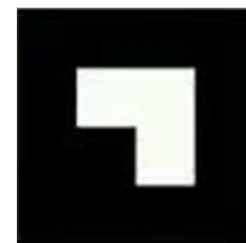
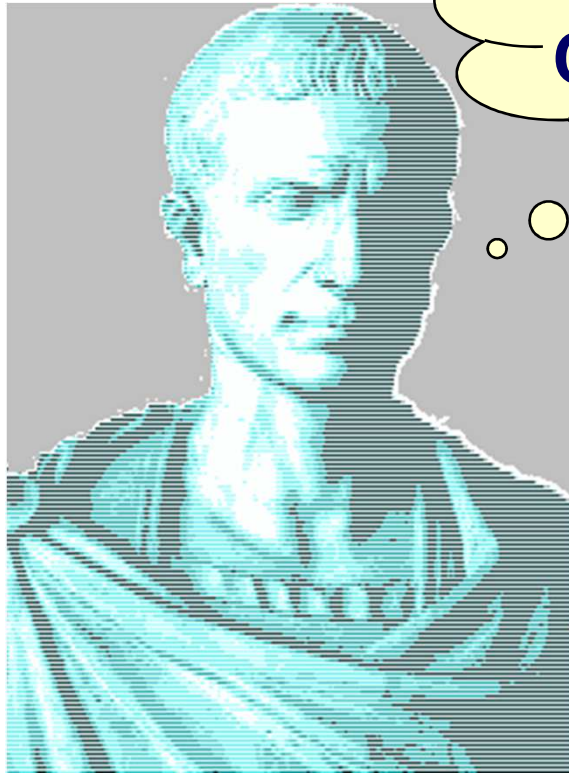


Funzioni



Sottoprogrammi



**Divide et impera!
Con i sottoprogrammi!**

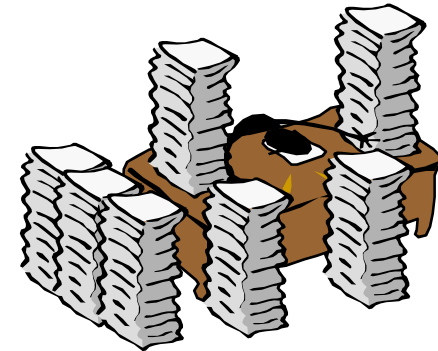
Sottoprogrammi

- Un programma realistico può consistere di migliaia di istruzioni
- Una banca ha migliaia di programmi, anche corrispondenti a 100 Milioni di istruzioni e una singola funzione può contenere anche 100.000 istruzioni



Sottoprogrammi

- Sebbene fattibile, una soluzione “monolitica” del problema:
 - Non è molto produttiva:
 - Riutilizzo del codice?
 - Comprensione del codice?
 - Non è intuitiva:
 - Tendenza ad “organizzare” in modo strutturato
 - Struttura gerarchica a partire dal problema complesso fino a sottoproblemi sempre più semplici
- Approccio *top-down*



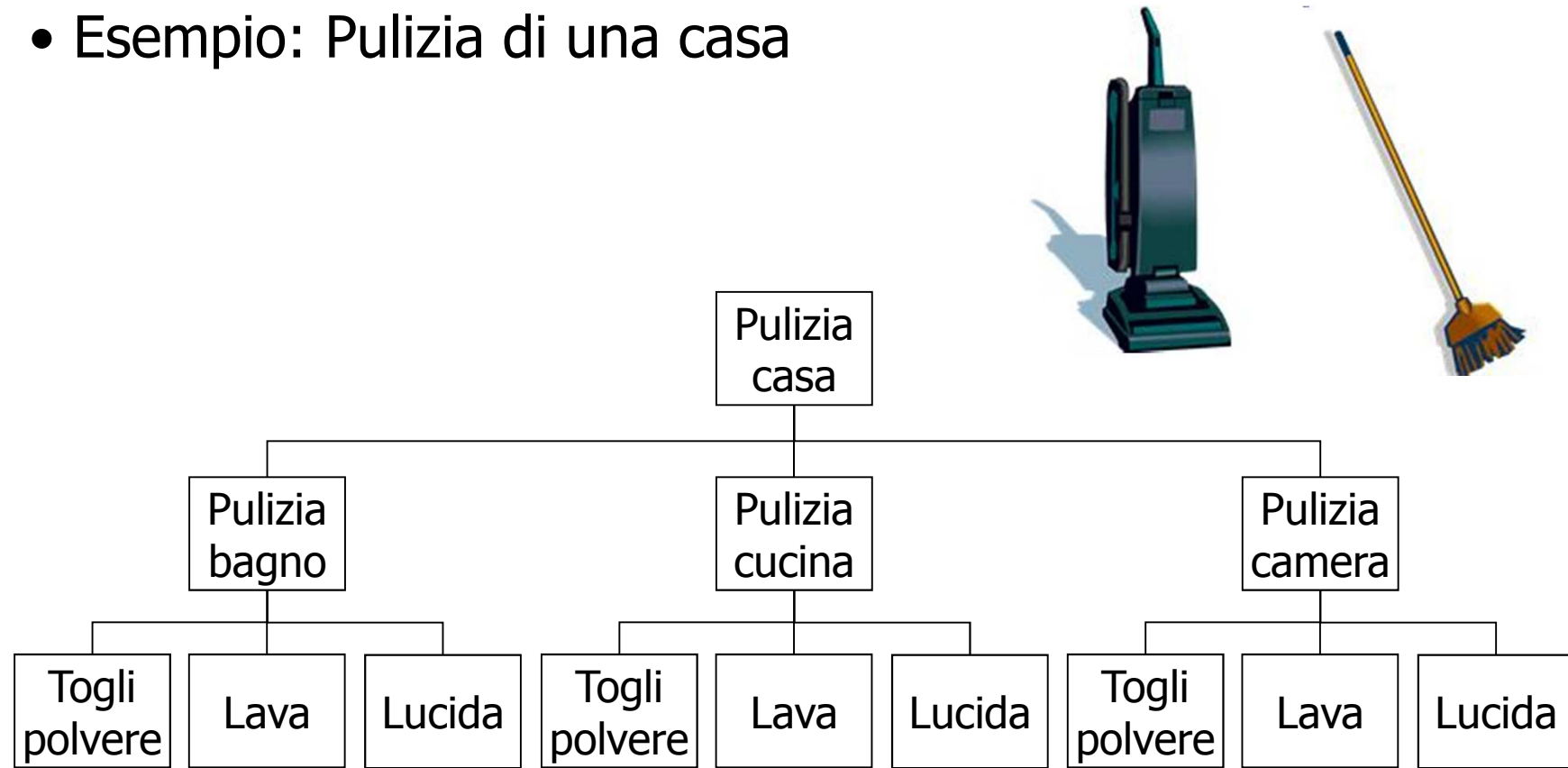
Approccio top-down

- Decomposizione del problema in sottoproblemi più semplici (soluzione ad albero)
- Ripetibile su più livelli
- Sottoproblemi “terminali” = Risolvibili in modo “semplice”



Approccio top-down (Cont.)

- Esempio: Pulizia di una casa



Approccio top-down (Cont.)

- Esempio: Pulizia di una casa



Approccio top-down (Cont.)

- I linguaggi di programmazione permettono di suddividere le istruzioni in blocchi detti **sottoprogrammi, moduli,...**
- La gerarchia delle operazioni si traduce in una gerarchia di sottoprogrammi

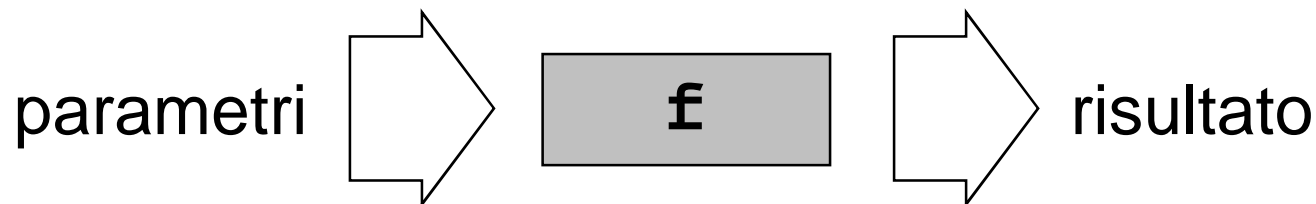
procedure

funzioni

subroutine

Funzioni e procedure

- Procedure:
 - Sottoprogrammi che NON ritornano un risultato
- Funzioni:
 - Sottoprogrammi che ritornano un risultato (di qualche tipo primitivo o non)
- In generale, procedure e funzioni hanno dei *parametri* (o *argomenti*)
 - Vista funzionale:




Funzioni e procedure in C

- Nel C K&R:
 - Esistono solo funzioni (tutto ritorna un valore)
 - Si può ignorare il valore ritornato dalle funzioni
- Dal C89 (ANSI) in poi:
 - Funzioni il cui valore di ritorno deve essere ignorato (`void`)
 - Funzioni `void` \leftrightarrow procedure

Chiamate di funzioni

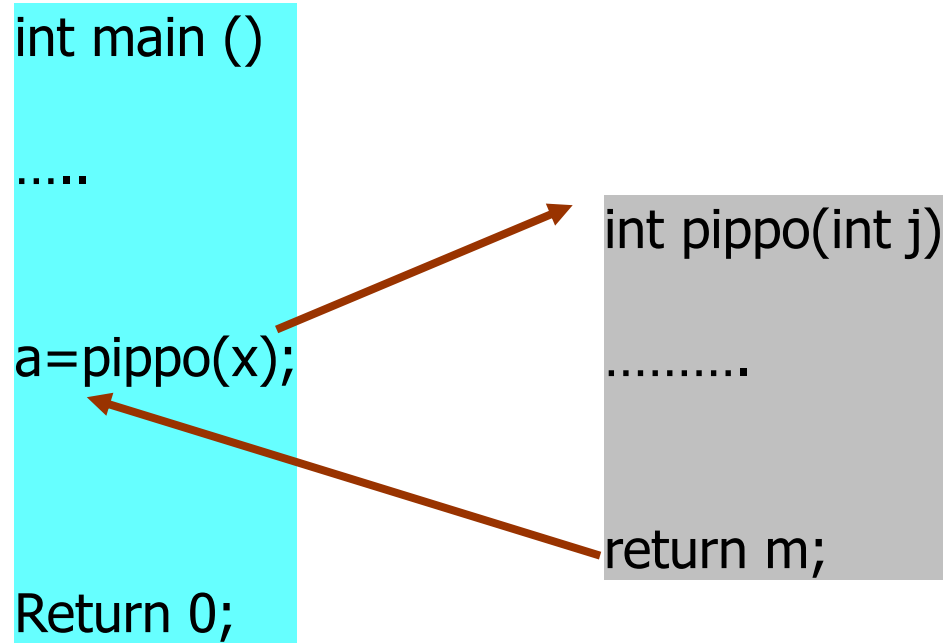
```
int main ()  
.....  
.....  
.....  
return 0;
```



Estrarre una porzione di codice



Chiamate di funzioni



- `int cubo (int a);`

- `int main()`
- `{`
- `int x,n;`
- `printf("numero=");`
- `scanf("%d",&x);`
- `n=cubo(x);`
- `printf ("cubo= %d",n);`
- `return 0;`
- `}`
- `int cubo (int a){`
- `int c;`
- `c=a*a*a;`
- `return c;`
- `}`

dichiarazione o prototipo

- `int cubo (int a);`

- `int main()`

- `{`

- `int x,n;`

- `printf("numero=");`

- `scanf("%d",&x);`

- `n=cubo(x);`

- `printf ("cubo= %d",n);`

- `return 0;`

- `}`

- `int cubo (int a){`

- `int c;`

- `c=a*a*a;`

- `return c;`

- `}`

chiamata



- `int cubo (int a);`

- `int main()`

- `{`

- `int x,n;`

- `printf("numero=");`

- `scanf("%d",&x);`

- `n=cubo(x);`

- `printf ("cubo= %d",n);`

- `return 0;`

- `}`

- `int cubo (int a){`

- `int c;`

- `c=a*a*a;`

- `return c;`

- `}`

definizione o corpo



Definizione di una funzione

- Stabilisce un “nome” per un insieme di operazioni

- Sintassi:

```
<tipo risultato> <nome funzione> (<parametri formali>)  
{  
    <istruzioni>  
}
```

- Se la funzione non ha un risultato, <**tipo risultato**> deve essere `void`
- Per ritornare il controllo alla funzione *chiamante*, nelle <**istruzioni**> deve comparire una istruzione
 - `return <valore>;` **se non** `void`
 - `return;` **se** `void`

Definizione di una funzione (Cont.)

- Tutte le funzioni sono definite allo stesso livello del `main()`
 - NON si può definire una funzione dentro un'altra
- `main()` è una funzione!
 - Tipo del valore di ritorno: `int`
 - Parametri: Vedremo più avanti!

Principio di funzionamento (1/3)

```
int main(void)
{
    int x, y ;

    /* leggi un numero
       tra 50 e 100 e
       memorizzalo
       in x */
    /* leggi un numero
       tra 1 e 10 e
       memorizzalo
       in y */

    printf("%d %d\n",
           x, y ) ;
}
```

Principio di funzionamento (2/3)

```
int main(void)
{
    int x, y ;

    x = leggi(50, 100) ;
    y = leggi(1, 10) ;

    printf("%d %d\n",
        x, y ) ;
}
```

Principio di funzionamento (3/3)

```
int main(void)
{
    int x, y ;

    x = leggi(50, 100) ;
    y = leggi(1, 10) ;

    printf("%d %d\n",
           x, y ) ;
}
```

```
int leggi(int min,
           int max)
{
    int v ;

    do {
        scanf("%d", &v) ;
    } while( v<min ||
            v>max) ;

    return v ;
}
```

Principio di funzionamento (3/3)

```
int main(void)
{
    int x, y ;

    x = leggi(50, 100) ;
    y = leggi(1, 10) ;

    printf("%d %d\n",
           x, y ) ;
}
```

min=50

max=100

```
int leggi(int min,
          int max)
{
    int v ;

    do {
        scanf("%d", &v) ;
    } while( v<min ||
            v>max) ;

    return v ;
}
```

Principio di funzionamento (3/3)

```
int main(void)
{
    int x, y ;

    x = leggi(50, 100) ;
    y = leggi(1, 10) ;

    printf("%d %d\n",
        x, y ) ;
}
```

Chiamante

min=50

max=100

```
int leggi(int min,
          int max)
{
    int v ;

    do {
        scanf("%d", &v) ;
    } while( v<min ||
            v>max) ;

    return v ;
}
```

Chiamato

Principio di funzionamento (3/3)

```
int main(void)
{
    int x, y ;

    x = leggi(50, 100) ;
    y = leggi(1, 10) ;

    printf("%d %d\n",
           x, y ) ;
}
```

min=1

max=10

```
int leggi(int min,
          int max)
{
    int v ;

    do {
        scanf("%d", &v) ;
    } while( v<min ||
            v>max) ;

    return v ;
}
```

Dichiarazione o prototipo

```
int leggi(int min, int max) ;
```

Tipo del
valore di
ritorno

Nome della
funzione

Tipi e nomi
dei parametri
formali

Punto-e-
virgola

Definizione o implementazione

Tipo del
valore di
ritorno

Nome della
funzione

Tipi e nomi
dei parametri
formali

```
int leggi(int min, int max)
{
    ... codice della funzione ...
}
```

Corpo della funzione
{ ... }

Nessun
punto-e-virgola

Chiamata o invocazione

Funzione chiamante

Valori dei parametri attuali

```
int main(void)
{
    int x, a, b ;
    ...
    x = leggi(a, b) ;
    ..
}
```

Uso del valore di ritorno

Chiamata della funzione

Prototipi

- Così come per le variabili, è buona pratica dichiarare all'inizio del programma le funzioni prima del loro uso (**prototipi**)
- Sintassi:
 - Come per la definizione, ma si omette il contenuto (istruzioni) della funzione

Prototipi: Esempio

```
#include <stdio.h>

int func1(int a);
int func2(float b);
...

main ()
{
...
}

int func1(int a)
{
...
}

int func2(float b)
{
...
}
```

Corpo della funzione



Variabili locali

- All'interno del corpo di una funzione è possibile definire delle **variabili locali**

```
int leggi(int min,  
          int max)  
{  
    int v ;  
    scanf("%d", &v) ;  
    return v ;  
}
```

Istruzioni eseguibili

- Il corpo di una funzione può contenere qualsiasi combinazione di istruzioni eseguibili
- Ricordare l'istruzione return

```
int leggi(int min,  
          int max)  
{  
    int v ;  
    scanf("%d", &v) ;  
    return v ;  
}
```

Utilizzo di una funzione

- Deve rispettare l'interfaccia della definizione
- Utilizzata come una normale istruzione
<variabile> = <nome funzione> (<parametri attuali>);
- Può essere usata ovunque
 - Una funzione può anche invocare se stessa (funzione ricorsiva)

Utilizzo di una funzione: Esempio

```
#include <stdio.h>

int modabs(int v1, int v2);    //prototipo

main() {
    int x,y,d;
    scanf("%d %d",&x,&y);
    d = modabs(x,y);          // utilizzo
    printf("%d\n",d);
}

int modabs (int v1, int v2)    // definizione
{
    int v;
    if (v1>=v2) {
        v = v1-v2;
    } else {
        v = v2-v1;
    }
    return v;
}
```

Passaggio dei parametri



Funzioni e parametri

- Parametri e risultato sono sempre associati ad un tipo

- Esempio:

```
float media(int a, int b)
```



- I tipi di parametri e risultato devono essere rispettati quando la funzione viene utilizzata!

- Esempio:

```
float x; int a,b;  
x = media(a, b);
```

Parametri formali e attuali

- E' importante distinguere tra:
 - Parametri **formali**:
Specificati nella definizione di una funzione
 - Parametri **attuali**:
Specificati durante il suo utilizzo
- Esempio:
 - funzione `Func`
 - Definizione: `double Func(int x, double y)`
 - Parametri formali: `(x, y)`
 - Utilizzo: `z = Func(a*2, 1.34);`
 - Parametri attuali: (Risultato di `a*2`, `1.34`)

Parametri formali e attuali (Cont.)

- Vista funzionale:

- Definizione:



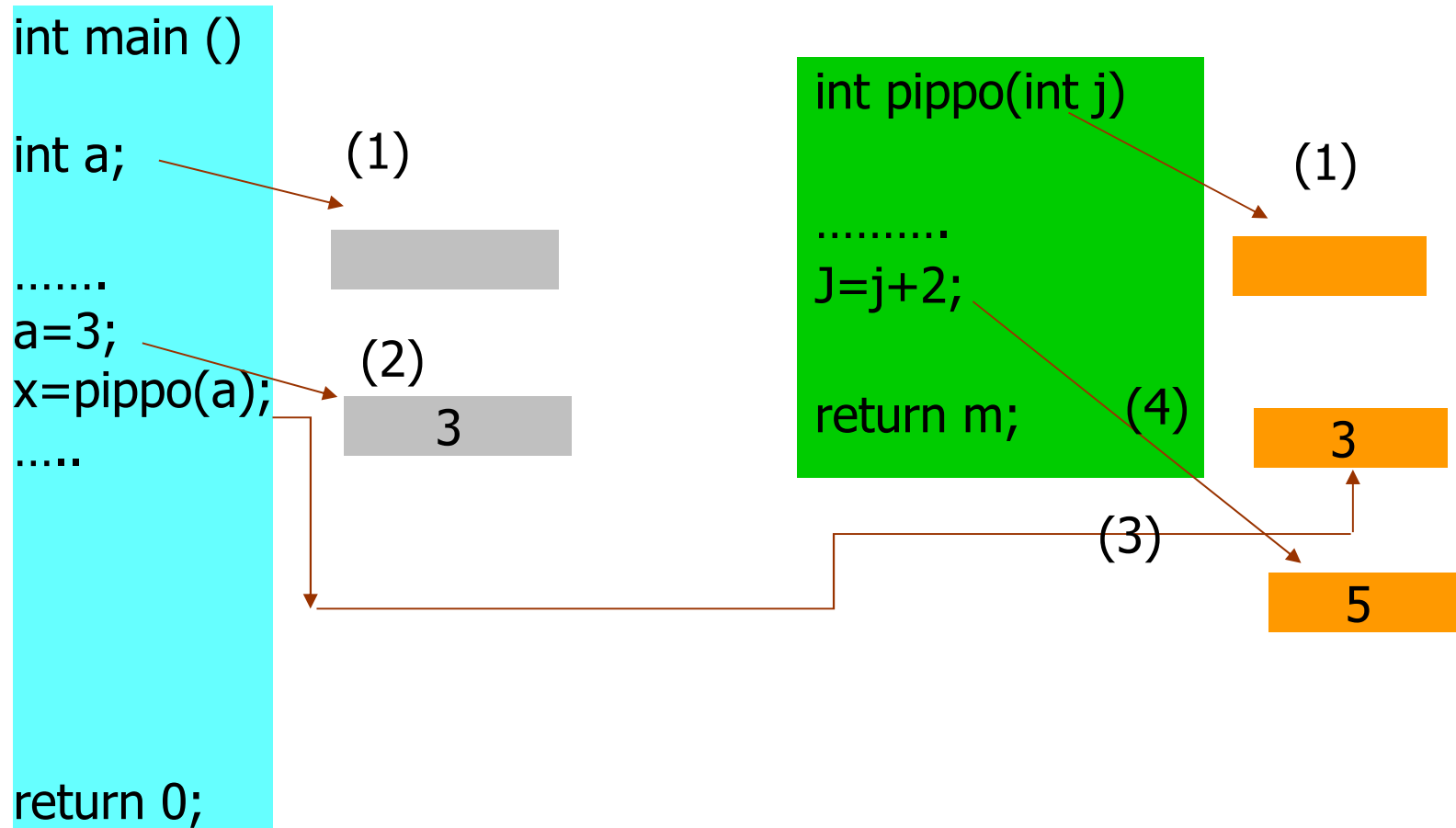
- Utilizzo:



Conseguenza

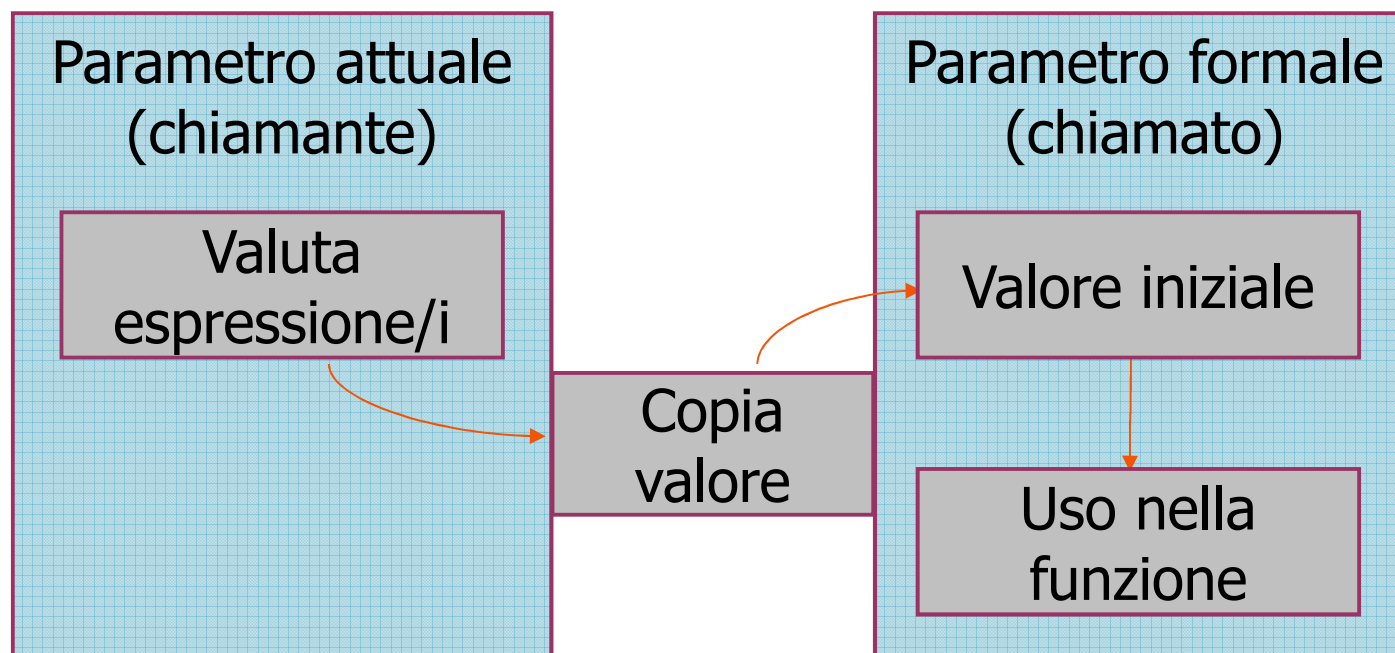
- La funzione chiamata non ha assolutamente modo di
 - Conoscere il nome delle variabili utilizzate come parametri attuali
 - Ne conosce solo **il valore** corrente
 - Modificare il valore delle variabili utilizzate come parametri attuali
 - Riceve solamente una **copia** del valore
- Questo meccanismo è detto passaggio "by value" dei parametri

Passaggio parametri by value



Passaggio dei parametri

- Ogni volta che viene chiamata una funzione, avviene il trasferimento del valore corrente dei parametri attuali ai parametri formali



Passaggio dei parametri

- In C, il passaggio dei parametri avviene *per valore*
 - Significato: Il valore dei parametri attuali viene copiato in variabili locali della funzione
- Implicazione:
 - I parametri attuali non vengono **MAI** modificati dalle istruzioni della funzione

Passaggio dei parametri: Esempio

```
#include <stdio.h>

void swap(int a, int b);

main() {
    int x,y;
    scanf("%d %d",&x,&y);
    printf("%d %d\n",x,y);
    swap(x,y);
    /* x e y NON VENGONO MODIFICATI */

    printf("%d %d\n",x,y);
}

void swap(int a, int b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
    return;
}
```

Passaggio dei parametri (Cont.)

- E' possibile modificare lo schema di passaggio per valore in modo che i parametri attuali vengano modificati dalle istruzioni della funzione
- Passaggio *per indirizzo (by reference)*
 - Parametri attuali = indirizzi di variabili
 - Parametri formali = puntatori al tipo corrispondente dei parametri attuali
 - Concetto:
 - Passando gli indirizzi dei parametri formali posso modificarne il valore
 - La teoria dei puntatori verrà ripresa in dettaglio più avanti
 - Per il momento è sufficiente sapere che:
 - '&'<**variabile**> fornisce l'indirizzo di memoria di <**variabile**>
 - '*'<**puntatore**> fornisce il dato contenuto nella variabile puntata da <**puntatore**>

Parametri di tipo vettoriale

- Il meccanismo di passaggio "by value" è chiaro nel caso di parametri di tipo scalare
- Nel caso di parametri di tipo array (vettore o matrice), il linguaggio C prevede che:
 - Un parametro di tipo array viene passato trasferendo una copia dell'indirizzo di memoria in cui si trova l'array specificato dal chiamante
 - Passaggio "by reference"

Conseguenza

- Nel passaggio di un vettore ad una funzione, il chiamato utilizzerà l'indirizzo a cui è memorizzato il vettore di partenza
- La funzione potrà quindi modificare il contenuto del vettore del chiamante

Passaggio dei parametri: Esempio

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

```
void swap(int *a, int *b);
```

```
main() {
```

```
    int x,y;
```

```
    scanf("%d %d",&x,&y);
```

```
    printf("%d %d\n",x,y);
```

```
    swap(&x,&y);
```

```
    /* x e y SONO ORA MODIFICATI */
```

```
    printf("%d %d\n",x,y);
```

```
}
```

```
void swap(int *a, int *b)
```

```
{
```

```
    int tmp;
```

```
    tmp = *a;
```

```
    *a = *b;
```

```
    *b = tmp;
```

```
    return;
```

```
}
```

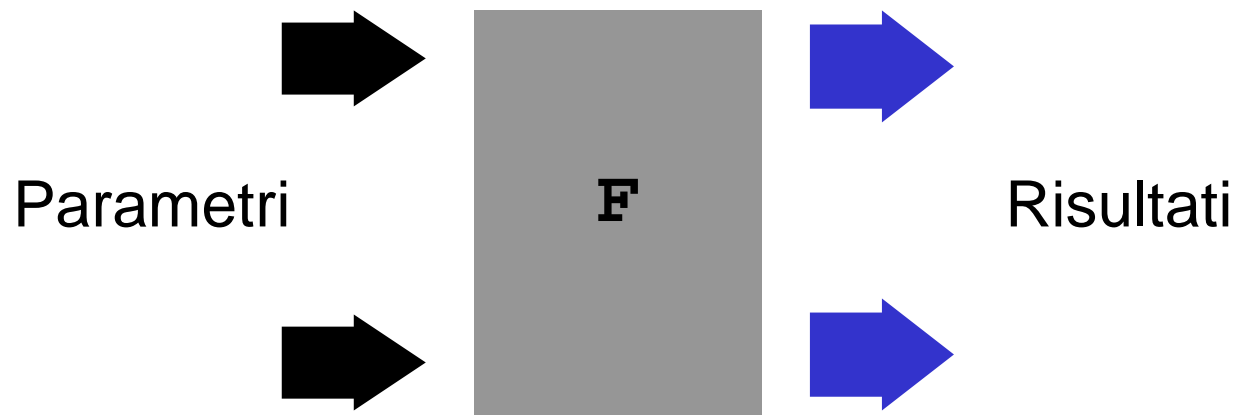
Passo l'indirizzo
di x e y

Vedremo più avanti il significato di * e &
Prima di una variabile

Uso *a e *b
come "interi"

Passaggio dei parametri (Cont.)

- Il passaggio dei parametri per *indirizzo* è indispensabile quando la funzione deve ritornare più di un risultato



Vettori e funzioni

- Le funzioni possono avere come parametri dei vettori
 - Parametri formali (prototipo)
 - Si indica il nome del vettore, con "[]" senza dimensione
 - Parametri attuali (chiamata)
 - Il nome del vettore SENZA "[]"
- Il nome del vettore indica l'indirizzo del primo elemento, quindi il vettore è passato per indirizzo!

Vettori e funzioni (Cont.)

- Conseguenza:
 - Gli elementi di un vettore passato come argomento vengono SEMPRE modificati!
- **ATTENZIONE:** Dato che il vettore è passato per indirizzo, la funzione che riceve il vettore come argomento **non ne conosce la lunghezza!!!!**
- Occorre quindi passare alla funzione anche la dimensione del vettore!

Dichiarazione o prototipo

```
int vettore(float a[], int dim) ;
```

Tipo del
valore di
ritorno

Nome della
funzione

Tipi e nomi
del vettore

Dimensio
ne del
vettore

Esercizio

- Scrivere una funzione `nonnull()` che ritorni il numero di elementi non nulli di un vettore di interi passato come parametro

- Soluzione:

```
int nonnull(int v[], int dim)
{
    int i, n=0;
    for (i=0; i<dim; i++) {
        if (v[i] != 0)
            n++;
    }
    return n;
}
```

← All'interno della funzione
bisogna sapere la dimensione
del vettore

← Se `v[]` fosse modificato dentro la
funzione, il valore sarebbe
modificato anche nella funzione
chiamante

Esercizio "Duplicati"

- Scrivere una funzione che, ricevendo due parametri
 - Un vettore di `double`
 - Un intero che indica l'occupazione effettiva di tale vettore

determini se vi siano valori duplicati in tale vettore

- La funzione ritornerà un intero
- pari a 1 nel caso in cui vi siano duplicati,
- pari a 0 nel caso in cui non ve ne siano

Soluzione (1/3)

```
int duplicati(double v[], int N) ;  
/*  
Riceve in ingresso il vettore v[] di double  
che contiene N elementi (da v[0] a v[N-1])  
  
Restituisce 0 se in v[] non vi sono duplicati  
Restituisce 1 se in v[] vi sono duplicati  
  
Il vettore v[] non viene modificato  
*/
```

Soluzione (2/3)

```
int duplicati(double v[], int N)
{
    int i, j ;

    for(i=0; i<N; i++)
    {
        for(j=i+1; j<N; j++)
        {
            if(v[i]==v[j])
                return 1 ;
        }
    }
    return 0 ;
}
```

Soluzione (3/3)

```
int main(void)
{
    const int MAX = 100 ;
    double dati[MAX] ;
    int Ndati ;
    int dupl ;

    ...
    dupl = duplicati(dati, Ndati) ;
    ...
}
```



Errore frequente

- Nel passaggio di un vettore occorre indicarne solo il nome

```
dupl = duplicati(dati, Ndati) ;
```

```
dupl = duplicati(dati[], Ndati) ;
```

```
dupl = duplicati(dati[MAX], Ndati) ;
```

```
dupl = duplicati(dati[Ndati], Ndati) ;
```

```
dupl = duplicati(&dati, Ndati) ;
```


Osservazione

- Nel caso dei vettori, il linguaggio C permette solamente il passaggio by reference
 - Ciò significa che il chiamato ha la possibilità di modificare il contenuto del vettore
- Non è detto che il chiamato effettivamente ne modifichi il contenuto
 - La funzione duplicati analizza il vettore senza modificarlo
 - Esplicitarlo sempre nei commenti di documentazione

Funzioni matematiche

- Utilizzabili includendo in testa al programma

```
#include <math.h>
```

- NOTA: Le funzioni trigonometriche (sia dirette sia inverse) operano su angoli espressi in radianti

$$\frac{\alpha^{(\circ)}}{\alpha^{\text{rad}}} = \frac{360^\circ}{2\pi}$$

$$\alpha^{\text{rad}} = \frac{2\pi}{360^\circ} \cdot \alpha^{(\circ)}$$

math.h

<i>funzione</i>	<i>definizione</i>
<code>double sin (double x)</code>	$\sin (x)$
<code>double cos (double x)</code>	$\cos (x)$
<code>double tan (double x)</code>	$\tan (x)$
<code>double asin (double x)</code>	$\operatorname{asin} (x)$
<code>double acos (double x)</code>	$\operatorname{acos} (x)$
<code>double atan (double x)</code>	$\operatorname{atan} (x)$
<code>double atan2 (double y, double x)</code>	$\operatorname{atan} (y / x)$
<code>double sinh (double x)</code>	$\sinh (x)$
<code>double cosh (double x)</code>	$\cosh (x)$
<code>double tanh (double x)</code>	$\tanh (x)$

math.h (Cont.)

funzione	<i>definizione</i>
<code>double pow (double x, double y)</code>	x^y
<code>double sqrt (double x)</code>	radice quadrata
<code>double log (double x)</code>	logaritmo naturale
<code>double log10 (double x)</code>	logaritmo decimale
<code>double exp (double x)</code>	e^x

$$\log_b x = \frac{\log_k x}{\log_k b}$$

math.h (Cont.)

<i>funzione</i>	<i>definizione</i>
<code>double ceil (double x)</code>	ceil (x)
<code>double floor (double x)</code>	floor (x)
<code>double fabs (double x)</code>	valore assoluto
<code>double fmod (double x, double y)</code>	modulo
<code>double modf (double x, double *ipart)</code>	restituisce la parte frazionaria di x e memorizza la parte intera di x in ipart

Funzioni matematiche: Esempio

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

double log2(double x);

main()
{
    int nogg, nbit;

    printf("Dammi il numero di oggetti: ");
    scanf("%d", &nogg);
    nbit=ceil(log2((double)nogg));
    printf("Per rappresentare %d oggetti servono %d
           bit\n", nogg, nbit);
}

double log2(double x)
{
    return log(x)/log((double)2);
}
```

Fine Capitolo

