


LA RICORSIONE

- Una funzione matematica è definita ***ricorsivamente*** quando nella sua definizione compare un riferimento a se stessa
- La ricorsione consiste nella possibilità di ***definire una funzione in termini di se stessa***
- È basata sul *principio di induzione* matematica:
 - se una proprietà P vale per $n=n_0$  CASO BASE
 - e si può provare che, ***assumendola valida per n*** , allora vale per $n+1$allora P vale per ogni $n \geq n_0$

LA RICORSIONE

Operativamente, risolvere un problema con un ***approccio ricorsivo*** comporta

- di identificare un “**caso base**”, con soluzione nota
- di riuscire a **esprimere la soluzione al caso generico n in termini dello stesso problema in uno o più casi più semplici** ($n-1$, $n-2$, etc.)

LA RICORSIONE: ESEMPIO

Servitore & Cliente:

```
int fact(int n) {  
    if (n<=0) return 1;  
    else return n*fact(n-1);  
}  
  
int main() {  
    int fz, z = 5;  
    fz = fact(z-2);  
}
```

Si valuta l'espressione che costituisce il parametro attuale (nell'environment del main) e si trasmette alla funzione fact() una copia del valore così ottenuto (3)

fact(3) effettuerà poi analogamente una nuova chiamata di funzione fact(2)

LA RICORSIONE: ESEMPIO

Servitore & Cliente:

```
int fact(int n) {  
    if (n<=0) return 1;  
    else return n*fact(n-1);  
}
```

```
int main() {  
    int fz, z = 5;  
    fz = fact(z-2);  
}
```

Analogamente, `fact(2)` effettua una nuova chiamata di funzione. ***$n-1$*** nell'environment di `fact()` vale **1** quindi viene chiamata `fact(1)`

E ancora, analogamente, per `fact(0)`

LA RICORSIONE: ESEMPIO

Servitore & Cliente:

```
int fact(int n) {  
    if (n<=0) return 1;  
    else return n*fact(n-1);  
}
```

```
int main() {  
    int fz, z = 5;  
    fz = fact(z-2);  
}
```

Il nuovo servitore lega il parametro n a 0. *La condizione $n \leq 0$ è vera e la funzione `fact(0)` torna come risultato 1 e termina*

LA RICORSIONE: ESEMPIO

Servitore & Cliente:

```
int fact(int n) {  
    if (n<=0) return 1;  
    else return n*fact(n-1);  
}
```

```
int main() {  
    int fz, z = 5;  
    fz = fact(z-2);  
}
```

*Il controllo torna al servitore precedente fact(1) che può valutare l'espressione $n * 1$ ottenendo come risultato 1 e terminando*

E analogamente per fact(2) e fact(3)

LA RICORSIONE: ESEMPIO

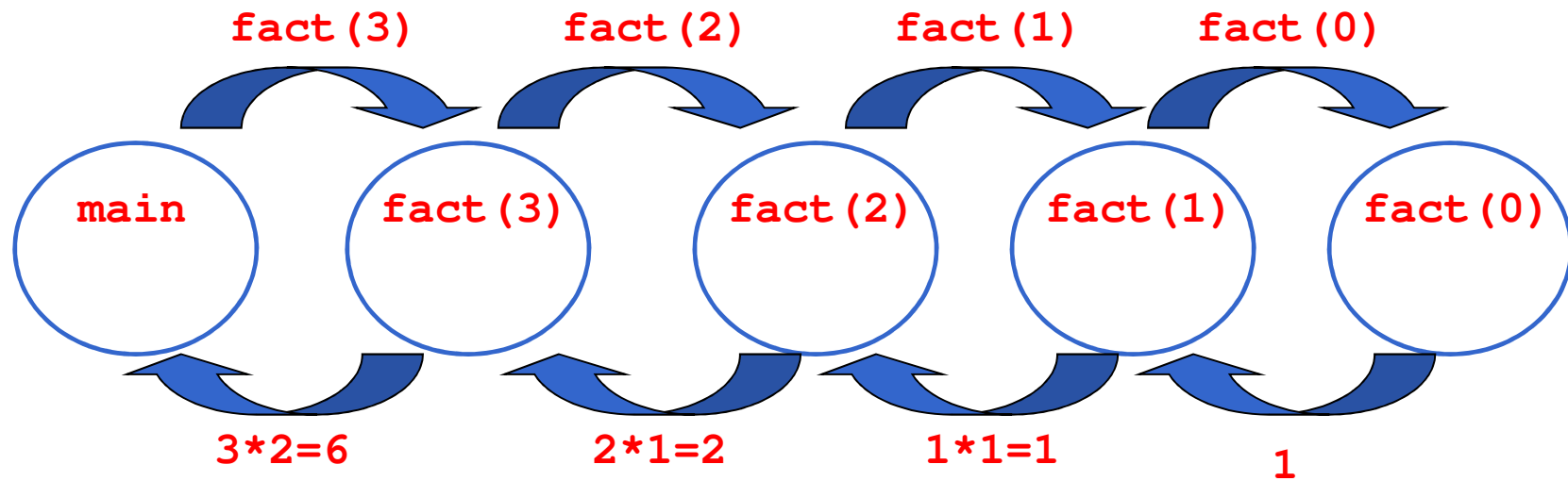
Servitore & Cliente:

```
int fact(int n) {  
    if (n<=0) return 1;  
    else return n*fact(n-1);  
}
```

```
int main() {  
    int fz, z = 5;  
    fz = fact(z-2);  
}
```

*il controllo passa infine al main
che assegna a fz il valore 6*

LA RICORSIONE: ESEMPIO



main **fact(3) = 3 * fact(2) = 2 * fact(1) = 1 * fact(0)**

Cliente di
fact(3)

Cliente di
fact(2)
Servitore
del main

Cliente di
fact(1)
Servitore
di fact(3)

Cliente di
fact(0)
Servitore
di fact(2)

Servitore
di fact(1)



LA RICORSIONE: ESEMPIO

Problema:

calcolare la somma dei primi N interi

Specifica:

Considera la somma $1+2+3+\dots+(N-1)+N$ come composta di due termini:

- $(1+2+3+\dots+(N-1))$ 
- N  *Valore noto*

Il primo termine non è altro che lo stesso problema in un caso più semplice: calcolare la somma dei primi N-1 interi

Esiste un caso banale ovvio: CASO BASE

- la somma fino a 1 vale 1

LA RICORSIONE: ESEMPIO

Problema:
calcolare la somma dei primi N interi

Algoritmo ricorsivo

Se N vale 1 allora la somma vale 1

altrimenti la somma vale $N +$ il risultato della
somma dei primi $N-1$ interi

LA RICORSIONE: ESEMPIO

Problema:

calcolare la somma dei primi N interi

Codifica:

```
int sommaFinoA(int n) {  
    if (n==1) return 1;  
    else return sommaFinoA(n-1)+n;  
}
```

LA RICORSIONE: ESEMPIO

Problema:

calcolare l'N-esimo numero di Fibonacci

$$\text{fib}(n) = \begin{cases} 0, & \text{se } n=0 \\ 1, & \text{se } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{altrimenti} \end{cases}$$

LA RICORSIONE: ESEMPIO

Problema:

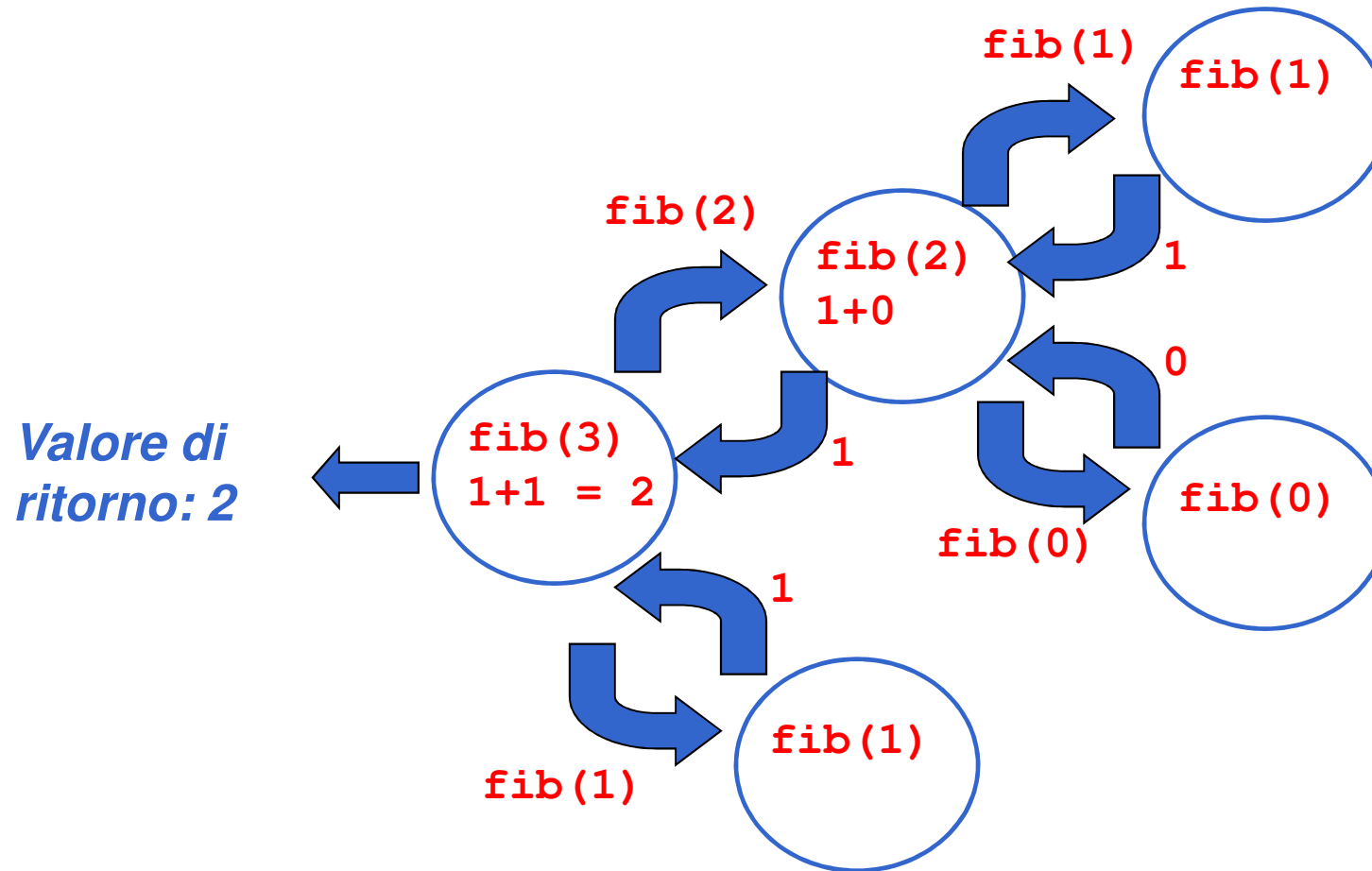
calcolare l'N-esimo numero di Fibonacci

Codifica:

```
unsigned fibonacci(unsigned n) {  
    if (n<2) return n;  
    else return fibonacci(n-1)+fibonacci(n-2);  
}
```

Ricorsione non lineare: ogni invocazione del servitore causa due nuove chiamate al servitore medesimo

RICORSIONE NON LINEARE: ESEMPIO



UNA RIFLESSIONE

Negli esempi visti finora si inizia a sintetizzare il risultato **SOLO DOPO** che si sono aperte tutte le chiamate, “*a ritroso*”, mentre le chiamate si chiudono

*Le chiamate ricorsive decompongono via via il problema, **ma non calcolano nulla***

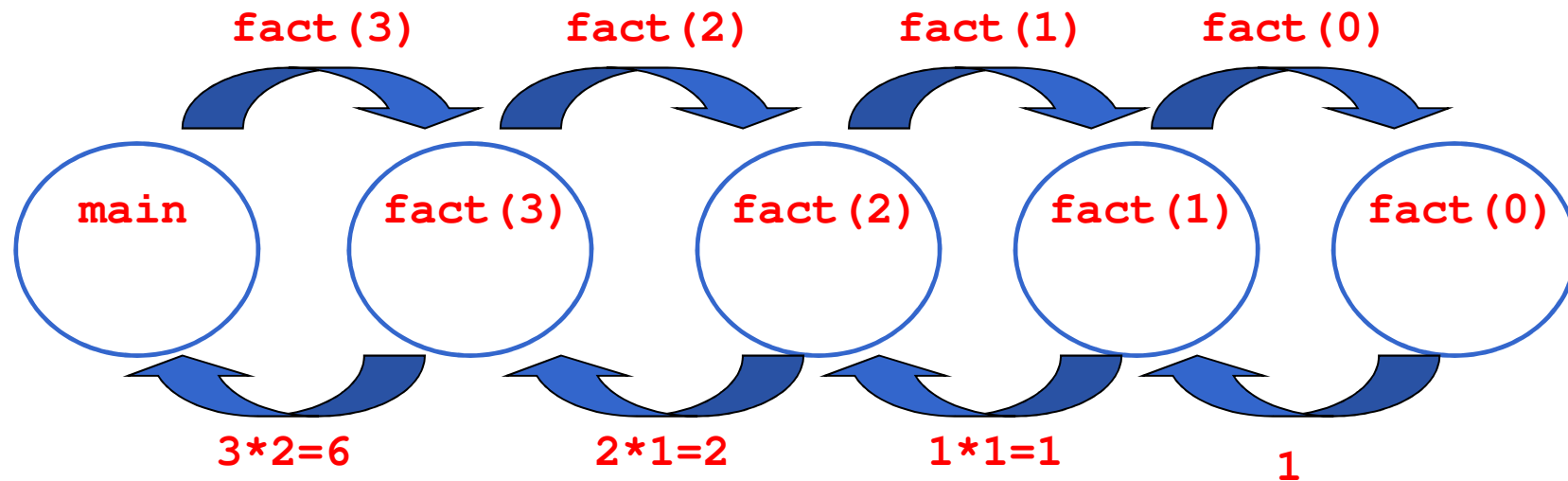
Il risultato viene sintetizzato a partire dalla fine, perché *prima occorre arrivare al caso “banale”*:

- il caso “banale” fornisce il valore di partenza
- poi si sintetizzano, “a ritroso”, i successivi risultati parziali



Processo computazionale effettivamente ricorsivo

LA RICORSIONE



PASSI:

- 1) `fact(3)` chiama `fact(2)` passandogli il controllo
- 2) `fact(2)` calcola il fattoriale di 2 e termina restituendo 2
- 3) `fact(3)` riprende il controllo ed effettua la moltiplicazione $3*2$
- 4) termina anche `fact(3)` e torna il controllo al `main`

PROCESSO COMPUTAZIONALE ITERATIVO

- In questo caso il risultato viene sintetizzato *“in avanti”*
- Ogni processo computazionale che computi “in avanti”, per accumulo, costituisce una **ITERAZIONE**, ossia è un *processo computazionale iterativo*
- La caratteristica fondamentale di un **processo computazionale ITERATIVO** è che ***a ogni passo è disponibile un risultato parziale***
 - dopo k passi, si ha a disposizione il risultato parziale relativo al caso k
 - questo ***non è vero nei processi computazionali ricorsivi***, in cui nulla è disponibile fino al caso elementare

FATTORIALE ITERATIVO

Definizione:

$$n! = 1 * 2 * 3 * \dots * n$$

Detto $v_k = 1 * 2 * 3 * \dots * k$:

$$1! = v_1 = 1$$

$$(k+1)! = v_{k+1} = (k+1) * v_k$$

$$n! = v_n$$

per $k \geq 1$

per $k = n$

FATTORIALE ITERATIVO

Costruiamo ora una funzione che calcola il fattoriale in modo iterativo

```
int fact(int n) {  
    int i=1;  
    int F=1; /*inizializzazione del fattoriale*/  
    while (i <= n)  
        { F=F*i;  
          i=i+1; }  
    return F;  
}
```

**DIFFERENZA CON LA
VERSIONE RICORSIVA: ad
ogni passo viene
accumulato un risultato
intermedio**

La variabile F accumula risultati intermedi: se $n = 3$ inizialmente $F=1$, poi al primo ciclo $F=1$, poi al secondo ciclo F assume il valore 2. Infine all'ultimo ciclo $i=3$ e F assume il valore 6

- Al primo passo F accumula il fattoriale di 1
- Al secondo passo F accumula il fattoriale di 2
- Al passo i -esimo F accumula il fattoriale di i

FUNZIONI: IL MODELLO A RUN-TIME

Ogni volta che viene invocata una funzione:

- si crea una ***nuova attivazione (istanza)*** del servitore
- viene ***allocata la memoria*** per i parametri e per le variabili locali
- si effettua il passaggio dei parametri
- si trasferisce il controllo al servitore
- si esegue il codice della funzione

IL MODELLO A RUN-TIME: ENVIRONMENT

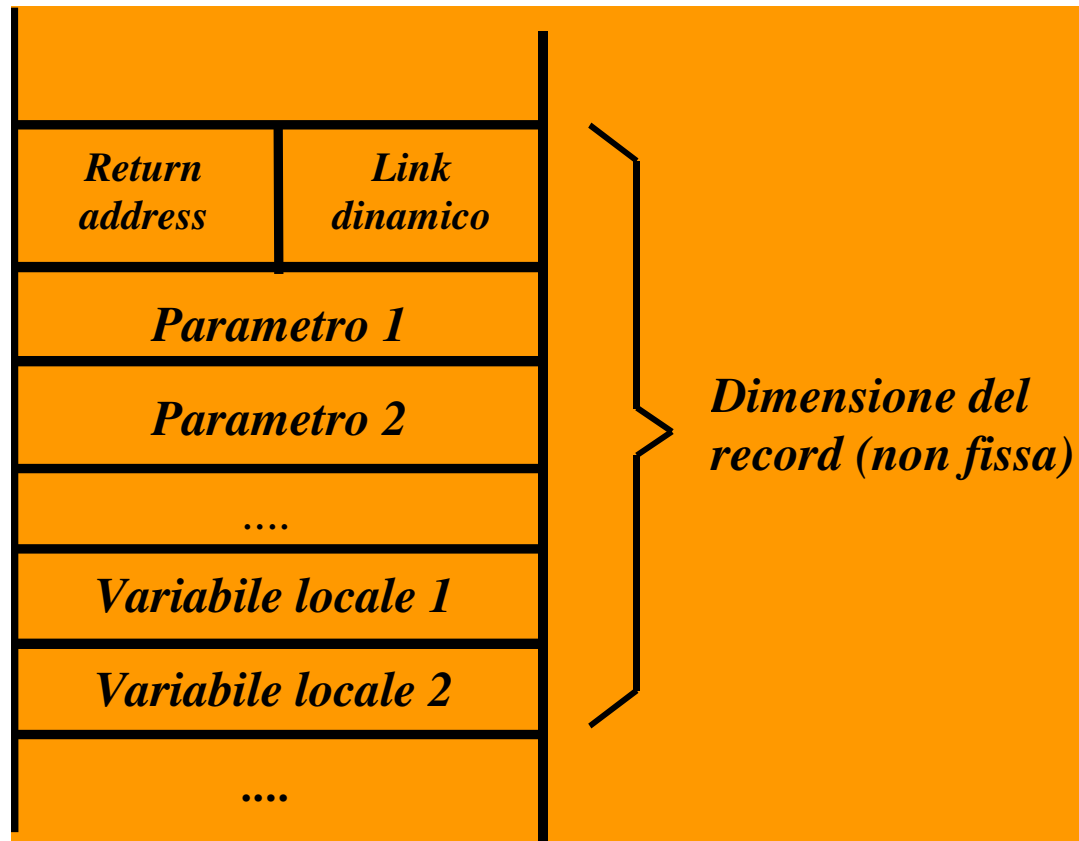
- La definizione di una funzione introduce un ***nuovo binding*** nell'environment in cui la funzione è definita
- Al momento dell'*invocazione*, viene creata una **struttura dati che contiene i *binding* dei parametri e degli identificatori definiti localmente** alla funzione detta ***RECORD DI ATTIVAZIONE***

RECORD DI ATTIVAZIONE

È il “*mondo della funzione*”: *contiene tutto ciò che ne caratterizza l’esistenza*

- i **parametri** ricevuti
- le **variabili locali**
- l’**indirizzo di ritorno (*Return Address RA*)** che indica il punto a cui tornare (nel codice del cliente) al termine della funzione, per permettere al cliente di proseguire una volta che la funzione termina
- un **collegamento al record di attivazione del cliente (*Dynamic Link DL*)**

RECORD DI ATTIVAZIONE



RECORD DI ATTIVAZIONE

- Rappresenta il “*mondo della funzione*”: *nasce e muore con essa*
 - è **creato** al momento della **invocazione** di una funzione
 - **permane** per tutto il tempo in cui la funzione è in **esecuzione**
 - è distrutto (**deallocato**) al termine dell’esecuzione della funzione stessa
- Ad **ogni chiamata** di funzione viene **creato un nuovo record**, *specifico per quella chiamata di quella funzione*
- La dimensione del record di attivazione
 - varia da una funzione all’altra
 - *per una data funzione, è fissa e calcolabile a priori*

RECORD DI ATTIVAZIONE

Funzioni che chiamano altre funzioni danno luogo a una ***sequenza di record di attivazione***

- allocati secondo l'ordine delle chiamate
- deallocati in ordine inverso

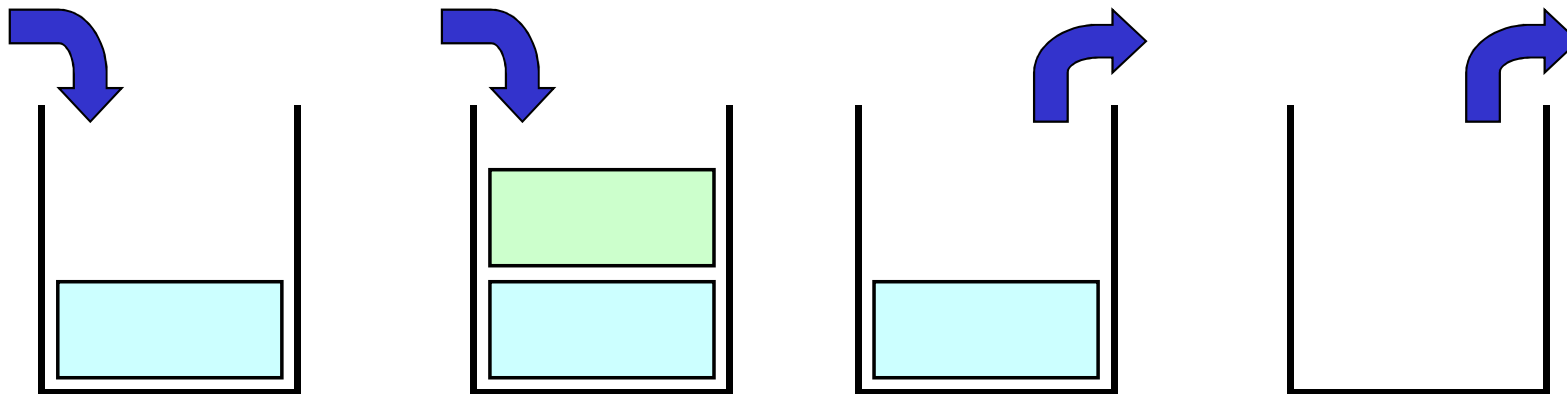
La sequenza dei link dinamici costituisce la cosiddetta *catena dinamica*, che rappresenta *la storia delle attivazioni* (“*chi ha chiamato chi*”)

RECORD DI ATTIVAZIONE

Per catturare la semantica delle chiamate annidate (una funzione che chiama un'altra funzione che...), l'area di memoria in cui vengono allocati i record di attivazione deve essere gestita come una pila

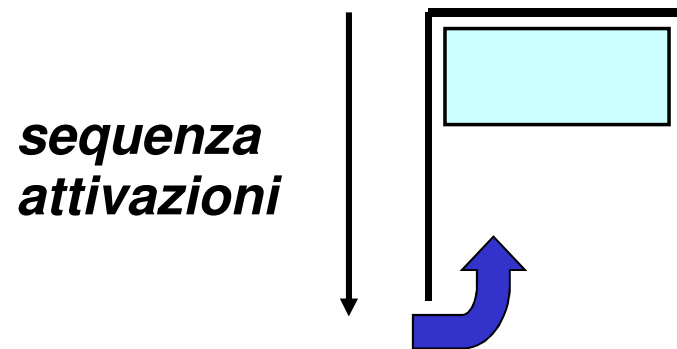
STACK

Una struttura dati gestita con politica LIFO (Last In, First Out - l'ultimo a entrare è il primo a uscire)

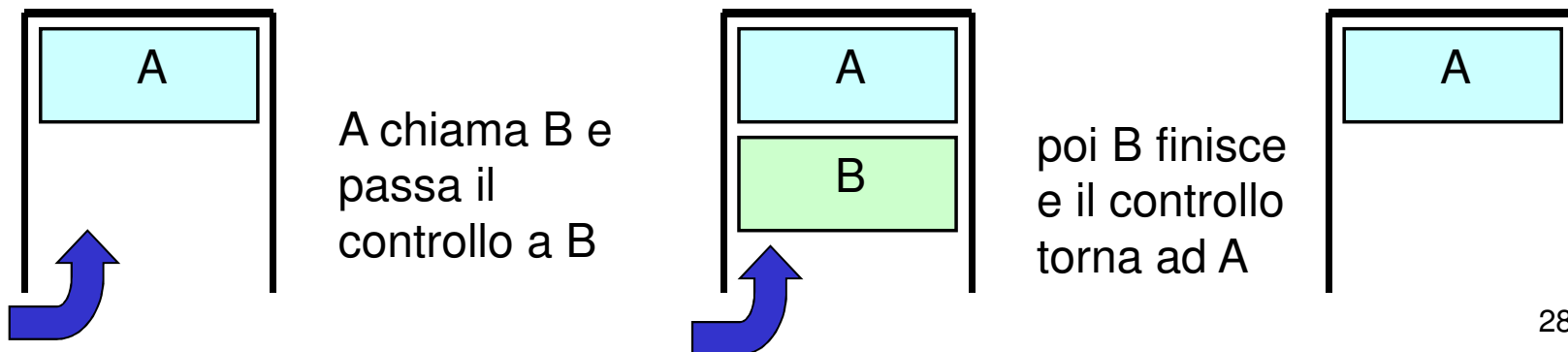


RECORD DI ATTIVAZIONE

Normalmente lo STACK dei record di attivazione si disegna nel modo seguente



Quindi, se la funzione A chiama la funzione B lo stack evolve nel modo seguente



ESEMPIO DI CHIAMATE ANNIDATE

Programma:

```
int R(int A) { return A+1; }  
int Q(int x) { return R(x); }  
int P(void) { int a=10; return Q(a); }  
int main() { int x = P(); }
```

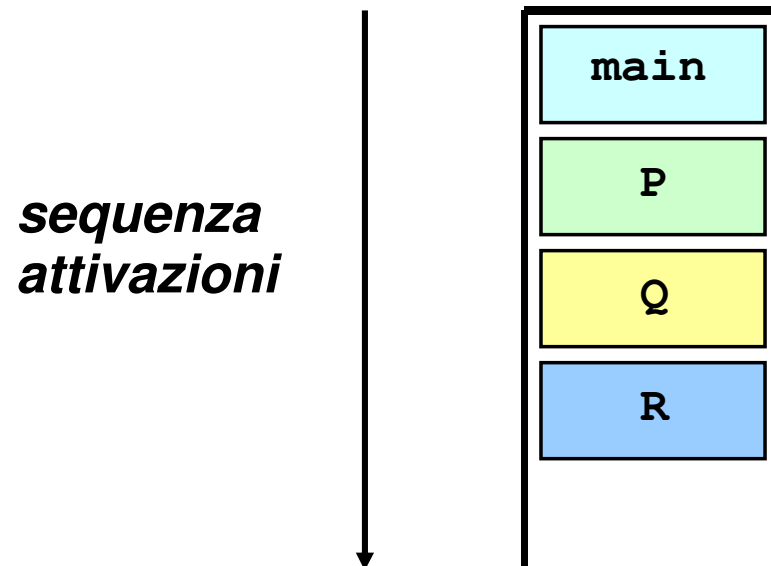
Sequenza chiamate:

$SO \rightarrow \text{main} \rightarrow P() \rightarrow Q() \rightarrow R()$

ESEMPIO DI CHIAMATE ANNIDATE

Sequenza chiamate:

`SO` → `main` → `P()` → `Q()` → `R()`



ESEMPIO: FATTORIALE

```
int fact(int n) {  
    if (n<=0) return 1  
    else return n*fact(n-1);  
}
```

```
int main(){  
    int x, y;  
    x = 2;  
    y = fact(x);  
}
```

NOTA: anche `main()`
è una funzione

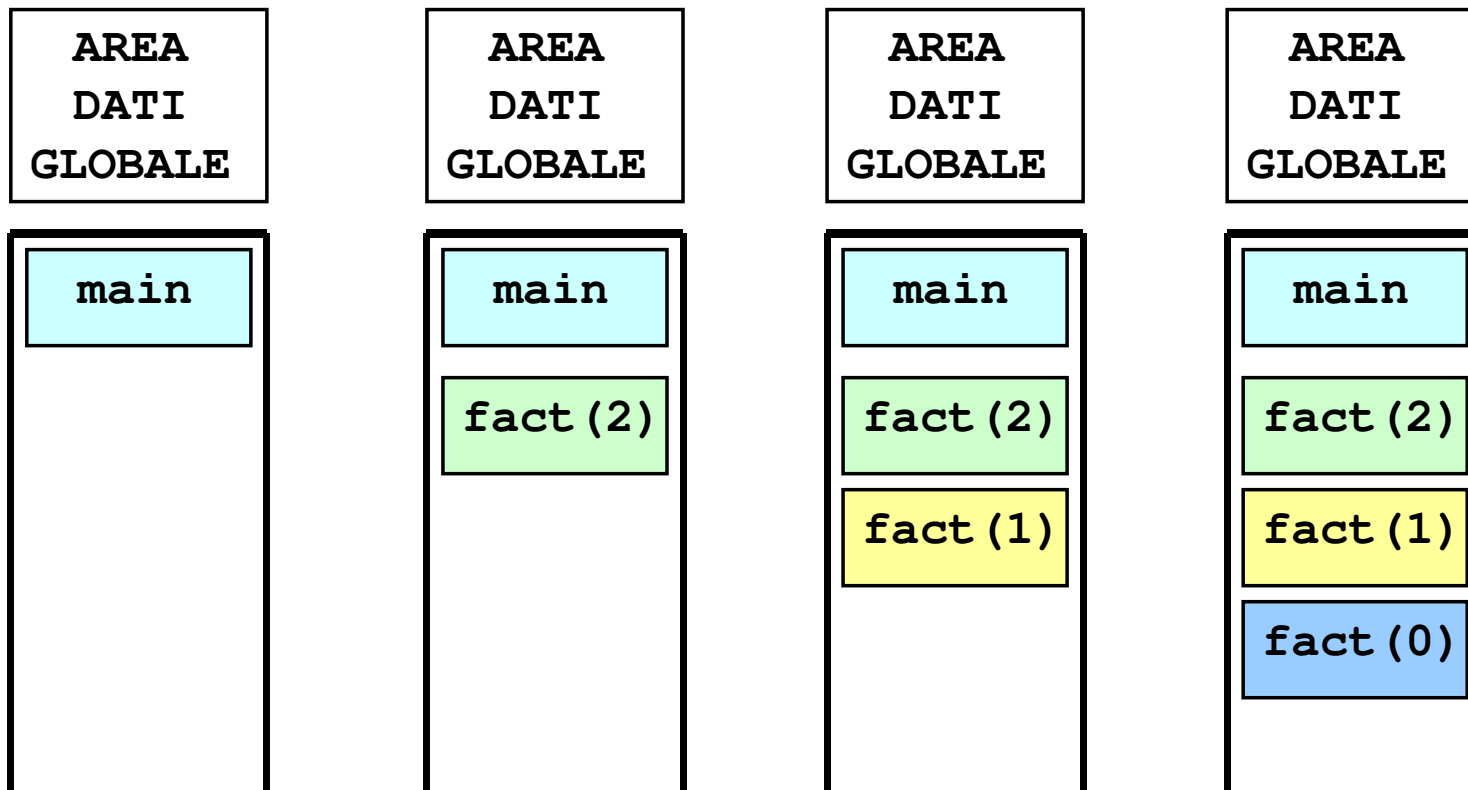
ESEMPIO: FATTORIALE

Situazione all'inizio
dell'esecuzione del
`main()`

`main()`
chiama
`fact(2)`

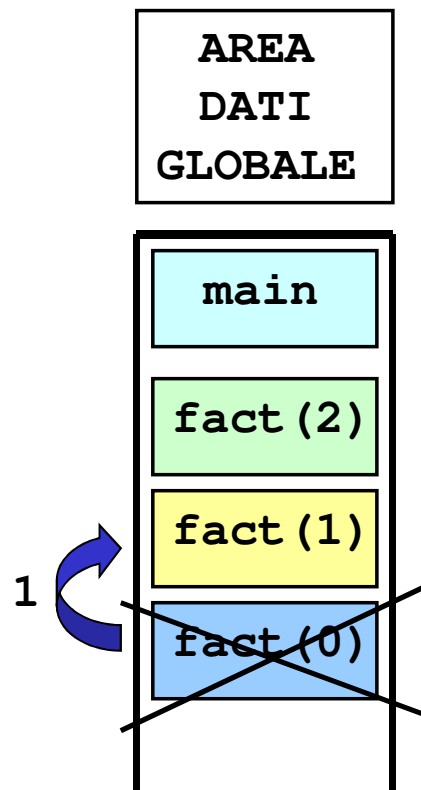
`fact(2)`
chiama
`fact(1)`

`fact(1)`
chiama
`fact(0)`

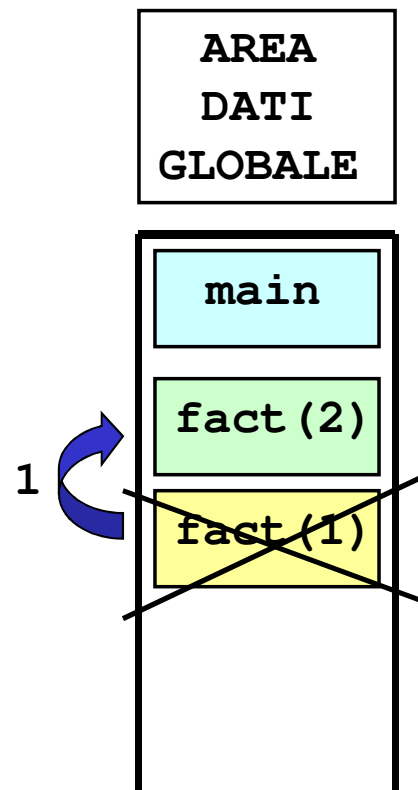


ESEMPIO: FATTORIALE

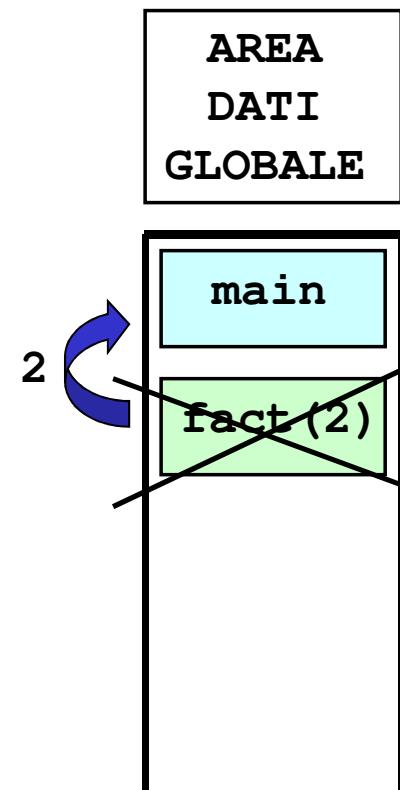
`fact (0)` termina restituendo il valore 1. Il controllo torna a `fact (1)`



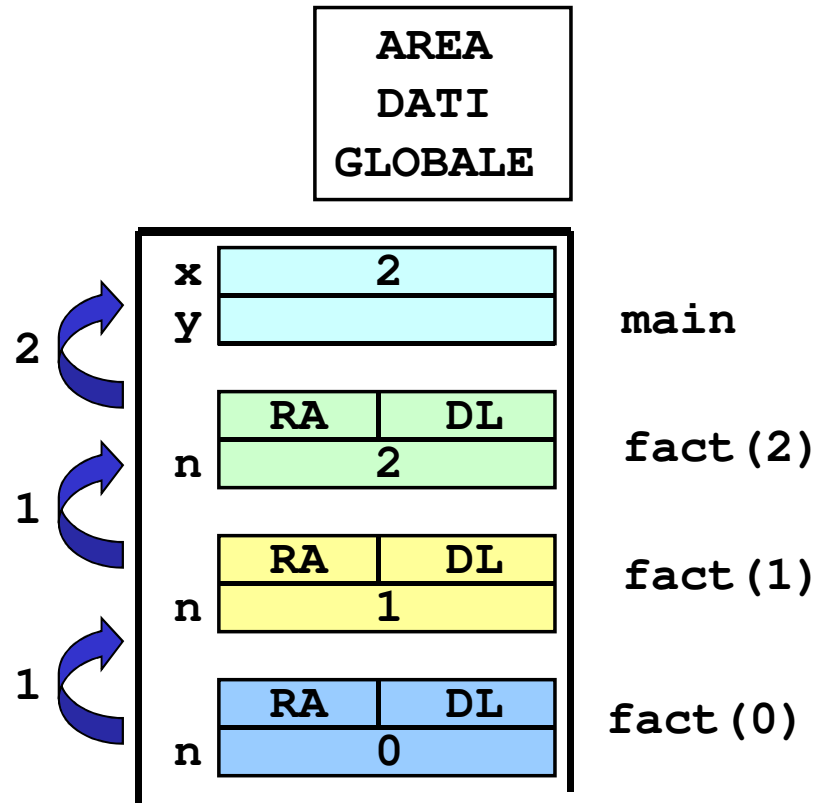
`fact (1)` effettua la moltiplicazione e termina restituendo il valore 1. Il controllo torna a `fact (2)`



`fact (1)` effettua la moltiplicazione e termina restituendo il valore 2. Il controllo torna al `main ()`



RECORD DI ATTIVAZIONE IN DETTAGLIO



RICORSIONE vs. ITERAZIONE

A volte processi computazionali ***ricorsivi*** rispecchiano meglio il problema e/o la ***soluzione del problema*** (ad es. *strutture dati ricorsive* quali liste - le vedremo nel dettaglio più avanti...)

MA:

nei processi computazionali ***ricorsivi*** ogni funzione che effettua una chiamata ricorsiva deve ***aspettare il risultato del servitore*** per ***effettuare operazioni su questo***; solo ***in seguito può terminare***

→ ***Maggiore occupazione di memoria per record attivazione*** a meno di “ottimizzazioni” da parte del compilatore (*tail recursion optimization* non presente in C e Java, ma utilizzata in Prolog)