Programmazione I

Il Linguaggio C

I puntatori

Daniel Riccio

Università di Napoli, Federico II

15 novembre 2021

Sommario

- Argomenti
  - Assegnamento dei puntatori
  - Allocazione e deallocazione di memoria

Vettori di puntatori

Gli elementi di questi vettori sono puntatori

```
int *vett[10];
```

definisce un vettore di 10 puntatori a `int`
(le **[ ]** hanno priorità maggiore dell'operatore *****)

Esempio di inizializzazione

```
int a, b, c;
int *vett[10]={NULL};
vett[0]=&a;
vett[1]=&b;
vett[2]=&c;
```

I valori da `vett[3]` a `vett[9]` sono tutti NULL in quanto è stato inizializzato il primo elemento (i successivi sono automaticamente a 0 e 0 viene convertito in NULL)

Vettori di puntatori

Esempio di inizializzazione errata

```
int a, b, c;
int *vett[10]={&a, &b, &c};
```

È errato perché questi inizializzatori sono indirizzi di variabili automatiche

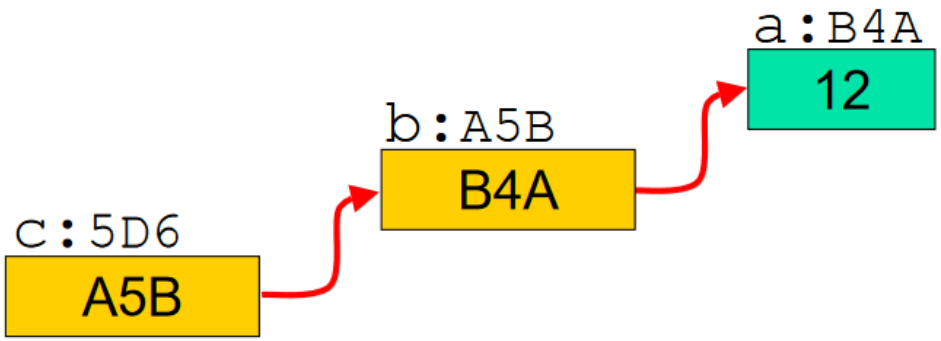
Puntatori a puntatori

Variabili che contengono l'indirizzo di memoria di un puntatore

Esempio:

```
int a, *b, **c;
a = 12;
b = &a;
c = &b;
```

Il puntatore `c` punta ad una variabile (`b`) che punta ad un `int` (`a`)



Puntatori a puntatori

Esempio:

```
int a=10, b=20, c=30;
int *v[3], int **w;
v[0]=&a; v[1]=&b; v[2]=&c;
w = v;
w++;
```

v è un vettore di puntatori, cioè è l'indirizzo di memoria (“puntatore”) di un puntatore

Quindi **v**+1 è …

… l'indirizzo del secondo puntatore del vettore **v** (ossia è pari a **&v**[1], punta a **b**)

Anche **w** è un puntatore ad un puntatore, quando viene incrementato, punta al puntatore successivo, come **v**

Puntatori a puntatori

Una matrice è un vettore di vettori e quindi, considerando che l'associatività di [] è da sinistra a destra, si ha che:

```
int Mx[7][5];
```

definisce un vettore di 7 elementi ciascuno dei quali è un vettore di 5 `int`

Gli elementi del vettore `Mx` sono i 7 vettori identificati da `Mx[i]`;
quindi `Mx[i]` è l'indirizzo di memoria di ciascuno dei 7 vettori di 5 **`int`**

`Mx` è di tipo puntatore-a-`int` come **`int`** `*p`? **NO**

`Mx` è di tipo puntatore-a-puntatore-a-`int` come **`int`** `**p`? **NO**

`Mx` è di tipo puntatore-a-vettore-di-5-`int` come **`int`** `(*p)[5]` **OK**
quindi **`Mx`**`+1` punta al secondo vettore di 5 `int`

Puntatori e matrici

Nella definizione di puntatore seguente

```
int (*p)[5]
```

è necessario che la dimensione delle colonne (5) sia specificata perché definisce un puntatore-a-vettore-di-**cinque**-interi

Poiché **`Mx`** è un puntatore-a-vettore-di-**5**-interi (e non un puntatore ad un intero), allora:

`Mx``+1` punta all'elemento successivo, ossia al successivo vettore di 5 interi (la seconda riga della matrice):

`Mx``+1`

aggiunge all'indirizzo di **`Mx`** il numero corretto di byte per puntare al secondo vettore di **5** **`int`**

Puntatori e matrici

Ricapitolando:

`int` **`Mx`**`[7][5];` è un puntatore-a-vettore-di-5-`int`

- **`Mx`**`[1][2]`
  - è un valore scalare
  - è di tipo **`int`**
  - è modificabile
  - **`Mx`**`[1][2]+1` somma 1 al contenuto di **`Mx`**`[1][2]`
- **`Mx`**`[1]`
  - è l'indirizzo di un vettore di 5 **`int`**
  - è di tipo puntatore-a-**`int`**
  - non è modificabile
  - **`Mx`**`[1]+1` punta a **`Mx`**`[1][1]`
- **`Mx`**
  - è l'indirizzo di un vettore di 7 vettori di 5 `int`
  - è di tipo puntatore-a-vettore-di-`int`
  - non è modificabile
  - `Mx+1` è l'indirizzo di memoria di `Mx[1]`

`int *p;` puntatore-a-**`int`**
`int (*q)[5];` puntatore-a-vett-di-5-**`int`**

`p = Mx;` **NO** è errata
`q = Mx;` **OK** `q+1` è l'indirizzo del secondo vettore di 5 elementi (equivale a `Mx[1]`)
`p = &Mx[0][0];` **OK** `p` è l'indirizzo del primo elemento dei `Mx[0]`

Vettori di stringhe

Essendo una stringa un vettore di caratteri, un vettore di stringhe è in realtà un vettore di vettori di caratteri, cioè una matrice di `char`

```
char str[4][20]={"uno", "due"};
```

definisce un vettore di 4 stringhe di 20 `char`

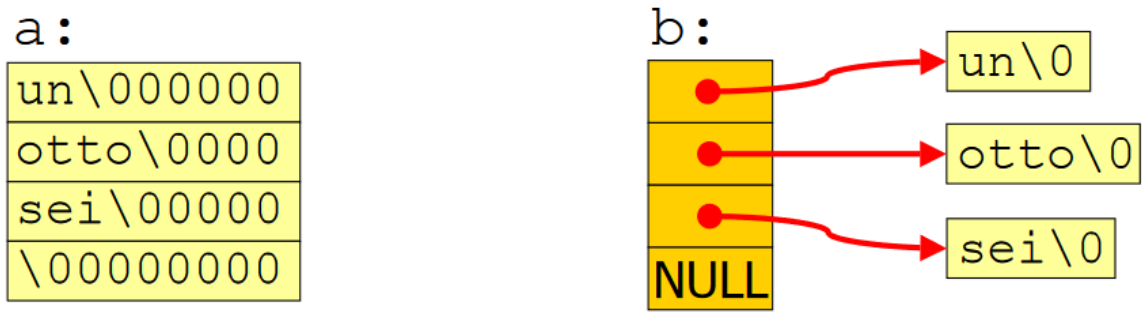
Le 4 stringhe sono identificate da `str[i]`

	str:																			
str[0]	u	n	o	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
str[1]	d	u	e	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
str[2]	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
str[3]	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

Puntatori a puntatori

Si notino le differenze tra:

```
char a[4][8]={"un","otto","sei"};
char *b[4]={"un","otto","sei"};
```



`a[i]` è l'indirizzo costante della riga `i` di `a`, tale riga è una stringa variabile di 8 `char`

`b[i]` è una variabile puntatore con l'indirizzo della riga `i`, stringa costante di 4 caratteri:

ad esempio `b[2]` contiene l'indirizzo di `"sei\0"`

Vettori di puntatori e matrici

```
a[2] = "hello";
```
ERRORE: `a[2]` non è un puntatore

```
strcpy(a[2], "hello");
```
CORRETTO

```
b[2] = "hello";
```
CORRETTO: `b[2]` è una variabile puntatore a cui viene assegnato l'indirizzo di memoria di una stringa costante

```
strcpy(b[2], "hello");
```
ERRORE: `b[2]` punta a una stringa costante

Entrambi `a[2][0]` e `b[2][0]` sono il carattere `'h'`

```
a[1][0]='m';
```
SI'! L'oggetto puntato da `a[1]` è una stringa variabile

```
b[1][0]='m';
```
NO! L'oggetto puntato da `b[1]` è una stringa costante

Const per puntatori a puntatori

```
int ** x;
```
x è …

una variabile di tipo puntatore a un puntatore variabile a un oggetto variabile di tipo `int`

```
const int ** x;
```
x è …

una variabile di tipo puntatore a un puntatore variabile a un oggetto costante di tipo `int`

```
int ** const x;
```
x è …

una costante di tipo puntatore a un puntatore variabile a un oggetto variabile di tipo `int`

```
int * const * x
```
x è …

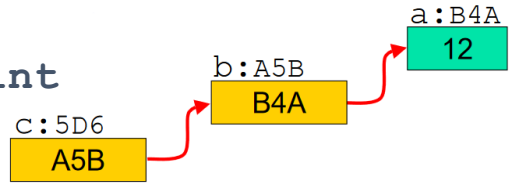
una variabile di tipo puntatore a un puntatore costante a un oggetto variabile di tipo `int`

```
const int * const * x
```
x è …

una variabile di tipo puntatore a un puntatore costante a un oggetto costante di tipo `int`

```
const int * const * const x
```
x è …

una costante di tipo puntatore a un puntatore costante a un oggetto costante di tipo `int`

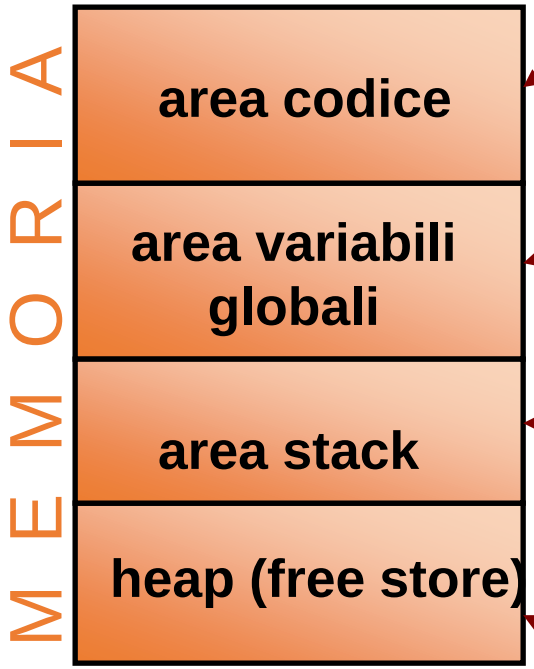


Assegnare i puntatori

- In memoria, un puntatore è un indirizzo di memoria
  - (...di una variabile)
  - (...di cui è noto il tipo)
- Bene, ma quale indirizzo?
  - Modo 1: prendere l'indirizzo di una variabile esistente
    - il puntatore punterà a quella variabile
  - Modo 2: allocare (riservare, prenotare) della memoria libera
    - il puntatore punterà ad una nuova variabile, memorizzata nella memoria così riservata
    - la nuova variabile è **allocata dinamicamente**

Organizzazione della memoria

Quattro aree:



Qui viene tenuto il codice che viene eseguito (**linguaggio macchina**)

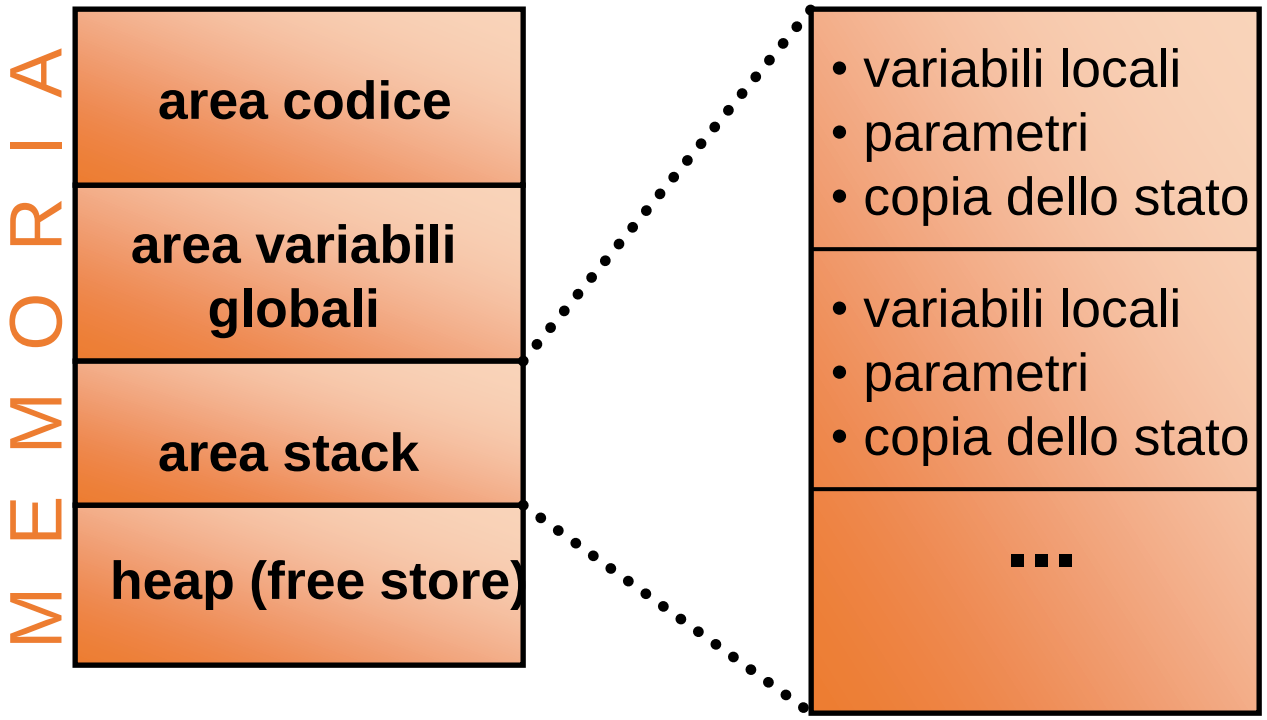
Qui risiedono le **variabili globali**

Qui risiedono le **variabili locali**

il resto, **memoria libera**.
Riserva da cui si attinge lo spazio per le variabili allocate dinamicamente (con le malloc e calloc)

Organizzazione della memoria

Quattro aree:



record di attivazione per la prima funzione chiamata

record di attivazione per la funzione chiamata dalla prima funzione

Organizzazione della memoria

Cosa conosce il compilatore (staticamente)

- delle variabili globali: l'**indirizzo**
- delle variabili locali: l'**offset**
  - rispetto al record di attivazione
  - il vero indirizzo sarà:
    (posizione dell'attuale record di attivazione) + (offset)
- delle costanti: il **valore**

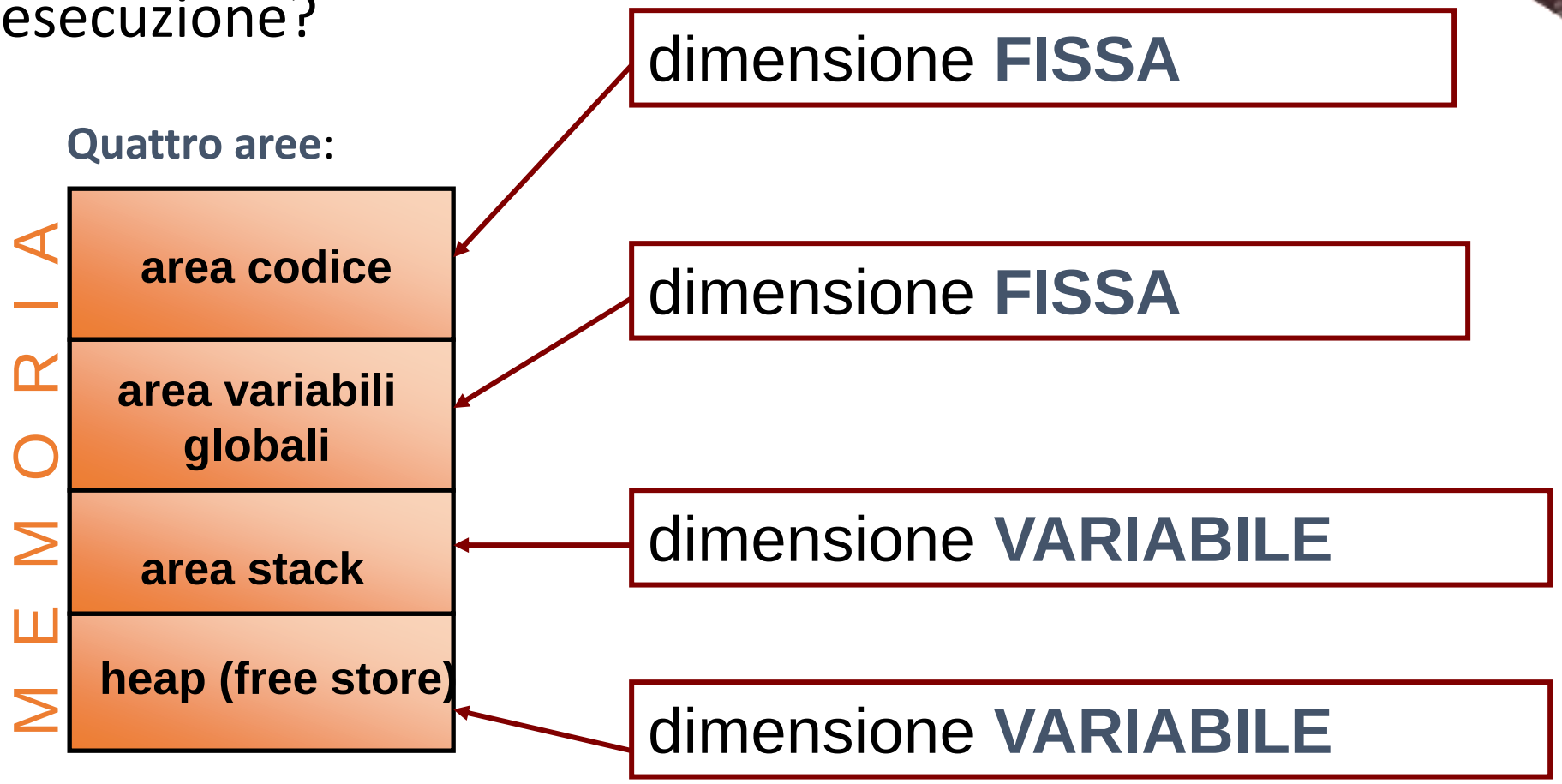
```
const int A=10;
int b=10;

int main(int p){
   int d;
   ...
}
```

tabella dei Simboli del compilatore

ide.	*tipo*	*locazione o valore*	*o offeset*
`A`	int	`---`	`10`	`---`
`b`	int	`0xA12F345A`	`---`	`---`
`p`	int	`---`	`---`	`0x00000020`
`d`	int	`---`	`---`	`0x00000030`

Organizzazione della memoria

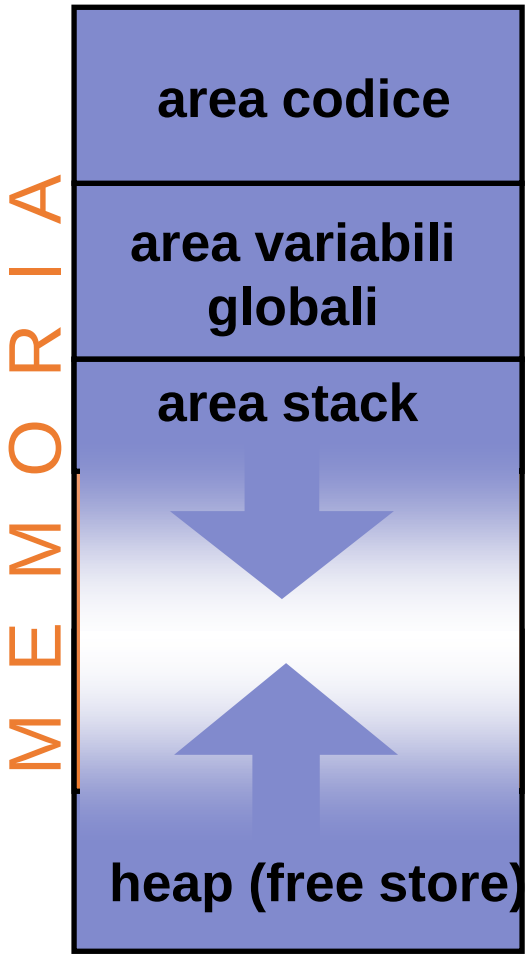
Nota: cambia o no la dimensione delle aree durante l'esecuzione?



Organizzazione della memoria

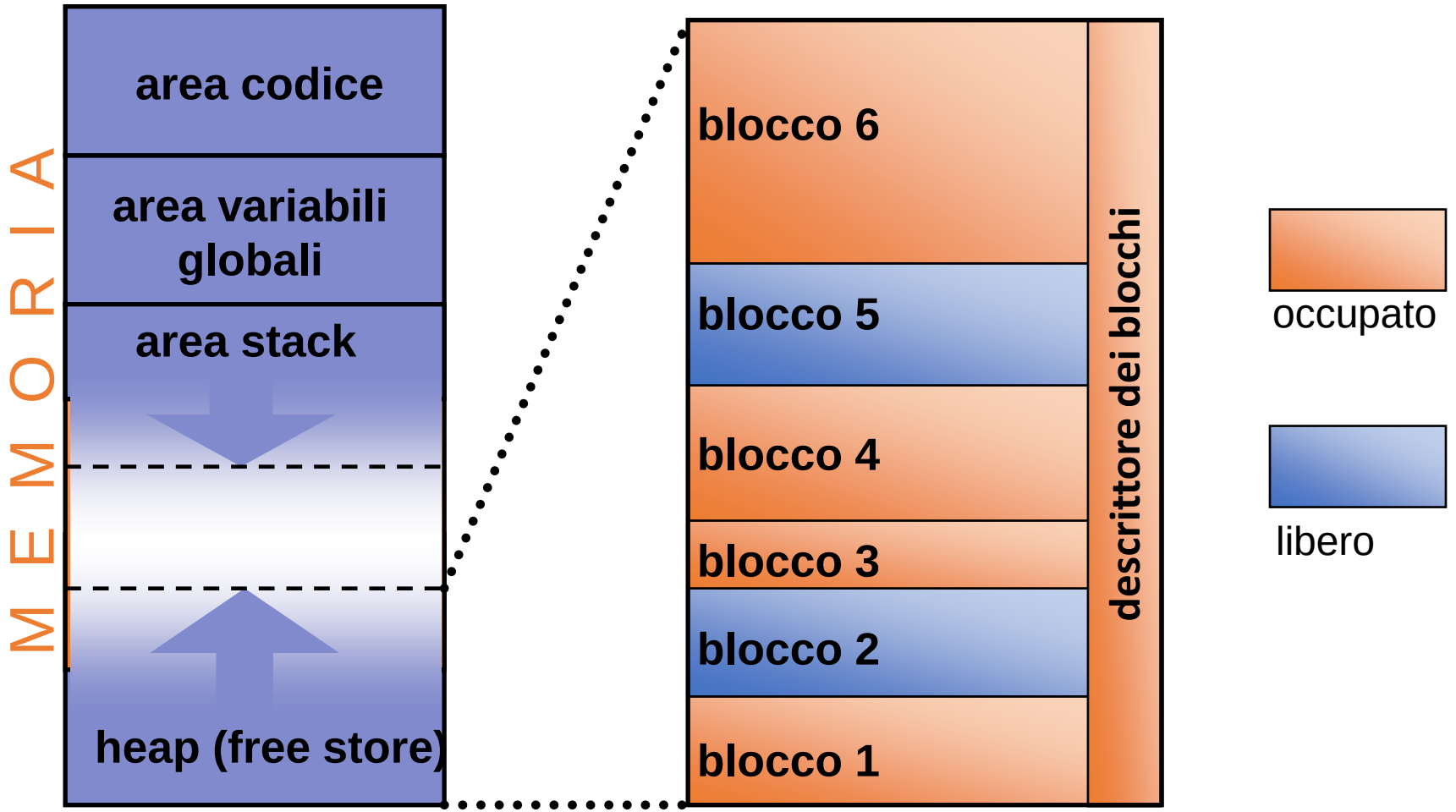
Nota: cambia o no la dimensione delle aree durante l'esecuzione?

Tipica implementazione:



Organizzazione della memoria

Nota: cambia o no la dimensione delle aree durante l'esecuzione?



Allocazione

```
void* malloc(unsigned int n);
```

funzione **malloc** = **m**-emory **alloc**-ation

1 - alloca `n` bytes di memoria.

2 - restituisce l'**indirizzo** della memoria appena allocata

- sotto forma di puntatore *generico* !
- `void*` puntatore generico,
  puntatore senza tipo, in pratica, un semplice indirizzo di memoria

Allocazione - esempio

```
int* p;
p = malloc( 4 );
```

Errore di tipo!

A **sx** si ha un (int*) mentre a **dx** un (void*)

Il tipo è diverso, ma si tratta sempre di un indirizzo di memoria

Soluzione: basta fare un type-cast:

```
int* p;
p = (int*) malloc(4);
```

Allocazione – memoria esaurita

```
void* malloc(unsigned int n);
```

Se non c'è più memoria, l'allocazione **fallisce**
e **malloc** restituisce il valore speciale **NULL**

- semanticamente, **NULL** è un "puntatore che non punta a nulla"
- in memoria, **NULL** è rappresentato dal valore 0

Il valore resituito dalle **malloc** va controllato

```
int* p;
p = (int*) malloc(4);
if (p == NULL) {
  /* memoria finita... */
}
```

oppure,
più coincisamente

```
if(!p) {
```

Allocazione e record

```
typedef struct {
  \*blah blah... Campi dell'array...*\
} NuovoTipo

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

Il costrutto **`sizeof`** è estremamente utile con le **malloc**

Usare sempre, anche con i tipi base

- int, short, float, double...
- nota: il C non prescrive quanti bytes occupano!

Deallocazione

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

libera la memoria che era stata allocata all'indirizzo **p**.

Nota: **p** deve essere il risultato di una **malloc**

```
int* p;
p = (int*) malloc(sizeof(int));
... /* uso *p */
free(p);
```

se si dimentica di deallocare, si ha un cosiddetto **memory leak**

Nota: non c'è alcuna **garbage collection** in **C**

Deallocazione

k viene **automaticamente allocato** (i 4 bytes di memoria necessari al suo immagazzinamento vengono riservati).

k viene inizializzato (a 15)

All'uscita dalla procedura, i 4 bytes sono resi di nuovo disponibili

```
int main() {
  int k;
  k=15;

  ... /* lavora con k */

  return 0;
};
```

Usando l'allocazione dinamica:

k viene **esplicitamente allocato** (a tempo di esecuzione, si allocano i 4 bytes di memoria necessari al suo immagazzinamento.
La locazione viene memorizzata in **k**).

k viene inizializzato (a 15)

All'uscita dalla procedura, bisogna rendere i 4 bytes di nuovo disponibili esplicitamente

```
int main() {
  int* k;
  k = (int*)malloc(sizeof(int));

  *k = 15;

  ... /* lavora con *k */

  free(k);
  return 0;
};
```

Allocazione di vettori

```
(void*) calloc(unsigned int n, unsigned int size);
```

calloc = **c**-ontiguous **alloc**-ation

Alloca **n** elementi contigui ciascuno grande **size**

In pratica, alloca un area di memoria grande **n** x **size**

Per il resto funziona come **`malloc`**

Esempio:

```
int* p;
p = (int*)calloc(100000,sizeof(int) );
```

Alloca un vettore di 100000 interi.

Allocazione di vettori

Da ricordare:

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) fixed size vector:

```
int v[100000];
```

In entrambi i casi:

Si ha un vettore di 100000 interi

Si può scrivere ad esempio:

```
v[2]= v[0] + 3 * v[1];
```

Allocazione di vettori

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) fixed size vector:

```
int v[100000];
```

Differenza 1: dimensione variabile

- se **X** è una variabile intera, si può scrivere:

```
int* v = (int*)calloc(x,sizeof(int));
```

- ma non posso scrivere:

```
int v[x];
```

qui è richiesta una costante

Allocazione di vettori

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) fixed size vector:

```
int v[100000];
```

Differenza 2: bisogna deallocare la memoria

```
int* v = (int*)calloc(100000,sizeof(int));

... /* usa v */

free(v);
```

Allocazione di vettori

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) Vettore di dimensione fissa:

```
int v[100000];
```

Differenza 3: dimensione fissa = più efficiente

- il solito prezzo da pagare per l'uso dei puntatori...
- ...maggiorato
- Si supponga che **v** valga `0xAA000000`:

- se fissa `v[2]` → compilazione →

```
READ TEMP 0xAA000008
```

`0xAA000000 + 2 x sizeof(int)` ma precalcolato staticamente

- se dinamco `v[2]` → compilazione →

```
READ TEMP0 0xAA000000
ADD  TEMP0 8
READ TEMP  TEMP0
```

Allocazione di vettori

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) fixed size vector:

```
int v[100000];
```

Differenza 4:

- vengono allocati in zone diverse della memoria...

Esempio allocazione di memoria

Scrivere un programma che:

1) Dichiari un puntatore a puntatore ad intero
2) Chieda in input all'utente un intero N
3) Usi il puntatore per allocare una matrice di NxN interi
4) Ponga nella posizione (i,j) della matrice il valore i+j, $\forall$ i,j
5) Stampi a video la matrice
6) Deallochi la memoria riservata per la matrice

Esempio allocazione di memoria

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

int main()
{
    int **Valori = NULL;
    int i,j;
    int N;

    printf("Inserisci la dimensione della matrice:");
    scanf("%d", &N);

    // Allochiamo il vettore di righe
    Valori = (int **)malloc(N*sizeof(int));


    // Per ciascuna riga allochiamo le colonne
    for(i=0; i<N; i++)
       Valori[i] = (int *)malloc(N*sizeof(int));
```

Esempio allocazione di memoria

```
// inizializziamo la matrice con dei valori casuali
    for(i=0; i<N; i++)
       for(j=0; j<N; j++)
          Valori[i][j] = i+j;


    // visualizziamo la matrice
    for(i=0; i<N; i++){
        printf("\n");

       for(j=0; j<N; j++)
          printf("%d ", Valori[i][j]);
    }

    // liberiamo la memoria
    for(i=0; i<N; i++)
       free(Valori[i]);

    free(Valori);
```

```
Inserisci la dimensione della matrice:5

0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Esercizio

Scrivere un programma che legga da input al massimo un certo numero MAXRIGHE di righe di testo e le memorizzi in una matrice di caratteri (MAXRIGHE x 100), una riga per ciascuna riga della matrice.

Si definisca un vettore di puntatori a carattere e lo si inizializzi in modo che il primo puntatore punti alla prima riga, il secondo alla seconda, ecc.

Si ordinino le stringhe scambiando tra di loro solo i puntatori e le si visualizzino ordinate.