Ricorsione

La ricorsione è il meccanismo di programmazione in cui una funzione fa al suo interno una chiamata a se stessa. I linguaggi moderni, quali il C, permettono infatti a una funzione di richiamare se stessa al loro interno (al contrario di linguaggi quali il Fortran, in cui questo non è possibile).

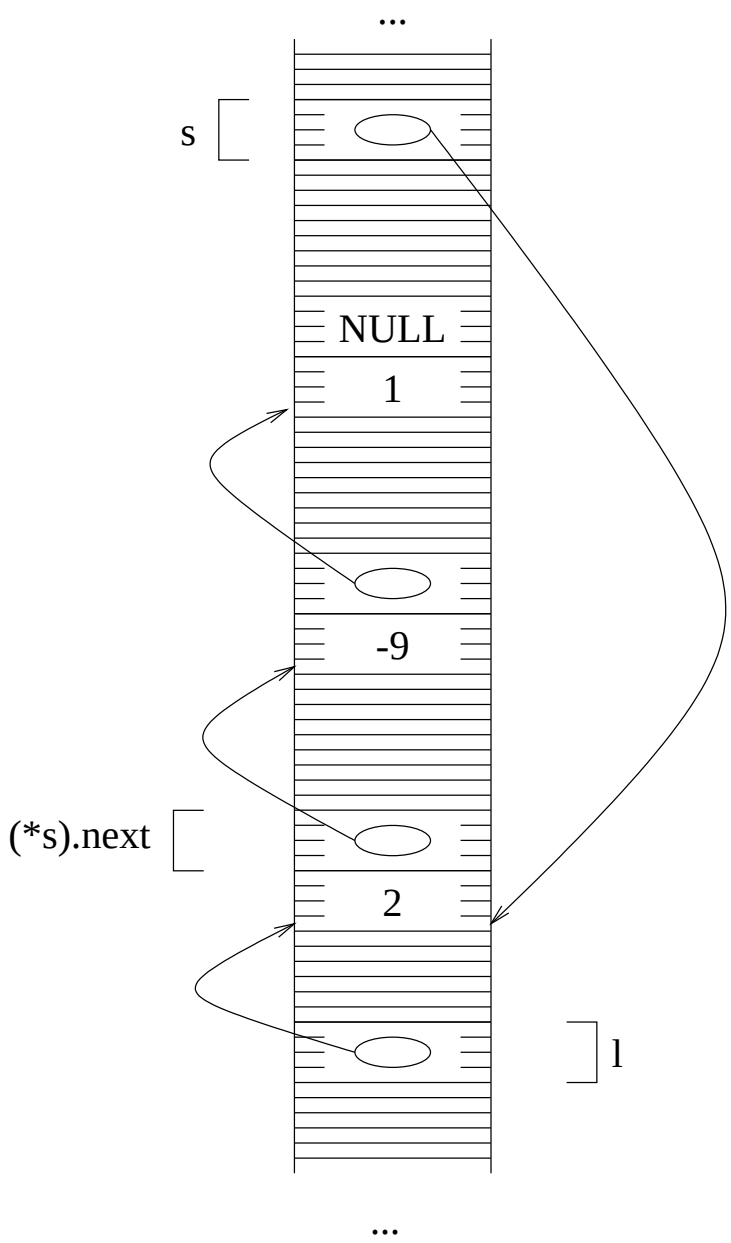
Il metodo di rappresentazione di variabili in memoria che permette di effettuare chiamate ricorsive è quello dello stack dei record di attivazione, in cui ogni attivazione di una procedura crea una nuova area di memoria in cui sono contenute le variabili locali di questa attivazione.

L'uso della ricorsione può rendere più semplice la programmazione. Nel caso delle lista, il vantaggio consiste nel fatto che funzioni ricorsive possono venire scritte tenendo conto della definizione ricorsiva di lista, per cui si ottengono facilmente dei programmi corretti senza dover simulare lo stato della memoria. Ci sono anche situazioni (per esempio, i programmi sugli alberi) in cui la ricorsione è il solo modo naturale di realizzare delle funzioni.

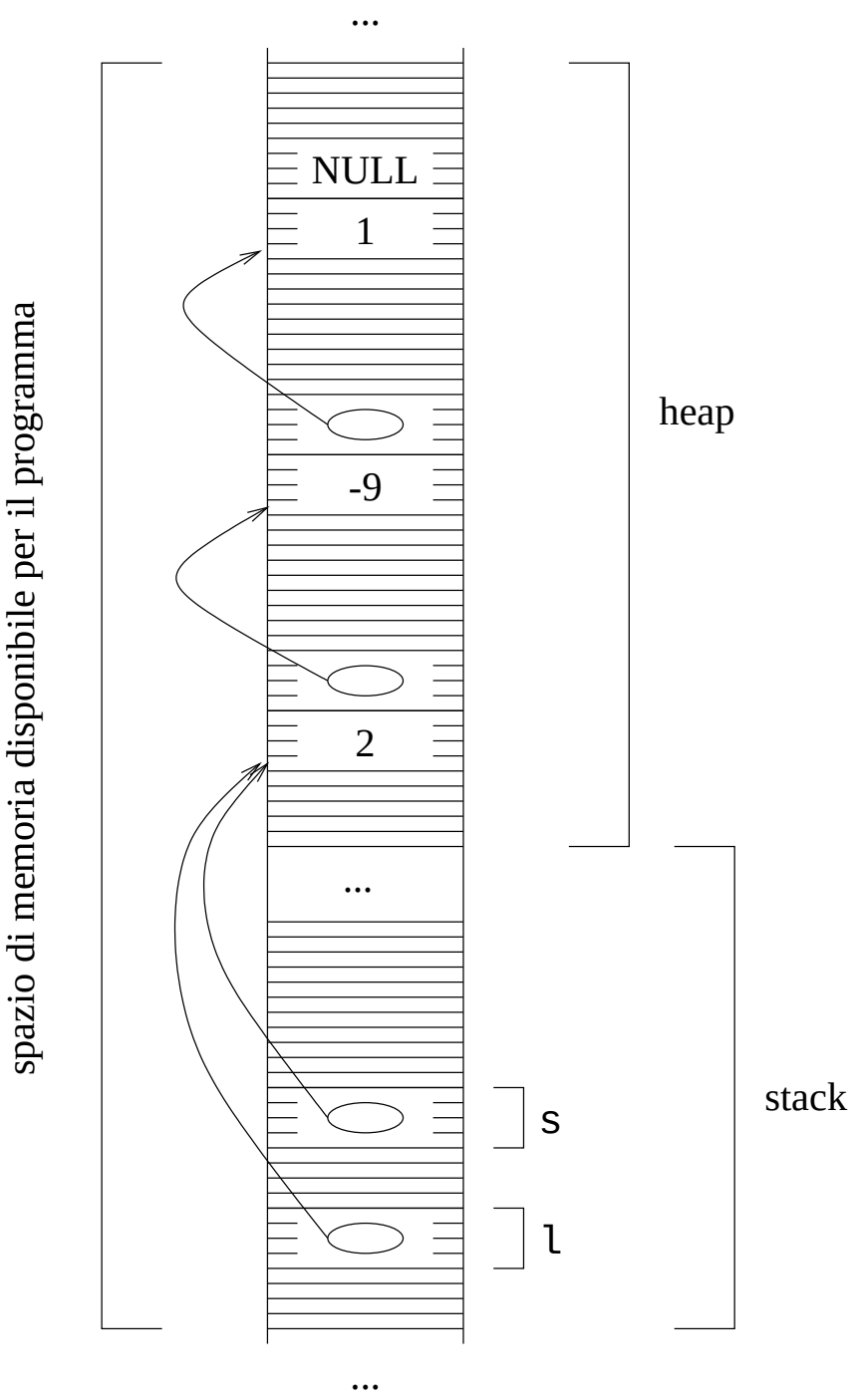
Stack, heap e record di attivazione

Stack, heap e record di attivazione

Finora abbiamo disegnato la memoria senza fare distinzioni fra le variabili e le zone di memoria allocate con `malloc`. Un esempio di figura fatta in questo modo è la seguente:



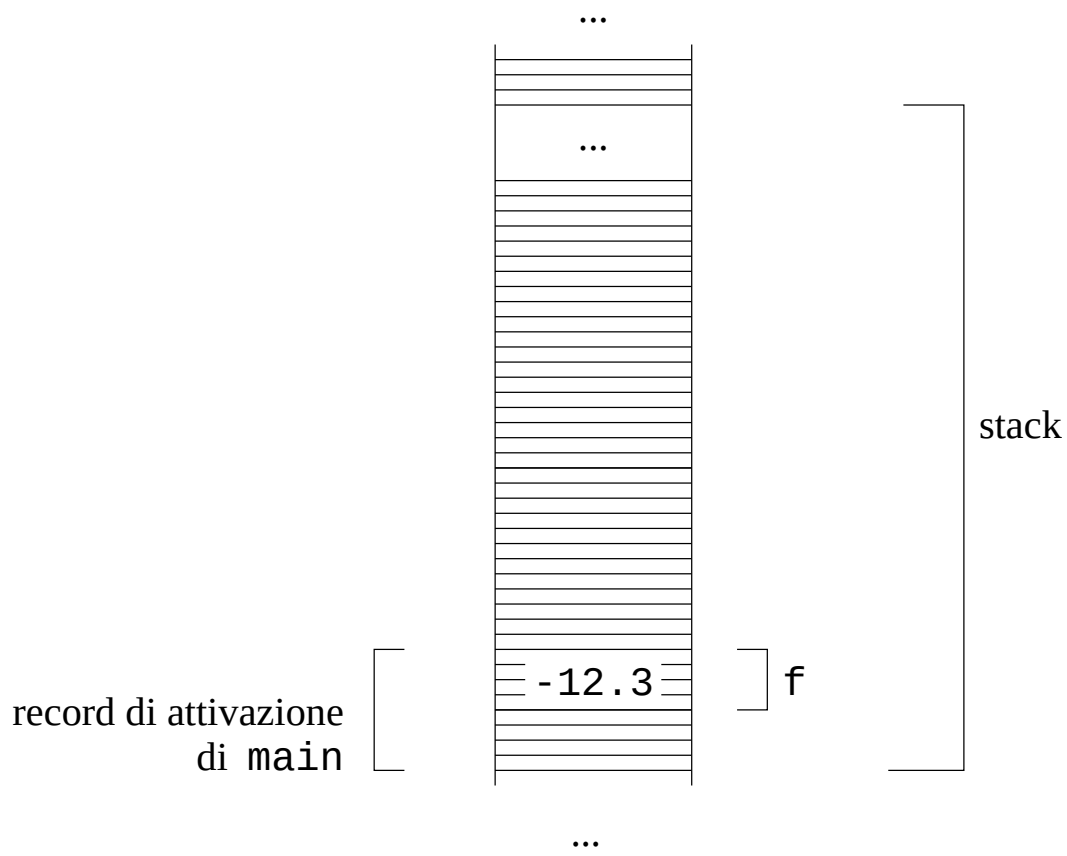
In realtà, le variabili del programma stanno in una zona di memoria detta *stack*, mentre le zone di memoria allocate con `malloc` stanno in una diversa zona, detta *heap*.



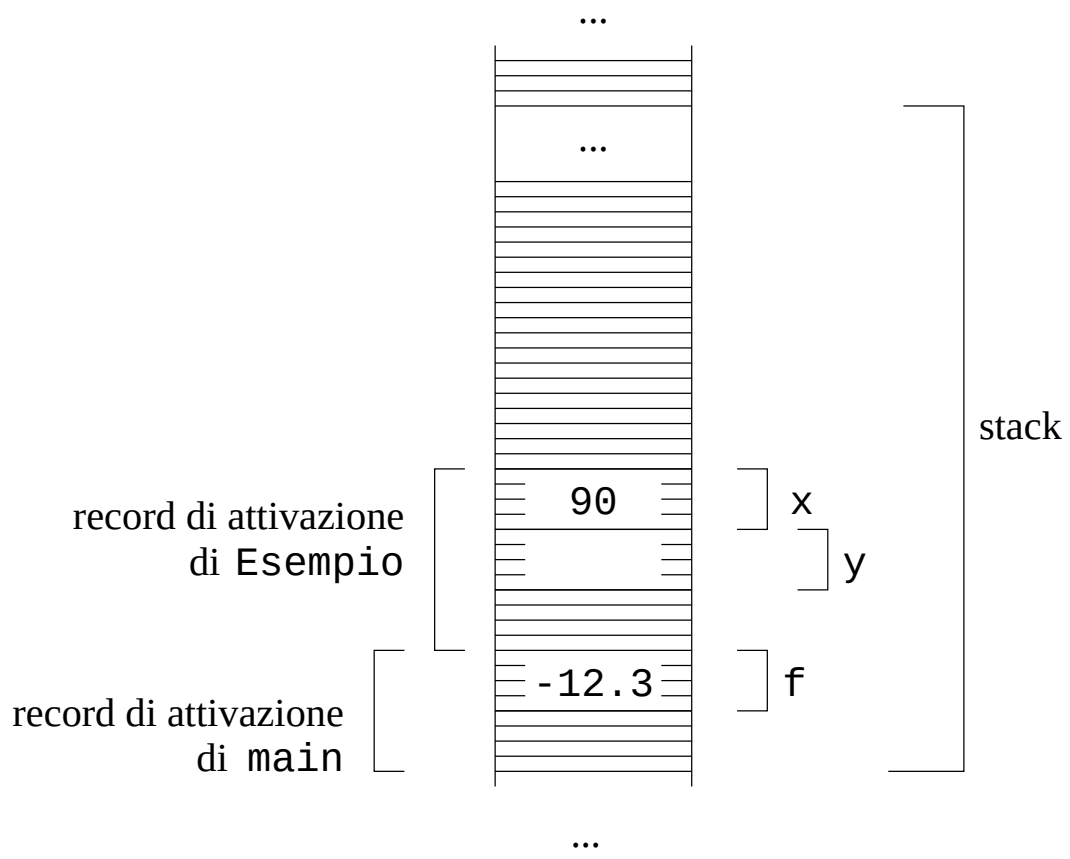
La struttura dello heap non ci interessa. Ci basta sapere che questa è la zona che contiene tutte le zone di memoria allocate con `malloc`. È invece importante capire in che modo è fatto lo stack. La struttura è la seguente: per ogni funzione, c'è una zona di memoria in cui si trovano le variabili locali della funzione. In particolare, la zona di memoria per la funzione `main` (ossia quella che contiene le variabili del programma principale) si trova nel punto più basso dello stack. Ogni volta che si chiama una funzione, viene creata una zona di memoria immediatamente sopra, e questa zona contiene le variabili locali della funzione chiamata (incluse le variabili che rappresentano i parametri). La zona di memoria che viene creata per una singola funzione viene detta *record di attivazione*. Consideriamo il seguente esempio:

```
void Esempio(int x) {
  int y;
}

void Altra(float m) {
}

int main() {
  float f=-12.3;

  Esempio(90);

  Altra(f);

  return 0;
}
```

Quando viene lanciata la funzione `main`, ossia quando si esegue il programma, si crea il primo record di attivazione, che è una zona di memoria che contiene le variabili locali del `main`.



Quando si esegue anche la funzione `Esempio`, viene creato un altro record di attivazione, che viene messo immediatamente sopra. Questo record di attivazione contiene sia le variabili che rappresentano i parametri formali della funzione (in cui vengono immediatamente scritti i valori passati) che le variabili locali del programma.

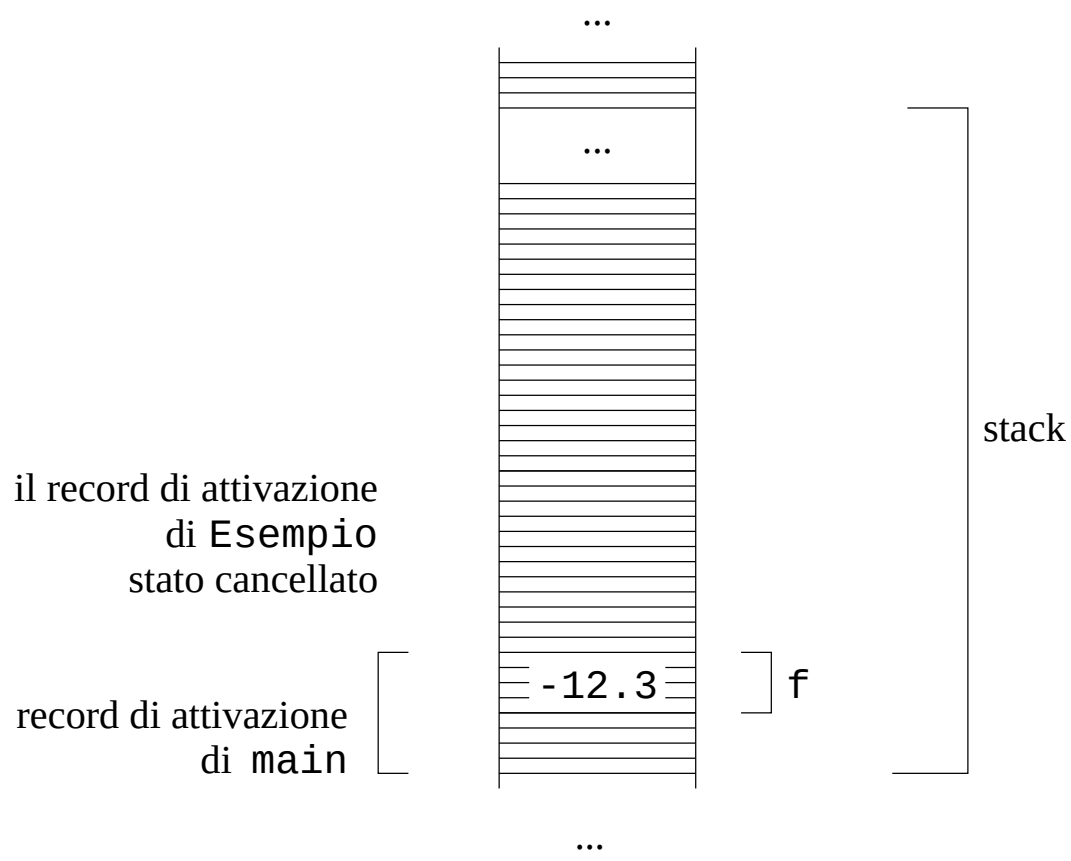


I record di attivazione contengono anche altre cose, ma questo per il momento non ci interessa.

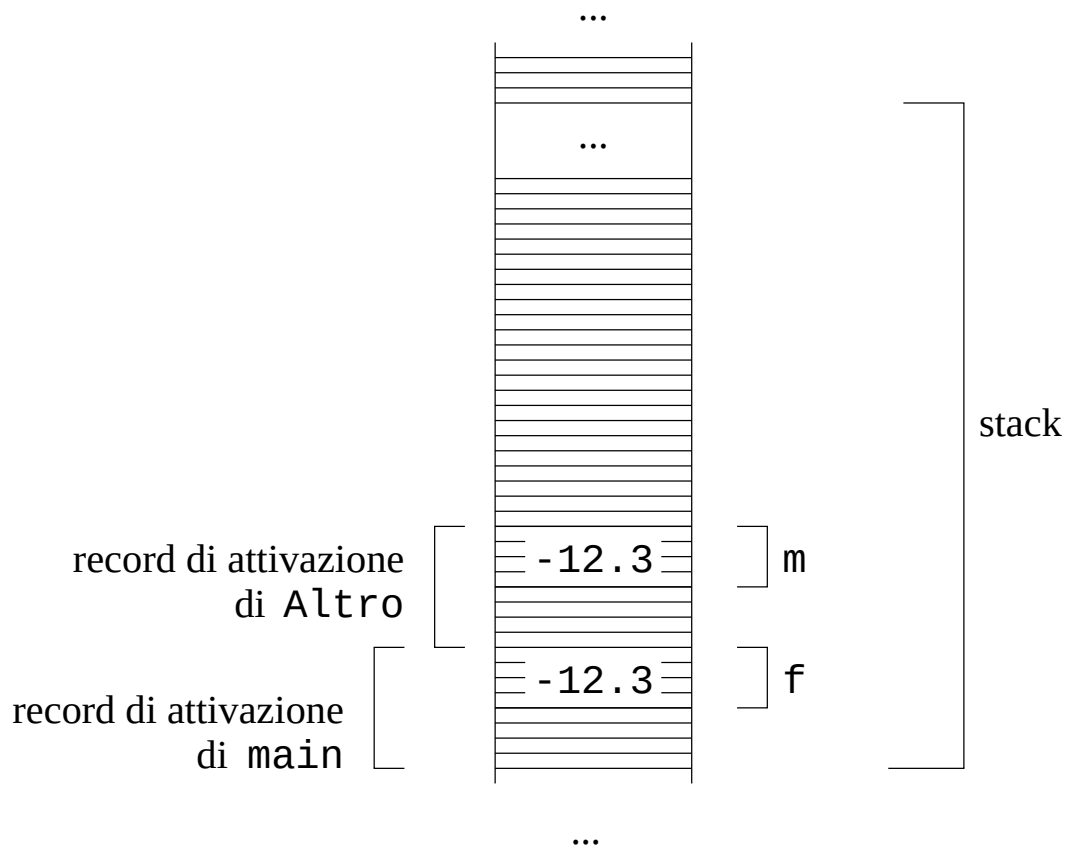
La regola precisa, per i record di attivazione, è:

> ogni volta che viene chiamata una funzione, si crea per essa un nuovo record di attivazione, che viene messo immediatamente sopra a quelli già presenti nello stack.

Ogni volta che una funzione termina, il suo record di attivazione viene cancellato. Nel caso di sopra, quando la funzione `Esempio`, termina, il record di attivazione che era stato creato per essa viene rimosso dallo stack:



Quando si chiama `Altra`, viene creato un nuovo record di attivazione, che si trova nella posizione più bassa libera dello stack. Dal momento che il record di attivazione di `Esempio` è stato cancellato, il nuovo record viene messo subito sopra quello di `main`.



Quando anche questa funzione termina, il suo record di attivazione viene cancellato. Il motivo della cancellazione è che le variabili locali non servono più, per cui si può liberare la memoria, che può quindi venire riusata per creare dei record di attivazione di altre funzioni (nell'esempio di sopra, parte della memoria del record di attivazione di `Esempio` veniva riusato per creare il record di attivazione di `Altro`).

Vediamo ora cosa succede quando si fanno chiamate a funzioni ricorsive. La regola da usare è sempre la stessa, ossia:

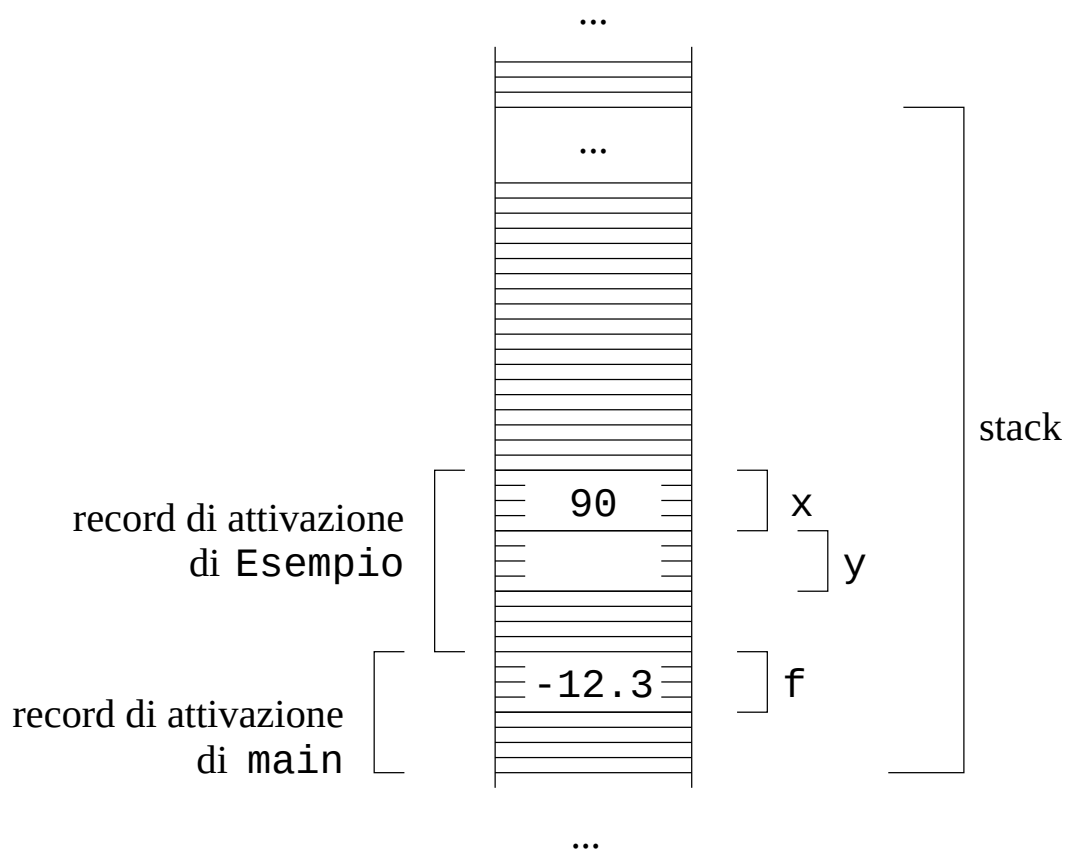
- quando di chiama una funzione, si crea per essa un record di attivazione nel punto più basso libero dello stack;
- questo record di attivazione è una zona di memoria che contiene i suoi parametri formali e le sue variabili locali;
- quando la funzione termina, il record di attivazione viene cancellato dallo stack, liberando quindi la memoria per successivi record di attivazione.

Si tratta solo di applicare questa regola. Supponiamo quindi che la funzione `Esempio` contenga una chiamata ricorsiva:

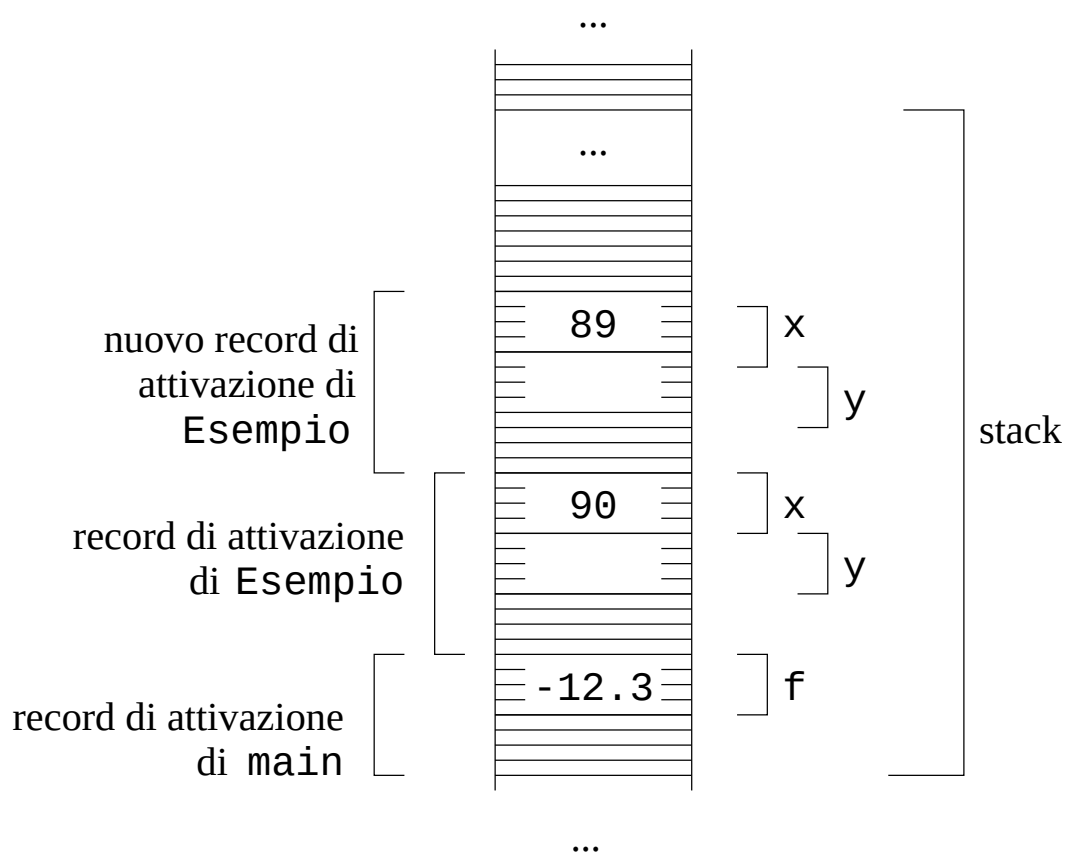
```
void Esempio(int x) {
  int y;

  Esempio(x-1);
}

int main() {
  float f=-12.3;
```

```
  Esempio(90);

  return 0;
}
```

Alla prima chiamata, si crea il record di attivazione per `Esempio`, per cui siamo nella solita situazione:

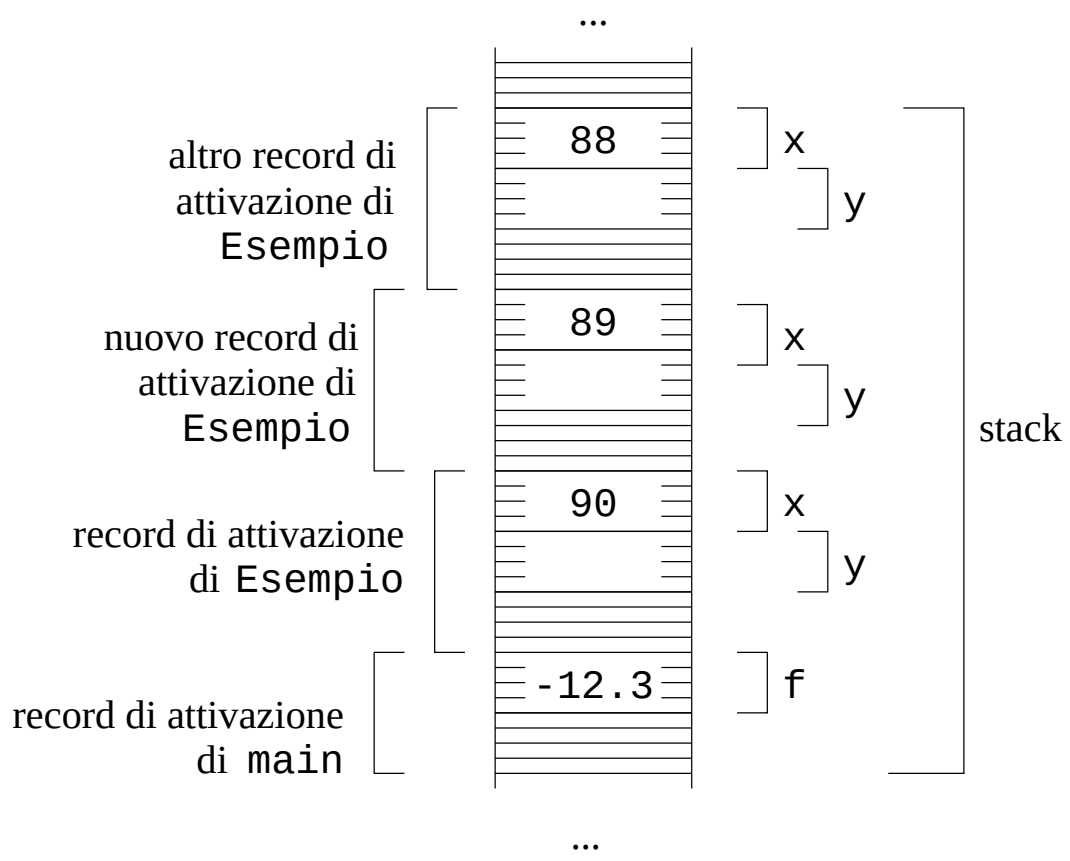


Viene fatta la chiamata ricorsiva. Come al solito, la regola va interpretata alla lettera: in particolare, la regola dice che si deve creare un nuovo record di attivazione. Dal momento che la chiamata ricorsiva avviene passando `x-1`, e che `x` vale `90`, il nuovo record di attivazione contiene `89` nella variabile `x`.



È importante notare come le variabili di `Esempio` siano state duplicate. In particolare, ogni record di attivazione contiene una sua versione di queste variabili. Si noti anche che le diverse variabili possono anche contenere valori differenti.

A questo punto si fa la terza chiamata. Si passa `x-1`. Quando vale `x`? In memoria abbiamo due versioni della variabile `x`: una che contiene `90` e una che contiene `89`. La regola, in questo caso, è che *in ogni esecuzione di una funzione, essa usa le variabili che si trovano nel record di attivazione che è stato creato per essa*. Dal momento che il record di attivazione che è stato creato per questa esecuzione di `Esempio` è quello più in alto dello stack, la `x` è che si usa è quella che contiene `89`. Quindi `x-1` vale `88`, ed è questo il valore che viene passato nella chiamata ricorsiva.



Nella nuova variabile `x` viene messo il valore che è stato passato, ossia `88`. La cosa va avanti cosí: ogni volta che si chiama una funzione, si crea per essa un nuovo record di attivazione che viene messo subito sopra i precedenti. Ogni volta che si attiva una funzione, questa ‘‘vede’’ le variabili che stanno scritte nel suo record di attivazioni. Per esempio, la prima attivazione di `Esempio` vede la `x` che contiene `90`, la seconda vede la `x` che contiene `89`, ecc.

In questo esempio, la funzione `Esempio` continua a richiamare se stessa fino a riempire lo stack con i suoi record di attivazione. Questo è un punto essenziale, quando si creano funzioni ricorsive: *prima o poi, la funzione deve smettere di richiamare se stessa*. Questo è un punto che è facile scordare.

Una prima verifica che occorre fare è quella di vedere se esiste una possibile esecuzione della funzione su cui non viene fatta la chiamata ricorsiva. In altre parole, devono esistere dei dati sui quali la funzione non chiama se stessa. Per esempio, la funzione di sopra non termina mai, perchè per qualsiasi valore del suo argomento `x`, viene comunque fatta una chiamata ricorsiva. Una alternativa possibile è la seguente:

```
void Esempio(int x) {
  int y;

  if(x==0)
    return;

  Esempio(x-1);
}

int main() {
  float f=-12.3;
```

```
  Esempio(90);

  return 0;
}
```

Qui si vede chiaramente che c;è un ‘‘punto di uscita’’ dalla funzione, ossia un percorso che non contiene la chiamata ricorsiva: se `x` vale `0`, allora si esce dalla funzione senza fare la chiamata alla funzione. Lo stack evolve in questo modo: si chiama `Esempio` con il valore `90`, e questa richiama se stessa con il valore `89`, che richiama ricorsivamente con il valore `88`, ecc, fino a che si arriva a una chiamata con il valore `0`. A questo punto, non si fanno chiamate ricorsive, ma si esce dalla funzione.

La regola, quando si esce dalla funzione, è semplice: si toglie il record di attivazione, e *si passa ad eseguire il resto della funzione chiamante*. In altre parole, ogni funzione viene attivata solo quando un’altra funzione contiene una chiamata ad essa (questo vale per tutte le funzioni, non solo quelle ricorsive). Alla fine della esecuzione, si deve eseguire quello che resta della funzione chiamante.

Nel nostro caso, la funzione con `x=1` chiama la funzione con `x=0`: quando questa è finita, occorre finire di eseguire la funzione con `x=1`. Nel nostro caso, la chiamata ricorsiva è l’ultima, per cui la funzione con `x=1` termina, e si torna alla funzione precedente, che è ancora la funzione `Esempio`, ma quella con `x=2`. È chiaro che quello che succede è che tutte le attivazioni della funzione, dato che sono già arrivate alla fine, terminano e i loro record di attivazione vengono tutti cancellati. Nella prossima pagine si vede un caso in cui questo non è vero.

Per concludere, va notato come quest’ultima funzione, anche se ha un punto di uscita, non necessariamente termina. Se per esempio si fa una chiamata `Esempio(-12)`, la chiamata ricorsiva avviene con un valore di `x` pari a `-13`, poi `-14`, ecc, e non si arriva mai a `0`. Questo è un altro punto importante sulla uscita della ricorsione: non solo ci deve essere un possibile punto di uscita, ma deve anche essere possibile dimostrare che, qualsiasi siano i parametri con cui la funzione viene chiamata, si arriva prima o poi a una attivazione in cui non si fa una chiamata ricorsiva. Nel nostro caso, una modifica che sicuramente termina è la seguente.

```
void Esempio(int x) {
  int y;

  if(x<=0)
    return;

  Esempio(x-1);
}

int main() {
  float f=-12.3;

  Esempio(90);

  return 0;
}
```

Il punto di ritorno

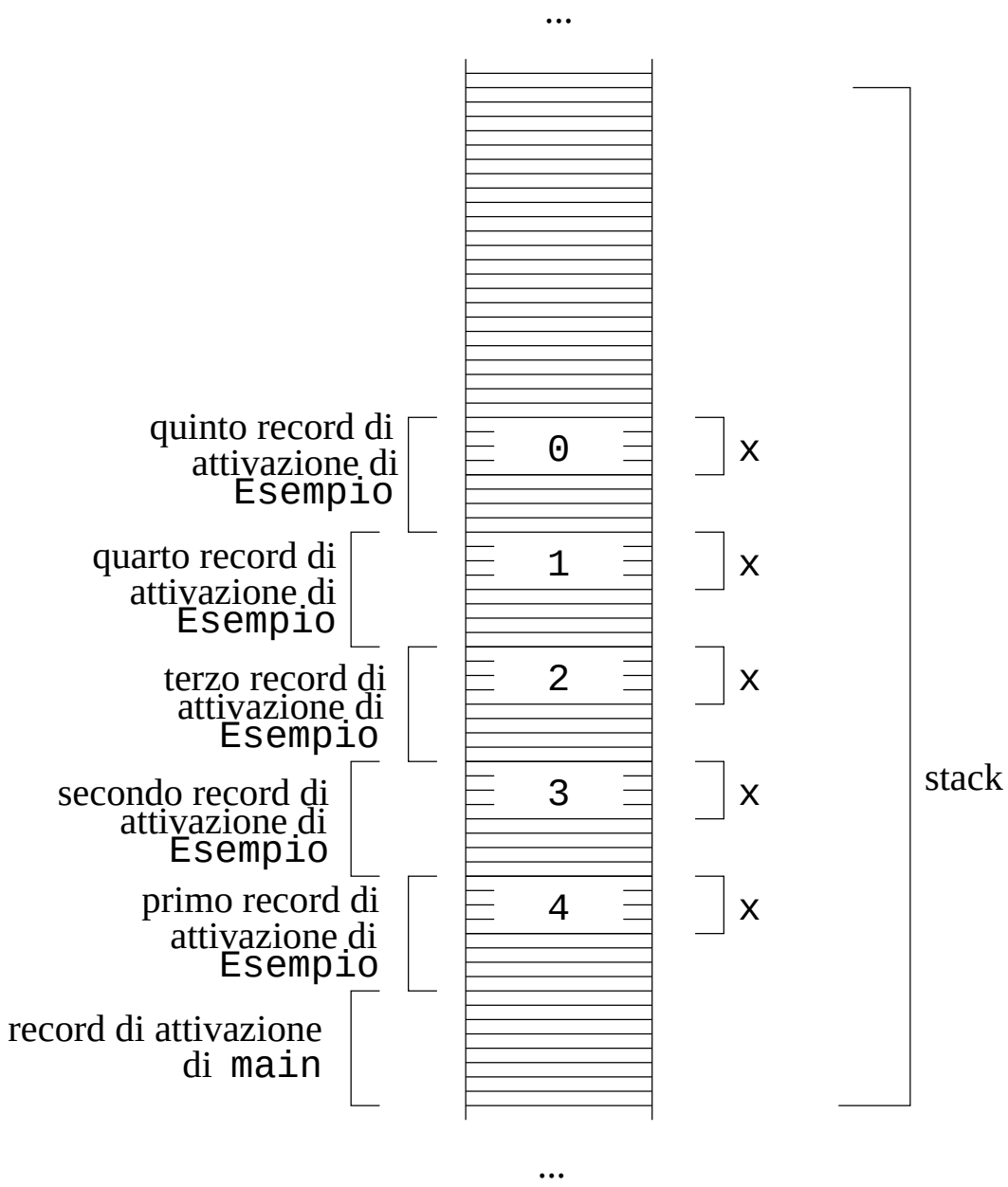
Il punto di ritorno

I record di attivazione delle funzioni non contengono soltanto le variabili locali delle funzioni.
Un'altra cosa che si trova nel record di attivazione di ogni funzione è il *punto di ritorno*, ossia il punto in cui la funzione è stata chiamata, e a cui deve ritornare. In altre parole: una funzione `A` viene attivata perchè un'altra funzione `B` la ha chiamata; quando `A` termina, occorre ricominciare con la istruzione di `B` che segue la chiamata di funzione. Il punto di ritorno di ogni funzione indica appunto quale è la istruzione da cui occorre ricominciare la funzione chiamante quando la funzione termina.

Consideriamo il seguente programma di esempio ritorno.c

```
/*
  Esempio che mostra il punto di ritorno.
*/

void Esempio(int x) {
  if(x<=0)
    return;

  printf("%d ", x);

  Esempio(x-1);

  printf("%d ", -x);
}

int main() {
  Esempio(4);

  return 0;
}
```

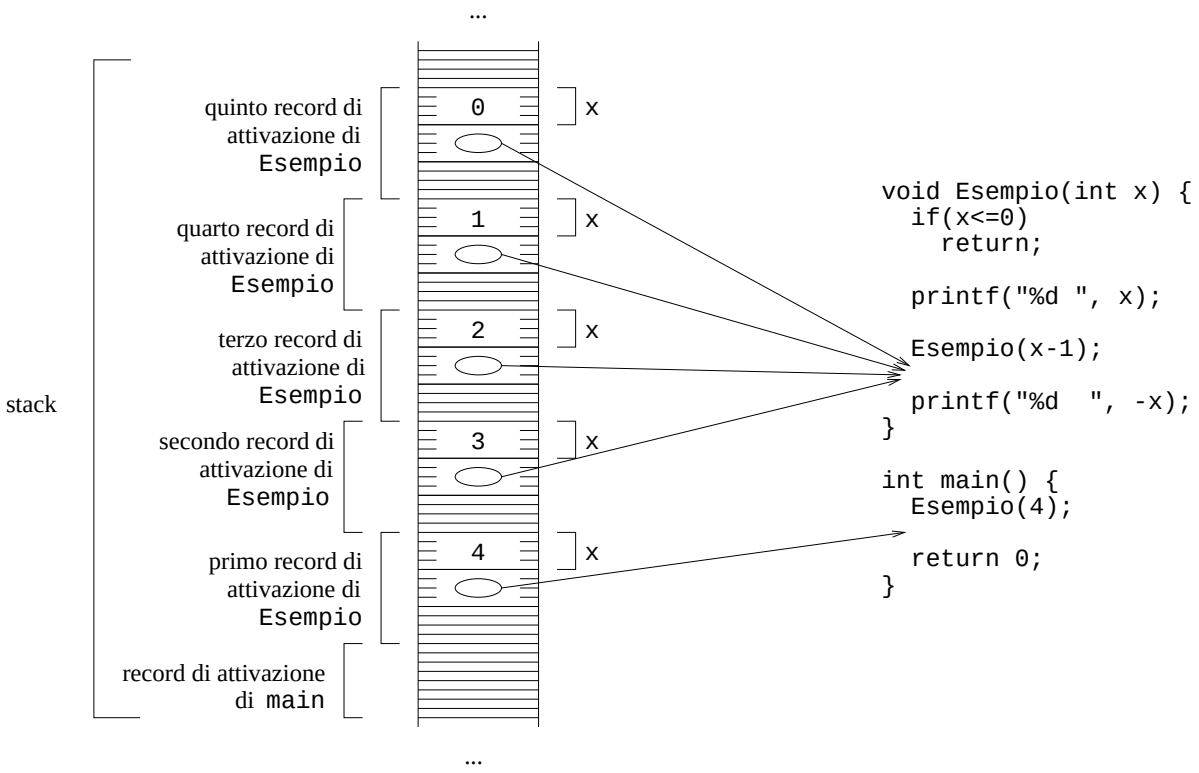
Ci chiediamo cosa fa il programma. La funzione viene chiamata con il valore `4`, per cui il primo record di attivazione contiene una variabile `x` il cui valore è `4`. Questo valore viene stampato. Viene poi chiamata ricorsivamente la funzione, passando il valore `x-1`, ossia `3`. Il nuovo record di attivazione contiene quindi una variabile `x` in cui c'è `3`, che viene stampato. Si arriva in questo modo a una chiamata in cui viene passato `0`. Fino a questo punto, sono stati stampati i valori `4 3 2 1`.



Quando la funzione viene chiamata con argomento `0`, si crea l'ultimo record di attivazione, quello in cui la `x` contiene `0`. Dato che la condizione `x<=0` è vera, si esce dalla funzione senza fare la chiamata ricorsiva. Cosa succede? Come per tutte le funzioni, si continua la esecuzione della funzione chiamante.

Questo significa che occorre ripartire con la istruzione che segue la chiamata di funzione. Il punto in cui occorre ricominciare viene memorizzato nel record di attivazione della funzione. In altre parole, il record di attivazione di una funzione non contiene solo le variabili locali, ma anche il punto in cui occorre ritornare quando la funzione stessa termina.

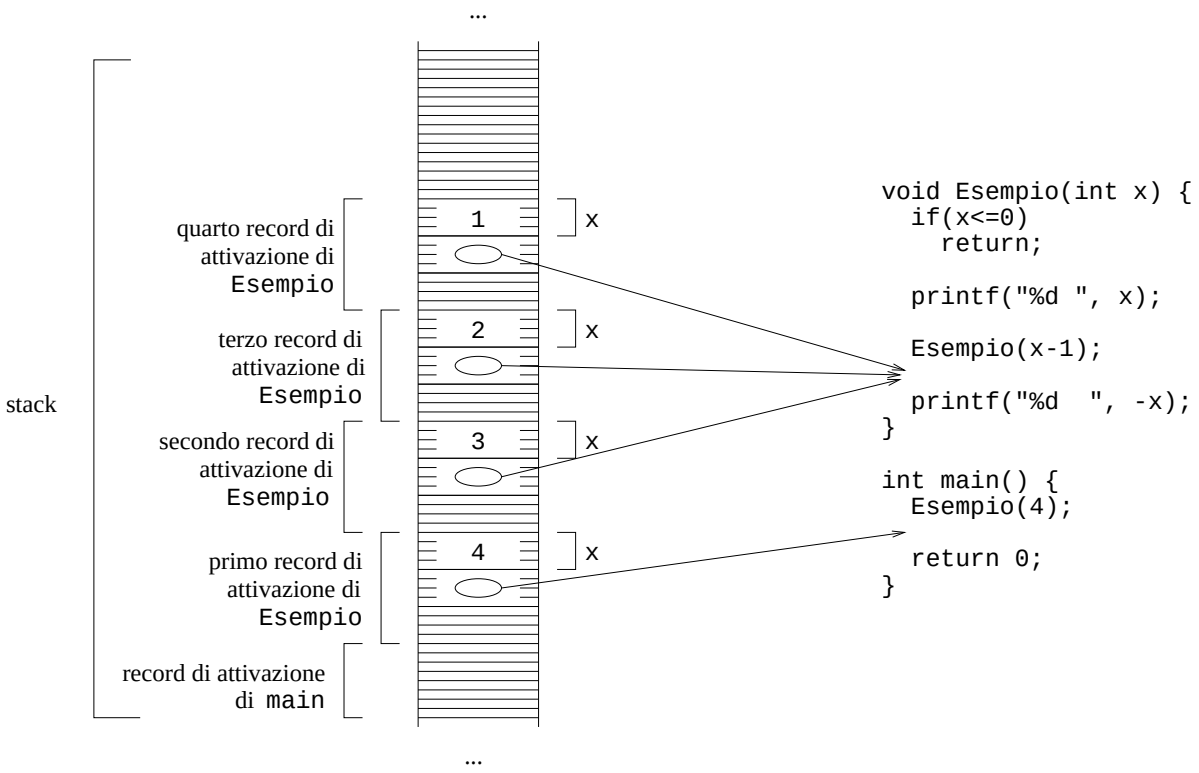
Il punto di ritorno, quando serve, lo indichiamo scrivendo il codice a fianco dello stack, e mettendo una freccia che parte dal record di attivazione e punta alla istruzione in cui la funzione è stata attivata. Nel nostro caso, abbiamo una situazione del genere:



Quando la funzione viene chiamata con parametro `0`, termina immediatamente. Quando una funzione termina, si fanno due cose:

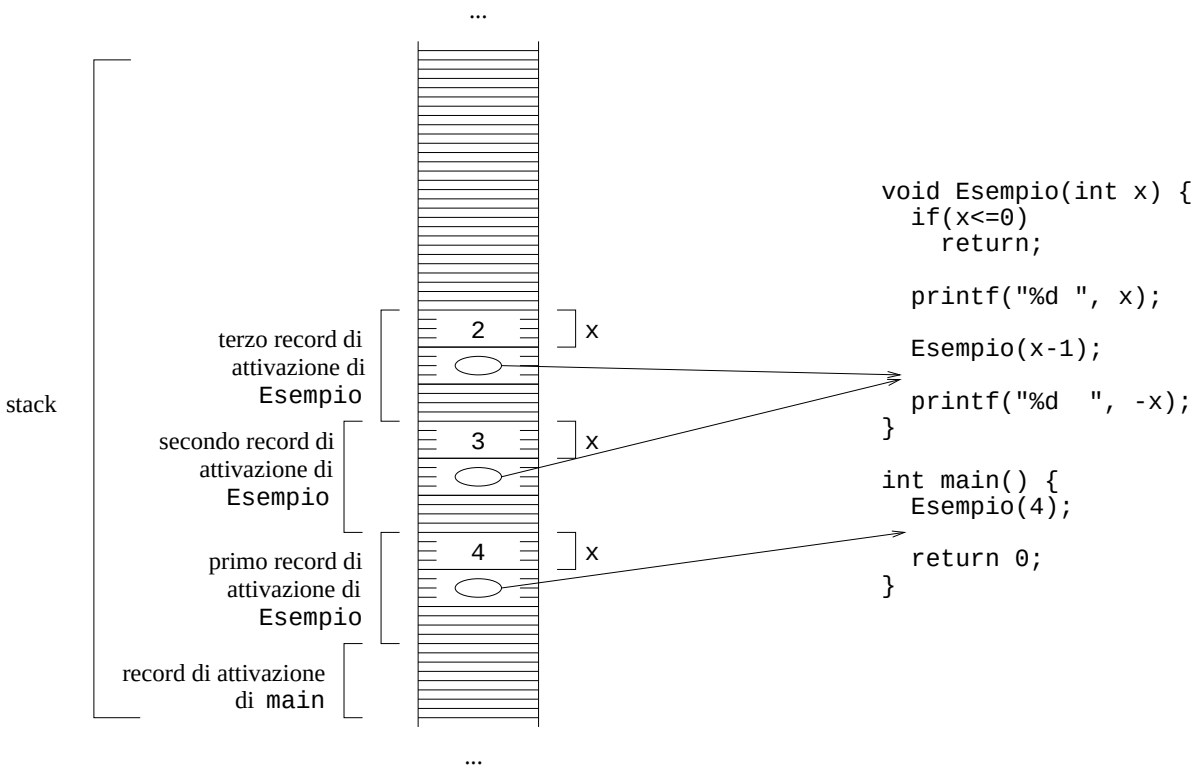
1. si dealloca il record di attivazione (si libera la memoria);
2. si ritorna ad eseguire a partire dal punto indicato dal punto di ritorno.

Questo significa che si delloca il record di attivazione che contiene la `x` che vale `0`, e si esegue la istruzione `printf("%d ", x);`. Questa istruzione usa il valore di `x` che si trova nel record di attivazione più in alto, per cui stampa `-1`, dato che lo stato dello stack, dopo la deallocazione dell'ultimo record, è il seguente:

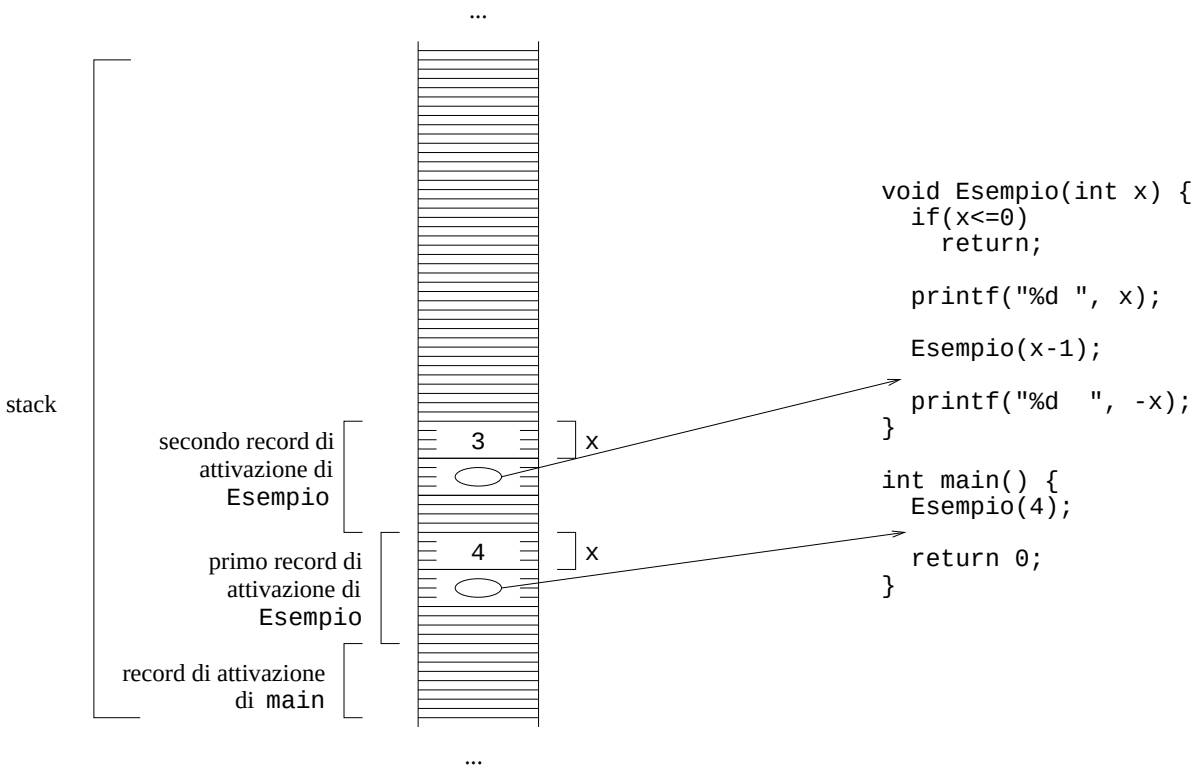


Vale la pena di notare che *le variabili che vengono usate sono sempre e solo quelle del record di attivazione più in alto*: infatti, quando si esegue una funzione, le variabili allocate per esse sono comunque quelle dell'ultimo record di attivazione creato.

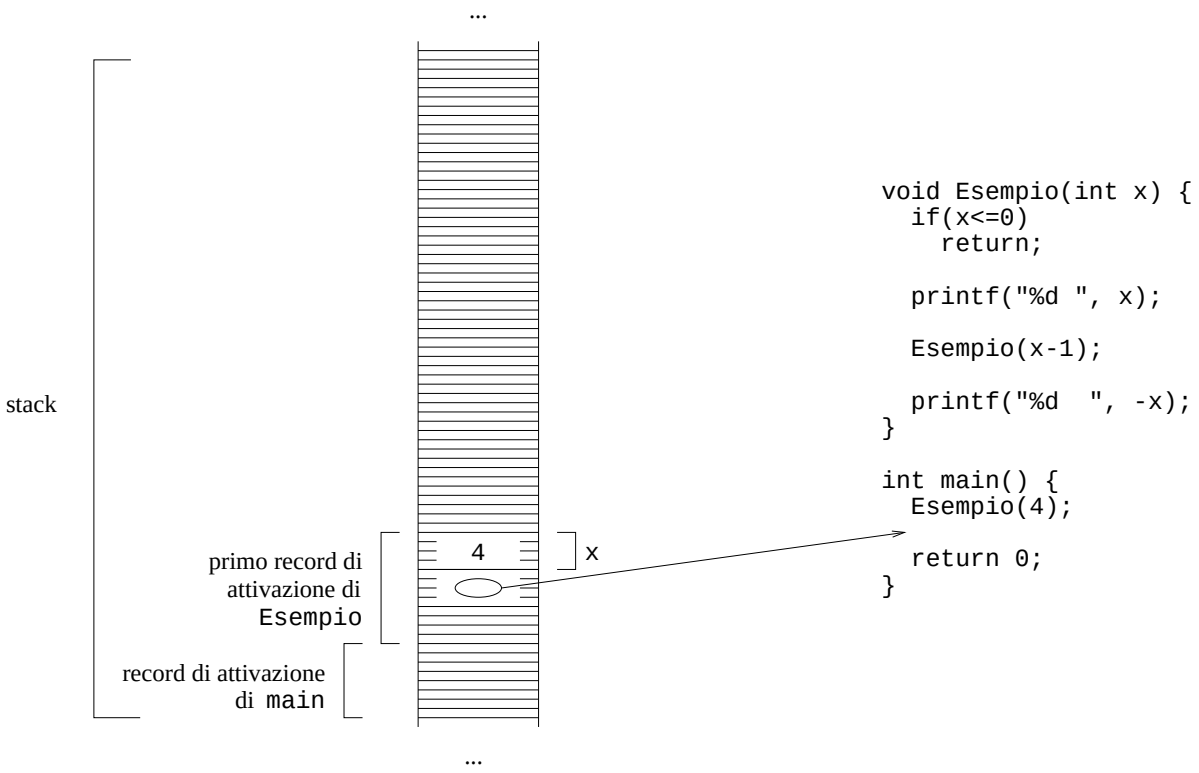
Dopo l'ultima istruzione di stampa, la funzione termina. Si dealloca il record di attivazione, e si riparte dal punto di ritorno scritto nel record di attivazione. Abbiamo quindi la seguente situazione, in cui si esegue la istruzione `printf("%d ", -x);`



La `x` che si usa è ancora quella del record che si trova in cima allo stack, per cui si stampa `-2`. La funzione termina, per cui si fa nuovamente la dellocazione, e si va al punto di ritorno. Ossia si esegue di nuovo la stampa, dopo aver dellocato il record.



Si stampa quindi il valore `-3`, dato che la `x` del record più in alto è `3`. Si fa di nuovo la deallocazione e si va al punto di ritorno:



Questo significa che si stampa `-4`, e si termina la funzione. Il record viene deallocato, e si torna al punto di ritorno, che questa volta è all'interno di `main`, per cui si esegue quello che resta del `main` (nell'esempio, non ci sono altre istruzioni, per cui il programma termina).

La serie di Fibonacci, implementata con una funzione ricorsiva

La serie di Fibonacci, con una funzione ricorsiva

Vediamo ora un esempio di uso della ricorsione. Implementiamo una funzione ricorsiva che stampa gli elementi della serie di Fibonacci fino a un certo valore. La serie di Fibonacci è una sequenza di numeri interi, i cui primi due sono 1 e 1. Ogni successivo elemento della sequenza si ottiene sommando i precedenti due elementi. Per esempio, il terzo numero della sequenza è la somma del primo e del secondo, ossia vale 2. Quindi, i primi tre elementi della sequenza sono 1 1 2. Il quarto si ottiene sommando gli ultimi due, cioè 1 e 2, e quindi vale 3. Dato che ora abbiamo 1 1 2 3, il quinto si ottiene sommando gli ultimi due, e quindi vale 2+3=5.

Possiamo quindi realizzare una funzione ricorsiva che prende come argomenti gli ultimi due elementi calcolati fino ad ora, calcola la somma, la stampa (se è minore del massimo prefissato), e poi fa la chiamata ricorsiva.

Per quello che riguarda la chiamata ricorsiva, occorre tenere presente che la funzione deve prendere come argomenti gli ultimo due valori calcolati. Questo significa che il valore nuovo che abbiamo calcolato è in effetti l'ultimo valore calcolato, mentre quello che prima era l'ultimo ora è il penultimo, semplicemente perchè ora ne abbiamo trovato uno nuovo.

La funzione deve avere tre parametri, che sono gli ultimi due valori calcolati, e poi il limite (quello che dice a che punto ci dobbiamo fermare). Come nome per le prime due variabili usiamo i nome `penultimo` e `ultimo`, che dice appunto quali sono gli ultimi due valori trovati. Per il valore

massimo da calcolare usiamo un parametro che chiamiamo `limite`.

La funzione cosa fa? Calcola il prossimo valore della serie, e questo si ottiene semplicemente sommando i due ultimi valori calcolati. Mettiamo questo nuovo valore trovato in una variabile, per esempio usiamo il nome di variabile `nuovo`. Si confronta il valore trovato con il limite: se il limite è stato raggiunto oppure superato, si termina, dato che abbiamo raggiunto il massimo valore che andava calcolato. Se invece il massimo non è stato raggiunto, stampiamo l'ultimo valore calcolato. A questo punto, le variabili `penultimo`, `ultimo` e `nuovo` contengono in effetti gli ultimi tre valori della serie che abbiamo trovato. Si noti che il penultimo valore trovato si trova in `ultimo`, mentre l'ultimo valore trovato si trova in `nuovo`. La variabile `ultimo` conteneva l'ultimo valore trovato, ma ora ne abbiamo trovato un altro, per cui la variabile ora contiene in effetti il penultimo valore.

Per far funzionare il tutto, occorre ora fare la chiamata ricorsiva per calcolare i successivi valori. Dal momento che il limite è lo stesso di quello che abbiamo ricevuto, bisogna solo capire cosa dobbiamo mettere come primi due argomenti della chiamata ricorsiva. Dato che la funzione riceve gli ultimi due valori trovati, e questi stanno nelle variabili `ultimo` e `nuovo`, la chiamata ricorsiva deve passare come parametri `ultimo`, `nuovo` e `limite`. La funzione che si ottiene è qui sotto.

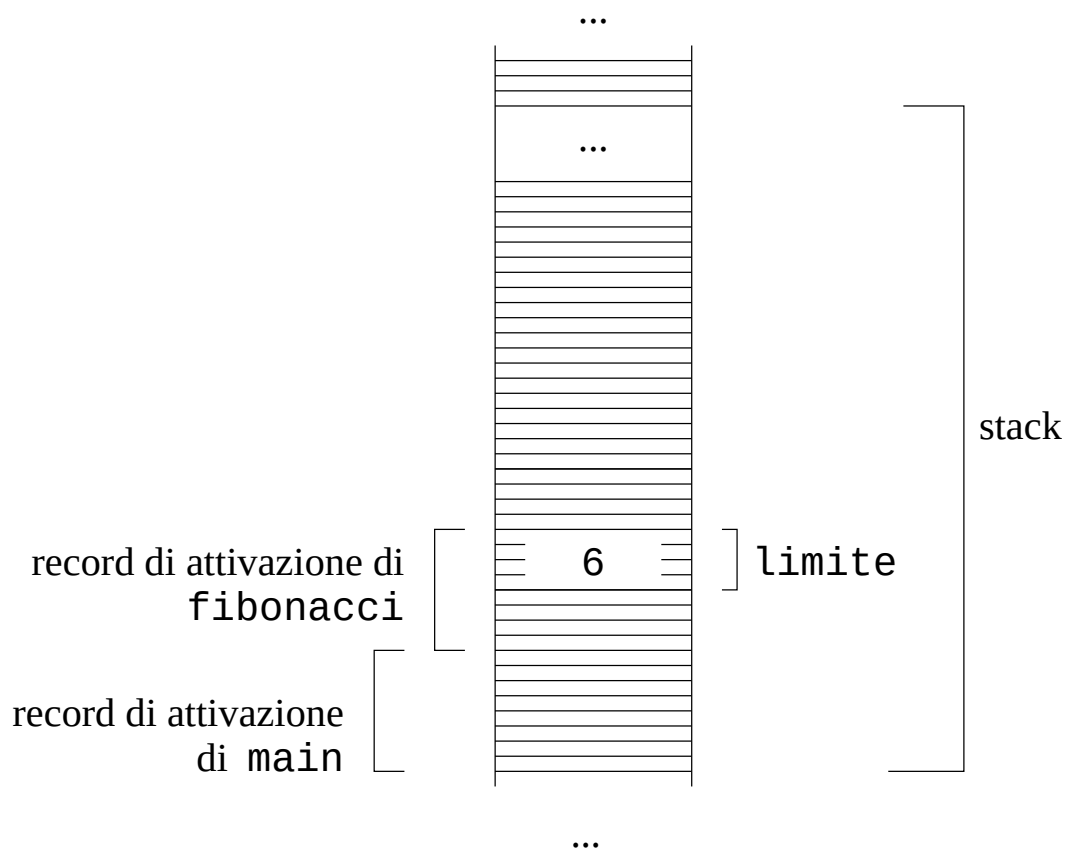
```
void fibonacciRicorsiva(int penultimo, int ultimo, int limite) {
  int nuovo;

  nuovo=penultimo+ultimo;

  if(nuovo<limite) {
    printf("%d ", nuovo);

    fibonacciRicorsiva(ultimo, nuovo, limite);
  }
}
```

Occorre ora capire in che modo la funzione va chiamata. Occorre dare come primi due parametri 1 e 1, dal momento che questi sono i primi due elementi della sequenza. Il terzo parametro è semplicemente il limite che viene dato. Volendo realizzare una funzione che stampa gli elementi della serie fino al limite, quello che occorre fare è stampare i primi due elementi della serie, e poi chiamare la funzione ricorsiva per stampare gli altri. Quindi, occorre realizzare una funzione come quella qui sotto.

```
void fibonacci(int limite) {
  printf("1 ");
  printf("1 ");

  fibonacciRicorsiva(1, 1, limite);
}
```

Il programma completo fiboric.c contiene un programma principale che riceve il limite come argomento, e fa una semplice chiamata alla funzione di sopra.

Vediamo ora la simulazione della esecuzione del programma quando si passa `6` come limite alla funzione `fibonacci`. Non ci interessa come è fatto il record di attivazione di `main`. Guardiamo solo i record di attivazione successivi. Dal momento che `main` chiama `fibonacci` passando il valore `6`, nel record di attivazione della funzione `fibonacci` c'è il valore `6` nella variabile (parametro formale) `limite`.



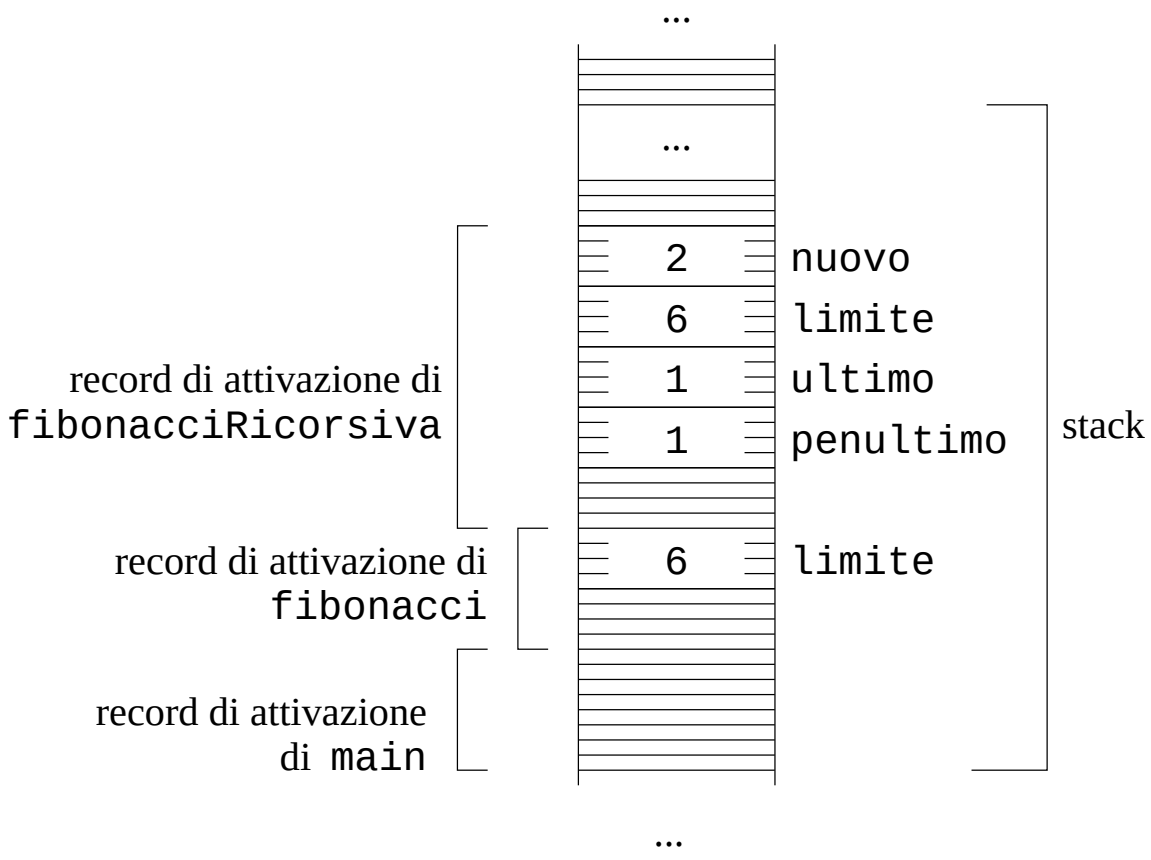
Viene ora chiamata la funzione `fibonacciRicorsiva`. A questa funzione si passano come argomenti i valori `1`, `1` e il valore di `limite`, ossia `6`. Questo significa che la chiamata `fibonacciRicorsiva(1, 1, limite);` è equivalente a:

```
fibonacciRicorsiva(1, 1, 6);
```

La intestazione della funzione è:

```
void fibonacciRicorsiva(int penultimo, int ultimo, int limite)
```

Questo siginifica che i valori dei parametri formali `penultimo`, `ultimo` e `limite` valgono `1`, `1` e `6`. Si esegue poi la somma, che si mette in `nuovo`. A questo punto, lo stack dei record di attivazione è il seguente.



Dato che `nuovo` è minore di `limite`, si esegue la stampa e la chiamata ricorsiva. La chiamata ricorsiva crea un nuovo record di attivazione, in cui si trovano i parametri formali. Per capire cosa ci va messo in questi parametri, occorre seguire questa regola:

1. si scrive la chiamata ricorsiva mettendo i valori effettivi degli argomenti. Dato che la chiamata ricorsiva è `fibonacciRicorsiva(ultimo, nuovo, limite)`, si vanno a vedere i valori delle variabili nel record corrente, ossia quello più in alto (nello stato attuale, lo stack è quello della figura sopra). Dato che `ultimo` vale `1`, mentre `nuovo` vale `2` e `limite` vale `6`, i tre valori nella chiamata ricorsiva sono:

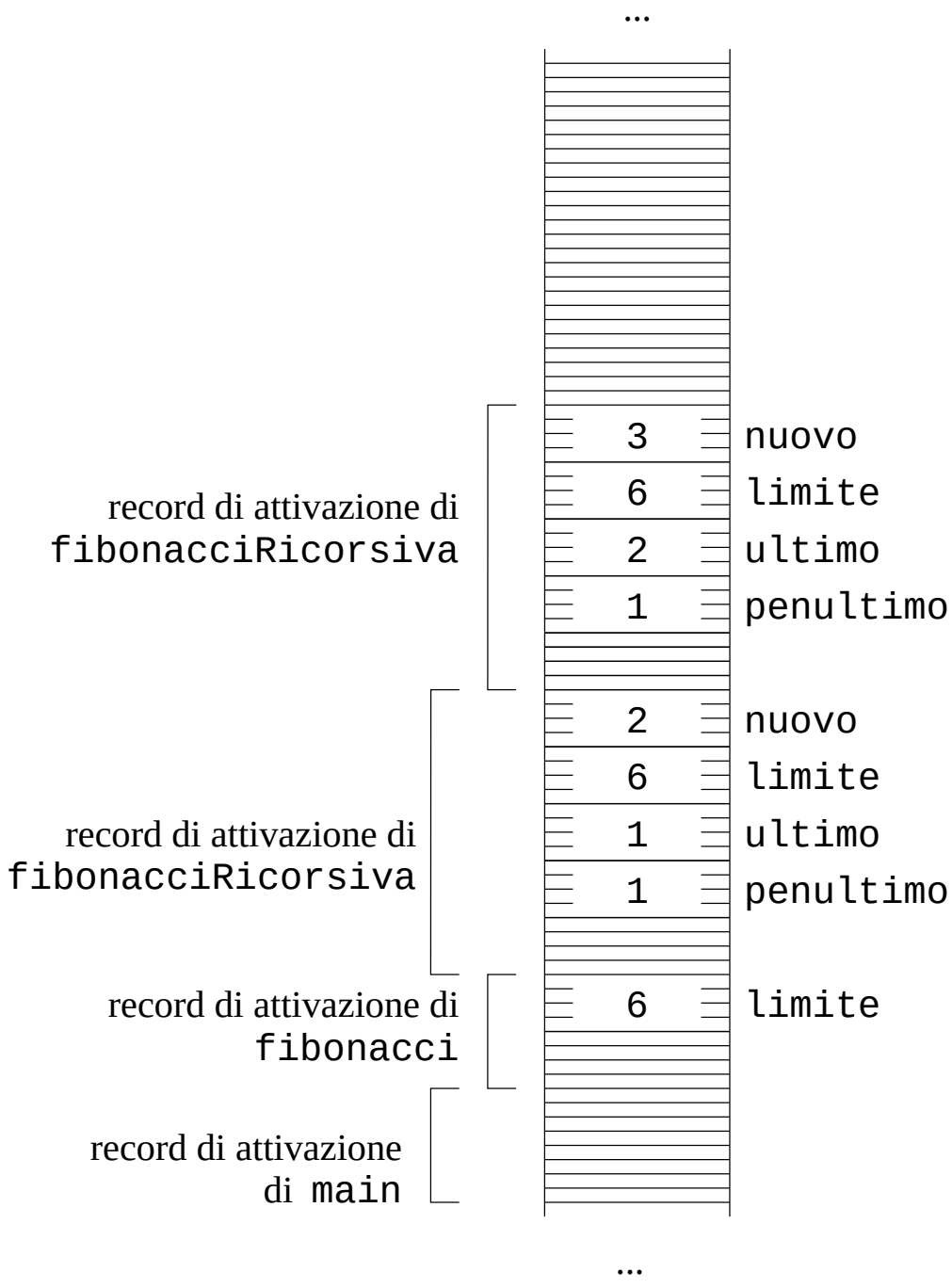
   ```
   fibonacciRicorsiva(1, 2, 6);
   ```

2. si scrive la intestazione della funzione:

   ```
   void fibonacciRicorsiva(int penultimo, int ultimo, int limite)
   ```

3. in ogni variabile della intestazione si mette il corrispondente valore che sta fra parentesi nella chiamata ricorsiva. Nel nostro caso, dal confronto si vede che in `penultimo` ci va `1`, in `ultimo` ci va `2`, mentre in `limite` ci va `6`.

Abbiamo quindi la situazione seguente (nella chiamata ricorsiva, si calcola il valore di `nuovo` come somma di `penultimo` e `ultimo`).



Si noti che la regola scritta sopra (valutare gli argomenti sulla base dei valori delle variabili nell'elemento dello stack affiorante) va *presa alla lettera*. In particolare, dalla regola si capisce che i valori delle variabili nella nuova chiamata ricorsiva non sono necessariamente uguali a quelli del record sottostante. In particolare, questo avviene solo se il valore di una variabile viene passato direttamente nella chiamata ricorsiva. Inoltre, le variabili locali che non sono i parametri formali della funzione, hanno un valore iniziale indefinito (non si può assumere che abbiano inizialmente gli stessi valori che si trovano nel record precedente).

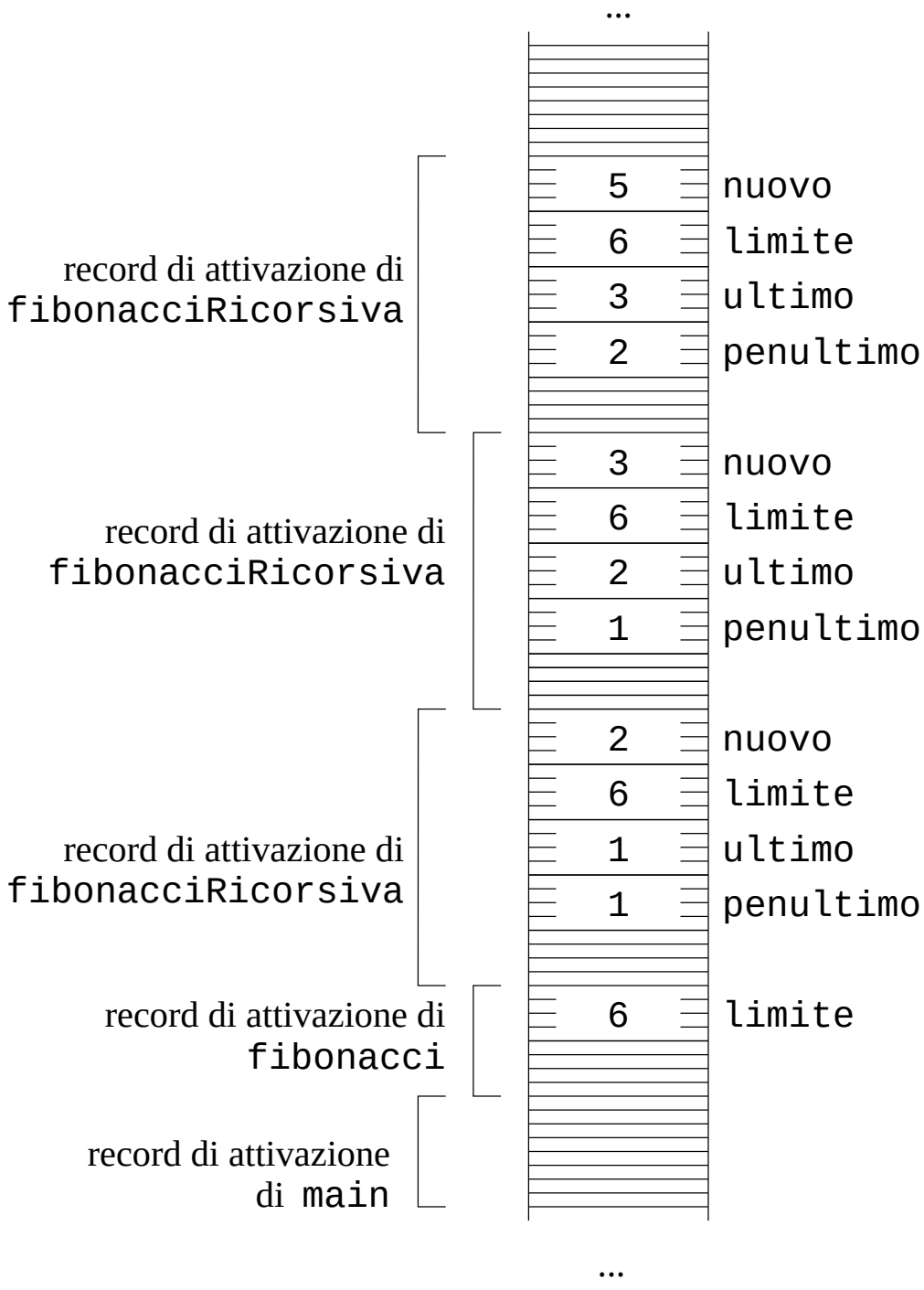
Dato che ora `nuovo` vale meno di `limite`, si stampa e si esegue nuovamente la chiamata ricorsiva. Qui seguiamo ancora la stessa regola: la chiamata è:

```
fibonacciRicorsiva(ultimo, penultimo, limite);
```

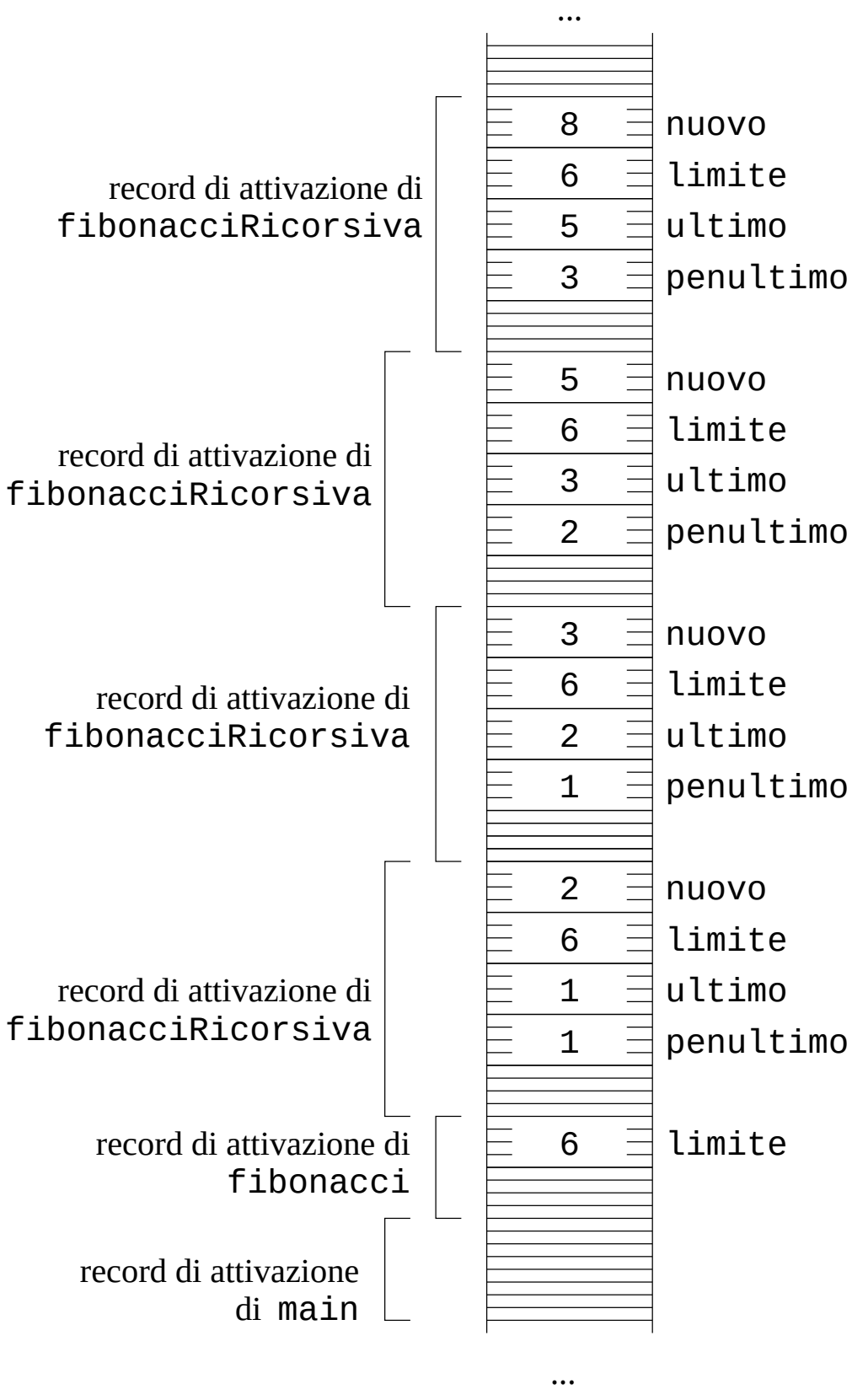
Si valutano le variabili, guardando l'ultimo record di attivazione.

```
fibonacciRicorsiva(2, 3, 6)
```

Dal confronto con la intestazione della funzione, si capiscono i valori dei parametri nel nuovo record di attivazione (che sono appunto 2, 3 e 6).



A questo punto, si ripete tutto di nuovo. Il valore calcolato per `nuovo` è ancora minore del valore di `limite`, per cui si stampa il numero, e si effettua la chiamata ricorsiva. Per i valori dei parametri formali nella nuova chiamata ricorsiva, si segue ancora la regola si sopra, per cui si vede che i valori iniziali di `penultimo`, `ultimo` e `limite` sono `3`, `5` e `6`.



Il calcolo del valore di `nuovo` produce `8`, che è superiore al valore di `limite`, per cui la funzione termina senza fare né la stampa né la chiamata ricorsiva. A questo punto si torna ad eseguire quello che mancava della funzione chiamante. In questo caso, la chiamata ricorsiva era l'ultima istruzione,

per cui tutte le funzioni terminano, e si ritorna direttamente al programma principale, deallocando tutti i record di attivazione.

La ricorsione su liste, stampa di una lista

La ricorsione su liste, stampa di una lista

La ricorsione risulta particolarmente utile sulle liste collegate. Questo è dovuto al fatto che le liste si possono definire in modo ricorsivo, per cui una funzione ricorsiva si può dimostrare facilmente essere corretta, basandosi appunto su questa definizione ricorsiva.

La definizione di lista usata fino a questo momento è: una lista è una sequenza di dati dello stesso tipo (per esempio, interi). Una definizione alternativa è la seguente:

> una lista è: la lista vuota, oppure un elemento seguito da un'altra lista.

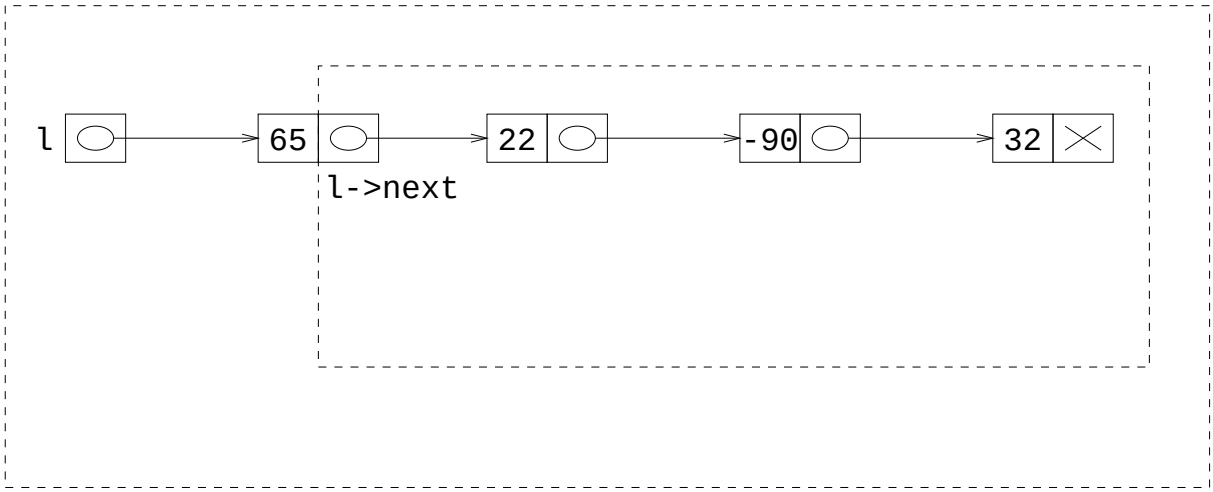
In altre parole, una sequenza di elementi può essere composta da zero elementi, oppure da un elemento seguito da altri elementi in sequenza. La definizione di tipo che si usa in C riflette esattamente questa definizione: infatti, una variabile di tipo lista `l` può valere `NULL` (che rappresenta la lista vuota), oppure può essere un puntatore a una struttura che contiene un dato più un altro puntatore. Si noti che il puntatore che sta nella struttura è di tipo `struct NodoLista *`, e che `TipoLista` è in effetti lo stesso tipo. Possiamo quindi dire che la struttura è composta da un elemento e da un puntatore, che rappresenta un'altra lista.

Per essere ancora più chiari, possiamo usare la seguente definizione di tipo: dato che `TipoLista` è un puntatore a una struttura `struct NodoLista`, possiamo dire che il campo `next` della struttura è di tipo `TipoLista`, e questo è equivalente alla definizione di tipo data in precedenza:

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

typedef struct NodoLista *TipoLista;
```

Con questa nuova definizione, la ricorsività appare abbastanza evidente: una variabile dichiarata come `TipoLista l` può valere `NULL`, e quindi rappresentare la lista vuota, oppure può essere un puntatore a una struttura che contiene un valore e un'altra variabile `l->next` che rappresenta un'altra lista.

Per essere più precisi, la variabile di tipo lista `l->next` rappresenta la lista che è composta da tutti gli elementi tranne il primo. Questo discende immediatamente dalla regola che dice quale è la lista rappresentata da una variabile: è la lista che si ottiene guardando la struttura puntata e poi seguendo i puntatori fino a `NULL`.



Dalla figura risulta abbastanza chiaro che la lista rappresentata da `l` è `(65 22 -90 32)`. Nello stesso modo, la lista rappresentata da `l->next` si ottiene seguendo le freccie, per cui il primo valore è `22` seguito da `-90` seguito da `32`, per cui la lista è `(22 -90 32)`, che è appunto la stessa lista rappresentata da `l` tranne il primo elemento. Questo fatto si può capire anche passando `l->next` alla funzione di stampa di una lista: è evidente che verrà stampata tutta la lista, tranne il primo elemento.

Dalla figura risulta anche chiaro che l'unica cosa che realmente distingue le variabili `l` e `l->next` è solo il fatto che la seconda è un campo di una struttura. Come si è però già detto, i campi delle strutture si possono considerare come variabili, e quindi usare in qualsiasi contesto in cui si possono usare variabili normali dello stesso tipo. La lista rappresentata da tutti gli elementi di quella originaria tranne il primo si dice *resto della lista*. Quindi, per esempio, `l->next` è il resto della lista `l`.

Le funzioni ricorsive su liste hanno in genere questa caratteristica: sono funzioni che hanno come argomento una lista `l`, e al loro interno c'è una chiamata ricorsiva a cui si passa `l->next`. Queste funzioni normalmente operano su `l->val` (il primo elemento della lista), e poi agiscono sul resto della lista solo attraverso la chiamata ricorsiva, a cui viene passato appunto il resto della lista, che è `l->next`.

La regola per scrivere una corretta funzione ricorsiva è la seguente: si assume che la chiamata ricorsiva al suo interno funzioni (ossia che la funzione operi già correttamente sul resto della lista), e si scrive una funzione che opera correttamente su tutta la lista (incluso il primo elemento). Occorre poi che ci siano casi in cui la funzione non chiami ricorsivamente se stessa. Questo viene generalmente garantito dal fatto che la funzione deve essere corretta anche nel caso in cui la lista è vuota (per cui non esiste un resto della lista su cui fare la chiamata ricorsiva).

Consideriamo quindi il problema di stampare una lista. Una funzione di questo genere è abbastanza banale da realizzare in modo iterativo. Vediamo qui la versione ricorsiva. Per prima cosa, una lista o è la lista vuota, oppure è composta da un elemento seguito da un'altra lista. Sia `void StampaLista(TipoLista l)` la intestazione della funzione, per cui `l` è la lista passata. Se `l` rappresenta la lista vuota, non si stampa niente; si esce semplicemente dalla funzione senza fare nulla. Nel caso in cui la lista non è vuota, allora è composta da un elemento `l->val` e dal resto della lista. Il principio è che possiamo assumere che la chiamata ricorsiva sul resto della lista funziona, per cui mettere dentro la funzione la istruzione `StampaLista(l->next)` stampa effettivamente il resto della lista, ossia tutti gli elementi tranne il primo.

Abbiamo quindi coperto il caso della lista vuota, e sappiamo che mettendo la istruzione `StampaLista(l->next)` riusciamo a stampare gli elementi della lista a partire dal secondo, fino alla fine. Dato che vogliamo stampare tutti gli elementi della lista (a partire dal primo), possiamo stampare il primo elemento e poi mettere la istruzione `StampaLista(l->next)`. In altre parole, dato che quest'ultima istruzione stampa gli elementi della lista a partire dal secondo, e che `printf("%d ", l->val)` stampa il primo, per fare la stampa di tutti basta mettere prima la `printf` che stampa il primo, e poi la chiamata ricorsiva che stampa tutti i successivi. La funzione complessiva è quindi la seguente:

```
void StampaLista(TipoLista l) {
  if(l==NULL)
    return;

  printf("%d ", l->val);
  StampaLista(l->next);
}
```

La parte fondamentale di una funzione ricorsiva è chiaramente il caso in cui viene fatta un'altra chiamata alla funzione stessa. In questo caso, si assume che la funzione sia corretta quando viene chiamata sul resto della lista. Non va però dimenticato che la funzione, per essere corretta, deve avere un caso base, ossia *si deve sempre, prima o poi, arrivare a una situazione in cui non viene fatta la chiamata ricorsiva*. Questo caso viene a volte coperto dal caso in cui la lista è vuota, per cui se la funzione va bene su lista vuota e su lista di più elementi, allora è corretta. Ci sono però dei casi in cui occorre trattare il caso di lista composta da un solo elemento in modo particolare. Questo verrà visto in dettaglio più avanti.

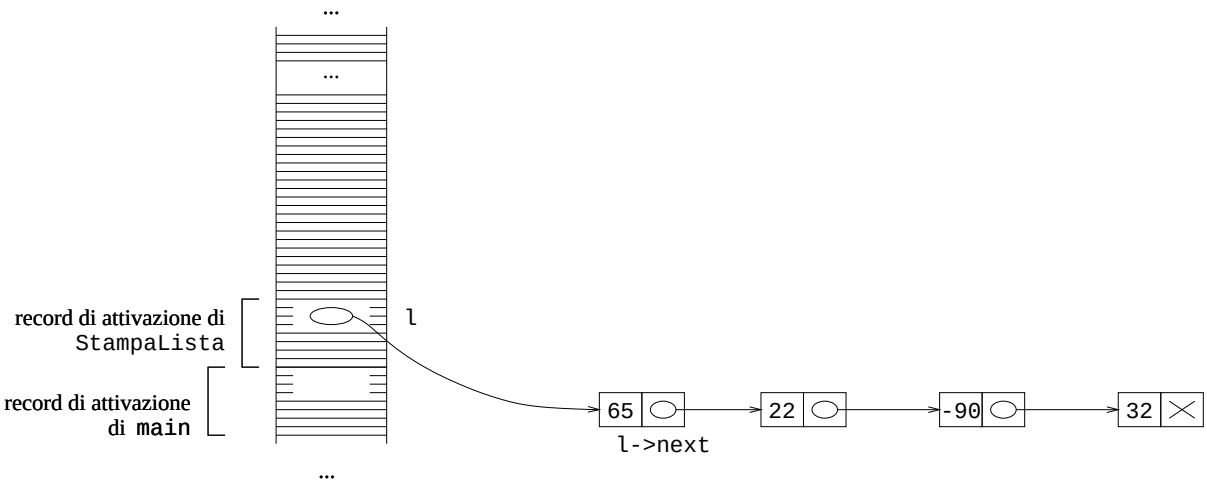
Possiamo quindi riassumere il metodo di scrittura di funzioni ricorsive su liste come segue:

1. si assume che la chiamata ricorsiva su `l->next` funzioni correttamente;
2. si scrive la funzione usando questa assunzione, e tenendo conto che la lista può essere vuota;
3. si verificano i casi limite, quali appunto la lista vuota, la lista composta di un solo elemento, e i casi che possono apparire rilevanti per il problema da risolvere (per esempio, se si devono rimuovere elementi in mezzo un caso da controllare è quello di elementi consecutivi da eliminare).

Per concludere, notiamo che questo metodo di scrittura delle funzioni ricorsive è analogo al meccanismo di dimostrazione per induzione in matematica. Quello che in matematica è il caso base, per noi è la lista vuota. Nelle dimostrazioni per induzione, si assume che la tesi sia vera per un certo numero `n-1`, e la si dimostra vera per `n`. Nel nostro caso, scriviamo la funzione in modo che sia corretta per una lista di lunghezza generica `n` assumendo che la chiamata ricorsiva funzioni su una lista di lunghezza `n-1` (il resto della lista).

Il programma completo che contiene la stampa della lista con funzione ricorsiva è prntric.c

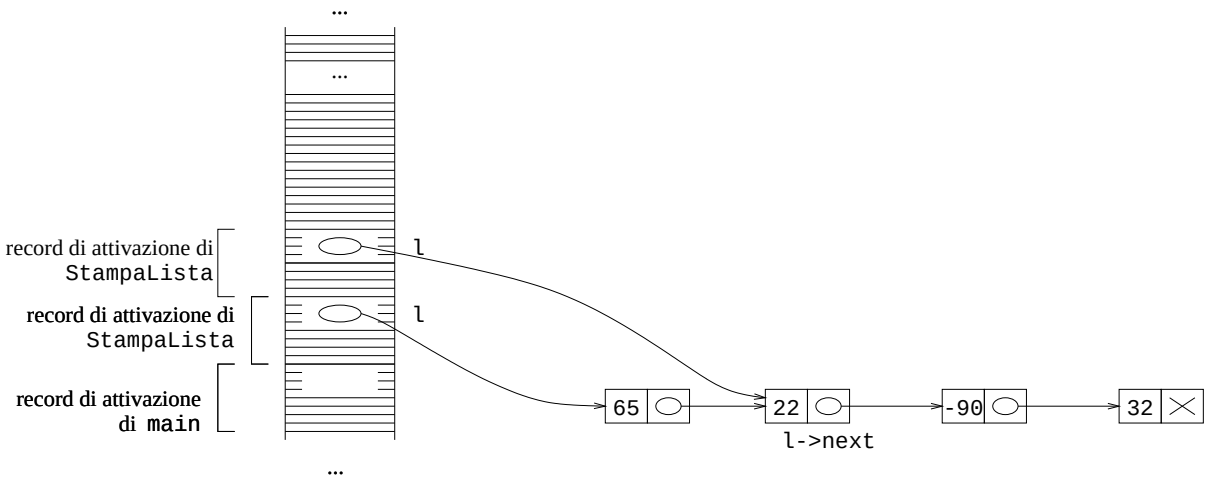
Vediamo ora l'esecuzione della funzione sulla lista di sopra, ossia `(65 22 -90 32)`. Alla prima chiamata della funzione, abbiamo la seguente situazione.



Disegnamo la lista con la notazione grafica delle liste. Per i record di attivazione questa rappresentazione non è adatta, dato che comunque le variabili devono essere organizzate in gruppi (i record di attivazione) che vanno messi l'uno sopra l'altro.

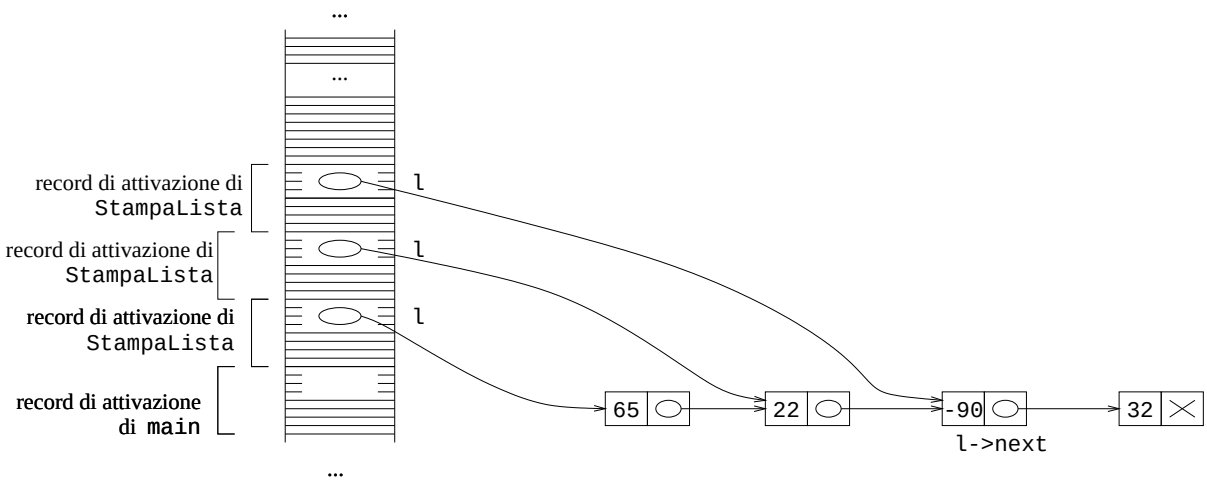
La esecuzione della funzione provoca la stampa del primo elemento della lista, `l->val`.

Quando si effettua la prima chiamata ricorsiva, `StampaLista(l->next)`, si calcola il valore di `l->next`. Nel nostro caso, `l->next` è l'indirizzo della seconda struttura della lista. Quindi, il valore che viene passato alla funzione è l'indirizzo della seconda struttura della lista. Mettiamo quindi un nuovo record di attivazione, e nel parametro formale ci va l'indirizzo della seconda struttura della lista. Questo equivale a mettere una freccia dal nuovo `l` alla seconda struttura.

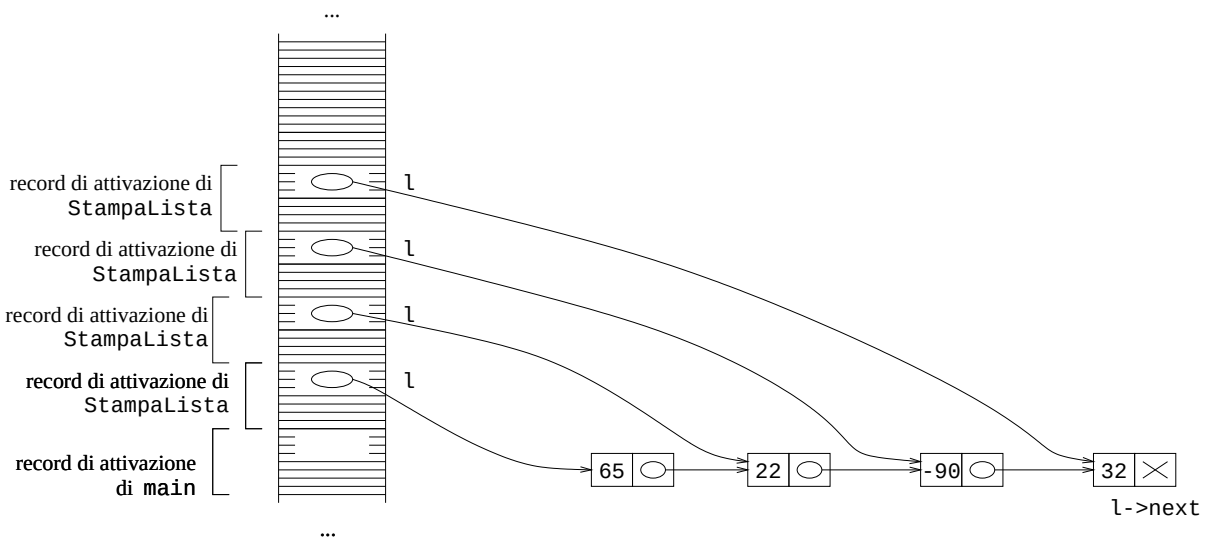


Si fa la stampa di `l->val`. In questo caso, `l` ha il valore che si trova nel record più in alto, quindi è l'indirizzo della seconda struttura della lista. Quindi `l->val` è il secondo elemento della lista. Finora abbiamo quindi stampato il primo e il secondo elemento della lista.

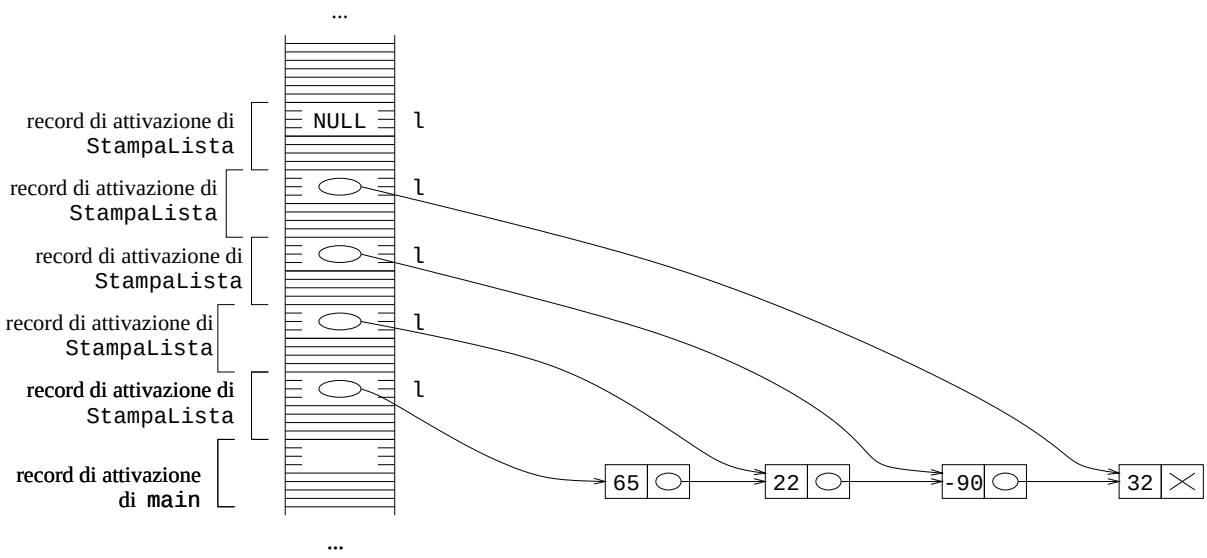
Si fa ora la chiamata ricorsiva. Dato che `l` contiene l'indirizzo della seconda struttura della lista, `l->next` è l'indirizzo della terza. Quindi, il valore che viene passato nella chiamata ricorsiva è l'indirizzo della terza struttura. Questo valore viene messo nel parametro formale `l` del nuovo record di attivazione. Quindi, nel nuovo record di attivazione c'è una variabile `l` che contiene l'indirizzo della terza struttura della lista, e questo equivale a mettere una freccia.



Si stampa di nuovo `l->val`, che è il terzo elemento della lista, e si fa di nuovo la chiamata ricorsiva. Viene passato `l->next`, che è l'indirizzo della quarta struttura, e quindi nel nuovo record di attivazione c'è una variabile `l` che contiene l'indirizzo della quarta struttura, cosa che si rappresenta con la freccia.



Si stampa ancora `l->val`, che ora è il quarto elemento della lista. Si noti che, a questo punto, abbiamo effettivamente stampato tutti gli elementi della lista, in ordine. La chiamata ricorsiva ora viene fatta con `l->next`, che contiene `NULL`. Si passa quindi `NULL`, che viene messo nella varibile `l` del nuovo record di attivazione.



Dato che `l` vale `NULL`, la funzione termina senza fare altre stampe e chiamate ricorsive. Dato che le chiamate ricorsive sono state fatte sempre alla fine della esecuzione di ogni funzione, tutte le chiamate ricorsive terminano, con conseguente deallocazione di tutti i record di attivazione della funzione `StampaLista`.

Si noti che sono stati stampati tutti gli elementi della lista, in ordine. Si vede anche facilmente che la funzione è corretta anche nel caso in cui la lista di partenza è vuota (non si stampa niente). Anche altri casi particolari, come la lista composta da un solo elemento, vengono stampati correttamente.

Stampa di una lista al contrario

Stampa di una lista al contrario

Consideriamo il problema di stampare una lista all'indietro, ossia partendo dall'ultimo elemento per arrivare a stampare il primo.

Questo problema si risolve banalmente in modo ricorsivo. Sappiamo che una lista o è vuota, oppure è composta da un elemento e da un resto (che è la lista composta dai restanti elementi). Stampare al contrario la lista vuota significa non stampare niente. Per l'altro caso (la lista ha un elemento e il resto), uso l'assunzione che la chiamata ricorsiva sul resto della lista stampi effettivamente il resto della lista al contrario.

Sia quindi `void StampaListaRovesciata(TipoLista l)` la intestazione della funzione. Sappiamo che la istruzione `StampaListaRovesciata(l->next)` stampa in ordine inverso il resto della lista. Dato che il resto della lista è composto da tutti gli elementi dal secondo fino all'ultimo, questa istruzione stampa prima l'ultimo elemento, poi il penultimo, ecc. fino al secondo. Dobbiamo ora fare in modo che venga stampata al contrario *tutta* la lista, incluso il primo elemento.

A questo punto, dovrebbe essere abbastanza evidente che, se `StampaListaRovesciata(l->next)` stampa dall'ultimo elemento al secondo, per stampare dal secondo al primo basta eseguire questa istruzione e poi una stampa del primo elemento con `printf("%d ", l->val)`.

La funzione complessiva è quindi la seguente: se la lista è vuota, non si fa nulla. Se ci sono elementi, si stampa prima il resto della lista al contrario (con la chiamata ricorsiva) e poi il primo elemento. È facile verificare come i casi come quello di lista composta da un solo elemento vengono trattati correttamente. Il programma che contiene questa funzione, riportata qui sotto, è prntric.c

```
void StampaListaRovesciata(TipoLista l) {
  if(l==NULL)
    return;

  StampaListaRovesciata(l->next);
  printf("%d ", l->val);
}
```

Per simulare il comportamento della lista occorre tenere presenti i punti di ritorno. I record di attivazione hanno lo stesso contenuto di quelli del programma della pagina precedente (ricorsione su liste). L'unica differenza è che la stampa avviene solo dopo che la chiamata ricorsiva è terminata. In altre parole, prima si costruiscono tutti i record di attivazione. Quando ogni chiamata ricorsiva termina, la precedente fa la stampa e poi termina anch'essa. Questo produce effettivamente la stampa degli elementi della lista al contrario.

Lettura di una lista da file, in ordine, con ricorsione

Lettura di una lista da file, in ordine, con ricorsione

Questa è la prima funzione ricorsiva che crea/modifica una lista. Si tratta di una funzione che legge gli elementi che sono scritti su un file, e restituisce la lista composta da questi elementi nello stesso ordine in cui sono scritti sul file.

Usiamo ancora la definizione ricorsiva di lista (una lista può essere vuota, oppure composta da un elemento e dal resto). Soltanto che, questa volta, la lista va costruita, piuttosto che usata.

Per prima cosa, in che modo possiamo arrivare alla situazione in cui la lista da costruire è vuota? Se tentiamo di leggere un elemento da file, e la cosa non riesce, possiamo dire che non ci sono altri elementi da leggere, per cui la lista da restituire è vuota. Quindi, facciamo una `fscanf`, e se questa non ritorna 1, allora diciamo che non ci sono altri elementi da leggere su file, per cui ritorniamo la lista vuota.

D'altra parte, se la lettura riesce, allora la lista che va restituita non è vuota. In particolare, il primo elemento della lista è quello che abbiamo letto da file. Bisogna ora leggere il resto della lista. Qui sfruttiamo l'ipotesi ricorsiva, ossia che chiamando la funzione stessa, questa restituisce la lista dei rimanenti elementi che stanno su file, ossia assumiamo che la chiamata ricorsiva legga correttamente il resto della lista.

Consideriamo quindi una funzione che riceve un descrittore di un file già aperto, e che restituisce la lista degli elementi letti in ordine. L'intestazione di questa funzione è quindi: `TipoLista LeggiFileRic(FILE *fd)`. In questa funzione, tentiamo di leggere un elemento da file. Se la cosa non riesce, diciamo che la lista è vuota, per cui si ritorna `NULL`. Se la lettura ha successo, l'elemento letto deve essere il primo della lista, mentre il resto della lista può venire creato con una chiamata ricorsiva. Quindi, si fa la allocazione di una struttura; nel campo `val` ci mettiamo il valore letto da file, mentre nel campo `next` ci mettiamo la lista creata dalla chiamata ricorsiva. La funzione è quindi la seguente.

```
TipoLista LeggiFileRic(FILE *fd) {
  TipoLista l;
  int x;
  int res;

  res=fscanf(fd, "%d", &x);
  if(res!=1)
    return NULL;
  else {
    l=malloc(sizeof(struct NodoLista));
    l->val=x;
    l->next=LeggiFileRic(fd);
    return l;
  }
}
```

Questa funzione richiede come parametro il descrittore di un file già aperto. Se vogliamo una funzione che prende come argomento il nome di un file (ossia una stringa) e restituisce la lista letta da file, dobbiamo aprire il file e passare il descrittore alla funzione di sopra. La lista da restituire è semplicemente quella ritornata dalla funzione di sopra. La funzione che fa questo è qui sotto.

```
TipoLista LeggiFile(char *nomefile) {
  FILE *fd;

  fd=fopen(nomefile, "r");
  if(fd==NULL) {
    perror("Errore in apertura del file");
    exit(1);
  }

  return LeggiFileRic(fd);
}
```

Capita abbastanza spesso, nella pratica, che si riesca a scrivere una funzione ricorsiva abbastanza facilmente, ma che la funzione da realizzare richieda dei passi preliminari. Un altro caso già visto è quello della funzione di stampa della serie di Fibonacci, che richiedeva di stampare i primi due valori della serie prima della chiamata ricorsiva. Se alcune operazioni vanno fatte solo all'inizio, è bene creare una funzione ausiliaria che effettua queste operazioni e poi chiama la funzione ricorsiva. Questo vale anche nel caso in cui la funzione ricorsiva ha dei parametri, che la funzione che ci occorre non deve avere: questo è per esempio il caso della funzione che stampa la serie di Fibonacci, in cui la funzione ricorsiva richiede tre parametri, mentre vogliamo poter chiamare una funzione di stampa che abbia come unico parametro il limite.

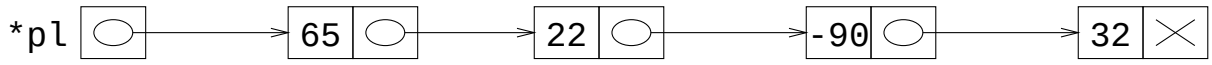
Cancellazione ultimo elemento con funzione ricorsiva

Cancellazione ultimo elemento con funzione ricorsiva

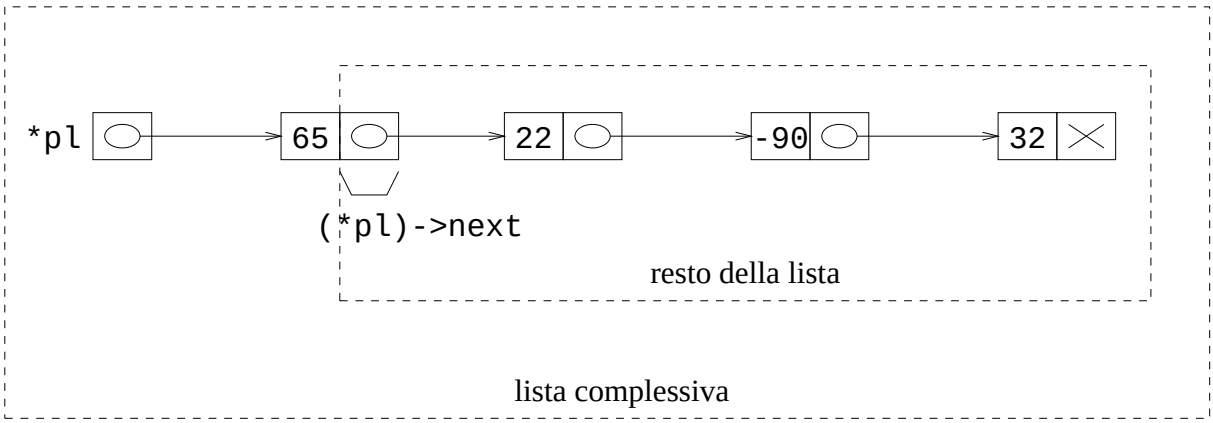
Risolviamo il problema della cancellazione dell'ultimo elemento di una lista usando la ricorsione. Come si vedrà, la funzione che si ottiene è molto più semplice di quella iterativa.

Consideriamo per prima cosa la definizione ricorsiva di lista. Una lista può essere o vuota oppure composta da un elemento e dal resto della lista. Nel caso di lista vuota, si può considerare la cancellazione dell'ultimo elemento come un errore, per cui si stampa un messaggio di errore e si termina il programma. Potremmo anche assumere che cancellare l'ultimo elemento da una lista vuota non abbia effetti: questo dipende da come il problema viene interpretato.

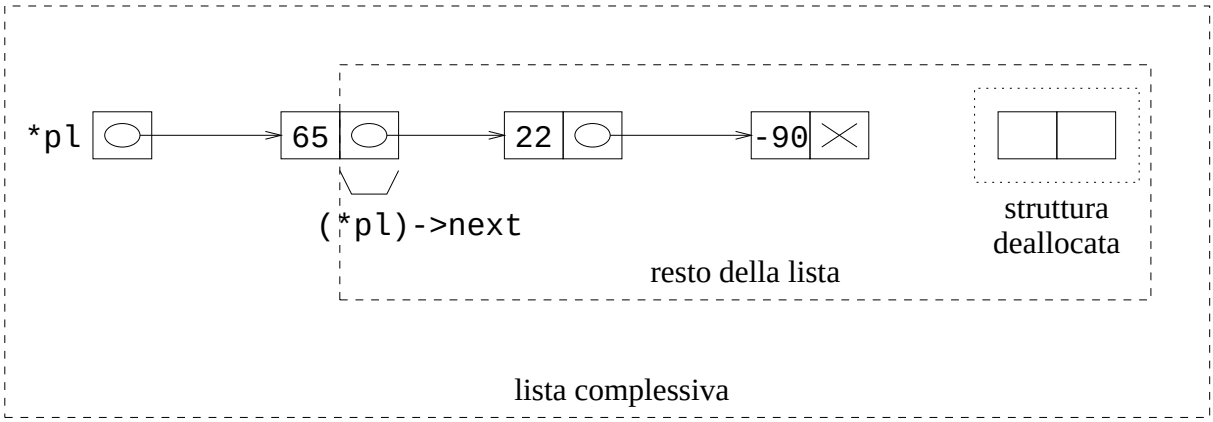
Vediamo ora il passo ricorsivo. Per prima cosa, scriviamo la intestazione della funzione. La lista va passata per indirizzo, dato che, nel caso di lista di un solo elemento, il puntatore iniziale va alterato. Quindi, abbiamo la seguente intestazione: `void EliminaUltimoRicorsiva(TipoLista *pl)`. La ipotesi ricorsiva, in questo caso, è che la funzione realizza la cancellazione dell'ultimo elemento, quando viene passato il resto della lista. In altre parole, si assume che la chiamata ricorsiva sul resto della lista `EliminaUltimoRicorsiva(& (*pl)->next )` effettivamente cancelli l'ultimo elemento dal resto della lista. Dalla rappresentazione grafica si capisce che eliminare l'ultimo elemento dal resto della lista, effettivamente lo elimina anche dalla lista complessiva. Supponiamo quindi che la lista passata sia la seguente:



Disegnamo chiaramente la lista complessiva e il resto della lista.



L'ipotesi ricorsiva dice che la chiamata di funzione, passando il resto della lista, elimina l'ultimo elemento dal resto della lista, ossia si arriva alla seguente situazione.



la lista e il resto, dopo l'eliminazione dell'ultimo elemento del resto

Per definizione, il resto della lista, dopo la cancellazione dell'ultimo elemento, non è più collegata all'ultima struttura, che è stata deallocata. Come si vede chiaramente dalla figura, anche la lista complessiva non contiene più l'ultimo elemento. La funzione ha quindi realizzato la eliminazione dell'ultimo elemento dalla lista.

Occorre ora verificare l'esistenza di un caso base, ossia di un caso in cui non viene effettuata la chiamata ricorsiva. Nel nostro caso, partendo da una lista generica, si fa sempre la chiamata ricorsiva sulla lista che è il resto. Questo termina quando si arriva all'ultima struttura, il cui resto è `NULL`, ossia la lista vuota. A questo punto, si genera un errore. È quindi chiaro che il passo base non funziona. Anche usando la politica per la quale la cancellazione dalla lista vuota è la lista vuota, quello che si ottiene è che l'ultimo elemento non viene mai cancellato.

In questo caso, la lista vuota genera un situazione particolare, e non può quindi essere il vero passo base della ricorsione. Se consideriamo i casi particolari, ci rendiamo conto che il problema è la lista fatta di un solo elemento. In questo caso, la lista che si ottiene cancellando l'ultimo elemento *non si ottiene cancellando l'ultimo elemento dal resto*. La lista fatta di un solo elemento è costituita dall'elemento stesso, e il resto della lista è la lista vuota. Il risultato della cancellazione, in questo caso, non è la lista ottenuta cancellando l'ultimo elemento dal resto. È invece la lista vuota. Quindi, l'errore sta nel fatto che la chiamata ricorsiva non va fatta, se la lista contiene un solo elemento. Al contrario, in questo caso si deve deallocare l'unico elemento della lista, che deve diventare la lista vuota.

La funzione finale è la seguente. Nel caso in cui la lista è vuota, si può generare un messaggio di errore e terminare. Se la lista ha un solo elemento, allora questo viene cancellato. In tutti gli altri casi (lista fatta di almeno due elementi), si cancella l'ultimo elemento dal resto della lista. Dato che la funzione prende una lista per indirizzo, e il resto della lista è `(*pl)->next`, la funzione va chiamata passando `& (*pl)->next`.

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

                        /* lista con un solo elemento */
  if( (*pl)->next==NULL ) {
    free(*pl);
    *pl=NULL;
    return;
  }

                        /* lista con piu' elementi */
  EliminaUltimoRicorsiva( & (*pl)->next );
}
```

Il programma complessivo che contiene questa funzione è delultri.c

Inserimento in lista ordinata con funzione ricorsiva

Inserimento in lista ordinata con funzione ricorsiva

Consideriamo di nuovo il problema di inserire un elemento in una lista ordinata, in modo tale che la lista rimanga ordinata. Anche in questo caso, usiamo una funzione ricorsiva, che risulterà essere molto più semplice della funzione iterativa vista in precedenza.

Usiamo ancora la definizione ricorsiva di lista. Se la lista è vuota, allora l'elemento va inserito nella lista, che dopo conterrà quindi solo l'elemento da inserire.

Consideriamo ora il caso di lista non vuota. Sia quindi la lista costituita da un primo elemento `x` e dal resto della lista. Sono possibili due casi. Se `x` è maggiore dell'elemento da inserire, allora questa va inserito in testa alla lista. Se l'elemento da inserire è invece maggiore di `x`, allora l'elemento non va inserito prima di `x`. Quindi va inserito dopo, ossia va inserito nel resto della lista. Anche qui, facciamo l'ipotesi che la chiamata ricorsiva inserisce correttamente l'elemento nel resto della lista.

In questo caso, è fondamentale il fatto che l'inserimento in lista ordinata funziona anche nel caso in cui l'elemento va inserito in prima posizione. In questo caso, se l'elemento va messo prima di `x`, allora lo si mette in prima posizione. Nel caso in cui va inserito dopo `x`, allora va messo o in seconda posizione oppure in una posizione successiva. Questo non fa nessuna differenza, dato che si tratta di inserire l'elemento, eventualmente in prima posizione, nel resto della lista.

La funzione complessiva, contenuta nel programma insordri.c, è la seguente.

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

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

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

                        /* l'elemento va aggiunto al resto della lista */
  InserimentoListaOrdinata(&(*pl)->next, e);
}
```

Il caso di lista vuota, evidentemente, questa volta non è un errore: al contrario, si deve semplicemente inserire l'elemento come unico elemento della lista. Se la lista contiene degli elementi, si confronta l'elemento da inserire con il primo della lista. Se è minore, allora va inserito in prima posizione (dopo aver fatto questo, si può terminare la esecuzione senza fare altre chiamate ricorsive). Se l'elemento da inserire è invece maggiore del primo della lista, allora va inserito nel resto della lista, eventualmente in prima posizione. Questo viene realizzato (anche nel caso di inserimento in testa nel resto) facendo la chiamata ricorsiva.

Non è difficile verificare che la funzione è corretta anche nel caso in cui l'elemento va inserito in ultima posizione. Suppondendo che la lista sia `(3 9 12 90)` e che l'elemento da inserire sia `100`, quello che succede è che la prima chiamata ricorsiva confronta `100` con `3`, e quindi chiama la funzione sul resto della lista e su `100`. Il resto della lista è `(9 12 90)`, per cui si confronta `9` con `100`. Quindi si fa la chiamata ricorsiva con `(12 90)`, e quindi anche su `(90)`. Dato che `90` è minore di `100`, si deve fare una ulteriore chiamata ricorsiva. Ora, la lista passata contiene solo `(90)`, per cui è la lista composta dal numero `90`, e il resto della lista è vuoto. L'ulteriore chiamata ricorsiva riceve la lista vuota, per cui l'elemento viene aggiunto come unico elemento. Questo significa che la lista che prima era composta solo da `90` ora è diventata `(90 100)`, dato che il resto della lista (che prima era

vuoto) ora contiene `100`. È quindi chiaro che anche tutti gli altri resti di lista sono stati alterati in questo modo, per cui `100` è stato aggiunto in coda alla lista originale.

Il metodo del puntatore a puntatore

Il metodo del puntatore a puntatore (non in programma)

Nell'ultima pagina si è vista una cosa che in apparenza contrasta con quello che è stato detto nel caso iterativo: infatti, passiamo la lista vuota a una funzione, e questo modifica la lista di cui quella passata era il resto. In altre parole, sappiamo che inserire in coda nel modo seguente non funziona:

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

  r=*pl;

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

  InserisciTestaLista(&r, e);
}
```

Questo non funziona perchè la aggiunta in testa a `r` crea una nuova struttura il cui indirizzo viene messo in `r`, ma non viene messo nel campo `next` dell'ultima struttura della lista. La funzione non è corretta perchè porta troppo avanti il puntatore, per cui non abbiamo più la possibilità di modificare il campo `next` dell'ultima struttura.

D'altra parte, la funzione di sopra non è la versione iterativa della funzione ricorsiva di inserimento in lista. Infatti, la chiamata ricorsiva ha come argomento non il *valore* del campo `next` della prima struttura, ma il suo *indirizzo*. Infatti, la chiamata avviene passando come parametro `&(*pl)->next`. Questo è esattamente l'indirizzo del campo `next` della struttura. Questo significa che, quando viene fatta l'ultima chiamata ricorsiva, ossia quando il campo `next` vale `NULL`, non viene passato il valore `NULL`, ma viene passato l'indirizzo della variabile che contiene il valore `NULL`. Quando si inserisce in lista, abbiamo l'indirizzo del campo `next` dell'ultima struttura, e lo usiamo per mettere in questo campo l'indirizzo della nuova struttura che è stata creata.

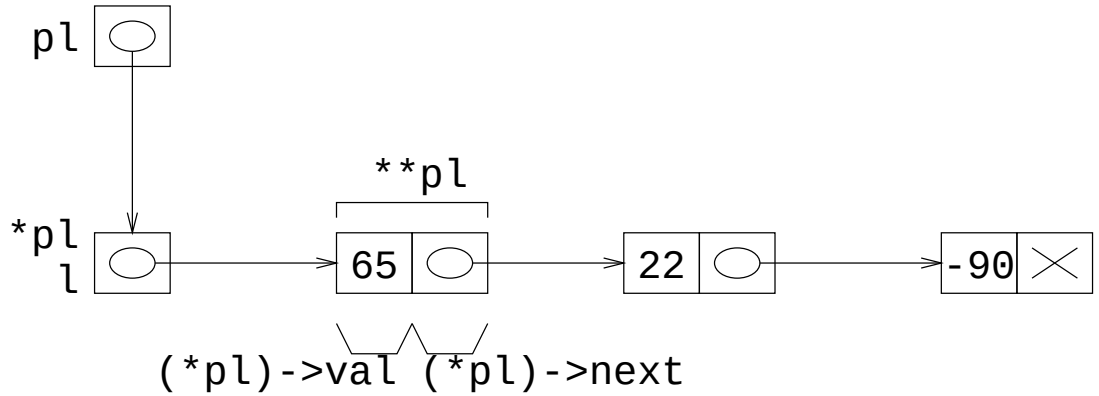
Vediamo ora come si può effettuare l'inserimento in coda a una lista usando in modo iterativo il metodo di usare un puntatore al campo `next`. Il punto fondamentale è questo: la funzione riceve come parametro un puntatore a una variabile di tipo lista. Per evitare di avere un comportamento differente a seconda se l'elemento va inserito come primo o come successivo, facciamo la scansione, però facciamo sempre in modo tale che la variabile di scansione non sia di tipo lista, ma sia dello stesso tipo del parametro, ossia sia un puntatore a una variabile di tipo lista.

All'inizio la variabile `pl` contiene l'indirizzo della variabile di tipo lista. Portare avanti il puntatore significa metterci dentro l'indirizzo del resto della lista. In altre parole, la successiva variabile di tipo lista è `(*pl)->next`. In `pl` ci mettiamo non il valore di questa variabile, ma il suo indirizzo. Siamo all'ultima struttura quando `*pl` vale `NULL`, per cui usiamo questa condizione per uscire dal ciclo di scansione. Quando siamo fuori dal ciclo, possiamo aggiungere la nuova struttura, mettendo il suo indirizzo in `*pl`. Si noti che `pl` contiene l'indirizzo del campo `next` dell'ultima struttura della lista, per cui mettere un indirizzo in `*pl` equivale a metterlo nel campo `next` dell'ultima struttura.

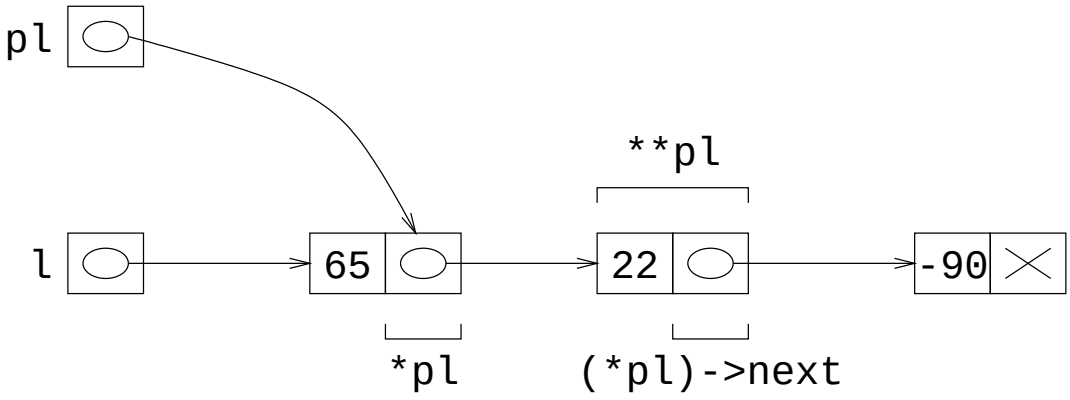
La funzione, che insieme ad altre dello stesso genere è contenuta nel file puntpunt.c, è la seguente.

```
void AggiungiCodaLista(TipoLista *pl, int e) {

                /* scansione */
  while(*pl!=NULL)
    pl=&(*pl)->next;

                /* aggiunta */
  *pl=malloc(sizeof(struct NodoLista));
  (*pl)->val=e;
  (*pl)->next=NULL;
}
```

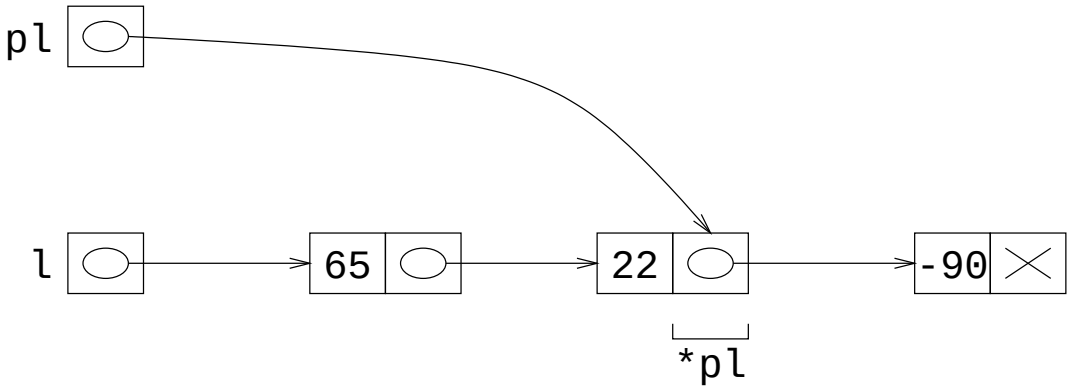
Supponiamo di chiamare la funzione passando come parametri la lista `(65 22 -90)` e l'intero da inserire `3`. La situazione iniziale è la seguente (dato che si passa l'indirizzo di una lista, `*pl` e `l` sono la stessa cosa).



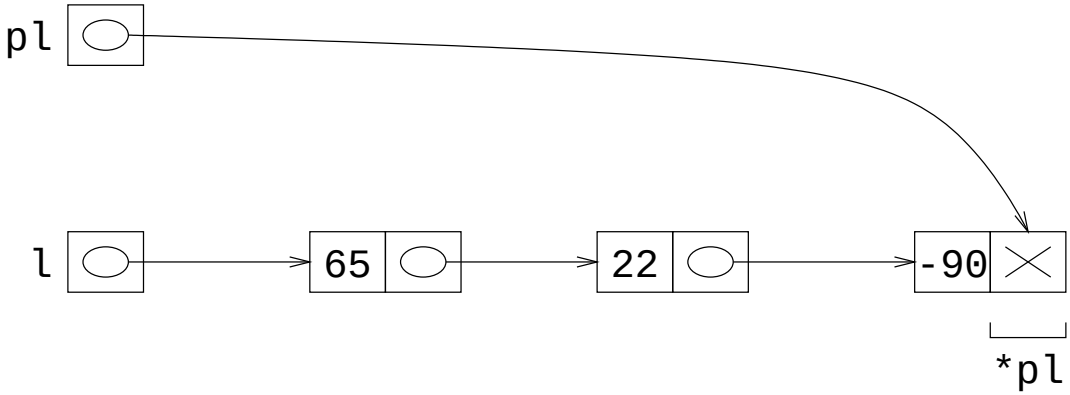
Dato che `*pl` non vale `NULL` si porta avanti il puntatore. In questo caso, `pl` non contiene l'indirizzo di una struttura, ma l'indirizzo del solo campo `next` della struttura. Infatti, `*pl` contiene l'indirizzo della prima struttura, per cui `(*pl)->val` e `(*pl)->next` sono i due campi della prima struttura. L'espressione `&(*pl)->next` dà l'indirizzo del campo `next` della prima struttura.



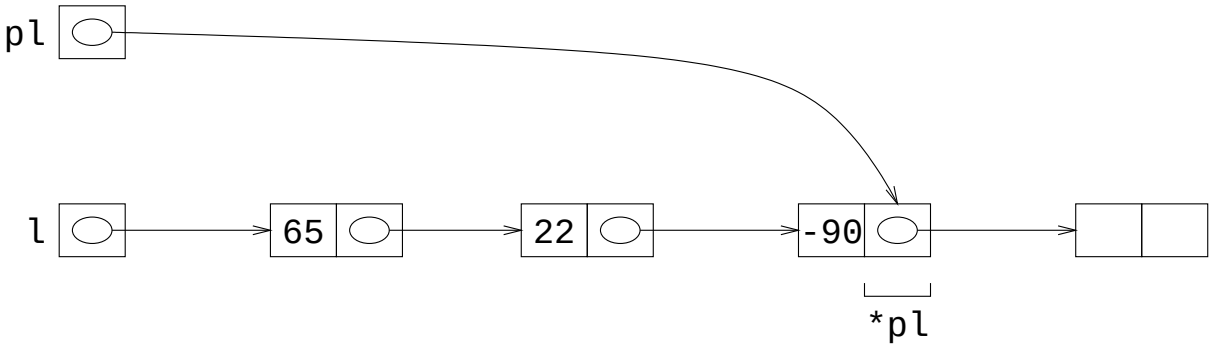
Dato che `*pl` non vale `NULL`, si fa la stessa assegnazione `pl=&(*pl)->next`, il cui effetto è ancora quello di ‘‘portare avanti’’ il puntatore.



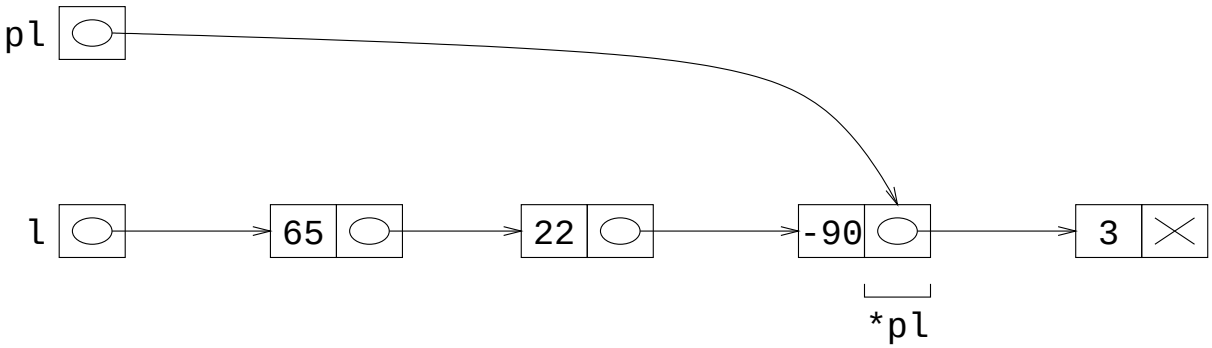
Dato che `*pl` non vale `NULL`, si porta ancora avanti il puntotore.



A questo punto, `*pl` vale `NULL`. Si fa la allocazione di una nuova struttura, e il suo indirizzo viene messo in `*pl`. Si noti che `pl` contiene l'*indirizzo* dell'ultima struttura della lista, per cui mettere qualcosa in `*pl` equivale a metterlo nel campo `next` dell'ultima struttura della lista. Quando mettiamo l'indirizzo della nuova struttura in `*pl`, lo stiamo mettendo nel campo `next` dell'ultima struttura della lista. Questo effettivamente crea una nuova struttura in coda alla lista.



A questo punto, possiamo riempire la struttura con i valori esatti (il numero da inserire e il valore `NULL` che chiude la lista).



Come si vede, la lista è stata modificata: un nuovo elemento è stato aggiunto in coda.

Il metodo del puntatore a puntatore presenta un vantaggio nel caso in cui la lista va modificata. Il vantaggio è che in qualsiasi momento della scansione (sia che siamo all'inizio, alla fine, o in mezzo alla lista) quello che abbiamo è sempre un puntatore a una variabile di tipo lista. Questo fa sí che si abbia un comportamento uniforme, ossia si devono fare le stesse cose sia che l'elemento vada inserito/cancellato all'inizio, in mezzo, oppure in coda. Il programma completo puntpunt.c contiene altre funzioni di inserimento e cancellazione che usano questo metodo.