

# Allocazione dinamica

VLA, malloc() e dintorni

# Rappresentazione di sequenze ....

- È molto comune dover rappresentare sequenze di elementi tutti dello stesso tipo e fare operazioni su di esse:

- Es: sequenza di interi (23 46 5 28 3), sequenze di misurazioni (2.1 3.4 5.7678), sequenza di caratteri ('x' 'r' 'f') etc ....

- Finora abbiamo usato gli array per realizzare le sequenze,

- Questo però porta talvolta a sprecare memoria, o a scrivere codice inefficiente: ad esempio

- Se la sequenza viene modificata nel tempo con molte inserzioni di nuovi elementi ed eliminazione di elementi presenti

- Quando non sappiamo un limite superiore al numero di elementi da inserire! In questo caso, il nostro programma con un array statico potrebbe non funzionare più una volta raggiunto il limite degli elementi allocati

- Vediamo meglio il problema

# Sequenze di lunghezza ignota

```
/* il problema...lettura e memorizzazione di una sequenza di lunghezza non
nota, valori reali terminati da 0.0 */

#include <stdio.h>

#define N 10

typedef enum {false,true} boolean;

int main( void ) {
    int i = 0;
    boolean fine=false;
    double value[N];
    do { printf("Inserisci il valore %d:",i);
        scanf("%lf",&value[i]);
        if ( value[i] == 0.0 ) fine=true;
        i++;
    } while (( i < N )&& !fine) ;
    printf("Letti %d valori della sequenza\n",i);
    /* poi elaborazione..... */ }
```

Devo bloccarmi a N  
Anche se ci sono altri  
valori da leggere!

# Sequenze di lunghezza ignota

```
/* il problema...lettura e memorizzazione di una sequenza di lunghezza non nota,
valori reali terminati da 0.0 */

#include <stdio.h>

#define N 10

typedef enum {false,true} boolean;

int main( void ) {
    int i = 0;
    boolean fine=false;
    double value[N];
    do { printf("Inserisci il valore %d:",i);
        scanf("%lf",&value[i]);
        if ( value[i] == 0.0 ) fine=true;
        i++;
    } while (( i < N )&& !fine) ;
    printf("Letti %d valori della sequenza\n",i);
    /* poi elaborazione..... */ }
```

Il problema è che  
non conosco la  
lunghezza degli array che mi  
servono in fase di compilazione!

# Servono altri strumenti ...

- **VLA (Variable length array)**
  - Array di lunghezza variabile allocati sullo stack (C99)
- **Allocazione dinamica di memoria**
  - Funzioni di libreria che permettono di allocare dinamicamente memoria (non solo array ...) sullo heap
- **Liste, alberi (tipi ricorsivi)**
  - Tipi con caratteristiche diverse dagli array che permettono di rappresentare le sequenze in modo più flessibile
  - ad esempio anche eliminando ed aggiungendo elementi in mezzo alla sequenza senza dover modificare o ricopiare tutti quello che c'e' intorno
  - Più altre proprietà interessanti che vedremo in seguito ...

# Variable Length Array (VLA)

- Introdotti nel C99, sono array di lunghezza nota a tempo di esecuzione allocati sullo stack
  - Ma una volta allocati **non possono variare la propria dimensione**
- **Possono solo essere dichiarati come variabili locali (all'inizio di un blocco)**
  - **E vengono deallocati automaticamente all'uscita del blocco (fanno parte del frame sullo stack!)**
  - **In particolare se allocati dentro una funzione sono deallocati all'uscita**

```

/* nel nostro esempio sono utili solo se qualcuno ci dice la lunghezza massima
possibile della sequenza */

#include <stdio.h>

int main( void ) {

    boolean fine=false;

    int i=0, n;

    printf("Inserisci la lunghezza:"); scanf("%d",&n);

{ /* inizio blocco, necessario dichiarare il VLA */

double value[n]; /* VLA */

    do { printf("Inserisci il valore %d:",i);

        scanf("%lf",&value[i]);

        if ( value[i] == 0.0 ) fine =true;

        i++;

    } while (( i < n )&& !fine) ;

/* tutto l'eventuale calcolo deve rimanere dentro il blocco in cui è
dichiarato il VLA */

} /* fine blocco VLA */

return 0; } /* fine main */

```

# Variable Length Array (VLA)

- Possono solo essere dichiarati come variabili locali (all'inizio di un blocco)
  - E vengono deallocati automaticamente all'uscita del blocco (fanno parte del frame sullo stack!)
  - In particolare se allocati dentro una funzione sono deallocati all'uscita
- È anche sconsigliato usare VLA troppo grandi perchè essendo allocati sullo stack rischiano di saturare lo spazio assegnato

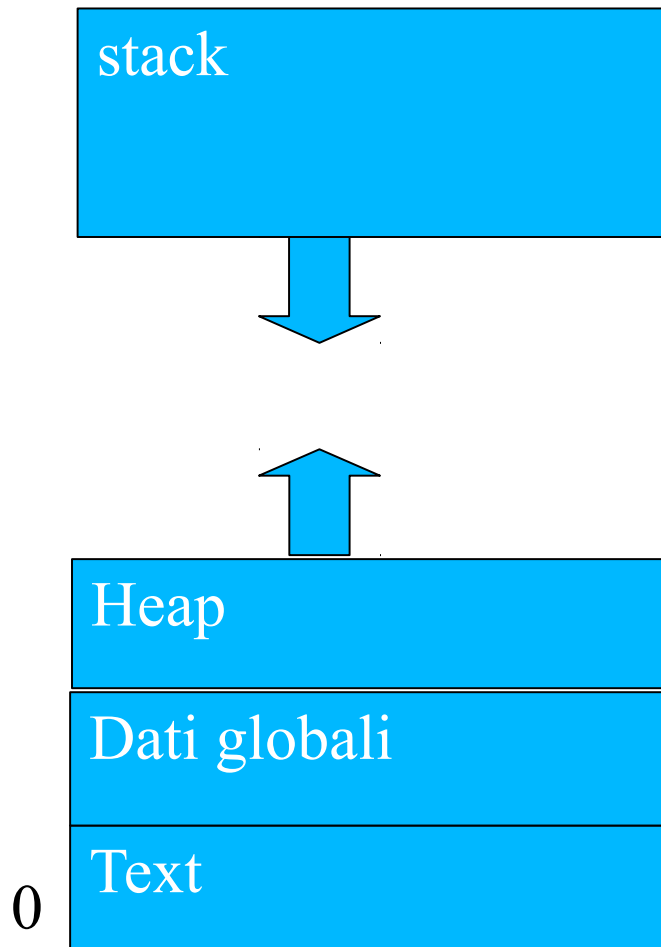


# Allocazione dinamica

- L'allocazione dinamica della memoria è possibile in C grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard ( `<stdlib.h>` )
  - L'allocazione avviene sullo heap ....

# Heap

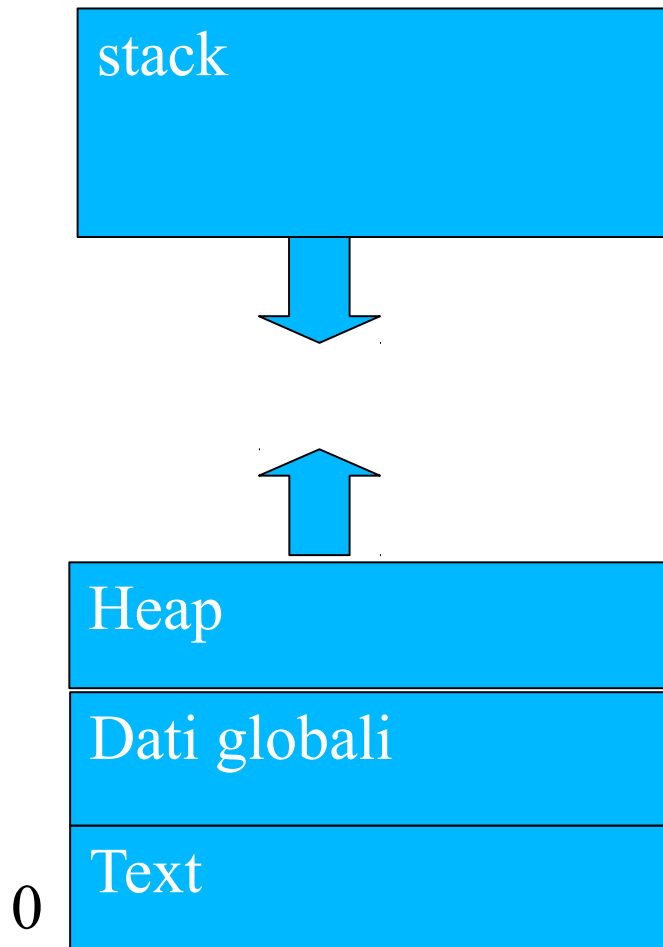
- Ricordiamo come vede la memoria un programma C in esecuzione



Variabili globali  
la dimensione di  
quest'area è fissata a  
tempo di compilazione  
e non si può espandere  
a run time (esecuzione)

# Heap

- Ricordiamo come vede la memoria un programma C in esecuzione



Variabili globali e locali allocate in fase di esecuzione.

L'area si espande verso gli indirizzi di valore maggiore in direzione dello stack

# Allocazione dinamica

- L'allocazione dinamica della memoria è possibile in C grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard ( `<stdlib.h>` )
  - L'allocazione avviene sullo heap ....
- Ci sono 3 funzioni che permettono la allocazione di nuovo spazio
  - `malloc()` , `calloc()` e `realloc()`
- Ed una funzione che permette di liberare lo spazio allocato
  - `free()`


# Allocazione dinamica: malloc

- La malloc() alloca memoria sullo heap e restituisce un puntatore alla memoria allocata
- Analizziamo il prototipo:

```
void* malloc(size_t size)
```



Tipo speciale che denota il puntatore generico, significa che  
Il puntatore restituito dalla malloc() può essere  
assegnato (e quindi convertito con casting automatico)  
a un puntatore di qualsiasi tipo



È una ridenominazione di un tipo intero.  
Si aspetta il numero di byte da allocare sullo  
heap.

# Allocazione dinamica: malloc

- La malloc() alloca memoria sullo heap e restituisce un puntatore alla memoria allocata
- Analizziamo il prototipo:

```
void* malloc(size_t size)
```

- Come si usa tipicamente:

```
/* ad esempio per allocare un double  
   array lungo N */  
  
double * p;  
  
p = malloc( sizeof(double) * N )
```

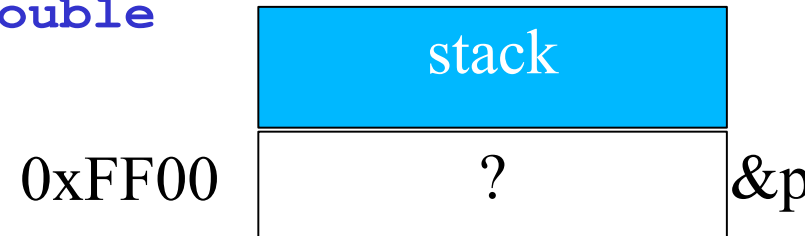
# Allocazione dinamica: malloc

- Come si usa tipicamente:

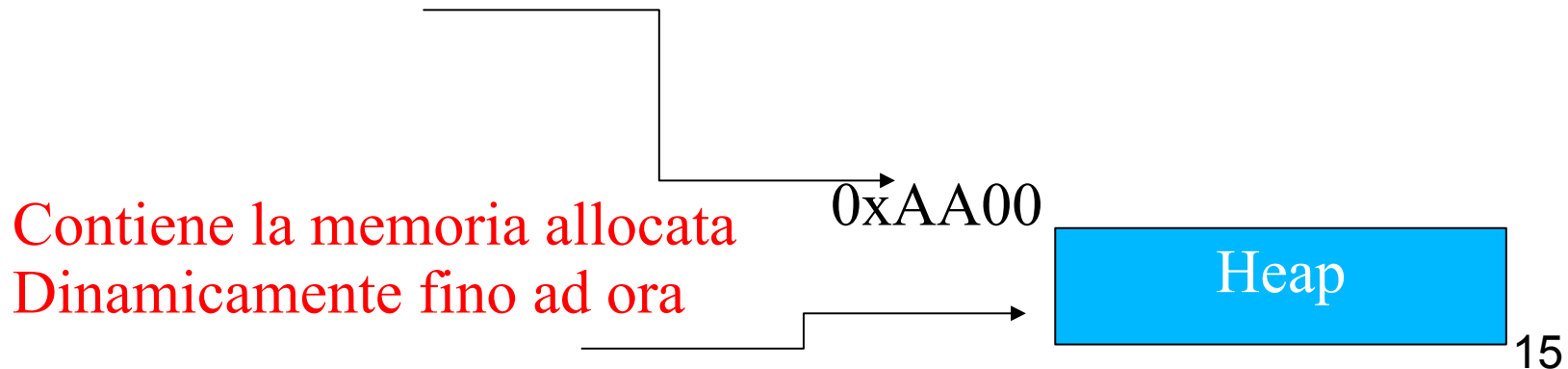
```
/* ad esempio per allocare un double  
   array lungo N */
```

```
double * p;
```

```
p = malloc( sizeof(double) * N )
```



Primo indirizzo disponibile



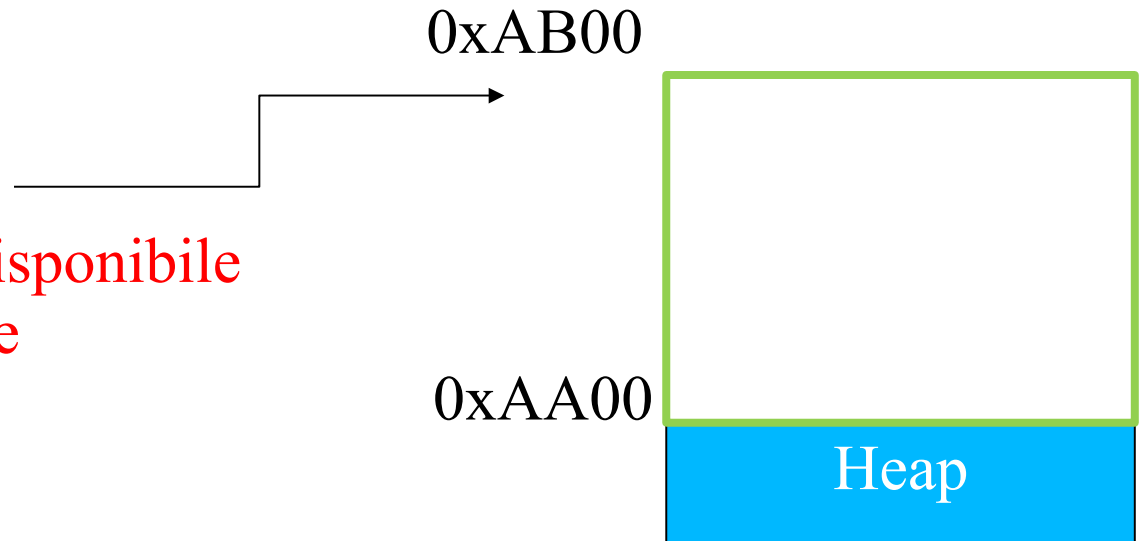
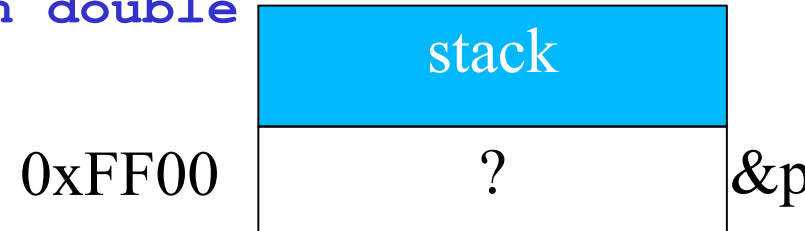
# Allocazione dinamica: malloc

- Come si usa tipicamente:

```
/* ad esempio per allocare un double  
   array lungo N */
```

```
double * p;
```

```
p = malloc( sizeof(double) * N )
```



Primo indirizzo disponibile  
Dopo l'allocazione



# Allocazione dinamica: malloc

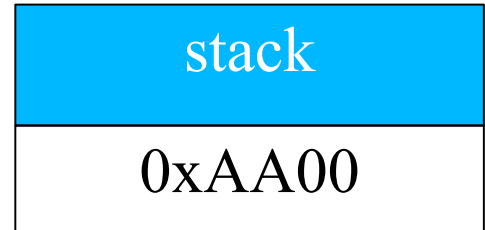
- Come si usa tipicamente:

```
/* ad esempio per allocare un double  
   array lungo N */
```

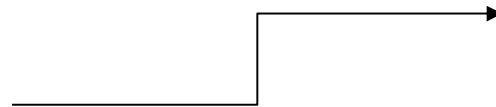
```
double * p;
```

```
p = malloc( sizeof(double) * N )
```

0xFF00



0xAB00



Primo indirizzo disponibile  
Dopo l'allocazione

0xAA00



# Allocazione dinamica: malloc

- Come si usa tipicamente:

```
/* ad esempio per allocare un double
   array lungo N */

double * p;

p = malloc( sizeof(double) * N )

/* da questo punto in poi usabile come un normale
array */

p[0] = 13.3;

p[1] = p[0] + 0.2;
```

- La memoria NON è inizializzata:
  - Quindi prima di utilizzarla va inizializzata adeguatamente!

# Allocazione dinamica: malloc

- E se non c'e' più spazio nello heap?
  - la **malloc()** ritorna NULL
  - esempio di funzione di libreria che può ritornare un errore davvero serio (*fatal*)
  - Per programmi non giocattolo deve essere sempre controllato prima di accedere!

# Allocazione dinamica: malloc

```
double * p;  
  
p = malloc( sizeof(double) * N );  
  
if ( p == NULL ) { /* gestione errore */}  
  
p[0] = 13.3;
```

- In questo caso l'unica gestione sensata è uscire con una indicazione dell'errore, il fallimento della malloc significa un problema davvero serio del vostro programma
- Cosa succede se elimino il controllo e vado ad accedere alla memoria dopo che la malloc è fallita ?

# Allocazione dinamica: malloc

```
/* per completezza, potete trovarlo scritto anche cosi' */  
double * p;  
if ( ( p = malloc( sizeof(double) * N ) ) == NULL ){  
    /* errore fatale */  
    fprintf(stderr, "Errore fatale!\n");  
    exit(EXIT_FAILURE);  
}  
p[0] = 13.3;
```

- Infatti un assegnamento è anche una espressione, che ha il valore assegnato, qua viene valutato e confrontato con NULL
- È il modo standard con cui si scrivono i controlli sulle funzioni di libreria nella programmazione avanzata C
- Se lo usate fate moltissima attenzione alle parentesi.....

# Ma torniamo al nostro esempio ...

- `malloc()` è utile se sappiamo la lunghezza dell'array
  - Quindi come VLA
- Però rispetto a VLA permette di eliminare il blocco esterno perchè l'array rimarrà sullo heap finchè non viene richiesto esplicitamente di rimuoverlo
  - Con una `free()` lo vediamo fra un attimo ....

```
/* nel nostro esempio malloc è utile solo se qualcuno ci dice la lunghezza massima
possibile della sequenza */

#include <stdio.h>

#include <stdlib.h>

typedef enum{false, true} boolean;

int main( void ) {

    int i = 0; int n;

    fine=false;

    double* value;

    printf("Inserisci la lunghezza massima:");

scanf("%d",&n);

value = malloc( sizeof(double) * n );

if (value == NULL) { /* gestione errore */}

do { printf("Inserisci il valore %d:",i);

    scanf("%lf",&value[i]);

    if ( value[i] == 0.0 ) fine=true;

    i++;

} while (( i < n )&& !fine) ;

/* calcolo sull'array value ... */

return 0; } /* fine main */
```

# Allocazione dinamica: free

- La `free()` libera memoria allocata con `malloc()`, `calloc()` o `realloc()`
- Analizziamo il prototipo:

```
void free(void* p) ;
```



Puntatore della memoria da deallocare  
(può essere un puntatore ad un tipo qualsiasi ...)



# Allocazione dinamica: free

- Situazione di partenza:

```
/* double  
   array lungo N */
```

```
double * p;
```

```
p = malloc( sizeof(double) * N );
```

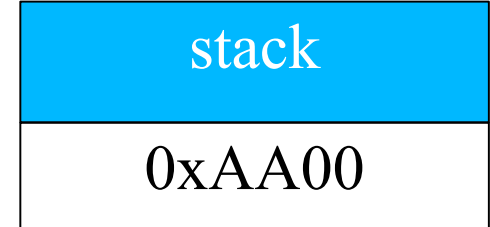
```
p[0] = 12;
```

....

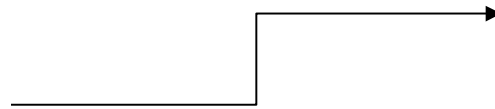
```
free (p) ;
```

Primo indirizzo disponibile  
Per l'allocazione

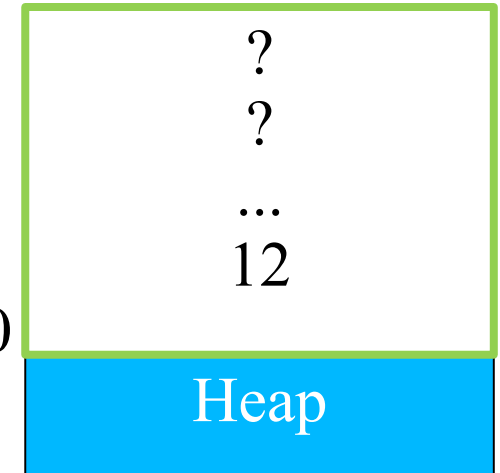
0xFF00



0xAB00



0xAA00



# Allocazione dinamica: free

- Situazione di partenza:

```
/* double  
   array lungo N */
```

```
double * p;
```

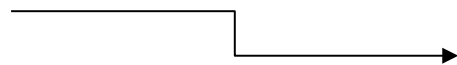
```
p = malloc( sizeof(double) * N );
```

```
p[0] = 12;
```

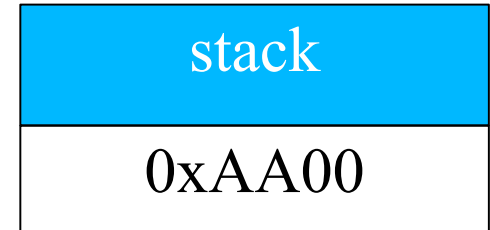
.....

```
free (p) ;
```

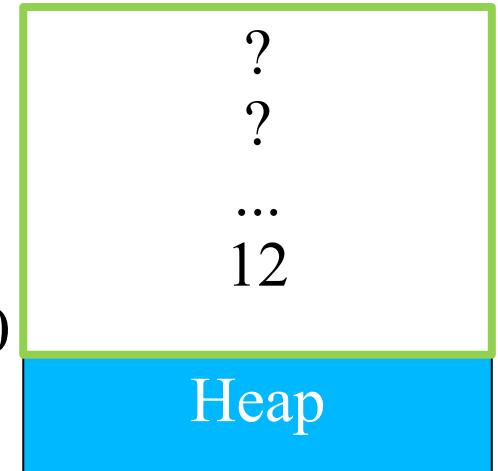
Primo indirizzo disponibile  
per l'allocazione



0xFF00



0xAB00



0xAA00

# Allocazione dinamica: free

- Situazione di partenza:

```
/* ad esempio per allocare un double  
array lungo N */
```

```
double * p;
```

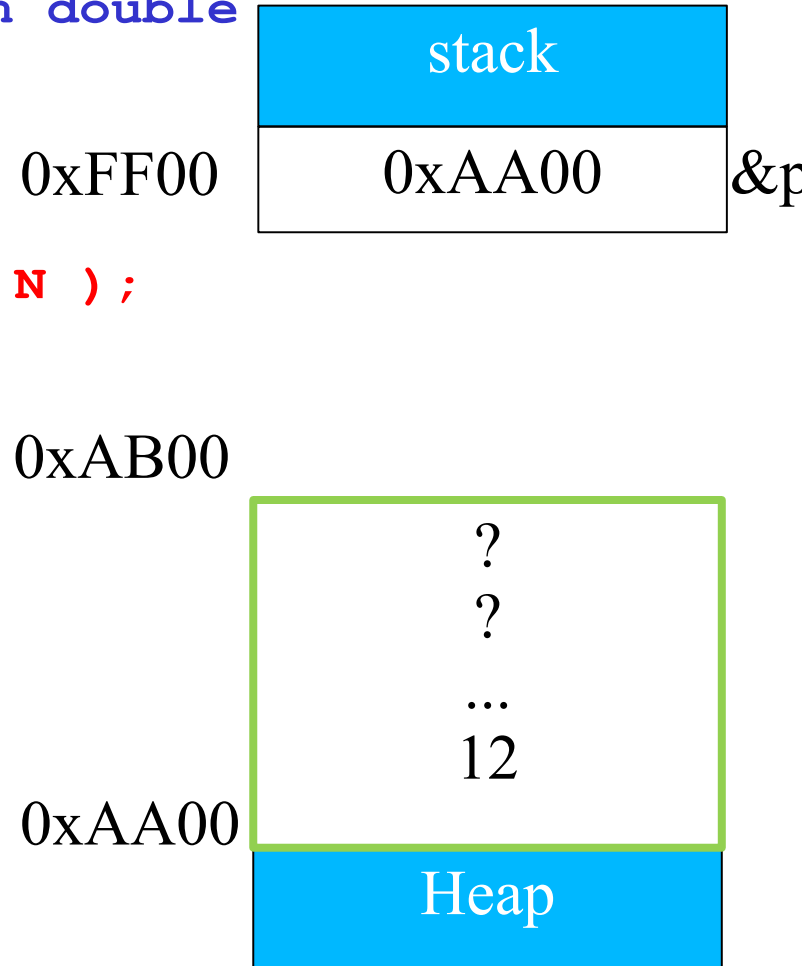
```
p = malloc( sizeof(double) * N );
```

```
p[0] = 12;
```

```
....
```

```
free(p);
```

Notate che il puntatore **p** non viene modificato, contiene ancora l'indirizzo e può essere usato!  
Il risultato però di tale accesso è indeterminato ....



# Allocazione dinamica: free

- Situazione di partenza:

```
/* ad esempio per allocare un double  
   array lungo N */
```

```
double * p;
```

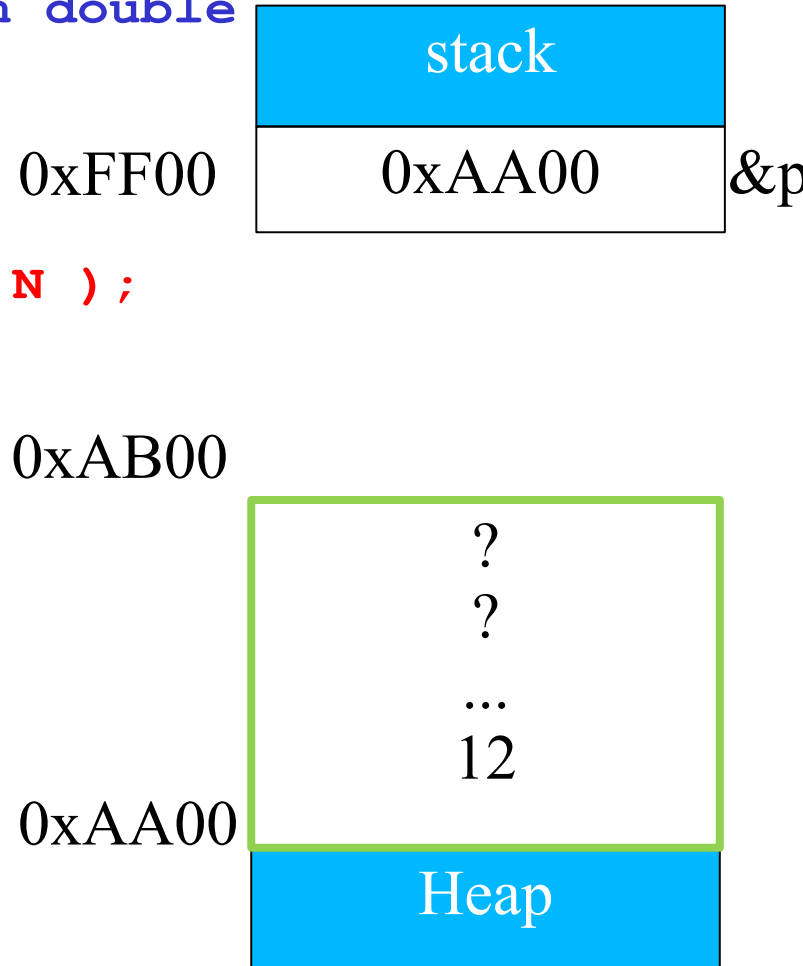
```
p = malloc( sizeof(double) * N );
```

```
p[0] = 12;
```

```
....
```

```
free(p);
```

Come fa la free a sapere  
Quanta memoria deve  
deallocare ?



# Allocazione dinamica: calloc

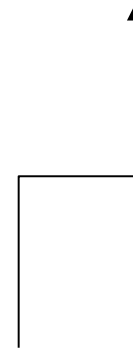
- La calloc() funziona come malloc, ma in più inizializza la memoria a 0
- Analizziamo il prototipo:

```
void* calloc(size_t nmemb, size_t size)
```

Numero degli elementi da allocare



Ampiezza in byte degli elementi da allocare



# Allocazione dinamica: calloc

```
/* esempio di uso */  
  
p = calloc( N, sizeof(double) );  
  
if ( p == NULL ) { /* gestione errore */}  
  
p[0] = 13.3;
```

# Allocazione dinamica: calloc

- Situazione di partenza:

```
/* uso calloc() per un double  
   array lungo N */
```

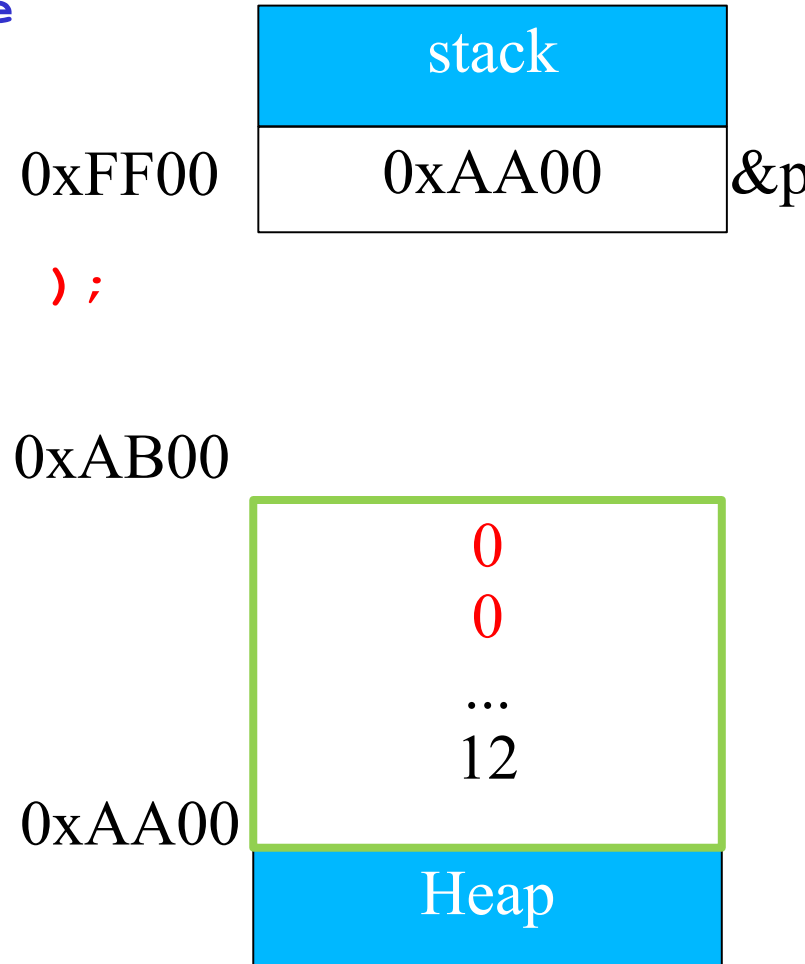
```
double * p;
```

```
p = calloc( N, sizeof(double) );
```

```
p[0] = 12;
```

```
.....
```

Considerando l'esempio precedente con malloc() l'unica differenza è che l'area di memoria è stata azzerata



# Allocazione dinamica: realloc

- La `realloc()` permette di ampliare o diminuire un'area di memoria già allocata ....
- Analizziamo il prototipo:

```
void* realloc(void* ptr, size_t size)
```

Puntatore all'area da riallocare  
(deve essere quello risultante  
Da una `malloc/calloc` precedente)

Ampiezza in byte dell'area dopo la riallocazione  
Generalmente maggiore, ma può esser anche minore o uguale



# Allocazione dinamica: realloc

- La `realloc()` permette di estendere o restringere un'area di memoria già allocata ....
- Analizziamo il prototipo:

```
void* realloc(void* ptr, size_t size)
```



Puntatore all'area riallocata.

Può essere diverso da quello precedente.

Ad esempio se chiediamo un'area maggiore e non c'è spazio contiguo disponibile tutta l'area viene ricopiata in uno spazio più grande dello heap (e viene liberata la memoria occupata precedentemente)

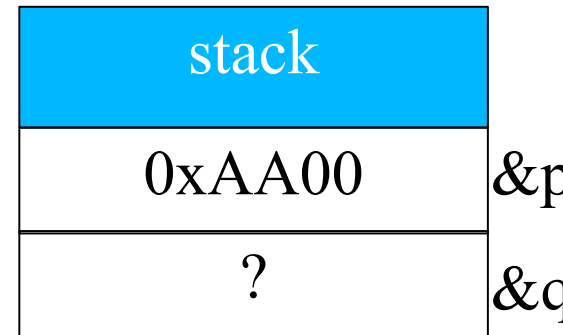
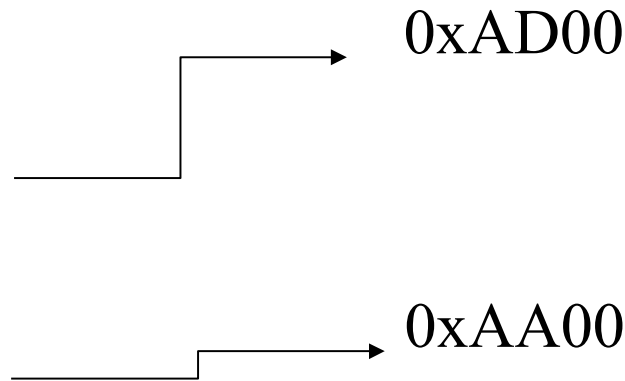
# Allocazione dinamica: realloc

- Ad esempio:

```
/* riallocare un double
   array N lungo 2 in un array
   M lungo 4 */
double * p, *q;
p = malloc( sizeof(double) * N );
p[0] = p[1] = 12;
q = realloc(p, M * sizeof(double));
q[2] = q[3] = 1;
```

Prima posizione libera

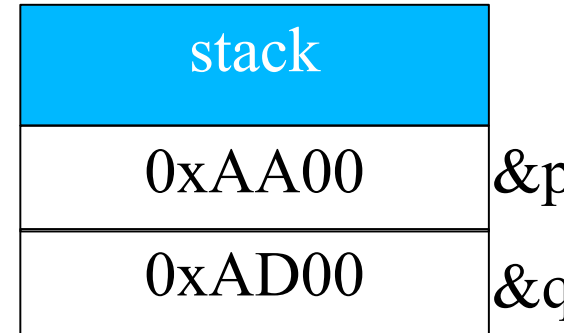
Prima posizione array



# Allocazione dinamica: realloc

- Ad esempio:

```
/* riallocare un double          0xFF00
   Array N lungo 2 in M lungo 4 */
double * p, *q;
p = malloc( sizeof(double) * N );
p[0] = p[1] = 12;
q = realloc(p, M * sizeof(double));
q[2] = q[3] = 1;
```



Nuova prima posizione array

0xAD00

0xAA00



# Allocazione dinamica: realloc

- Ad esempio:

```
/* riallocare un double  
   array lungo N(2) in M(4) */
```

```
double * p, *q;
```

```
p = malloc( sizeof(double) * N );
```

```
p[0] = p[1] = 12;
```

```
q = realloc(p, M * sizeof(double));
```

```
q[2] = q[3] = 1;
```

0xFF00

stack

0xAA00

0xAD00

&p

&q

0xAD00

1  
1  
12  
12

Altri dati in uso

0xAA00

12  
12

Heap

```
/* nel nostro esempio realloc è utile per riallocare la sequenza (anche se è molto
costoso)*/

#include <stdio.h>

#include <stdlib.h>

#define N 10

int main( void ) {

    int i = 0, lung; double* value;

    lung = N; fine=false;

value = malloc( sizeof(double) * lung );

do { printf("Inserisci il valore %d:",i);

    scanf("%lf",&value[i]);

    if ( value[i] == 0.0 ) fine=true;

    i++;

    if ( i == lung ) { lung += N;

        value =realloc(value,lung*sizeof(double) ); }

} while ( !fine ) ;

/* calcolo sull'array value ... */

return 0; } /* fine main */
```

# Bottom line su allocazione dinamica

- Sono operazioni molto costose e lente
  - Ogni volta chiamiamo in causa il sistema operativo
- Importantissimo ricordarsi di fare la **free ()**
  - I byte allocati rimangono lì per sempre, grosso problema per i programmi che girano a lungo
- Importantissimo evitare di uscire dalla memoria effettivamente allocata
  - I famosi *buffer overrun* .....
- Fondamentale il controllo degli errori
  - va fatto sempre, anche se per motivi di spazio a volte viene omessa nei lucidi
  - come in tutte le interazioni a basso livello con la macchina

# Bottom line su allocazione dinamica

- Da usare solo se serve veramente,
  - tutto quello che possiamo dobbiamo farlo con le variabili locali (veloci e deallocate automaticamente ...)
- Quali sono i casi in cui non la possiamo evitare?
  - Se abbiamo array di cui non conosciamo la lunghezza a tempo di compilazione
  - Se vogliamo allocare qualcosa in una funzione e fare in modo che sia ancora accessibile all'uscita della funzione stessa
  - Se utilizziamo strutture dati dinamiche tipo liste/alberi (le descriviamo in dettaglio più avanti)