IEIM 2017-2018

Esercitazione IX

"Puntatori, Enumerazione e Ricorsione"

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

Matrici e funzioni

Esercizio 1

MATRICI E FUNZIONI

Il passaggio di una matrice ad una funzione
e' sempre per indirizzo e mai per copia.

Esercizio 1: Matrici e Funzioni

```c
#include <stdio.h>

#define R 3
#define C 3

void foo1(int*);
void foo2(int mat[R][C]);
void foo3(int mat[][C]);

int main()
{
      int mat[R][C];
      foo1(mat);
      foo2(mat);
      foo3(mat);
}
```

```c
void foo1(int *mat)
{
      int i, j;
      for(i=0;i<R;i++)
            for(j=0;j<C;j++)
                  *(mat+i*C+j) = (i+j) * 2;
}
```

Puntatore ad intero!

Calcolo manuale dello spiazzamento

Valore contenuto all'indirizzo tra parentesi

```c
*(mat+i*C+j)
```

INDIRIZZO

Esercizio 1: Matrici e Funzioni

```c
#include <stdio.h>

#define R 3
#define C 3

void foo1(int*);
void foo2(int mat[R][C]);
void foo3(int mat[][C]);

int main()
{
      int mat[R][C];
      foo1(mat);
      foo2(mat);
      foo3(mat);
}
```

```c
void foo2(int mat[R][C])
{
      int i, j;
      for(i=0;i<R;i++)
            for(j=0;j<C;j++)
                  mat[i][j] = (i+j) * 3;
}
```

Esplicito che il puntatore punta al primo elemento di una matrice RxC

Accesso comodo alla matrice!

Esercizio 1: Matrici e Funzioni

```c
#include <stdio.h>

#define R 3
#define C 3

void foo1(int*);
void foo2(int mat[R][C]);
void foo3(int mat[][C]);

int main()
{
      int mat[R][C];
      foo1(mat);
      foo2(mat);
      foo3(mat);
}
```

Il valore di mat cambia

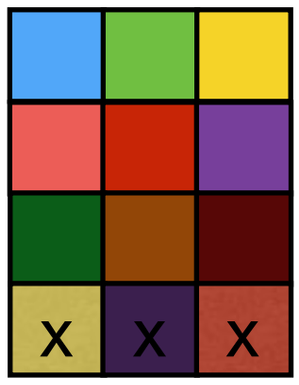
Il valore di mat cambia

Il valore di mat cambia

Esplicito che il puntatore punta al primo elemento di una matrice con C colonne

```c
void foo3(int mat[][C])
{
      int i, j;
      for(i=0;i<R;i++)
            for(j=0;j<C;j++)
                  mat[i][j] = (i+j) * 4;
}
```

Accesso comodo alla matrice!



NOTA BENE

Ai fini del calcolo dello spiazzamento il numero di righe non è essenziale!

L’enumerazione

Esercizio 3

Il parco macchine

- Quale è l'ouput del seguente codice?

```
#include <stdio.h>

   typedef enum{bianco,azzurro_chiaro,giallo,rosso,verde_scuro,rosa,
      azzurro_scuro,verde_chiaro,nero,marrone} colore;

int main(){



   int auto_codice[3];          //ATTENZIONE QUI!
   colore auto_colore[3];       // tipo_di_dato nome_varibile[]
   int i;

   auto_codice[0] = 1000;
   auto_colore[0] = giallo;

   auto_codice[1] = 4000;
   auto_colore[1] = rosa;

   auto_codice[2] = 8000;
   auto_colore[2] = nero;



   for (i = 0; i < 3; i++ )
      printf("L'auto con codice %d e' di colore %d\n",
         auto_codice[i], auto_colore[i]);

   return 0;
}
```

```
Terminal — bash — 80x24
> ./es3
> L'auto con codice 1000 e' di colore 2
> L'auto con codice 4000 e' di colore 5
> L'auto con codice 8000 e' di colore 8
```

MA QUINDI A COSA

Solo a semplificare la vita al programmatore! Non ti devi ricordare la corrispondenza numerica!a

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 (I)

- 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):

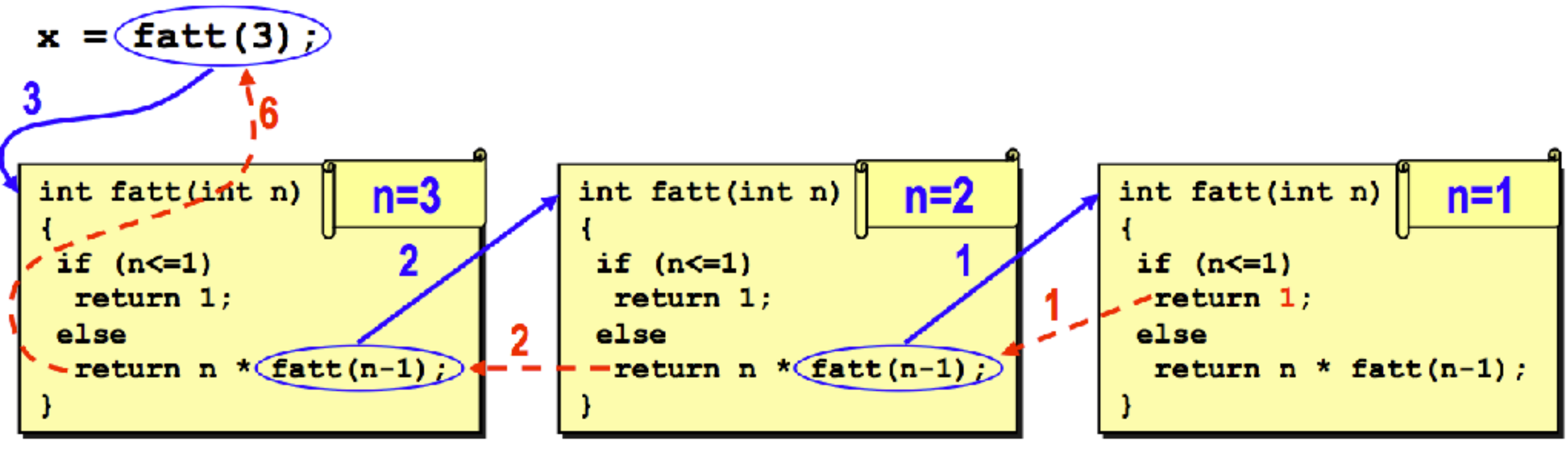
se $n \leq 1$ → $n! = 1$

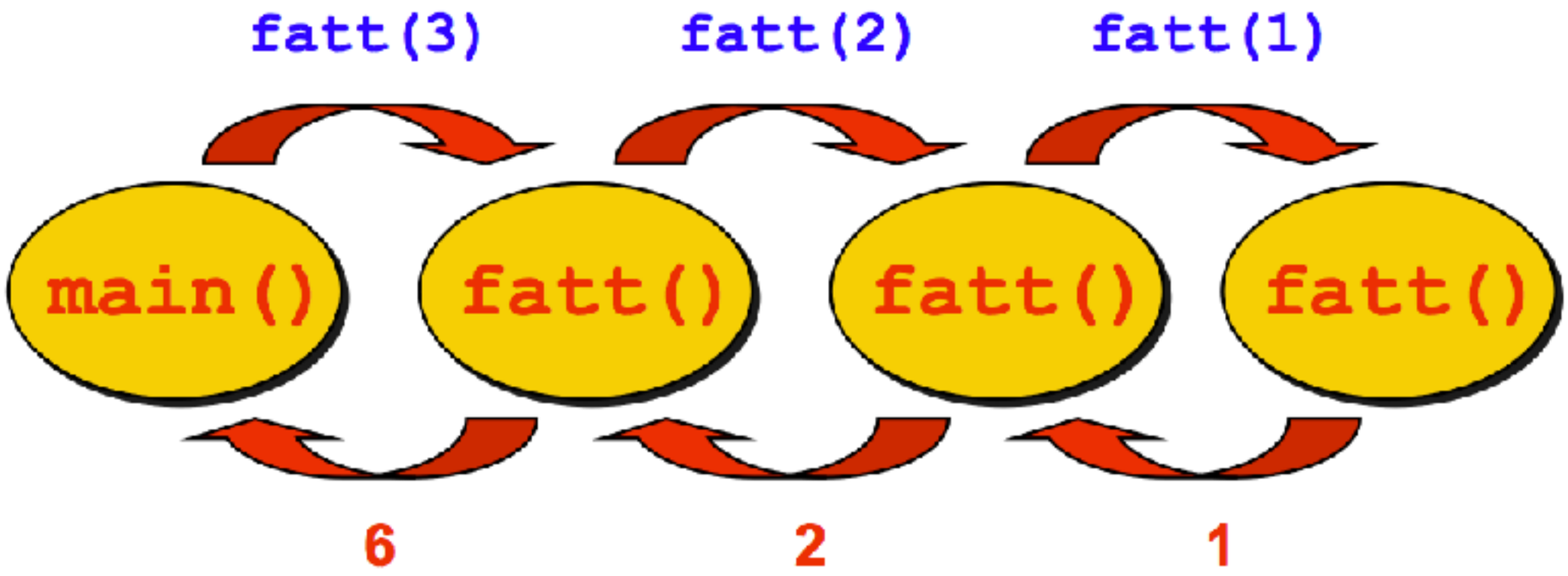
se $n > 1$ → $n! = n * (n-1)!$

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

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 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)
{

}
```

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

Tutte il materiale sarà disponibile sul mio sito internet!

alessandronacci.it