La segmentazione della memoria
La memoria heap (*free store*)

Liceo G.B. Brocchi – Bassano del Grappa (VI)
Liceo Scientifico – opzione scienze applicate
Giovanni Mazzocchin

La memoria di un programma eseguibile

memory layout of a compiled program

memoria per il codice del programma, assemblato in linguaggio macchina

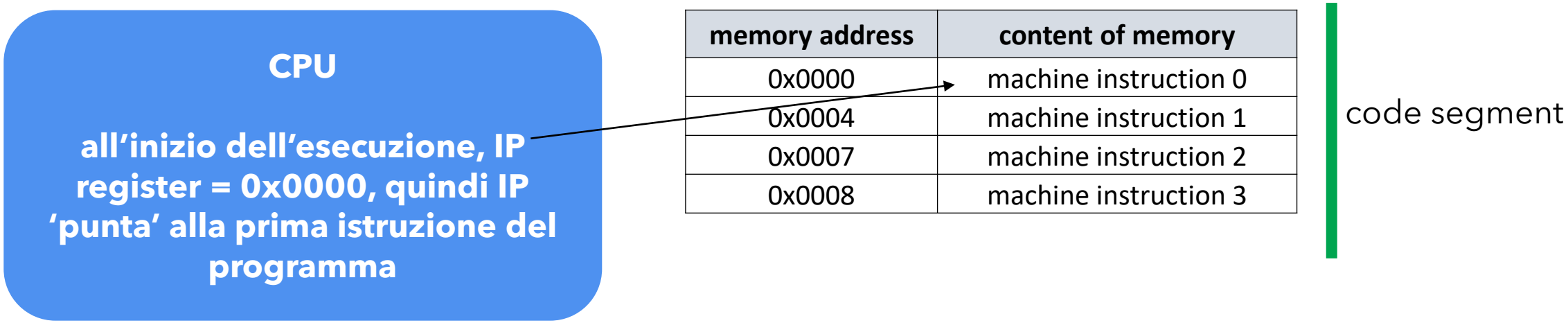
memoria per le variabili globali inizializzate (**data segment**) e non inizializzate (**bss segment**)

free store (aka ***heap***) se il programma ne ha bisogno, può utilizzarla

lo **stack**, per gestire i contesti delle funzioni

Il segmento del codice (*text segment*/*code segment*)

- Il processore legge le istruzioni da questo segmento, sequenzialmente. Chiaramente l'esecuzione non sarà sempre lineare, a causa delle istruzioni di salto
- Quando un programma viene eseguito, il registro della CPU **IP** (**Instruction Pointer**, detto anche **Program Counter**) viene impostato all'indirizzo della prima istruzione del segmento del codice



Il segmento del codice (*text segment*/*code segment*)

- Il **text segment** è un'area di memoria a **sola lettura** (**read-only**), in quanto contiene informazioni che non devono cambiare, a differenza delle variabili del programma
- Oltre a essere read-only, il *text segment* ha **dimensioni fisse**

I segmenti *data* e *bss*

- Il segmento **data** contiene le variabili globali e statiche **inizializzate**

- Il segmento **bss** (***b****lock* ***s****tarting* ***s****ymbol*) contiene le variabili globali e statiche **non inizializzate**

- Questo segmento può essere scritto, ma la sua dimensione non cambia, in quanto le variabili globali e statiche persistono indipendentemente dal contesto funzionale (a differenza delle variabili locali delle funzioni)

Il segmento *heap*

- Il segmento *heap* può essere controllato direttamente dal programmatore (poi vedremo come)
- Le dimensioni del segmento *heap* possono cambiare secondo le necessità del programma a *runtime*
- La memoria allocata sull'*heap* può quindi espandersi e contrarsi, su controllo del programmatore

Il segmento *stack*

- È il segmento che conosciamo già
- Viene utilizzato per memorizzare i contesti delle funzioni al momento della loro invocazione
- Ha dimensioni variabili, ad ogni invocazione di funzione cresce, ad ogni `return` decresce
- La chiamata di funzione è un salto ad un'istruzione contenuta ad un indirizzo del *text segment*: quindi in corrispondenza di una chiamata ad una funzione `func`, il registro **IP** assume il valore dell'indirizzo della prima istruzione della funzione `func`

Il segmento *stack*

- Se bastasse riassegnare l'IP sarebbe tutto troppo facile…
- Il programma deve 'sapere' a quale indirizzo andare al ritorno dalla funzione `func`: per questo motivo, sullo stack viene salvato anche l'*indirizzo di ritorno* da `func`, ossia il valore (indirizzo) da dare a **IP** quando `func` restituisce il controllo al chiamante con `return`
- Le variabili locali e l'indirizzo di ritorno sono memorizzate sullo stack all'interno di uno *stack frame*, o *activation record*

Il segmento *stack*

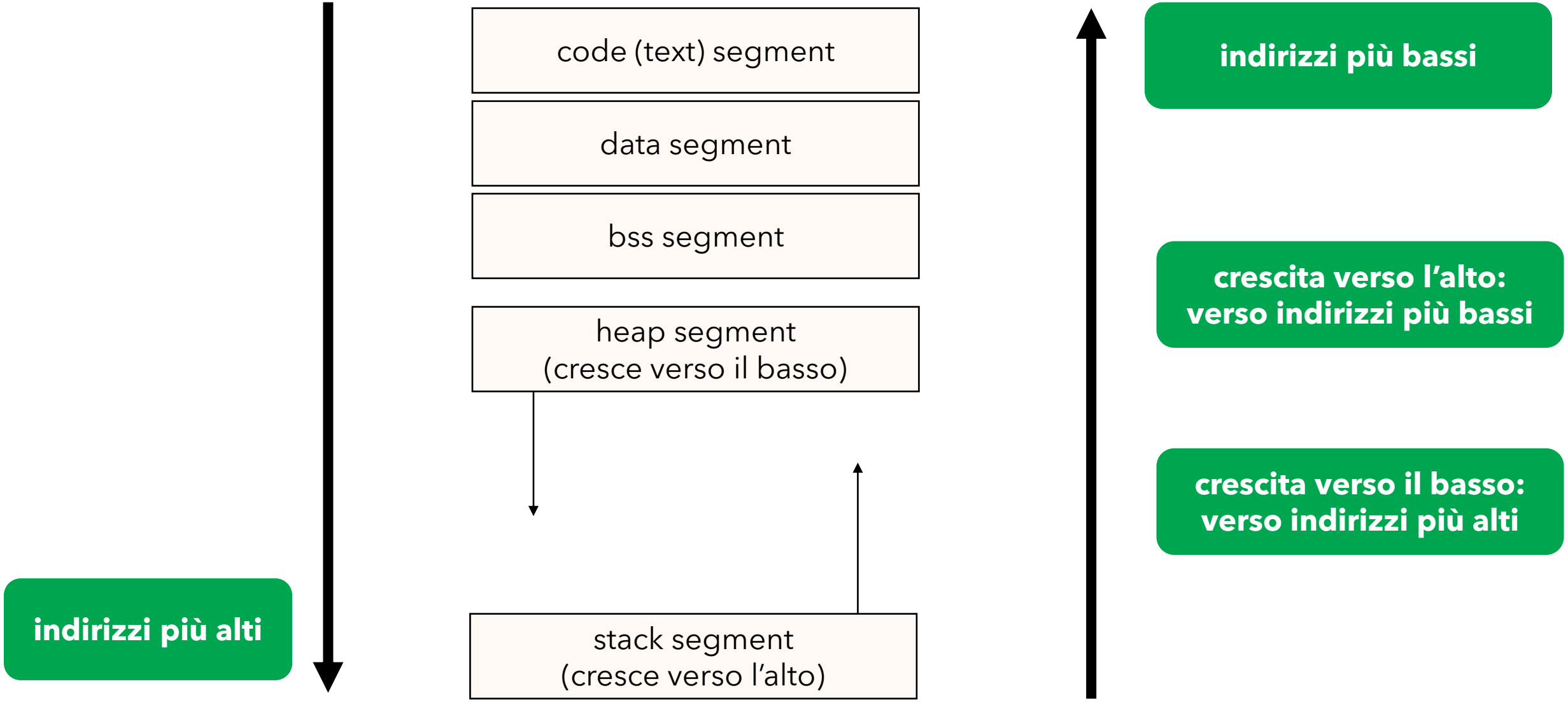
- Il registro **SP** (*stack pointer*) contiene l'indirizzo della cima dello stack, che cambia ad ogni *push/pop*

- Nella maggior parte delle architetture, l'espansione dello stack avviene **verso l'alto**, quindi **verso indirizzi di memoria più bassi**

- Il funzionamento LIFO dello stack potrebbe sembrare strano, ma viene utilizzato ovunque in Informatica **per memorizzare contesti con un livello di annidamento potenzialmente infinito**

- Una chiamata di funzione può essere vista come l'apertura di un nuovo contesto. Si può chiudere un contesto e tornare al contesto chiamante perché questo è stato precedentemente memorizzato sullo stack!

I segmenti di memoria in C



```c
#include <stdio.h>
#include <stdlib.h>

int global_var_uninitialized;  //uninitialized, global variable
int global_var_initialized = 9;

int g(){
   return 1;
}

void f() {
   int a_local_var_f;
   printf("f:a_local_var_f at address      %p\n\n", &a_local_var_f);
}
int main(int argc, char *argv[]) {
   int a_local_var_main = 4;
   int b_local_var_main = 12;
   int c_local_var_main = 3;

   static char static_var_uninitialized;
   static double static_var_initialized = 2.66f;

   int *heap_int_a = (int*) malloc(sizeof(int));
   int *heap_int_b = (int*) malloc(sizeof(int));
   int *heap_int_c = (int*) malloc(sizeof(int));
```

la funzione `malloc` della libreria standard C permette di allocare memoria sull'heap

```
    printf("TEXT SEGMENT********\n");
    printf("function g at    %p\n", g);
    printf("function f at    %p\n", f);
    printf("function main at %p\n\n", main);

    printf("DATA SEGMENT********\n");
    printf("global_var_initialized at %p\n", &global_var_initialized);
    printf("static_var_initialized at %p\n\n", &static_var_initialized);

    printf("BSS SEGMENT********\n");
    printf("global_var_unitialized at   %p\n", &global_var_uninitialized);
    printf("static_var_uninitialized at %p\n\n", &static_var_uninitialized);

    printf("HEAP SEGMENT********\n");
    printf("main:heap_int_a at address %p\n", heap_int_a);
    printf("main:heap_int_b at address %p\n", heap_int_b);
    printf("main:heap_int_b at address %p\n\n", heap_int_c);

    printf("STACK SEGMENT********\n");
    printf("main:a_local_var_main at address %p\n", &a_local_var_main);
    printf("main:b_local_var_main at address %p\n", &b_local_var_main);
    printf("main:c_local_var_main at address %p\n", &c_local_var_main);
    f();
}
```

```
TEXT SEGMENT********
function g at    0x555555555189
function f at    0x555555555198
function main at 0x5555555551c2

DATA SEGMENT********
global_var_initialized at 0x555555558010
static_var_initialized at 0x555555558018

BSS SEGMENT********
global_var_unitialized at   0x555555558024
static_var_uninitialized at 0x555555558028

HEAP SEGMENT********
main:heap_int_a at address 0x5555555592a0
main:heap_int_b at address 0x5555555592c0
main:heap_int_b at address 0x5555555592e0

STACK SEGMENT********
main:a_local_var_main at address 0x7fffffffdf84
main:b_local_var_main at address 0x7fffffffdf80
main:c_local_var_main at address 0x7fffffffdf7c
f:a_local_var_f at address       0x7fffffffdf4c
```

verso il basso: verso indirizzi più alti

notare che i 3 interi sull'heap non sono allocati su celle contigue

verso l'alto: verso indirizzi più bassi

notare che le variabili sono allocate contiguamente

Allocazione/deallocazione sull'heap

- Se per allocare memoria sui segmenti data, bss e stack è sufficiente dichiarare variabili, l'allocazione sull'heap richiede l'utilizzo di alcune funzioni della libreria standard C
- L'allocazione sull'heap viene effettuata tramite la funzione `malloc`, il cui prototipo (nell'header `stdlib.h`) è:

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

`malloc` accetta come unico argomento `size`, ossia il numero di byte da allocare sull'heap (il tipo `size_t` viene restituito dall'operatore `sizeof`; si tratta di un intero senza segno).
Se c'era disponibilità di memoria e l'allocazione è andata a buon fine, `malloc` restituisce un puntatore all'inizio dell'area di memoria allocata. `void*` si legge *puntatore a void*. È un modo per puntare a un'area di memoria il cui tipo è sconosciuto. `malloc` alloca solo byte senza avere *coscienza* dei tipi. Se l'allocazione non va a buon fine, `malloc` restituisce `NULL`, ossia il puntatore nullo

Allocazione/deallocazione sull'heap

- Anche la deallocazione di aree di memoria allocate sull'heap non è automatica. Il programmatore deve quindi occuparsi di liberare la memoria allocata precedentemente con una chiamata a `malloc`, invocando questa funzione (dichiarata sempre nell'header `stdlib.h`):

```
void free(void *ptr);
```

una chiamata a `free` libera la memoria allocata precedentemente con `malloc` e puntata da `ptr`

NB: liberare la memoria significa *renderla disponibile per eventuali allocazioni successive*

man malloc

void *malloc(size_t size);

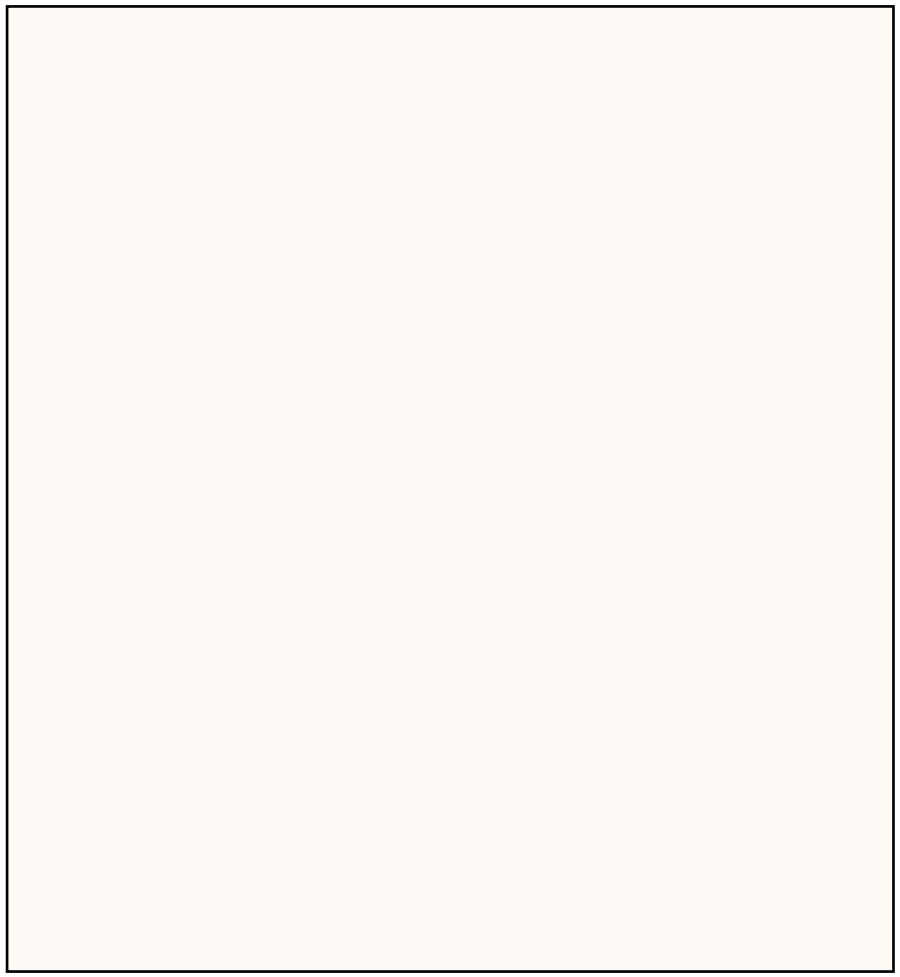
The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then malloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

void free(void *ptr);

The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc(), or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

Allocazione/deallocazione sull'heap

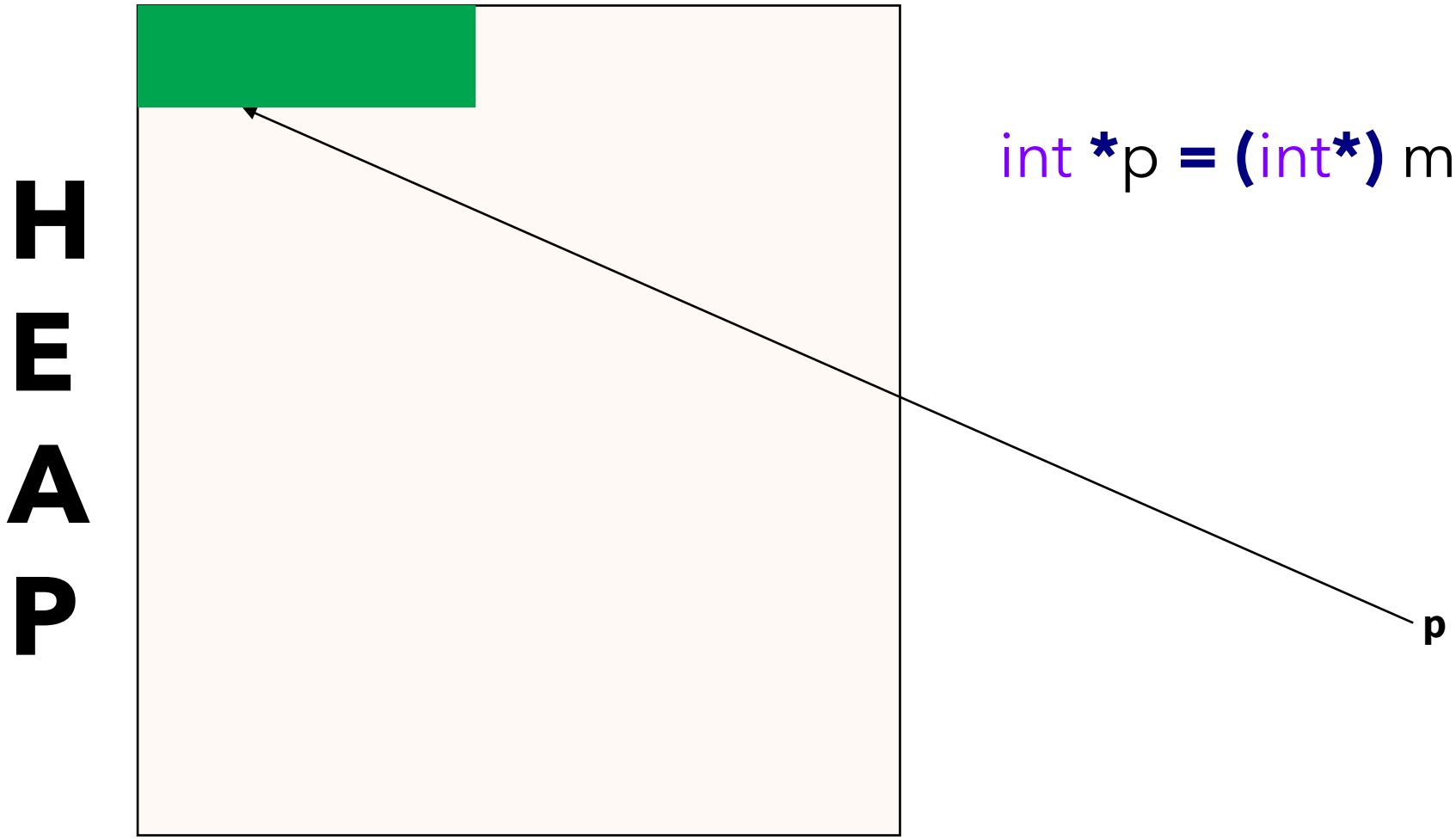
HEAP



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

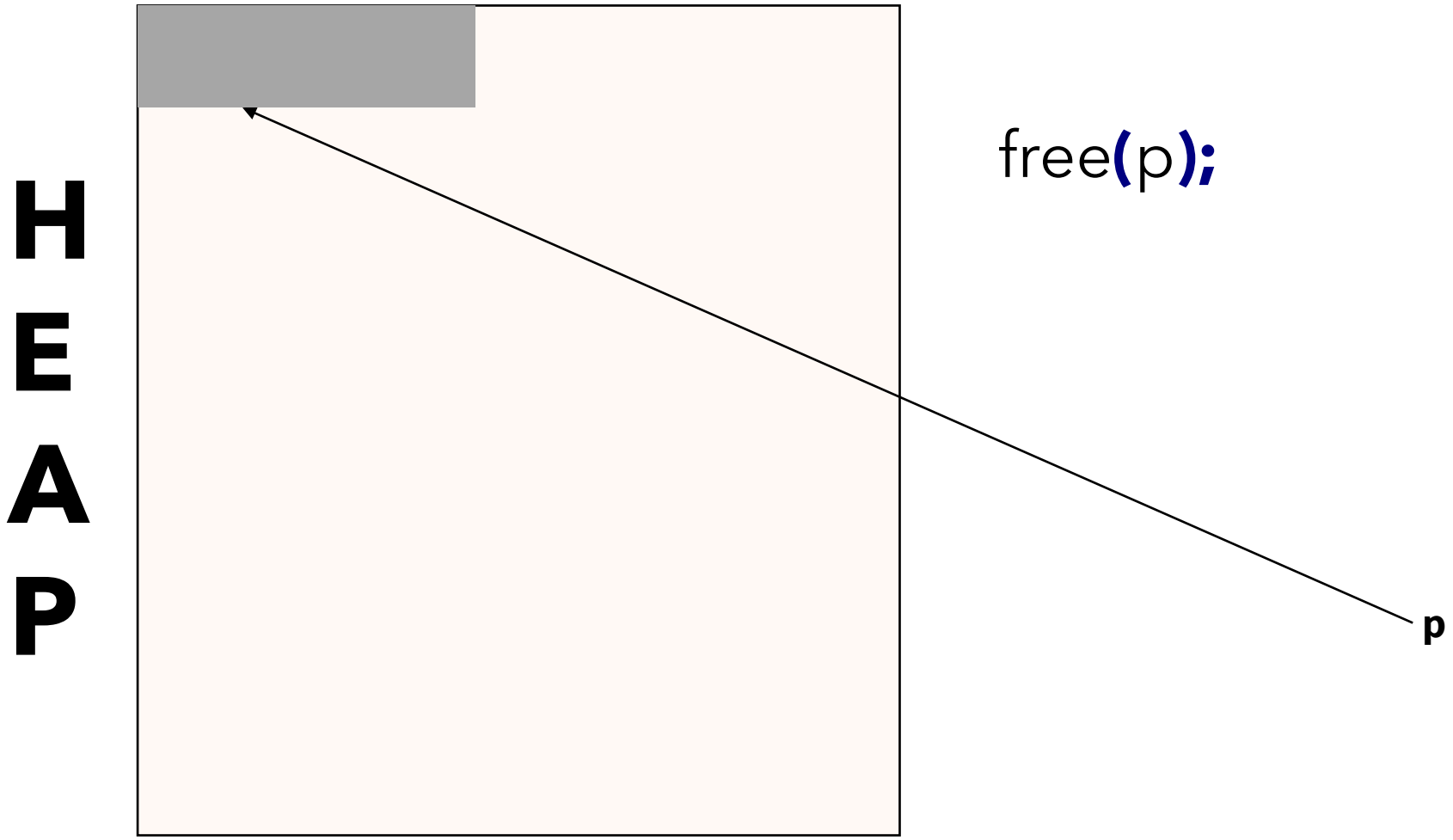
typecast (conversione di tipo) da `void*` (tipo restituito da `malloc`) a `int*` (puntatore a `int`)

Allocazione/deallocazione sull'heap




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

Allocazione/deallocazione sull'heap



Allocazione/deallocazione sull'heap

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   int *p1 = (int*) malloc(sizeof(int));
   if (p1) {
      *p1 = 128;
      printf("p1 points to heap address %p\n", p1);
      free(p1);
      int *p2 = (int*) malloc(sizeof(int));
      printf("p2 points to heap address %p\n", p2);
      printf("p1 points to heap address %p\n", p1);
      free(p1);
      int *p2 = (int*) malloc(sizeof(int));
      if (p2) {
         printf("p2 points to heap address %p\n", p2);
      }
      else {
         fprintf(stderr, "Error: could allocate on heap memory.\n");
      }
   }
   else {
      fprintf(stderr, "Error: could allocate on heap memory.\n");
   }
}
```

provate a compilare ed eseguire con e senza la chiamata a `free`, poi analizzate gli indirizzi

Esempio estremo di memory leak

```
int main(int argc, char *argv[]) {
  int *h;
  while (1) {
    /*allocates 128*sizeof(int) bytes on the heap
       at each iteration, without freeing them
    */
    h = (int*) malloc(sizeof(int) * 128);
  }
}
```

ovviamente questo programma crasha perché alloca memoria sull'heap senza mai liberarla. Sul mio computer crasha in pochi secondi.

```
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/c_lectures$ gcc -o heap heap.c
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/c_lectures$ ./heap
Killed
```

Memoria deallocata correttamente

```
int *h;
while (1) {
  /*allocates 128*sizeof(int) bytes on the heap
     at each iteration, without any free
  */
  h = (int*) malloc(sizeof(double) * 128);
  free(h);
}
```

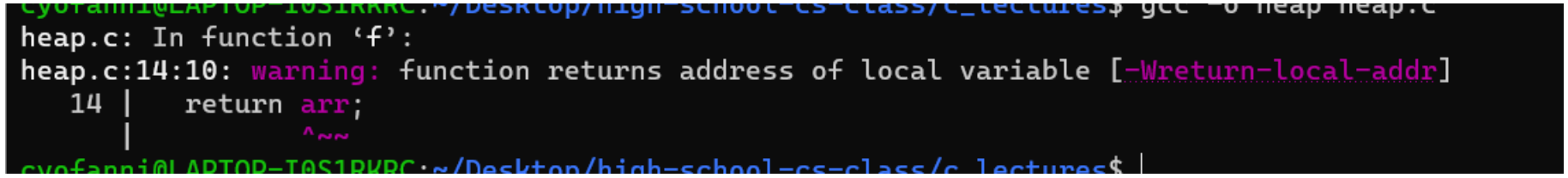
In questo esempio, la memoria allocata con `malloc` viene poi deallocata con `free`.
Questo programma non fa nulla di interessante, ma è un esempio utile

Funzioni che ritornano array

- Finora abbiamo mai scritto funzioni che restituiscono array?
- Se l'array è locale ad una funzione è allocato sullo stack, quindi verrebbe deallocato automaticamente quando lo stack frame della funzione viene poppato

```
int *f() {
  int arr[] = {6, 5, 2, 1};
  return arr;
}
```

abbiamo mai scritto una cosa del genere?
NO
Questo è un esempio di dangling pointer, il compilatore lo segnala con uno *warning*



Funzioni che ritornano array

```
int *ret_heap_array(int n) {
  int *ar = (int*) malloc(n * sizeof(int));

  return ar;
}
```

calloc e realloc

```
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

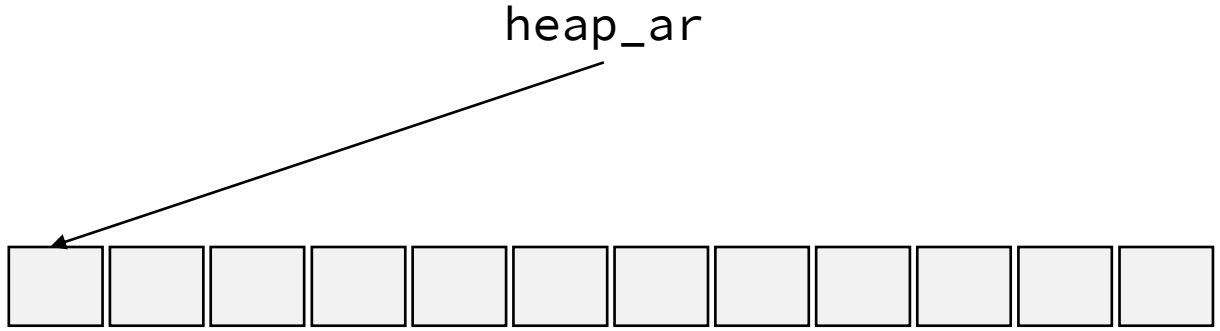
The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized.

calloc e realloc

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int heap_ar_len = 12;
  int *heap_ar = (int*) calloc(heap_ar_len, sizeof(int));
  for (int i = 0; i < heap_ar_len; i++) {
    heap_ar[i] = i;
  }
  for (int i = 0; i < heap_ar_len; i++) {
    printf("%d\t", heap_ar[i]);
  }
  putchar('\n');
  free(heap_ar);
}
```

allocazione sull'heap di un array di 12 interi, il primo byte del primo elemento dell'array è puntato da `heap_ar`



attenzione: le 12 caselle misurano `sizeof(int)` ciascuna, e non 1 byte soltanto

`calloc` e `realloc`

- Finora abbiamo visto:
  - gli array allocati sullo stack, di dimensione costante e stabilita a compile-time
  - gli array allocati dinamicamente sull'heap, la cui dimensione può essere anche una variabile (il cui valore può essere noto solo a runtime)
- Sarebbe molto utile avere a disposizione anche degli array che si allungano e si accorciano dinamicamente!
- Possiamo realizzarli con la funzione `realloc` della C standard library
- Immaginate un software che riceve dati in input da un sensore, da standard input, da una connessione di rete, da un database etc… e li inserisce in un array
- Ovviamente questo software non può accontentarsi di allocare un array di dimensioni fisse e immutabili per tutta l'esecuzione del programma

calloc e realloc

```
void unsafe_heap_write(size_t sz) {
  size_t heap_ar_len = sz;
  int * const heap_ar = (int * const) calloc(heap_ar_len, sizeof(int));
  int in;
  printf("%s ", "enter an integer: ");
  scanf("%d", &in);
  int item_cnt = 0;

  while (in != -1) {
    heap_ar[item_cnt] = in;
    printf("%s ", "enter an integer: ");
    scanf("%d", &in);
    item_cnt++;
  }
  for (int j = 0; j < item_cnt; j++) {
    printf("%d\t", heap_ar[j]);
  }
  putchar('\n');

  free(heap_ar);
}
```

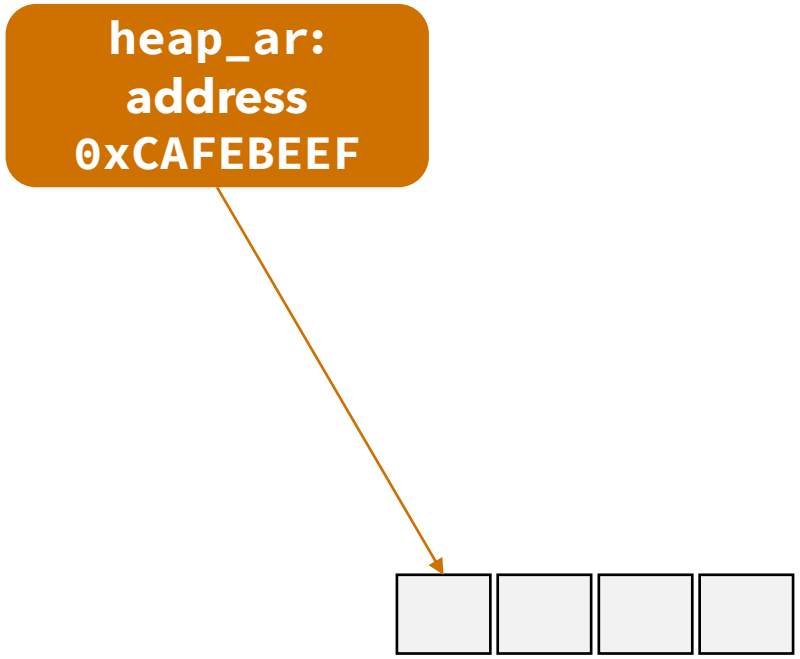
questa funzione legge interi da `stdin` e li scrive nell'array `heap_ar` allocato sull'heap, ma non effettua alcun controllo sul superamento dei limiti. Un programma del genere avrà un comportamento imprevedibile e sarà insicuro

calloc e realloc

```
void safe_heap_write(size_t sz) {
  size_t heap_ar_len = sz;
  int * heap_ar = (int*) calloc(heap_ar_len, sizeof(int));
  printf("heap_ar allocated with size %lu at address %p\n", heap_ar_len, heap_ar);

  int in;
  printf("%s ", "enter an integer: ");
  scanf("%d", &in);
  int item_index = 0;

  while (in != -1) {
    if (item_index + 1 > heap_ar_len - 1) {
      heap_ar_len *= 2;
      printf("heap_ar allocated with size %lu at address %p\n", heap_ar_len, heap_ar);
      heap_ar = (int*) realloc(heap_ar, heap_ar_len * sizeof(int));
    }
    heap_ar[item_index] = in;
    printf("%s ", "enter an integer: ");
    scanf("%d", &in);
    item_index++;
  }

  for (int j = 0; j < item_index; j++) {
    printf("%d\t", heap_ar[j]);
  }
  putchar('\n');

  free(heap_ar);
}
```

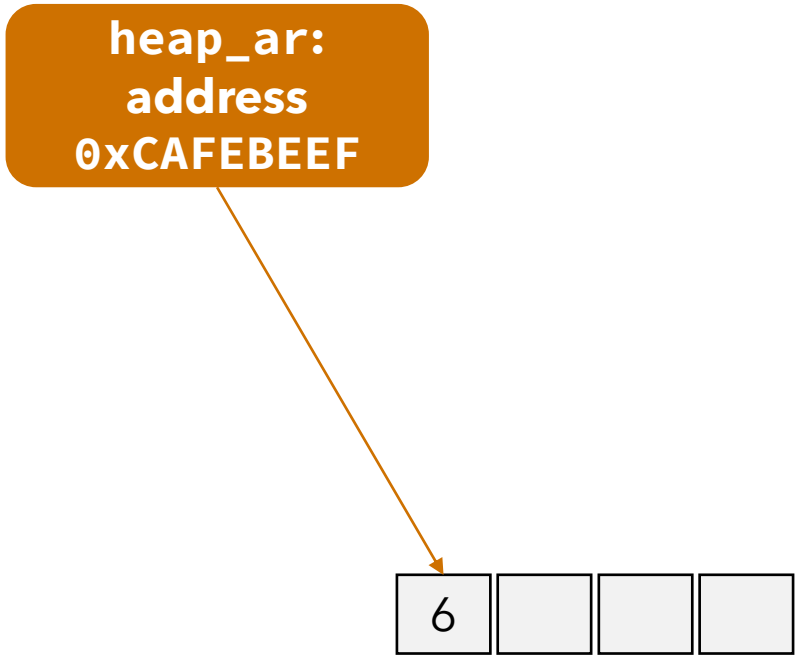
questa funzione invece effettua il controllo sul superamento del limite del buffer heap_ar allocato sull'heap.
Ad ogni superamento del limite, la dimensione del buffer viene raddoppiata.
Non era obbligatorio raddoppiarla, si può aumentare la dimensione in modo più complesso e più efficiente in termini di spazio

`calloc` e `realloc`



dopo l'invocazione di `calloc`, sull'heap c'è spazio per un array di 4 interi

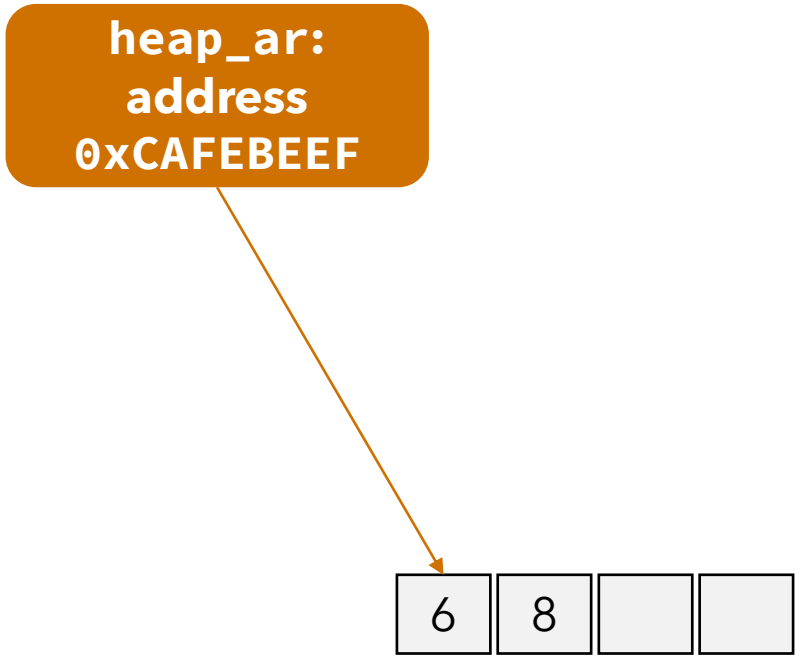
`calloc` e `realloc`



dopo l'invocazione di `calloc`, sull'heap c'è spazio per un array di 4 interi

il programma legge 6 da `stdin`

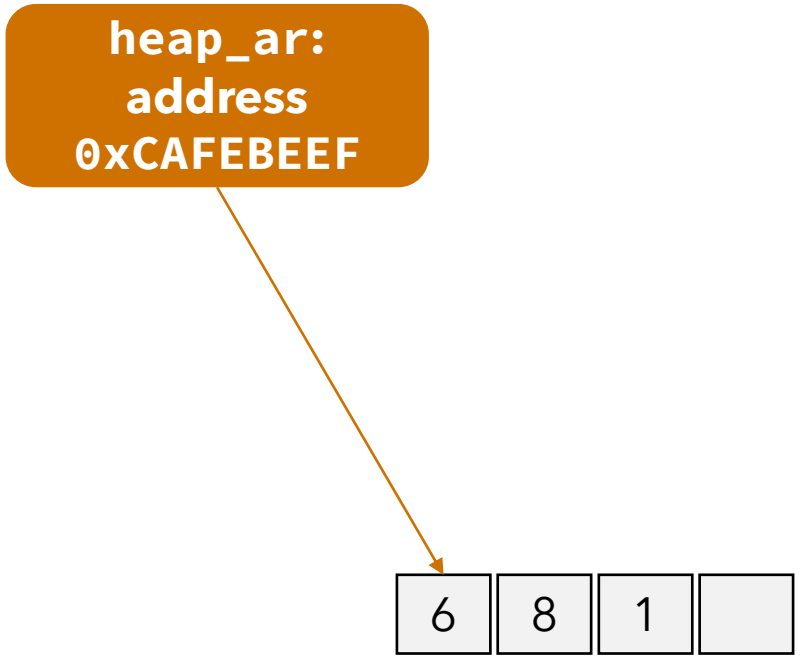
`calloc` e `realloc`



dopo l'invocazione di `calloc`, sull'heap c'è spazio per un array di 4 interi

il programma legge 8 da `stdin`

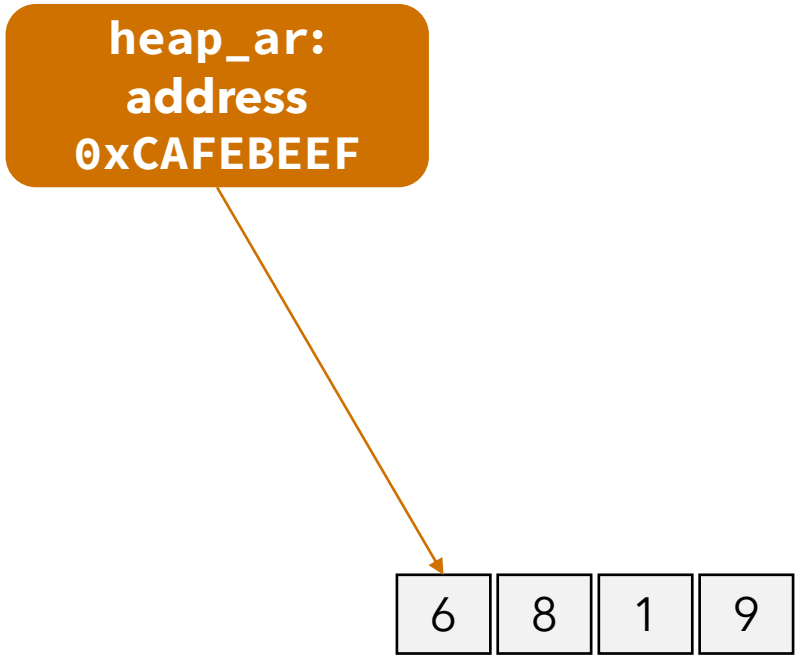
`calloc` e `realloc`



dopo l'invocazione di `calloc`, sull'heap c'è spazio per un array di 4 interi

il programma legge 1 da `stdin`

`calloc` e `realloc`



dopo l'invocazione di `calloc`, sull'heap c'è spazio per un array di 4 interi

il programma legge 9 da `stdin`

la dimensione dell'array va aumentata con `realloc`