
    (*pl)->next=s;
    return;
  }



                      /* la lista ha piu' di un elemento */
  s=*pl;
  while(s->next!=NULL) {

    if(s->next->val>e) {
      r=s->next;
      s->next=malloc(sizeof(struct NodoLista));
      s->next->val=e;
      s->next->next=r;
      return;
    }

    s=s->next;
  }


                      /* se arrivo qui, l'elemento va inserito in fondo */
  s->next=malloc(sizeof(struct NodoLista));
  s->next->val=e;
```

```
  s->next->next=NULL;


  return;
}
```

Inserimento delle medie

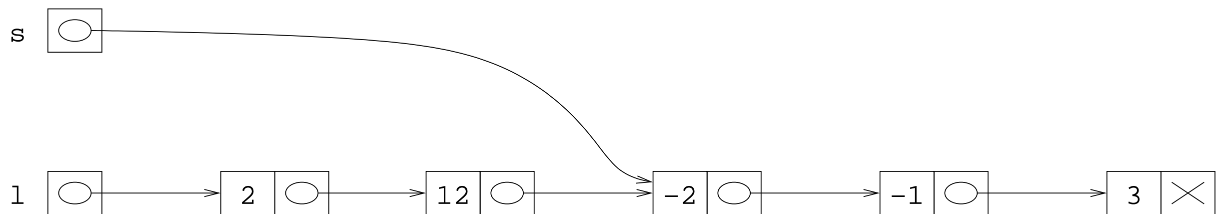
Si risolva il seguente esercizio: data una lista di interi, inserire *dopo* ogni elemento negativo, la media di tutti i numeri che lo precedono (includere il numero stesso nella media).

L'esercizio consiste ancora in un inserimento di elementi nella lista. Al contrario del caso precedente, l'elemento va inserito dopo quello negativo, per cui per esempio non è mai necessario inserire elementi all'inizio della lista.

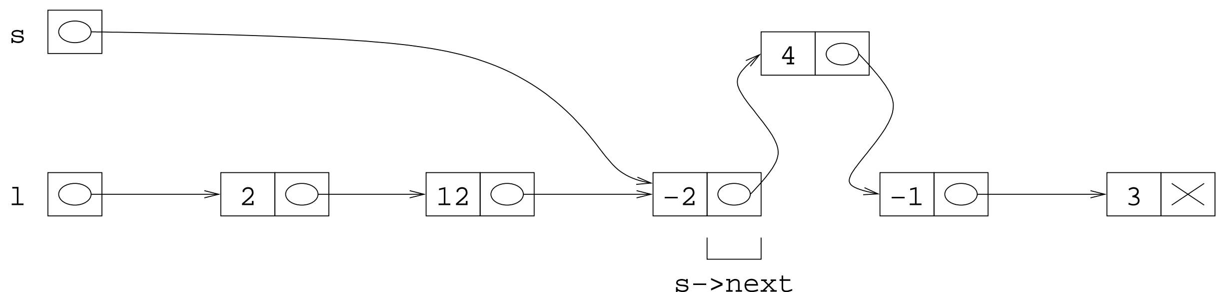
Un'altra conseguenza della definizione del problema è che ci basta avere un puntatore che si sposta su tutte le strutture della lista: se la struttura contiene un elemento negativo, è il campo `next` della struttura stessa che va modificato (al contrario, nel caso di inserimento in lista ordinata era il campo `next` della struttura precedente quello che andava modificato).

Facciamo quindi una scansione della lista, ossia un ciclo in cui un puntatore `s` contiene ogni volta l'indirizzo di una diversa struttura della lista. Usiamo inoltre due variabili `somma` e `numero` in cui mettiamo rispettivamente la somma e il numero degli elementi incontrati fino a questo momento nella scansione. Ogni volta che la variabile di scansione `s` punta a una struttura il cui campo `val` è negativo (ossia `s->val`), allochiamo una nuova struttura, e il suo indirizzo lo mettiamo in `s->next`.

La situazione che si viene quindi a creare quando `s` punta a una struttura con un campo `val` negativo è la seguente:



Dopo l'inserimento, lo stato della memoria deve essere invece quello che segue:



È quindi chiaro che dobbiamo allocare una nuova struttura, il cui campo `val` è la media e il cui campo `next` punta alla locazione in cui puntava prima `s->next`. Inoltre, `s->next` deve venire modificato in modo da puntare alla nuova struttura. A questo punto possiamo avanzare il puntatore. Si noti che la media può anche essere negativa. In questo caso, non va scritta nuovamente la media di seguito: infatti, la specifica del problema dice che la media va messa solo dopo agli elementi negativi che si

trovavano originariamente nella lista. Per questo motivo, dopo aver inserito il nuovo elemento, portiamo avanti il puntatore: si può verificare facilmente che, in questo modo, quando si inserisce un elemento, alla fine della iterazione del ciclo, il puntatore `s` è stato portato avanti di *due* posizioni, ossia punta al successivo elemento *della lista originaria*.

Il programma medie.c contiene la seguente funzione di aggiunta delle medie.

```
void InserisciMedie(TipoLista l) {
  TipoLista s, t;
  int somma, numero;

                      /* se la lista e' vuota, va lasciata cosi' */
  if(l==NULL)
    return;

                      /* inizializza somma parziale e numero elementi
                      incontrati finora */
  somma=0;
  numero=0;


                      /* scansione della lista */
  s=l;

  while(s!=NULL) {

                      /* aggiorna somma e numero di elementi trovati */
    somma=somma+s->val;
    numero++;


                      /* se l'elemento e' negativo, aggiunge la media
                      fra questo e il successivo */
    if(s->val<0) {
      t=s->next;
      s->next=malloc(sizeof(struct NodoLista));

      s=s->next;

      s->val=somma/numero;
      s->next=t;
    }

                      /* passa al successivo elemento */
    s=s->next;
  }
}
```

Si noti che la funzione va bene anche nel caso di lista vuota (non occorre fare niente), di lista con un solo elemento, e di lista in cui l'elemento finale è negativo. Questo può facilmente venire compreso graficando la evoluzione della memoria.

Per concludere, notiamo che la lista `l`, anche se viene modificata, è stata passata per valore. Questo si può spiegare con la regola del passaggio di puntatori a funzione:

> quando si passa un puntatore per valore, viene creata una copia locale del solo puntatore: gli elementi puntati non vengono copiati.

Dal momento che il puntatore iniziale della lista non viene mai modificato (punta sempre al primo elemento della lista, dal momento che nessun elemento viene mai inserito in testa), passarlo per valore o riferimento è in effetti equivalente. Quello che può succedere è che la prima struttura puntata, o una delle successive, vengono modificate. Dalla regola segue che la funzione lavora con una copia del puntatore, ma gli oggetti puntati sono quelli originali, cieè non sono copie. Quindi, la funzione modifica le strutture originali (non delle copie), ossia le stesse del programma principale.

Il resto del programma, per quello che riguarda le lista, è coperto dal libro di esercizi. La definizione di lista data nel libro è la seguente:

```
struct NodoLista {
  int val;
  struct NodoLista *next;
};

typedef struct NodoLista TipoNodoLista;
typedef TipoNodoLista *TipoLista;
```

Questa definizione differisce da quella usata sopra solo per il fatto che si dà un nome di tipo alla struttura `NodoLista`, ma per il resto è assolutamente identica.