

Corso di Algoritmi e Strutture Dati

APPUNTI SUL LINGUAGGIO C



Stack e Ricorsione

Funzioni: il modello a RUN-TIME

Ogni volta che viene invocata una **funzione**:

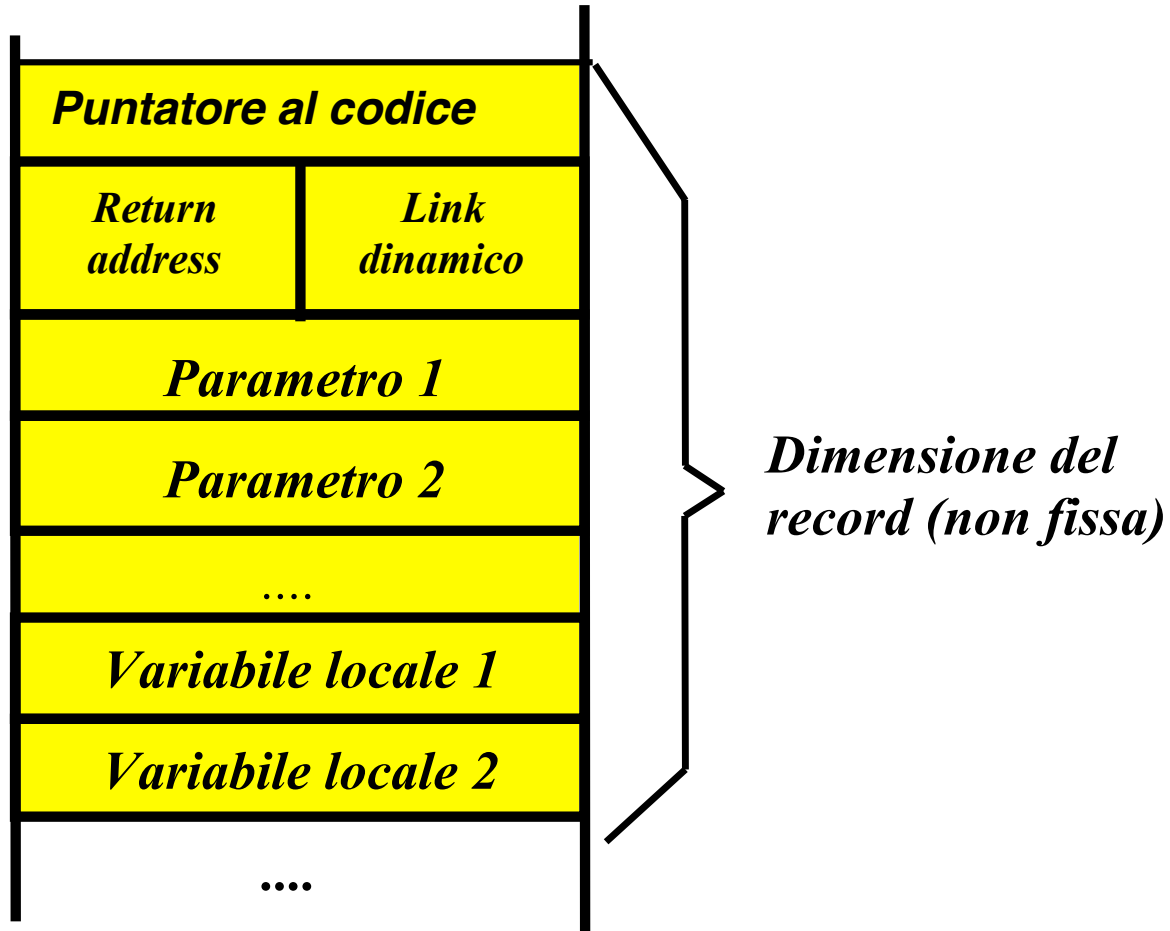
- ❑ si crea di una nuova **attivazione** (*istanza*) del **servitore**
- ❑ viene **allocata la memoria** per i *parametri* e per le *variabili locali*
- ❑ si effettua il **passaggio dei parametri**
- ❑ si **trasferisce il controllo** al *servitore*
- ❑ si **esegue il codice** della funzione

Record di attivazione

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 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* (**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**:

- ❑ è **creato** al momento della **invocazione** di **f**
- ❑ **permane** per tutto il tempo in cui la funzione **f** è **in esecuzione**
- ❑ è **distrutto** (*deallocated*) al **termine** dell'**esecuzione** di **f**.

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*

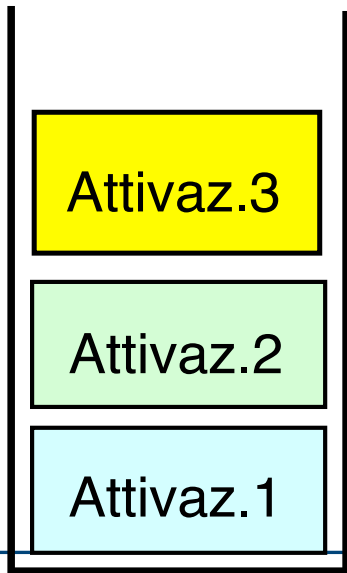
La **sequenza dei link dinamici** costituisce la cosiddetta **catena dinamica**, che rappresenta la storia delle attivazioni
(“*chi ha chiamato chi*”)

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**.

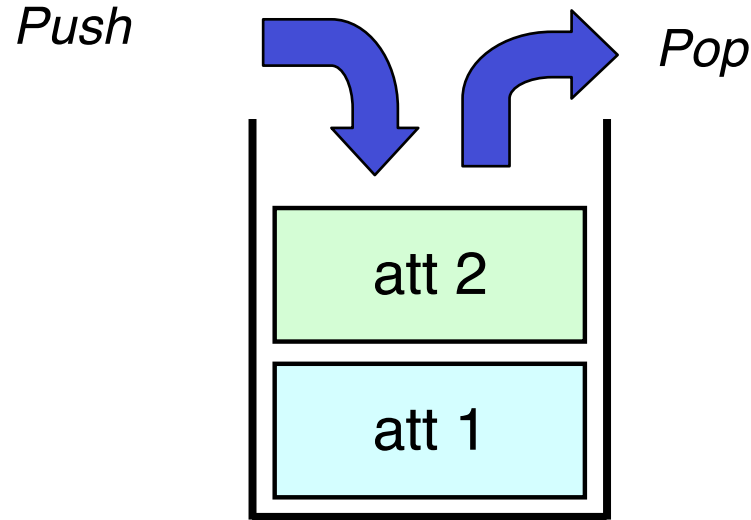


La gestione dello stack avviene mediante due operazioni:

- ❑ **push**: aggiunta di un elemento (in cima alla pila)
- ❑ **pop**: prelievo di un elemento (dalla cima della pila)

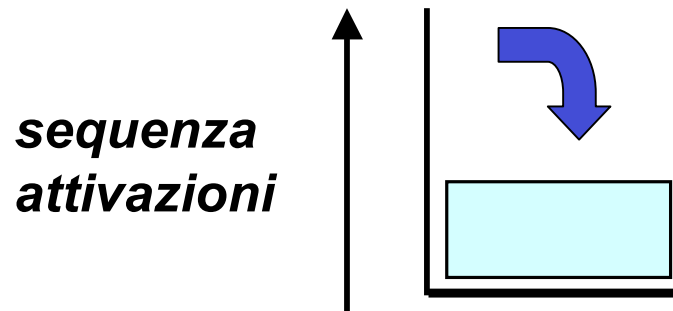
Stack

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

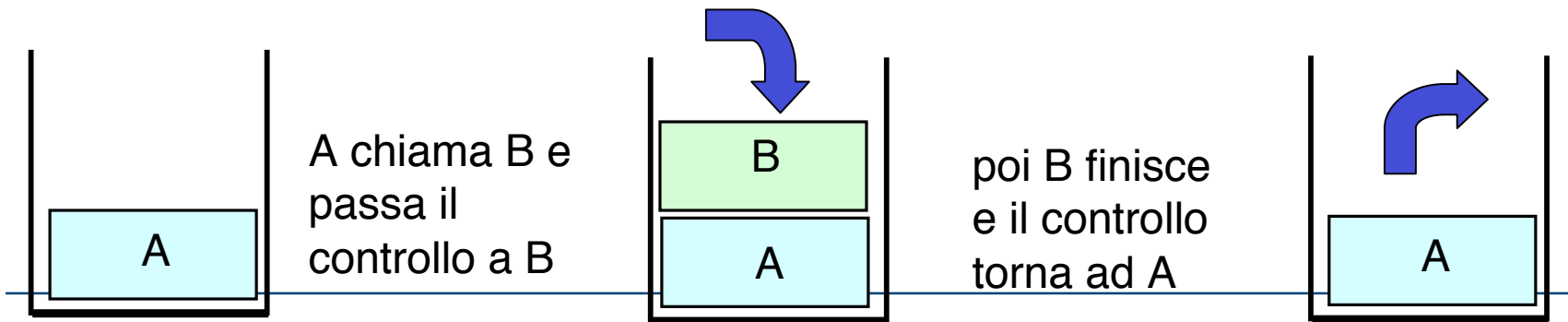


Record di attivazione

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

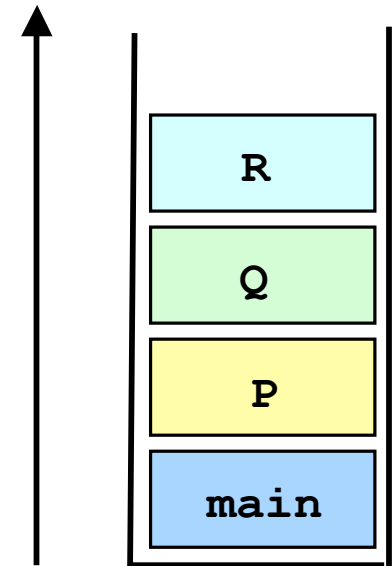
Programma:

```
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:

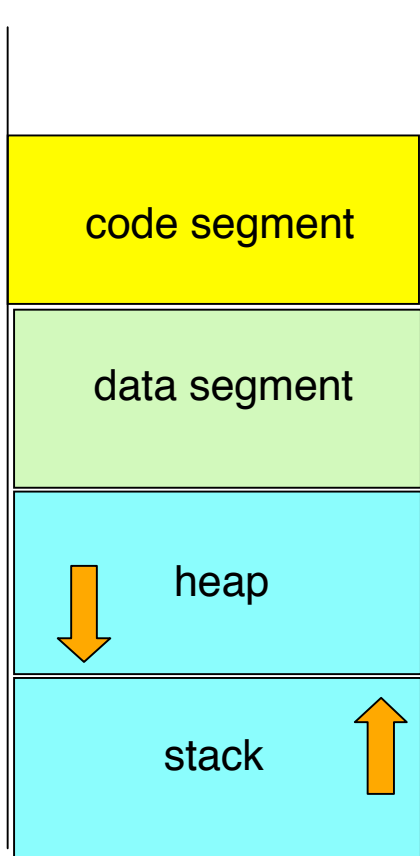
S.O. → main → P() → Q() → R()

*sequenza
attivazioni*



Spazio di indirizzamento

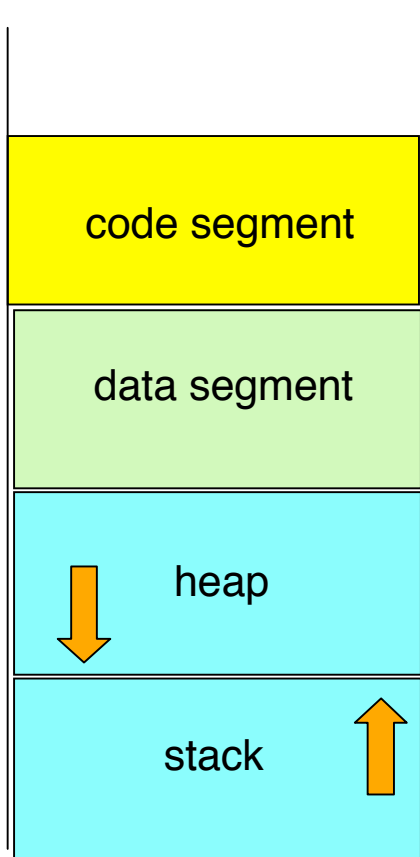
La **memoria allocata** a ogni programma in esecuzione è suddivisa in varie parti (**segmenti**), secondo lo schema seguente:



- ❑ **code segment**: contiene il codice eseguibile del programma
- ❑ **data segment**: contiene le variabili globali
- ❑ **heap**: contiene le variabili dinamiche
- ❑ **stack**: è l'area dove vengono allocati i record di attivazione
- **Code segment** e **data segment** sono di dimensione fissata staticamente (a tempo di compilazione).
- La dimensione dell'area associata a **stack + heap** è fissata staticamente: man mano che lo stack cresce, diminuisce l'area a disposizione dell'heap, e viceversa.

Segmentation Fault

Un **errore di segmentazione** (in inglese **segmentation fault**, spesso abbreviato in **segfault**) è una particolare condizione di errore che può verificarsi durante l'esecuzione di un programma.

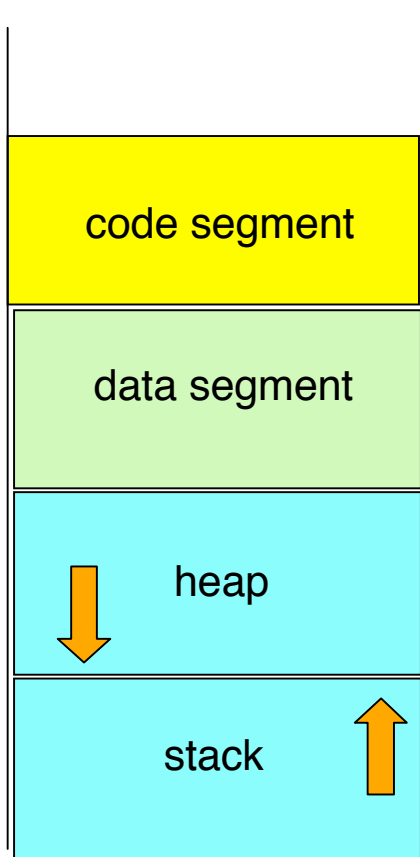


Un *errore di segmentazione* ha luogo quando

- ❑ un programma tenta di **accedere** ad una posizione di **memoria** alla quale **non gli è permesso accedere**
- ❑ un programma tenta di **accedere** ad una posizione di **memoria** in una **maniera** che non gli è **concessa** (ad esempio, **scrivere su una posizione di sola lettura**, oppure **sovrascrivere parte del sistema operativo**)

Segmentation Fault

Un **errore di segmentazione** (in inglese **segmentation fault**, spesso abbreviato in **segfault**) è una particolare condizione di errore che può verificarsi durante l'esecuzione di un programma.



```
main() {  
    int *aptr;  
    int *bptr;  
    int x = 5;  
    aptr = &x;  
  
    /* Increment the pointer by one,  
       making it misaligned */  
  
    bptr = aptr + sizeof(int);  
  
    /* Dereference it as an int pointer,  
       causing an unaligned access */  
  
    *bptr = 42;  
  
    printf("Done\n");  
}
```

Variabili Static

E' possibile imporre che una **variabile locale** di una funzione abbia un tempo di vita pari al tempo di esecuzione dell'**intero programma**, utilizzando il qualificatore **static**:

```
void f()  
{    static int cont=0;  
    ...  
}
```

la variabile **static int cont**:

- ❑ è creata all'inizio del programma, inizializzata a **0**, e deallocata alla fine dell'esecuzione;
- ❑ la sua visibilità è limitata al corpo della funzione **f**,
- ❑ il suo tempo di vita è pari al tempo di esecuzione dell'intero programma
- ❑ è allocata nell'area dati globale (**data segment**)

Esempio: Variabili Static

```
#include <stdio.h>

int f()
{ static int cont=0;
    cont++;
    return cont;
}

main()
{    printf("%d\n", f());
    printf("%d\n", f());
}
```

la variabile **static int cont**, è allocata all'inizio del programma e deallocata alla fine dell'esecuzione:

- ❑ essa persiste tra una attivazione di **f** e la successiva
 - la prima **printf** stampa **1**,
 - la seconda **printf** stampa **2**

La Ricorsione

Una *funzione matematica* è definita **ricorsivamente** quando nella sua definizione compare un **referimento a se stessa**

La **ricorsione** consiste nella possibilità di **definire** una **funzione mediante 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 **$n+1$** (**PASSO INDUTTIVO**)
- ❑ 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**” ($n = n_0$) in cui la soluzione sia 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).

Esempio: La Ricorsione

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 C

In linguaggio C è possibile definire funzioni **ricorsive**:

- Il corpo di ogni funzione ricorsiva contiene almeno una chiamata alla funzione stessa.

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

La Ricorsione in C

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

La Ricorsione in C

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

La funzione fact lega il parametro n a 3. Essendo 3 positivo si passa al ramo else. Per calcolare il risultato della funzione e' necessario effettuare una nuova chiamata di funzione fact(2)

```
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);  
}
```

La Ricorsione in C

Il nuovo servitore lega il parametro n a 2. Essendo 2 positivo si passa al ramo else. Per calcolare il risultato della funzione e' necessario effettuare una nuova chiamata di funzione. n-1 nell'environment di fact vale 1 quindi viene chiamata fact(1)

- **Servitore & Cliente:**

```
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);  
}
```

La Ricorsione in C

Il nuovo servitore lega il parametro n a 1. Essendo 1 positivo si passa al ramo else. Per calcolare il risultato della funzione e' necessario effettuare una nuova chiamata di funzione. n-1 nell'environment di fact vale 0 quindi viene chiamata fact(0)

- **Servitore & Cliente:**

```
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);  
}
```

La Ricorsione in C

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

- **Servitore & Cliente:**

```
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);  
}
```


La Ricorsione in C

- **Servitore & Cliente:**

```
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);  
}
```

*Il controllo torna al servitore precedente `fact(1)` che puo' valutare l'espressione $n * 1$ (valutando n nel suo environment dove vale 1) ottenendo come risultato 1 e terminando.*

La Ricorsione in C

*Il controllo torna al servitore precedente `fact(2)` che puo' valutare l'espressione $n * 1$ (valutando n nel suo environment dove vale 2) ottenendo come risultato 2 e terminando.*

- **Servitore & Cliente:**

```
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);  
}
```

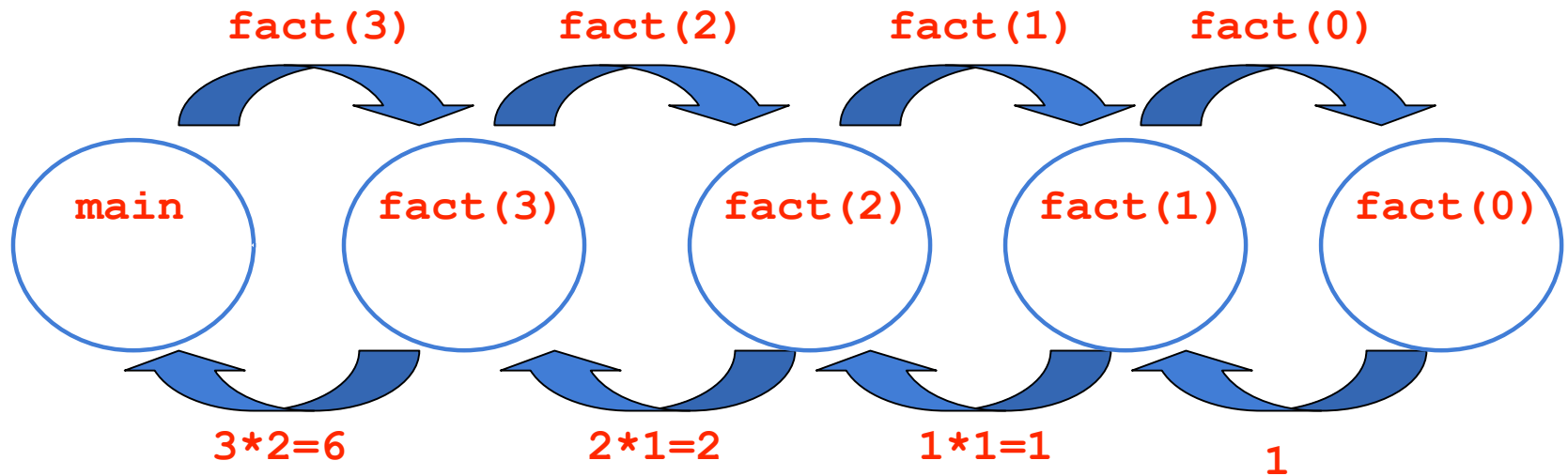
La Ricorsione in C

*Il controllo torna al servitore precedente
fact(3) che puo' valutare
l'espressione $n * 2$ (valutando n nel suo
environment dove vale 3) ottenendo
come risultato 6 e terminando.
IL CONTROLLO PASSA AL MAIN CHE
ASSEGNA A fz IL VALORE 6*

- **Servitore & Cliente:**

```
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);  
}
```

La Ricorsione in C



main

fact(3)

fact(2)

fact(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

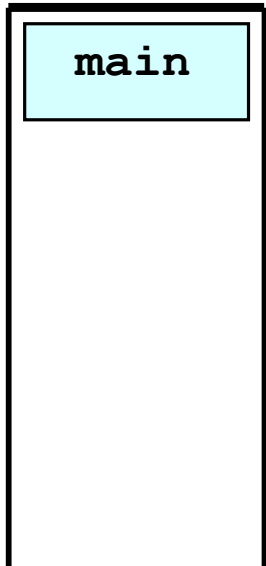
```
int fact(int n) {  
    if (n==0) return 1;  
    else return n*fact(n-1);  
}
```

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

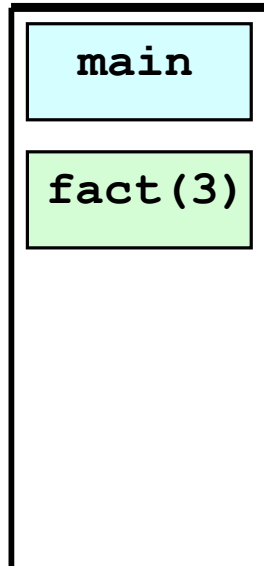
NOTA: Anche il
`main()` e' una funzione

Cosa succede nello stack

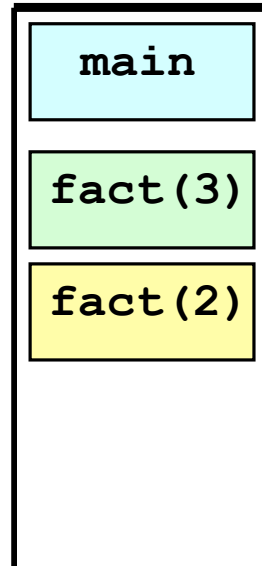
Situazione
iniziale



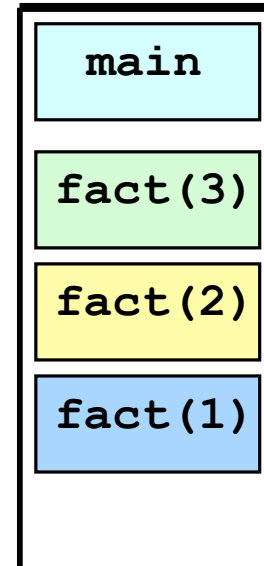
Il `main()`
chiama
`fact(3)`



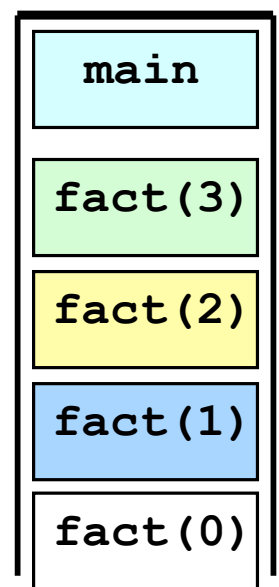
`fact(3)`
chiama
`fact(2)`



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

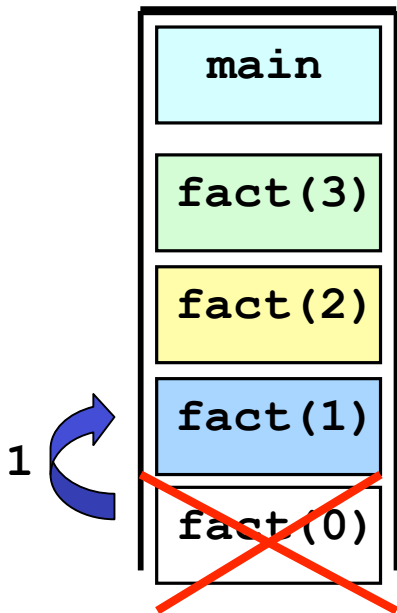


`fact(1)`
chiama
`fact(0)`

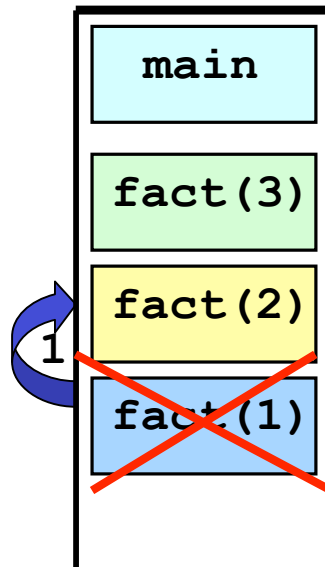


Cosa succede nello stack

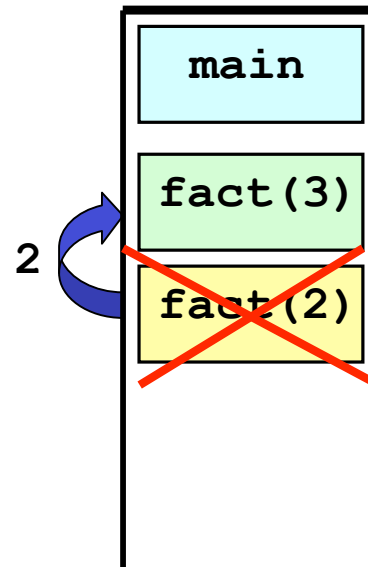
`fact(0)` termina restituendo il valore 1. Il controllo torna a `fact(1)`



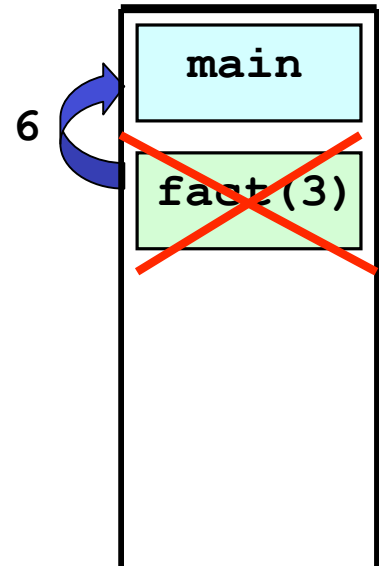
`fact(1)` effettua la moltiplicazione e termina restituendo il valore 1. Il controllo torna a `fact(2)`



`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`.



Esempio: Somma dei primi N numeri naturali

Problema:

calcolare la somma dei primi N naturali

Algoritmo ricorsivo:

Somma: $N \rightarrow N$

{	Somma(n)	vale 1	se $n == 1$
	Somma(n)	vale $n + \text{Somma}(n-1)$	se $n > 0$

Esempio: Somma dei primi N numeri naturali

Codifica:

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

Esempio: Somma dei primi N numeri naturali

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

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

```
main() {  
    int dato;  
    printf("dammi un intero positivo: ");  
    scanf("%d", &dato);  
    if (dato>0) printf("Risultato: %d", sommaFinoA(dato));  
    else printf("ERRORE!");  
}
```

Esercizio: seguire l'evoluzione dello stack nel caso in cui dato=4.

Calcolo iterativo del fattoriale

```
int fact(int n) {  
    int i;  
    int F=1; /*inizializzazione del fattoriale*/  
    for (i=2; i <= n; i++)  
        F=F*i;  
    return F;  
}
```

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

Calcolo iterativo del fattoriale

```
int fact(int n) {  
    int i;  
    int F=1; /*inizializzazione del fattoriale*/  
    for (i=2; i <= n; i++)  
        F=F*i;  
    return F;  
}
```

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

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

Processo Computazionale Iterativo

Nell'esempio precedente il risultato viene sintetizzato “**in avanti**”

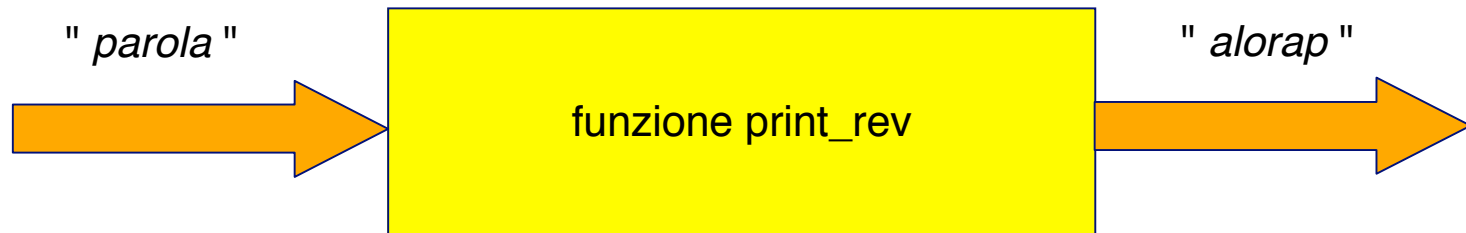
L'esecuzione di un algoritmo di calcolo che computi “**in avanti**”, **per accumulo**, è un **processo computazionale iterativo**.

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.

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 (o array di caratteri).




Esercizio

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

```
void print_rev(char car) ;
{ char c;
  if (car != '.')
  {   scanf("%c", &c);
      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

Esercizio

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

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

```
main() {  
    char k;  
    printf("\nIntrodurre una sequenza terminata da .:\t");  
    scanf("%c", &k);  
    print_rev(k);  
    printf("\n*** FINE ***\n");  
}
```


Stack

main

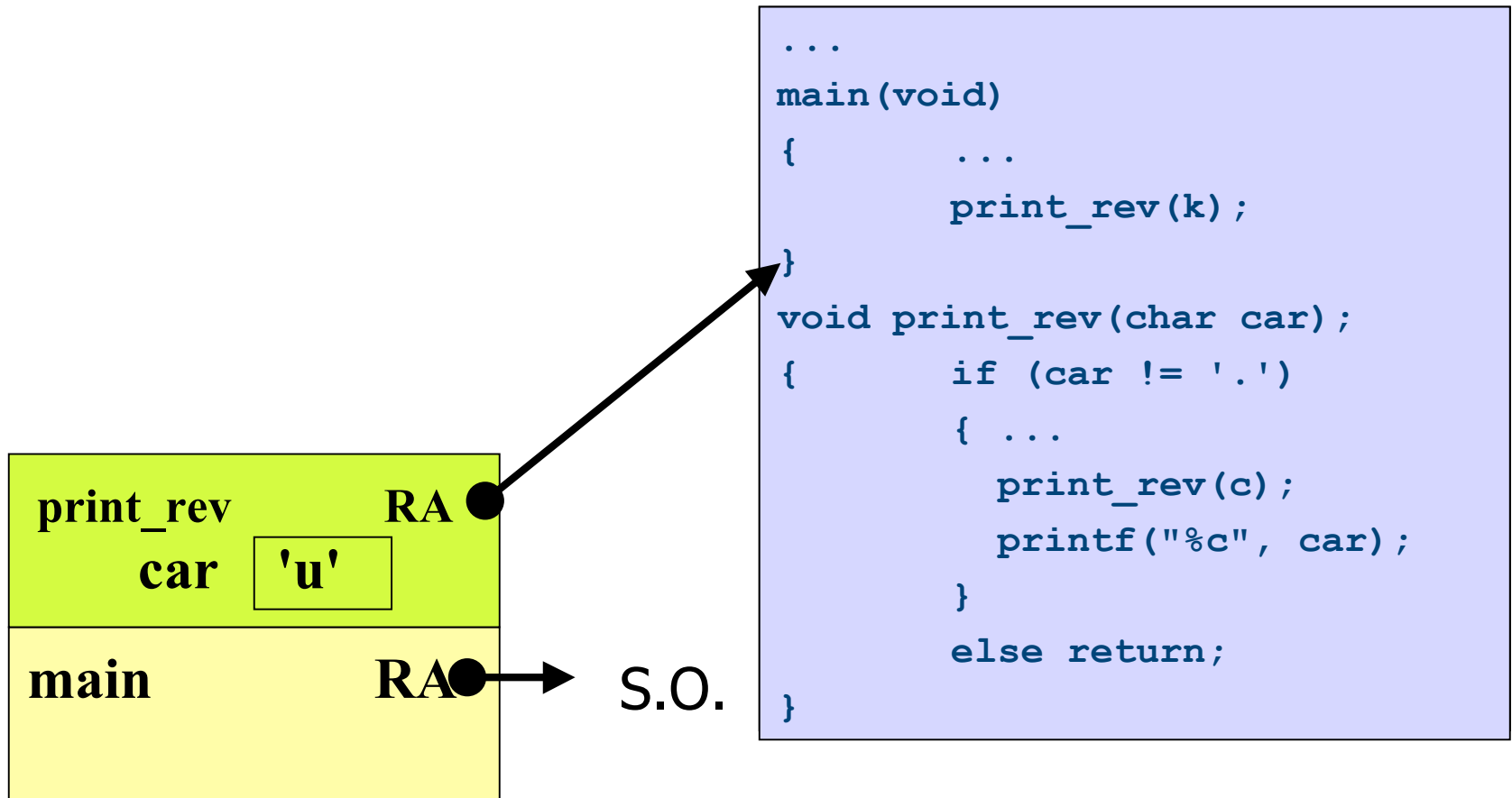
RA ● → **S.O.**

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

Standard Input:

"uno."

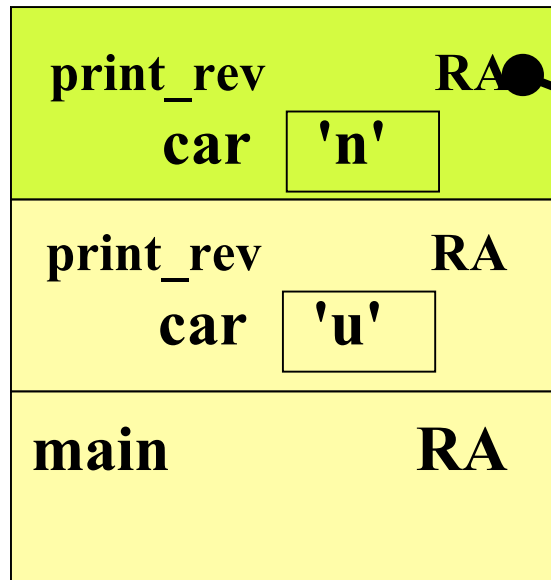
Stack



Standard Input:

"u"no."

Stack

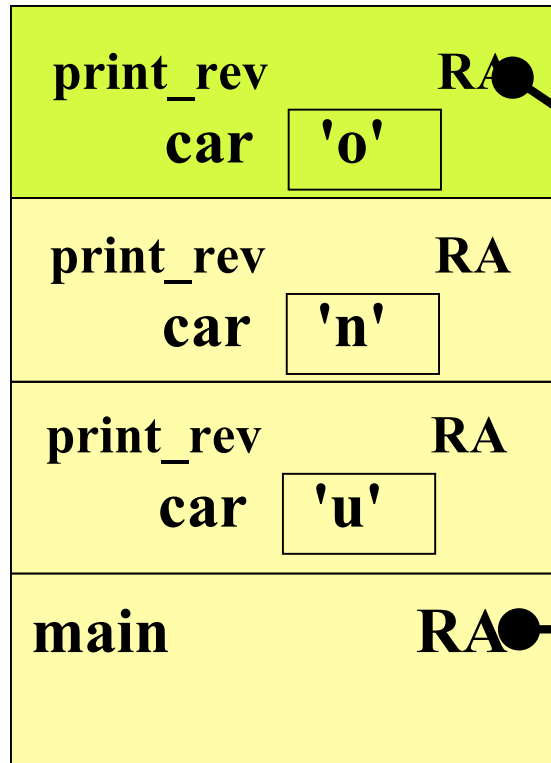


```
...  
main(void)  
{  
    ...  
    print_rev(k);  
}  
void print_rev(char car);  
{  
    if (car != '.')  
    { ...  
        print_rev(c);  
        printf("%c", car);  
    }  
    else return;  
}
```

Standard Input:

"uno."

Stack



```
...  
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

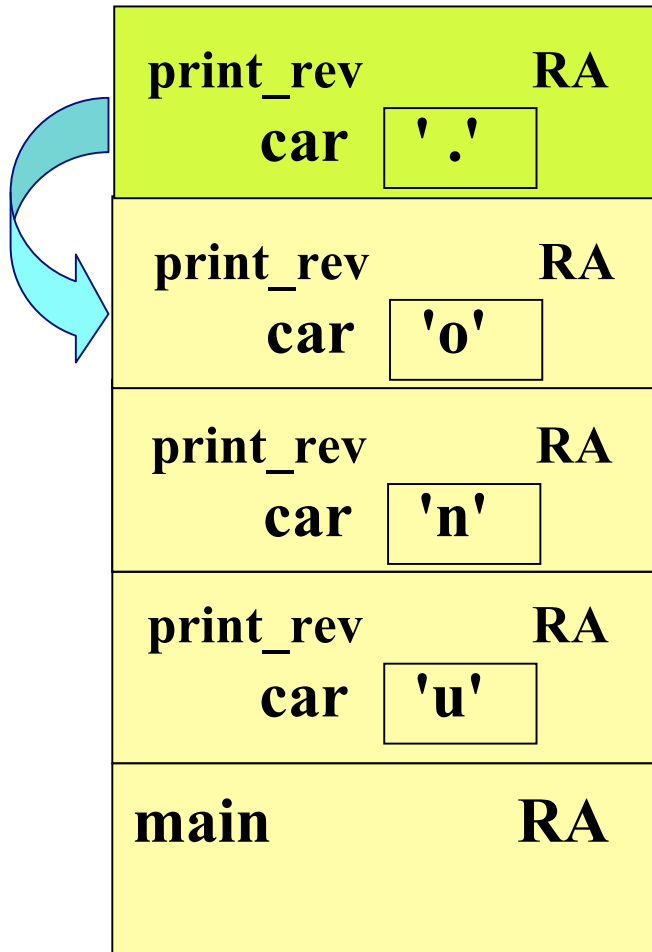
```
...
main(void)
{
    ...
    print_rev(k);
}
void print_rev(char car);
{
    if (car != '.')
    { ...
        print_rev(c);
        printf("%c", car);
    }
    else return;
}
```

Standard Input:

"uno."

Stack

Codice



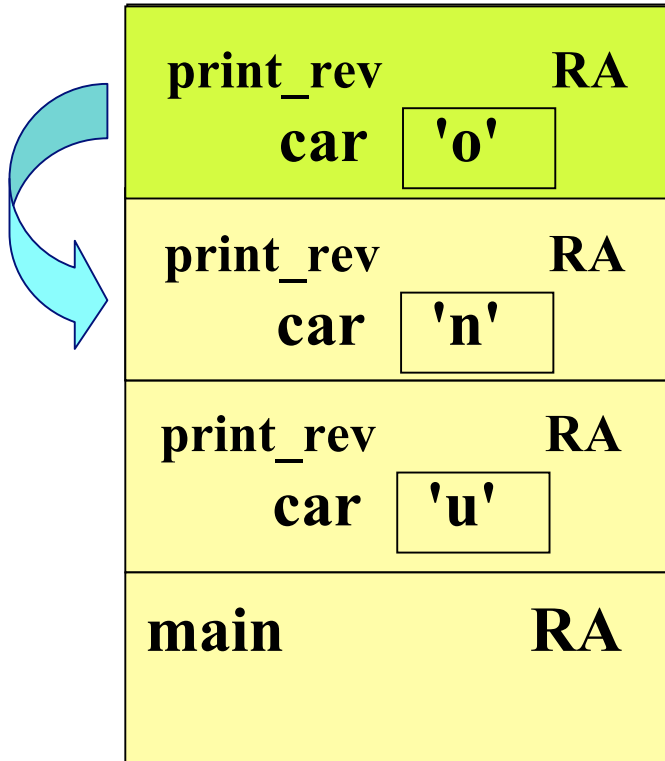
```
...
main(void)
{
    ...
    print_rev(k);
}

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:

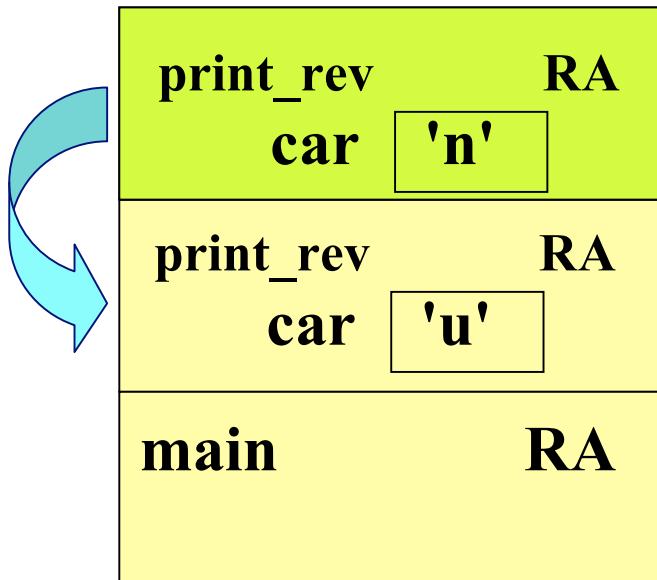
"o"

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;
}
```

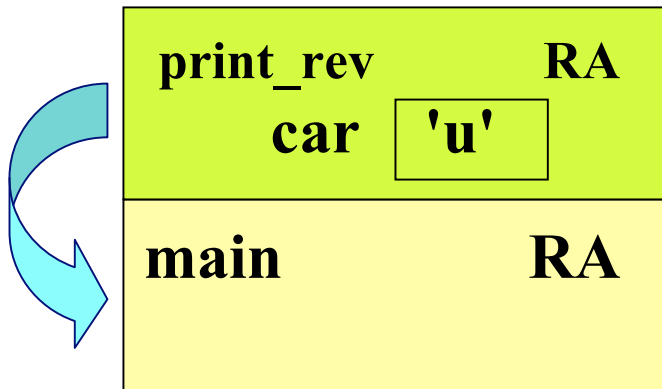
Standard output:
"on"

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

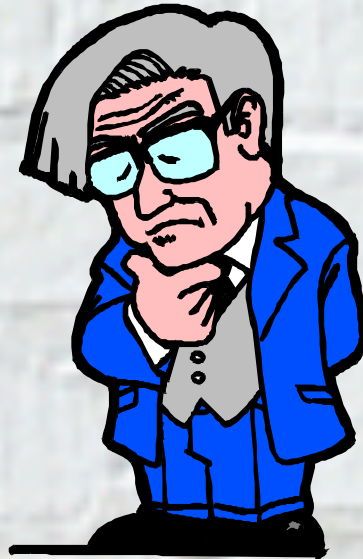
RA

Standard output:
"onu"

Standard Input:
"uno."

Corso di Algoritmi e Strutture Dati

APPUNTI SUL LINGUAGGIO C



Stack e Ricorsione

FINE