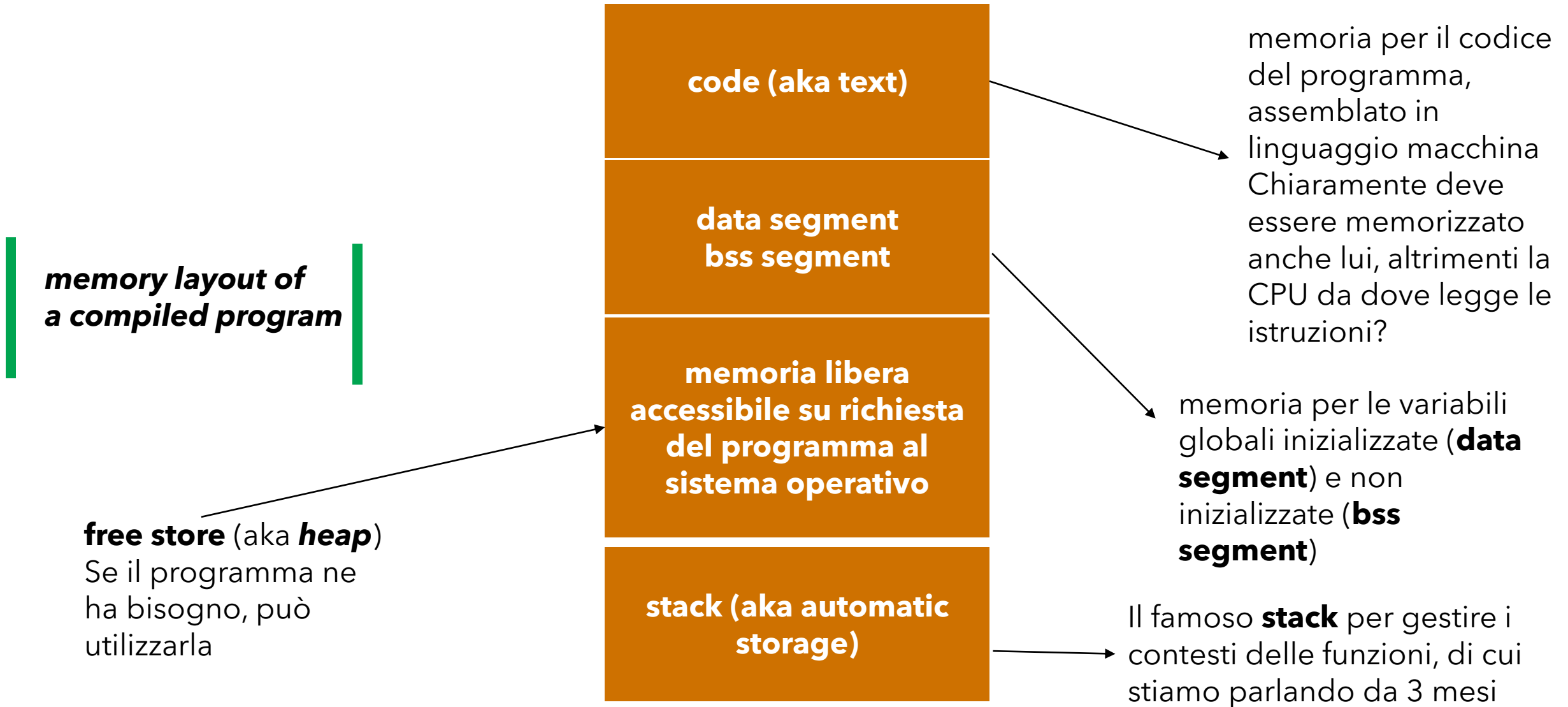


La segmentazione della memoria

La memoria *heap* (*free store*)

Liceo G.B. Brocchi
Classi terze Scientifico - opzione scienze applicate
Bassano del Grappa, Dicembre 2022
Prof. Giovanni Mazzocchin

La memoria di un programma eseguibile



Il segmento del codice (*text segment* o *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 l'**IP** (**Instruction Pointer**, detto anche **Program Counter**) viene impostato all'indirizzo della prima istruzione del segmento del codice

CPU

all'inizio dell'esecuzione, IP register = 0x0000, quindi IP 'punta' alla prima istruzione del programma

memory address	content of memory
0x0000	machine instruction 0
0x0004	machine instruction 1
0x0007	machine instruction 2
0x0008	machine instruction 3

code segment

Il segmento del codice (*text segment* o *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**

Il segmenti *data* e *bss*

- Il segmento ***data*** contiene le variabili globali e statiche **inizializzate**
- Il segmento ***bss*** (**block starting symbol**) 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* – esempio di esecuzione di un programma con chiamata di funzione senza parametri (NB: i valori degli indirizzi sono casuali)

func:

instruction 0 of func at text segment: 0x00FA
instruction 1 of func at text segment: 0x00FB
return at text segment: 0x00FD

instruction 0 at text segment: 0x0000
instruction 1 at text segment: 0x0004
instruction 2 at text segment: 0x0006
call func at text segment: 0x0008
instruction 4 at text segment: 0x0009
instruction 5 at text segment: 0x000B

IP

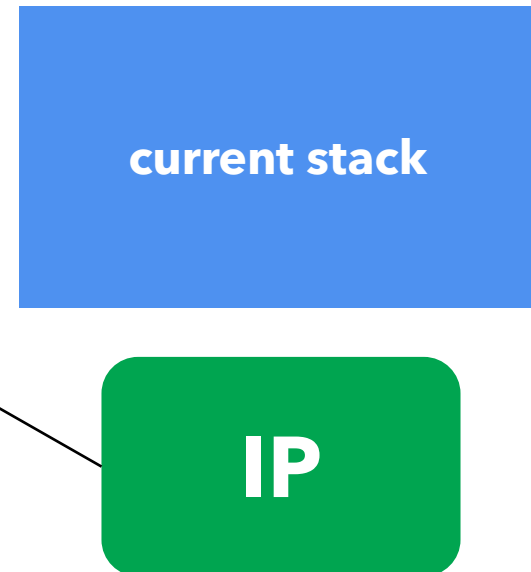
current stack

Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

```
instruction 0 of func  at text segment: 0x00FA  
instruction 1 of func  at text segment: 0x00FB  
return  at text segment: 0x00FD
```

```
instruction 0  at text segment: 0x0000  
instruction 1  at text segment: 0x0004  
instruction 2  at text segment: 0x0006  
call func    at text segment: 0x0008  
instruction 4  at text segment: 0x0009  
instruction 5  at text segment: 0x000B
```



Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA

instruction 1 of func at text segment: 0x00FB

return at text segment: 0x00FD

instruction 0 at text segment: 0x0000

instruction 1 at text segment: 0x0004

instruction 2 at text segment: 0x0006

call func at text segment: 0x0008

instruction 4 at text segment: 0x0009

instruction 5 at text segment: 0x000B



current stack

IP

Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA

instruction 1 of func at text segment: 0x00FB

return at text segment: 0x00FD

instruction 0 at text segment: 0x0000

instruction 1 at text segment: 0x0004

instruction 2 at text segment: 0x0006

call func at text segment: 0x0008

instruction 4 at text segment: 0x0009

instruction 5 at text segment: 0x000B



current stack

IP

Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA
instruction 1 of func at text segment: 0x00FB
return at text segment: 0x00FD

IP

A green rounded rectangle labeled 'IP' has an arrow pointing from its bottom-left corner to the first instruction of the 'func' block.

instruction 0 at text segment: 0x0000
instruction 1 at text segment: 0x0004
instruction 2 at text segment: 0x0006
call func at text segment: 0x0008
instruction 4 at text segment: 0x0009
instruction 5 at text segment: 0x000B

stack frame of func

Two blue rectangles are stacked vertically. The top rectangle is labeled 'stack frame of func' and the bottom rectangle is labeled 'current stack'.

current stack

Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA
instruction 1 of func at text segment: 0x00FB
return at text segment: 0x00FD

instruction 0 at text segment: 0x0000
instruction 1 at text segment: 0x0004
instruction 2 at text segment: 0x0006
call func at text segment: 0x0008
instruction 4 at text segment: 0x0009
instruction 5 at text segment: 0x000B

IP

stack frame of func

current stack

Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA
instruction 1 of func at text segment: 0x00FB
return at text segment: 0x00FD ←

instruction 0 at text segment: 0x0000
instruction 1 at text segment: 0x0004
instruction 2 at text segment: 0x0006
call func at text segment: 0x0008
instruction 4 at text segment: 0x0009
instruction 5 at text segment: 0x000B

IP

stack frame of
func

current stack

Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA

instruction 1 of func at text segment: 0x00FB

return at text segment: 0x00FD

instruction 0 at text segment: 0x0000

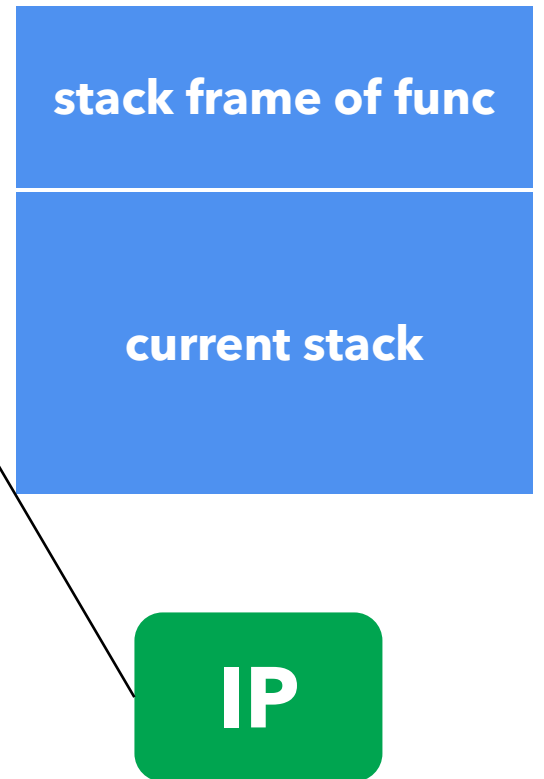
instruction 1 at text segment: 0x0004

instruction 2 at text segment: 0x0006

call func at text segment: 0x0008

instruction 4 at text segment: 0x0009

instruction 5 at text segment: 0x000B

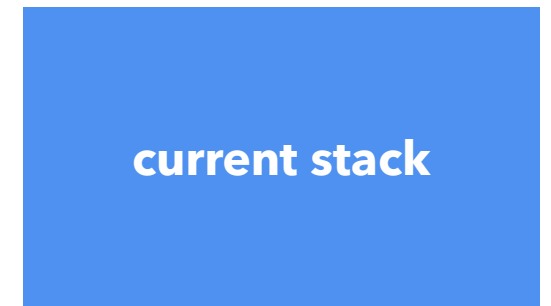


Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA
instruction 1 of func at text segment: 0x00FB
return at text segment: 0x00FD

instruction 0 at text segment: 0x0000
instruction 1 at text segment: 0x0004
instruction 2 at text segment: 0x0006
call func at text segment: 0x0008
instruction 4 at text segment: 0x0009
instruction 5 at text segment: 0x000B

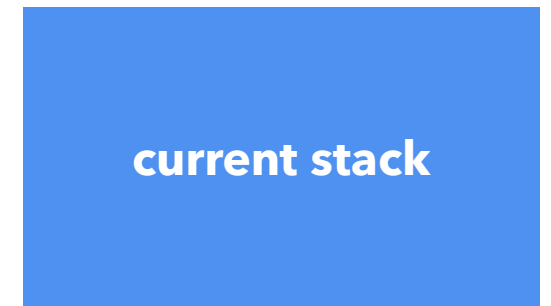


Il segmento *stack* – esempio di esecuzione di un programma con chiamata di funzione

func:

instruction 0 of func at text segment: 0x00FA
instruction 1 of func at text segment: 0x00FB
return at text segment: 0x00FD

instruction 0 at text segment: 0x0000
instruction 1 at text segment: 0x0004
instruction 2 at text segment: 0x0006
call func at text segment: 0x0008
instruction 4 at text segment: 0x0009
instruction 5 at text segment: 0x000B

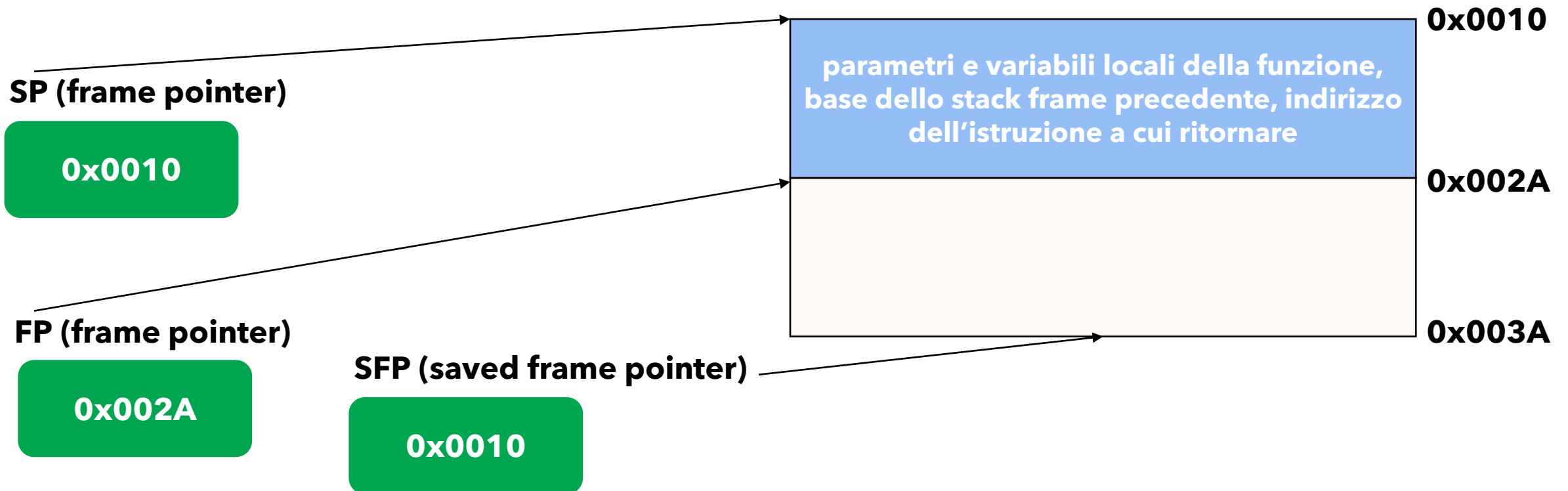


La cima dello stack e il registro *stack pointer*

- 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 registri *frame pointer*, *stack pointer* e *saved frame pointer*

Le scatole rappresentano gli *stack frame*. Gli indirizzi sono casuali, notare però che decrescono



I registri *frame pointer*, *stack pointer* e *saved frame pointer*

DOMANDE

L'indirizzo di ritorno all'istruzione successiva alla chiamata di funzione è memorizzato nello stack frame, ma è un indirizzo del segmento stack o di altri segmenti? Da quale registro viene utilizzato l'indirizzo di ritorno?

L'indirizzo della base dello stack frame precedente è memorizzato nello stack frame. E' un indirizzo del segmento stack?

Secondo voi, cosa può succedere se viene sovrascritto l'indirizzo di ritorno?

Fare pratica con *gdb*

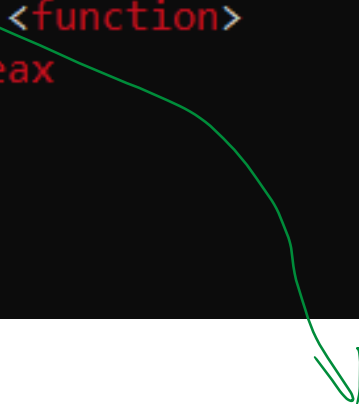
```
void function(int a, int b, int c, int d){  
    int flag;  
    char buffer[10];  
  
    flag = 31453;  
    buffer[0] = '5';  
}  
  
int main(){  
    function(1, 2, 3, 4);  
}
```

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

```
gcc -o stack_example -g -fno-stack-protector stack_example.c
```

Fare pratica con *gdb* – disassemblare il codice compilato

```
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/hacking$ gdb -q ./stack_example
Reading symbols from ./stack_example...
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000000014b <+0>:      endbr64
   0x000000000000014f <+4>:      push    %rbp
   0x0000000000000150 <+5>:      mov     %rsp,%rbp
   0x0000000000000153 <+8>:      mov     $0x4,%ecx
   0x0000000000000158 <+13>:     mov     $0x3,%edx
   0x000000000000015d <+18>:     mov     $0x2,%esi
   0x0000000000000162 <+23>:     mov     $0x1,%edi
   0x0000000000000167 <+28>:     call   0x1129 <function>
   0x000000000000016c <+33>:     mov     $0x0,%eax
   0x0000000000000171 <+38>:     pop     %rbp
   0x0000000000000172 <+39>:     ret
End of assembler dump.
(gdb)
```



vedere l'indirizzo iniziale di
function nella slide successiva

Fare pratica con *gdb* – disassemblare il codice compilato

```
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/hacking$ gdb -q ./stack_example
Reading symbols from ./stack_example...
(gdb) disassemble function
Dump of assembler code for function function:
    0x0000000000000129 <+0>:      endbr64
    0x000000000000012d <+4>:      push    %rbp
    0x000000000000012e <+5>:      mov     %rsp,%rbp
    0x0000000000000131 <+8>:      mov     %edi,-0x14(%rbp)
    0x0000000000000134 <+11>:     mov     %esi,-0x18(%rbp)
    0x0000000000000137 <+14>:     mov     %edx,-0x1c(%rbp)
    0x000000000000013a <+17>:     mov     %ecx,-0x20(%rbp)
    0x000000000000013d <+20>:     movl    $0x7add,-0x4(%rbp)
    0x0000000000000144 <+27>:     movb    $0x35,-0xe(%rbp)
    0x0000000000000148 <+31>:     nop
    0x0000000000000149 <+32>:     pop     %rbp
    0x000000000000014a <+33>:     ret
End of assembler dump.
(gdb) _
```


Fare pratica con *gdb* – settare *breakpoint*

```
End of assembly dump.  
(gdb) list  
1      void function(int a, int b, int c, int d){  
2          int flag;  
3          char buffer[10];  
4  
5          flag = 31453;  
6          buffer[0] = '5';  
7      }  
8  
9      int main(){  
10         function(1, 2, 3, 4);  
(gdb) break 6  
Breakpoint 1 at 0x1144: file stack_example.c, line 6.  
(gdb)
```

Fare pratica con *gdb* – esecuzione controllata

```
Reading symbols from ./stack_example...
(gdb) break 6
Breakpoint 1 at 0x1144: file stack_example.c, line 6.
(gdb) run
Starting program: /home/cyofanni/Desktop/high-school-cs-class/hacking/stack_example
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, function (a=1, b=2, c=3, d=4) at stack_example.c:6
6      buffer[0] = '5';
(gdb) disas main
Dump of assembler code for function main:
0x00005555555514b <+0>:      endbr64
0x00005555555514f <+4>:      push    %rbp
0x000055555555150 <+5>:      mov     %rsp,%rbp
0x000055555555153 <+8>:      mov     $0x4,%ecx
0x000055555555158 <+13>:     mov     $0x3,%edx
0x00005555555515d <+18>:     mov     $0x2,%esi
0x000055555555162 <+23>:     mov     $0x1,%edi
0x000055555555167 <+28>:     call   0x55555555129 <function>
0x00005555555516c <+33>:     mov     $0x0,%eax
0x000055555555171 <+38>:     pop     %rbp
0x000055555555172 <+39>:     ret
End of assembler dump.
(gdb) _
```

notare che gli indirizzi cambiano quando si esegue il programma. Questo perché il programma è stato caricato in memoria a partire da una locazione specifica

Fare pratica con *gdb* – esecuzione controllata

Dump of assembler code for function main:

```
0x00005555555514b <+0>:      endbr64
0x00005555555514f <+4>:      push   %rbp
0x000055555555150 <+5>:      mov    %rsp,%rbp
0x000055555555153 <+8>:      mov    $0x4,%ecx
0x000055555555158 <+13>:     mov    $0x3,%edx
0x00005555555515d <+18>:     mov    $0x2,%esi
0x000055555555162 <+23>:     mov    $0x1,%edi
0x000055555555167 <+28>:     call   0x55555555129 <function>
0x00005555555516c <+33>:     mov    $0x0,%eax
0x000055555555171 <+38>:     pop    %rbp
0x000055555555172 <+39>:     ret
```

End of assembler dump.

(gdb) disas function

Dump of assembler code for function function:

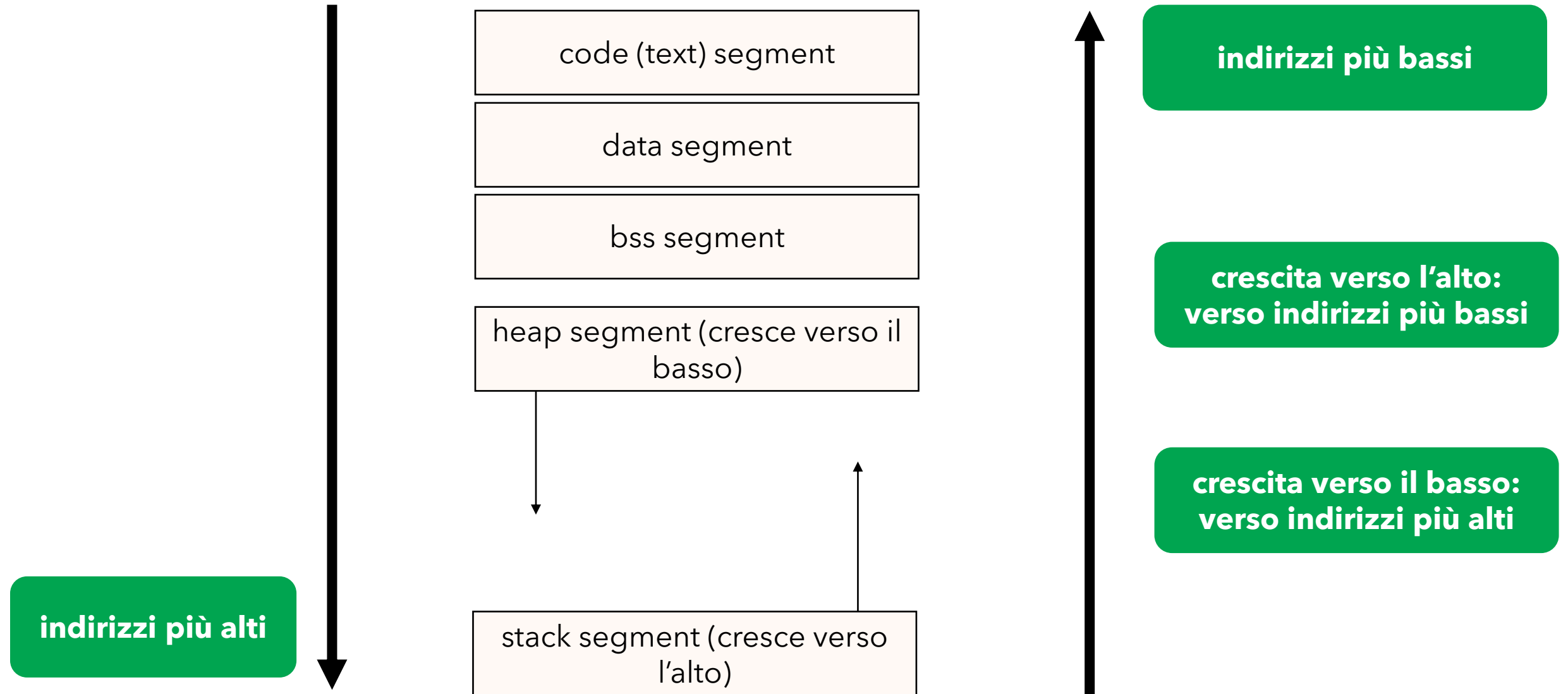
```
0x000055555555129 <+0>:      endbr64
0x00005555555512d <+4>:      push   %rbp
0x00005555555512e <+5>:      mov    %rsp,%rbp
0x000055555555131 <+8>:      mov    %edi,-0x14(%rbp)
0x000055555555134 <+11>:     mov    %esi,-0x18(%rbp)
0x000055555555137 <+14>:     mov    %edx,-0x1c(%rbp)
0x00005555555513a <+17>:     mov    %ecx,-0x20(%rbp)
0x00005555555513d <+20>:     movl   $0x7add,-0x4(%rbp)
=> 0x000055555555144 <+27>:     movb   $0x35,-0xe(%rbp)
0x000055555555148 <+31>:     nop
0x000055555555149 <+32>:     pop    %rbp
0x00005555555514a <+33>:     ret
```

function(1, 2, 3, 4);

il base pointer corrente viene pushato sullo stack

l'indirizzo di ritorno nel frame di function è 0x00005555555516c, ossia l'indirizzo dell'istruzione successiva alla chiamata

I segmenti di memoria in 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

una variabile *static* è indipendente dal contesto della funzione nella quale è definita; per ora prendetela come una specie di «variabile globale, ma definita all'interno di una funzione»

```

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_uninitialized 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_c 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      0x55555555189
function f at      0x55555555198
function main at 0x555555551c2
```

DATA SEGMENT*****

```
global_var_initialized at 0x555555558010
static_var_initialized at 0x555555558018
```

BSS SEGMENT*****

```
global_var_uninitialized 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
```

**indirizzi molto bassi che
crescono verso il basso,
ossia verso indirizzi più
alti.**

**Notare che i 3 interi sullo
heap non sono allocati su
celle contigue**

**indirizzi molto alti che
crescono verso l'alto, ossia
verso indirizzi più bassi
notare che le variabili sono
allocate contiguamente**

Allocazione/deallocazione sullo *heap*

- Se per allocare memoria sui segmenti *data*, *bss* e *stack* è sufficiente dichiarare variabili, l'allocazione sullo *heap* richiede l'utilizzo di alcune chiamate a funzioni della libreria standard C
- L'allocazione sullo *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 sullo *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 un puntatore nullo

Allocazione/deallocazione sullo *heap*

- Anche la deallocazione di aree di memoria allocate sullo heap non è automatica, a differenza dello stack. 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);

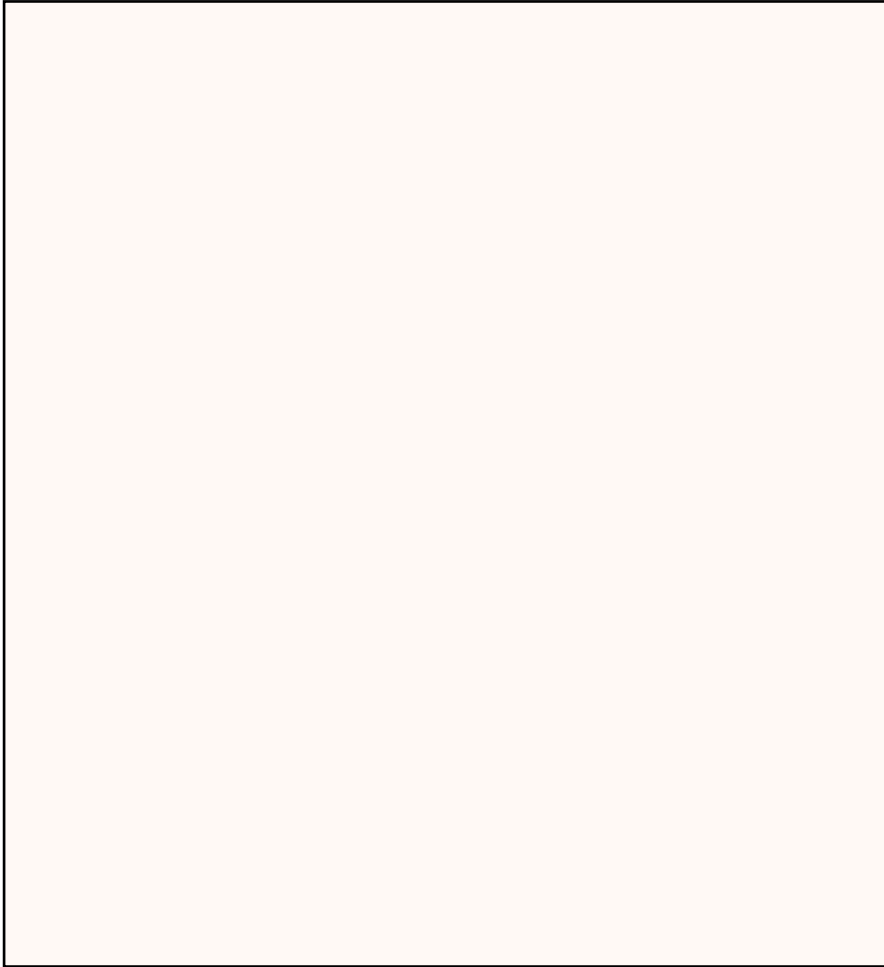
The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then malloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

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 sullo *heap*

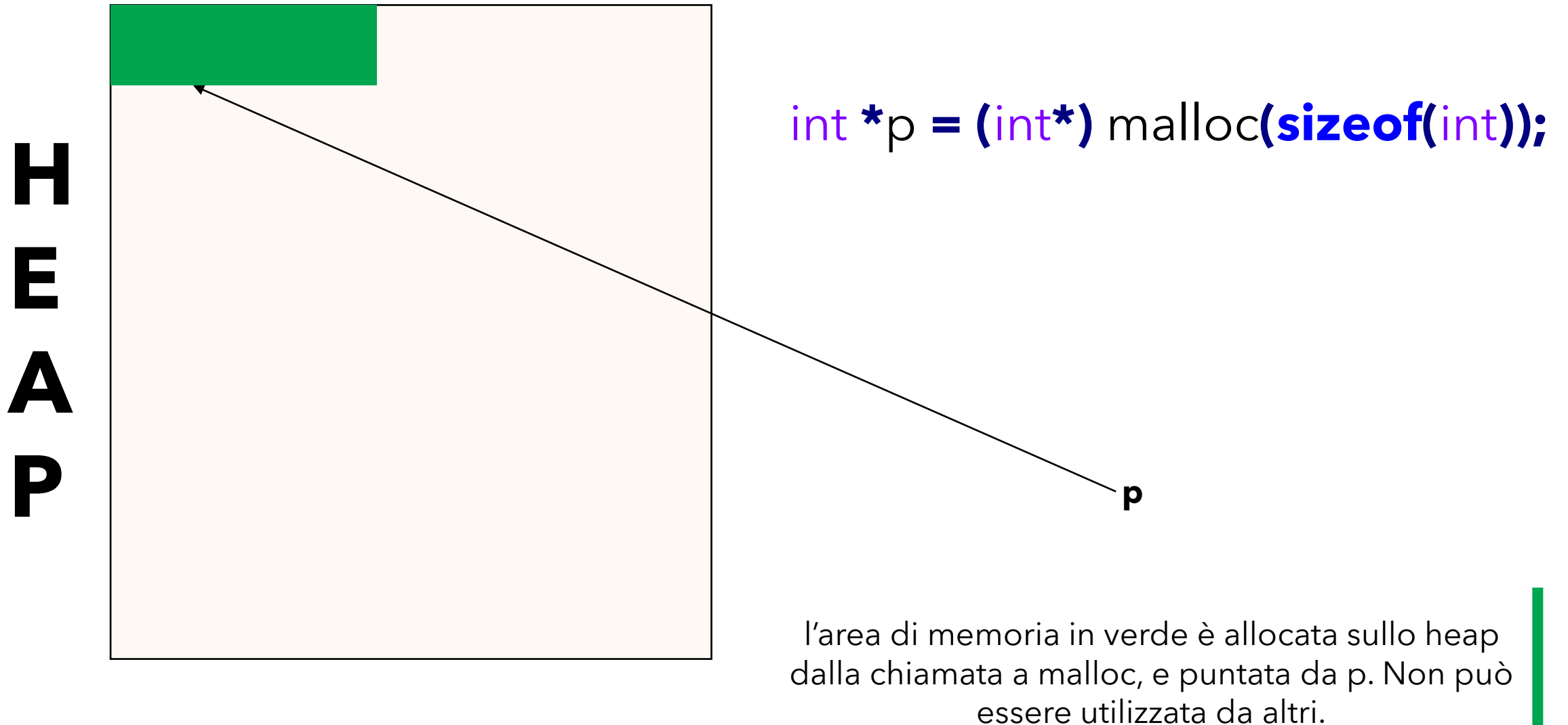
**H
E
A
P**



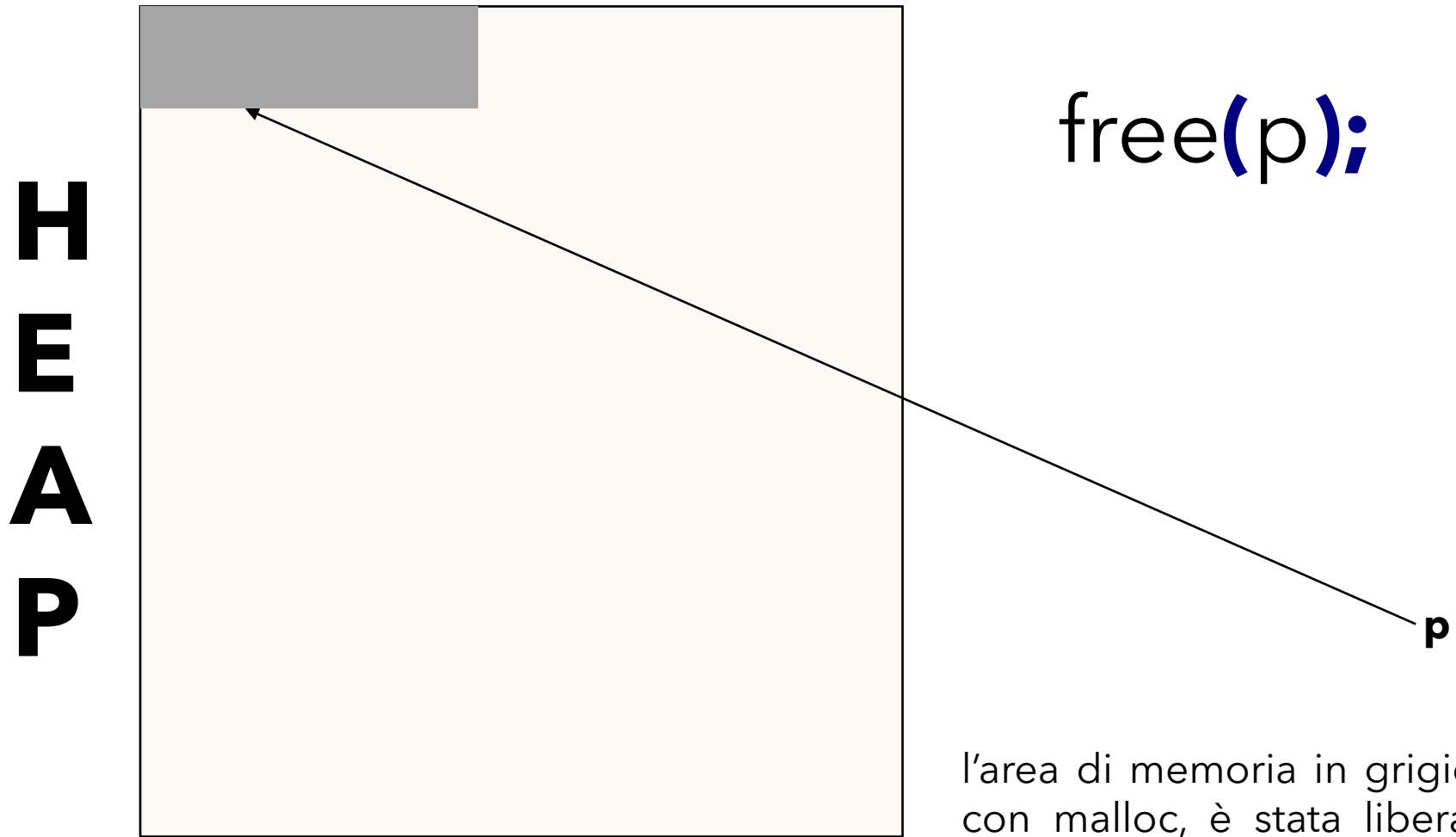
```
int *p = (int*) malloc(sizeof(int));
```

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

Allocazione/deallocazione sullo *heap*



Allocazione/deallocazione sullo *heap*



l'area di memoria in grigio, che era stata allocata con malloc, è stata liberata da free, e quindi è disponibile per altri utilizzi

Allocazione/deallocazione sullo *heap*

```
#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 free, poi
analizzate gli indirizzi**

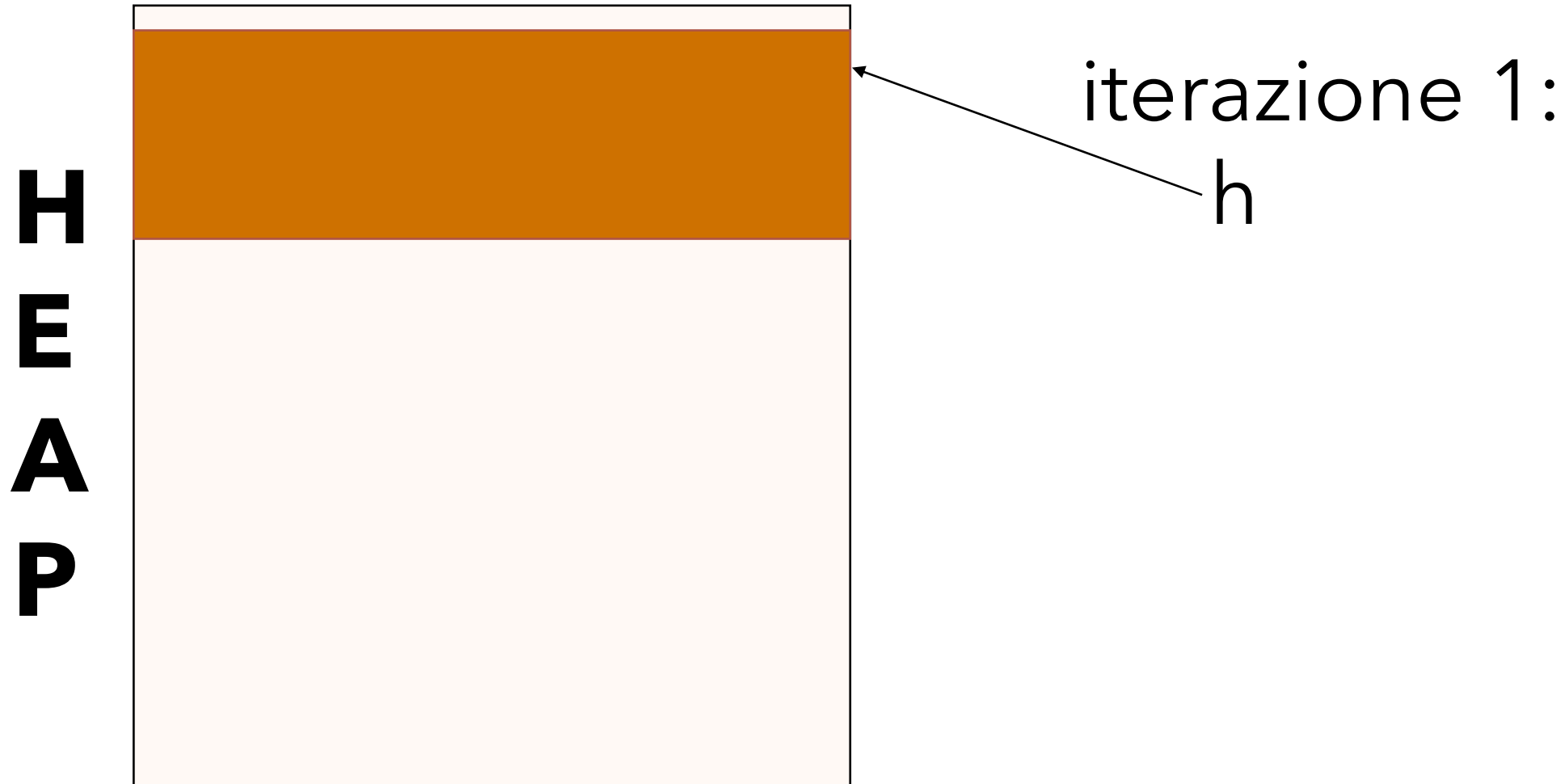
Esempio estremo di *memory leak*: memoria allocata e mai liberata in un ciclo infinito

```
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);  
    }  
}
```

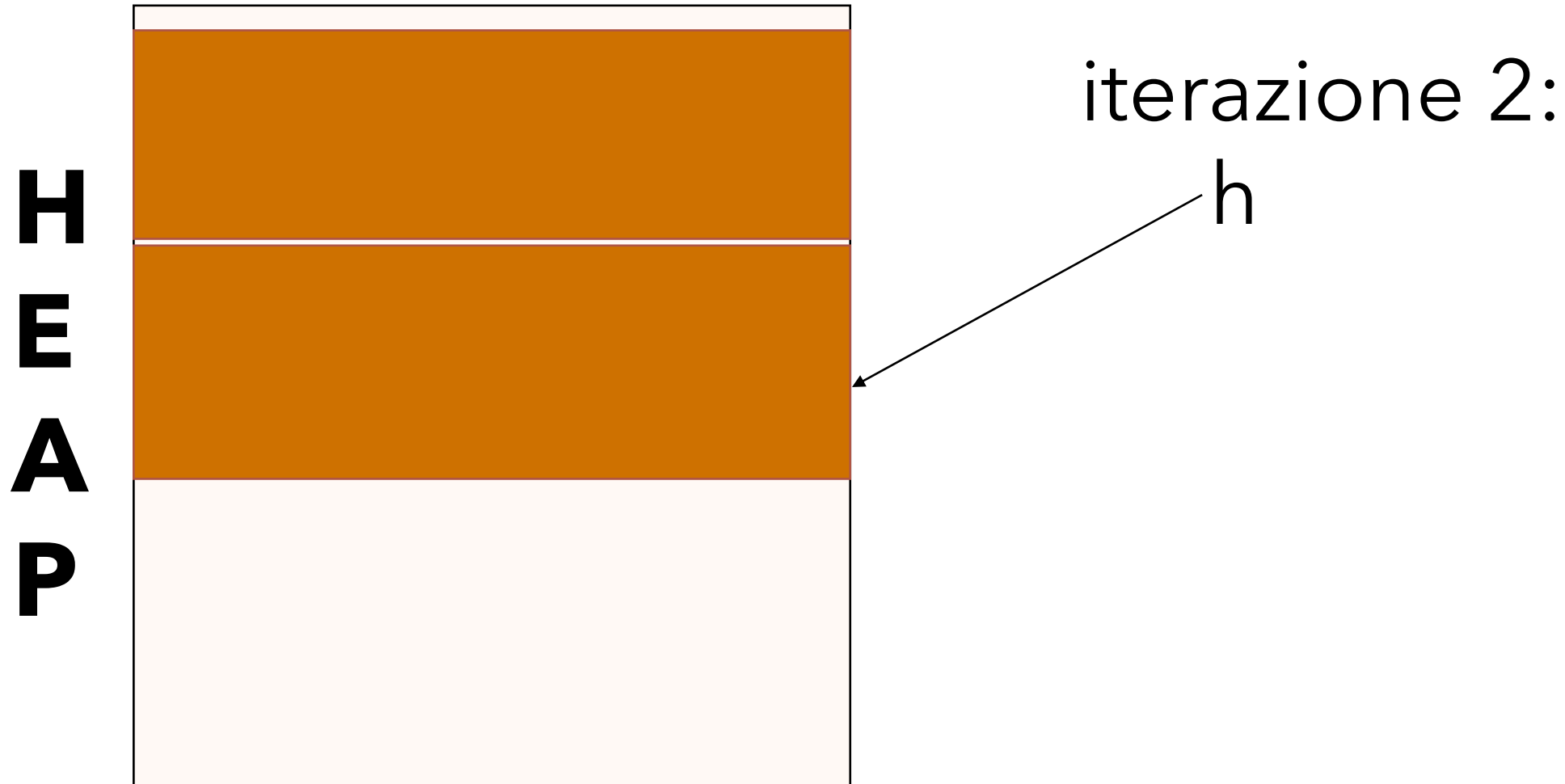
Ovviamente questo programma crasha perché alloca memoria sullo heap senza mai liberarla. Sul mio computer crasha in pochi secondi.

```
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
```

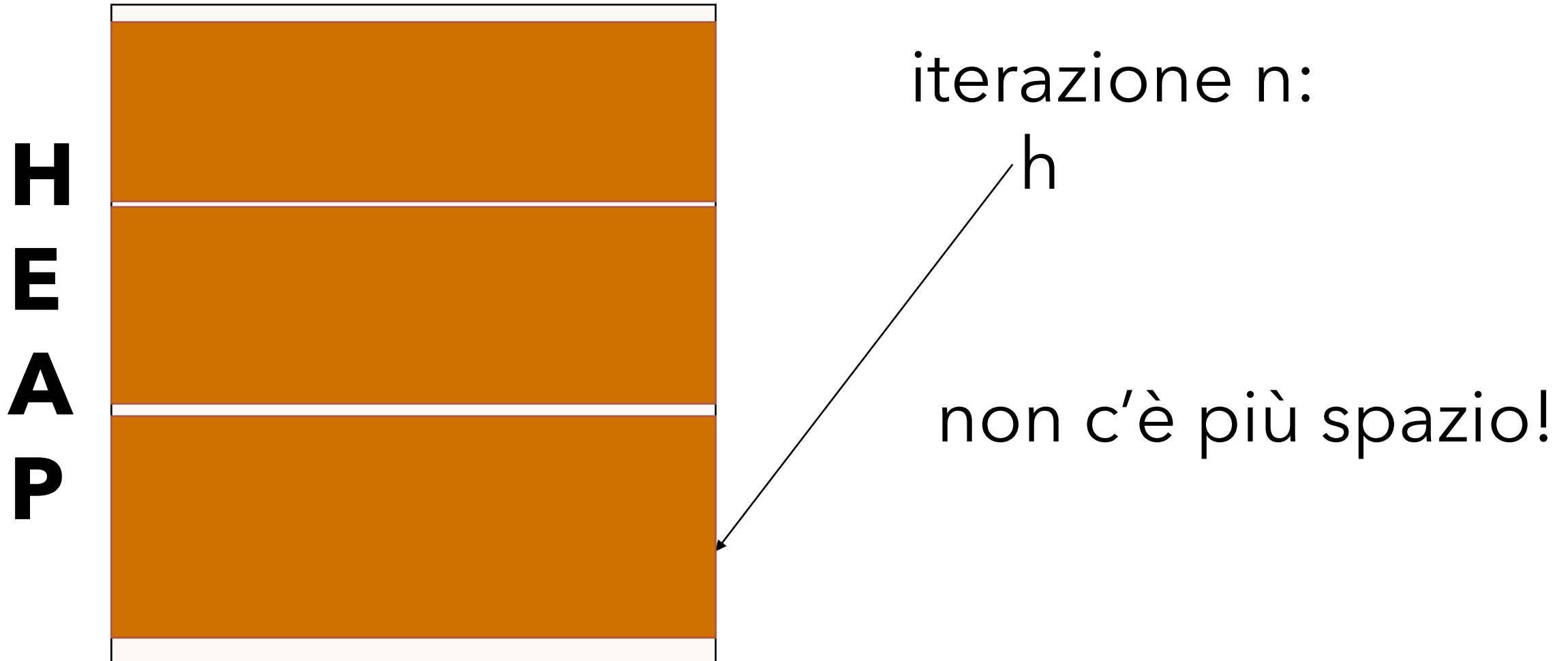
Esempio estremo di *memory leak*: memoria allocata e mai liberata all'infinito



Esempio estremo di *memory leak*: memoria allocata e mai liberata all'infinito



Esempio estremo di *memory leak*: memoria allocata e mai liberata all'infinito



Memoria deallocata correttamente, il programma non crasha

```
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 oltre a sprecare cicli di CPU, ma è un esempio utile.

Tipizzare la memoria allocata da malloc

- Cosa si può fare con un puntatore a void (void*)?
- Possiamo sicuramente usarlo per puntare alla memoria allocata da malloc, ma possiamo dereferenziarlo? NO

```
void *heap_bytes =  
malloc(N_HEAP_BYTES);  
*heap_bytes = 0x00;
```

void_pointer.c: In function 'main':

void_pointer.c:8:3: warning: dereferencing 'void *' pointer

```
8 | *heap_bytes = 0x00;  
  | ^~~~~~
```

void_pointer.c:8:15: error: invalid use of void expression

```
8 | *heap_bytes = 0x00;  
  | ^
```

Tipizzare la memoria allocata da malloc

- Usiamo l'aritmetica dei puntatori per verificare che heap_bytes punta al primo byte di una sequenza di byte
- È sufficiente leggere gli indirizzi di heap_bytes, heap_bytes + 1, heap_bytes + 2 e così via

```
#include <stdio.h>
#include <stdlib.h>
#define N_HEAP_BYTES 32
int main(int argc, char *argv[]) {
    void *heap_bytes = malloc(N_HEAP_BYTES);

    for (size_t i = 0; i < N_HEAP_BYTES; ++i) {
        printf("%p\t", heap_bytes + i);
    }
    putchar('\n');
}
```

```
0x5555555592a0  0x5555555592a1
0x5555555592a2  0x5555555592a3
0x5555555592a4  0x5555555592a5
0x5555555592a6  0x5555555592a7
0x5555555592a8
.....
```

La differenza tra gli indirizzi è sempre 1. Quindi è proprio vero un puntatore a void* punta ad una serie di byte

Tipizzare la memoria allocata da malloc

- Allochiamo un array di double sullo heap
- Dobbiamo fare in modo che il puntatore restituito da malloc punti alla memoria allocata, vista però come sequenza di double
- Se allochiamo 32 byte e li vediamo come double, allora abbiamo allocato $32/\text{sizeof}(\text{double})$

```
void alloc_doubles_heap(double d) {  
    double * const heap_bytes = (double*) malloc(N_HEAP_BYTES);  
    for (int i = 0; i < N_HEAP_BYTES / sizeof(double); ++i) {  
        printf("%p\n", heap_bytes + i);  
    }  
}
```

0x5555555592a0 0x5555555592a8 0x5555555592b0 0x5555555592b8

gli indirizzi distano 8! Effettivamente, `sizeof(double)`: 8 sulla mia macchina

Tipizzare la memoria allocata da malloc

```
void alloc_doubles_heap(double d) {  
    double * const heap_bytes = (double*) malloc(N_HEAP_BYTES);  
    for (int i = 0; i < N_HEAP_BYTES / sizeof(double); ++i) {  
        printf("%p\n", heap_bytes + i);  
    }  
    for (size_t i = 0; i < N_HEAP_BYTES / sizeof(double); ++i) {  
        heap_bytes[i] = d;  
    }  
    for (size_t i = 0; i < N_HEAP_BYTES / sizeof(double); ++i) {  
        printf("%f ", heap_bytes[i]);  
    }  
    putchar('\n');  
  
    free(heap_bytes);  
}
```

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 un errore del genere?

NO

Questo è un esempio di dangling pointer, il compilatore lo segnala con uno *warning*

```
cyofanni@LAPTOP-10S1KKRC:~/Desktop/high-school-cs-class/c_lectures$ gcc -o heap heap.c  
heap.c: In function 'f':  
heap.c:14:10: warning: function returns address of local variable [-Wreturn-local-addr]  
   14 |     return arr;  
      |           ^~~  
cyofanni@LAPTOP-10S1KKRC:~/Desktop/high-school-cs-class/c_lectures$ |
```


Una funzione che ritorna un array allocato sullo heap

```
int *compute_fibonacci_numbers(int n) {  
    int *fib_array = (int*) malloc((n + 2) * sizeof(int));  
    fib_array[1] = 1;  
    for (int i = 2; i <= n; i++) {  
        fib_array[i] = fib_array[i - 1] + fib_array[i - 2];  
    }  
  
    return fib_array;  
}
```

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.

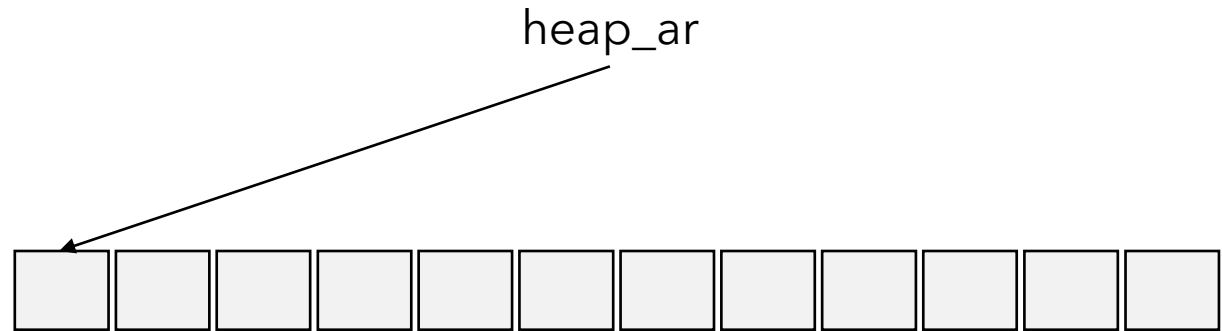
The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized.

Calloc

```
#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 sullo 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

Realloc e gli array dinamici

- Finora abbiamo visto:
 - gli array allocati sullo stack, di dimensione costante e stabilita a compile-time
 - gli array allocati dinamicamente sullo 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

Riempire un array con dati da standard input

```
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 sullo heap, ma non effettua alcun controllo sul superamento dei limiti.

Un programma del genere avrà un comportamento imprevedibile e sarà insicuro

Riempire un array con dati da standard input

```
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 sullo 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

Come si comporta realloc

heap_ar:
address
0xCAFEBEEF



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

Come si comporta realloc

heap_ar:
address
0xCAFEBEEF

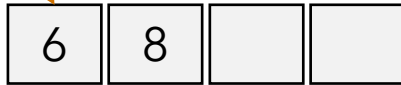


dopo l'invocazione di `calloc`, sullo heap c'è spazio per un array di 4 interi

il programma legge 6 da `stdin`

Come si comporta realloc

heap_ar:
address
0xCAFEDEED

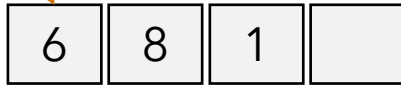


dopo l'invocazione di `calloc`, sullo heap c'è spazio per un array di 4 interi

il programma legge 8 da `stdin`

Come si comporta realloc

heap_ar:
address
0xCAFEBEEF

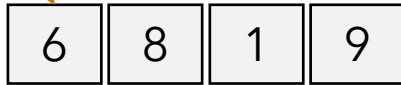


dopo l'invocazione di `calloc`, sullo heap c'è spazio per un array di 4 interi

il programma legge 1 da `stdin`

Come si comporta realloc

heap_ar:
address
0xCAFEBEEF



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

il programma legge 9 da stdin

Come si comporta realloc

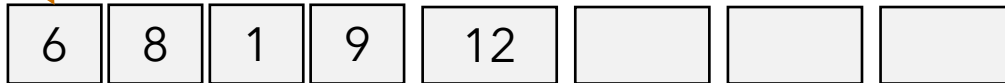
heap_ar:
address
0xCAFEBEEF



chiamata a realloc: la dimensione
dell'array viene raddoppiata

Come si comporta realloc

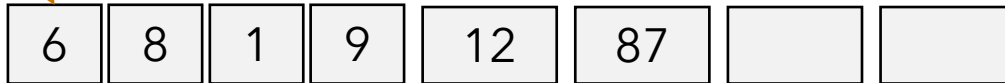
heap_ar:
address
0xCAFEBEEF



il programma legge 12 da stdin

Come si comporta realloc

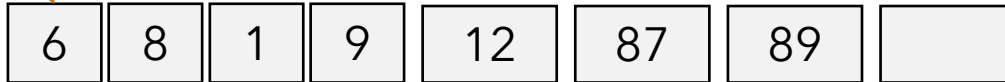
heap_ar:
address
0xCAFEBEEF



il programma legge 87 da stdin

Come si comporta realloc

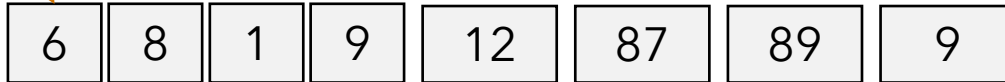
heap_ar:
address
0xCAFEBEEF



il programma legge 89 da stdin

Come si comporta realloc

heap_ar:
address
0xCAFEBEEF

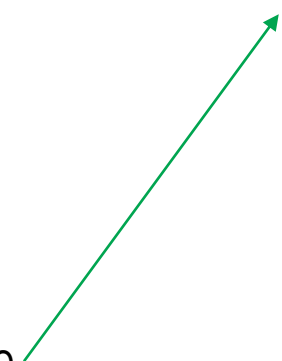


il programma legge 9 da stdin,
l'array è pieno... bisogna chiamare
realloc per allungarlo

Si tratta di una semplificazione. Se non c'è più spazio per riallocare l'array a partire dalla stessa posizione, l'array viene riallocato da un'altra parte sullo heap e il suo contenuto viene copiato elemento per elemento

Esempio di riallocazione dell'array a partire da una nuova posizione

```
cyofanni@LAPTOP-I0S1RKRC:~/Desktop/high-school-cs-class/c_lectures/heap_memory$  
./calloc_realloc_examples  
heap_ar allocated with size 4 at address 0x5575ea81f2a0  
enter an integer: 4  
enter an integer: 5  
enter an integer: 3  
enter an integer: 3  
heap_ar allocated with size 8 at address 0x5575ea81f2a0  
enter an integer: 1  
enter an integer: 2  
enter an integer: 3  
enter an integer: 4  
heap_ar allocated with size 16 at address 0x5575ea81fae0  
enter an integer: 5  
enter an integer: 7  
enter an integer: 6  
enter an integer: 5  
enter an integer: 8  
enter an integer: 7
```



l'indirizzo è cambiato!