IEIM 2015-2016

Esercitazione X

"Ripasso: array, puntatori, ricorsione"

Alessandro A. Nacci
alessandro.nacci@polimi.it - www.alessandronacci.it

- Ripasso su array e puntatori 1
- Ripasso su array e puntatori 2
- Ricorsione

Ripasso sugli array e sui puntatori 1

Materiale tratto da:
http://lia.deis.unibo.it/Courses/FondT1-1011-INF/lezioni/modulo1/14-Matrici.pdf

ARRAY DI PUNTATORI

- Non ci sono vincoli sul tipo degli elementi di un vettore
- Possiamo dunque avere anche ***vettori di puntatori***

Ad esempio:

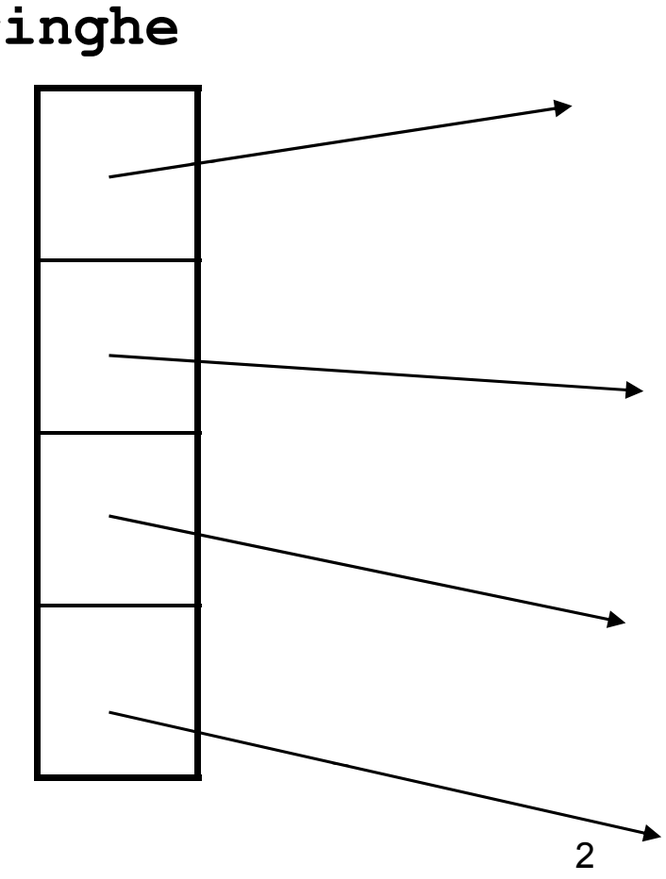
```
char * stringhe[4];
```

definisce un vettore di 4 puntatori a carattere (allocata memoria per 4 puntatori)

ARRAY DI PUNTATORI

`stringhe` sarà dunque una struttura dati rappresentabile nel modo seguente

I vari ***puntatori*** sono ***memorizzati in celle contigue***. Possono invece ***non essere contigue le celle che loro puntano***



INIZIALIZZAZIONE

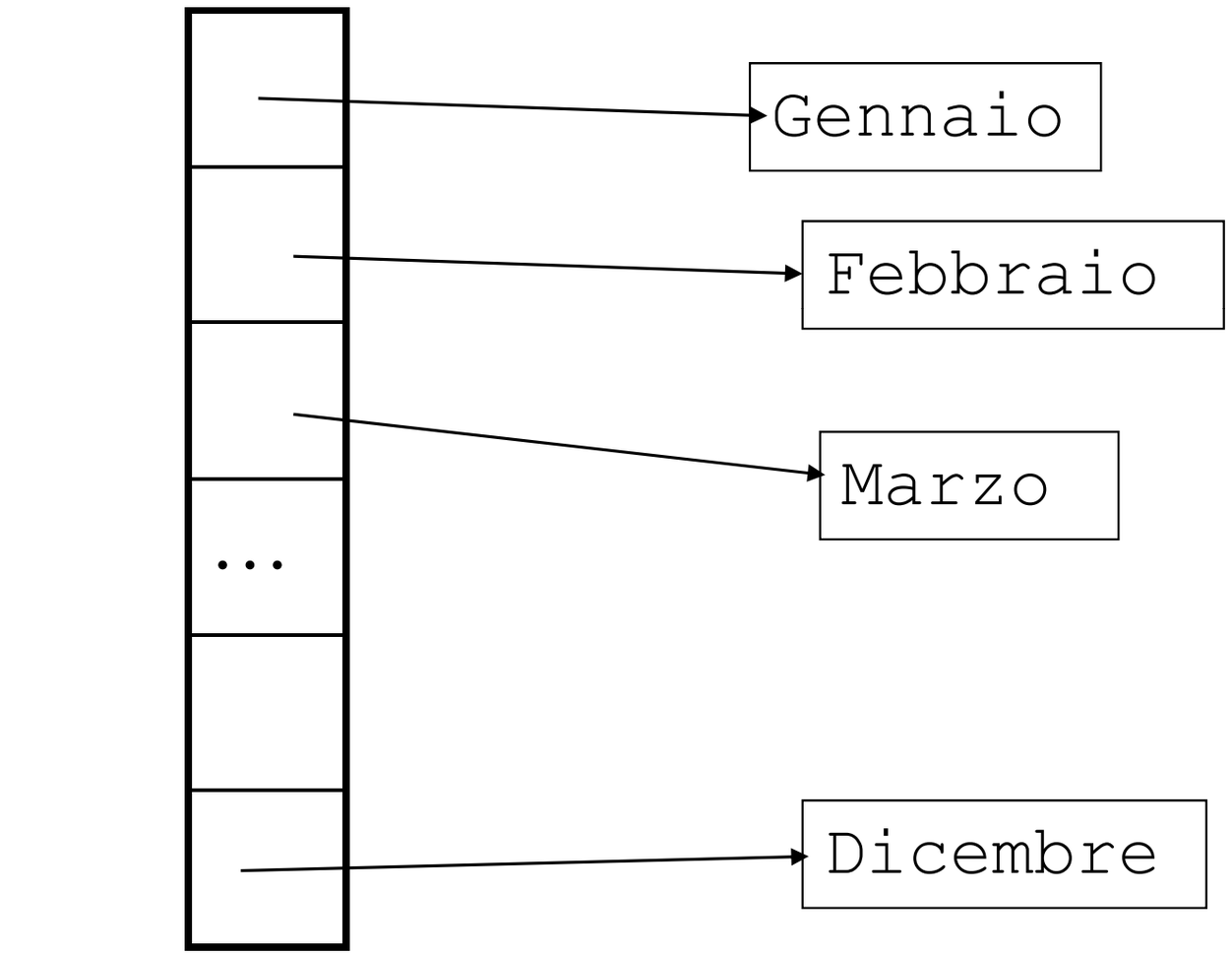
Come usuale, anche gli array di puntatori a stringhe possono essere ***inizializzati***

```
char * mesi[] =          {"Gennaio","Febbraio",
      "Marzo","Aprile","Maggio","Giugno",
      "Luglio","Agosto", "Settembre","Ottobre",
      "Novembre","Dicembre"};
```

I caratteri dell'i-esima stringa vengono posti in una locazione qualsiasi e ***in mesi[i] viene memorizzato un puntatore a tale locazione***

Come sempre, poiché l'ampiezza del vettore non è stata specificata, il compilatore conta i valori di inizializzazione e dimensiona il vettore di conseguenza

INIZIALIZZAZIONE



4

ARRAY MULTIDIMENSIONALI

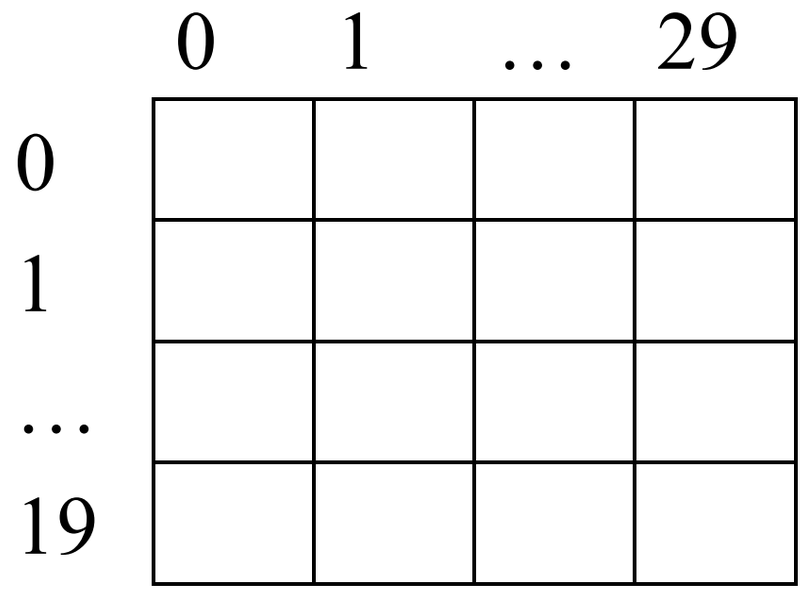
È possibile definire variabili di tipo array con più di una dimensione

```
<tipo> <identificatore>[dim1][dim2]…[dimn]
```

Array con due dimensioni vengono solitamente detti ***matrici***

```
float M[20][30];
```

Come sempre, ***allocazione statica***:
allocazione di 20x30 celle
atte a mantenere float



MATRICI

Per accedere all'elemento che si trova nella ***riga i*** e nella ***colonna j*** si utilizza la notazione

```
M[i][j]
```

Anche possibilità di vettori con più di 2 indici:

```
int C[20][30][40];
int Q[20][30][40][40];
```

MEMORIZZAZIONE

Le matrici multidimensionali sono ***memorizzate per righe in celle contigue***. Nel caso di M:

M[0][0]	M[0][1]	…	M[0][29]	M[1][0]	M[1][1]	…	M[1][29]	….	M[19][0]	M[19][1]	…	M[19][29]

E analogamente nel caso di più di 2 dimensioni: viene fatto variare prima l'indice più a destra, poi il penultimo a destra, e così via fino a quello più a sinistra

Per calcolare ***l'offset*** della cella di memoria dell'elemento (i,j) in una matrice bidimensionale (rispetto all'indirizzo di memorizzazione della prima cella – *indirizzo dell'array*):

LungRiga * i + j

Nel caso di M: **`M[i][j]`** elemento che si trova nella cella 30*i+j dall'inizio della matrice

Elemento: *(&M[0][0] + 30*i+j)

MEMORIZZAZIONE

In generale, se

<tipo> mat[dim_1][dim_2]…[dim_n]

mat[i_1][i_2]…[i_{n-1}][i_n]

è l'elemento che si trova nella cella

i_1*dim_2* *dim_n+ + i_{n-1}*dim_n+i_n a partire dall'inizio del vettore

MATRICI

Si possono anche dichiarare dei tipi vettore multidimensionale

```
typedef <tipo> <ident>[dim1][dim2]…[dimn]
```

```
typedef  float MatReali [20] [30];
MatReali Mat;
```

è equivalente a

```
typedef float VetReali[30];
typedef VetReali MatReali[20];
MatReali Mat;
```

INIZIALIZZAZIONE

Come al solito, i vettori multidimensionali possono essere inizializzati con una lista di valori di inizializzazione racchiusi tra parentesi graffe

```
int matrix[4][4]={{1,0,0,0},{0,1,0,0},
                  {0,0,1,0}, {0,0,0,1}}
```

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

Puntatori e Vettori MULTIDIMENSIONALI

Anche un vettore multidimensionale è ***implementato in C come un puntatore all'area di memoria*** da cui partono le ***celle contigue*** contenenti il vettore

```
int a[10][4];
```

a vettore di 10 elementi, ciascuno dei quali è un vettore a 4 interi
tipo = "puntatore a vettore di 4 interi" non "puntatore a intero"

```
int ** b; int * c;
b=a;
```

`c=a;` compila con warning

PUNTATORI E VETTORI MULTIDIMENSIONALI

Date le due definizioni

```
int a[10][4]; int *d[10];
```

la prima alloca 40 celle di ampiezza pari alla dim di un int mentre la seconda alloca 10 celle per contenere 10 puntatori a int

Uno dei vantaggi offerti dall'uso di vettori di puntatori consiste nel fatto che si possono realizzare ***righe di lunghezza variabile***

PUNTATORI E VETTORI MULTIDIMENSIONALI

```
char * mesi[] =     {"Gennaio","Febbraio",
     "Marzo","Aprile","Maggio","Giugno",
     "Luglio","Agosto", "Settembre",
    "Ottobre", "Novembre", "Dicembre"};
```

vengono create ***righe di lunghezza variabile***

Esempio: PRODOTTO MATRICI QUADRATE

Programma per il **prodotto (righe x colonne) di matrici quadrate NxN** a valori interi:

$$C[i,j] = \Sigma_{(k=1..N)} A[i][k]*B[k][j]$$

```
#include <stdio.h>
#define N 2
typedef int Matrici[N][N];

int main(){
int Somma,i,j,k;
Matrici A,B,C;
/* inizializzazione di A e B */
  for (i=0; i<N; i++)
      for (j=0; j<N; j++)
            scanf("%d",&A[i][j]);
  for (i=0; i<N; i++)
      for (j=0; j<N; j++)
            scanf("%d",&B[i][j]);
```

Esempio: PRODOTTO MATRICI QUADRATE

```
/* prodotto matriciale */
  for (i=0; i<N; i++)
      for (j=0; j<N; j++){
            Somma=0;
            for (k=0; k<N; k++)
                  Somma=Somma+A[i][k]*B[k][j];
          C[i][j]=Somma;}

/* stampa */
  for (i=0; i<N; i++) {
      for (j=0; j<N; j++)
            printf("%d\t",C[i][j]);
            printf("\n"); }
}
```

PASSAGGIO DI PARAMETRI

Nel caso di passaggio come parametro di un vettore bidimensionale a una funzione, nel prototipo della funzione ***va dichiarato necessariamente il numero delle colonne*** (ovvero la dimensione della riga)

Ciò è essenziale perché il compilatore sappia come accedere al vettore in memoria

PASSAGGIO DI PARAMETRI

Esempio: se si vuole passare alla funzione **`f()`** la matrice **`par`** occorre scrivere all'atto della definizione della funzione:

`f(float par[20][30],…)` oppure

`f(float par[][30],…)`

perché il numero di righe è irrilevante ai fini dell'aritmetica dei puntatori su **`par`**

In generale, come già detto, ***soltanto la prima dimensione di un vettore multidimensionale può non essere specificata***

Esempio: PRODOTTO MATRICI QUADRATE

```
#include <stdio.h>
#define N 2
typedef int Matrici[N][N];

void prodottoMatrici(int X[][N], int Y[][N],
                     int Z[][N]) {
int Somma,i,j,k;
for (i=0; i<N; i++)
      for (j=0; j<N; j++){
            Somma=0;
            for (k=0; k<N; k++)
                  Somma=Somma+X[i][k]*Y[k][j];
            Z[i][j]=Somma;}
}
```

Esempio: PRODOTTO MATRICI QUADRATE

```
int main(void){
int Somma,i,j,k;
Matrici A,B,C;

for (i=0; i<N; i++) /* inizializ. di A e B */
      for (j=0; j<N; j++)
            scanf("%d",&A[i][j]);
for (i=0; i<N; i++)
      for (j=0; j<N; j++)
            scanf("%d",&B[i][j]);

prodottoMatrici(A,B,C); //in C verrà caricato il
  risultato del prodotto

for (i=0; i<N; i++) {          /* stampa */
      for (j=0; j<N; j++)
            printf("%d\t",C[i][j]);
            printf("\n"); }}
```

Ripasso sugli array e sui puntatori 2

Materiale tratto da:
http://www.dis.uniroma1.it/~bloisi/didattica/comunicazioniElettronica1011/lezioni/7.3-esercizi-array.pdf

Puntatori e array

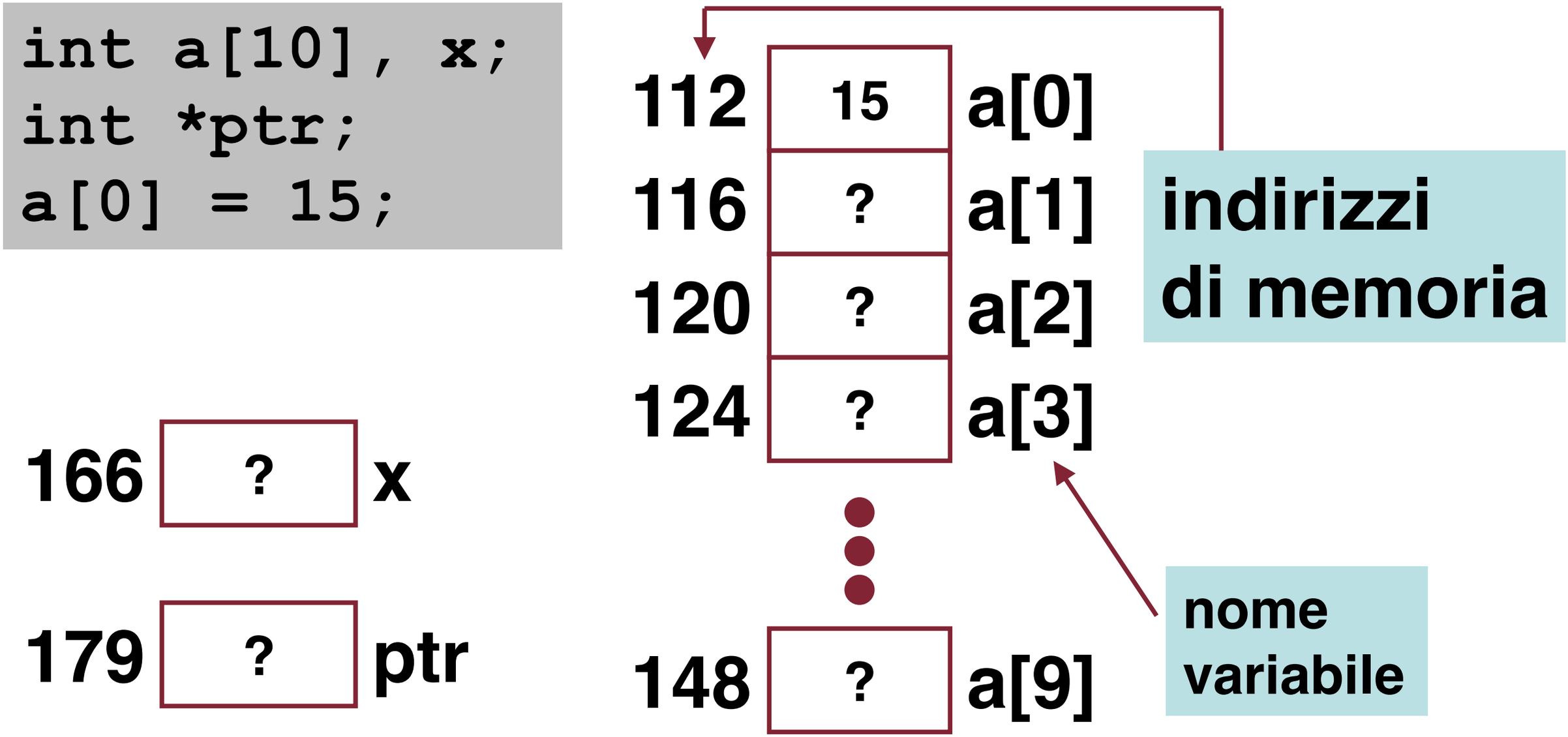
Un'array di elementi può essere pensato come disposto in un insieme di locazioni di memoria consecutive.

Consideriamo il seguente esempio:

```
int a[10], x;
int *ptr;
a[0] = 15;
ptr = &a[0]; /* ptr punta
                all'indirizzo di a[0] */
x = *ptr; /* x = contenuto di ptr (in
             questo caso, a[0]) */
```

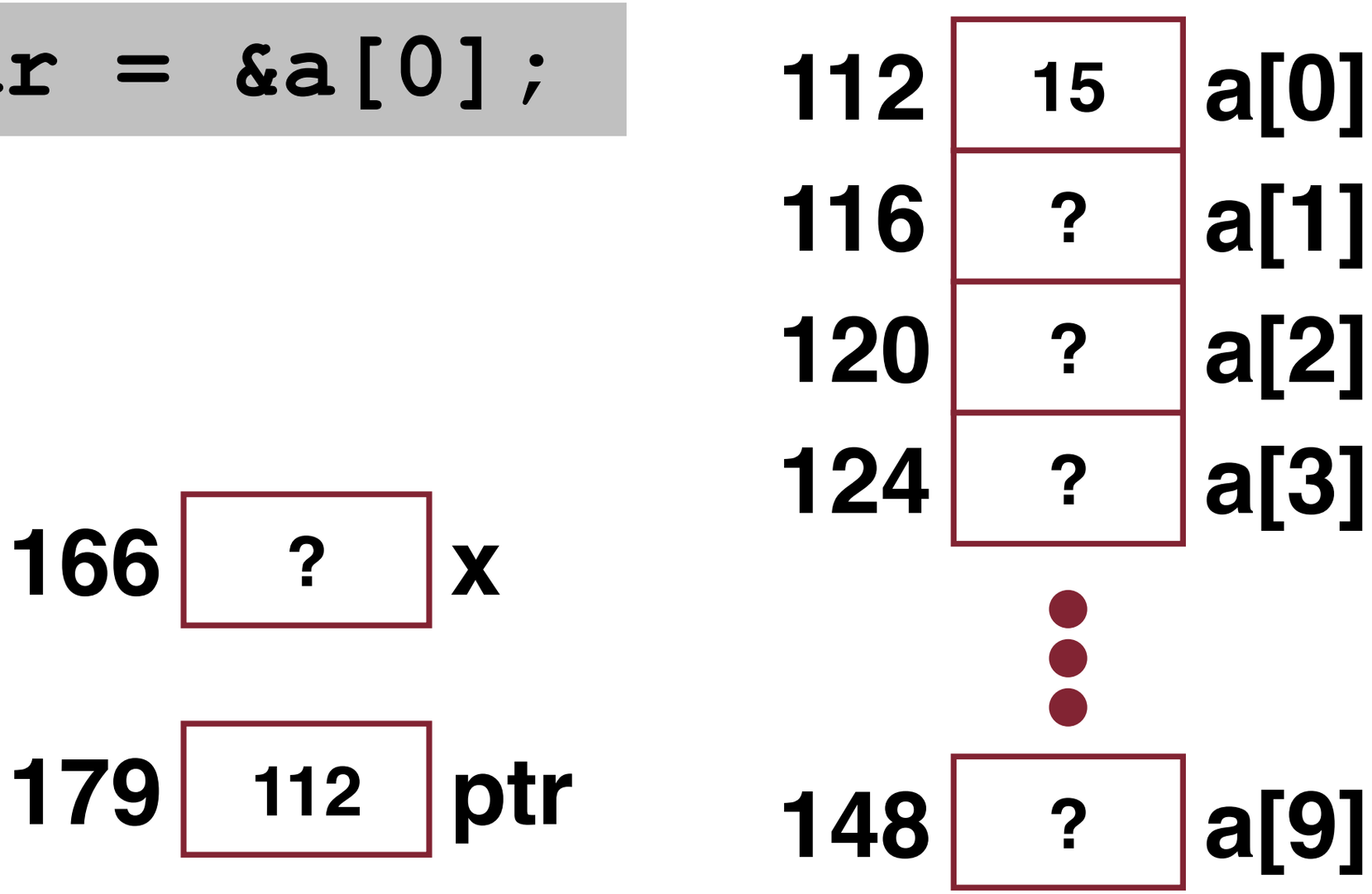
Cosa accade idealmente in memoria

```
int a[10], x;
int *ptr;
a[0] = 15;
```



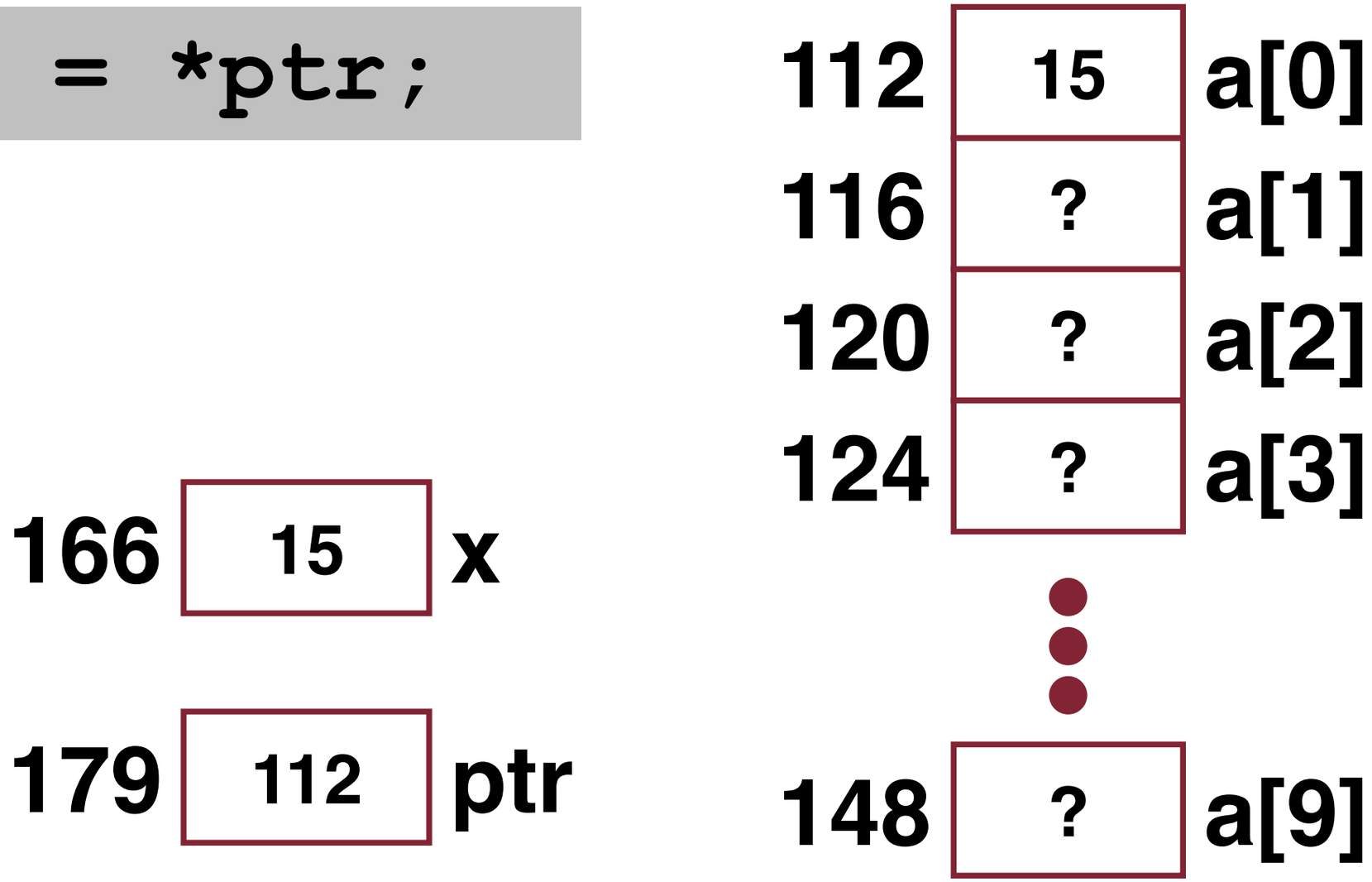
Cosa accade idealmente in memoria

```
ptr = &a[0];
```



Cosa accade idealmente in memoria

```
x = *ptr;
```



Aritmetica dei puntatori

Possiamo incrementare `ptr` con successive istruzioni del tipo

```
++ptr
```

ma potremo anche avere

```
(ptr + i)
```

che è equivalente ad `a[i]`,

con i = 0, 1, 2, 3,..., 9 .

Accesso ad array tramite puntatori

Esempio

```
*(ptr + i) = 30;
```

Attenzione: in C non c'è alcun controllo sul superamento dei limiti di un array, perciò è possibile oltrepassare la memoria prevista per un array e sovrascrivere altri blocchi di memoria.

Esercizio

Si scriva un programma che manipoli un array `c` di `char` tramite un puntatore `c_ptr`.
L'array deve avere valore iniziale

```
['L' 'U' 'C' 'A']
```

e applicando le nozioni di aritmetica dei puntatori a `c_ptr` si vuole trasformarlo in

```
['M' 'I' 'N' 'O']
```

Soluzione

```
#include <stdio.h>
int main() {
	char c[4] = {'L', 'U', 'C', 'A'};
	int i;
	for(i = 0; i < 4; i++) {
		printf("%c\n", c[i]);
	}
	printf("\n");
	char* c_ptr = c; //equivalente a  char* c_ptr = &c[0];
	*c_ptr = 'M';
	*(++c_ptr) = 'I';
	*(c_ptr + 1) = 'N';
	*(c_ptr + 2) = 'O';
	for(i = 0; i < 4; i++) {
		printf("%c\n", c[i]);
	}
	return 0;
}
```

Rapporto tra array puntatori

In C esiste un legame stretto tra array e puntatori.

Ad esempio se `ptr` è un puntatore a `int` e `a` è un array di `int` è possibile scrivere

```
ptr = a;
```

invece di

```
ptr = &a[0];
```

Rapporto tra array puntatori

`a[i]` può essere scritto come `*(a+i)`

cioè

```
&a[i] ≡ a + i
```

Inoltre, si può applicare l'operatore `[]` ad un puntatore, ottenendo il seguente significato

```
ptr[i] ≡ *(ptr+i)
```

Esercizio

Si scriva una funzione in linguaggio C che prenda in ingresso un array di 3 interi `a` e restituisca un puntatore ad un array creato dinamicamente contenente i valori di `a` posizionati in ordine inverso.

Esempio

Se `a = [1, 2, 3]` si vuole ottenere un puntatore ad un array creato dinamicamente pari a `[3, 2, 1]`

Soluzione

```
int* f(int a[3]) {
        int *b = (int*)malloc(3*sizeof(int));
        if(b == NULL) {
                return NULL;
        }
        else {
                int i;
                for(i = 0; i < 3; i++) {
                        b[i] = a[2-i];
                }
                return b;
        }
}
```

Note

1. **Poiché la dimensione di `a` è nota a priori (espressa esplicitamente nella specifica), il prototipo della funzione sarà**

   ```
   int* f(int a[3])
   ```

   con la lunghezza del parametro array in ingresso specificata (3)

2. **Nel caso in cui la `malloc` dovesse fallire restituirò un valore NULL, in modo da permettere alla funzione che chiama `f` (e.g., `main`) di poter gestire l'errore.**

Programma di prova

```
int main() {
        int d[3] = {1, 2, 3};
        int i;
        for(i = 0; i < 3; i++) {
                printf("%d ", d[i]);
        }
        printf("\n");
        int *e = f(d);
        if(e != NULL) {
                for(i = 0; i < 3; i++)
                        printf("%d ", e[i]);
                free(e); //libero la memoria allocata in f
        }
        else printf("Errore in f.\n");
        return 0;
}
```

Differenze tra array e puntatori

Puntatori e array sono diversi:

- **un puntatore è una variabile, per cui possiamo scrivere:**
  `ptr = a` **ed anche** `ptr++`

- **un array non è una variabile, quindi:**
  `a = ptr` **e** `a++` **sono istruzioni NON VALIDE**

Allocazione dinamica di matrici

In C è possibile creare una matrice allocando memoria dinamicamente. A tale scopo utilizzeremo un puntatore a puntatore.

Esempio

```
int **m;
```

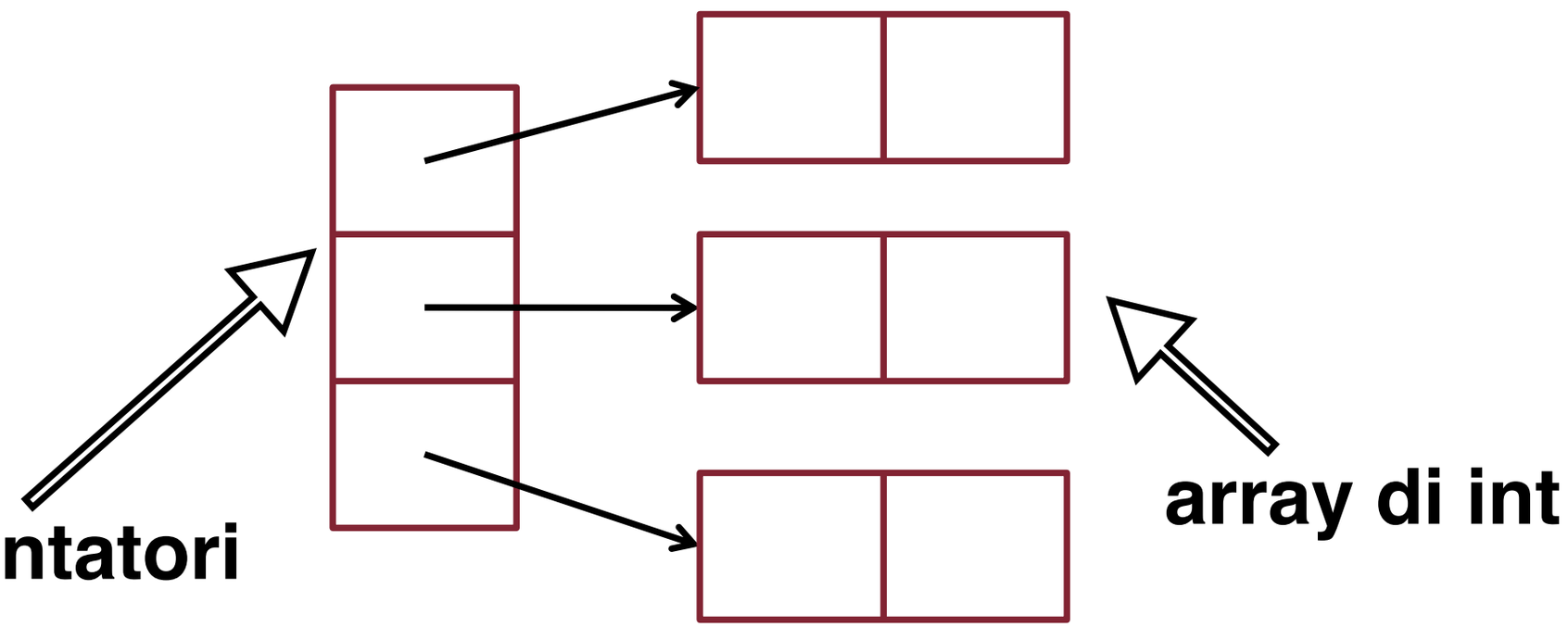
L'idea è quella di creare dinamicamente un array di puntatori che rappresenti le righe della matrice.
Ogni riga punterà ad una colonna, rappresentata da un array allocato dinamicamente.

Allocazione dinamica di matrici

```
int i;
int** m;
m = (int **)malloc(ROWS*sizeof(int*));
for(i = 0; i < ROWS; i++) {
      m[i] = (int *)malloc(COLUMNS*sizeof(int));
}
```

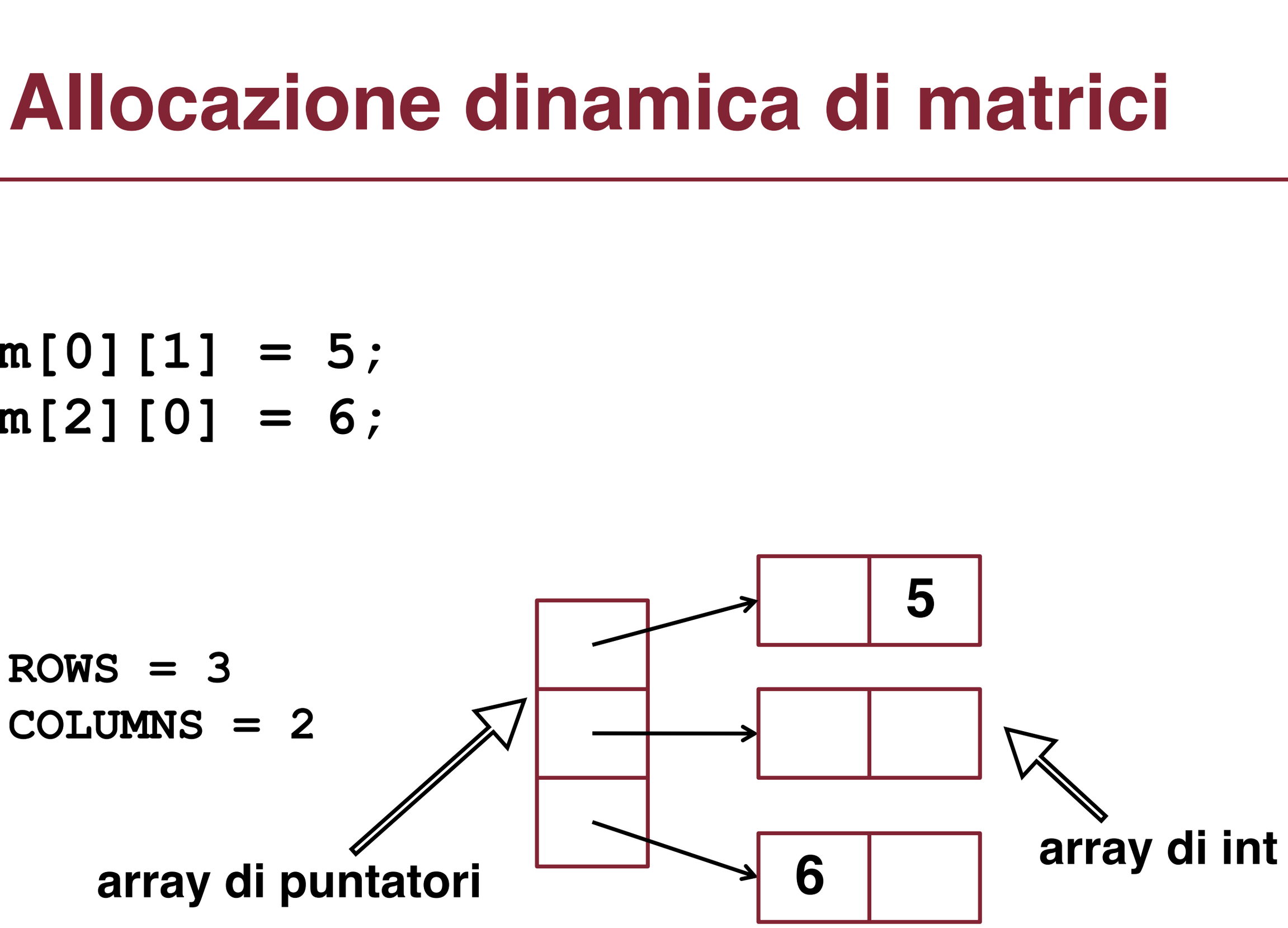
ROWS = 3

COLUMNS = 2



Allocazione dinamica di matrici

```
m[0][1] = 5;
m[2][0] = 6;
```



Esercizio

Si scriva una funzione `creaMat` che presi in ingresso due interi `r` e `c`, restituisca una matrice creata dinamicamente.

Si realizzi un programma di prova che utilizzi `creaMat` per creare la matrice

```
 3  6  8  9
12 54  6 89
```

stampandone il contenuto

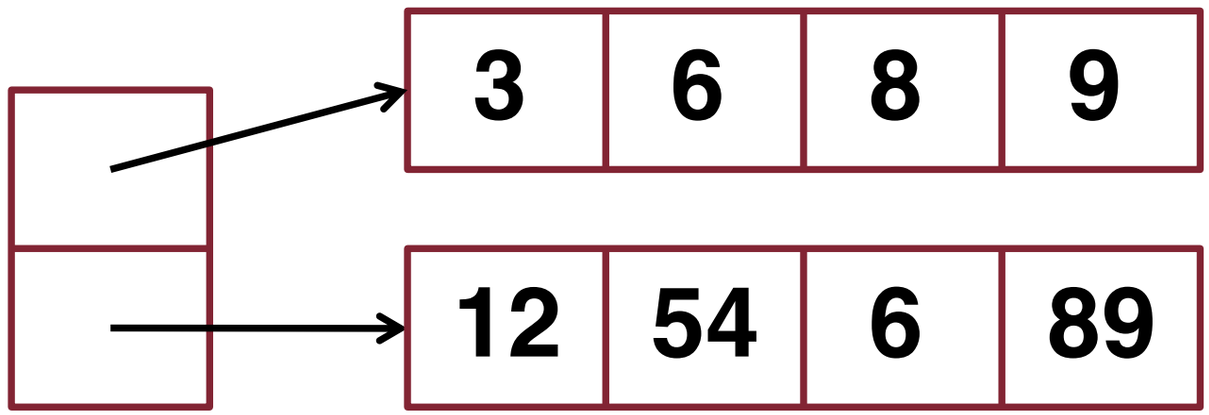
Funzione `creaMat`

```
int** creaMat(int r, int c) {
   int i;
   int** m;
   m = (int **)malloc(r * sizeof(int*));
   for(i = 0; i < r; i++) {
      m[i] = (int *)malloc(c * sizeof(int));
   }
   return m;
}
```

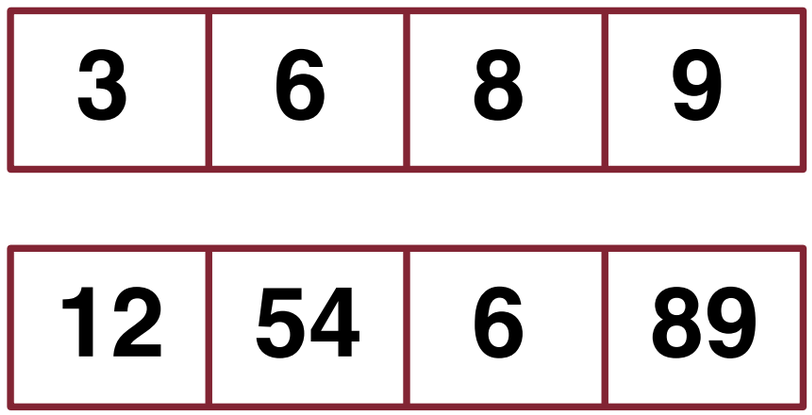
Main

```
int main() {
   int r = 2, c = 4;
   int i, j;
   int **m = creaMat(r, c);
   m[0][0] = 3; m[0][1] = 6; m[0][2] = 8; m[0][3] = 9;
   m[1][0] = 12; m[1][1] = 54; m[1][2] = 6; m[1][3] = 89;
   for(i = 0; i < r; i++) {
        for(j = 0; j < c; j++) {
          printf("%d ", m[i][j]);
        }
        printf("\n");
   }
   return 0;
}
```

Deallocazione di una matrice allocata dinamicamente

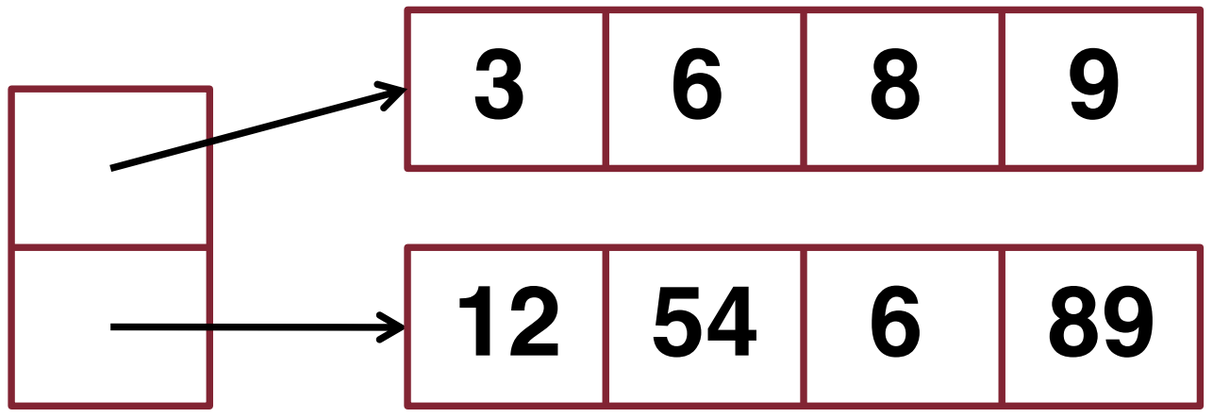


Cosa accade se si esegue `free(m)`?

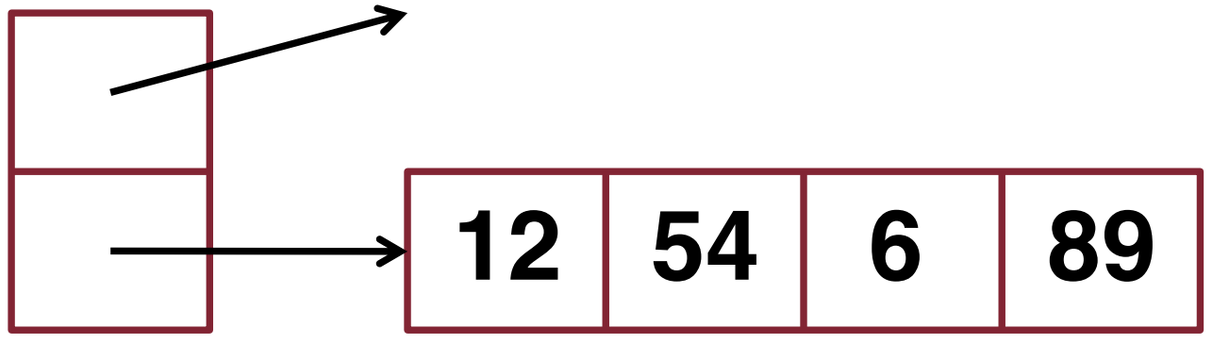


Deallocazione di una matrice allocata dinamicamente

Prima di eseguire `free(m)` è necessario effettuare una free su ogni riga.




```
free(m[0]);
```



Deallocazione di una matrice allocata dinamicamente

Deallocazione corretta

```
for(i = 0; i < r; i++) {
        free(m[i]);
}
free(m);
```

Main

```
int main() {
	int r = 2, c = 4;
	int i, j;
	int **m = creaMat(r, c);
	m[0][0] = 3; m[0][1] = 6; m[0][2] = 8; m[0][3] = 9;
	m[1][0] = 12; m[1][1] = 54; m[1][2] = 6; m[1][3] = 89;
	for(i = 0; i < r; i++) {
		for(j = 0; j < c; j++) {
			printf("%d ", m[i][j]);
		}
		printf("\n");
	}
	for(i = 0; i < r; i++) {
		free(m[i]);
	}
	free(m);
	return 0;
}
```

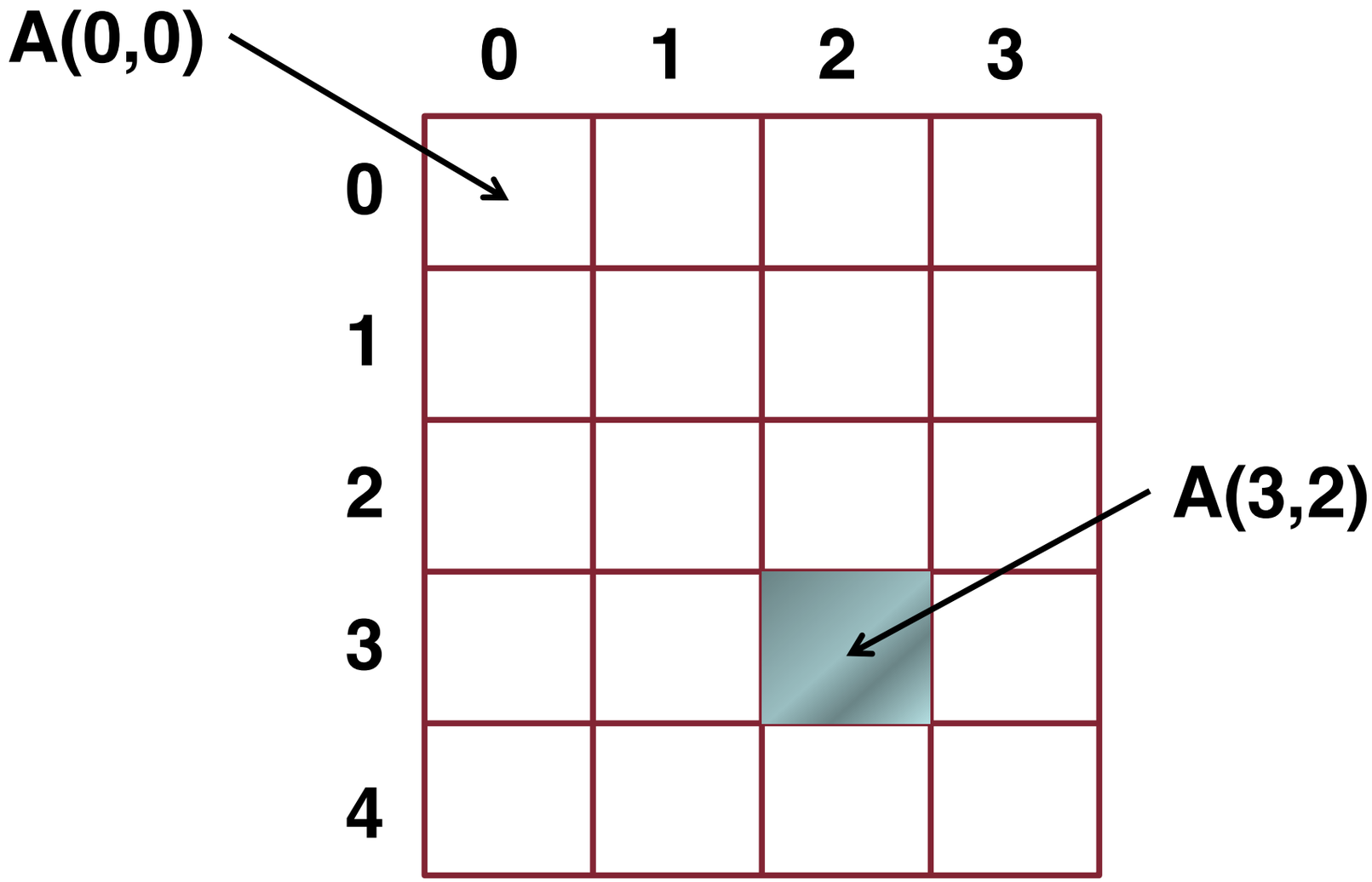
Accesso ad una matrice tramite aritmetica dei puntatori

Sia A una matrice 5x4 allocata dinamicamente.
Se si vuole accedere all'elemento A[3][2] è possibile utilizzare l'aritmetica dei puntatori nel seguente modo:

***(*(A+3)+2)**

(A+3) è il quarto puntatore dell'array che rappresenta le righe
***(A+3) è l'indirizzo del primo elemento della quarta riga**
***(A+3)+2 è l'indirizzo del terzo elemento della quarta riga**
***(*(A+3)+2) è il contenuto della cella puntata dal puntatore al terzo elemento della quarta riga**

Accesso ad una matrice tramite aritmetica dei puntatori



RICORSIONE

Divide et impera

- Metodo di approccio ai problemi che consiste nel dividere il problema dato in problemi più semplici
- I risultati ottenuti risolvendo i problemi più semplici vengono combinati insieme per costituire la soluzione del problema originale
- Generalmente, quando la semplificazione del problema consiste essenzialmente nella semplificazione dei DATI da elaborare (ad es. la riduzione della dimensione del vettore

La ricorsione (1)

- Una funzione è detta **ricorsiva** se chiama se stessa
- Se due funzioni si chiamano l'un l'altra, sono dette **mutuamente ricorsive**
- La funzione ricorsiva sa risolvere direttamente solo casi particolari di un problema detti **casi di base**: se viene invocata passandole dei dati che costituiscono uno dei casi di base, allora restituisce un risultato
- Se invece viene chiamata passandole dei dati che NON costituiscono uno dei casi di base, allora chiama se stessa (passo ricorsivo) passando dei DATI semplificati/ridotti

La ricorsione (II)

- Ad ogni chiamata si semplificano/riducono i dati, così ad un certo punto si arriva ad uno dei casi di base
- Quando la funzione chiama se stessa, sospende la sua esecuzione per eseguire la nuova chiamata
- L'esecuzione riprende quando la chiamata interna a se stessa termina
- La sequenza di chiamate ricorsive termina quando quella più interna (annidata) incontra uno dei casi di base
- Ogni chiamata alloca sullo stack (in stack frame diversi) nuove istanze dei parametri e delle variabili locali (non static)

Esempio: il fattoriale

Funzione ricorsiva che calcola il fattoriale di un numero n

Premessa (definizione ricorsiva):

se $n \leq 1$ → $n! = 1$

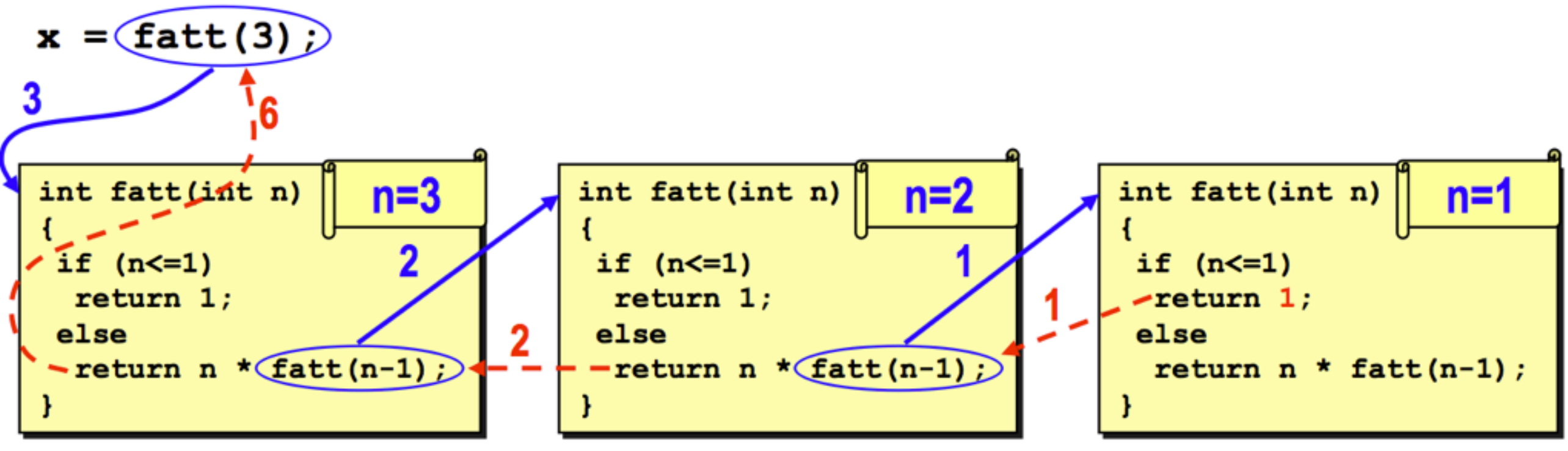
se $n > 1$ → $n! = n * (n-1)!$

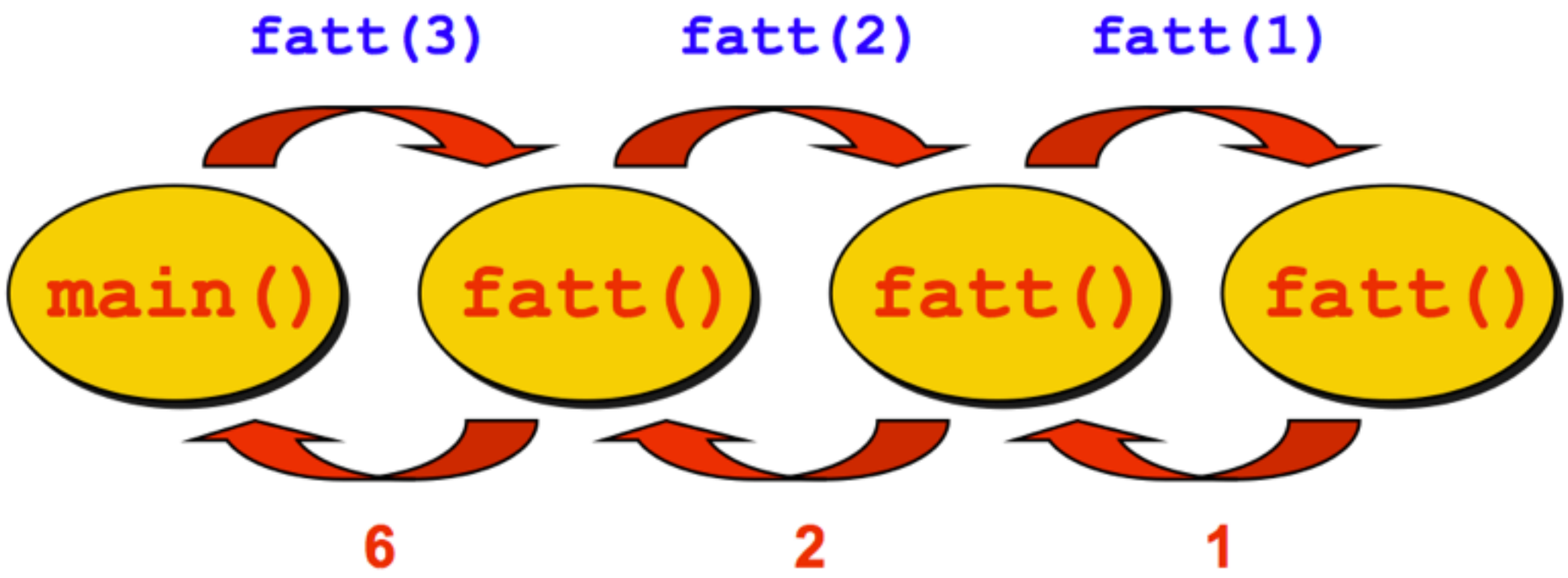
```
int fatt(int n)
{
   if (n<=1)
      return 1;
   else
      return n * fatt(n-1);
}
```

→ *Caso di base*

Semplificazione dei dati del problema

Esempio: il fattoriale - passo per passo





Ricorsione - Esercizio 1

- Scrivere una funziona ricorsiva che calcoli ricorsivamente la somma di tutti i numeri compresi tra 0 ed x
- Il prototipo della funzione è: int ric(int x)

```
int ric(int x) {



}
```

Ricorsione - Esercizio 1

- Scrivere una funziona ricorsiva che calcoli ricorsivamente la somma di tutti i numeri compresi tra 0 ed x
- Il prototipo della funzione è: int ric(int x)

```
int ric(int x) {
  if (x == 0)
      return 0;
  else
      return x + ric(x-1);
}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{

}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{

}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{ if (n==1) return a - 1/a;
  else return a - n/a + f(a, n-1);
}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{ if (n==1) return a - 1/a;
  else return a - n/a + f(a, n-1);
}
```

```
double f(double a, int n)
{ int i=1;
  double sum=0;
  while(i<=n)
       {sum = sum + a - i/a;
        i++;}
  return  sum;
}
```

Qualche considerazione

- L'apertura delle chiamate ricorsive semplifica il problema, ma non calcola ancora nulla
- Il valore restituito dalle funzioni viene utilizzato per calcolare il valore finale man mano che si chiudono le chiamate ricorsive: ogni chiamata genera valori intermedi a partire dalla fine
- Nella ricorsione vera e propria non c'è un mero passaggio di un risultato calcolato nella chiamata più interna a quelle più esterne, ossia le return non si limitano a passare indietro invariato un valore, ma c'è un'elaborazione intermedia

Quando utilizzare la ricorsione

- PRO
  Spesso la ricorsione permette di risolvere un problema anche molto complesso con poche linee di codice
- CONTRO
  La *ricorsione è poco efficiente* perché richiama molte volte una funzione e questo:
  - richiede tempo per la gestione dello stack (allocare e passare i parametri, salvare l'indirizzo di ritorno, e i valori di alcuni registri della CPU)
  - consuma molta memoria (alloca un nuovo stack frame ad ogni chiamata, definendo una nuova ulteriore istanza delle variabili locali non `static` e dei parametri ogni volta)

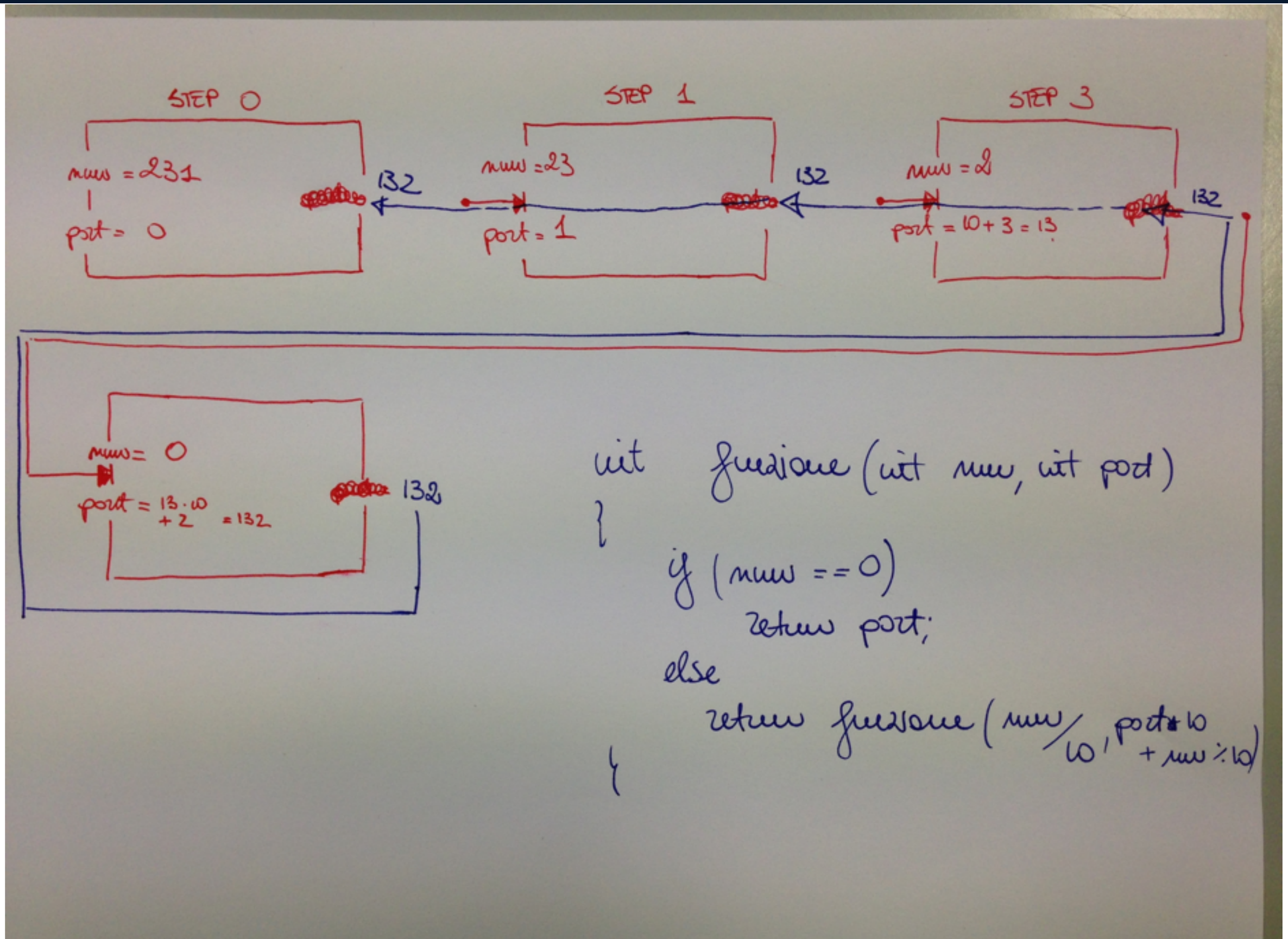
Quando utilizzare la ricorsione

- CONSIDERAZIONE
  Qualsiasi problema ricorsivo può essere risolto in modo non ricorsivo (ossia iterativo), ma la soluzione iterativa potrebbe non essere facile da individuare oppure essere molto più complessa
- CONCLUSIONE
  Quando non ci sono particolari problemi di efficienza e/o memoria, l'approccio ricorsivo è in genere da preferire se:
  - è più intuitivo di quello iterativo
  - la soluzione iterativa non è evidente o agevole

Ricorsione - Esercizio III

- Analizzare il comportamento della seguente funzione ricorsiva mostrando l'andamento delle variabili, considerando i seguenti input:
  - *funzione(231,0)*
- Dire quale funzione espleta

```
int mia_funzione(int num, int part) {
     if (num == 0)
          return part;
     else {
          return mia_funzione(num/10, part*10 + num%10);
     }
}
```




```
int funzione(int num, int port)
{
    if (num == 0)
        return port;
    else
        return funzione(num/10, port*10 + num%10);
}
```

Tutte il materiale sarà disponibile sul mio sito internet!

alessandronacci.it