

***RICORSIONE***

# Agenda

- Record di attivazione
- Ricorsione
- Iterazione
- Ricorsione Tail

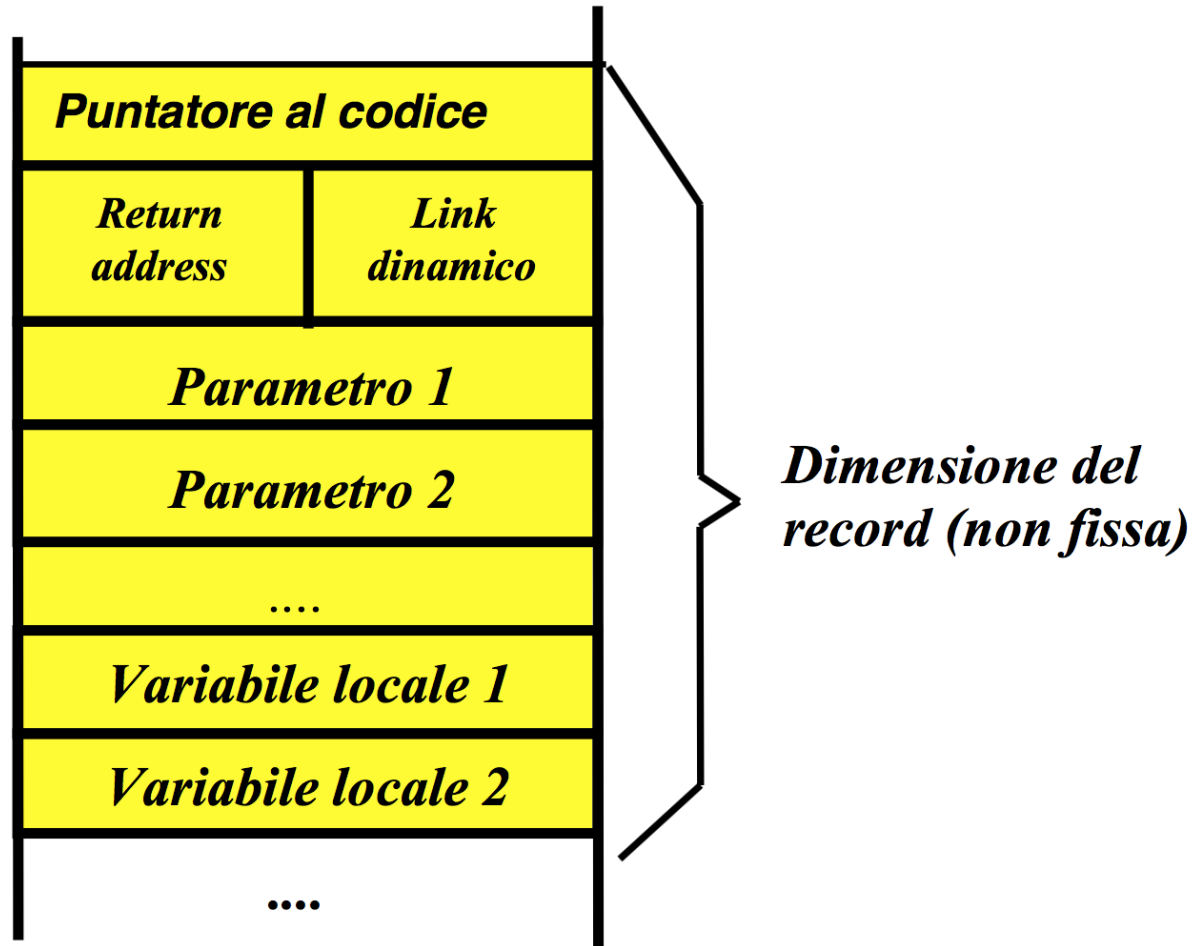
# Gestione dell'esecuzione di funzioni mediante record di attivazione

- Ogni volta che viene invocata una funzione:
  - si crea di una nuova **attivazione** (istanza) del servitore (la funzione chiamata)
  - 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

# Record di attivazione

- Al momento dell'invocazione:
  - viene creata dinamicamente una struttura dati che contiene il *binding* (legame) dei parametri e degli identificatori definiti localmente alla funzione detta **RECORD DI ATTIVAZIONE**.
- È il “**mondo della funzione**”: contiene tutto ciò che serve per la chiamata alla quale è associato:
  - i **parametri** formali
  - le **variabili locali**
  - l'**indirizzo di ritorno** (Return address RA) che indica il punto a cui tornare (nel codice della funzione chiamante, detta *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 (**Link Dinamico DL**)
  - l'**indirizzo del codice** della funzione (puntatore alla prima istruzione del corpo)

# Record di attivazione



# Record di attivazione

- Il record di attivazione associato a una chiamata di una funzione  $f$ :
  - creato al momento della invocazione di  $f$
  - permane per tutto il tempo in cui la funzione  $f$  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

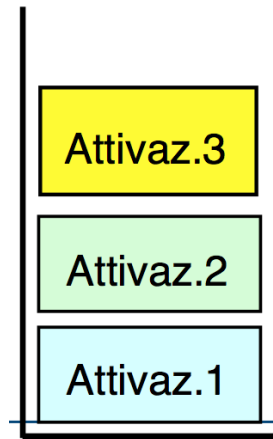
- Funzioni che chiamano altre funzioni danno luogo a una sequenza di record di attivazione
  - allocati secondo l'ordine delle chiamate
  - deallocati in ordine inverso
- La sequenza dei link dinamici costituisce la cosiddetta catena dinamica, che rappresenta la storia delle attivazioni (“chi ha chiamato chi”)

# Stack

- L'area di memoria in cui vengono allocati i record di attivazione viene gestita come una **pila**:

## STACK

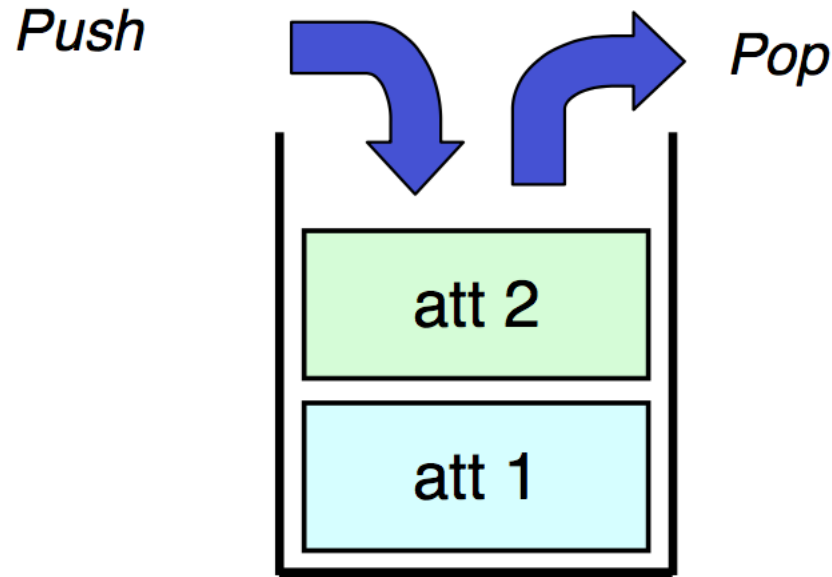
- E' una struttura dati gestita a tempo di esecuzione con politica LIFO (**Last In, First Out** - l'ultimo a entrare è il primo a uscire) nella quale ogni elemento è un record di attivazione.
- La gestione dello stack avviene mediante due operazioni:
  - **push**: aggiunta di un elemento (in cima alla pila)
  - **pop**: prelievo di un elemento (dalla cima della pila)





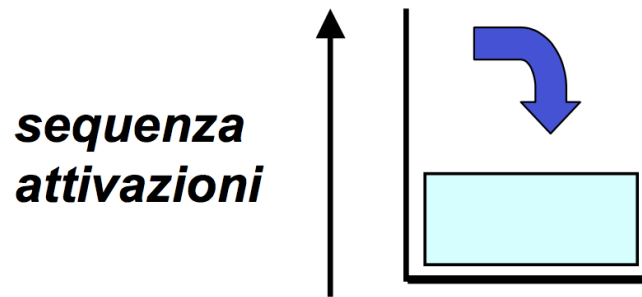
# Stack

- L'ordine di collocazione dei record di attivazione nello stack indica la cronologia delle chiamate:

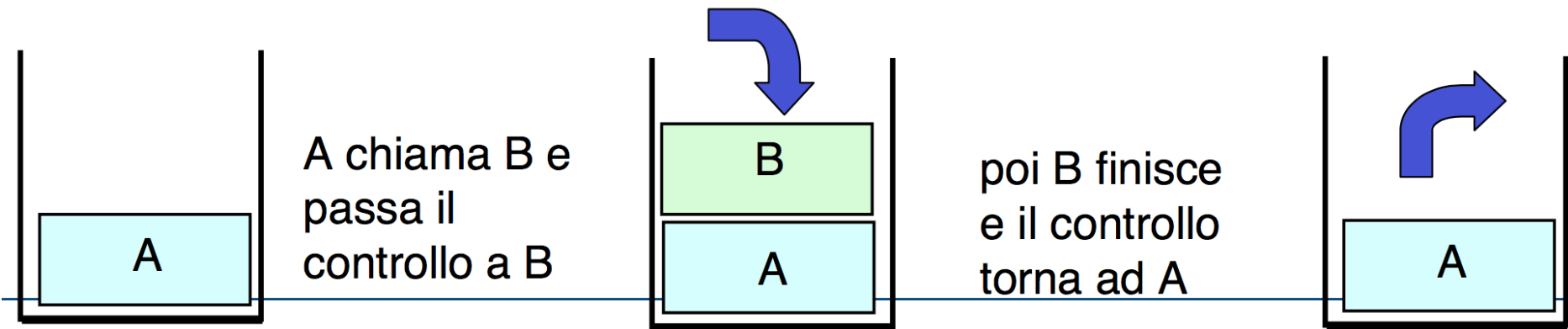


# Record di attivazione

- 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: 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); }`
- `main() { int x = P(); }`

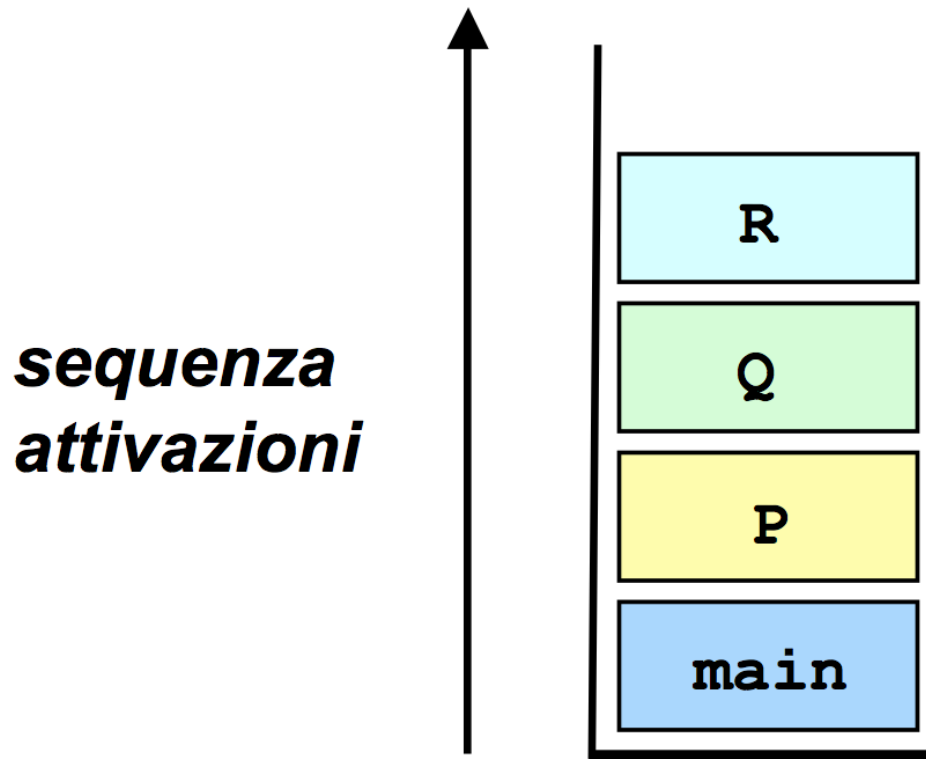
Sequenza chiamate:

- **S.O.**  $\rightarrow$  `main`  $\rightarrow$  `P()`  $\rightarrow$  `Q()`  $\rightarrow$  `R()`

# Esempio: chiamate annidate

Sequenza chiamate:

- **S.O.**  $\rightarrow$  **main**  $\rightarrow$  **P()**  $\rightarrow$  **Q()**  $\rightarrow$  **R()**



# Agenda

- Record di attivazione
- Ricorsione
- Iterazione
- Ricorsione Tail

# 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 dimostrare che, ***assumendola valida per  $n \geq n_0$*** , allora vale anche per  $n+1$
  - allora  $P$  vale per ogni  $n \geq n_0$

# Esempio di funzione matematica definita ricorsivamente: Il Fattoriale

**Esempio:** il fattoriale di un numero naturale  
`fact(n) = n!`

`n! : N → N`

$$\begin{cases} n! \text{ vale } 1 & \text{se } n == 0 \\ n! \text{ vale } n * (n-1)! & \text{se } n > 0 \end{cases}$$

# La Ricorsione in programmazione

- Operativamente, risolvere un problema con un approccio ricorsivo comporta
  - di identificare un “caso base”, con soluzione nota
  - di riuscire a esprimere la soluzione del caso generico  $n$  in termini dello *stesso problema in uno o più casi più semplici* ( $n-1$ ,  $n-2$ , etc.), dove  $n$  è la taglia del problema



# La Ricorsione in programmazione

(cont.)

- Un sottoprogramma ricorsivo è:
  - un sottoprogramma che richiama direttamente o indirettamente se stesso.
- Non tutti i linguaggi realizzano il meccanismo della ricorsione. Quelli che lo realizzano, di solito utilizzano la tecnica di **gestione mediante record di attivazione**: ad ogni chiamata è associato un record di attivazione (variabili locali e punto di ritorno).

# La Ricorsione: Il Fattoriale

- In C è possibile realizzare funzioni ricorsive
- Il corpo di ogni funzione ricorsiva contiene almeno una chiamata alla funzione stessa, direttamente o indirettamente.
- **Esempio: definizione in C della funzione ricorsiva fattoriale.**

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

# La Ricorsione: Il Fattoriale

- **Servitore & Cliente:** **fact** è sia servitore che cliente (di se stessa):

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

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

# La Ricorsione: Il Fattoriale (cont.)

## Servitore & Cliente:

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

```
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: Il Fattoriale (cont.)

## Servitore & Cliente:

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

```
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: Il Fattoriale (cont.)

## Servitore & Cliente:

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

```
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: Il Fattoriale (cont.)

## Servitore & Cliente:

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

```
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: Il Fattoriale (cont.)

## Servitore & Cliente:

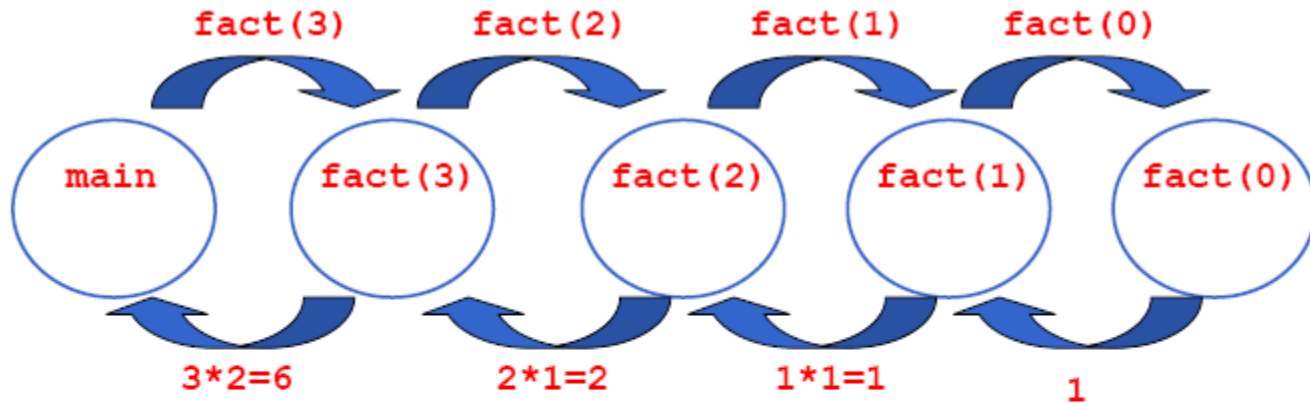
```
int fact(int n) {  
    if (n<=0) return 1;  
    else return n*fact(n-1);  
}
```

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

**IL CONTROLLO PASSA INFINE  
AL MAIN CHE ASSEGNA A fz IL  
VALORE 6**



# La Ricorsione: Il Fattoriale (cont.)



`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)`

# Cosa succede nello stack ?

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

```
main() {
    int fz, f6, z = 5;
    fz = fact(z-2);
}
```

NOTA: Anche il `main()` è una funzione

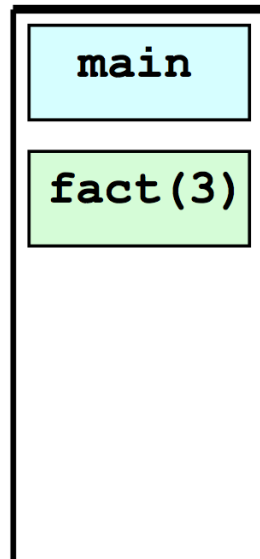
**Seguiamo l'evoluzione dello stack durante l'esecuzione:**

# Cosa succede nello stack ?

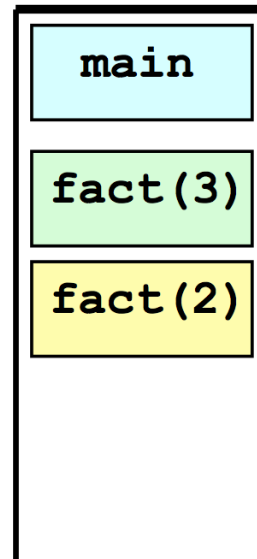
Situazione  
iniziale



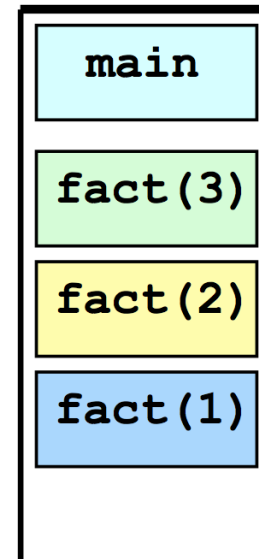
Il `main()`  
chiama  
`fact(3)`



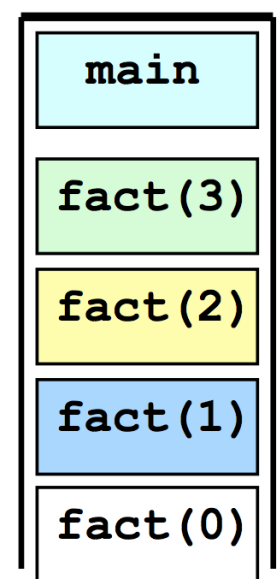
`fact(3)`  
chiama  
`fact(2)`



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

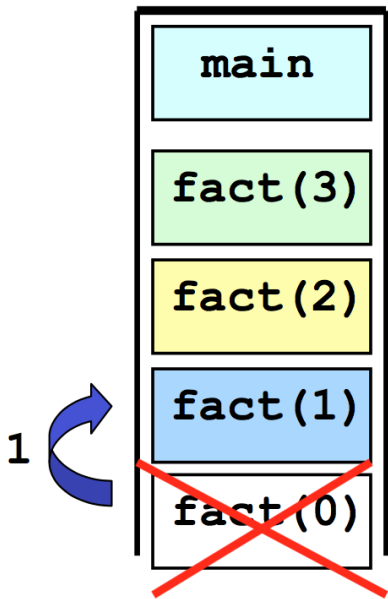


`fact(1)`  
chiama  
`fact(0)`

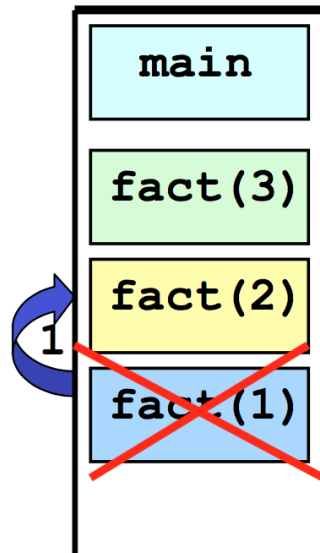


# Cosa succede nello stack ?

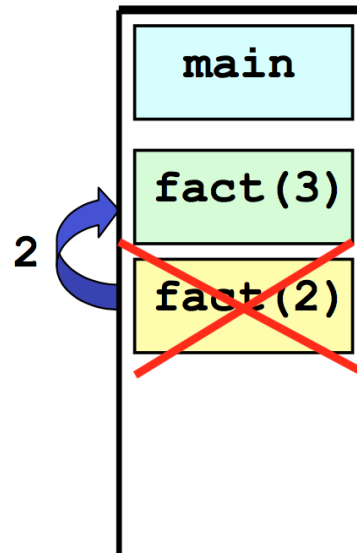
**fact(0)** termina restituendo il valore 1. Il controllo torna a **fact(1)**



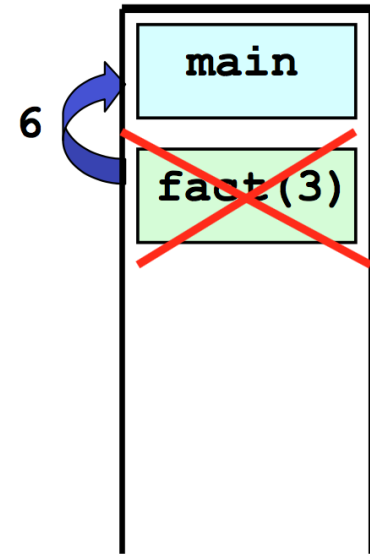
**fact(1)** effettua la moltiplicazione e termina restituendo il valore 1. Il controllo torna a **fact(2)**



**fact(2)** effettua la moltiplicazione e termina restituendo il valore 2. Il controllo torna a **fact(3)**



**fact(3)** effettua la moltiplicazione e termina restituendo il valore 6. Il controllo torna al **main**.



# La Ricorsione: La Somma dei Primi $n$ Interi

**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: La Somma dei Primi $n$ Interi (cont.)

**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: La Somma dei Primi $n$ Interi (cont.)

**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: successione di Fibonacci

**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: Fibonacci (cont.)

## Problema:

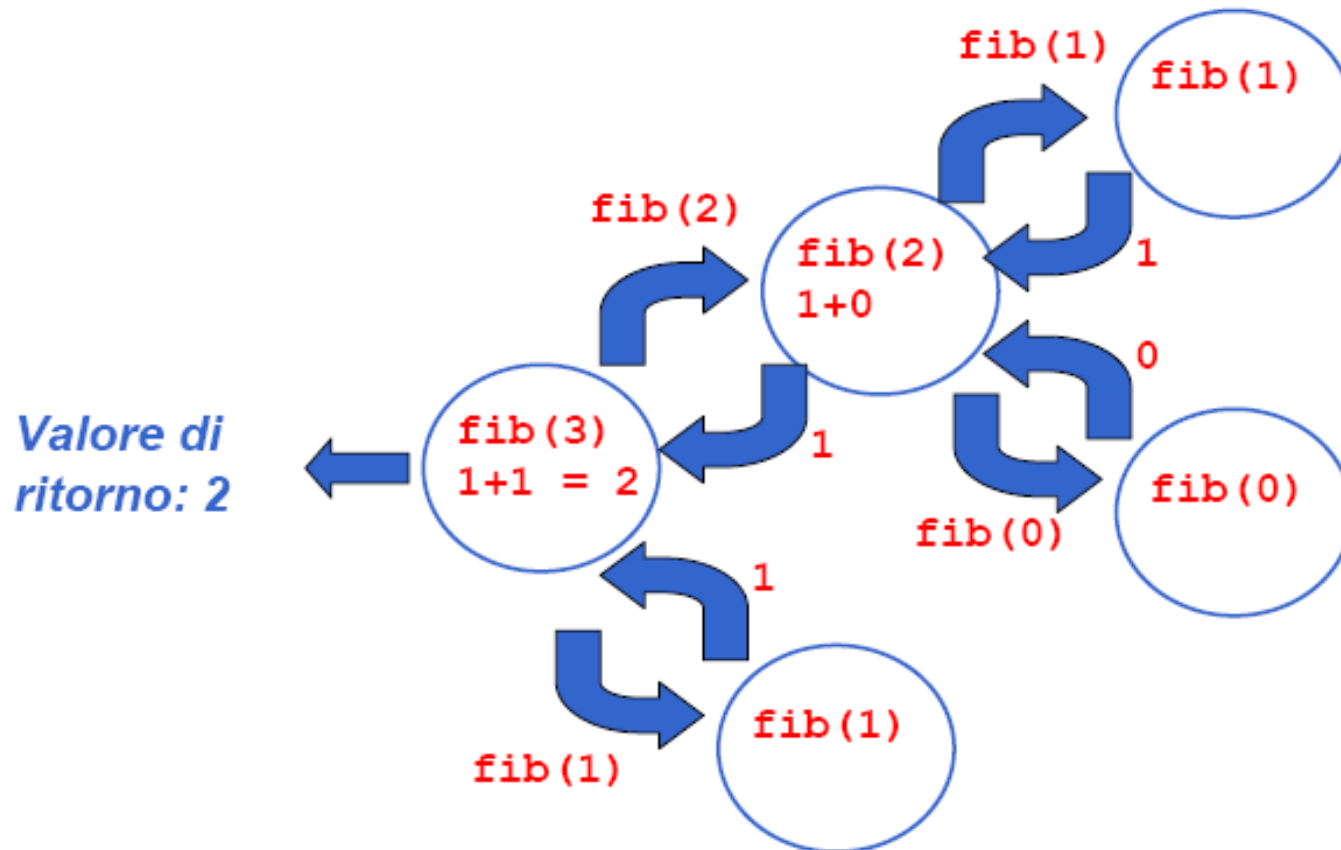
calcolare l'N-esimo numero di Fibonacci

## 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*

# La Ricorsione: Fibonacci (cont.)



# La Ricorsione: Riflessioni

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***

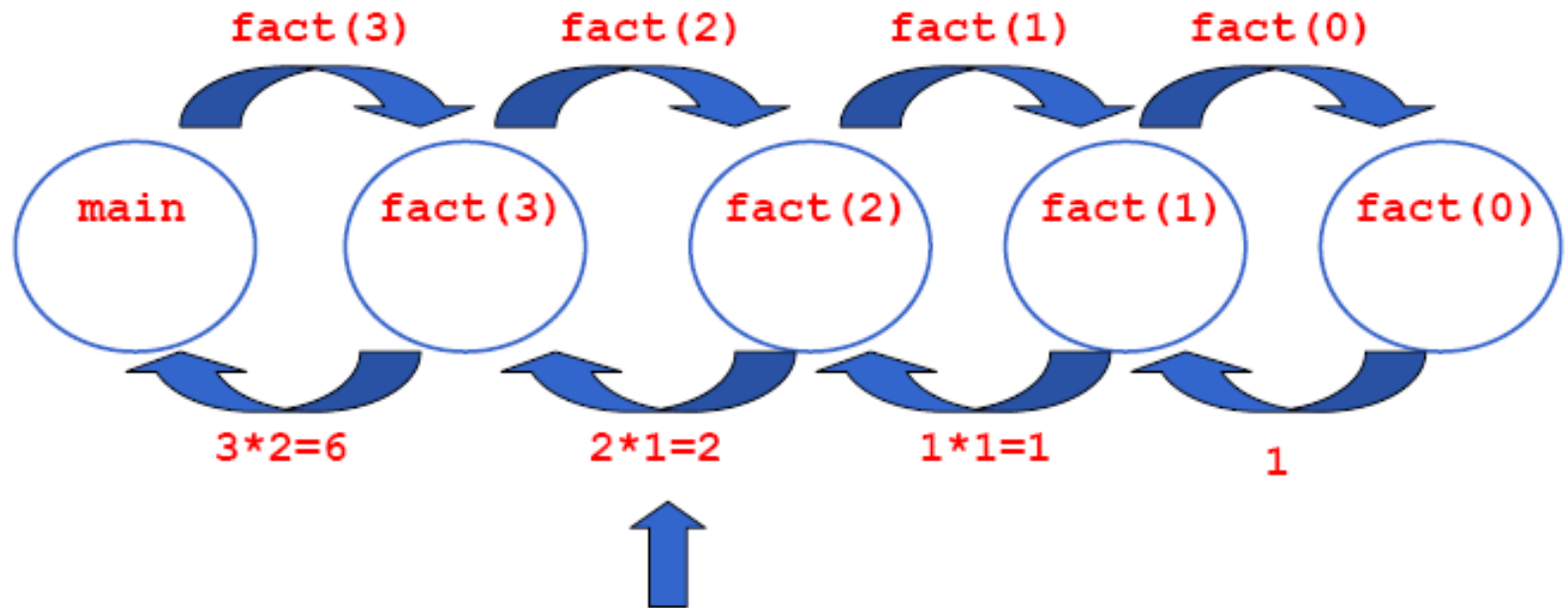
Il risultato viene sintetizzato a partire dalla fine, perché prima occorre arrivare al caso “banale”:

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



**Processo computazionale effettivamente ricorsivo**

# La Ricorsione: Riflessioni (cont.)



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

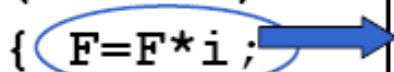
# Agenda

- Record di attivazione
- Ricorsione
- Iterazione
- Ricorsione Tail

# Iterazione: Fattoriale (cont.)

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  $i$

21

# Iterazione

- Nell'esempio precedente il risultato viene sintetizzato “*in avanti*”
- L'esecuzione di un algoritmo di calcolo che computi “in avanti”, per accumulo, 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 finché non si è giunti fino al caso elementare.

# Iterazione (cont.)

- Un processo computazionale iterativo si può realizzare anche tramite funzioni ricorsive
- Si basa sulla disponibilità di una variabile, detta *accumulatore*, destinata a esprimere in ogni istante la soluzione corrente
- Si imposta identificando quell'operazione di *modifica dell'accumulatore* che lo porta a esprimere, dal valore relativo al passo  $k$ , il valore relativo al passo  $k+1$



# Iterazione: Fattoriale (cont.)

## 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  $k=n$*

# Agenda

- Record di attivazione
- Ricorsione
- Iterazione
- Ricorsione Tail

# Ricorsione Tail

- Una ricorsione che realizza un processo computazionale *ITERATIVO* è una ricorsione apparente
- la chiamata ricorsiva è sempre l'ultima istruzione
  - i calcoli sono fatti prima
  - la chiamata serve solo, dopo averli fatti, per proseguire la computazione
- questa forma di ricorsione si chiama RICORSIONE TAIL (“ricorsione in coda”)

# Ricorsione Tail: Il Fattoriale

- il corpo del ciclo rimane *immutato*
- il ciclo diventa un **if** con, in fondo, la chiamata tail-ricorsiva

```
while (condizione) {  
    <corpo del ciclo>  
}
```

```
if (condizione) {  
    <corpo del ciclo>  
    <chiamata ricorsiva>  
}
```

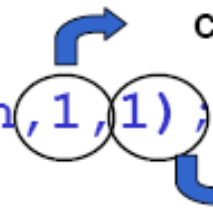
Naturalmente, può essere necessario **aggiungere nuovi parametri** nell'intestazione della funzione tail-ricorsiva, per “portare avanti” le variabili di stato

# Ricorsione Tail: Il Fattoriale (cont.)

```
int fact(int n){  
    return factIt(n, 1, 1);  
}
```


Inizializzazione dell'accumulatore:  
corrisponde al fattoriale di 1

Contatore del passo

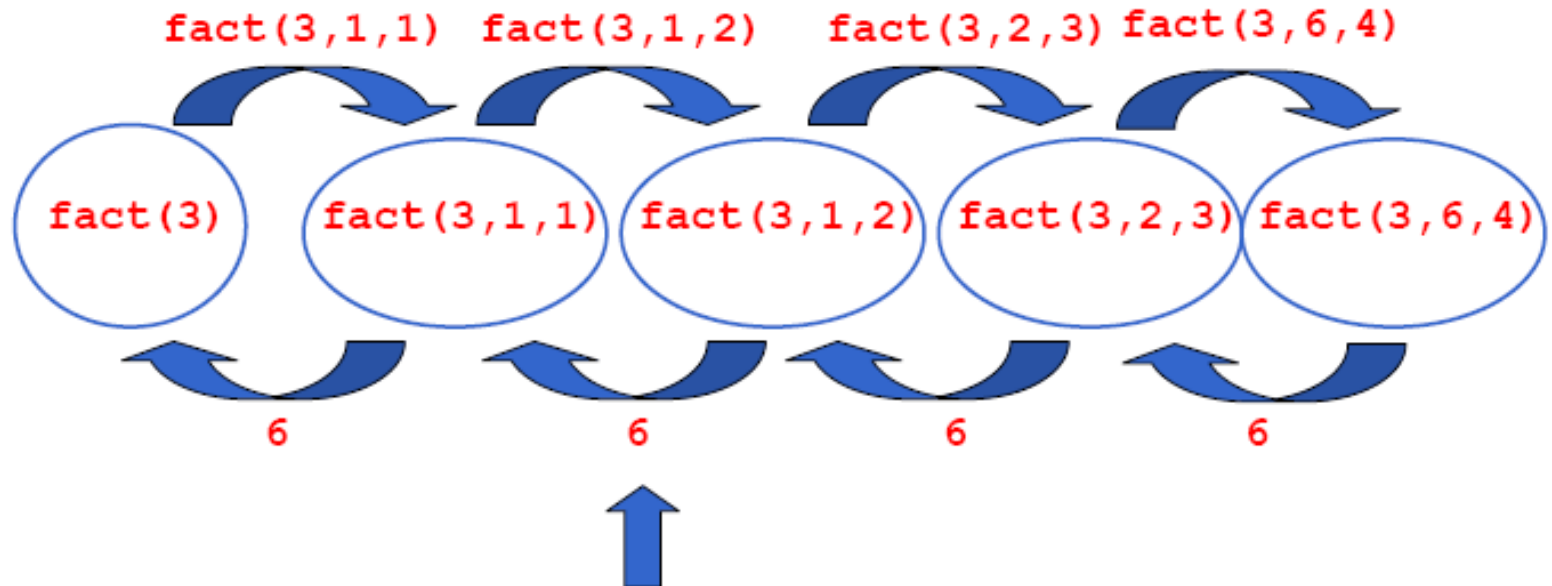


```
int factIt(int n, int F, int i){  
    if (i <= n)  
    {  
        F = i * F;  
        i = i + 1;  
        return factI(n, F, i);  
    }  
    return F;  
}
```

Accumulatore del risultato parziale



# Ricorsione Tail: Il Fattoriale (cont.)



**NOTA:** ciascuna funzione che effettua una chiamata ricorsiva si sospende, aspetta la terminazione del servitore e poi termina, cioè **NON EFFETTUA ALTRE OPERAZIONI DOPO**

# Ricorsione Tail: Il Fattoriale (cont.)

La soluzione ricorsiva individuata per il fattoriale è ***sintatticamente ricorsiva*** ma dà luogo a un ***processo computazionale ITERATIVO***

Ricorsione apparente detta **RICORSIONE TAIL**

Il risultato viene sintetizzato *in avanti*

- ogni passo *decompone e calcola*
- e *porta in avanti il nuovo risultato parziale* quando le chiamate si chiudono non si fa altro che riportare indietro, fino al cliente, il risultato ottenuto

# Ricorsione vs. Iterazione

- Ripetizione
  - Iterazione: ciclo esplicito
  - Ricorsione: chiamate di funzione ripetute
- Terminazione
  - Iterazione : il ciclo fallisce la condizione
  - Ricorsione : il caso base è riconosciuto
- Entrambe possono dar luogo a cicli infiniti
- Bilancio
  - Scegli tra performance (iterazione) e buona ingegneria del software (ricorsione)



# Ricorsione vs. Iterazione

- |                                    |                                     |
|------------------------------------|-------------------------------------|
| ■ Uso di un costrutto di selezione | ■ Uso di un costrutto di iterazione |
| ■ Condizione di terminazione       | ■ Condizione di terminazione        |
| ■ Non convergenza                  | ■ Loop infinito                     |

A differenza dell'iterazione, la ricorsione richiede un notevole sovraccarico (*overhead*) a tempo d'esecuzione dovuto alle chiamate di funzione.

Dato che un programma ricorsivo può essere sempre trasformato in un programma iterativo, perché usare la ricorsione?

# Ricorsione vs. Iterazione

- Algoritmi che per loro natura sono ricorsivi piuttosto che iterativi dovrebbero essere formulati con procedure ricorsive.
- Ad esempio, alcune strutture dati sono inerentemente ricorsive:
  - Strutture ad albero
  - Sequenze
  - .....
- e la formulazione ricorsiva di algoritmi su di esse risulta più naturale.
- La ricorsione deve essere evitata quando esiste una soluzione iterativa ovvia e in situazioni in cui le prestazioni del sistema sono un elemento critico

## Definizione di strutture dati ricorsive: le liste

- un elemento di tipo **lista** *lis* è:

una lista vuota, cioè  $lis = nil$

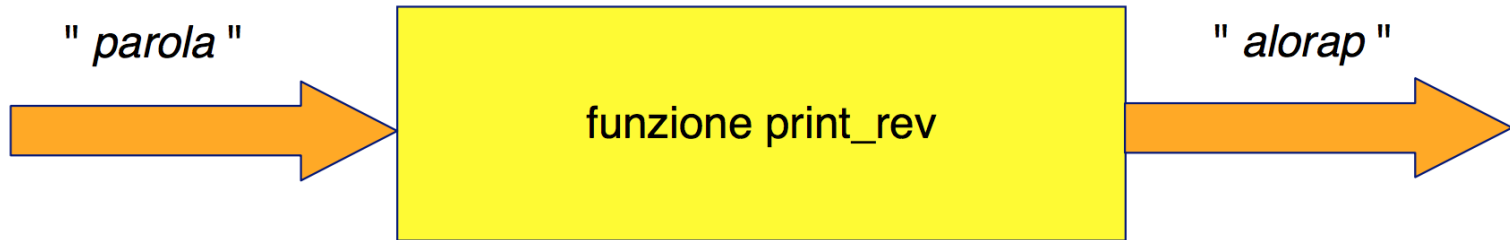
oppure

è una coppia  $(elem, lis1)$  dove  
*elem* è un item e *lis1* è una lista


Esempio:  $lis = (3, (7, (11, (4, nil))))$

# Esercizio

- Scrivere una funzione ricorsiva **print\_rev** che, data una sequenza di caratteri (terminata dal carattere '.') stampi i caratteri della sequenza in ordine inverso. La funzione non deve utilizzare stringhe.
- Ad esempio:




# Soluzione

- **Osservazione:** l'estrazione (pop) dei record di attivazione dallo stack avviene sempre in ordine inverso rispetto all'ordine di inserimento (push).
-  associamo ogni carattere letto a una nuova chiamata ricorsiva della funzione

- **Soluzione:**

```
void print_rev(char car)
{ char c;
  if (car != '.')
  { c = getchar();
    print_rev(c);
    printf("%c", car);
  }
  else return;
}
```

ogni record di attivazione nello stack memorizza un singolo carattere letto (*push*); in fase di *pop*, i caratteri vengono stampati nella sequenza inversa



# Soluzione

```
#include <stdio.h>
#include <string.h>
void print_rev(char car);
main(){
    char k;
    printf("\nIntrodurre una sequenza terminata da .:\t");
    k = getchar();
    print_rev(k);
    printf("\n*** FINE ***\n");
}
void print_rev(char car) {
    char c;
    if (car != '.') {
        c = getchar();
        printf("lettera %c\n", c);
        print_rev(c);
        printf("%c", car);
    }
    else return;
}
```

# Stack

## Codice

```
...  
main(void)  
{  
    ...  
    print_rev(k);  
}  
void print_rev(char car);  
{  
    if (car != '.')  
    {  
        ...  
        print_rev(c);  
        printf("%c", car);  
    }  
    else return;  
}
```

main

RA●

→ S.O.

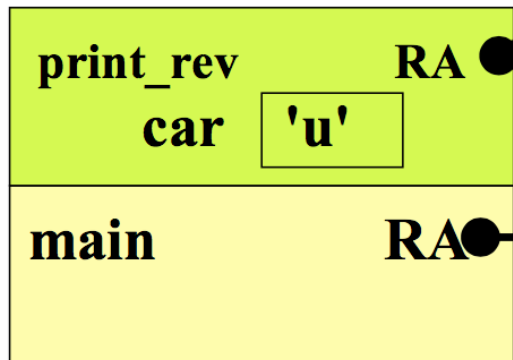
**Standard Input:**

"uno."

# Stack

## Codice

```
...  
main(void)  
{  
    ...  
    print_rev(k);  
}  
void print_rev(char car);  
{  
    if (car != '.')  
    { ...  
        print_rev(c);  
        printf("%c", car);  
    }  
    else return;  
}
```



S.O.

Standard Input:

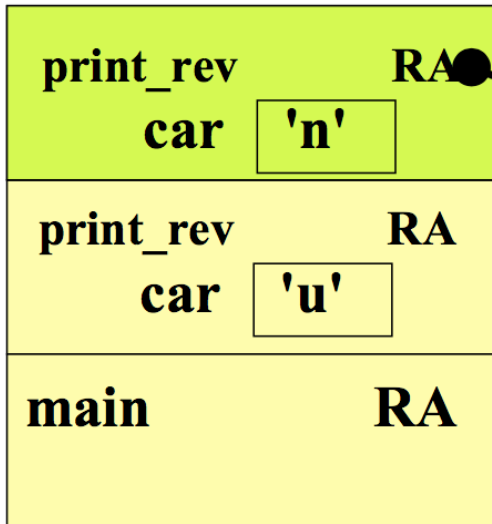
"u"no."



# Stack

## Codice

```
...  
main(void)  
{  
    ...  
    print_rev(k);  
}  
void print_rev(char car);  
{  
    if (car != '.')  
    { ...  
        print_rev(c);  
        printf("%c", car);  
    }  
    else return;  
}
```

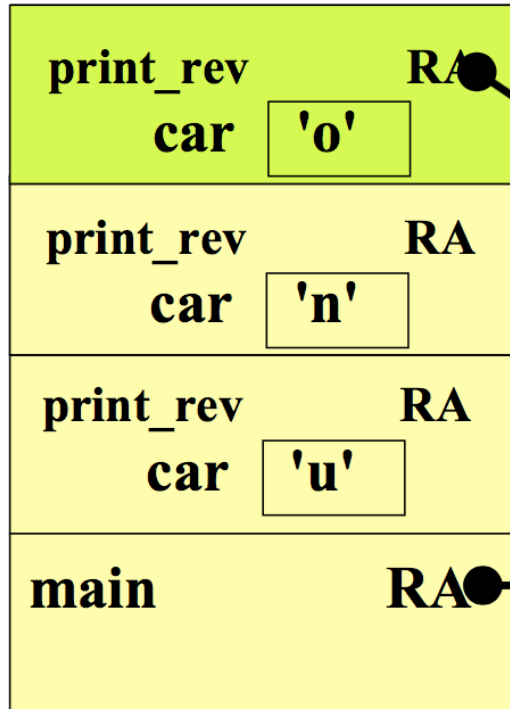


**Standard Input:**

"uno."

# Stack

## Codice



```
...  
main(void)  
{  
    ...  
    print_rev(k);  
}  
void print_rev(char car);  
{  
    if (car != '.')  
    { ...  
        print_rev(c);  
        printf("%c", car);  
    }  
    else return;  
}
```

Standard Input:

"uno."

# Stack

print_rev	RA
car	'.'
print_rev	RA
car	'o'
print_rev	RA
car	'n'
print_rev	RA
car	'u'
main	RA

## Codice

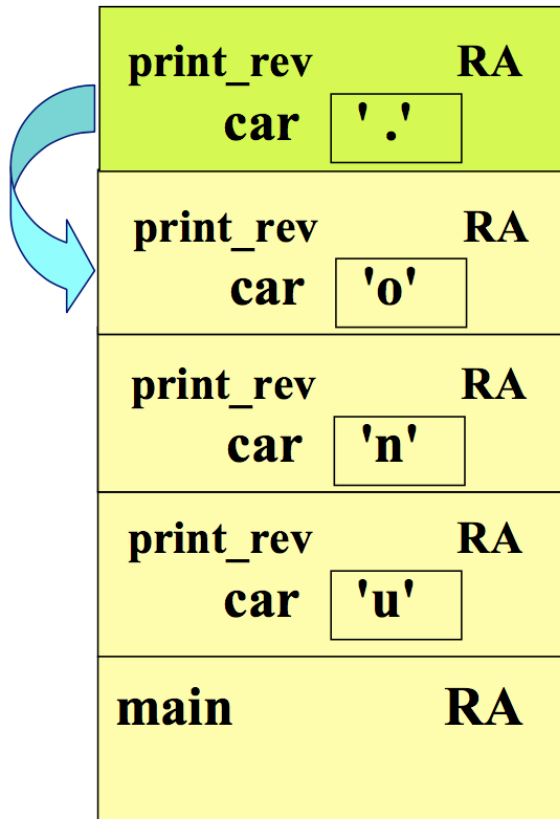
```
...
main(void)
{
    ...
    print_rev(k);
}

void print_rev(char car);
{
    if (car != '.')
    { ...
        print_rev(c);
        printf("%c", car);
    }
    else return;
}
```

Standard Input:

"uno."

# Stack



## Codice

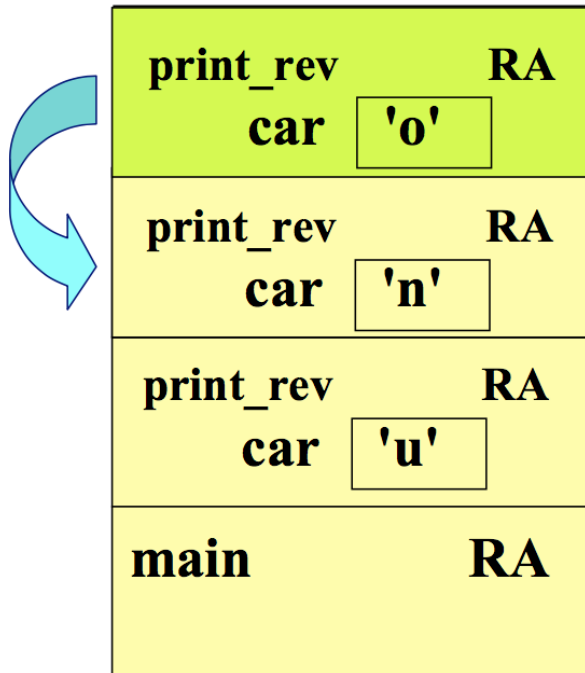
```
...
main(void)
{
    ...
    print_rev(k);
}
void print_rev(char car);
{
    if (car != '.')
    { ...
        print_rev(c);
        printf("%c", car);
    }
    else return;
}
```

**Standard Input:**  
"uno."

# Stack

## Codice

```
...
main(void)
{
    ...
    print_rev(k);
}
void print_rev(char car);
{
    if (car != '.')
    { ...
        print_rev(c);
        printf("%c", car);
    }
    else return;
}
```



Standard output:

"o"

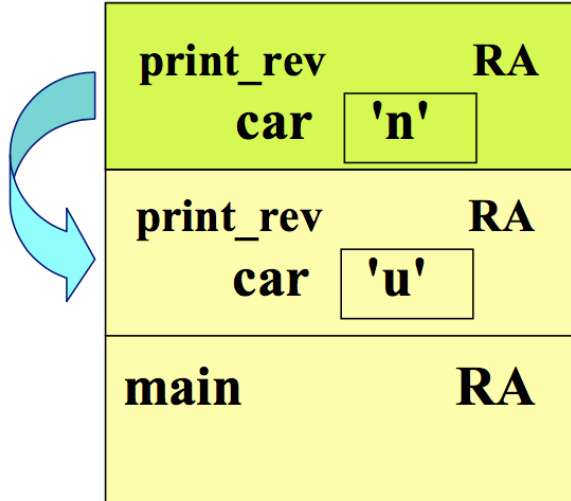
Standard Input:

"uno."

# Stack

## Codice

```
...  
main(void)  
{  
    ...  
    print_rev(k);  
}  
void print_rev(char car);  
{  
    if (car != '.')  
    {  
        ...  
        print_rev(c);  
        printf("%c", car);  
    }  
    else return;  
}
```



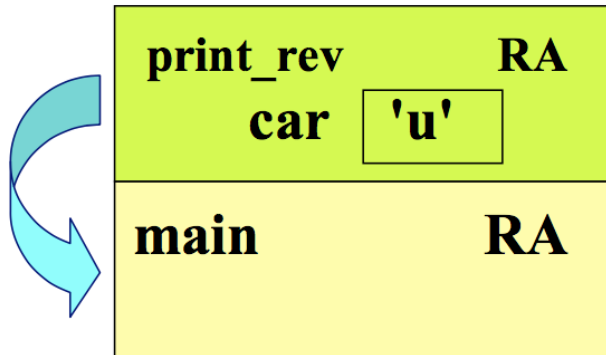
Standard output:  
"on"

Standard Input:  
"uno."

# Stack

## Codice

```
...  
main(void)  
{  
    ...  
    print_rev(k);  
}  
void print_rev(char car);  
{  
    if (car != '.')  
    { ...  
        print_rev(c);  
        printf("%c", car);  
    }  
    else return;  
}
```



**Standard output:**

"onu"

**Standard Input:**

"uno."

# Stack

## Codice

```
...
main(void)
{
    ...
    print_rev(k);
}
void print_rev(char car);
{
    if (car != '.')
    { ...
        print_rev(c);
        printf("%c", car);
    }
    else return;
}
```

**main**

**RA**

**Standard output:**  
"onu"

**Standard Input:**  
"uno."