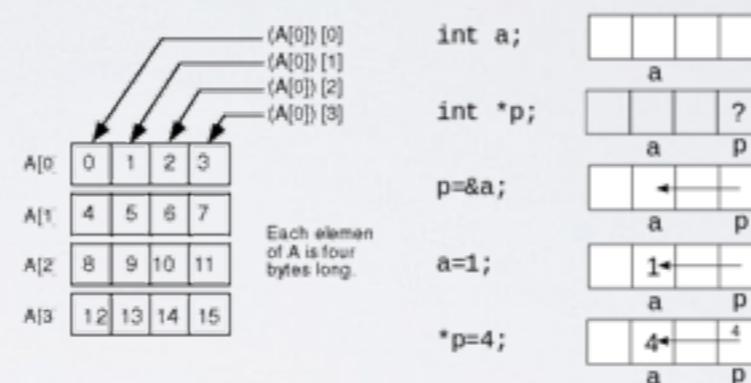


# Olimpiadi Italiane di Informatica



## OLIMPIADI ITALIANE DI INFORMATICA

funzioni, procedure, puntatori, array, funzioni ricorsive

# INDICE DELLA LEZIONE

- 1. Array
- 2. Puntatori
- 3. Puntatori ad array
- 4. Funzioni e procedure
- 5. Funzioni ricorsive

# ARRAY

# ARRAY

- Un array (**o vettore**) è:
  - un gruppo di posizioni (o locazioni di memoria) consecutive
  - hanno tutti lo stesso nome e lo stesso tipo di dato

C[0]	C[1]	C[2]	C[3]
-2	32	11	24

Tutti gli elementi hanno lo stesso nome: C

# ARRAY

- Per far riferimento ad un elemento bisogno specificare:
  - Il nome del vettore (es. C)
  - il numero di posizione (o indice) (es. [2])
- **nome\_vettore [numero\_posizione]**
- Il primo elemento è nella posizione 0
- Per un vettore di n elementi avremo:
  - $c[0], c[1], \dots, c[n-1]$

C[0]	C[1]	C[2]	C[3]
-2	32	11	24

# ARRAY MONODIMENSIONALI

# ARRAY MONODIMENSIONALI

- Sintassi per la dichiarazione:
  - **tipo nome\_array [dimensione] ;**
  - **tipo** : dichiara il tipo di dati degli elementi che formano l'array
  - **dimensione** : definisce quanti elementi conterrà l'array
- Esempio:
  - **int vet [10] ;**

Dichiara un array di 10  
elementi di tipo intero

vet[0]	vet[1]	vet[2]	vet[3]	vet[4]	vet[5]	vet[6]	vet[7]	vet[8]	vet[9]

# ARRAY MONODIMENSIONALI

- Esempio:
  - **vet [0] = 2 ;**

Dichiara un array di 10  
elementi di tipo intero

vet[0]	vet[1]	vet[2]	vet[3]	vet[4]	vet[5]	vet[6]	vet[7]	vet[8]	vet[9]
2									

# ARRAY MONODIMENSIONALI

- Esempio:

- **vet [0] = 2 ;**
- **vet [4] = 32 ;**

vet[0]	vet[1]	vet[2]	vet[3]	vet[4]	vet[5]	vet[6]	vet[7]	vet[8]	vet[9]
2				32					

# ARRAY MONODIMENSIONALI

- Esempio:
  - **vet [0] = 2 ;**
  - **vet [4] = 32 ;**
  - **number = vet[4];**



# INIZIALIZZAZIONE ARRAY

- Ci sono due alternative per inserire i valori negli elementi di un array:
  - **I MODO)** In fase di dichiarazione:
  - **tipo nome\_array [] = { el,e2,...,en } ;**
- Esempio:
  - **int a [] = { 4, 64, 12 } ;**
  - ~~**int a [ 2 ] = { 4, 51, 23 } ;**~~
  - **int a [ 5 ] = { 4, 51, 12 } ;**

Elementi in IV e V  
posizione saranno posti a 0

# INIZIALIZZAZIONE ARRAY

- Ci sono due alternative per inserire i valori negli elementi di un array:
  - **IL MODO**) tramite indicizzazione diretta all'interno del programma:
- Esempio:
  - **int a [3] ;**
  - **a [0] = 4;**
  - **a [1] = 64;**
  - **a [2] = 12;**

# COPIA DI ARRAY

- In C NON è possibile assegnare un array ad un altro attraverso l'operatore di assegnamento (=), almeno non direttamente

```
int a[10], b[10];  
a = b /* ERRORE!!!*/
```

- Per trasferire il contenuto di un array in un altro è necessario assegnare individualmente ogni valore!

# ESEMPIO USO ARRAY

```
1  /*
2   Histogram printing program */
3 #include <stdio.h>
4 #define SIZE 10
5
6 int main()
7 {
8     int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
9     int i, j;
10
11    printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
12
13    for ( i = 0; i <= SIZE - 1; i++ ) {
14        printf( "%7d%13d      ", i, n[ i ] );
15
16        for ( j = 1; j <= n[ i ]; j++ ) /* print one bar */
17            printf( "%c", '*' );
18
19        printf( "\n" );
20    }
21
22    return 0;
23 }
```

# ESEMPIO USO ARRAY

```
1 /*  
2  Histogram printing program */  
3 #include <stdio.h>  
4 #define SIZE 10  
5  
6 int main()  
7 {  
8     int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };  
9     int i, j;  
10  
11    printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );  
12  
13    for ( i = 0; i <= SIZE - 1; i++ ) {  
14        printf( "%7d%13d      ", i, n[ i ] );  
15  
16        for ( j = 1; j <= n[ i ]; j++ ) /* print one bar */  
17            printf( "%c", '*' );  
18  
19        printf( "\n" );  
20    }  
21  
22    return 0;  
23 }
```

Element	Value	Histogram
0	19	*****
1	3	***
2	15	*****
3	7	*****
4	11	*****
5	9	*****
6	13	*****
7	5	***
8	17	*****
9	1	*

Esecuzione

# ARRAY DI CARATTERI

- I vettori di caratteri vengono detti “STRINGHE”

**char str [ ] = “first”;**

- Le stringhe terminano con il carattere nullo ‘\0’
- L’array str ha in realtà 6 elementi, equivale a dichiararlo nel seguente modo:

**char str [ ] = { ‘f’, ‘i’, ‘r’, ‘s’, ‘t’, ‘\0’};**

# ARRAY DI CARATTERI

- Il nome del vettore è l'indirizzo del primo elemento del vettore stesso, pertanto non è necessario l'operatore & nella scanf()

```
char str [ ] = “first”;
```

```
scanf (“%s”, str);
```



Legge i caratteri finché non incontra uno spazio bianco (legge singole parole) e può scrivere anche oltre la fine del vettore

anche oltre la fine del vettore

# ESEMPIO USO ARRAY DI CHAR

```
1  /*
2   Treating character arrays as strings */
3 #include <stdio.h>
4
5 int main()
6 {
7     char string1[ 20 ], string2[] = "string literal";
8     int i;
9
10    printf(" Enter a string: ");
11    scanf( "%s", string1 );
12    printf( "string1 is: %s\nstring2 is: %s\n"
13           "string1 with spaces between characters is:\n",
14           string1, string2 );
15
16    for ( i = 0; string1[ i ] != '\0'; i++ )
17        printf( "%c ", string1[ i ] );
18
19    printf( "\n" );
20    return 0;
21 }
```

# ESEMPIO USO ARRAY DI CHAR

```
1 /*  
2  Treating character arrays as strings */  
3 #include <stdio.h>  
4  
5 int main()  
6 {  
7     char string1[ 20 ], string2[] = "string literal";  
8     int i;  
9  
10    printf(" Enter a string: ");  
11    scanf( "%s", string1 );  
12    printf( "string1 is: %s\nstring2 is: %s\n"  
13          "string1 with spaces between characters is:\n",  
14          string1, string2 );  
15  
16    for ( i = 0; string1[ i ] != '\0'; i++ )  
17        printf( "%c ", string1[ i ] );  
18  
19    printf( "\n" );  
20    return 0;  
21 }
```

```
Enter a string: Hello there  
string1 is: Hello  
string2 is: string literal  
string1 with spaces between characters is:  
H e l l o
```



Esecuzione

# ARRAY DI DIMENSIONI VARIABILI

- In C (ANSI C89) NON è possibile definire array di dimensione variabile, il loro utilizzo, anche se accettato da alcuni compilatori, è illegale!
- La dimensione deve essere nota in fase di compilazione!

Programma non conforme  
allo standard ANSI C89

```
#include <stdio.h>

int main()
{
    int i;
    scanf("%d",&i);
    if(i>10)
    {
        float a[i];
    }
    return 0;
}
```

Programma Corretto

```
#include <stdio.h>

int main()
{
    int i;
    scanf("%d",&i);
    if(i>10)
    {
        float a[100];
    }
    return 0;
}
```

# ARRAY DI DIMENSIONI VARIABILI

- La dimensione dell'array può essere specificata tramite una costante letterale:

```
int array[12];
```

- Una costante simbolica creata con #define:

```
#define MONTHS 12
```

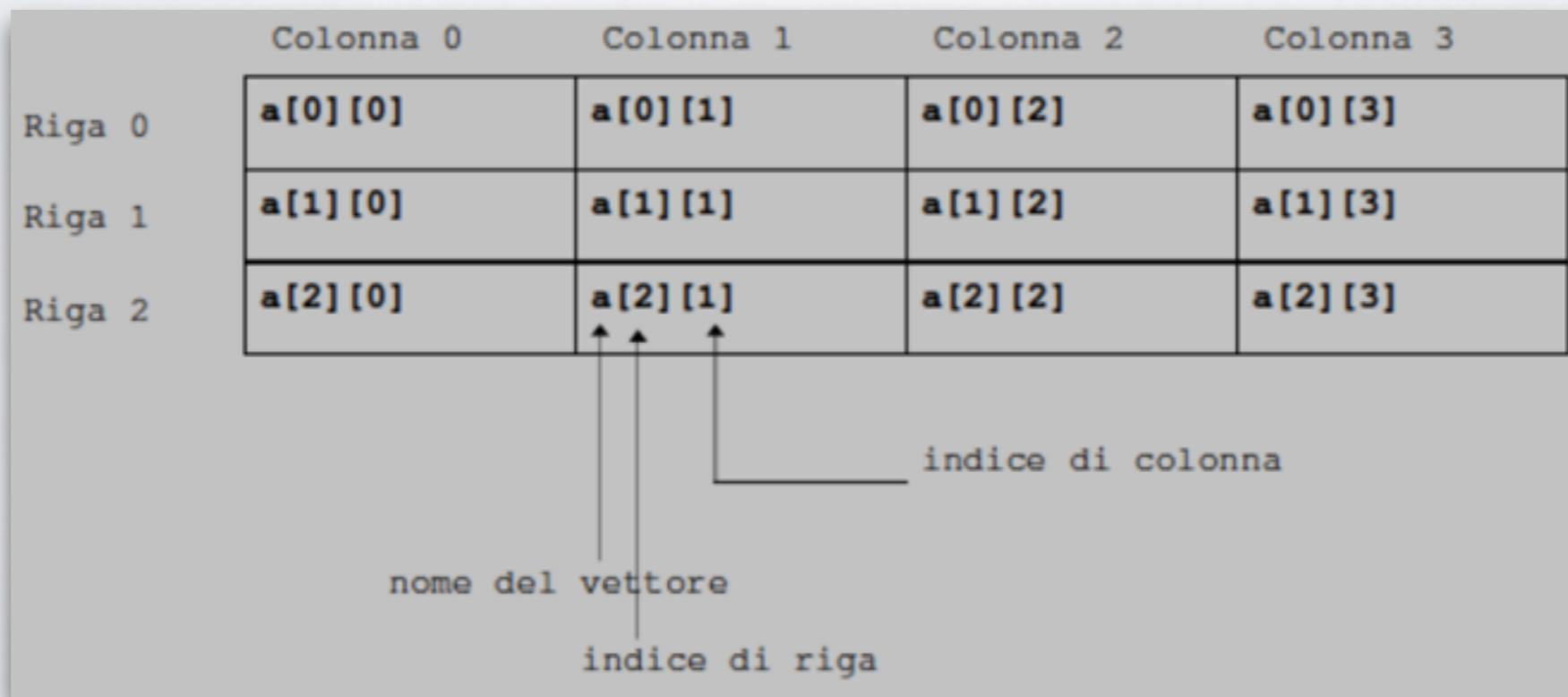
```
int array[MONTHS];
```

```
    } }
```

# ARRAY MULTIDIMENSIONALI

# ARRAY MULTIDIMENSIONALI

- Sono vettori con più indici, appunto più dimensioni
- Nel caso di vettori bidimensionali abbiamo:
  - tavole (matrici) con righe e colonne ( $m \times n$ )
    - il primo indice identifica la riga, il secondo la colonna



# ARRAY MULTIDIMENSIONALI

- Gli array a più dimensioni vengono inizializzati in modo simile a quelli monodimensionali:
- **int sqrs [4][2] = { 1, 1, 2, 4, 3, 9, 4, 16 } ;**

indici	0	1
0	1	1
1	2	4
2	3	9
3	4	16

# ARRAY MULTIDIMENSIONALI

- Gli array a più dimensioni vengono inizializzati in modo simile a quelli monodimensionali:

```
int sqrs [4][2] = {  
    1, 1,  
    2, 4,  
    3, 9,  
    4, 16  
};
```

indici	0	1
0	1	1
1	2	4
2	3	9
3	4	16

# ARRAY MULTIDIMENSIONALI

- Gli array a più dimensioni vengono inizializzati in modo simile a quelli monodimensionali:

```
int b [2][2] = {1,2,3,4} ;
```

indici	0	1
0	1	2
1	3	4

# ARRAY MULTIDIMENSIONALI

- Gli array a più dimensioni vengono inizializzati in modo simile a quelli monodimensionali:

```
int b [2][2] = { {1,2} , {3,4} } ;
```

indici	0	1
0	1	2
1	3	4

# ARRAY MULTIDIMENSIONALI

- Gli array a più dimensioni vengono inizializzati in modo simile a quelli monodimensionali:

```
int b [2][2] = { {1}, {3,4} } ;
```

# ARRAY MULTIDIMENSIONALI

- Gli array a più dimensioni vengono inizializzati in modo simile a quelli monodimensionali:

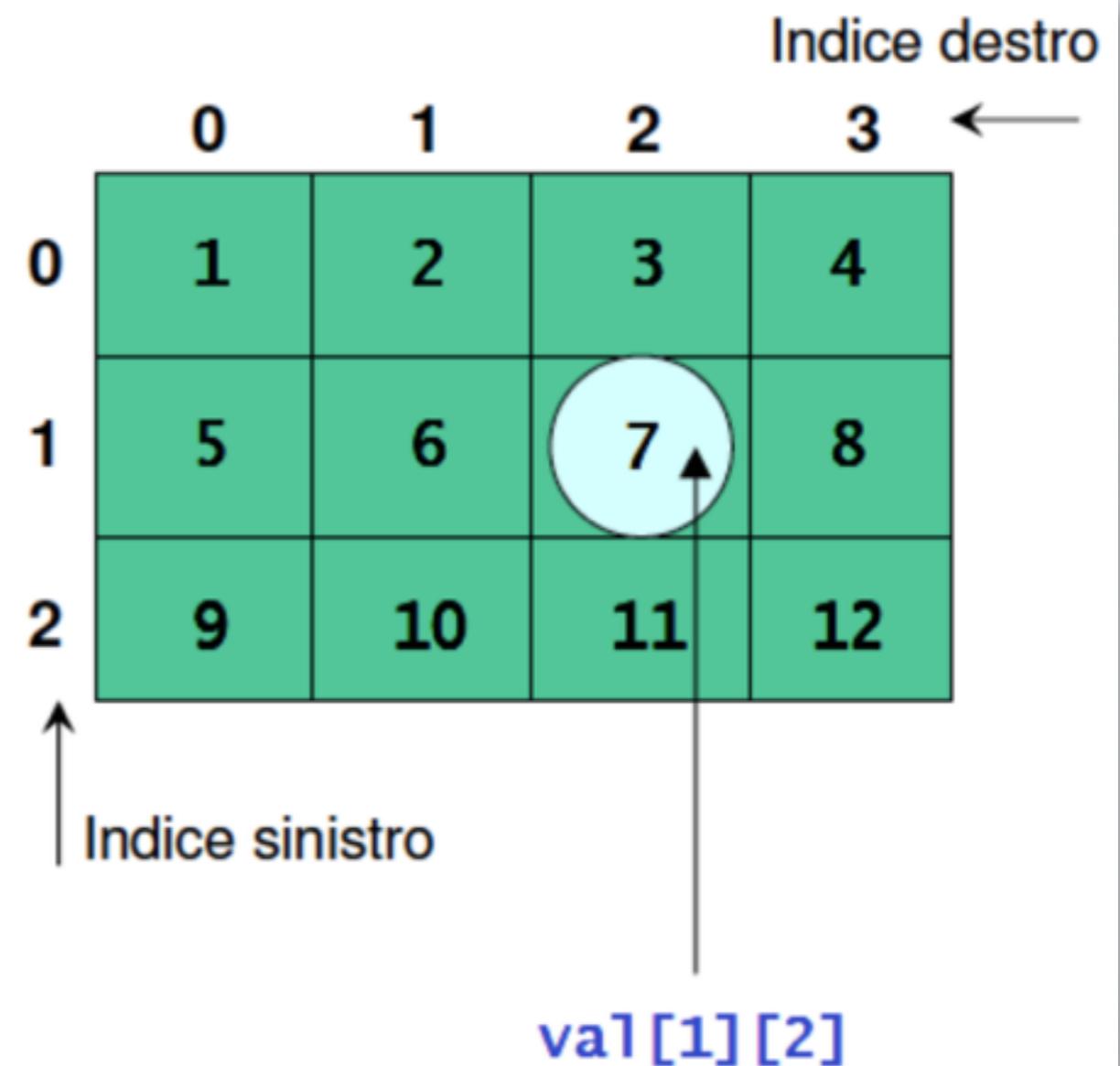
```
int b [2][2] = { {1}, {3,4} } ;
```

indici	0	1
0	1	0
1	3	4

# ARRAY MULTIDIMENSIONALI

```
#include <stdio.h>

int main()
{
    int i,j,val[3][4];
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            val[i][j]=(i*4)+j+1;
        }
    }
    return 0;
}
```



# ESERCIZI

- Esercizio n.14 selezione 2010-2011 (1 punto)
- Esercizio n.8 selezione 2012-2013 (2 punto)
- Esercizio n.12 selezione 2012-2013 (3 punto)

# PUNTATORI

# PUNTATORI

- una variabile, di qualunque tipo essa sia, rappresenta un valore posto da qualche parte in memoria RAM
- il compilatore si prende cura di tradurre i riferimenti agli spazi di memoria rappresentati simbolicamente attraverso gli identificatori.
- esempio:
  - $\text{int } x = 2;$

indirizzo	contenuto
$\&x = 1000$	2

# PUNTATORI

- una variabile, di qualunque tipo essa sia, rappresenta un valore posto da qualche parte in memoria RAM
- il compilatore si prende cura di tradurre i riferimenti agli spazi di memoria rappresentati simbolicamente attraverso gli identificatori.
- esempio:
  - `int x = 2;`

**l'operatore unario & (indirizzo-di)  
fornisce l'indirizzo di una variabile**

indirizzo	contenuto
<code>&amp;x = 1000</code>	2

# PUNTATORI

- Un puntatore è una variabile particolare che contiene indirizzi di memoria di altre variabili
- esempio:
  - `int x = 2;`
  - `int *ptr=&x;`

indirizzo	contenuto
<code>&amp;x = 1000</code>	2
<code>&amp;ptr=1500</code>	<code>&amp;x=1000</code>

# PUNTATORI

- Un puntatore è una variabile particolare che contiene indirizzi di memoria di altre variabili
- esempio:
  - `int x = 2;`
  - `int *ptr=&x;`
  - `*ptr=56;`

**l'operatore unario `*` (de-referenziazione) fornisce accesso alla variabile associata precedentemente**

indirizzo	contenuto
<code>&amp;x = 1000</code>	56
<code>&amp;ptr=1500</code>	<code>&amp;x=1000</code>

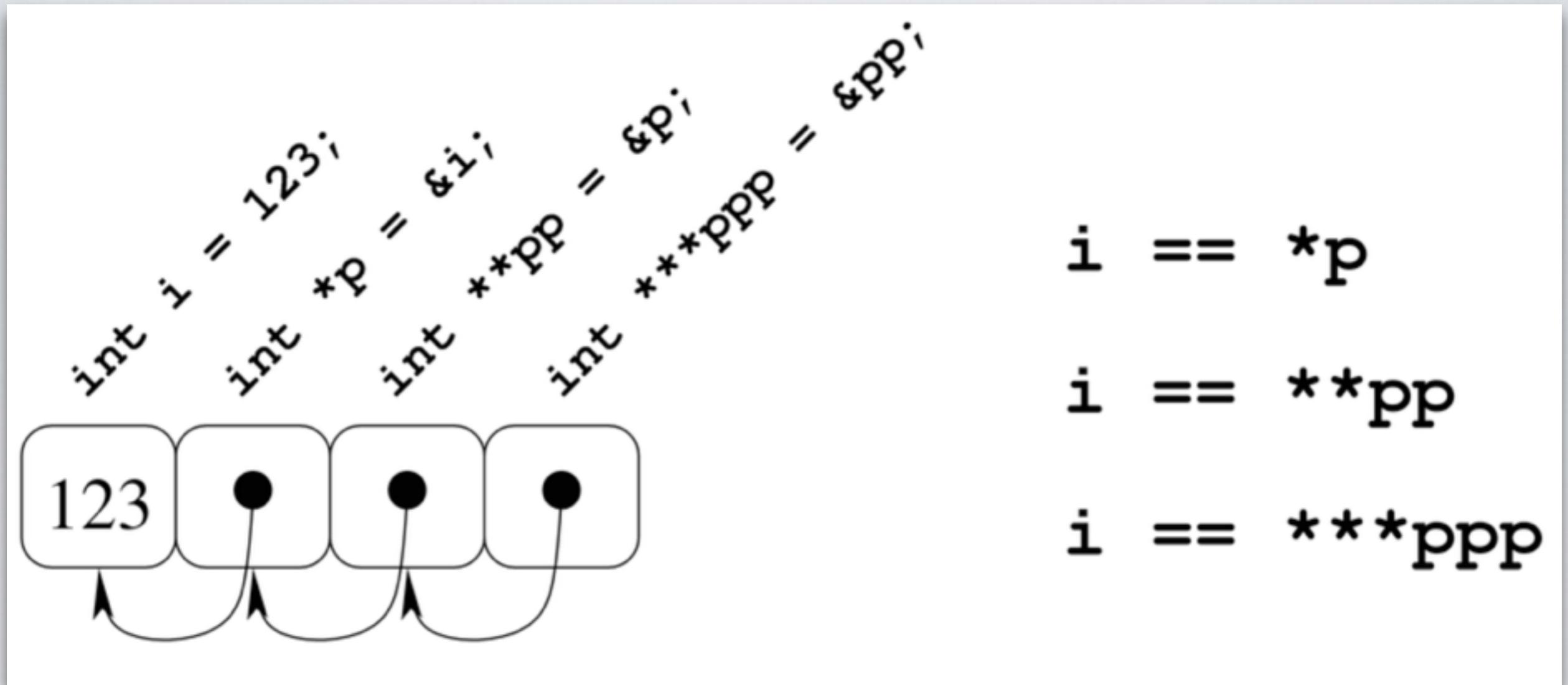
# PUNTATORI

- Un puntatore è una variabile particolare che contiene indirizzi di memoria di altre variabili
- esempio:
  - int x = 2;
  - int \*ptr=&x;
  - \*ptr=56;
- printf("%p, %p", &ptr, &x);
- output = **0x7fff5fbff8bc, 0x7fff5fbff8bc**

**%p è lo specificatore di formato per il puntatore!**

indirizzo	contenuto
&x = 1000	56
&ptr=1500	&x=1000

# PUNTATORI A PUNTATORI



# PUNTATORI & ARRAY

# PUNTATORI & ARRAY

- esempio:
  - `int x[] = {1,2,3};`

indirizzo	contenuto
<code>a=&amp;a[0]=1000</code>	1
<code>&amp;a[1]=1004</code>	2
<code>&amp;a[2]=1008</code>	3

# PUNTATORI & ARRAY

- esempio:
  - `int x[] = {1,2,3};`
  - `int *ptr=&a[0]; //equivalente *ptr=a;`

**DOMANDA:** perché specificare il tipo del puntatore (es. int) se poi esso dovrà contenere sempre e solo un indirizzo di memoria???

memoria

indirizzo	contenuto
<code>a=&amp;a[0]=1000</code>	1
<code>&amp;a[1]=1004</code>	2
<code>&amp;a[2]=1008</code>	3
<code>&amp;ptr=2000</code>	1000

# PUNTATORI & ARRAY

- esempio:
  - `int x[] = {1,2,3};`
  - `int *ptr=&a[0]; //equivalente *ptr=a;`
  - `ptr++;`

indirizzo	contenuto
<code>a=&amp;a[0]=1000</code>	1
<code>&amp;a[1]=1004</code>	2
<code>&amp;a[2]=1008</code>	3

**DOMANDA: a quale parte di memoria punterà adesso ptr???**

# PUNTATORI & ARRAY

- esempio:
  - `int x[] = {1,2,3};`
  - `int *ptr=&a[0]; //equivalente *ptr=a;`
  - `ptr++;`
  - `*ptr=32; //equivalente`  
`ptr[1]=32;`  
oppure  
`*(ptr+1)=0;`

indirizzo	contenuto
<code>a=&amp;a[0]=1000</code>	<code>1</code>
<code>&amp;a[1]=1004</code>	<code>32</code>
<code>&amp;a[2]=1008</code>	<code>3</code>
<code>&amp;ptr=2000</code>	<code>1004</code>

# PUNTATORI & ARRAY

- esempio:
  - `int x[] = {1,2,3};`
  - `int *ptr=&a[0]; //equivalente *ptr=a;`
  - `ptr++;`
  - `*ptr=32; //equivalente ptr[1]=32;`

oppure

`*(ptr+1)=0;`

**incrementare un puntatore vuol dire farlo avanzare di tanti byte quanto è specificato dal tipo puntato (es. `sizeof(int)=4` o `sizeof(double)=8` saranno incrementi diversi)!!!**

indirizzo	contenuto
<code>a=&amp;a[0]=1000</code>	<code>1</code>
<code>&amp;a[1]=1004</code>	<code>32</code>
<code>&amp;a[2]=1008</code>	<code>3</code>
<code>&amp;ptr=2000</code>	<code>1004</code>

# FUNZIONI & PROCEDURE

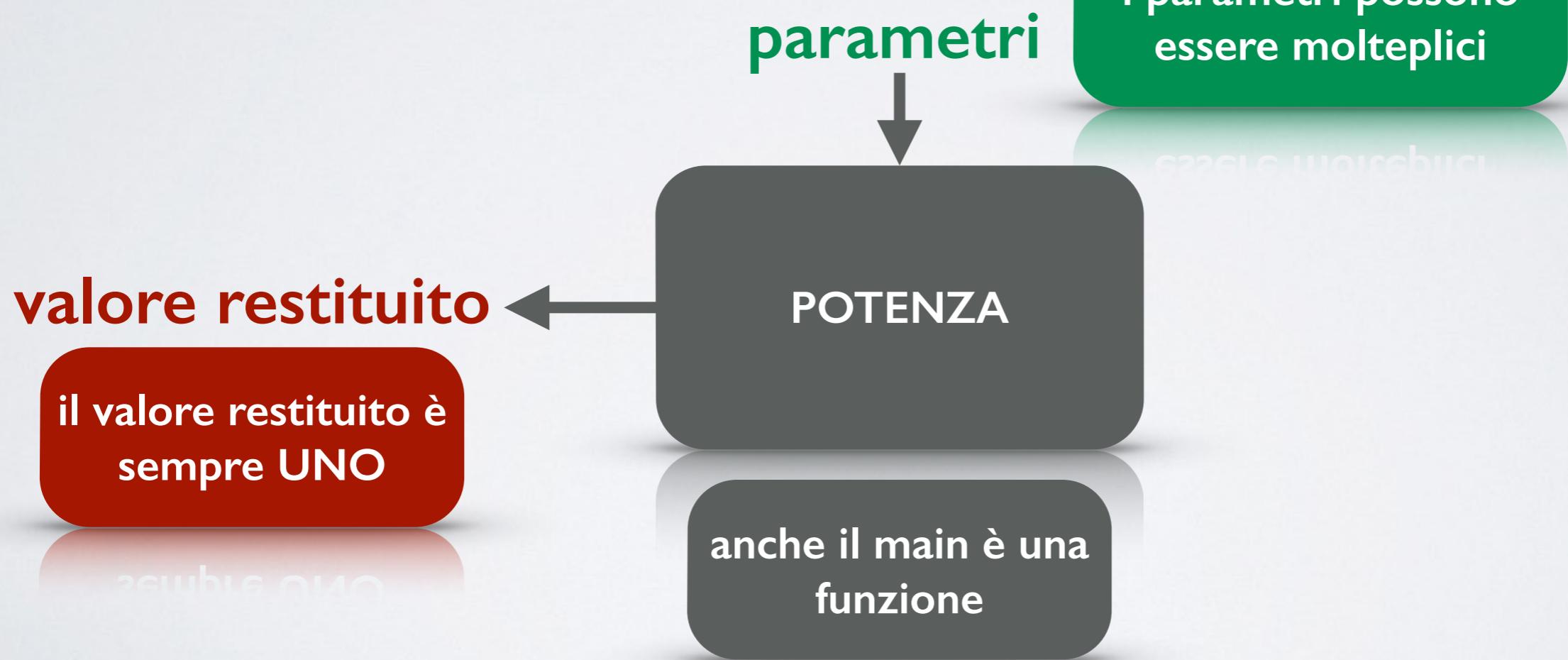
# FUNZIONI

- Una **funzione** è una porzione di codice, detto anche **sottoprogramma** o, in inglese, **subroutine**, che può essere richiamata più volte in programma
- Le funzioni possono:
  - Essere scritte dal **programmatore** ed utilizzate in un programma
  - Essere reperite in **librerie** di codice

libreria <stdio.h>

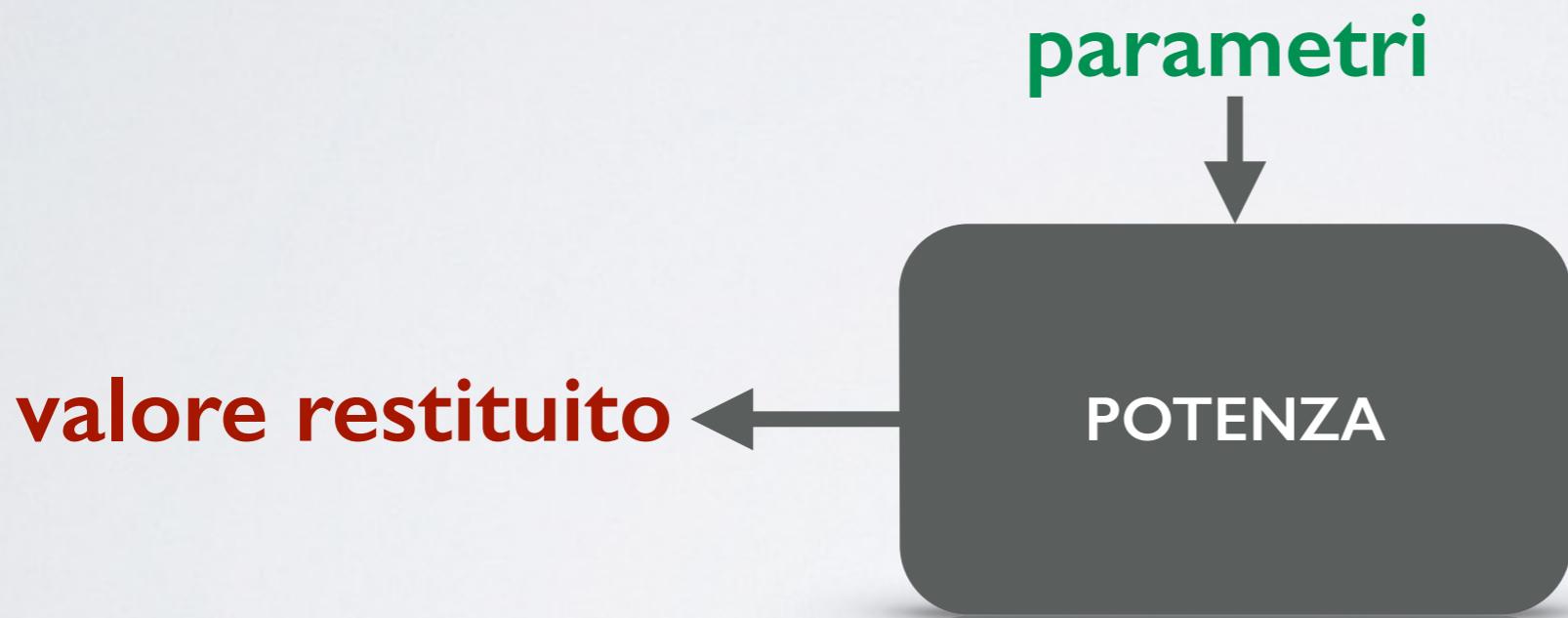
# FUNZIONI

- Possiamo immaginare una funzione come una **black-box!**
- Esempio: definizione di **potenza()**



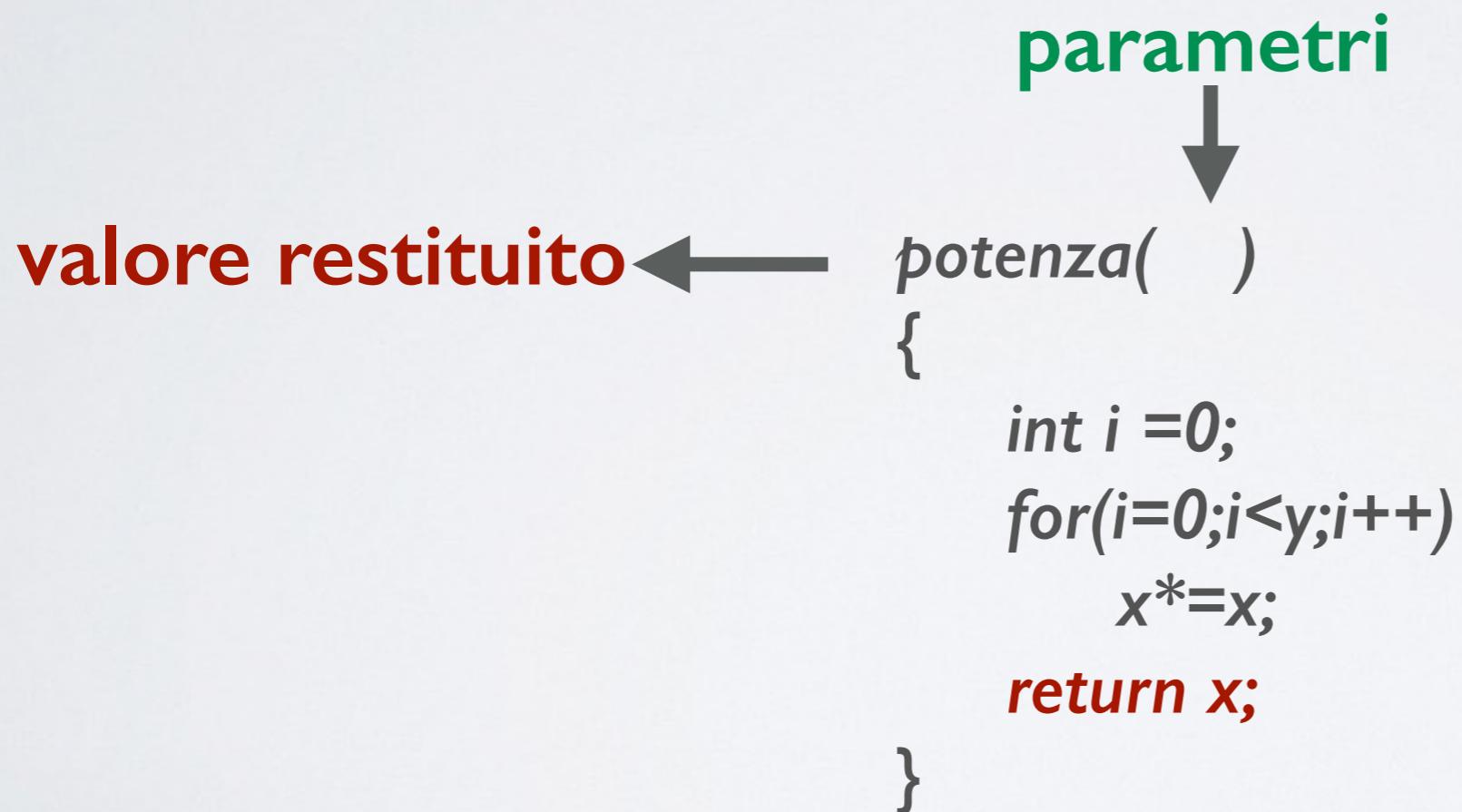
# FUNZIONI

- Possiamo immaginare una funzione come una **black-box!**
- Esempio: definizione di **potenza()**



# FUNZIONI

- Possiamo immaginare una funzione come una **black-box!**
- Esempio: definizione di **potenza()**



# FUNZIONI

- Possiamo immaginare una funzione come una **black-box!**
- Esempio: definizione di **potenza()**

parametri



```
int potenza( )  
{  
    int i =0;  
    for(i=0;i<y;i++)  
        x*=x;  
    return x;  
}
```

# FUNZIONI

- Possiamo immaginare una funzione come una **black-box!**
- Esempio: definizione di **potenza()**

```
int potenza(int x, int y)
{
    int i =0;
    for(i=0;i<y;i++)
        x*=x;
    return x;
}
```

$x^*=x$ ; equivale a  $x=x^*x$ ;

# USO DI FUNZIONI

- una funzione che restituisce un valore è solitamente presente come **RIGHT-VALUE**

LEFT-VALUE = RIGHT-VALUE

# USO DI FUNZIONI

- una funzione che restituisce un valore è solitamente presente come **RIGHT-VALUE**

LEFT-VALUE

= *potenza(x);*

# USO DI FUNZIONI

- una funzione che restituisce un valore è solitamente presente come  
**RIGHT-VALUE**

...

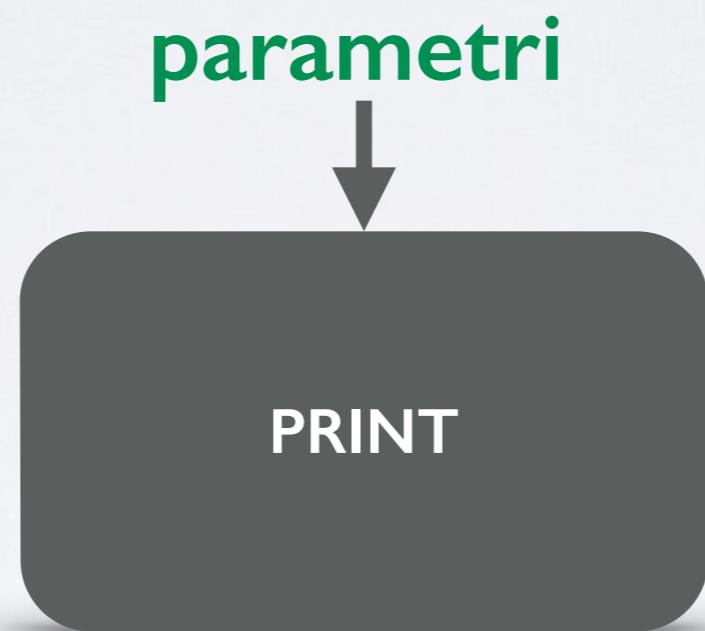
*ris = potenza(x);*

...

# PROCEDURE

# PROCEDURE

- Una *funzione* che non restituisce un valore viene detta *procedura*



# PROCEDURE

- Una *funzione* che non restituisce un valore viene detta *procedura*

parametri



```
void print( )  
{  
    printf("%d",x);  
}
```

# PROCEDURE

- Una *funzione* che non restituisce un valore viene detta *procedura*

```
void print(int x)
{
    printf("%d",x);
}
```

# USO DI PROCEDURE

- una procedure NON restituisce un valore ed è semplicemente invocata (come per esempio la printf())

PROCEDURA

# USO DI PROCEDURE

- una procedure NON restituisce un valore ed è semplicemente invocata (come per esempio la printf())

```
...
print(x);
...
```

# DICHIARAZIONE E DEFINIZIONE DI FUNZIONI/PROCEDURE

# DEFINIZIONE

```
int main(int argc, const char * argv[])
{
    print(3);
    return 0;
}

void print(int x){printf("%d",x);}
```

# DEFINIZIONE

```
int main(int argc, const char * argv[])
{
    print(3);
    return 0;
}

void print(int x){printf("%d",x);}
```

quale sarà l'output del seguente  
programma?

# DEFINIZIONE

```
int main(int argc, const char * argv[])
{
    print(3);
    return 0;
}
```

⚠ Implicit declaration of function 'print' is invalid in C99

❗ Conflicting types for 'print'

# DEFINIZIONE

```
int main(int argc, const char * argv[])
{
    print(3);
    return 0;
}
```

⚠ Implicit declaration of function 'print' is invalid in C99

! void print(int x){printf("%d",x);} ⚠ Conflicting types for 'print'

come risolvere l'errore ?

# DICHIARAZIONE

```
void print(int x);

int main(int argc, const char * argv[])
{
    print(3);
    return 0;
}

void print(int x){printf("%d",x);}
```

aggiungendo la DICHIARAZIONE  
(prototipo)!

# DICHIARAZIONE

```
void print(int x);

int main(int argc, const char * argv[])
{
    print(3);
    return 0;
}

void print(int x){printf("%d",x);}
```

aggiungendo la DICHIARAZIONE  
(prototipo)!

in alternativa bisogna inserire la definizione sopra ma può comportare dei problemi poiché il codice per le funzioni successive non è visto dalle precedenti

# ESEMPI DI FUNZIONI

# FUNZIONE SWAP

```
#include <stdio.h>

void swap(int p1, int p2); //dichiarazione della funzione swap

int main()
{
    int a,b;
    a=3;
    b=2;
    swap(a,b);
    printf("\n a=%d b=%d \n",a,b);
}

void swap(int p1, int p2) //definizione della funzione swap
{
    int tmp;
    tmp = p1;
    p1 = p2;
    p2 = tmp;
}
```

Quale sarà il risultato?

# FUNZIONE SWAP

```
#include <stdio.h>

void swap(int p1, int p2); //dichiarazione della funzione swap

int main()
{
    int a,b;
    a=3;
    b=2;
    swap(a,b);
    printf("\n a=%d b=%d \n",a,b);
}

void swap(int p1, int p2) //definizione della funzione swap
{
    int tmp;
    tmp = p1;
    p1 = p2;
    p2 = tmp;
}
```

a=3 b=2

Dunque lo scambio NON è stato  
effettuato

# FUNZIONE SWAP

```
#include <stdio.h>

void swap(int *p1, int *p2); //dichiarazione della funzione swap

int main()
{
    int a,b;
    a=3;
    b=2;
    swap(&a,&b);
    printf("\n a=%d b=%d \n",a,b);
}

void swap(int *p1, int *p2) //definizione della funzione swap
{
    int tmp;
    tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
```

Quale sarà il risultato questa volta?

# FUNZIONE SWAP

```
#include <stdio.h>

void swap(int *p1, int *p2); //dichiarazione della funzione swap

int main()
{
    int a,b;
    a=3;
    b=2;
    swap(&a,&b);
    printf("\n a=%d b=%d \n",a,b);
}

void swap(int *p1, int *p2) //definizione della funzione swap
{
    int tmp;
    tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
```

a=2 b=3

Adesso i due valori sono stati  
scambiati

# FUNZIONI RICORSIVE

# RICORSIONE

- Una funzione è ricorsiva quando nel suo corpo c'è un'invocazione a se stessa

*<<L'iterazione è umana, la ricorsione è divina>>*

*[Peter Deutsch]*

# RICORSIONE

- **Struttura di una funzione ricorsiva:**

- Ci sono uno o più casi semplici (**casi base**), per i quali esiste una soluzione immediata NON ricorsiva
- I rimanenti casi (**casi induttivi**) possono essere ricondotti a problemi che sono più vicini ai casi base
- Prima o poi il problema si riduce a soli casi base

# RICORSIONE

- **Struttura di una funzione ricorsiva:**

**if (sono in un caso base)** lo risolvo

**else** riduco il problema a un caso più semplice utilizzando la **ricorsione**

# FUNZIONI RICORSIVE: FATTORIALE

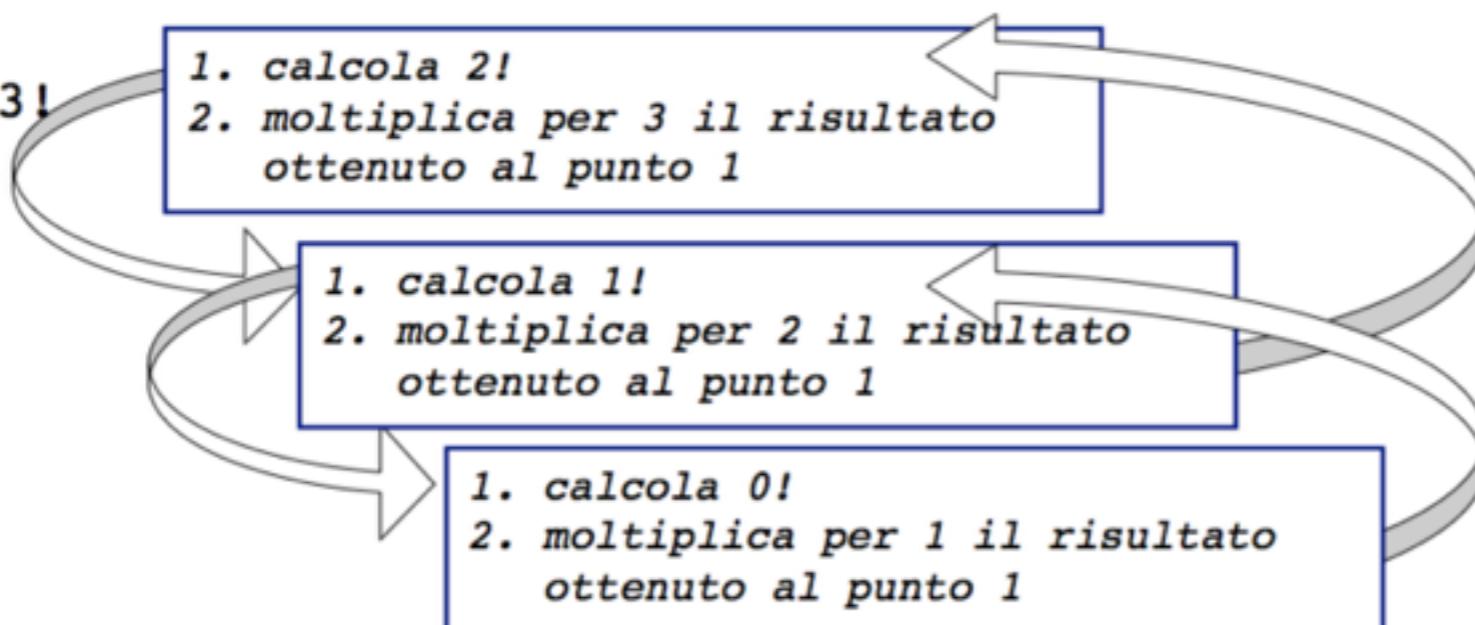
# RICORSIONE: FATTORIALE

**Fattoriale**:  $n! := \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$

dove  $0! := 1$

ricorsivamente  $n! := \begin{cases} 1 & \text{se } n = 0; \\ n(n-1)! & \text{se } n \geq 1. \end{cases}$

il calcolo di  $3!$



# RICORSIONE: FATTORIALE

$$\text{Fattoriale : } n! := \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

**fattoriale(n) = n\*fattoriale(n-1)** altrimenti

# RICORSIONE: FATTORIALE

$$\text{Fattoriale : } n! := \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

**fattoriale(n) = n\*fattoriale(n-1)** altrimenti

# Versione ricorsiva

```
long fattoriale(long n)
{
    if (n==0) // caso base
        return 1;
    else
        return n* fattoriale(n-1);
}
```

# RICORSIONE: FATTORIALE

$$\text{Fattoriale : } n! := \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

**fattoriale(n) = n\*fattoriale(n-1)** altrimenti

# Versione iterativa

# Versione ricorsiva

```
long fattoriale(long n)
{
    if (n==0) // caso base
        return 1;
    else
        return n* fattoriale(n-1);
}
```

```
long fattorialeIterativo (long n)
{
    long fatt = 1;
    int i;

    for(i=n;i>=1;i--)
        fatt = fatt * i;

    return fatt;
}
```

# FUNZIONI RICORSIVE: M.C.D.

# RICORSIONE: M.C.D.

**Massimo comun divisore:** di  $a$  e  $b$  è il numero più grande per il quale entrambi possono essere divisi senza resto

$$\text{MCD}(p, 0) = p$$

$$\text{MCD}(p, q) = \text{MCD}(q, p \bmod q)$$

# RICORSIONE: M.C.D.

**Massimo comun divisore:** di  $a$  e  $b$  è il numero più grande per il quale entrambi possono essere divisi senza resto

$$\text{MCD}(p, 0) = p \quad \text{se } q=0$$

$$\text{MCD}(p, q) = \text{MCD}(q, p \bmod q) \quad \text{altrimenti}$$

# RICORSIONE: M.C.D.

**Massimo comun divisore:** di  $a$  e  $b$  è il numero più grande per il quale entrambi possono essere divisi senza resto

$$\begin{aligned} \text{MCD}(p, 0) &= p && \text{se } q=0 \\ \text{MCD}(p, q) &= \text{MCD}(q, p \bmod q) && \text{altrimenti} \end{aligned}$$

## Versione ricorsiva

```
int mcd(int p, int q){  
    if (q == 0)  
        return p;  
    else  
        return mcd(q, p % q);  
}
```

# RICORSIONE: M.C.D.

**Massimo comun divisore:** di  $a$  e  $b$  è il numero più grande per il quale entrambi possono essere divisi senza resto

$$\text{MCD}(p, 0) = p \quad \text{se } q=0$$

$$\text{MCD}(p, q) = \text{MCD}(q, p \bmod q) \quad \text{altrimenti}$$

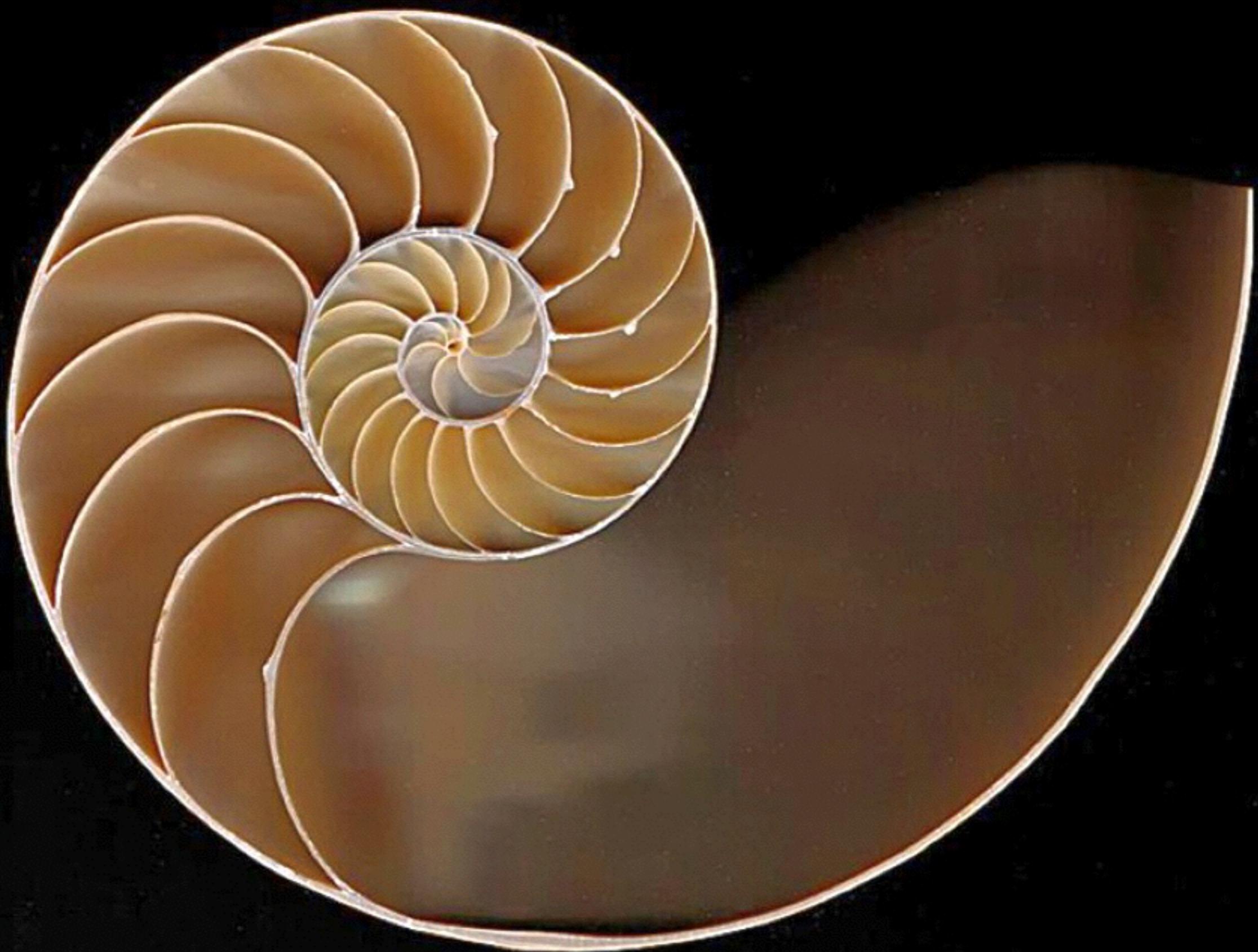
Versione iterativa

Versione ricorsiva

```
int mcd(int p, int q){  
    if (q == 0)  
        return p;  
    else  
        return mcd(q, p % q);  
}
```

```
int mcdIterativo(int p, int q){  
    int r;  
    while (q != 0) {  
        r = p % q;  
        p = q;  
        q = r;  
    }  
    return p;
```

# FUNZIONI RICORSIVE: FIBONACCI



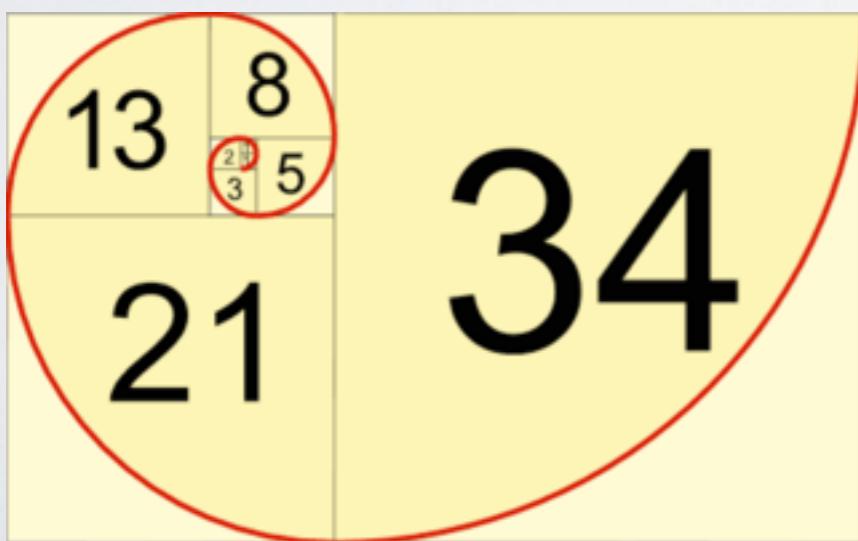
# RICORSIONE: FIBONACCI

*Successione di Fibonacci* : sequenza di numeri in cui ogni termine, a parte i primi due, è la somma dei due che lo precedono

$$\text{Fibonacci}(0) = 0$$

$$\text{Fibonacci}(1) = 1$$

$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$



# RICORSIONE: FIBONACCI

*Successione di Fibonacci* : sequenza di numeri in cui ogni termine, a parte i primi due, è la somma dei due che lo precedono

$$\text{Fibonacci}(n) = n$$

$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$

se  $n=0$  oppure  $n=1$

altrimenti

# RICORSIONE: FIBONACCI

*Successione di Fibonacci* : sequenza di numeri in cui ogni termine, a parte i primi due, è la somma dei due che lo precedono

$$\text{Fibonacci}(n) = n$$

$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$

se  $n=0$  oppure  $n=1$

altrimenti

## Versione ricorsiva

```
long fibonacci(long n){  
    if(n==0 || n==1) return n;  
    else return fibonacci(n-1)+fibonacci(n-2);  
}
```

# RICORSIONE: FIBONACCI

*Successione di Fibonacci* : sequenza di numeri in cui ogni termine, a parte i primi due, è la somma dei due che lo precedono

$$\text{Fibonacci}(n) = n$$

$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$

se  $n=0$  oppure  $n=1$   
altrimenti

## Versione iterativa

### Versione ricorsiva

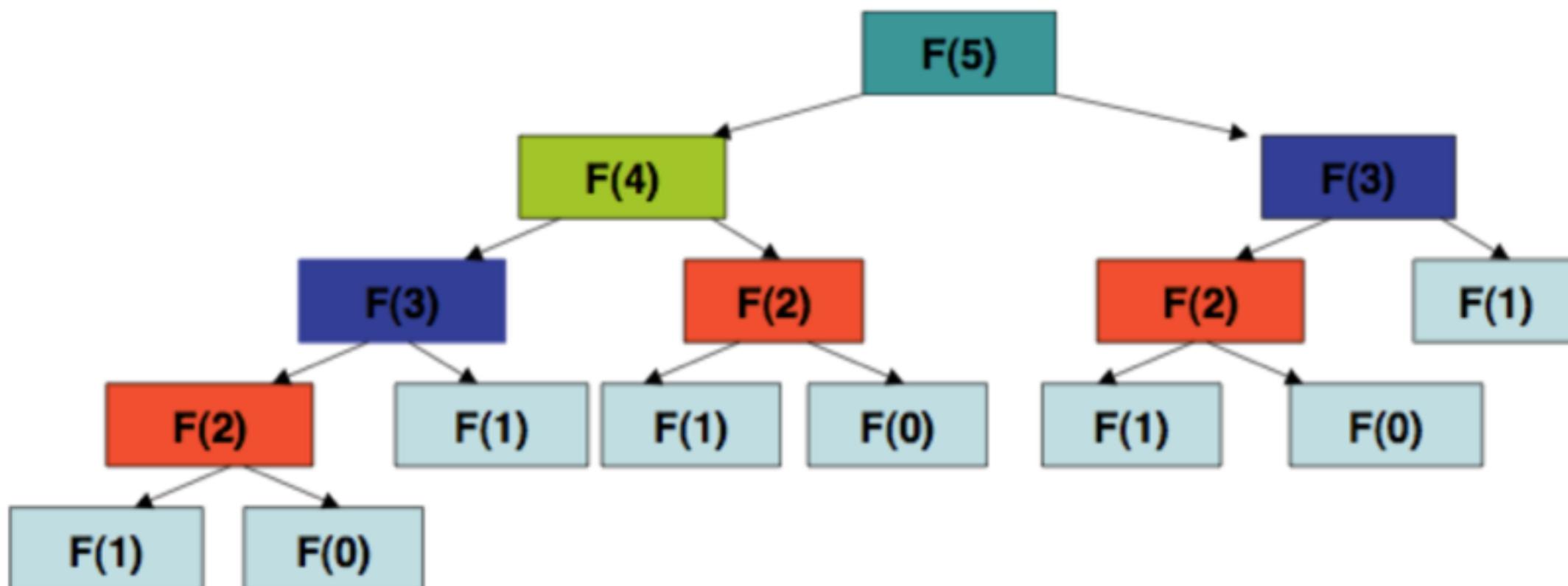
```
long fibonacci(long n){  
    if(n==0 || n==1) return n;  
    else return fibonacci(n-1)+fibonacci(n-2);  
}
```

```
long fibonacciIterativo(long n){  
    long fn1=1;  
    long fn2=0;  
    long tmp;  
    int i;  
  
    for(i=n;i>1;i--){  
        tmp=fn1;  
        fn1=fn2+fn1;  
        fn2=tmp;  
    }  
  
    return fn1;  
}
```

# RICORSIONE: FIBONACCI

**Nota:** la versione *ricorsiva* della funzione di fibonacci è inefficiente!

- I valori intermedi vengono ricalcolati ogni volta che occorrono



- $\text{fib}(3)$  viene calcolato 2 volte,     $\text{fib}(2)$  viene calcolato 3 volte, ...

# RICORSIONE VS ITERAZIONE

- Qualunque **funzione ricorsiva** può essere trasformata in una **funzione iterativa** e viceversa
- Le **funzioni ricorsive** sono più chiare, più semplici, più brevi e più facili da comprendere delle corrispondenti versioni iterative
  - il programma riflette fedelmente la strategia di soluzione del problema
- Le **funzioni iterative** sono più veloci, ma talvolta rischiano di diventare infinitamente complesse da scrivere e molto più soggette a contenere errori

# ESERCIZI

- Esercizio n.9 selezione 2012-2013 (2 punto)
- Esercizio n.11 selezione 2012-2013 (3 punto)

GRAZIE A TUTTI PER LA CORTESE ATTENZIONE...

