Modulo: Allocazione dinamica della memoria in C [P2_05]

Unità didattica: Approfondimento C: classi di memorizzazione [1-AC]

Titolo: Allocazione in memoria e massimo size di array in C

Argomenti trattati:

- ✓ Allocazione di array (locali, globali, …)
- ✓ Classi di memorizzazione (automatica, statica e dinamica)
- ✓ Esempi

Prerequisiti richiesti: allocazione dinamica in C

Qual è il massimo size di un array?

```
#include <stdio.h>
  #define MAX_SIZE 1048576 // =1MB=1024^2: Win SI, Linux SI
//#define MAX_SIZE 2097152 // =2MB        : Win NO, Linux SI
int main()
{ unsigned char A[MAX_SIZE];
  int k;
  printf("(OK!)\tSIZE = %d\n", SIZE);
  for (k=0; k<SIZE; k++)
      A[k]=k%10;
  printf("\A[MAX_SIZE-1] = %u\n\n"), A[MAX_SIZE-1]);
return 0;}
```

A array locale allocaz. statica

Win + Code::Blocks

Funziona* per MAX_SIZE = 1048576 byte = 1MB

Non funziona* per MAX_SIZE = 2097152 = 2MB

* Dipende dal PC/SO

Linux

Funziona* per MAX_SIZE=7340032 = 7MB

Non funziona* per MAX_SIZE=8388608 = 8MB = 8192KB

Linux

```
$ ulimit -s
8192   [in KB]
```

Qual è il massimo size di un array?

```
#include <stdio.h>
  #define MAX_SIZE 1073741824 // 1GB: Win: SI, Linux: SI
//#define MAX_SIZE 2147483648 // 2GB: Win: NO, Linux: NO
unsigned char A[MAX_SIZE];
int main()
{ int k;
  printf("(OK!)\tSIZE = %d\n", SIZE);
  for (k=0; k<SIZE; k++)
      A[k]=k%10;
  printf("\A[MAX_SIZE-1] = %u\n\n"), A[MAX_SIZE-1]);
return 0;}
```

A array globale
allocaz. statica

Win + Code::Blocks

Funziona* per SIZE = 1073741824 byte = 1GB

Non funziona* per SIZE = 2147483648 = 2GB

* Dipende dal PC/SO

Linux

Funziona* per SIZE= 1073741824 byte =1GB

Non funziona* per SIZE= 2147483648 =2GB

Qual è il massimo size di un array?

pA array dinamico

```
#include <stdio.h>
#include <stdlib.h>
typedef unsigned int MY_TYPE;
int main()
{ unsigned int k, MEGABYTE;     MY_TYPE *pA;
  MEGABYTE=1024*1024;
 k=1; pA=(MY_TYPE *)malloc(k*MEGABYTE*sizeof(MY_TYPE));
 while (pA>0)
 {  printf("\nk = %d:  Allocati %d MB di memoria dinamica",
                                          k,k*sizeof(MY_TYPE));
     free(pA);
     k++; pA=(MY_TYPE *)malloc(k*MEGABYTE*sizeof(MY_TYPE));
 }
return 0;}
```

finché l'allocazione viene eseguita ...

Win + Code::Blocks 13.12*

k = 464: Allocati 1856 MB ≈ 2^{31} byte > 1GB di memoria dinamica

[log2(1856*1024^2)=30.858]

Linux 64bit*

k = 4095: Allocati 16380 MB ≈ 16 GB di memoria dinamica

... dipende dalla RAM

* Dipende dal PC_{RAM}/SO

Dove sono allocate le variabili di un programma C in esecuzione?

Tre aree di memoria per allocare le variabili in C:

- memoria automatica, detta "stack", (auto keyword)
- memoria dinamica , detta "heap", (malloc/free)
- memoria statica (static keyword/global space)

```
int main()
{ ...
  unsigned char A[MAX_SIZE];
 ...}
```

memoria stack

```
unsigned char A[MAX_SIZE];
int main()
{ ...
}
```

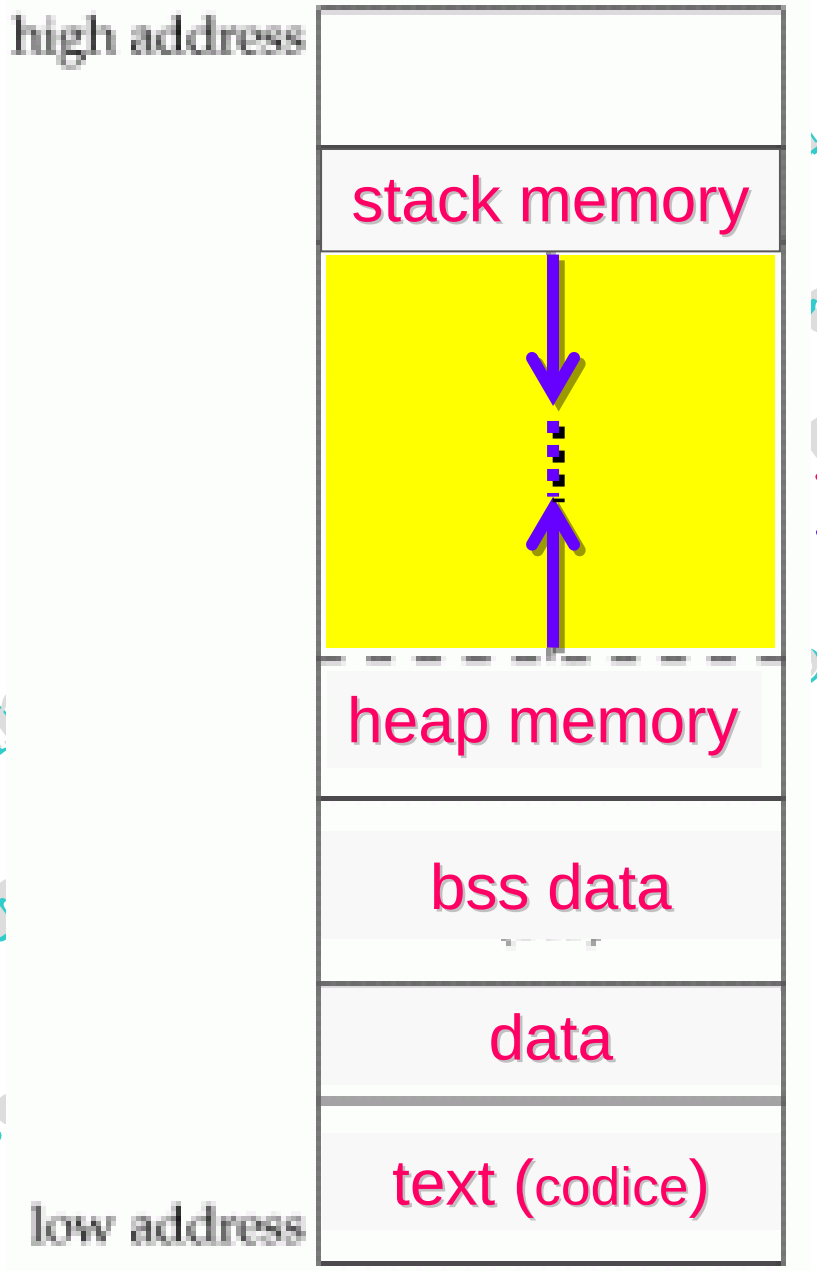
memoria statica

```
int main()
{ ...
  pA = malloc(…);
 ...}
```

memoria heap

Solo la memoria automatica è limitata superiormente; le memorie dinamica e statica potenzialmente possono occupare tutto lo spazio messo a disposizione dal sistema operativo

Il segmento di memoria di un programma in esecuzione



L'ammontare delle memorie stack e heap varia durante l'esecuzione

non inizializzati da programma

inizializzati da programma

Esempio: Linux

```
$ size /usr/bin/gcc
```
← eseguibile

```
   text       data        bss        dec        hex    filename
 759962       8344      80992     849298      cf592    /usr/bin/gcc
```

```
  759962 +
    8344 +
   80992 =
  ------
  849298
```

codice (text)

dati (data) [inizializzati da programma]

dati bss [non inizializzati da programma]

```
$ ulimit -s
8192
```

← `-s` per la memoria stack

Limite, in KB, per la memoria automatica

Esempio: differenza tra "data" e "bss"

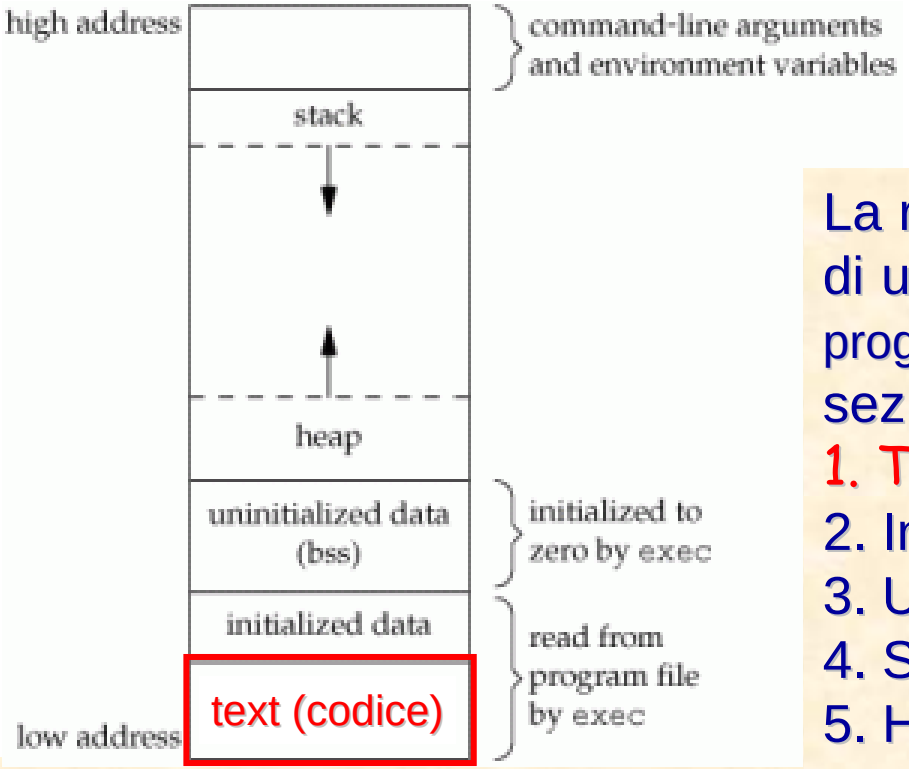
```c
#include <stdio.h>

int i[2];      /* memoria statica (bss) */
int j[2]={1}; /* memoria statica (data: inizializzati) */

int main()
{
    int k[2];          /* memoria automatica (stack) */
    static int m[2]; /* memoria statica (bss) */

    printf("int i[2];         (bss)   ==>  i = %d, %d\n", i[0],i[1]);
    printf("static int m[2]; (bss)   ==>  m = %d, %d\n", m[0],m[1]);
    printf("int j[2]={1};     (data)  ==>  j = %d, %d\n", j[0],j[1]);
    printf("int k[2];         (stack) ==>  k = %d, %d\n", k[0],k[1]);

    return 0;
}
```

```
int i[2];          (bss)    ==>  i = 0, 0
static int m[2];(bss)    ==>  m = 0, 0
int j[2]={1};     (data)   ==>  j = 1, 0
int k[2];          (stack) ==>  k = 61,2
```

inizializzato dal compilatore

inizializzato da programma

non inizializzato



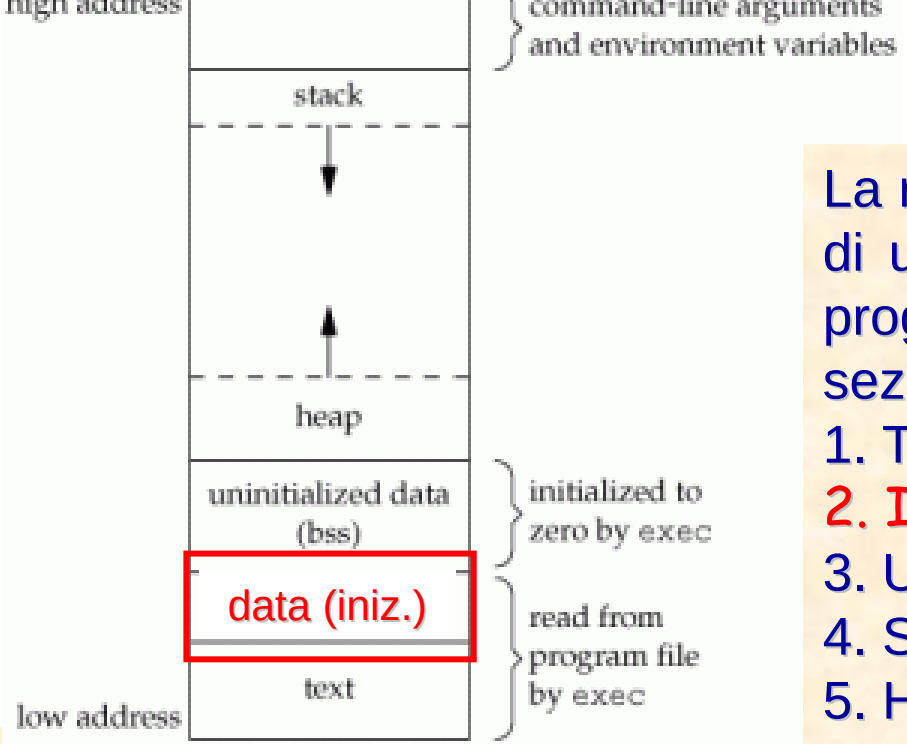
La rappresentazione tipica della memoria di un processo in esecuzione (come da un programma C) consiste delle seguenti sezioni:

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap

1. Text Segment

Un segmento text (detto anche code segment) è una delle sezioni di un programma in un object file o in memoria. Contiene le istruzioni eseguibili. È allocato in memoria al di sotto dell'area heap/stack per prevenire che un overflow dell'heap/stack possa sporcarlo.

Solitamente un segmento text può essere condiviso (sharable) in modo che solo una copia sia necessaria in memoria per i programmi eseguiti frequentemente (come i text-editor, il compilatore C, etc.). Inoltre, spesso, esso è read-only, per prevenire che un programma modifichi accidentalmente le sue istruzioni.



La rappresentazione tipica della memoria di un processo in esecuzione (come un programma C) consiste delle seguenti sezioni:

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap

2. Initialized data segment

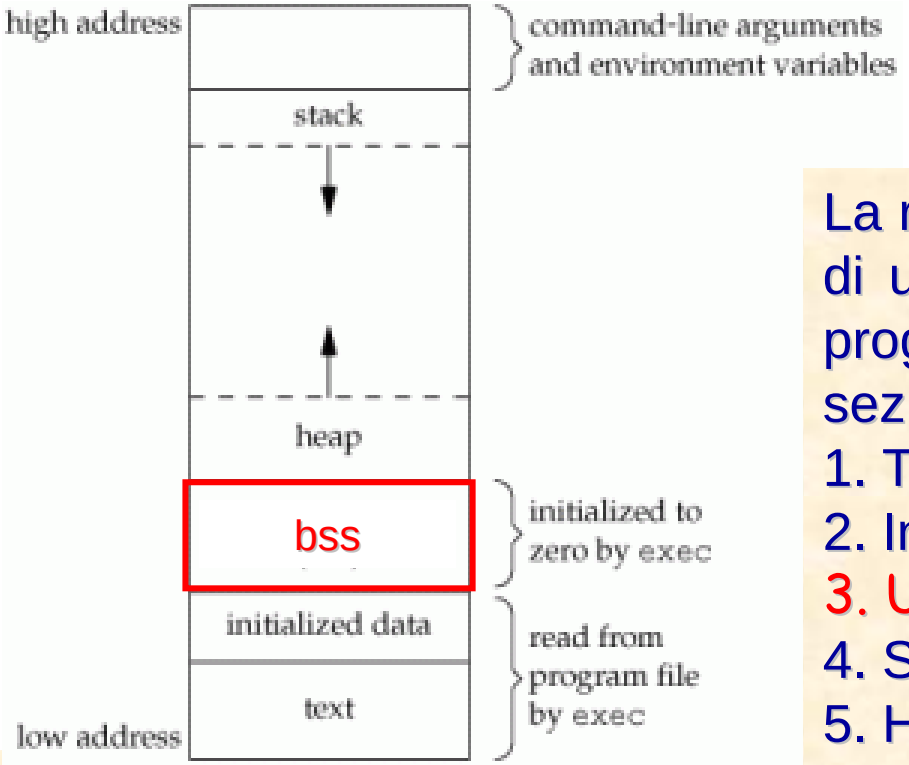
Il segmento dati inizializzato (detto data segment) è una porzione dello spazio degli indirizzi virtuali di un programma. Contiene le variabili globali, le variabili `static`, `const` ed `extern` che sono inizializzate nel programma. Il segmento dati non è tutto read-only: esso può essere ulteriormente suddiviso in initialized read-only area e initialized read-write area. Per esempio:

(globale) `char s[]="hello world"` ⇒ `s` è memorizzata in initialized read-write area

(globale) `int debug=1` ⇒ `debug` in initialized read-write area

`static int i=10` ⇒ `i` è memorizzata in initialized read-write area

(globale) `const char* string = "hello world"` ⇒ la stringa costante è memorizzata in initialized read-only area, mentre il puntatore è memorizzato in initialized read-write area.



La rappresentazione tipica della memoria di un processo in esecuzione (come un programma C) consiste delle seguenti sezioni:

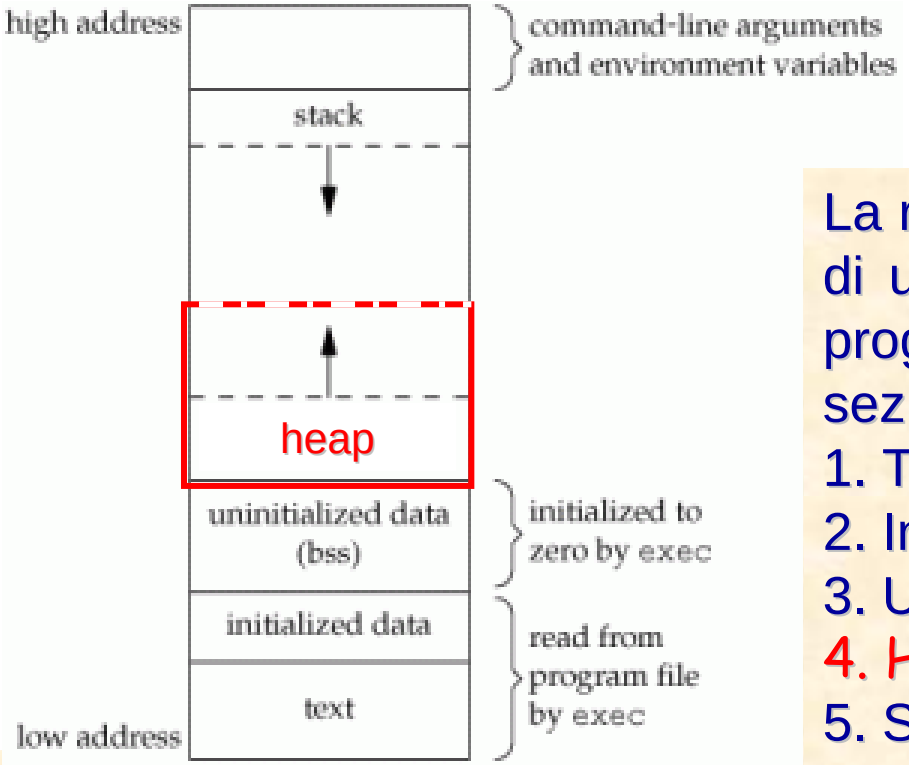
1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap

3. Uninitialized data segment

Il segmento dati non inizializzato (detto storicamente "**bss** *segment*") contiene le variabili *globali*, `static` ed `extern` che sono inizializzate dal kernel a 0 prima che il programma cominci l'esecuzione. In un **object file** questa sezione non occupa spazio: l'object file distingue tra variabili inizializzate e non inizializzate per efficienza di spazio; le variabili non inizializzate non occupano spazio su disco nell'object file. Il segmento **bss** comincia alla fine del **data segment** e contiene tutte le variabili globali e quelle `static` che sono inizializzate a 0 o non hanno un'esplicita inizializzazione nel codice sorgente. Per esempio:

`static int i;` ⇒ `i` è memorizzata in **bss**

(globale) `int j;` ⇒ `j` è memorizzata in **bss**

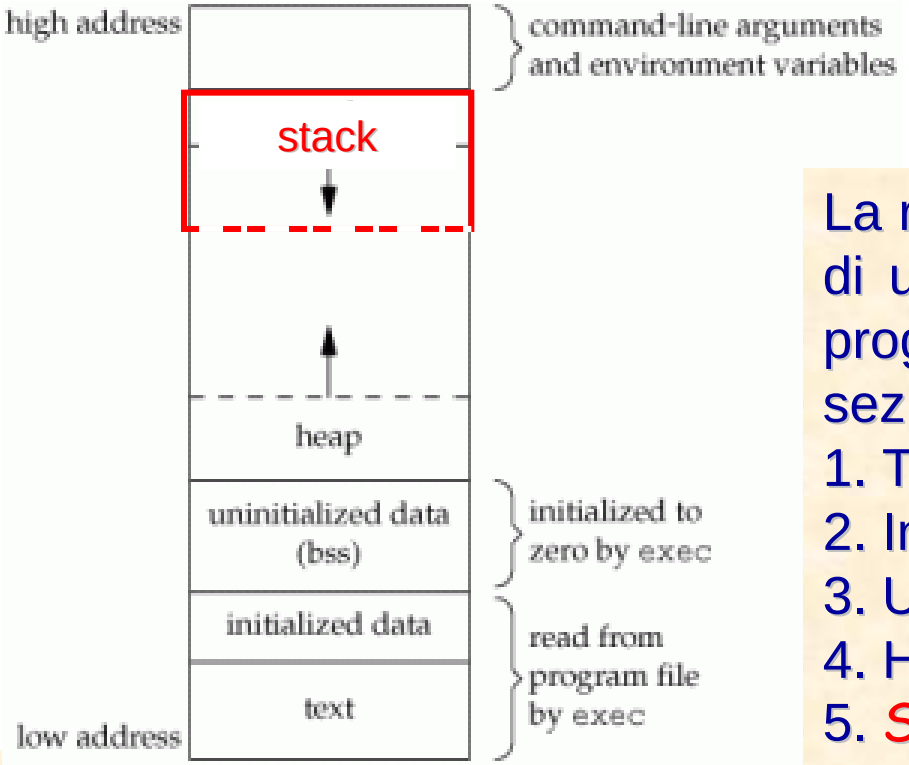


La rappresentazione tipica della memoria di un processo in esecuzione (come un programma C) consiste delle seguenti sezioni:

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Heap
5. Stack

4. Heap

Il segmento heap è quello dove avviene l'allocazione dinamica della memoria. Esso comincia alla fine del *segmento bss* e cresce verso la memoria alta. In C l'*area heap* è gestita dalle funzioni malloc(), calloc(), realloc() e free().



La rappresentazione tipica della memoria di un processo in esecuzione (come un programma C) consiste delle seguenti sezioni:

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Heap
5. Stack

5. Stack

L'area stack è adiacente all'**heap** e cresce nella direzione opposta; quando lo **stack pointer** incontra l'**heap pointer**, la memoria libera si è esaurita.

Il **segmento stack** è usato per memorizzare tutte le variabili locali ed è usato per passare gli argomenti alle funzioni e l'indirizzo di ritorno della istruzione che dev'essere eseguita dopo che la funzione è finita. Le variabili locali hanno come "**scope**" il blocco ({…}) in cui sono introdotte; esse sono create quando il controllo entra nel blocco e sono cancellate quando il controllo esce dal blocco. Anche tutte le chiamate di una funzione ricorsiva sono aggiunte allo **stack**. I dati sono allocati o deallocati nello stack secondo la filosofia LIFO (Last-In-First-Out).