Variabili locali

- Il blocco che costituisce il corpo di una funzione/procedura può contenere dichiarazioni di variabili.

  Esempio:

```
void leggiVettore(int v[], int dim)
{
  int i;              /* i E' UNA VARIABILE LOCALE */
  for (i = 0; i < dim; i++) { ... }
}
```

  - sono variabili proprie della funzione
  - hanno tempo di vita limitato alla durata della chiamata
  - più in generale: un identificatore dichiarato nel corpo di una funzione è detto locale alla funzione e non è visibile all'esterno della funzione (ad esempio nel `main`), ma solo nel corpo della stessa

- In realtà, ciò non è altro che un caso particolare di regole generali che governano la visibilità e il tempo di vita degli identificatori di un programma.

Struttura generale di un programma C

- parte direttiva
- parte dichiarativa globale che comprende:
  - dichiarazioni di costanti
  - dichiarazioni di tipi (li vedremo . . .)
  - dichiarazioni di variabili (variabili globali)
  - prototipi di funzioni/procedure
- il programma principale (`main`)
- le definizioni di funzioni/procedure

Esempio

```
#include <stdio.h>        /* parte direttiva */
#define LUNG 10

int i = 1;               /* variabili globali */
int j = 2;

int Q(int);              /* prototipi di funzioni e procedure */
void P(int *);

main()                  /* programma principale */
{
int x = 10;
char c = 'a';
x = Q(x);
P(&x);
}

int Q(int v) { ... }     /* definizioni di funzioni e procedure */
void P(int *z) { ... }
```

Blocchi

- il corpo di una funzione/procedura, cosi' come il corpo del programma principale, è un blocco.
- In C un blocco è costituito da
  - una parte dichiarativa (può non esserci)
  - una parte esecutiva (sequenza di istruzioni)
- Nel `main` o nel corpo delle funzioni possono comparire diversi blocchi, che possono essere
  - annidati: un blocco è una delle istruzioni di un altro blocco
  - paralleli: blocchi che fanno parte della medesima sequenza di istruzioni

```
{
   int x;
   x = 10;
   {
     int z;
     z = 20 ;
     ...
   }
   ...
}
```

```
{
   int x;
   x = 10;
   ...
}
{
   int z;
   z = 20;
   ...
}
```

- Anche la parte esecutiva del programma principale e di una funzione/procedura è un blocco
- Gli identificatori dichiarati nella parte dichiarativa di un blocco sono detti nomi locali del blocco e devono essere tutti diversi tra loro
  - nel caso di una funzione/procedura, fanno parte dei nomi locali anche gli identificatori utilizzati per i parametri formali

Esempio:

```
{
int x;   /* NO! identificatore x dichiarato   */
char x;  /* due volte nello stesso blocco     */
...
}

void p(int x, char y)
{
int x;      /* NO! identificatore x gia' usato per un parametro formale */
...
}
```

- In blocchi diversi possono essere utilizzati gli stessi identificatori

Esempio:

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
   {
     char x;     /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;          /*x,y: variabili locali della procedura p  */
...
}
```

- Un programma C può avere una struttura molto complessa a seguito dell'uso di funzioni, procedure e blocchi.
- È necessario definire regole precise per regolamentare l'uso dei nomi utilizzati all'interno di un programma.
- A questo scopo introduciamo alcune definizioni utili.

  Ambiente globale: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa globale del programma

  Ambiente locale di una funzione: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa della funzione e nella sua intestazione

  Ambiente locale di un blocco: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa del blocco
- Quanto detto informalmente in precedenza può essere meglio precisato:

  $\Longrightarrow$ è possibile dichiarare più volte lo stesso identificatore (anche con significati diversi) purché in ambienti diversi
- Se ciò evita il proliferare di identificatori, causa il problema di stabilire il significato di un riferimento ad un identificatore in un generico punto del programma

Esempio: Riprendiamo l'esempio precedente

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
   {
     char x;     /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;          /*x,y: variabili locali della procedura p  */
...
}
```

- Se in un punto del programma viene eseguita l'istruzione `x = ...`, a quale delle tre dichiarazioni di `x` ci si riferisce?
- Dipende dal punto in cui si trova tale assegnamento e dalle regole di visibilità (o regole di scoping).

Regole di visibilità

- Gli identificatori presenti nell'ambiente globale sono visibili in tutte le funzioni e in tutti i blocchi del programma.
  Se un identificatore è definito in più punti (in blocchi e/o funzioni), la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.
  N.B. Gli identificatori predefiniti del linguaggio si intendono parte dell'ambiente globale.
- Gli identificatori presenti nell'ambiente locale di una funzione sono visibili nel corpo della funzione (ivi compresi eventuali blocchi in esso contenuti).
  Se un identificatore è definito in più punti del corpo, la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.
- Gli identificatori presenti nell'ambiente locale di un blocco sono visibili nella parte esecutiva del blocco (ivi compresi eventuali blocchi in essa contenuti).
  Se un identificatore è definito in più punti di un blocco, la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.

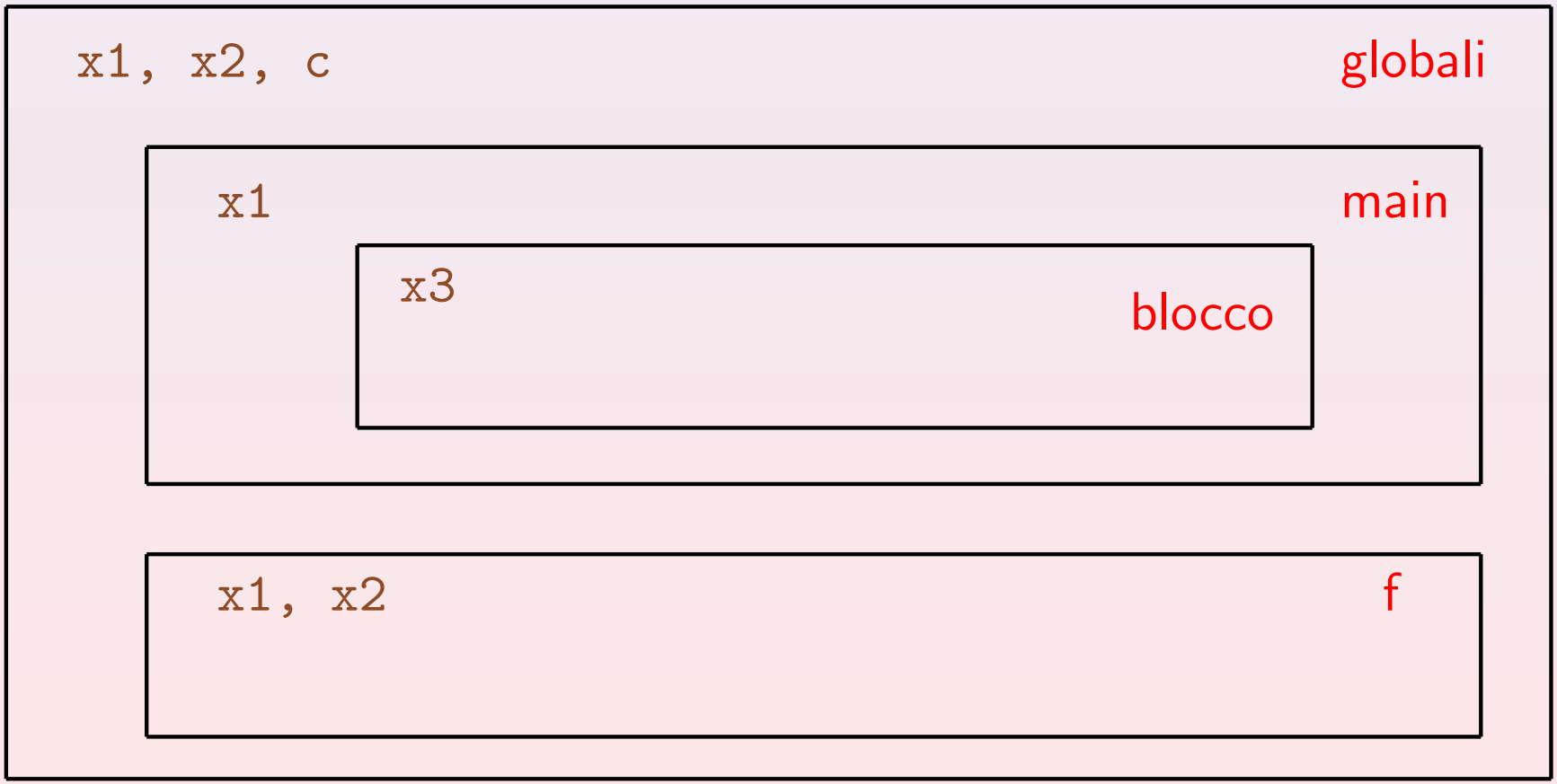
- Detto altrimenti, l'ambito di visibilità di un identificatore è determinato dalla posizione della sua dichiarazione:
  - gli identificatori dichiarati all'interno di un blocco hanno ambito di visibilità a livello di blocco
    $\Longrightarrow$ una variabile dichiarata in un blocco è visibile solo in quel blocco (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'interno di una funzione (compresi quelli nell'intestazione) hanno ambito di visibilità a livello di funzione
    $\Longrightarrow$ una variabile dichiarata in una funzione è visibile solo nel corpo della funzione (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'esterno delle funzioni e del main hanno ambito di visibilità a livello di programma
    $\Longrightarrow$ una variabile globale è visibile ovunque nel programma

Esempio:

```
int x1=10, x2=20;
char c='a';

int f(int);

main()
{
int x1=30;    /* nasconde la variabile globale x1 */
x2 = x1+x2;   /* x1 e' quella locale, x2 e' globale */
printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=30  x2=50  */
  { int x3=50;
    x1=f(x3); /* x1 e' quella locale al primo blocco */
    printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=150  x2=50 */
  }
}

int f(int x1)  /* nasconde la variabile globale x1 */
{ int x2;       /* nasconde la variabile globale x2 */
  x2 = x1 + 100;  /* x1 e' il parametro formale, x2 la var. locale */
  return x2;
}
```

Rappresentazione Grafica: Modello a contorni

- Si rappresenta ogni ambiente mediante un rettangolo con gli identificatori in esso contenuti.



Durata delle variabili

- Una variabile ha un suo tempo di vita.
  - viene creata (ovvero ad essa viene riservata uno spazio di memoria)
  - viene (o può essere) distrutta (ovvero viene rilasciato il corrispondente spazio di memoria).
- Si distinguono due classi di variabili:
  - variabili automatiche: vengono create ogni volta che si entra nel loro ambiente di visibilità e vengono distrutte all'uscita di tale ambiente
    - es. variabili locali di un blocco: vengono create all'ingresso del blocco { distrutte all'uscita dal blocco }
    - es. variabili locali di una funzione: vengono create al momento della chiamata e distrutte all'uscita
  - variabili statiche: vengono create una sola volta e vengono distrutte solo al termine dell'esecuzione del programma (non ne faremo uso ...)
- N.B. nel caso di funzioni/blocchi eseguiti più volte (es. funzione chiamata in punti diversi, blocco all'interno di un ciclo):

  le variabili automatiche corrispondenti possono essere associate di volta in volta a locazioni di memoria diverse, quindi
  il loro valore non persiste tra una esecuzione e la successiva

Gestione della memoria a tempo di esecuzione (run-time)

- Il codice macchina e i dati risiedono entrambi in memoria, ma in zone separate:
  - la memoria per il codice macchina è fissata a tempo di compilazione
  - la memoria per i dati (in particolare per le variabili automatiche) cresce e decresce dinamicamente durante l'esecuzione: viene gestita a pila
- Una pila (o stack) è una struttura dati con accesso LIFO: Last In First Out = l'ultimo entrato è il primo ad uscire (es.: pila di piatti da lavare).
- Il sistema gestisce in memoria la pila dei record di attivazione (RDA)
  - per ogni chiamata di funzione viene creato un nuovo RDA in cima alla pila
  - al termine della chiamata della funzione il RDA viene rimosso dalla pila
- Ogni RDA contiene:
  - le locazioni di memoria per i parametri formali (se presenti)
  - le locazioni di memoria per le variabili locali (se presenti)
  - altre informazioni che non analizziamo
- Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

Esempio:

```
int f(int);
main()
{
    int x, y, z;
    x=10;
    y=20;              /* PUNTO 1: blocco principale */
    z = f(x);          /* PUNTO 2: prima chiamata di f */
       {               /* PUNTO 3: uscita da f e ingresso nel blocco annidato*/
         int x=50;
         y=f(x);       /* PUNTO 4: seconda chiamata di f */
         z=y;          /* PUNTO 5: uscita da f */
       }
    ...                /* PUNTO 6: uscita dal blocco  */
}
int f(int a)
{
   int z;
   z = a + 1;
   return z;
}
```

Evoluzione della pila

PUNTO 1

x	10
y	20
z	?

PUNTO 2

a	10
z	?

x	10
y	20
z	?

PUNTO 3

x	50

x	10
y	20
z	11

Evoluzione della pila - continua

PUNTO 4

a	50
z	?

x	50

x	10
y	20
z	11

PUNTO 5

x	50

x	10
y	51
z	11

PUNTO 6

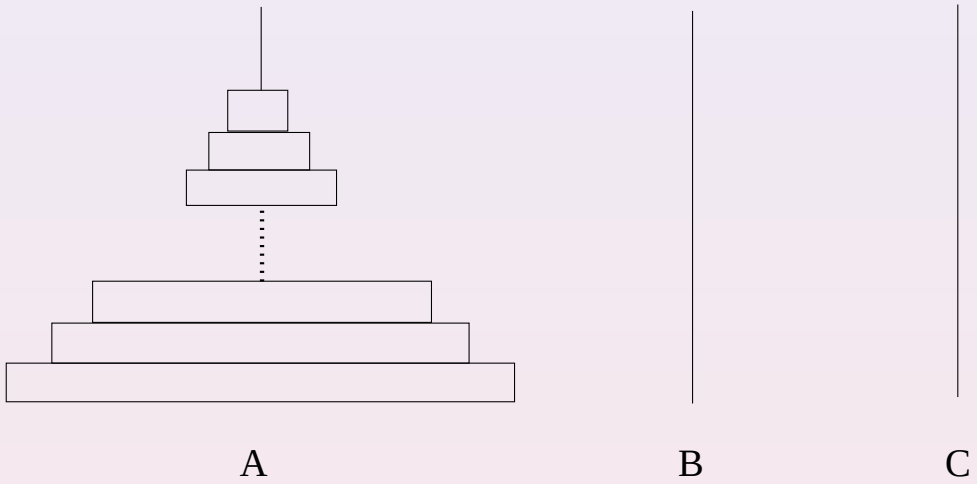
x	10
y	51
z	51

Programmazione ricorsiva: cenni

- In quasi tutti i linguaggi di programmazione evoluti è ammessa la possibilità di definire funzioni/procedure ricorsive: durante l'esecuzione di una funzione F è possibile chiamare la funzione F stessa.
- Ciò può avvenire
  - direttamente: il corpo di F contiene una chiamata a F stessa.
  - indirettamente: F contiene una chiamata a G che a sua volta contiene una chiamata a F.
- Questo può sembrare strano: se pensiamo che una funzione è destinata a risolvere un sottoproblema $\mathcal{P}$, una definizione ricorsiva sembra indicare che per risolvere $\mathcal{P}$ dobbiamo . . . saper risolvere $\mathcal{P}$!

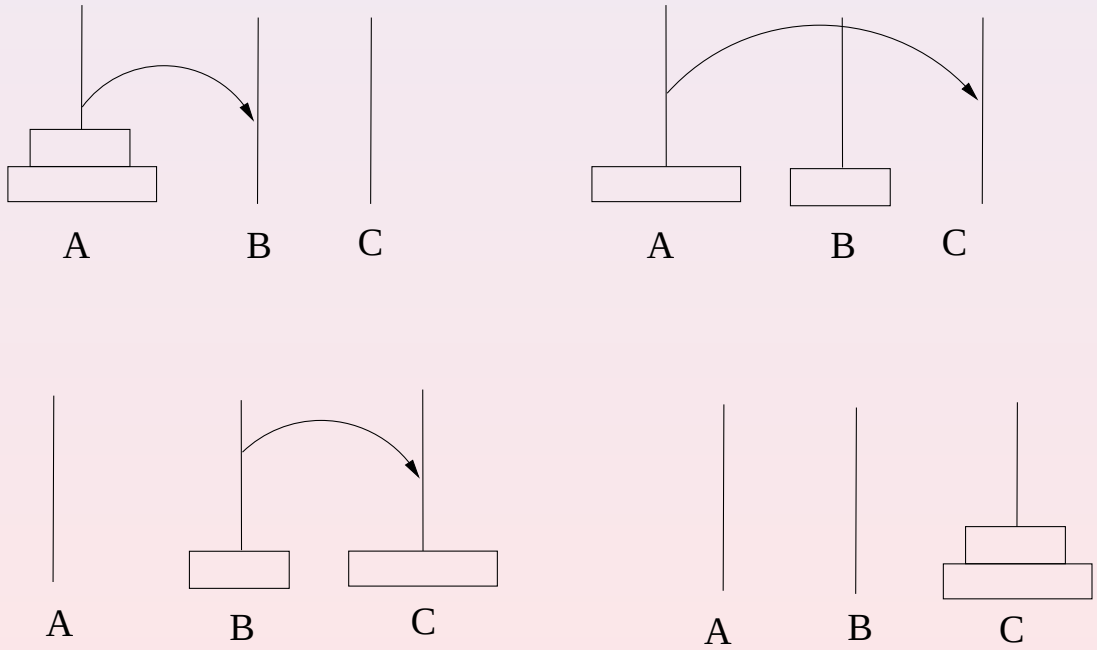
- In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.
- La programmazione ricorsiva trova radici teoriche nel principio di induzione ben fondata che può essere visto come una generalizzazione del principio di induzione sui naturali
- La soluzione di un problema viene individuata supponendo di saperlo risolvere su casi più semplici.
- Bisogna poi essere in grado di risolvere direttamente il problema sui casi più semplici di qualunque altro.

Esempio: Torre di Hanoi (leggenda Vietnamita).



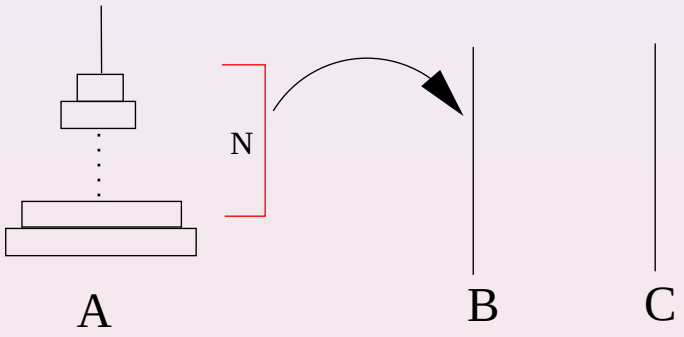
- pila di dischi di dimensione decrescente su un perno A
- vogliamo spostarla sul perno C, usando un perno di appoggio B
- vincoli:
  - possiamo spostare un solo disco alla volta
  - un disco più grande non può mai stare su un disco più piccolo
- secondo la leggenda: i monaci stanno spostando 64 dischi: quando avranno finito, ci sarà la fine del mondo

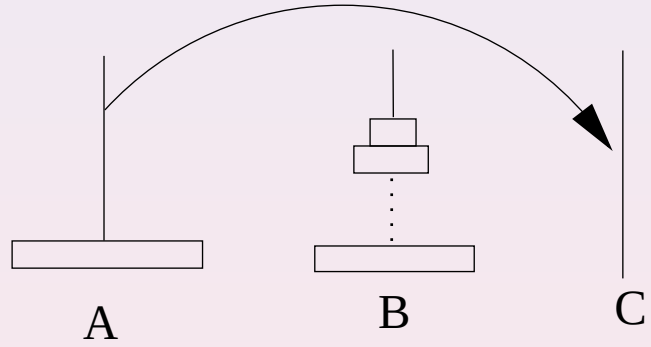
- Come individuare una soluzione per un numero N di dischi arbitrario?
  - per N=1 la soluzione è immediata: spostiamo l'unico disco da A a C
  - se sappiamo risolvere il problema per N=1 lo sappiamo risolvere anche per N=2: come?

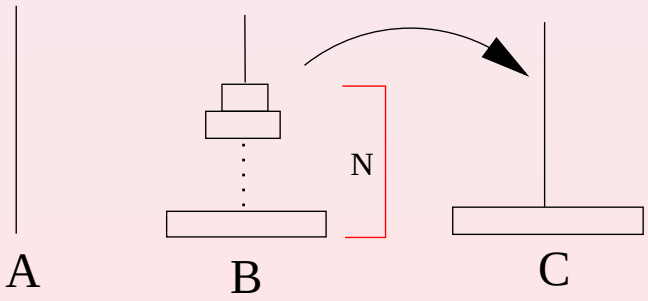


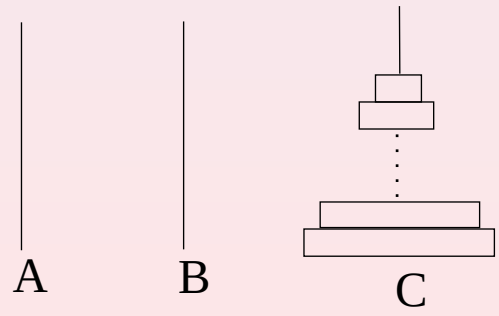
- Notiamo l'utilizzo del perno ausiliario B

- Possiamo generalizzare il ragionamento? Se sappiamo risolvere il problema per N dischi, possiamo individuare una soluzione per lo stesso problema ma con N+1 dischi?









- Formalizziamo il ragionamento
- Indichiamo con hanoi(N, P1, P2, P3) 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);
       }
```

Esempio: Soluzione di hanoi(3,A,C,B)

hanoi(3,A,C,B) =
- hanoi(2,A,B,C) =
  - hanoi(1,A,C,B) = sposta(A,C)
  - sposta(A,B)
  - hanoi(1,C,B,A) = sposta(C,B)
- sposta(A, C)
- hanoi(2,B,C,A) =
  - hanoi(1,B,A,C) = sposta(B,A)
  - sposta(B,C)
  - hanoi(1,A,C,B) = sposta(A,C)

- Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo induttivo.
  Esempio: Definizione induttiva di somma tra due interi non negativi:

$$somma(x, y) = \begin{cases} x & \text{se } y{=}0 \\ 1 + (somma(x, y-1)) & \text{se } y > 0 \end{cases}$$

  - La somma di x con 0 viene definita in modo immediato;
  - la somma di x con il successore di y viene definita come il successore della somma tra x e y.
- **Esempio:** somma di 3 e 2:

$$\begin{aligned} somma(3,2) = \quad & 1 + (\underline{somma(3,1)}) = \\ & 1 + (1 + (\underline{somma(3,0)})) = \\ & 1 + (1 + (3)) = \\ & 1 + 4 = \\ & 5 \end{aligned}$$

Esempio: Funzione fattoriale.

- definizione iterativa: $fatt(n) = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- definizione induttiva:

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 \quad \text{(caso base)} \\ n \cdot fatt(n-1) & \text{se } n > 0 \quad \text{(caso induttivo)} \end{cases}$$

- È essenziale il fatto che, applicando ripetutamente il caso induttivo, ci riconduciamo prima o poi al caso base.

$$\begin{aligned} fatt(3) = \quad & 3 \cdot \underline{fatt(2)} = \\ & 3 \cdot (2 \cdot \underline{fatt(1)}) = \\ & 3 \cdot (2 \cdot (1 \cdot \underline{fatt(0)})) = \\ & 3 \cdot (2 \cdot (1 \cdot 1)) = \\ & 3 \cdot (2 \cdot 1) = \\ & 3 \cdot 2 = \\ & 6 \end{aligned}$$

Il codice delle due diverse versioni

- definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
      ris=ris*i;
    return ris;
}
```

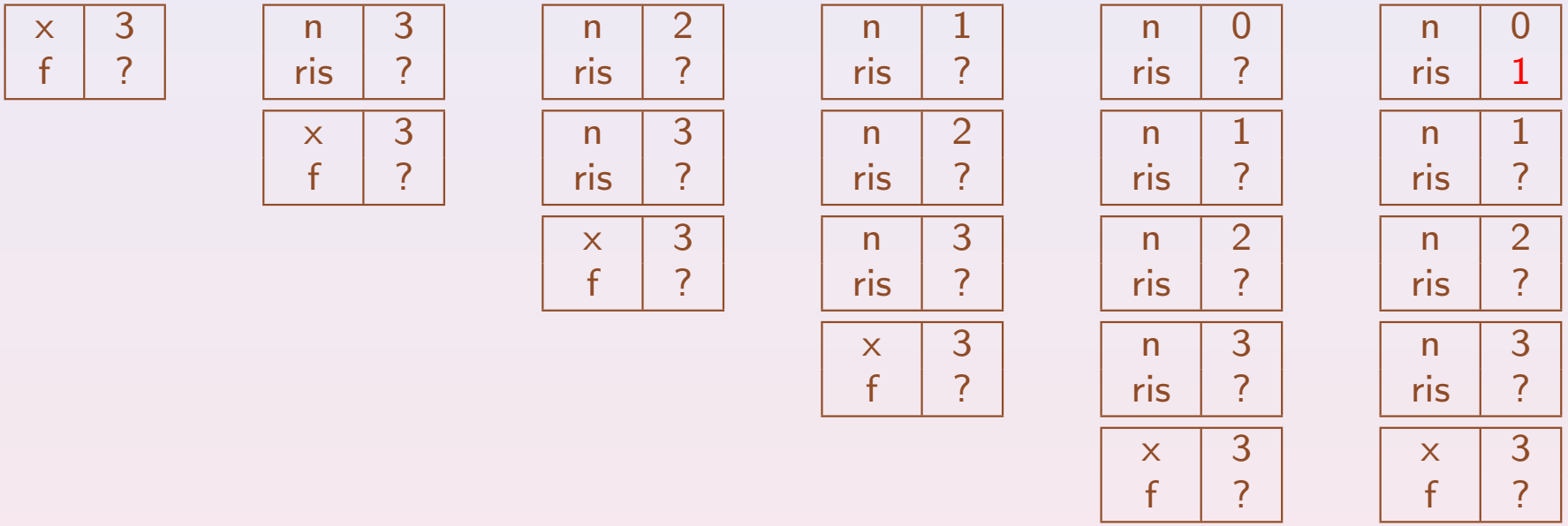
- definizione ricorsiva:

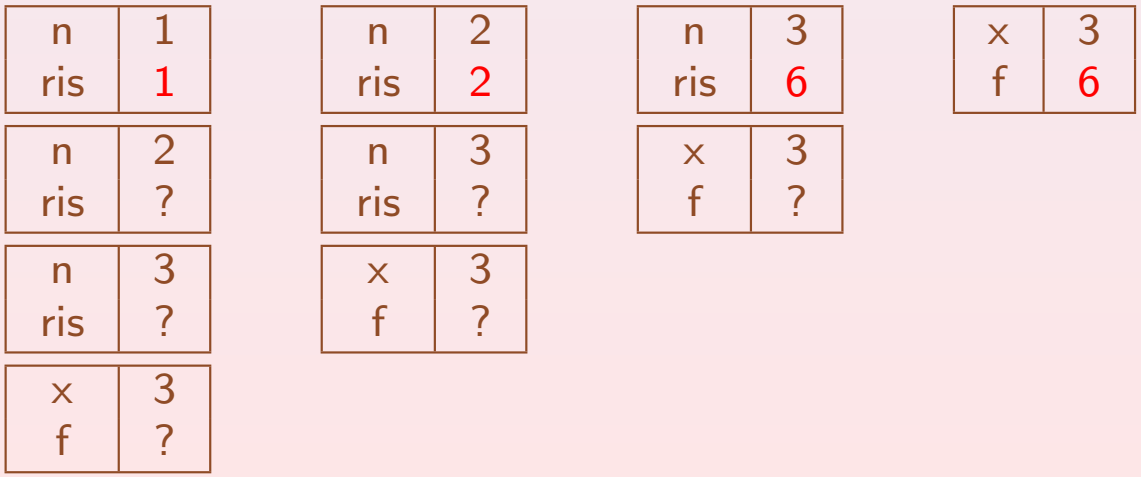
```
int fattric(int n) {
  if (n == 0)
    return 1;
  else
    return n * fattric(n-1);
 }
```

Esempio: Programma che usa una funzione ricorsiva.

```
#include <stdio.h>

int fattric (int);

main()
{
int x, f;
scanf("%d", &x);
f = fattric(x);
printf("Fattoriale di %d:  %d\n", x, f);
}

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

Evoluzione della pila (supponendo x=3).





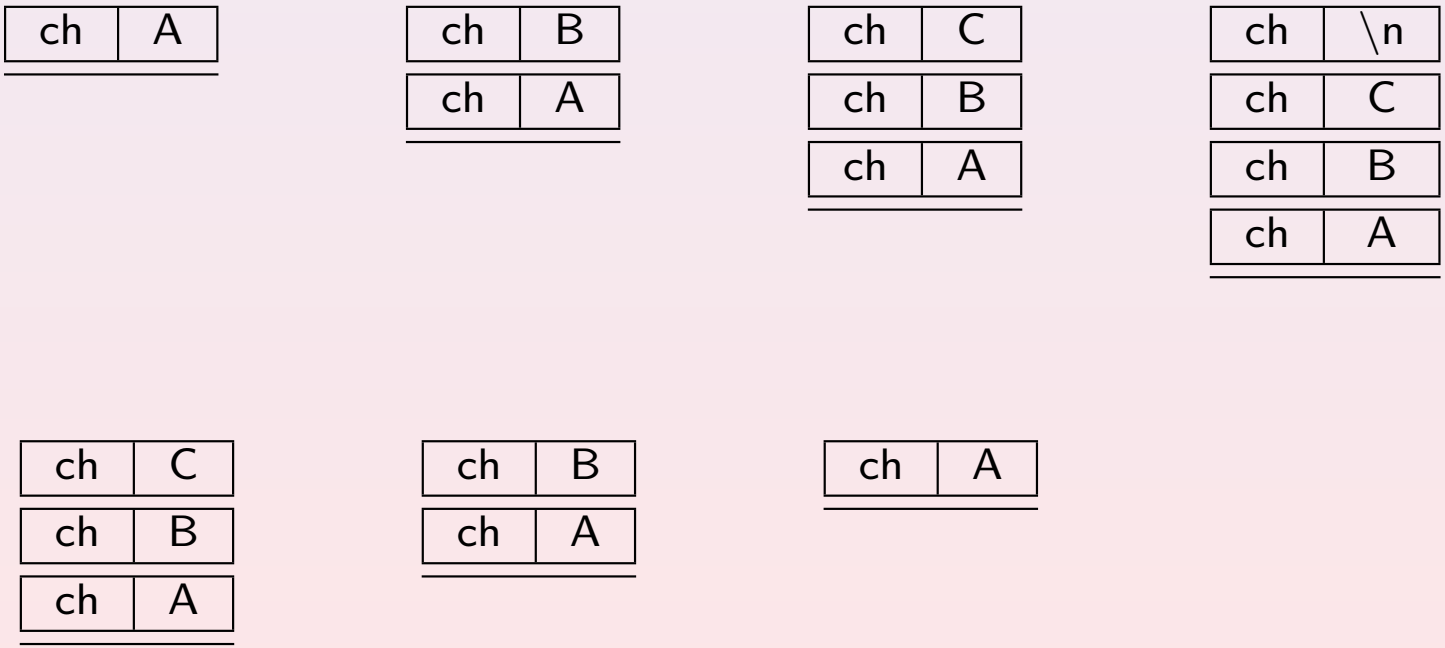
Esempio: Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `casa` $\Longrightarrow$ `asac`

- Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma dinamica (liste, le vedremo più avanti)
  2. usando un procedimento ricorsivo.
     - leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;
     - il caso base è rappresentato dalla lettura del carattere di fine sequenza.

```
void invertInputRic()
 {  char ch;

  ch = getchar();
  if (ch != '\n')
    {
    invertInputRic();
    putchar(ch);
    }
  else
    printf("Sequenza invertita: ");
}
```

```
 main()
{
  printf("Immetti una sequenza di caratteri\n");
  invertInputRic();
  printf("\n");
}
```

Vediamo come evolve la pila per l'input `ABC\n`



L'output prodotto è il seguente

```
Sequenza invertita:  CBA
```

Ricorsione multipla

- Si ha ricorsione multipla quando un'attivazione di una funzione può causare più di una attivazione ricorsiva della stessa funzione (es. torre di Hanoi)

  Esempio: Definizione induttiva dei numeri di Fibonacci.

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \qquad \text{se } n > 1 \end{aligned}$$

- $F(0)$, $F(1)$, $F(2)$, ... è detta sequenza dei numeri di Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
 #include <stdio.h>

int fibonacci (int);

main() {
  int n;

  printf("Inserire un intero >= 0: ");
  scanf("%d", &n);
  printf("Numero %d di Fibonacci: %d\n", n, fibonacci(n));
}

 int fibonacci(int i)
  {
  int ris;
  if (i == 0)
    ris = 0;
  else if (i == 1)
    ris = ;
  else
    ris =  fibonacci(i-1) + fibonacci(i-2);
  return ris;
  }
```

Esempi di funzioni ricorsive

- Tradurre in C la definizione induttiva già vista:

$$somma(x, y) = \begin{cases} x & \text{se } y = 0 \\ 1 + (somma(x, y-1)) & \text{se } y > 0 \end{cases}$$

```
int somma (int x, int y)
{
  int ris;
  if (y==0)
     ris = x;
  else
     ris = 1 + somma(x, y-1);
  return ris;
}
```

- Calcolo ricorsivo di x^y (si assume $y \geq 0$)

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ x \cdot x^{y-1} & \textit{altrimenti} \end{cases}$$

```
int exp (int x, int y)
{
  int ris;
  if (y==0)
     ris = 1;
  else
     ris = x * exp(x, y-1);
  return ris;
}
```

- Calcolare ricorsivamente la somma degli elementi nella porzione di un array v compresa tra gli indici from e to.
- Esprimiamo formalmente quanto richiesto:

$$sumVet(v, from, to) = \sum_{i=from}^{to} v[i]$$

- È evidente che:

$$\sum_{i=from}^{to} v[i] = \begin{cases} 0 & \text{se } from > to \\ v[from] + \sum_{i=from+1}^{to} v[i] & \text{se } from \leq to \end{cases}$$

- La traduzione in C è immediata.

Una soluzione

```
int sumVet(int *v, int from, int to)
{
  if (from > to)
    return 0;
  else
    return v[from] + sumvet(v,from+1,to);
}
```

Un'altra soluzione

```
int sumVet(int *v, int from, int to)
{
  int somma;
  if (from > to)
    somma = 0;
  else
    somma = v[from] + sumvet(v,from+1,to);
  return somma;
}
```

- Calcolare ricorsivamente il numero di occorrenze dell'elemento x nella porzione di un array v compresa tra gli indici from e to.

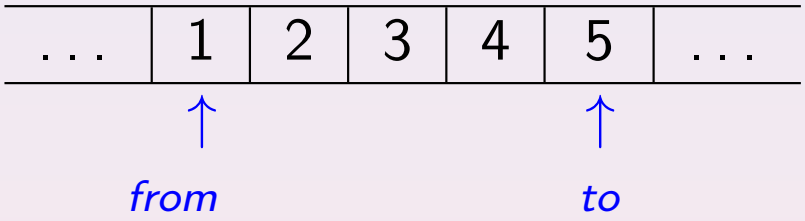
$$f(v, x, from, to) = \#\{i \in [from, to] \mid v[i] = x\}$$

- Anche in questo caso ragioniamo induttivamente:

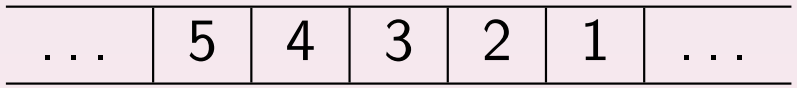
$$f(v, x, from, to) = \begin{cases} 0 & \text{se } from > to \\ f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] \neq x \\ 1 + f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] = x \end{cases}$$

```
int occorrenze (int *v, int x, int from, int to)
{
  int occ;

  if (from > to)
    occ= 0;
  else
    if (v[from]!=x)
        occ = occorrenze(v,x,from+1,to);
    else
        occ = 1+occorrenze(v,x,from+1,to);
}
```

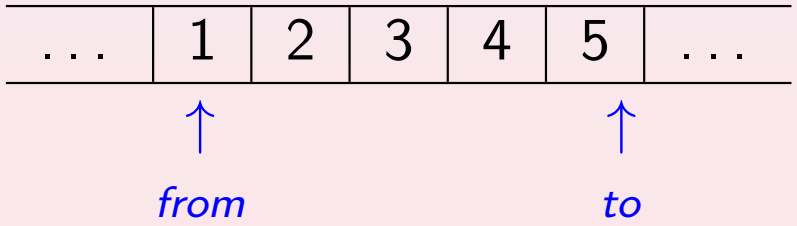
- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.



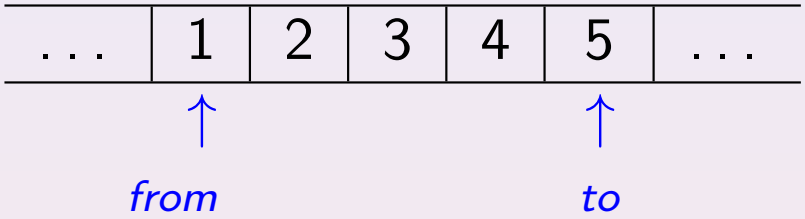
- Vogliamo ottenere:



- Induttivamente:

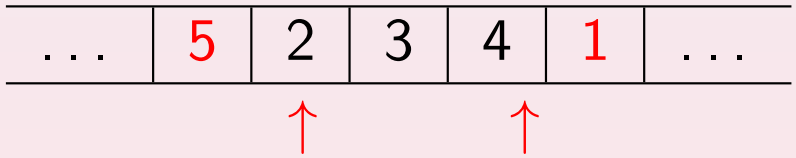


- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.



- Vogliamo ottenere:

- Induttivamente:



- Questa situazione corrisponde alla chiamata ricorsiva su una porzione più piccola del vettore

Programma

```
void swap(int *v, int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

void invertiric (int *v, int from, int to)
{
  if (from < to)
    {
        swap(v, from, to);
        invertiric(v, from+1, to-1);
    }
}
```

- Si noti che la procedura non fa niente se la porzione individuata dal secondo e terzo parametro è vuota (from>to) o contiene un solo elemento (from=to)