

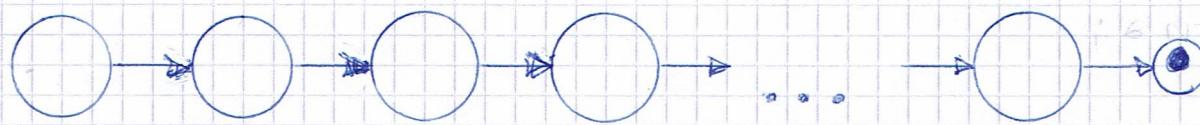
funzione termina. La funzione restituisce correttamente il puntatore alla prima cella dell'array (ricordiamo che, dato un array  $\geq [n]$ ,  $\geq$  equivale ad  $\&[0]$ , cioè è un puntatore alla prima cella dell'array), ma è un puntatore che, terminata la funzione, non ha più significato, dato che non punta più ad alcuna zona di memoria significativa: infatti l'array  $\geq [n]$  è stato "distrutto" assieme a tutto il record di attivazione della funzione `alloc_array - 2` non appena la funzione è terminata.

### Le strutture liste

La struttura "list" è una struttura dati che permette di gestire la memoria dinamica, o meglio, che sfrutta la memoria dinamica, in quanto consente di avere un numero variabile di elementi, non definito a priori.

Definiamo innanzitutto cos'è una lista dal punto di vista concettuale. Una lista è semplicemente una sequenza ordinata di elementi, sequenza dei nodi. Con "ordinata" intendiamo che c'è un primo elemento, seguito da un secondo, un terzo... e così via. Questo ordine non è definito in base a particolari caratteristiche degli elementi (ad esempio, se ho una lista di studenti, non necessariamente i nominativi sono disposti secondo l'ordine alfabetico, l'età, il numero di matricola...).

Graficamente possiamo rappresentare una lista come una serie di cerchi in cui ciascuno è collegato al successivo, fino all'ultimo cerchio che è collegato ad un segnale di terminazione, che sta a significare che la lista è terminata. Ogni cerchio rappresenta in modo, ovvero uno degli elementi della lista.



Quindi il primo nodo della lista mi dice chi è il secondo, il secondo chi è il terzo... e via dicendo, fino all'ultimo nodo della lista che è collegato ad un "nodo speciale" che ha la funzione di indicare che non ci sono più elementi all'interno della lista.

Piamo ora una definizione più formale e precisa di cosa si intende per lista. Per definire una lista servono solo due punti:

- Una lista vuota è una lista
- Un nodo seguito da una lista è una lista

Vediamo ora come, a partire da queste due sole regole, possiamo andare a costruire liste di qualunque dimensione.

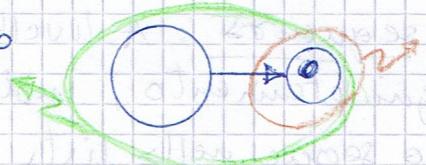
1) Il caso a) (detto anche caso base) ci dice che la lista vuota è una lista. Graficamente si può rappresentare come una lista il cui unico elemento sia il "nodo speciale" di fine lista

• lista vuota: in base al caso a) è comunque una lista.

Perché abbiano bisogno di una lista vuota? Ad esempio perché non è detto che, in qualsiasi momento ci siano elementi che soddisfano un determinato criterio: per esempio, in questo momento, la lista degli studenti del primo anno che hanno superato il corso di programmazione 2 sarà necessariamente una lista vuota, dato che, ad ora, nessuno studente del primo anno ha ancora avuto la possibilità di sostenere l'esame di programmazione 2, non avendo ancora nessuno studente del primo anno terminato il corso.

2) Applicando il caso b) alla lista ottenuta al punto 1), ottengo una lista di un solo elemento, ovvero combinando il caso b) con il caso a), posso definire liste con un solo elemento

Caso b): un nodo seguito da una lista è una lista

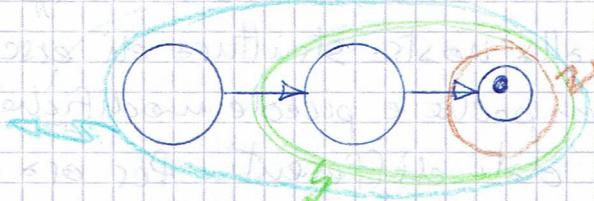


Caso a): una lista vuota è una lista

Quindi, data una lista vuota (che, per quanto stabilito dal caso a) è una lista, se si collega un nodo alla lista vuota si ottiene un nodo seguito da una lista, cosa che, in base al caso b), a sua volta una lista. Quindi anche la lista composta da un solo elemento è una lista.

3) Applicando di nuovo il caso b) alla lista ottenuta al punto 2) si ottiene una lista di due elementi, ovvero applicando una volta il caso a) e due volte il caso b) ottieniamo liste con due elementi

Di nuovo il caso b): è una lista



Caso a): è una lista

Caso b): è una lista

Quindi applicando una volta il caso a) e un numero di volte b) per ottenere il caso b) posso ottenere liste di qualunque lunghezza, tali da permettere di gestire un numero arbitrario di elementi.

Alcune considerazioni:

- La lunghezza della lista non è parte della definizione della lista stessa
- La lista vuota (ovvero, la lista che non contiene nemmeno oltre a quello di fine lista) è una lista a tutti gli effetti (e non è un errore!)
- È parte della definizione di lista la notione di sequenza ("seguito da"; nella definizione, ogni elemento indica il suo successivo).
- Vediamo, quale è la struttura dati in C che permette di gestire una lista e la definisce, che non c'è da nessuna parte la menzionazione della sua lunghezza, ovvero non c'è una variabile o un campo che memorizzi la lunghezza di una lista. Se si ha bisogno di questo dato, va calcolato.
- Ancora, la lista vuota è una lista a tutti gli effetti. Non è un errore: semmai è un caso particolare di lista. Quindi tutti i programmi che lavorano con le liste dovranno prevedere e gestire il possibile caso di ricevere in input una lista vuota.
- Inoltre, nella definizione di lista c'è una notione implicita essenziale, che è quella di sequenza. A livello pratico, ciò significa che ogni nodo, cioè ogni elemento, indica esplicitamente quale sarà il nodo che lo segue nella lista. Quindi nel primo nodo sono contenute informazioni su dove si trova il secondo nodo, nel secondo nodo ci sono informazioni su dove si trova il terzo... e così via. Quindi in ogni nodo ci sono le informazioni necessarie per poter passare al nodo seguente (ma non al nodo precedente). Queste liste nelle quali ogni elemento contiene informazioni sul successivo vengono chiamate liste linkate o singolarmente linkate.

Vediamo ora come si realizza una lista in C.

La prima cosa che abbiamo è un "typedef" molto semplice, così che permette alla nostra struttura di essere abbastanza generica, al modo che, con poche e piccole modifiche, si arriva a gestire molte tipologie di dati differenti. Per ora, per semplicità, si lavorerà solo su tipologie di dati interi, ovvero di tipo int.

Nella vieta però di prendere gli stessi algoritmi che vediamo e generalizzarli con altri tipi di dati. Un modo molto semplice per lavorare sulla struttura è cambiare la typedef iniziale.

Nel momento in cui ho definito "DATA", che è la parte concettuale che rappresenta le informazioni dell'elemento (un intero, nel caso indicato), nulla vieta di poter cambiare la typedef con un char, o un qualsiasi altro tipo di dato, eventualmente anche definito dal programmatore (ad esempio il "point" visto nella lezione precedente).

Vediamo poi quella che è la parte centrale delle nostre lists, ovvero l'elemento. Lo si definisce tramite una struct che andiamo a chiamare "linked-list", composta dai due componenti:

- 1) Il dato, l'informazione che vogliamo memorizzare in ogni elemento
- 2) Ciò che rappresenta il contesto di sequenza, ovvero quelle informazioni che mi permettono di identificare l'elemento successivo nella lista

```
typedef int DATA;  
struct linked-list  
{  
    DATA d;  
    struct linked-list *next;  
};
```

typedef è il tipo di dato  
Parte centrale della lista, cioè l'elemento. Si definisce un struct che ha due componenti:  
1) Il dato  
2) Il puntatore all'elemento successivo

La prima delle due componenti, ovvero il dato, è un contesto abbastanza intuitivo.

La seconda che definizione è una "struct nome lists \*": ovvero un puntatore a qualcosa. Ma a che cosa? La definizione è in qualche modo ricorsiva: sto definendo struct linked-list e all'interno di un suo campo, di una sua parte, si trova lo stesso riferimento alla struttura. Siccome stiamo parlando di elementi che verranno allocati dinamicamente, noi vogliamo fornire indicazioni su dove si trova la zona di memoria che contiene l'elemento successivo della lista. Se si trattasse di un semplice array non ci sarebbe bisogno di queste informazioni: essendo un array allocato in celle di memoria continue fra loro non c'è la necessità di indi-

ère dove si trova l'elemento successivo: supponiamo, per esempio, che gli elementi della lista siano di tipo int e che il prossimo elemento sia nella cella di memoria contigua a quella dell'elemento considerato.

Se però gli elementi vengono creati tramite delle malloc, non è affatto detto che il prossimo elemento sia contiguo a quello appena creato: infatti se richiediamo della memoria per un dato tramite un malloc, questa memoria può venire allocata in qualunque punto della heap; se andiamo a richiedere altre memorie subito dopo non è detto che l'allocazione avvenga nella zona della heap contigua a quella precedentemente allocata: può essere, ma non è detto che lo sia.

Quindi l'unico modo certo per poter accedere all'elemento successivo di una lista è che da qualche parte ci sia scritto l'indirizzo di memoria dove l'elemento si trova. Il tipo di dato necessario per memorizzare un indirizzo di memoria è il puntatore. Non mi basta però sapere l'indirizzo delle zone di memoria, ma devo sapere anche che queste zone di memoria hanno un determinato tipo di dato allocato, che in questo caso deve corrispondere all'elemento corrente, quindi corrispondente alla struttura stessa che stiamo andando a creare. Per semplicità, aggiungeremo due ulteriori typedef e riscriviamo l'intero codice

```
typedef int DATA;  
struct linked-list  
{  
    DATA d;  
    struct linked-list *next;  
};  
typedef struct linked-list ELEMENT; ①  
typedef ELEMENT *LIST; ②
```

- ① La prima delle due typedef aggiunte mi dice che la struttura linked-list appena definita sopra è un elemento
- ② La seconda è quella importante e mi va a definire in C che cos'è il concetto di lista: mostreremo poi tramite una rappresentazione grafica perché questo è importante. Questo secondo typedef mi dice che un dato di tipo LIST è un puntatore ad ELEMENT,

ossia sul elemento, è nullo.

In C esiste un keyword adatto ai puntatori per dire che quel puntatore non accede a nessuna zona di memoria. L'keyword è NULL - se si assegna questo valore ad un puntatore in realtà sto dicendo che non c'è alcuna zona di memoria a cui accedere e quindi il puntatore in realtà non punta a nulla.

Appoggiandosi a questo keyword possiamo evitare di avere due tipi di nodi. Ripensiamo alla rappresentazione grafica dell'inizialmente delle liste: avevamo due tipi di nodi, ovvero quelli "normali" (il pallino più grande vuoto) e quello per indicare la fine della lista (il pallino più piccolo con un punto nero al suo interno). Usando l'keyword NULL già possiamo indicare che quel puntatore non accede a nessuna zona di memoria: quindi se assegniamo il valore NULL al puntatore di un elemento, stiamo indicando che dopo l'elemento in cui ci si trova non c'è più nulla, e che quindi la lista è terminata.

Quindi suonando a vedere dove si trova il puntatore che contiene NULL si potrà capire se la lista è immediatamente vuota.

Si può sfruttare queste cose per fare in modo che all'inizio della lista tramite un semplice valore (e quindi anche un semplice test) si possa sapere se la lista è immediatamente vuota oppure se c'è un elemento. Mi basta testare il contenuto della prima parte di LIST, ovvero di quella che noi andremo a chiamare testa della lista, per sapere se la lista è vuota o meno.

Vediamo ora la rappresentazione grafica di una lista con quattro elementi, tenendo conto di quanto ora detto.



Nel momento in cui verrà definito ad esempio una variabile "l1" di tipo LIST, sarà allocato lo spazio per un puntatore a elemento, quindi un indirizzo di memoria per sapere dove si trova (se allocato!) il primo elemento della lista. In questa lista ci sono 4 nodi, ovvero 4 elementi; la mia lista l1 inizia dalla prima cella a sinistra, che contiene solo lo spazio per un puntatore e non ha nessuna parte dati. È l'cosiddetta testa della lista, che

non è un nodo

Seguire 4 elementi; ognuno di questi ha due componenti:  
una parte dati e una zona "next" che contiene un puntatore che permette di accedere all'elemento successivo. Ricordiamo che i puntatori sono mono direzionali, infatti vengono rappresentati come una freccia che va in un'unica direzione: quindi quando ci si trova in un nodo si può capire quale sarà il nodo successivo, ma non si può risalire > quello che era il nodo precedente.  
Quindi una lista può venire percorso dalla testa fino alla fine, ma non posso fare il contrario, ovvero non ho posso percorrere dalla fine all'inizio.

Nella parte "next" dell'ultimo nodo verrà memorizzato il valore NULL, che mi indica che dopo non c'è più alcun valore, e che cioè la lista è terminata.

