

Allocazione dinamica della memoria

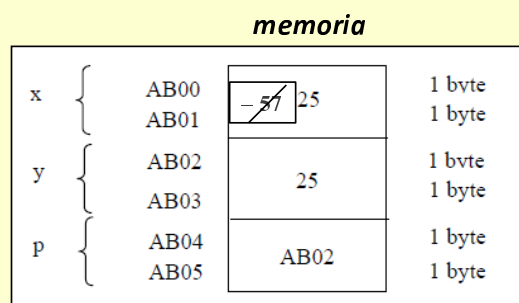
1

Allocazione statica della memoria

Cosa succede nella memoria del computer quando il un programma vengono **dichiarate** delle variabili?

Ad esempio:

```
int main()
{ int x= - 57 , y=25;
  int *p;
  p=&y;
  x=*p;
}
```



Le dichiarazioni determinano la presenza, in questo caso, di **tre locazioni di memoria x, y e p** allocate nell'area dati, come mostrato nella seguente figura.

Questo modalità di **allocazione** della memoria viene detta **statica**.

Le variabili dichiarate in questo modo **possono variare il loro contenuto**, ma **non possono variare le loro caratteristiche** a tempo di esecuzione.

Ad esempio, se dichiariamo: `int V[5];`

non possiamo modificare l'array a tempo di esecuzione in modo che questo contenga un numero di elementi che sia superiore a 5.

2

Allocazione *statica* della memoria

- La quantità di **memoria** da allocare è **determinata** e **fissata a tempo di compilazione** automaticamente dal sistema.
- Ogni variabile statica ha un **nome**, attraverso il quale la si può riferire.
- Il **programmatore non ha la possibilità di influire sul tempo di vita** di queste variabili che verranno deallocate alla terminazione del programma o alla chiusura di un blocco di istruzioni (es.: { ... }, for(int i=....)).

Allocazione *dinamica* della memoria

- Permette di **definire variabili che possono essere create o accresciute in fase di esecuzione**.
- Le variabili dinamiche devono essere **allocate e deallocate esplicitamente dal programmatore**.
- L'area di **memoria** in cui vengono allocate le variabili dinamiche si chiama **heap** (in C) o **memoria libera** (in C++).
- Le variabili dinamiche non hanno un **identificatore (nome)**, ma possono essere riferite **soltanto** attraverso il loro **indirizzo** (mediante i **puntatori**).
- Il tempo di vita delle variabili dinamiche è l'intervallo di tempo che intercorre tra l'allocazione e la deallocazione (che sono impartite esplicitamente dal programmatore).

3

Allocazione della memoria

Il programma compilato è costituito da due parti distinte:

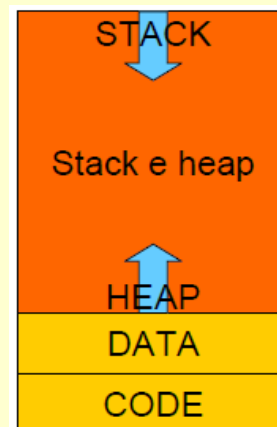
- Code Segment
- Data Segment

Quando il programma viene eseguito, il Sistema Operativo riserva uno spazio di memoria per allearvi **stack** e **heap**.

Se lo spazio per stack e heap ha dimensione fissa (dipende dal S.O.), questo può esaurirsi per effetto di ripetute chiamate a funzione (stack) non chiuse o allocazioni dinamiche (heap o memoria libera).

Esiste infatti una proprietà che lega heap e stack:

allocando memoria in una delle due aree, questa aumenta mentre l'altra diminuisce, rientrando ovviamente nei limiti imposti dalla quantità di memoria disponibile.



4

Allocazione dinamica della memoria in C

Le **funzioni** che il programmatore può usare per accedere allo **heap** (in C):

- Si trovano in <stdlib.h>
- Allocano un generico **blocco contiguo** di byte
- **Restituiscono l'indirizzo di memoria** (di tipo puntatore-a-void) del primo byte del blocco, (**NULL in caso di errore**: allocazione non riuscita, dovrebbe essere sempre controllato)
- **malloc (dim)** **alloca un blocco di byte** non inizializzato composto da un numero di byte contigui pari a $dim = N * dimensione_elemento$ e restituisce il puntatore a tale area.

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

Il blocco allocato di byte non ha di per sé alcun tipo, il cast sul puntatore restituito fa sì che il blocco di byte sia considerato dal compilatore come avente il tipo indicato nel cast.

Nell'esempio il cast (int *) fa sì che il compilatore consideri il blocco di byte sufficiente a memorizzare un intero.

Non si può applicare l'operatore sizeof a un blocco di memoria allocato dinamicamente in quanto sizeof viene valutato dal compilatore.

5

Allocazione dinamica della memoria in C

Le **funzioni** che il programmatore può usare per accedere allo **heap** (in C):

- **calloc (N, dimensione_elemento)** **alloca un blocco di byte** inizializzato, composto da un numero di byte pari a $N * dimensione_elemento$.

calloc inizializza anche a 0 lo spazio di memoria allocato. È quindi simile alla **malloc**, ma, a differenza di questa, è meno veloce.

```
int *p;
p = (int *) calloc(sizeof(1,int));
```

La sintassi è:

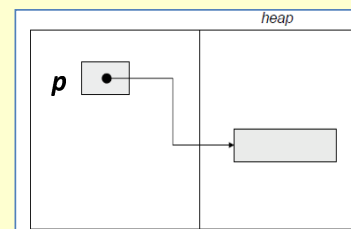
```
p = (tipodato *) malloc (sizeof(tipodato));
p = (tipodato *) calloc (N,sizeof(tipodato));
```

dove:

- **tipodato** è il tipo della variabile puntata
- **p** è una variabile di tipo **tipodato ***
- **sizeof()** è una funzione standard che calcola il numero di byte che occupa il dato specificato come argomento

La **malloc** e la **alloc**

- provocano la creazione di una variabile dinamica nell'**heap**
- restituiscono **l'indirizzo alla prima cella dell'area allocata.**



6

Allocazione dinamica della memoria in C

Esempi di allocazione

Vettore unidimensionale dinamico

```
int *p;
p=(int *) malloc (sizeof(int)*100); // Viene allocato un blocco di byte delle
// dimensioni di 100 int;
// è l'assegnazione ad un puntatore a int
// (il cast è opzionale) che fa sì
// che il C lo "veda" come un vettore di int
```

Vettore unidimensionale dinamico

```
int *p;
p=(int *) calloc (100, sizeof(int)); // Viene allocato un blocco di byte delle
// dimensioni di 100 int inizializzati a 0;
```

Il cast esplicito non è necessario usando un compilatore C standard, perché l'assegnazione di un puntatore void ad un puntatore non-void non lo richiede necessariamente. Alcuni compilatori (in particolare quelli che compilano anche codice C++) lo richiedono comunque e quindi è bene aggiungerlo, anche per chiarezza e documentazione.

Utilizzo:

```
*p = 3;           oppure p[0] = 3;
*(p+12) = 19;    p[12] = 19;
```

7

Rilascio della memoria dinamica in C

La memoria allocata dinamicamente **non viene rilasciata automaticamente** (ovviamente questo capita comunque quando il programma termina in quanto tutta la memoria associata al programma viene rilasciata).

Il C non ha un **garbage collector** che "recupera" a run-time la memoria inutilizzata.

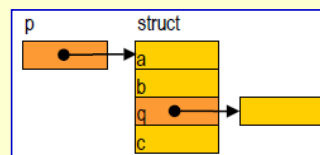
A run-time la memoria può essere rilasciata solo in modo esplicito, questo permette di riutilizzare quella porzione di memoria per servire successive allocazioni.

- **free(p)** rilascia il blocco di memoria puntata dal puntatore **p** (il sistema mantiene memoria del numero di byte che era stata allocato)

La sintassi è: **free (p);**

Se un puntatore **p** punta ad una variabile dinamica contenente un puntatore **q** ad un'altra variabile dinamica, bisogna rilasciare prima **q** e poi **p**:

```
free(p->q);
free(p);
```



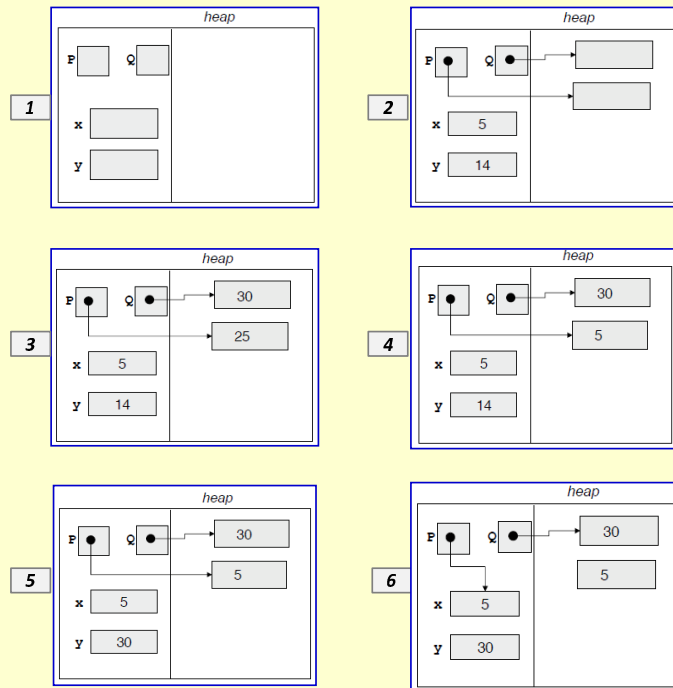
8

Allocazione dinamica e rilascio della memoria dinamica in C

Eempio:

```
#include <iostream>
```

```
int main()
{ int *P, *Q, x, y;
  x=5;
  y=14;
  P=(int*) malloc(sizeof(int));
  Q=(int *) malloc(sizeof(int));
  *P = 25;
  *Q = 30;
  *P = x;
  y = *Q;
  P = &x;
  ....
  free (Q);
}
```



l'ultimo assegnamento ha come effetto collaterale la **perdita dell'indirizzo di una variabile dinamica** (quella precedentemente referenziata da P) che rimane allocata, ma **non é più utilizzabile!**

9

Problemi legati all'uso dei Puntatori

1. Aree inutilizzabili:

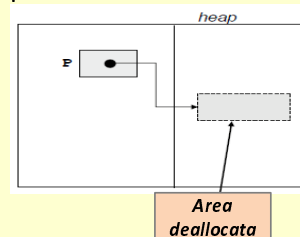
Possibilità di perdere l'indirizzo di aree di memoria allocate al programma che quindi non sono più accessibili. (v. esempio precedente).

Se queste mancate deallocazioni delle aree di memoria aumentano si provoca un **memory leak** dovuto all'esaurimento della memoria.

2. Riferimenti pendenti (dangling reference) :

Possibilità di fare riferimento ad aree di memoria non più allocate.

```
int *P;
P = (int *) malloc(sizeof(int));
...
free(P);
*P = 100; /* Da non fare! */
```



3. Riferimenti inesistenti:

La malloc non garantisce il successo dell'operazione. Se non c'è memoria disponibile, o se il programma ha superato il limite di memoria che può referenziare, malloc restituirà un puntatore null.

Molti programmi non controllano questa eventualità. Può quindi accadere di utilizzare il puntatore null restituito dalla malloc come se fosse un normale puntatore ad un'area di memoria allocata, il programma andrà in crash.

Allocazione dinamica e rilascio della memoria dinamica in C

Riassumendo

I prototipi delle funzioni **malloc**, **calloc**, **realloc** e **free** sono i seguenti:

<pre>void * malloc (dim_totale);</pre> <pre>void * calloc (int num_elementi, int dim_elemento);</pre>	<p>Restituiscono un puntatore void che indica che si tratta di un puntatore ad una regione di dati di tipo sconosciuto. Restituiscono NULL se non riescono ad allocare la quantità di memoria richiesta.</p>
<pre>void * realloc (void *ptr, int dim_totale);</pre>	<p>Essa modifica la dimensione di un blocco di memoria puntato da ptr a dim_totale bytes. Se ptr è NULL, l'invocazione è equivalente ad una malloc(dim_totale). Se dim_totale è 0, l'invocazione è equivalente ad una free(ptr).</p>
<pre>void free (void *ptr);</pre>	<p>Essa libera lo spazio di memoria puntato da ptr, il cui valore proviene da una precedente malloc() o calloc() o realloc() e lo restituisce al S.O. Se ptr è NULL non viene eseguita nessuna operazione.</p>

11

Allocazione dinamica della memoria in C

Esercizi

Utilizzare variabili e/o vettori allocati dinamicamente per risolvere i seguenti problemi.

1. Acquisire tre valori reali e calcolare il massimo e il minimo (allocare dinamicamente tre variabili).
2. Creare un array di float di dimensione specificata dall'utente, caricare da tastiera i valori e comunicarlo in ordine inverso.
3. Calcolare la media degli elementi di una sequenza di valori double inseriti dall'utente. La fine della sequenza è identificata dal numero 0.
4. Siano dati N punti appartenenti al piano cartesiano (caricarli in una tabella) e, successivamente, dato un numero > 0, corrispondente alla posizione nella tabella di un qualsiasi punto, comunicare la lunghezza della spezzata che unisce tutti i punti fino a quello specificato.

12