# Il linguaggio C

## Funzioni

# Funzioni C

- A cosa servono ?
  - a raccogliere il codice comune a più parti del programma per poterlo defininire e mettere a punto una volta per tutte ...
    - Es: ho replicato 37 volte il codice per il calcolo delle radici di un polinomio e mi accorgo che c'è un piccolo errore ....
  - A mettere a disposizione ad altri codice che fa qualcosa di utile attraverso il meccanismo delle librerie (es. **`printf(), sqrt(),`** ...)
    - Vedremo come fare più avanti, richiedono la creazione della libreria e del file header (il .h) corrispondenti

# Funzioni C

## Come si realizzano ?

- Attraverso la definizione di funzione :
  - la definizione di una funzione è costituita da:

    1)una intestazione (*head*) che fornisce il tipo ed il nome dei parametri da passare alla funzione (parametri formali), ed il tipo del valore restituito
    - es: $f(x, y) = x + y$

intestazione

```
int somma (int x, int y) {
   return (x + y);
}
```

# Funzioni C

## Come si realizzano ?

- Attraverso la definizione di funzione :
  - la definizione di una funzione è costituita da:

    2) un corpo (*body*) costituito da un blocco che specifica le istruzioni da eseguire

    - L'istruzione **`return`** permette di specificare

    quale valore restituire come risultato della funzione

    - es: $f(x, y) = x + y$

```
int somma (int x, int y) {
   return (x + y);
}
```

corpo

# Funzioni C

## Come si utilizzano?

```
int somma (int x, int y) {
  return (x + y);
}
int main (void) {
  int a = 3, b = 5, c;
  a = somma(10, a+b+1);
  c = somma (a,b)
  printf("Il risultato è %d \n", c);
  return 0;
}
```

# Funzioni C

## Come si utilizzano?

- Attraverso la chiamata di funzione :
  - La chiamata di funzione permette di eseguire le istruzioni contenute nel corpo di una funzione fornendo dei valori ai parametri formali
  - Basta utilizzare il nome della funzione e fornire una espressione (parametro attuale) per ogni parametro formale

    Es: `int a = 3;  int b = 5; ....`

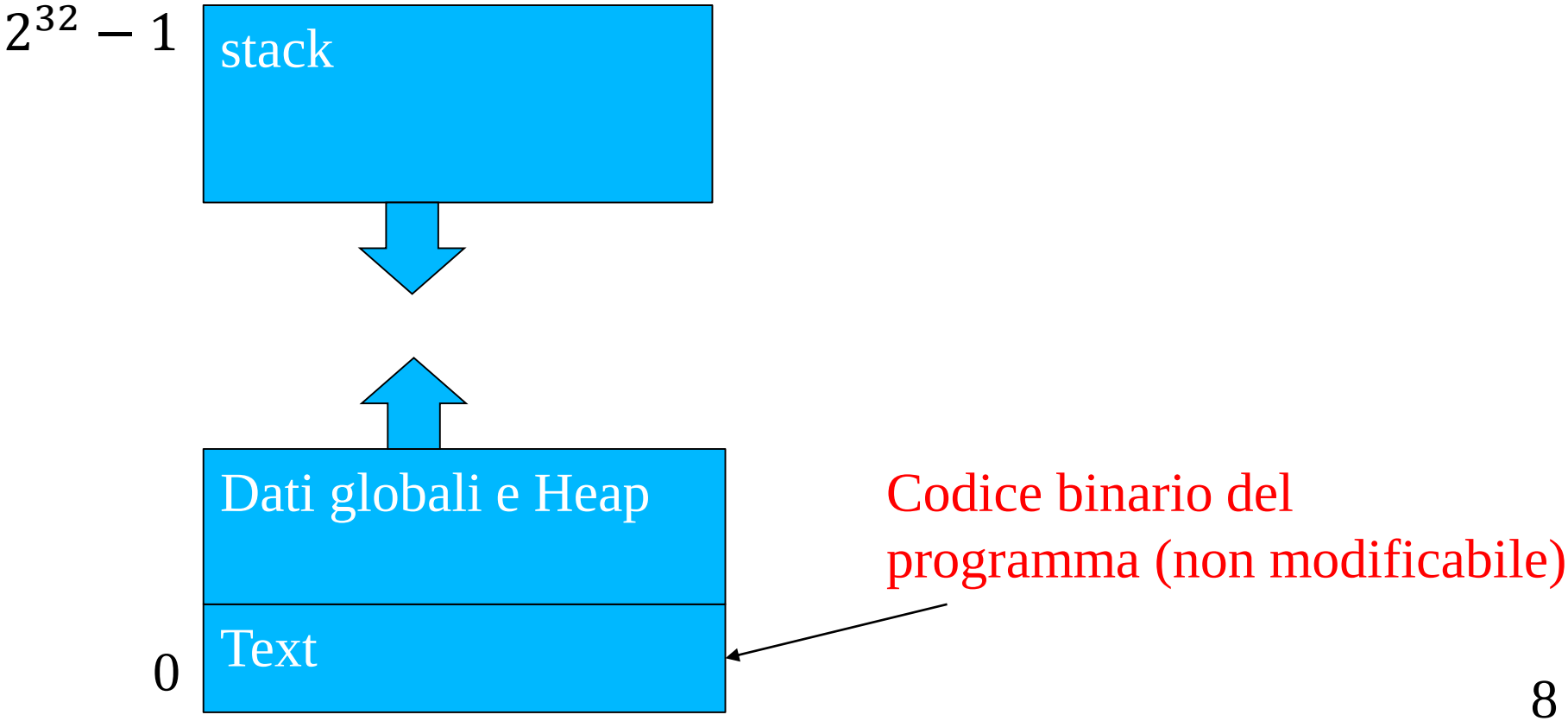
    `a = somma(10, a+b+1);`
  - Ogni parametro attuale viene valutato ed il valore assegnato al parametro formale corrispondente prima di eseguire il corpo

# Esempio: funzione max

```
#include <stdio.h>

int max (int a, int b) {
    int tmp;
    if (a < b) tmp = b; else tmp = a;
    return tmp;
} /* può essere utilizzata da qua in poi */

int main (void){
   int x;
   x = max(10,2);
   printf("Il massimo è %d \n",x);
   return 0;
}
```

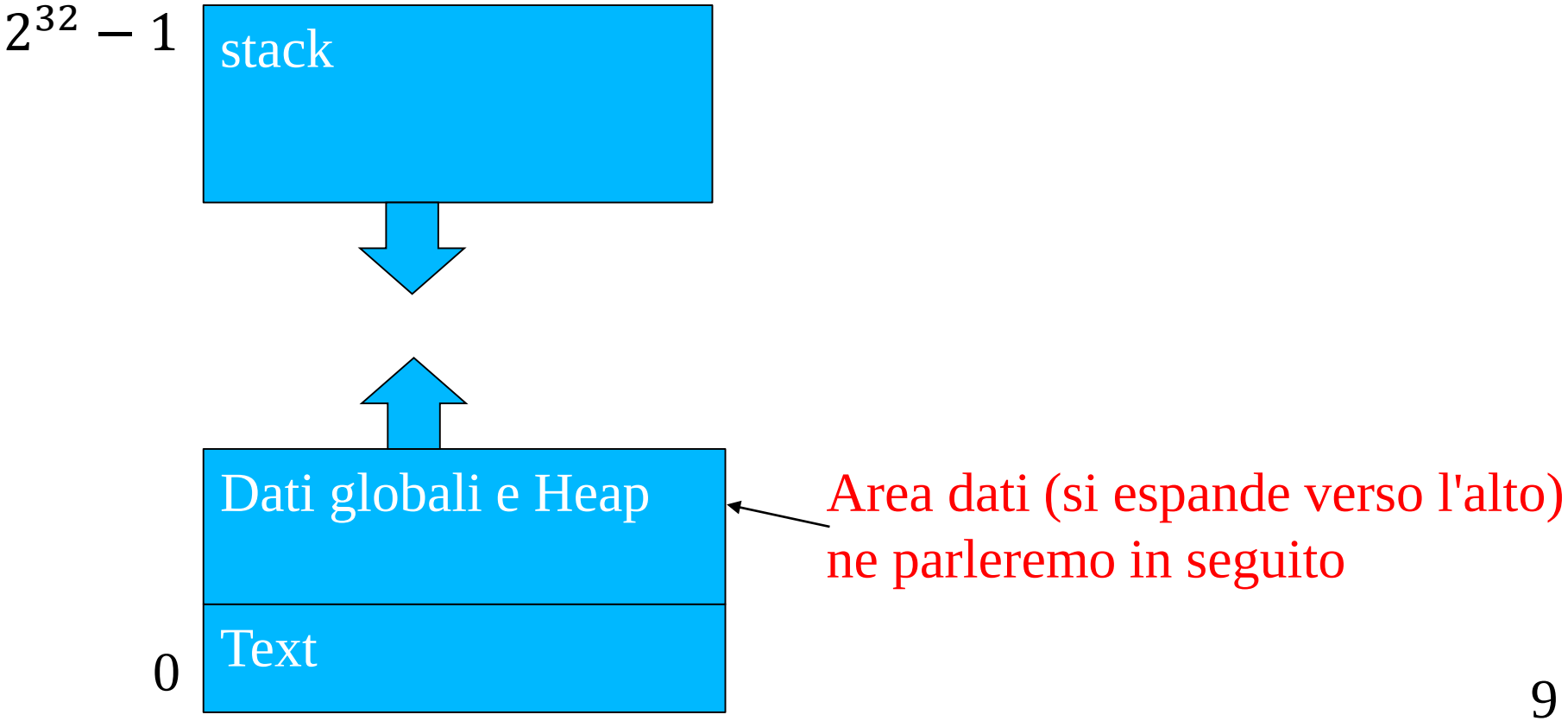
# Funzioni C

- Cosa succede veramente quando viene eseguita una funzione ?
  - come vede la memoria un programma C in esecuzione ?



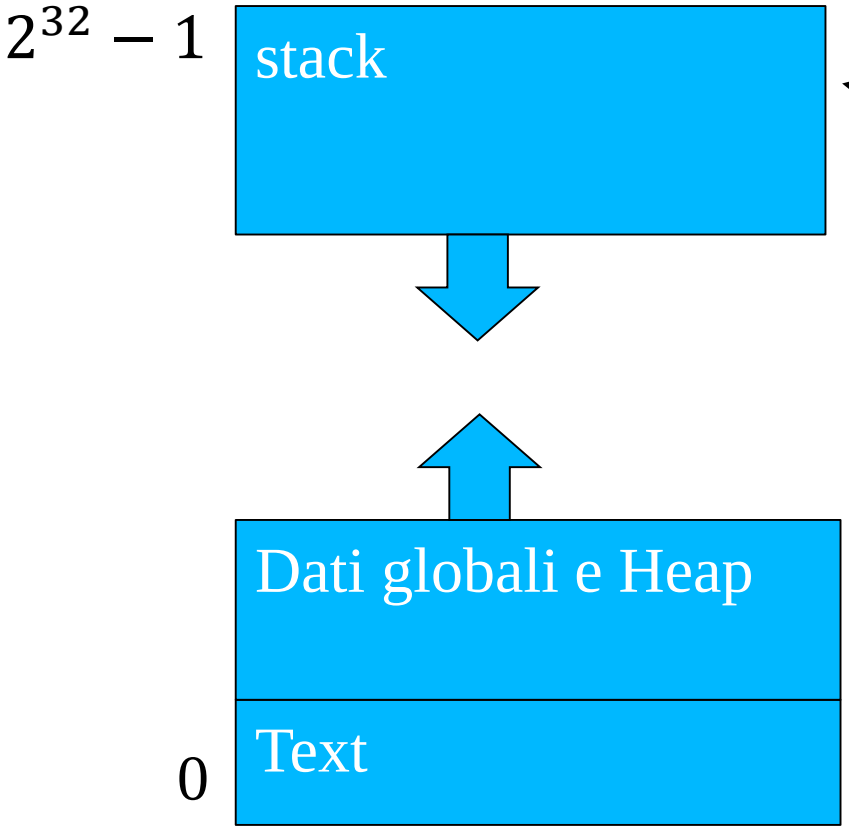
# Funzioni C

- Cosa succede veramente quando viene eseguita una funzione ?
  - come vede la memoria un programma C in esecuzione ?



# Funzioni C

- Cosa succede veramente quando viene eseguita una funzione ?
  - come vede la memoria un programma C in esecuzione ?




Area utilizzata per realizzare le esecuzioni delle funzioni attraverso delle strutture (*frame*) Che contengono:

- Le variabili della funzione
- I *parametri attuali* (copia dei valori con cui viene attivata)
- Indirizzo da dove ricominciare quando l'esecuzione è finita (*indirizzo di ritorno*)

# Frame: Esempio

```
#include <stdio.h>

int max (int a, int b) {
    int tmp;
    if (a < b) tmp = b; else tmp = a;
    return tmp;
}

int main (void){
   int x;
   x = max(10,2);
   printf("Il massimo è %d \n", x);
   return 0;
}
```

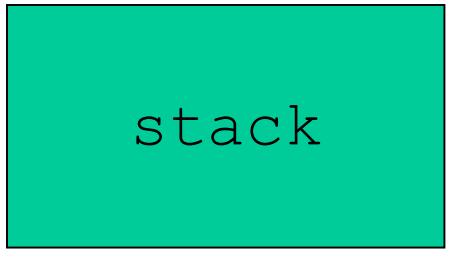
# Frame: esempio

```
#include <stdio.h>

int max (int a, int b) {
    int tmp;
 if (a < b) tmp = b; else tmp = a;
    return tmp;
}

int main (void){
   int x;
   x = max(10,2);
   printf("Il massimo è %d \n", x);
   return 0;
}
```

Parametri attuali il cui valore è copiato sullo stack

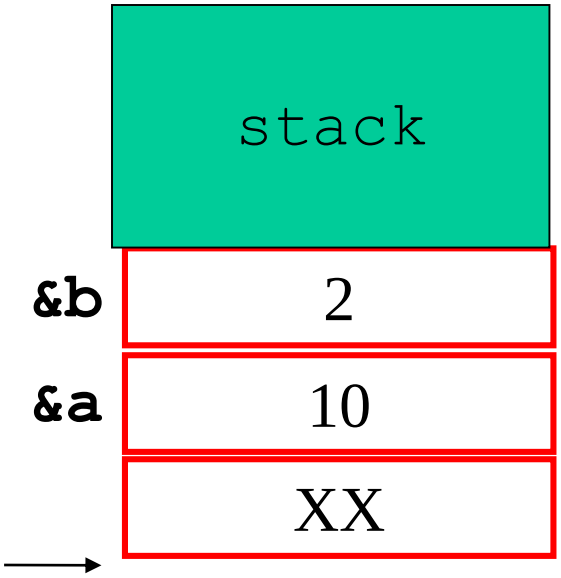
# Frame: esempio

```
    x = max(10,2);
XX: printf("Il massimo è %d \n", x);
    return 0;
}

int max (int a, int b) {
    int tmp;
    if (a < b) tmp = b;
    else tmp = a;
    return tmp;
}
```



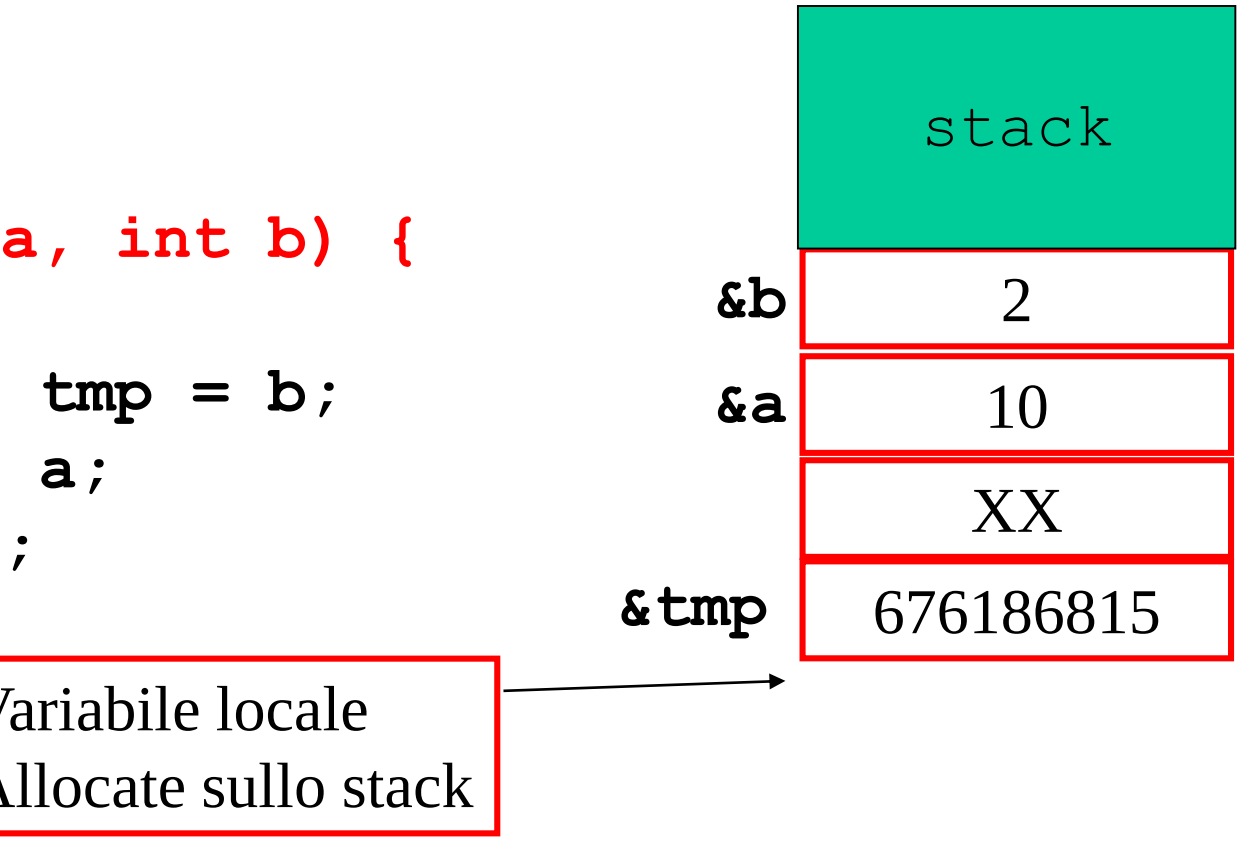
# Frame: esempio

```
    x = max(10,2);
XX: printf("Il massimo è %d \n", x);
    return 0;
}

int max (int a, int b) {
    int tmp;
    if (a < b) tmp = b;
    else tmp = a;
    return tmp;
}
```



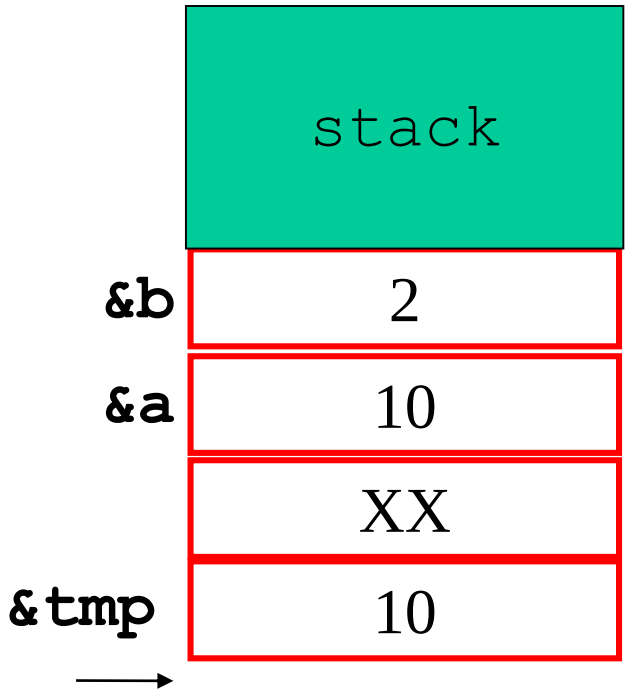
# Frame: esempio

```
x = max(10,2);
XX: printf("Il massimo è %d \n", x);
    return 0;
}

int max (int a, int b) {
    int tmp;
    if (a < b) tmp = b;
    else tmp = a;
    return tmp;
}
```



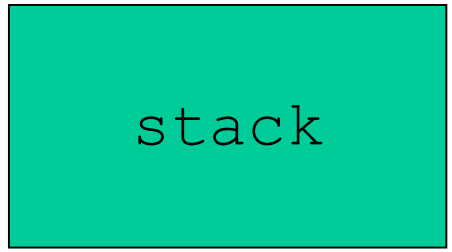
# Frame: esempio

```
   x = max(10,2);
XX: printf("Il massimo è %d \n", x);
    return 0;
}

int max (int a, int b) {
    int tmp;
    if (a < b) tmp = b;
    else tmp = a;
    return tmp;
}
```



# Frame: esempio

```
    x = max(10,2);
XX: printf("Il massimo è %d \n", x);
    return 0;
}

int max (int a, int b) {
    int tmp;
    if (a < b) tmp = b;
    else tmp = a;

    return tmp;
}
```



# Funzioni C : prototipi

- E se voglio usare la funzione prima di averla definita ?
  - Posso usare la dichiarazione di funzioni (*prototipo*):
  - Anticipa l'intestazione della funzione: nome della funzione, il tipo del valore restituito ed il tipo di tutti i parametri formali utilizzati

# Prototipo: Esempio

```
#include <stdio.h>

int max (int a, int b); /* prototipo */

int main (void){
  printf("Il massimo è %d \n", max(10,2));
  return 0;
}

int max (int a, int b) {
   int tmp;
   if (a < b) tmp = b;
   else tmp = a;
   return tmp;
}
```

# Prototipo: esempio

- prototipo della funzione di somma di due interi

```
int somma (int, int);
```

oppure

```
int somma (int x, int y);
```

**x, y** sono inutili, ma vengono convenzionalmente specificati per documentazione

# Funzioni C: commenti

- È molto importante commentare e documentare le funzioni
  - Format che useremo nel corso

```
/** breve descrizione della funzione
 \param x significato del parametro
 \param y significato del parametro
  .....
 \retval n valore restituito se ...
 \retval e valore restituito se ...
  ......
 altre informazioni utili */
```

# Funzioni C: commenti

- Esempio: funzione polinomio ....

```
/** calcola le radici di a*x^2 + b*x + c




*/
double polinomio(double a, double b, double c);
```

# Funzioni C: commenti

- Esempio: funzione polinomio ....

```
/** calcola le radici di a*x^2 + b*x + c
 \param a coeff. Grado 2
 \param b coeff. Grado 1
 \param c coeff. Grado 0
  .....
 \retval r_1 la radice reale maggiore (Se il polinomio
  ha radici reali)
 \retval 0 se il polinomio non ha radici reali
*/
double polinomio(double a, double b, double c);
```

# Funzioni C : variabili

- Le variabili dichiarate all'inizio del blocco che definisce il corpo della funzione sono dette *locali* o *automatiche*:
  - sono allocate sullo stack,
  - sono accessibili  e visibili solo dentro il blocco in cui sono dichiarate (ed eventuali blocchi interni)
  - perdono il valore fra una esecuzione e l'altra del blocco dove sono dichiarate
- In C ci sono altri tipi di variabili !!!

# Funzioni C : variabili globali

- Le variabili globali sono variabili dichiarate al di fuori delle funzioni
  - sono accessibili  all'interno di tutte le funzioni che si trovano nello stesso file
  - sono allocate nell'area dati e vengono deallocate solo alla terminazione del programma
  - Le globali sono sconsigliate a meno di casi motivati!
    - E devono essere sempre adeguatamente documentate

# Esempio: variabile globale

```
#include <stdio.h>
int max (int a, int b);
int k = 0;                          /* var globale */

int main (void){
  printf("Il massimo è %d \n", max(10,2));
  printf("Il valore di k è %d \n", k);
  return 0;
}

int max (int a, int b) {
   k = k + 1;                       /* side effect */
   return (a < b)? b : a ;
}
```

# Esempio: variabile globale

Se compiliamo ed eseguiamo si ottiene:

```
$ ./max
Il massimo è 10
Il valore di k è 1
$
```

# Esempio: variabile globale

```
#include <stdio.h>
int max (int a, int b);
/** conta il numero di attivazioni
    della funzione */
int k = 0;
int main (void){
  extern k;
  printf("Il massimo è %d \n", max(10,2));
  printf("Il valore di k è %d \n", k);
  return 0;
}
int max (int a, int b) {
   extern k;
   k = k + 1;
   return (a < b)? b : a ;
}
```

# Funzioni C: variabile globale

```
/** ...
  \param ...

  \retval ...
 incrementa k, globale, ad ogni
 invocazione */
   int max (int a, int b) {
      extern k;
      k = k + 1;
      return (a < b)? b : a ;
   }
```

# Funzioni C: variabili static

- Sono variabile locali che mantengono il valore fra una invocazione e l'altra della funzione:
  - sono introdotte dalla parola chiave **`static`**
  - sono accessibili  all'interno del blocco in cui sono dichiarate
  - mantengono il valore fra una esecuzione e l'altra del blocco che le contiene
  - Sono allocate nell'area dati come le globali ma sono protette negli accessi
  - Devono essere documentate al solito ....

# Esempio: variabile statica

```
#include <stdio.h>

int max (int a, int b);
int main (void){
  printf("Il massimo è %d \n", max(10,2));
  /*k non è più accessibile fuori da max*/
  return 0;
}

int max (int a, int b) {
   static int k = 0;
   k++;
   printf("Il valore di k è %d \n", k);
   return (a < b)? b : a ;
}
```

# Esempio: variabile statica

Se compiliamo ed eseguiamo si ottiene un risultato simile ma le stampe sono invertite:

```
$ ./max
Il valore di k è 1
Il massimo è 10

$
```

# Tipica organizzazione di un file .c

```
/* direttive al preprocessore */
#include …
#define …
/* dichiarazioni di varibili globali*/
int k;
/* dichiarazione di funzioni (prototipi)*/
int somma (int x, int y);

int main (…) {… somma(4,5); … }

/* definizione di funzioni */
int somma (int x, int y) {….}
```

# Funzioni & tipo void

Combinando le funzioni con il tipo speciale **`void`** possiamo definire funzioni che non producono un valore ma producono delle modifiche dell'ambiente esterno...oppure fare a meno dei parametri

# Il tipo **`void`**

- Può essere utilizzato al posto degli usuali tipi nella intestazione/dichiarazione di una funzione.
  - Es:

```
double leggi_da_input (void);
void stampa_d (double x);
void stampa_versione (void);
```

- Usiamolo per fattorizzare codice che "fa delle cose" invece che solo calcolare valori

# Esempio 1

```
double leggi_da_input (void) {
  double tmp;
  printf("Inserisci un double:\n");
  scanf("%lf",&tmp);
  return tmp;
}

int main (void) {
  double x;
  x = leggi_da input();
  printf("%f\n\n",x);
  return 0;
}
```

# Esempio 1: output

Se compiliamo ed eseguiamo si ottiene :

```
$ ./leggi
Inserisci un double:
```

Inserendo 3.12 e invio ↓

```
3.12

$
```

- Infatti lo standard input e lo standard output sono condivisi da tutte le funzioni di uno stesso programma!

# Esempio 2

```
double leggi_da_input (void) {.... }
void stampa_d (double a) {
  printf("%f\n\n",a);
}
void stampa_versione (void) {
  printf("Versione: 1.1\n",a);
}
int main (void) {
  double x;
  x = leggi_da input(); stampa_d(x);
  stampa_versione();
  return 0;
}
```

# Esempio 2: output

Se compiliamo ed eseguiamo si ottiene :

```
$ ./leggi
Inserisci un double:
```

Inserendo 3.12 e invio ↓

```
3.12
3.12

Versione: 1.1
$
```

# Esempio 2

```
double leggi_da_input (void) {.... }
void stampa_d (double a) {
  printf("%f\n\n",a);
  /* manca l'istruzione di return !!!!*/
}
void stampa_versione (void) {
  printf("Versione: 1.1\n",a); }
int main (void) {
  double x;
  x = leggi_da input(); stampa_d(x);
  stampa_versione(x);
  return 0;
}
```

# Funzioni Ricorsive

Ovvero funzioni che richiamano se stesse...

A cosa servono, come funzionano ed esempi....

# Introduzione

- Quasi tutti i linguaggi di programmazione permettono ad una funzione di richiamare se stessa dentro il body che la definisce ...
- Vediamo con un esempio:
  - che definizioni che usano la ricorsione ci sono già note dalla matematica
  - e che la cosa può servire anche nella programmazione
  - E daremo l'intuizione del funzionamento

# Fattoriale

Consideriamo la definizione:

$$n! = 1 * \cdots * n = \prod_{i=1}^{n} i$$

è possibile fornire una definizione induttiva:

$$n! = \begin{cases} 1 & se\ n = 1 \\ n * (n-1)! & altrimenti \end{cases}$$

- Di fatto questa seconda definizione riusa se stessa al suo interno, possiamo quindi vederla come un primo esempio di ricorsione
- In questo caso supponiamo di avere già definito la funzione per *n-1* e spieghiamo come usare questo per definire la funzione per *n*

# Fattoriale

```
/* definizione iterativa */
int fattoriale(int n) {
   int i, fatt=1;
   for (i = 2; i <=n ; i++)
     fatt *=i;
   return fatt;
}
```

# Fattoriale

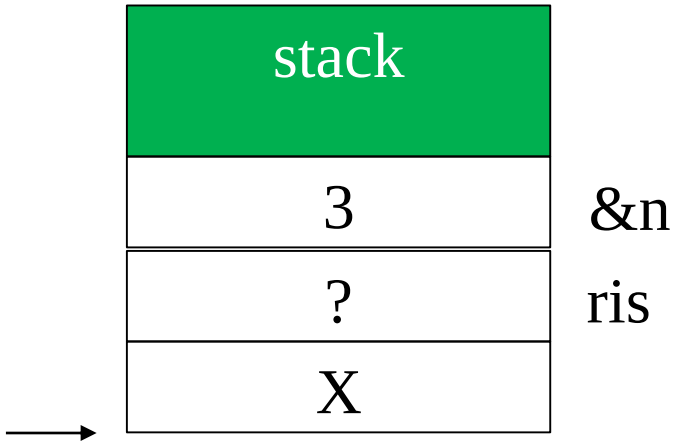
```
/* definizione iterativa */
int fattoriale(int n) {
  int i, fatt=1;
  for (i = 2; i <=n ; i++)
    fatt *=i;
  return fatt;
}
/* definizione ricorsiva */
int fattoriale_r(int n) {
  if ( n == 1 ) return 1;
  return (n * fattoriale_r(n-1));
}
```

# Fattoriale: lo stack

```
/* definizione ricorsiva */
int fattoriale_r(int n) {
   if ( n == 1 ) return 1;
   return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```

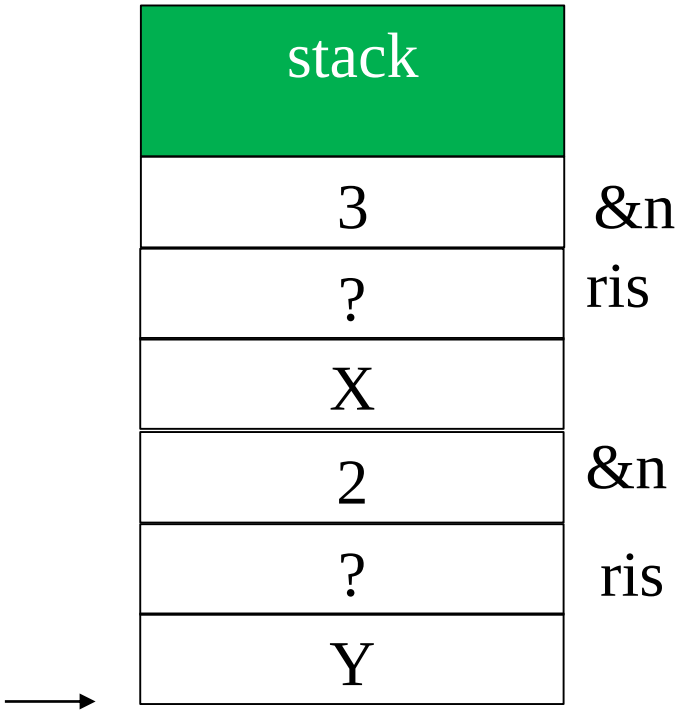
# Fattoriale: lo stack

```
/* definizione ricorsiva */
int fattoriale_r(int n) {
   if ( n == 1 ) return 1;
   return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```



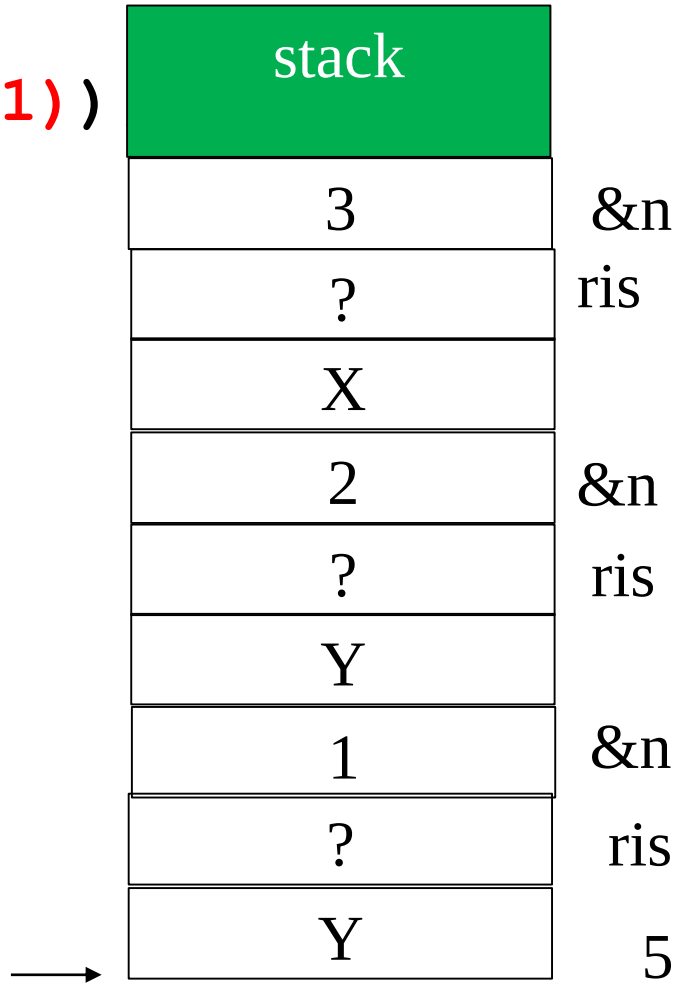
# Fattoriale: lo stack

```
/* definizione ricorsiva */
int fattoriale_r(int n) {
  if ( n == 1 ) return 1;
Y:  return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```



# Fattoriale: lo stack

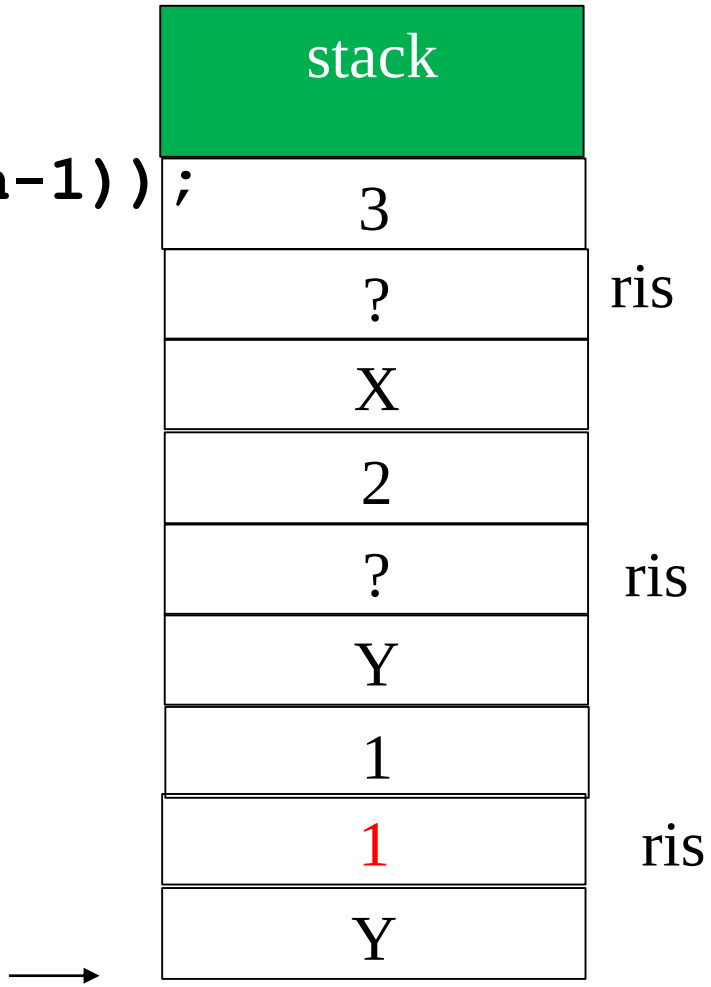
```
/* definizione ricorsiva */
int fattoriale_r(int n) {
  if ( n == 1 ) return 1;
Y:return (n * fattoriale_r(n-1))
}
...
X: a = fattoriale_r(3);
...
```



# Fattoriale: lo stack

```
/* definizione ricorsiva */
int fattoriale_r(int n) {
   if ( n == 1 ) return 1;
Y:return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```

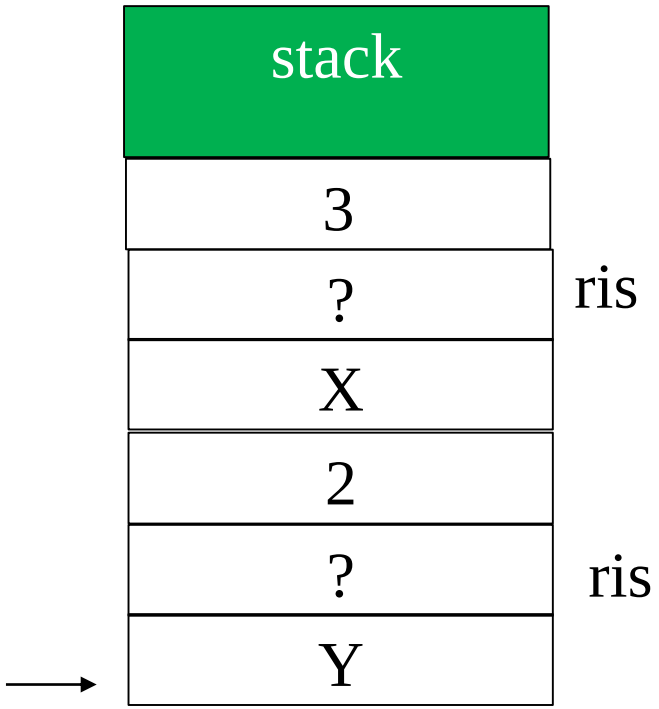
Abbiamo raggiunto il caso base, si ritorna il valore 1



# Fattoriale: lo stack

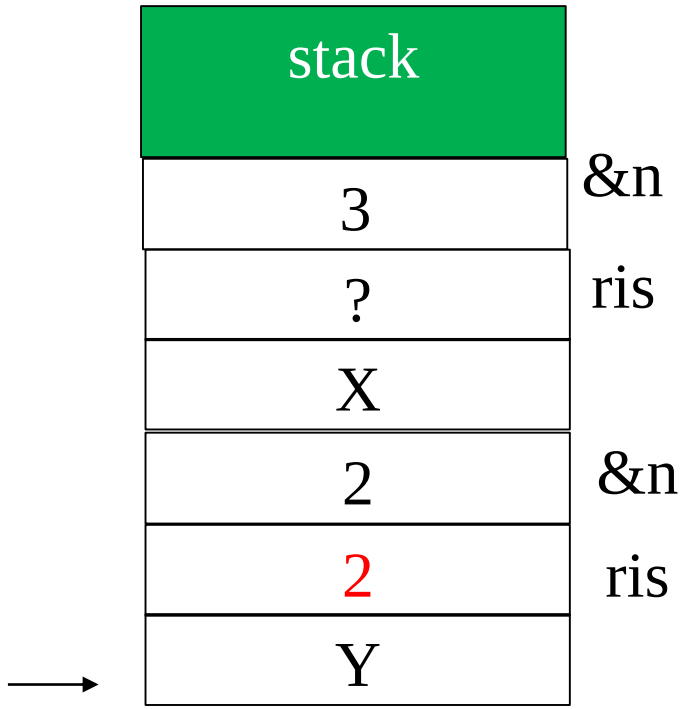
```
/* definizione ricorsiva */
int fattoriale_r(int n) {
   if ( n == 1 ) return 1;
Y:   return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```

si libera lo stack ritornando ad eseguire da Y questa chiamata viene sostituita dal valore 1



# Fattoriale: lo stack

```
/* definizione ricorsiva */
int fattoriale_r(int n) {
  if ( n == 1 ) return 1;
Y:  return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```

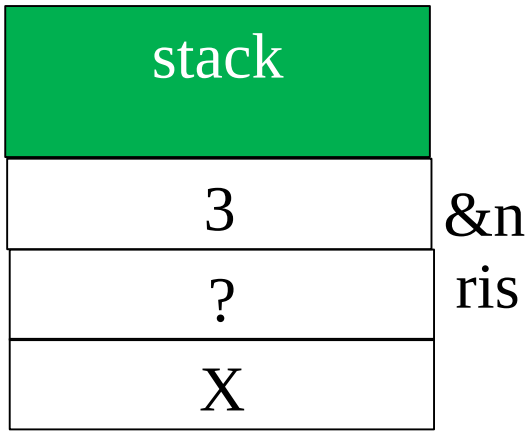



Calcoliamo 2 * 1 e ritorniamo 2 liberando lo stack, ricominciamo ancora ad eseguire da Y

# Fattoriale: lo stack

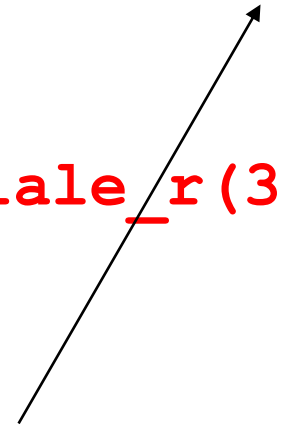
```
/* definizione ricorsiva */
int fattoriale_r(int n) {
   if ( n == 1 ) return 1;
Y:   return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```

Questa chiamata viene sostituita da 2

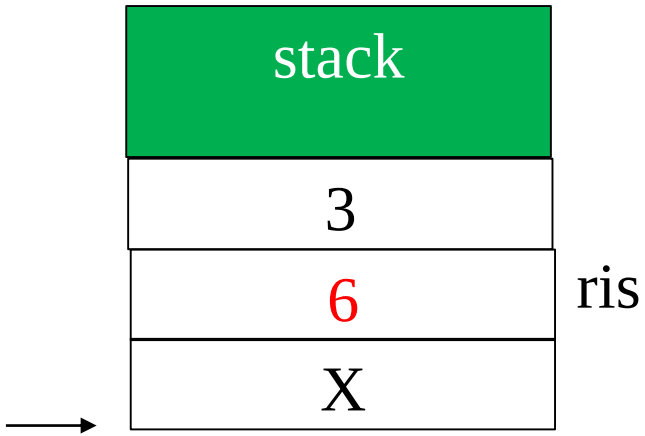


# Fattoriale: lo stack

```
/* definizione ricorsiva */
int fattoriale_r(int n) {
   if ( n == 1 ) return 1;
Y:  return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```

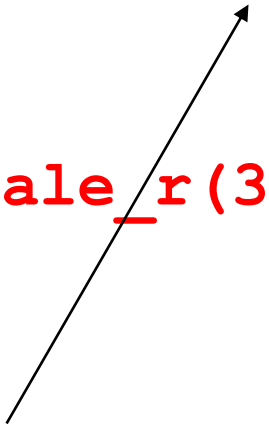


Calcoliamo 3 * 2 = 6



# Fattoriale: lo stack

```
/* definizione ricorsiva */
int fattoriale_r(int n) {
   if ( n == 1 ) return 1;
Y:  return (n * fattoriale_r(n-1));
}
...
X: a = fattoriale_r(3);
...
```




Liberiamo lo stack e ricominciamo
ad eseguire da X assegnando 6 ad **a**

# Programmazione ricorsiva

Quindi:

- la programmazione ricorsiva si basa sull'osservazione che a volte la risoluzione di un problema si può ridurre alla risoluzione di istanze più semplici dello stesso problema combinando i risultati poi in qualche modo
- Come per *l'induzione ben fondata*, è fondamentale che via via il problema si semplifichi in modo da raggiungere un caso base risolubile in modo diretto, altrimenti possiamo non terminare mai!

# Esempio 2: invertire una sequenza di caratteri

- Voglio leggere una sequenza di caratteri da standard input (terminata da \n) e stamparla rovesciata su standard output

**Casa**  **asaC**

Come possiamo comportarci ?

(usando la ricorsione ci possiamo riuscire senza usare array ....)

# Esempio 2: invertire una sequenza di caratteri

- Osserviamo che se so come elaborare correttamente una sequenza di n-1 cifre per stampare una sequenza lunga n posso:
  - Stampare l'ultimo carattere

    **Casa** → **a**
  - Chiamare ricorsivamente la stessa funzione per stampare gli *n-1* caratteri precedenti

    **Casa** → **asaC**
  - In questo caso il caso base è la sequenza vuota per cui non dobbiamo fare niente

# Inversione di una stringa

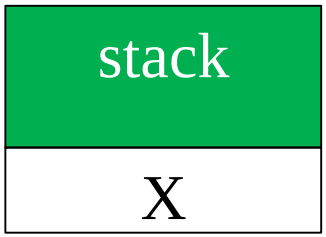
```
void inverti(void) {
   char c;
   c = getchar();
   if ( c == '\n' ) {
     /* caso base la sequenza è finita, stampo
   una intestazione */
      printf("Sequenza invertita: ");
      return ;
}
inverti();
putchar(c);
return;
}
```

# Inversione di una stringa

- Lo eseguiremo in laboratorio per convincerci che funziona...
- Cosa accade stavolta sullo stack ?

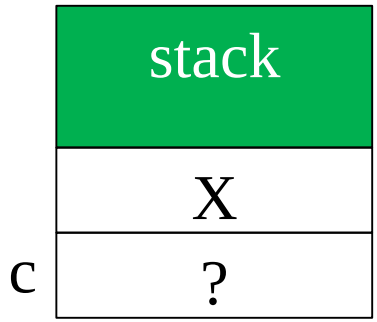
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```



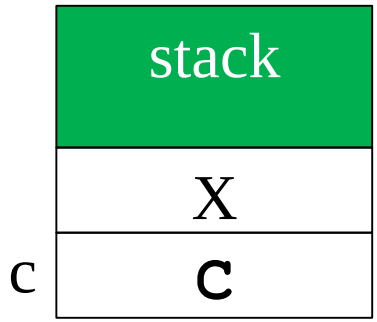
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}


...
inverti();
X: printf("Fine...\n");
...
```



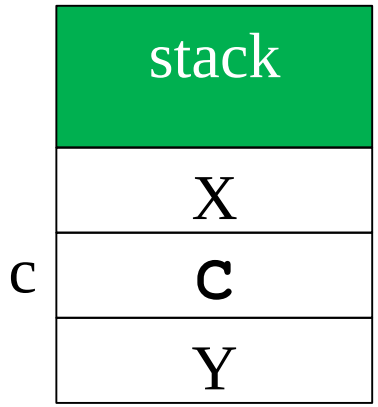
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```



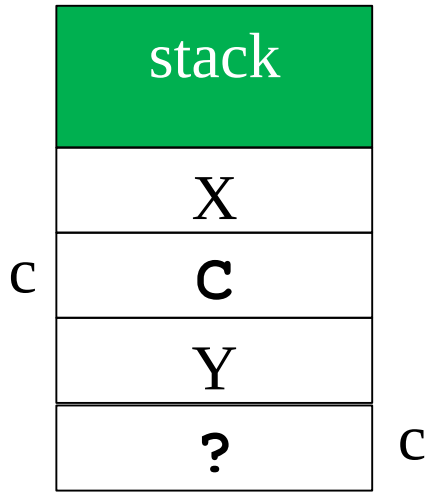
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```



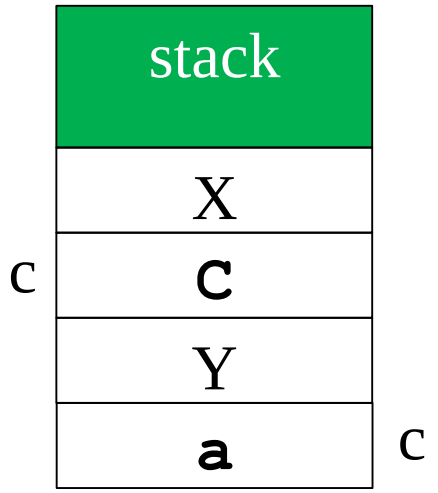
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}


...
inverti();
X: printf("Fine...\n");
...
```



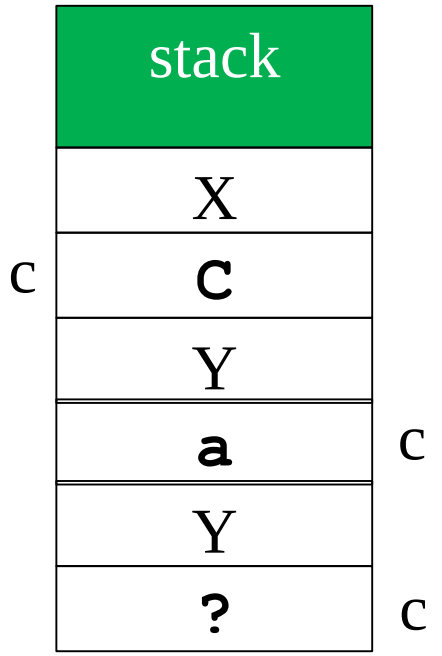
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```



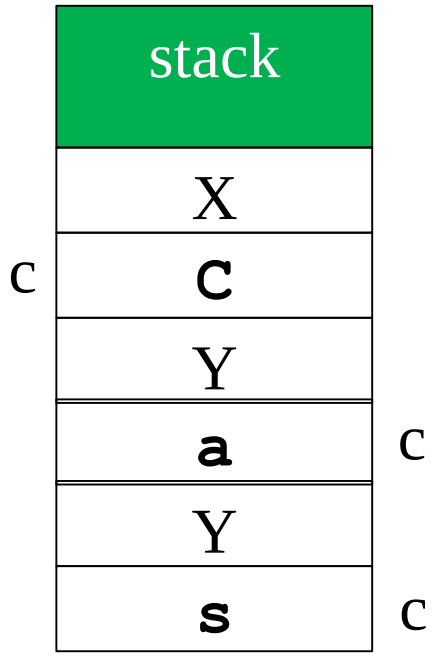
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```



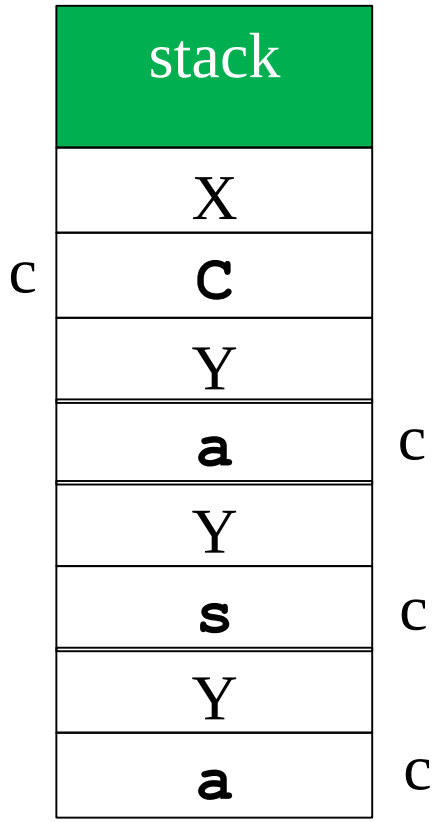
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```



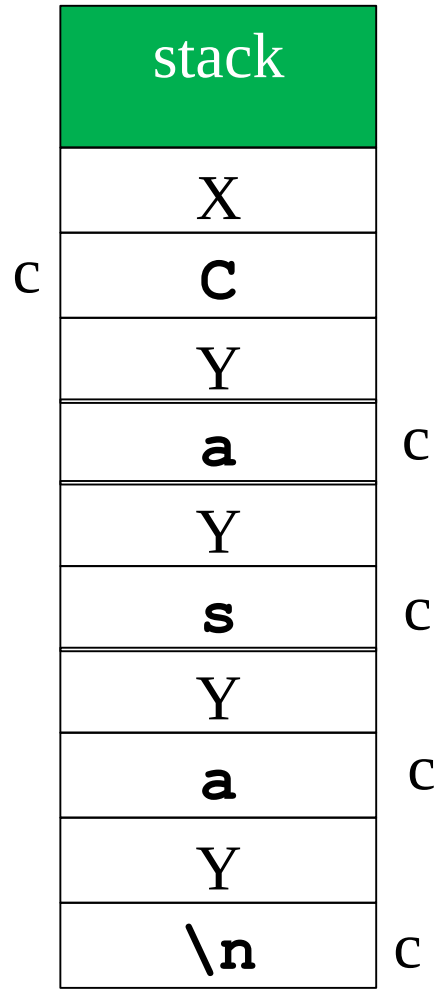
# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```



# Inversione di una stringa: lo stack

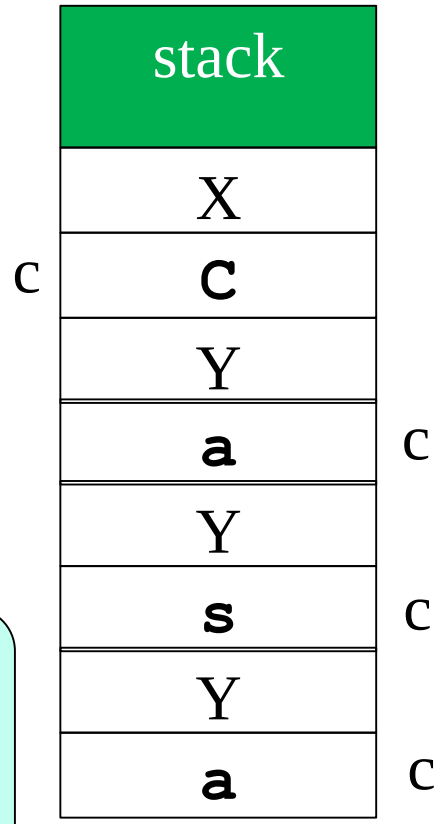
```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}


...
inverti();
X: printf("Fine...\n");
...
```



# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```

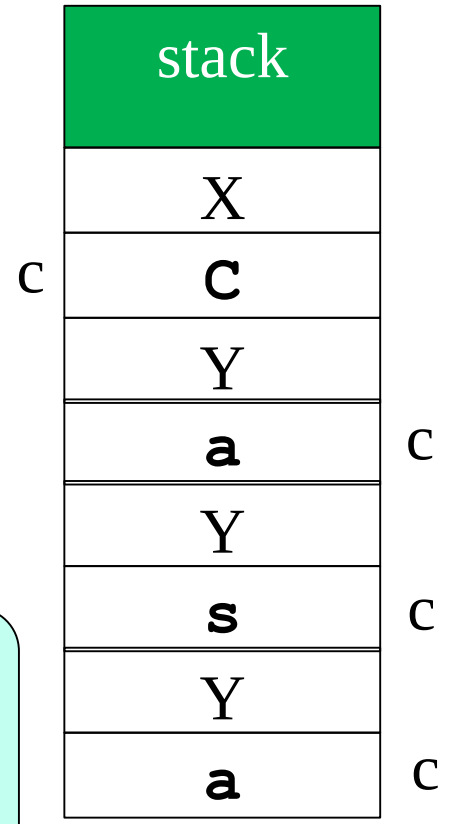
Sequenza invertita:



# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```

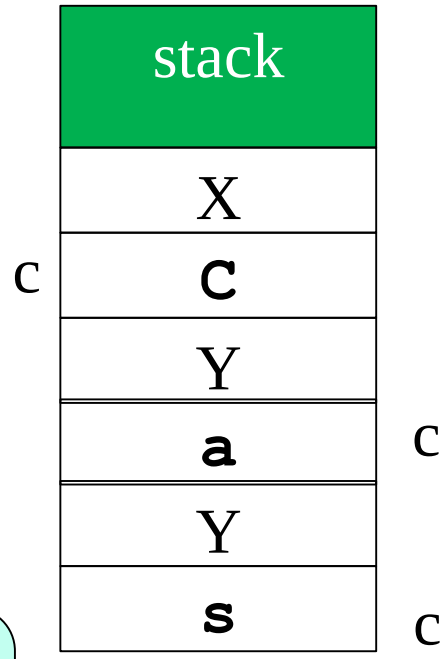
```
Sequenza invertita: a
```



# Inversione di una stringa: lo stack

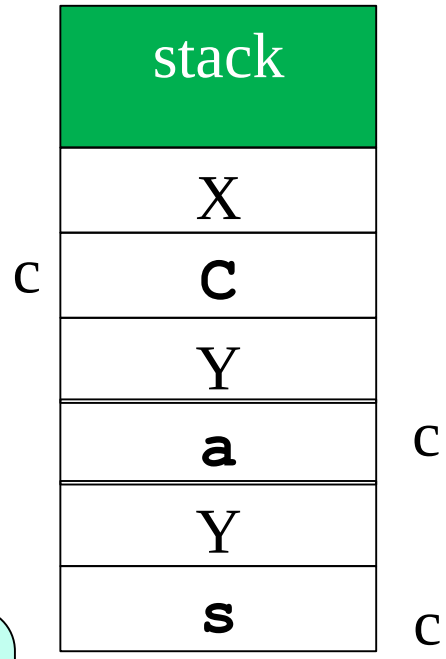
```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```

```
Sequenza invertita: a
```



# Inversione di una stringa: lo stack

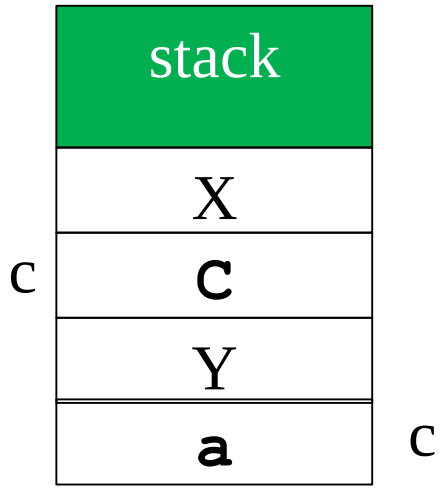
```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```




```
Sequenza invertita: as
```

# Inversione di una stringa: lo stack

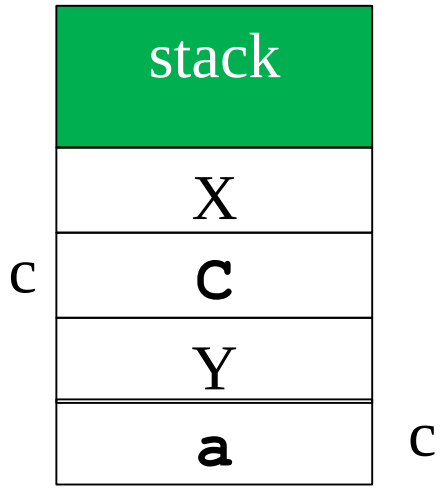
```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```




```
Sequenza invertita: as
```

# Inversione di una stringa: lo stack

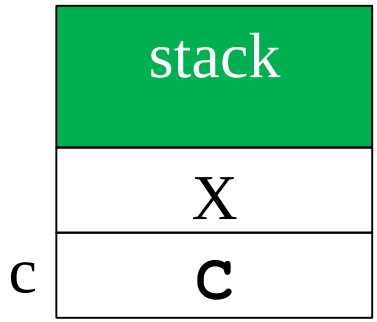
```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```




```
Sequenza invertita: asa
```

# Inversione di una stringa: lo stack

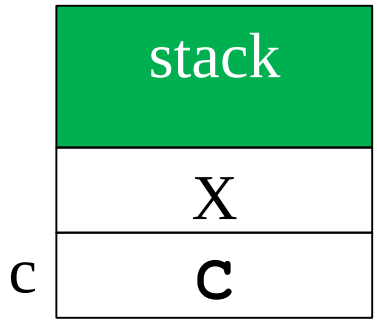
```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```




```
Sequenza invertita: asa
```

# Inversione di una stringa: lo stack

```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```




```
Sequenza invertita: asaC
```

# Inversione di una stringa: lo stack

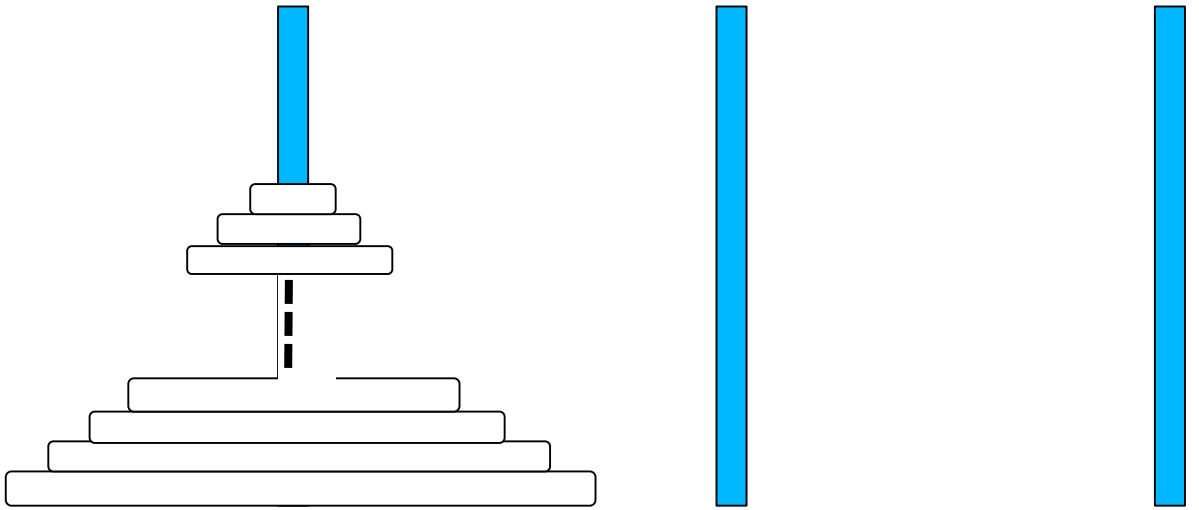
```
void inverti(void) {
  char c;
  c = getchar();
  if ( c == '\n' ) {
   printf("Sequenza invertita: ");
     return ;
}
inverti();
Y: putchar(c);
return;
}

...
inverti();
X: printf("Fine...\n");
...
```

stack

```
Sequenza invertita: asaC
```

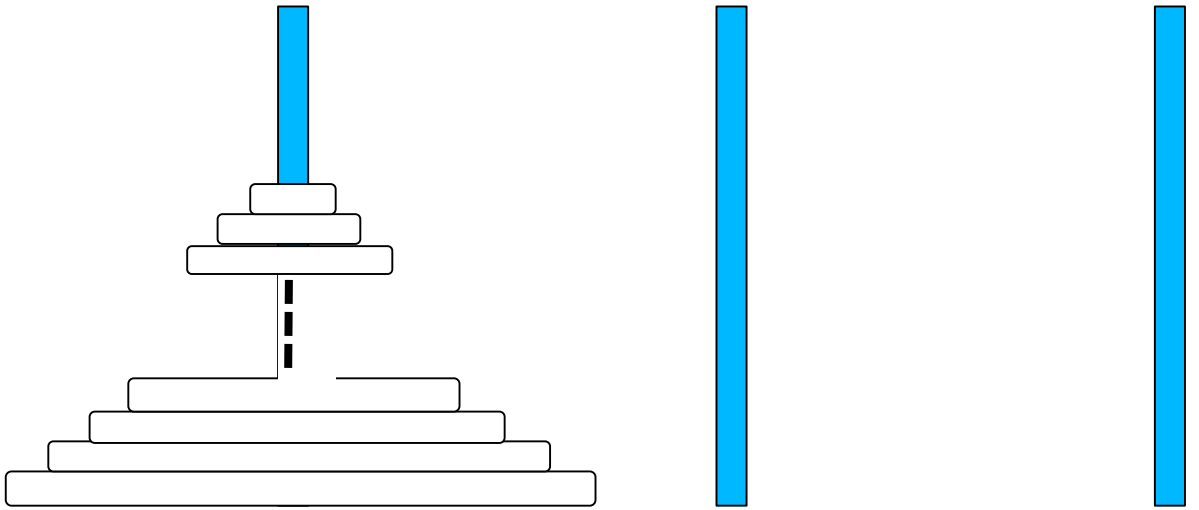
# La torre di Hanoi

- Ecco un esempio più complesso in cui la ricorsione aiuta a trovare una strategia di soluzione:



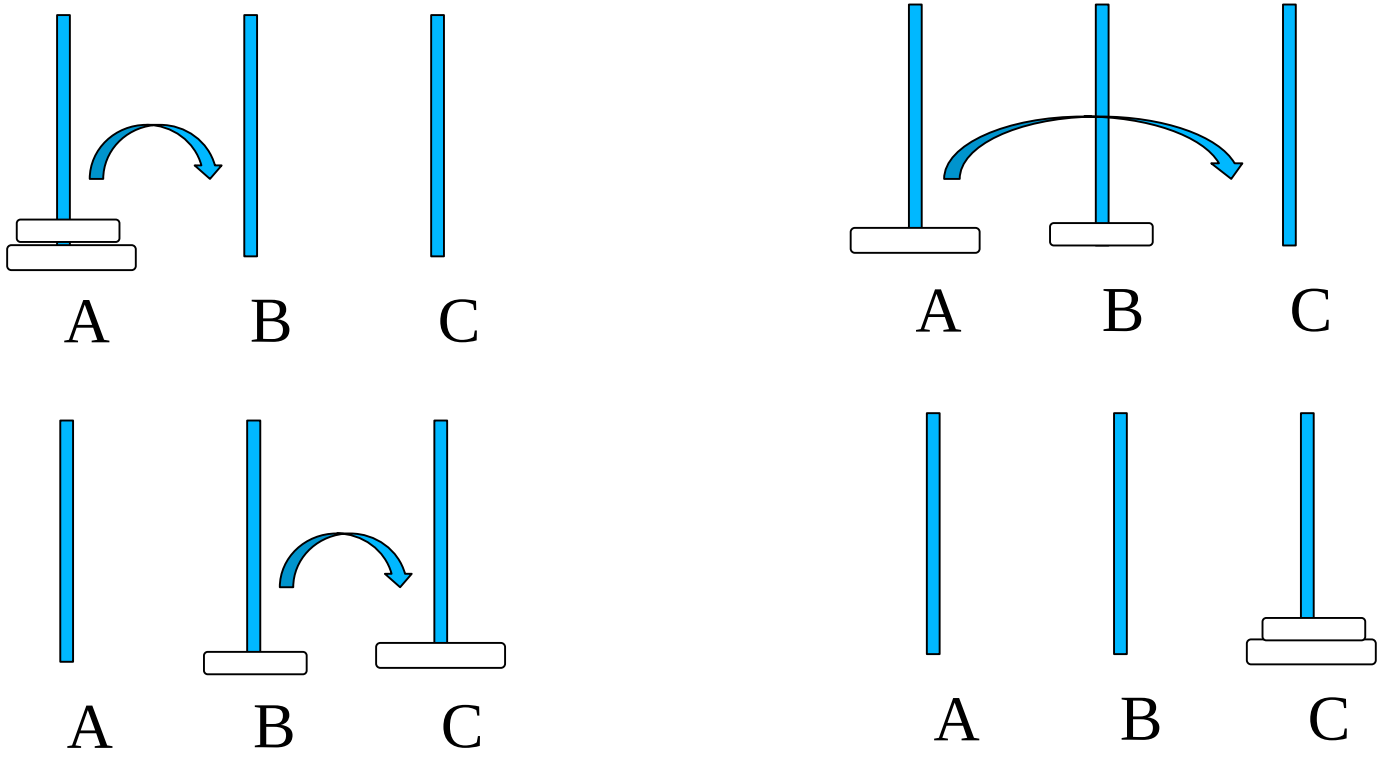
# La torre di Hanoi

- Vincoli:
  - Andare da A a C con B perno di appoggio
  - Spostare un solo disco alla volta
  - Un disco più grande non può mai stare su un disco più piccolo



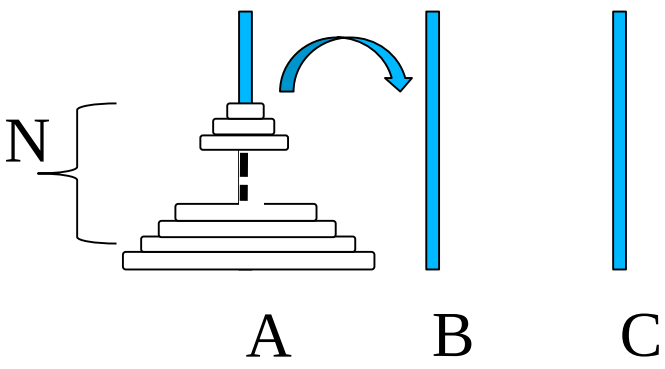
# La torre di Hanoi

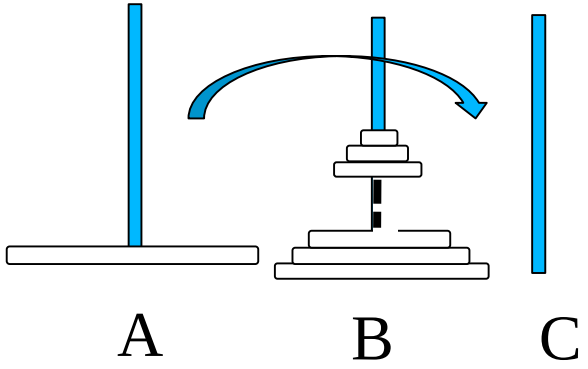
- Come si individua la soluzione per N dischi ?
  - Per 1 disco e' ovvio ....
  - Per 2 abbiamo

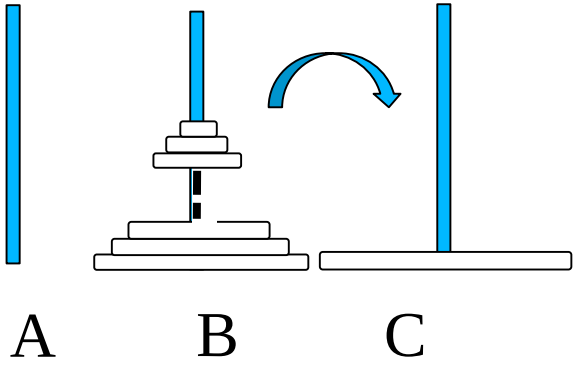


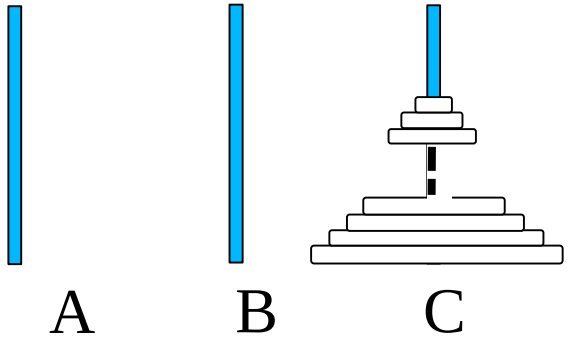
# La torre di Hanoi

- Generalizziamo ?
  - Se sappiamo risolvere per N come si risolve per N+1 ?



# La torre di Hanoi

## Formalizziamo il ragionamento ...

Indichiamo con hanoi(N, P1, P2, P3) la funzione che risolve il problema: "spostare N dischi dal perno P1 al perno P2 utilizzando P3 come perno d'appoggio".

```
hanoi(N, P1, P2, P3) {
   if (N=1)
       sposta da P1 a P2;
  else {
         hanoi(N-1, P1, P3, P2);
         sposta da P1 a P2;
         hanoi(N-1, P3, P2, P1);
}
```

# La torre di Hanoi

## Esempio: hanoi(3, A, C, B) ...

hanoi(3,A,C,B)

hanoi(2,A,B,C)

hanoi(1,A,C,B) sposta (A,B) hanoi(1, C, B, A)

sposta(A,C) sposta(C,B)

sposta (A,C)

hanoi(2, B, C, A)

hanoi(1,B,A,C) sposta (B,C) hanoi(1, A, C, B)

sposta(B,A) sposta(A,C)