

Allocare / deallocare memoria

In questa lezione si vedrà come allocare e deallocare memoria in un programma C. Quando abbiamo visto la suddivisione della memoria fatta da un programma C, abbiamo visto che c'è una zona specifica della memoria, ovvero la heap, dedicata a gestire le richieste dinamiche (= runtime) di allocazione o di deallocazione della memoria; abbiamo anche indicato diversi comandi (`malloc`, `free` e `realloc`) che permettono di fare ciò. In questa lezione andremo a studiare tali comandi in dettaglio.

`malloc`: permette di allocare (= richiedere) memoria

`free`: permette di liberare memoria precedentemente allocata

La funzione `malloc()`:

- Sintassi: `void *malloc (size_t dim)`
- Libreria: `<stdlib.h>`
- Input: quantità di memoria richiesta
- Cosa fa: richiede di allocare una quantità di memoria pari a `dim byte`
- Cosa restituisce: il puntatore iniziale alla zona di memoria allocata.

Allora la funzione `malloc` ha un solo parametro in input (ovvero la dimensione in byte della quantità di memoria che vogliamo allocare) e restituisce in output un puntatore a `void`; la `malloc` richiede (ed ottiene) un blocco di memoria all'interno della `heap` di dimensioni pari al numero dei byte specificati nell'unico dato che la funzione riceve in input; restituisce dove si trova quel blocco di memoria (di dimensione specificata in input) all'interno della `heap`, tramite un puntatore a `void`, che è il metodo tramite il quale si accederà a tale zona di memoria.

Inizieremo a chiarire cosa si intende con "puntatore a `void`": un certo è dire "`void`", un altro certo è dire "puntatore a `void`".

Nel momento in cui scriviamo una funzione tipata "`void`" ciò significa che quella funzione non restituisce nulla, cioè non ha nessun valore di ritorno.

Invece una funzione di tipo "`void *`" ha un valore di ritorno, ossia ritorna un puntatore ad una zona di memoria, senza però che

quelle zone sia tipata (ovvero non è specificato il tipo preciso).

Quindi

- Funzione di tipo void : non restituisce nulla
- Funzione di tipo void* : restituisce un puntatore a void, ovvero un puntatore ad una zona di memoria non tipata

La funzione malloc è una funzione generale, che deve funzionare per qualunque tipo di richiesta : sarebbe scimmodo avere tante malloc diverse a seconda del tipo di dato per il quale vado a fare una richiesta di allocazione (sarebbe centare che, come visto nella lezione precedente, il programmatore può definire dei nuovi tipi di dati) ; ipotizzando l'esistenza di malloc specifiche a seconda dei vari tipi di dati, comunque ci mancherebbero le malloc per i tipi di dato definiti dal programmatore).

Quindi la malloc restituisce un puntatore a void, ovvero senza specificare il tipo : cioè mi permette di accedere ad una zona di memoria non dicendo esplicitamente cosa troverò dall'altra parte ; questo lo deve fare il programmatore che, per lavorare correttamente su queste zone di memoria, dovrà immediatamente effettuare un cast esplicito : vedremo successivamente cosa questo significa dal punto di vista del codice che c'è da scrivere.

Riassumendo :

- La funzione malloc () restituisce un puntatore a void, ossia un indirizzo di memoria, un puntatore senza tipo
- per assegnarselo ad uno specifico puntatore occorre un cast esplicito, che va fatto dal programmatore.

Abbiamo detto che la quantità di memoria da richiedere tramite malloc è espressa in byte. Come posso sapere quanta memoria devo richiedere ? La quantità di memoria utilizzata per una variabile cambia, oltre che in base al tipo della variabile, anche in base all'architettura del pc (32 o 64 bit) e in base al sistema operativo : quindi non possiamo immaginare di specificarne nella malloc un numero preciso di byte e immaginare che il programma funzioni correttamente su qualunque macchina e su qualunque sistema operativo.

Oltre tutto, se passassi a scrivere direttamente il numero di byte, ogni volta obbligherei consultare tabelle per sapere quale sia la dimensione

sare necessario per il tipo di dato che mi serve, le specifiche della macchina, quelle del sistema operativo... sarebbe un processo ingiunzoso e dispendioso in termini di tempo.

Possiamo quindi usare in C un operatore che si occupa di questo compito: l'operatore sizeof.

L'operatore sizeof riceve in input il tipo di una variabile e mi restituisce il quantitativo di byte occupato da una variabile di quel tipo.

Operatore sizeof

- Sintassi: `sizeof (data-type)`
- Input: il tipo di dato di cui si vuole sapere la dimensione occupata
- Cosa fa: calcola quanti byte occupa quel tipo di dato

Il data-type può sia essere uno dei tipi predefiniti in C (char, int, float, double...), sia uno dei tipi che l'utente va a definire (come visto nella lezione precedente con typedef)

Per esempio, se voglio sapere quanti byte occupa una variabile di tipo int, semplicemente scrivo `sizeof (int)`; per avere il quantitativo di memoria occupato da n interi, semplicemente scrivo `n * sizeof (int)` (es. 12 interi $\Rightarrow 12 * \text{sizeof} (\text{int})$)

Vediamo alcuni esempi, andando a vedere cosa succede in memoria.

Consideriamo questo semplice programma:

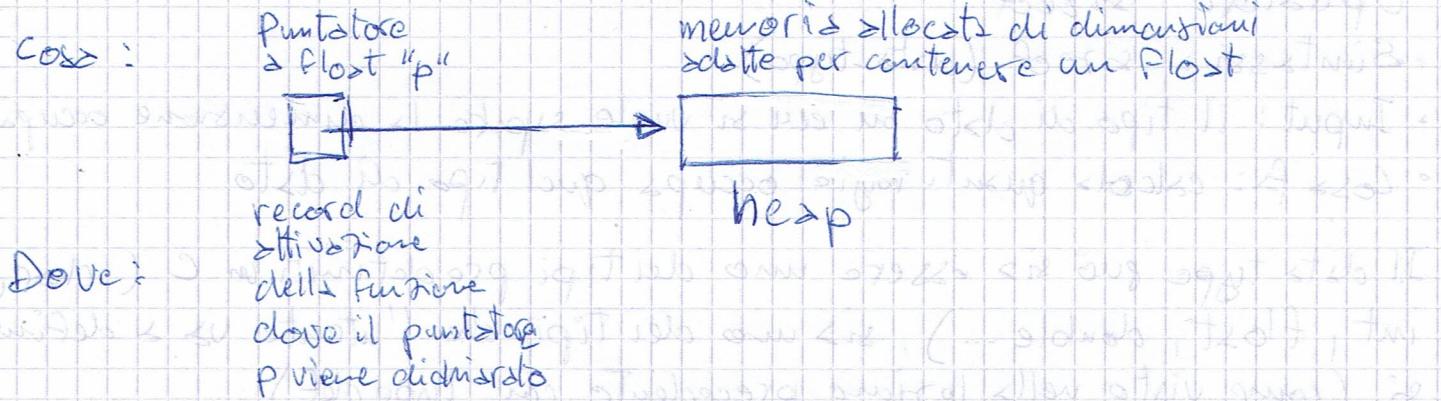
```
float* p;  
p = (float*) malloc(sizeof(float));
```

Alla prima riga si dichiara un puntatore p ad un float: il puntatore p è dichiarato, ma per il momento non punta a nulla, ovvero non gli è stato assegnato alcun valore, alcun contenuto.

L'assegnamento di un valore a p è proprio ciò che si va a fare nella seconda riga: a p si assegna l'indirizzo di memoria di un blocco all'interno dell'heap di dimensioni pari al quantitativo di byte occupato da un float, ovvero `sizeof (float)`; come si vede, visto che `malloc` restituisce di suo un puntatore void mentre io ho bisogno in questo caso di un puntatore a float, la prima cosa che faccio in sede di assegnamento è fare un cast esplicito al tipo di dato che mi serve, cioè in

questo caso un puntatore \rightarrow float; Ecco spiegato il "(float*)"
subito dopo "p =": non è altro che un cast esplicito che
un "trasforma" il puntatore \rightarrow void (restituito dalla malloc)
 \rightarrow un puntatore \rightarrow float.

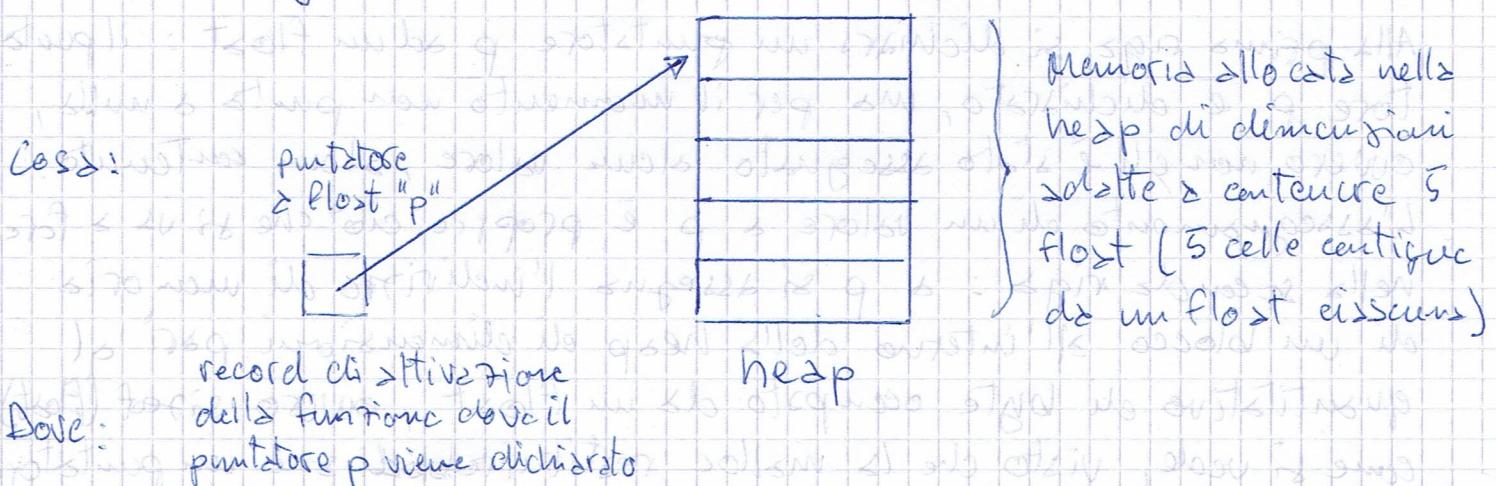
Avremo nel puntatore p viene scritto l'indirizzo della cella di
memoria allocata nella heap tramite il comando malloc; la dimensione
allocata sarà quella voluta \rightarrow contenere una variabile di
tipo float. A questo punto potrò usare il puntatore p per
accedere alla cella di memoria allocata nella heap.



Cosa cambia se invece di richiedere lo spazio per un float richiede lo spazio per 5 float? Vediamo nell'esempio seguente

```
float* p;  
p = (float*) malloc (5*sizeof(float));
```

In questo caso si alloca nella heap lo spazio per contenere 5 float,
ossia 5 celle di memoria contigue, ma in seguito all'allocazione
nel puntatore p verrà scritto l'indirizzo della prima di queste
5 celle contigue



Quindi in pratica ha realizzato un array di dimensione 5 per dei float:
quindi si può lavorare su p tramite l'algebra dei puntatori, oppure come
se fosse un normalissimo array di float; l'unica differenza con un norma-

l'array di float è che in questo caso le zone di memoria in cui queste 5 celle contigue per dei float non è legata ad una singola funzione, cioè non è in uno specifico record di attivazione. Ma è nella heap.

Abbiamo visto come allocare della RAM nella heap mediante la funzione malloc(), vediamo ora come disallocare zone di memoria precedentemente allocate tramite la funzione free.

La funzione free()

- sintassi: void free(void*)
- libreria: <stdlib.h>
- Input: il puntatore ad una zona di memoria allocata precedentemente con malloc.
- Cosa fa: la zona di memoria indicata viene liberata (cioè viene marcata come libera); il puntatore non punta più ad una zona di memoria significativa.
- Cosa restituisce: è una funzione di tipo void, ossia non ha nessun valore di ritorno.

La funzione free richiede in input un puntatore ad una zona di memoria precedentemente allocata con malloc; non ha bisogno di sapere quale tipo di dato fosse contenuto, cioè non richiede (contrariamente alla malloc) un esatto esplicito, ma si limita a liberare la zona di memoria puntata dal puntatore che riceve in input; non restituisce alcun valore.

Pubblidiemolo: possiamo disallocare solo e soltanto blocchi di memoria cui abbiamo fatto precedentemente richiesta di allocazione esplicita tramite malloc e nulla' altro.

Vediamo un esempio di programma in cui si usano queste due funzioni, ad esempio andando ad allocare un array di float il cui numero è ricevuto in input dall'utente.

```
1 #include<stdio.h>    7 printf("Dimensione dell'array? ");
2 #include <stdlib.h>    8 scanf("%d", &n);
3 int main ()           9 v= (float*) malloc(n * sizeof (float));
4 {                      10 // --- uso dell'array --,
5     float* v;          11 free(v);
6     int n;             12 return 0;
7 }                      13 ?
```

Il programma nel main definisce due variabili locali: v, che è un puntatore a float, e n, che è un intero; quindi chiede all'utente una dimensione per l'array e mette questo valore in n; quindi, in riga 9, va a fare una richiesta di allocazione esplicita di una zona di memoria pari ad "n" volte la dimensione di un float e, attraverso un cast esplicito ad un puntatore a float, mette in v l'indirizzo della prima di queste n celle contigue; dopo dichiara l'array verso utilizzato in qualche modo; alla fine del suo utilizzo si svolge (riga 11) a liberare la zona di memoria allocata semplicemente con free(v); quindi il programma termina.

Facciamo un altro esempio: scrivere una funzione che prende in input un intero e allochi e restituisce un array di interi della dimensione specificata. Rispetto all'esempio precedente (oltre al fatto che in questo caso si tratti di un array di interi anziché di float, ma dal punto di vista del programma questa differenza non è significativa) la differenza principale è che il processo di allocazione non avviene nel main, ma all'interno di una specifica funzione.

La funzione "alloc-array" quindi deve prendere in input un intero (la dimensione dell'array) e restituire un puntatore ad intero che entri nell'indirizzo della prima delle celle contigue che costituiscono la zona di memoria allocata. L'intestazione della funzione sarà dunque

```
int* alloc-array (int n)
```

La funzione è molto semplice: di fatto è costituita solo da un return del puntatore desiderato, ovvero esegue la richiesta di allocazione esplicita direttamente nella return

```
1 #include <stdlib.h>
2
3 int* alloc-array (int n)
4 {
5     return (int*) malloc (n * sizeof (int));
6 }
```

Ovviamente nulla avrebbe vietato di definire una variabile locale puntatore ad intero interno alla funzione, eseguire la malloc relativamente a questo puntatore e ritornare il puntatore stesso

```
1 #include <stdlib.h>
2
3 int* alloc_array (int n)
4 {
5     int* p;
6     p = (int*) malloc (n * sizeof (int));
7     return p;
8 }
```

Perché questo sistema funziona? Perché lo spazio che uso ad allocare non è una variabile locale della funzione, ovvero non appartiene al record di attivazione di "alloc_array", ma viene collocato nella heap: quindi avendo a disposizione un puntatore che punta a dove inizia questo spazio, posso utilizzarlo in qualsiasi parte del programma, magari o qualunque altra funzione che sia (cioè esternamente alla funzione che ha eseguito la malloc)

Vediamo una versione errata della funzione alloc_array

```
1 int* alloc_array-2 (int n)
2 {
3     int a [n];
4     return a;
5 }
```

Dà un punto di vista meramente sintattico, questa funzione non ha errori: la tipografia di questa funzione è identica alla precedente (la funzione riceve un intero in input e restituisce un puntatore ad intero).

Il problema è che l'array a di dimensione n che viene dichiarato in riga 3 è una variabile locale della funzione alloc_array-2, cioè viene memorizzato nello stack all'interno del record di attivazione della funzione, quindi smette di esistere appena la

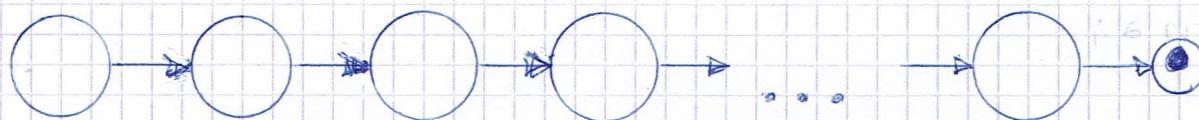
funzione termina. La funzione restituisce correttamente il puntatore alla prima cella dell'array (ricordiamo che, dato un array $\geq [n]$, \geq equivale ad $\&[0]$, cioè è un puntatore alla prima cella dell'array), ma è un puntatore che, terminata la funzione, non ha più significato, dato che non punta più ad alcuna zona di memoria significativa: infatti l'array $\geq [n]$ è stato "distrutto" assieme a tutto il record di attivazione della funzione `allocarray-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, ciascuno dei quali node. Con "ordinata" intendiamo che c'è un primo elemento, seguito da un secondo, intorno... 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 un 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:

a) Una lista vuota è una lista

b) Un nodo seguito da una lista è una lista