



# Linguaggio C++

## Sottoprogrammi 1<sup>a</sup> parte

(fino al passaggio di parametri per valore)



# Metodologia Top-Down

- Una metodologia di risoluzione dei problemi è una guida o uno schema di riferimento cui adeguarsi nell'affrontare un problema.
- La più nota è probabilmente quella denominata TOP-DOWN (“dall’alto al basso”, “dal generale al particolare”), basata sull’idea di “dividere” i problemi in sottoproblemi, quando appaiono troppo difficili per essere affrontati direttamente.



# Metodologia Top-Down (2)

- La metodologia top-down è descrivibile in termini ricorsivi:

**Metodologia Top-Down** applicata ad un problema P

SE P è elementare (“sufficientemente” semplice)

ALLORA risolvo direttamente

ALTRIMENTI

1. Individua in P un certo numero di sottoproblemi
2. Risolvi con la **Metodologia Top-Down** ciascun sottoproblema
3. Costruisci la soluzione finale sfruttando le sotto-soluzioni

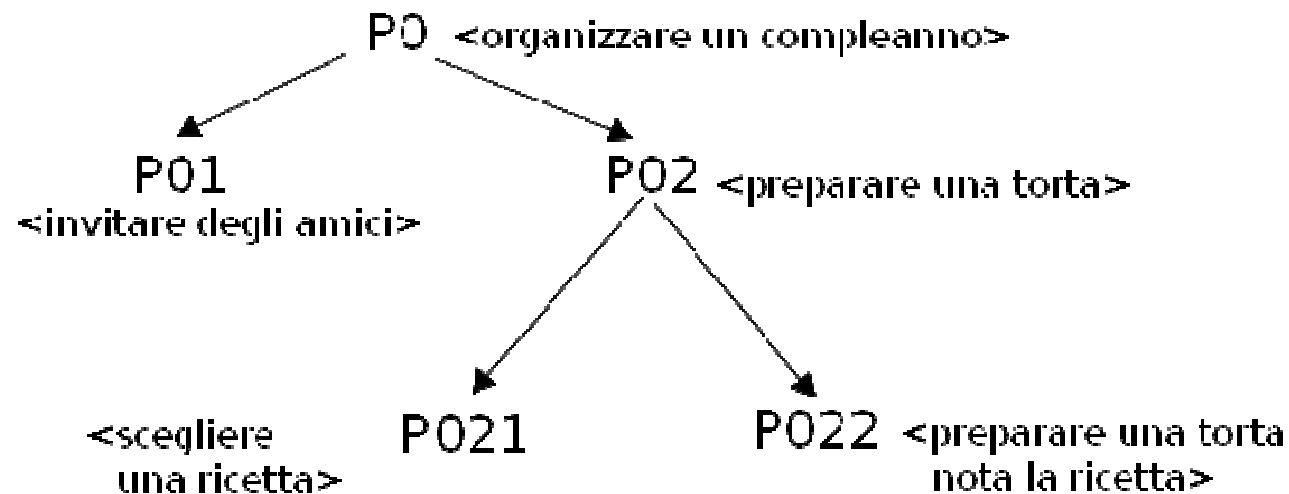


# Esempio

- La metodologia Top-Down viene implicitamente seguita in molte attività umane.
- Problema: organizzare un compleanno

# Dal problema ... ai sottoproblemi

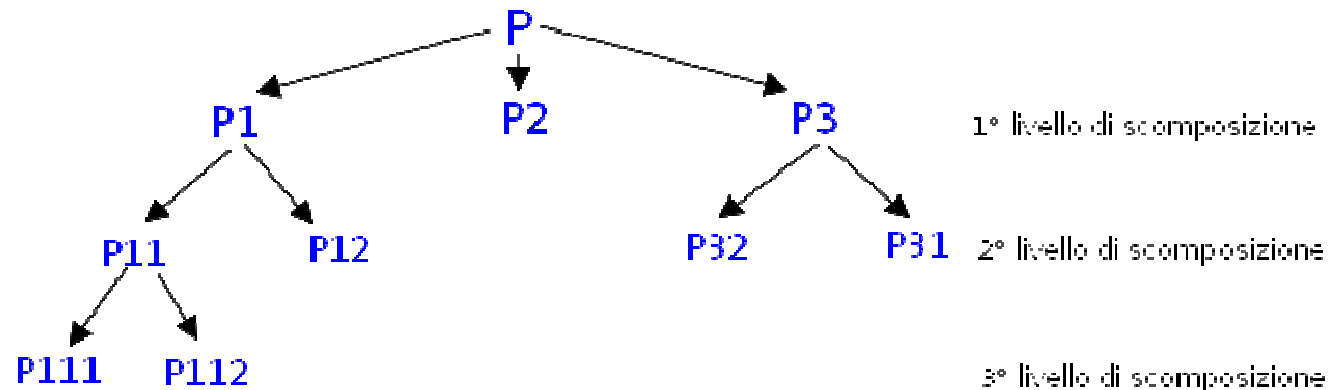
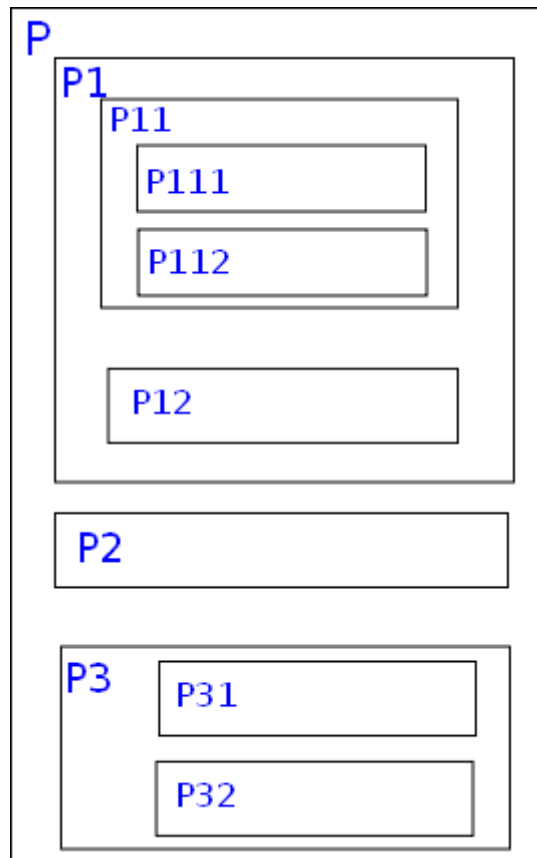
## Esempio



- Struttura gerarchica: albero, esso giustifica anche il nome Top-Down per questa metodologia.

Top-Down: dal generale al particolare, per raffinamenti successivi.

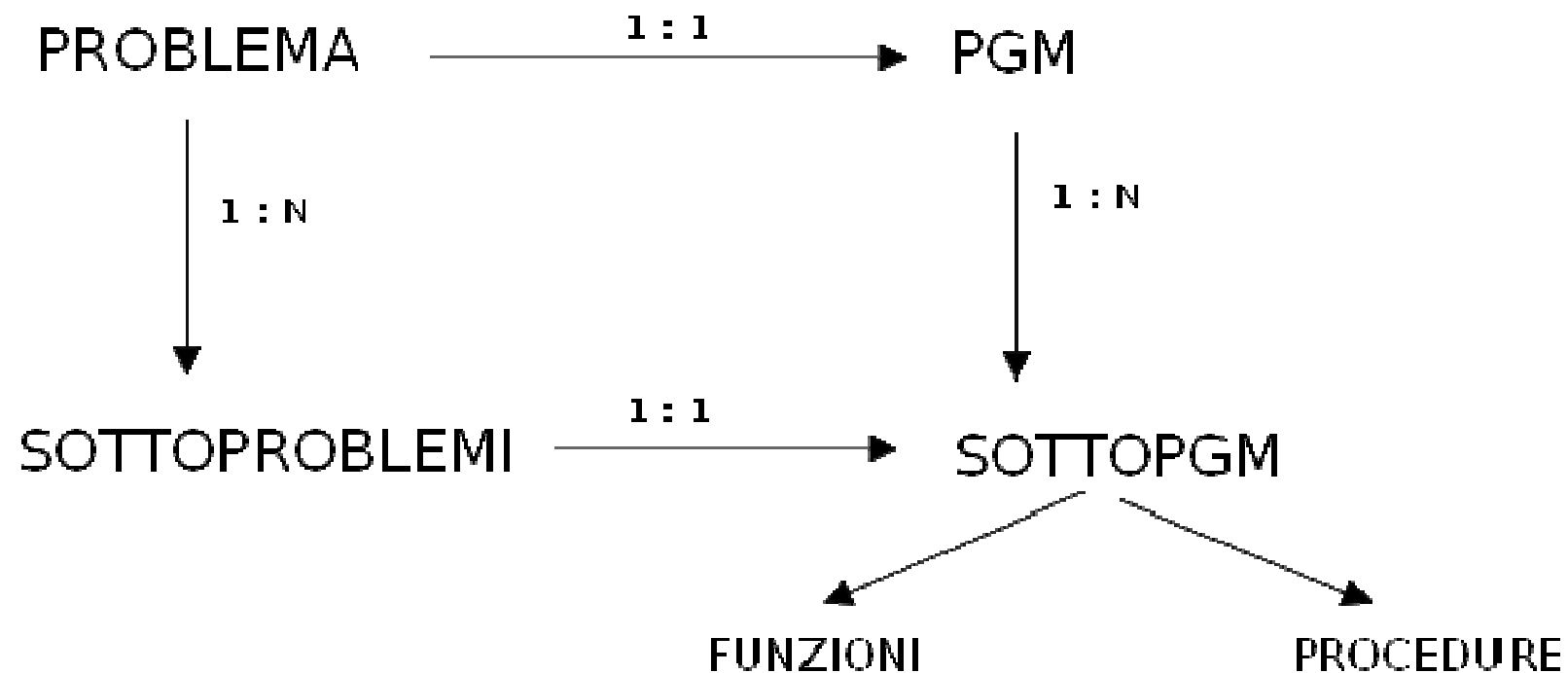
## Esempi grafici di scomposizione di un problema P in sottoproblemi secondo la metodologia Top-Down



Ad ogni livello di scomposizione, i sottoproblemi vengono considerati come **<black box>**, nel senso che si conosce che cosa devono fare, ma non come lo fanno.

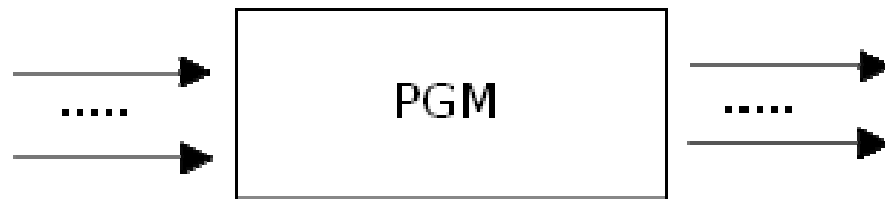


# Dal programma ... ai sottoprogrammi





# Dal programma ... ai sottoprogrammi

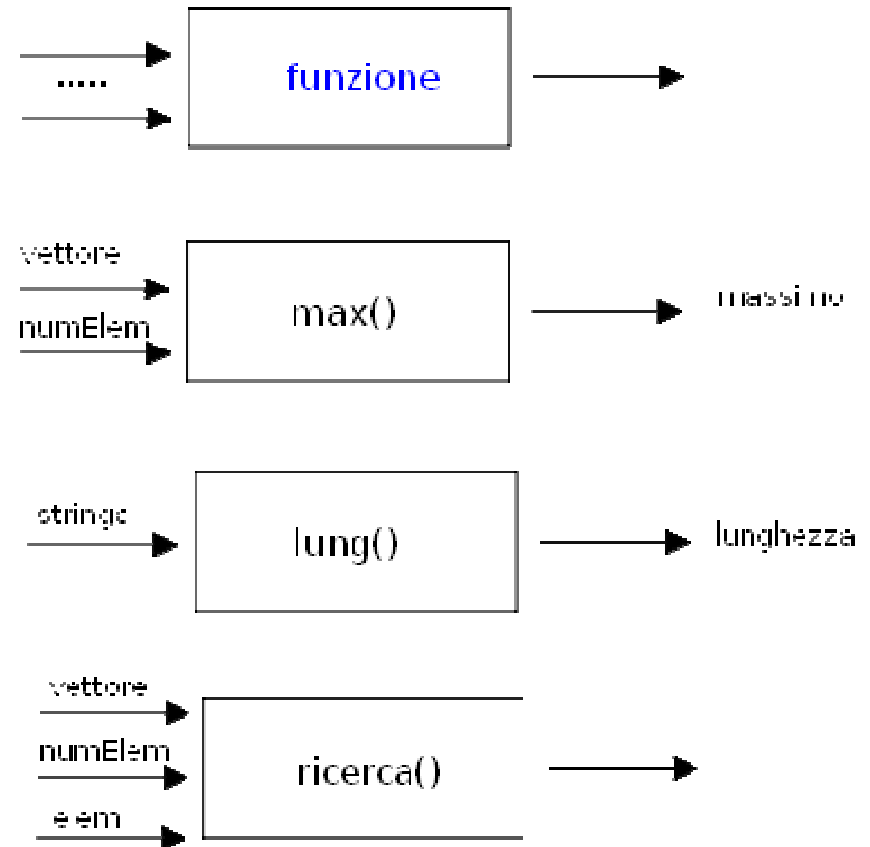


- ☐ Ogni sottopgm deve fare una ed una sola cosa, deve cioè svolgere un'unica funzione logica.
- ☐ Sottopgm  $\leftrightarrow$  "Black box"



# Le funzioni

- Le **funzioni** sono sottopgm che, applicate ai dati senza apportarvi modifiche, producono un unico risultato che ritornano al chiamante.



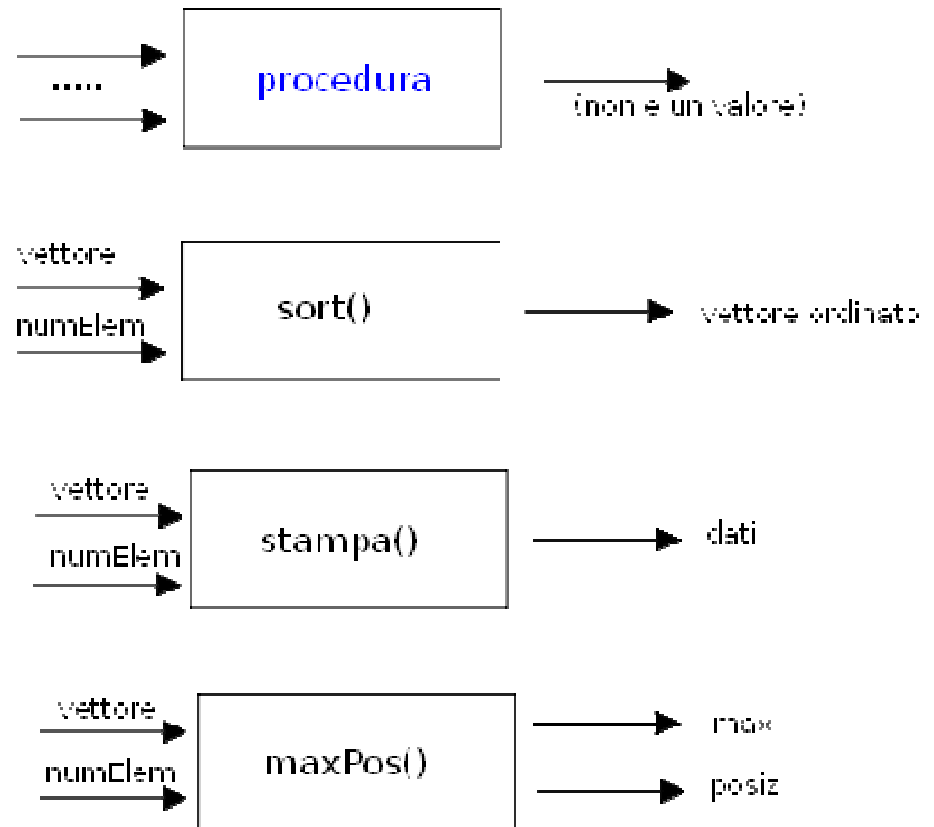


# Interfaccia di una funzione

- L'interfaccia (o prototipo) di una funzione comprende
  - Nome della funzione
  - Lista dei parametri
  - Tipo del valore da essa restituito

# Le procedure

- Le **procedure** sono sottopgm che rappresentano un blocco di codice (un insieme di istruzioni).
- Una procedura esegue un comando.
- Le procedure sono simili alle funzioni, ma non hanno il compito di calcolare un risultato e ritornarlo come un valore. Lo scopo di una procedura non è quindi di produrre un valore, ma di modificare lo stato, cioè il contenuto della memoria di un programma.





# Tipi di sottoprogrammi

## ■ Procedura:

E' un'astrazione della nozione di ***istruzione***. E' un'istruzione non primitiva attivabile in un qualunque punto del programma in cui può comparire un'istruzione.

## ■ Funzione:

E' l'astrazione del concetto di ***operatore***. Si può attivare durante la valutazione di una qualunque espressione e **restituisce un valore**.



# Sottopgm in C/C++

- In C/C++ tutti i sottopgm si chiamano funzioni e in generale ritornano almeno un valore (per default un intero).
- Un pgm C/C++ non è altro che una raccolta di una o più funzioni.

$\text{PGM C/C++} = \{F_0, F_1, F_2, \dots, F_n\}$

dove:

- $F_0$  = funzione principale: `main()`
- $F_i$  = funzioni di libreria (standard) + funzioni utente
- Libreria = raccolta di sottopgm in forma compilata.



# Esecuzione di un pgm C/C++

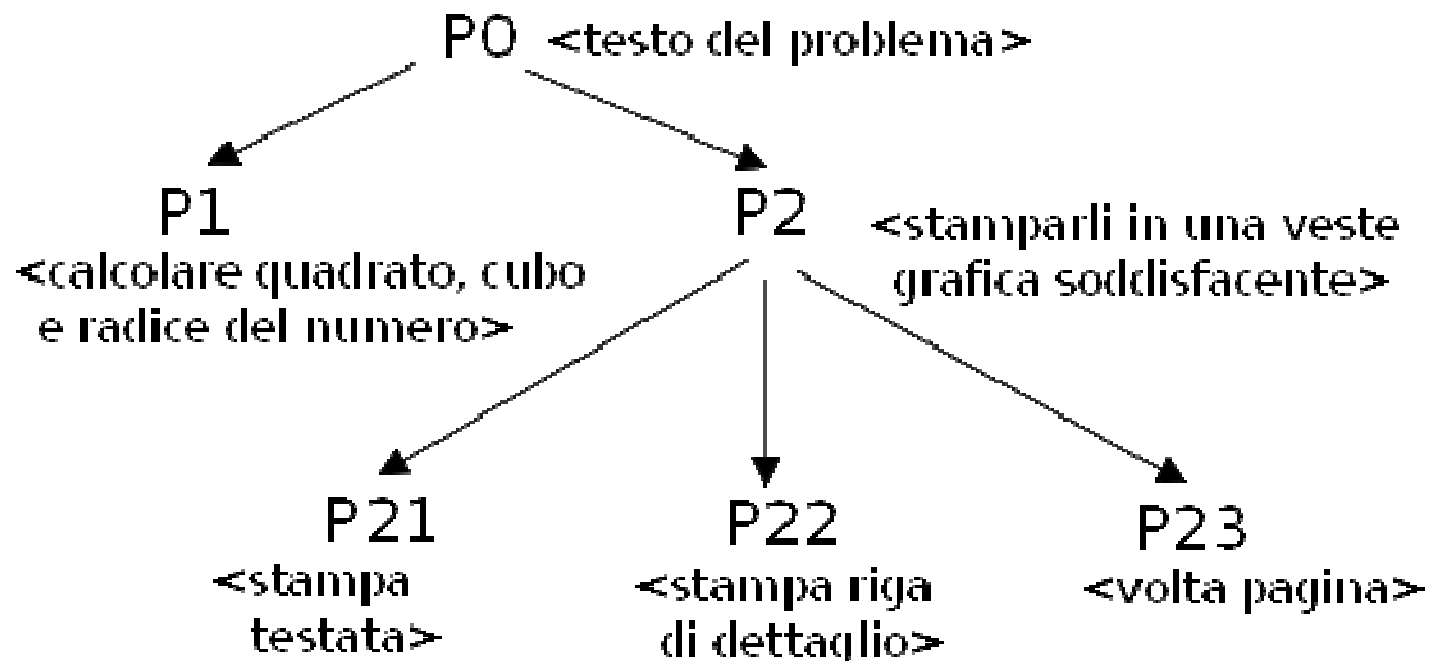
- Tutti i pgm C/C++ devono contenere una funzione `main()`, che segnala l'inizio dell'esecuzione.
- L'esecuzione del pgm termina alla fine della funzione `main()`.



# Problema

- Per i numeri da 1 a 100 stampare una tavola che contenga in ogni riga il numero, il quadrato, il cubo e la radice dello stesso.
- Dare una veste grafica soddisfacente alle singole pagine stampate.

# Scomposizione top-down del problema







# Costruzione del main pgm (in pseudocodice)

PER  $i \leftarrow 1$  A 100 ESEGUI

INIZIO

SE sei all'inizio pagina

ALLORA stampa la testata

calcola il valore dei dati richiesti

stampa la riga di dettaglio

SE hai scritto 20 righe

ALLORA volta pagina

FINE



# NOTA

- Costrutto iterativo determinato (o costrutto iterativo enumerativo)

PER  $i \leftarrow 1$  A 100 ESEGUI

INIZIO

.....

FINE



# NOTA

## ■ Ciclo

- ☐ Indeterminato

- ☐ Determinato

PER <indice> ← <inizio> [INDIETRO] A <fine> [PASSO <n>] ESEGUI  
INIZIO

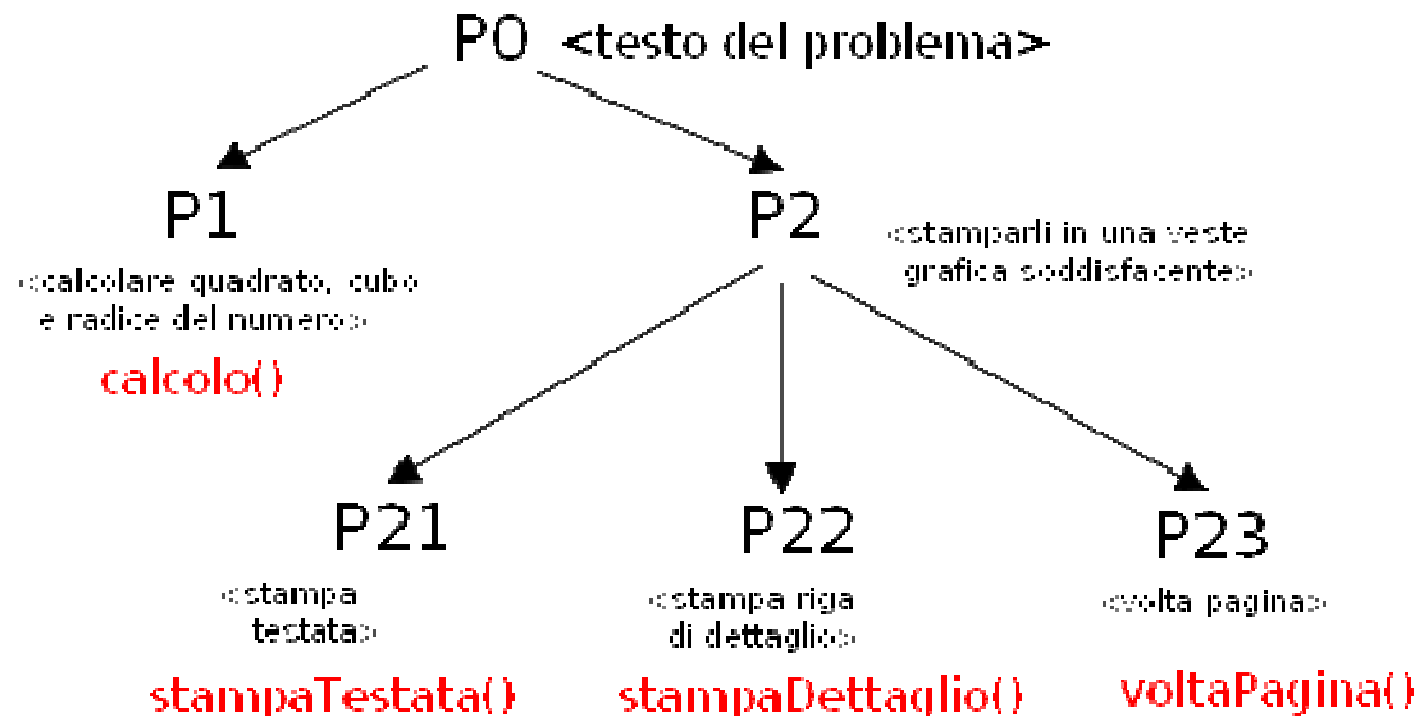
.....  
FINE

In assenza del passo non è necessario incrementare o decrementare la variabile indice in quanto ciò avviene automaticamente.

Vedi sul libro di testo il flow-chart del ciclo determinato.



# Dai sottoproblemi ... ai sottoprogrammi





- Ora, avendo individuato i sottopgm, riscriviamo il main pgm (in pseudocodice).
- N.B. I sottogm, a questo punto dell'analisi, vengono visti come “black box”.



# Main pgm (in pseudocodice)

PER  $i \leftarrow 1$  A 100 ESEGUI

INIZIO

SE sei all'inizio pagina

ALLORA **stampaTestata()**

**calcolo()**

**stampaDettaglio()**

SE hai scritto 20 righe

ALLORA **voltaPagina()**

FINE



- Ora passiamo dalla fase di **analisi** ... alla fase di **codifica**.
- Analisi: Problema → Algoritmo (pseudocodice, metodologia top-down)
- Codifica: Algoritmo → Programma (linguaggio di programmazione, sottopgm)



# Pgm in C++

```
#define NUM_MAX 100  
#define RIGA_MAX 20
```

```
void stampaTestata();  
void calcolo();  
void stampaDettaglio();  
void voltaPagina();
```

```
int riga;  
int inizio;  
long num;  
long quadrato;  
long cubo;  
float radice;
```

```
int main(){  
    riga=0;  
    inizio=1;  
  
    for(num=1; num <= NUM_MAX; num++){  
        if (inizio)  
            stampaTestata();  
        calcolo();  
        stampaDettaglio();  
        riga++;  
        if (riga== RIGA_MAX) {  
            cin.get();  
            voltaPagina();  
        }  
    }  
    cin.get();  
    return 0;  
}
```



```

void stampaTestata() {
    cout<<"TAVOLA NUMERICA\n";
    cout<<"-----\n";
    cout<<"| Num |  Quad |  Cubo |  Rad  |\n";
    cout<<"-----\n";
    inizio=0;
}

void calcolo(){
    quadrato=num*num;
    cubo=quadrato*num;
    radice=sqrt(num);
}

void stampaDettaglio(){
    cout<<"| "<<num<<" | "<<quadrato<<" | "<<cubo<<" | "<<radice<<" |"<<endl;
}

void voltaPagina(){
    inizio=1;
    riga=0;
}

```



# Osservazioni

- Variabili (costanti) globali e locali
- Concetto di prototipo (= in C/C++ è la dichiarazione di una funzione)
- Sintassi per definire un sottoprogramma in C/C++
- Tipo di dato: void (nessun valore)



# Concetto di funzione in informatica

- In informatica, una funzione è un sottopgm che:
  - riceve dal pgm chiamante una serie di valori (*solo in lettura*) o argomenti, specificati nei parametri attuali;
  - esegue le istruzioni contenute nel corpo della funzione;
  - restituisce al chiamante un valore (valore della funzione).



# Esempio di funzione di libreria

- sqrt(): calcola la radice quadrata
- Dominio: numeri reali positivi o nulli
- Codominio: numeri reali positivi o nulli
- $f(x) = \text{sqrt}(x)$
- In C/C++: `radice=sqrt(x);` // x: argomento o parametro della funzione  
//  $x \geq 0$



# Struttura generale di una funzione in C/C++

```
tipoValoreRestituito nomeFunzione ( elenco parametri ) {  
    //dichiarazioni/definizioni di costanti, tipi, variabili (locali alla funzione)  
  
    //sezione istruzioni  
  
    return valore;  
}
```

dove: elenco parametri = tipoPar1 par1, ..., tipoParN parN



# Funzione cubo()

```
#include <iostream>
```

```
double cubo (double num);    //prototipo: dichiarazione della funzione cubo()
```

```
int main(){  
double a, b;
```

```
    cout<<"Inserisci un numero \n";
```

```
    cin>>a;
```

```
    b = cubo(a);                //chiamata della funzione, argomenti della funzione, parametri attuali
```

```
    cout<<a<<" elevato al cubo = "<< b<<endl;
```

```
    return 0;
```

```
}
```

```
//definizione della funzione
```

```
double cubo (double num){    //intestazione della funzione, parametri formali
```

```
double risultato;
```

```
    risultato = num * num * num;
```

```
    return risultato;        //ritorno del valore
```

```
}
```

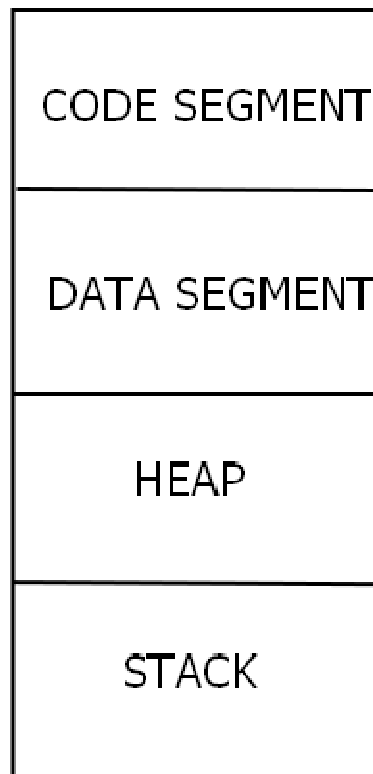


# NOTA

I parametri costituiscono i “canali”  
attraverso i quali i sottopgm si  
scambiano i dati (colloquiano).



# Aree di memoria



- Area del codice: contiene il codice eseguibile del pgm.
- Area dati globali: contiene le variabili globali e statiche (static) e costanti
- Area stack: contiene i record di attivazione delle funzioni (parametri e variabili locali)
- Area heap: disponibile per allocazioni dinamiche





# STACK

- L'area stack contiene i record di attivazione delle funzioni.
- Un record di attivazione rappresenta l'ambiente (area di memoria (lavoro)) della funzione e contiene
  - I parametri ricevuti
  - Le variabili locali
  - L'indirizzo (del chiamante) a cui tornare alla fine
  - Un riferimento al record di attivazione del chiamante



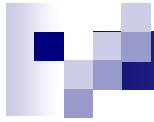
# STACK (2)

- Ad ogni attivazione di funzione viene creato un nuovo record di attivazione specifico per quella chiamata.
- Per catturare la semantica delle chiamate annidate (una funzione che chiama un'altra funzione che ...), quest'area di memoria è gestita come una pila (stack):  
Last In, First Out → LIFO



# STACK (3)

- La dimensione del record di attivazione:
  - varia da una funzione all'altra
  - ma, per una data funzione, è fissa e calcolabile a priori.
- Il record di attivazione:
  - viene creato automaticamente nel momento in cui la funzione viene chiamata
  - rimane sullo stack per tutto il tempo in cui la funzione è in esecuzione
  - viene deallocato automaticamente quando la funzione termina.



# STACK (4)

- Funzioni che chiamano altre funzioni danno luogo a una sequenza di record di attivazione
  - allocati secondo l'ordine delle chiamate
  - deallocati in ordine inverso



# STACK (5)

- Quando la funzione termina, il controllo torna al chiamante, che deve:
  - riprendere la sua esecuzione dall'istruzione successiva alla chiamata della funzione
  - trovare il suo “mondo” inalterato
- A questo scopo, quando il chiamante chiama la funzione, si inseriscono nel record di attivazione della funzione anche:
  - l'indirizzo di ritorno (return address), ossia l'indirizzo della prossima istruzione del chiamante che andrà eseguita quando la funzione terminerà
  - il link dinamico (dynamic link), ossia un collegamento al record di attivazione del chiamante, in modo da poter ripristinare l'ambiente del chiamante quando la funzione terminerà.

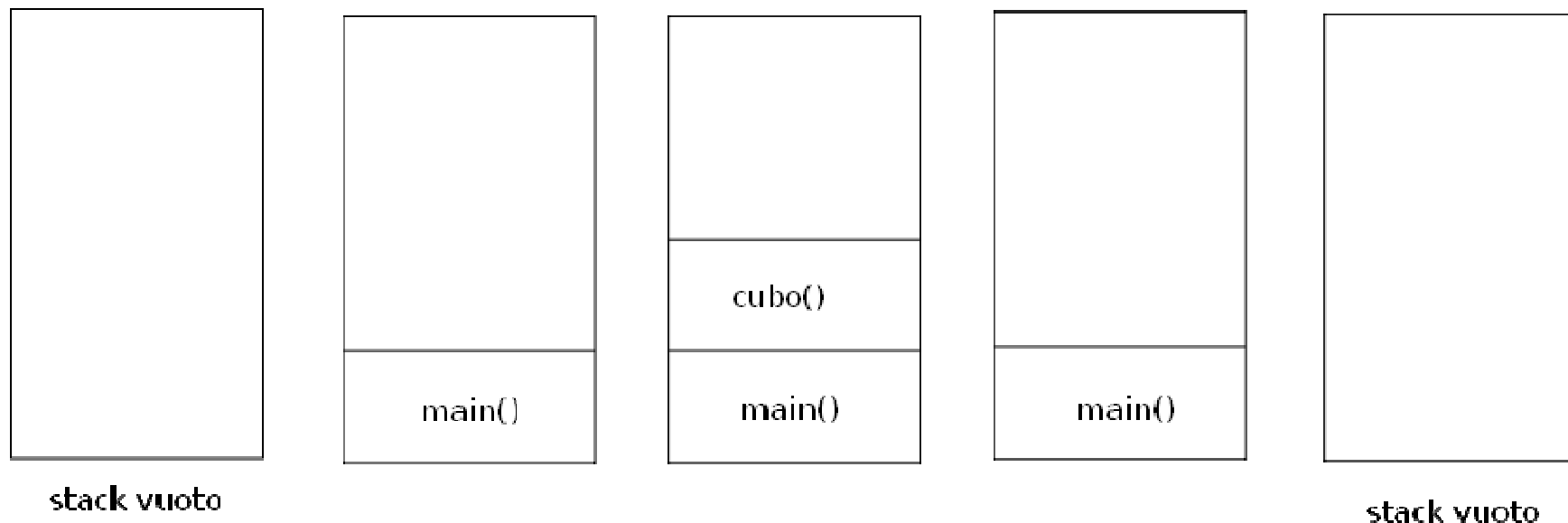


# STACK (6)

- Per le funzioni, spesso il record di attivazione prevede una ulteriore cella, destinata a contenere il risultato della funzione.
- Tale risultato viene copiato dal chiamante prima di deallocare il record della funzione appena terminata.
- Altre volte, il risultato viene restituito dalla funzione al chiamante lasciandolo in un registro della CPU.

# Evoluzione dello stack

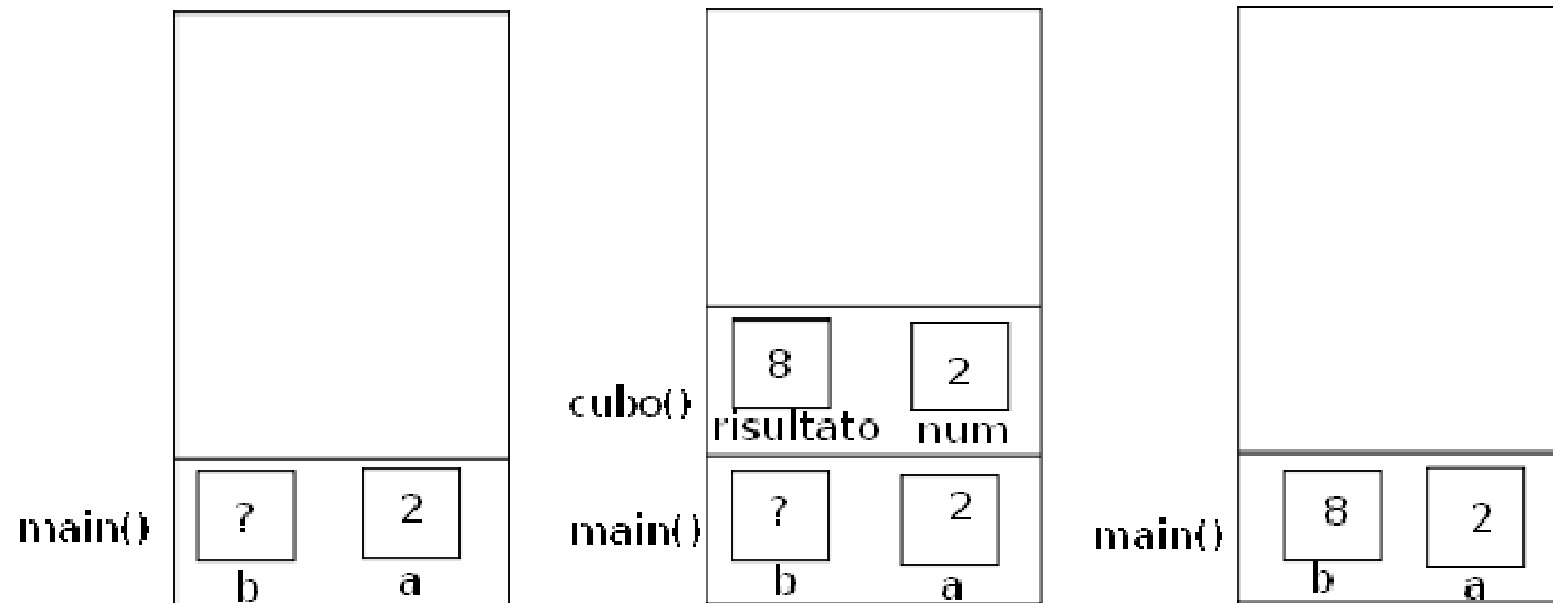
(visto come una black box, relativamente al pgm che usa la funzione cubo())





# Evoluzione dello stack (2)

(relativamente al pgm che usa la funzione cubo())







# Regola di esistenza delle variabili locali

- Le variabili dell'ambiente locale vengono create (allocate) automaticamente all'atto della chiamata e vengono distrutte (deallocate) automaticamente all'atto del ritorno al chiamante.



# NOTA

- Quando l'esecutore incontra una chiamata a sottopgm svolge nell'ordine le seguenti azioni:
  - Sospende l'esecuzione del pgm o del sottopgm che ha effettuato la chiamata (chiamante) conservando tutte le informazioni necessarie per riprendere l'esecuzione;
  - Effettua il passaggio di parametri;
  - Esegue il corpo del sottopgm;
  - Ritorna al chiamante riprendendo l'esecuzione dal punto successivo alla chiamata.



# Parametri

- Programma e sottopgm (o sottopgm fra loro) colloquiano attraverso il passaggio di parametri.
- Questo meccanismo di comunicazione prevede che tra i parametri specificati di volta in volta in ciascuna chiamata (parametri attuali) e quelli definiti nell'intestazione del sottopgm (parametri formali) ci sia una precisa corrispondenza:
  - nel numero
  - nel tipo
  - nell'ordineperché il compilatore possa interpretare correttamente il testo del pgm



# Prototipo (di una funzione)

- Il prototipo di una funzione è una dichiarazione che descrive
  - il nome della funzione
  - i tipi dei parametri
  - il tipo del valore restituito
- Il prototipo serve al compilatore per effettuare un controllo tra parametri attuali e parametri formali.



# Osservazione

- In C/C++ tutti i sottopg sono allo stesso livello (globali), non posso definire cioè un sottopgm dentro un altro.



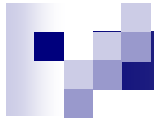
# Passaggio di parametri per valore

- In C++ esistono due modalità di passaggio di parametri
  - per valore
  - per riferimento
- Nel passaggio di parametri per valore viene fatta una **copia**: il contenuto del parametro attuale viene copiato nel corrispondente parametro formale.



## Passaggio di parametri per valore (2)

- Nel passaggio di parametri per valore, il valore contenuto nel parametro attuale del chiamante NON viene modificato dalle eventuali operazioni compiute sul relativo parametro formale (che è una variabile locale del sottopgm) poiché il parametro formale e quello attuale fanno riferimento a due aree di memoria diverse.
- Nel passaggio per valore i parametri sono solo variabili di input.



# Il tipo void

## ■ Si usa per:

- ☐ Le funzioni che non ritornano valori (sono procedure!)
- ☐ Le funzioni che non hanno parametri.
- ☐ Un puntatore di tipo generico.





# Funzione main() e tipo void

<pre>main(void){     .....     return 0; }</pre>	<p>← Definizione della funzione main().</p> <p>In C/C++ tutte le funzioni restituiscono per default un intero.</p>
<pre><b>int main(){</b>     .....     <b>return 0;</b> }</pre>	<pre>void main(void){     ..... }</pre>



# Esempio di uso di void

```
void saluto(){  
    cout<<"Salve! \n";  
}
```

← E' una procedura!



# L'istruzione di return

- Consente di restituire un valore alla funzione chiamante.
- Permette di terminare anticipatamente una funzione restituendo il controllo al codice di chiamata.



# Esempi

```
int max (int a, int b){  
    int temp;  
    if(a > b)  
        temp = a;  
    else  
        temp = b;  
    return temp;  
}
```

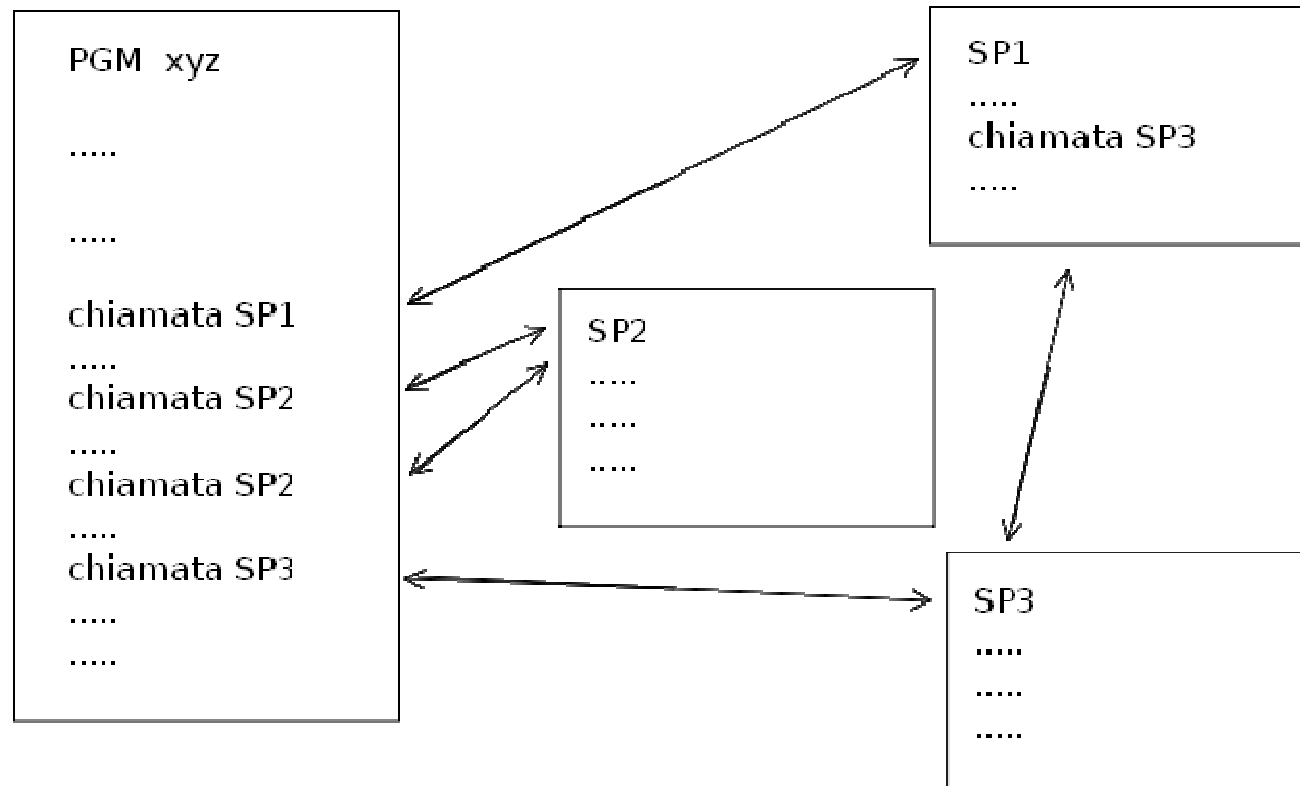
```
int max (int a, int b){  
    if(a > b)  
        return a;  
    else  
        return b;  
}
```



# Nota

- Una funzione C/C++ può contenere più istruzioni di return: accorgimento a cui si ricorre spesso anche per semplificare l'algoritmo (vedremo che in alcuni casi si può evitare l'uso di variabili flag).

# Vediamo ora una situazione più complessa



Evoluzione dello stack (alla lavagna).



# Esercizio

- Scrivi un pgm C++ che utilizza una funzione per verificare se un anno è bisestile.
- Un anno è bisestile se è divisibile per 4, oppure, nel caso in cui si tratti di anni secolari, se è divisibile per 400.
- Ad esempio il 2000 è stato bisestile, mentre il 2100 non lo sarà.



# Esempio: funzione menu()

```
int main(){
int scelta;

.....

    scelta = menu();
    switch(scelta){
        .....
    }
    .....
}
```

```
int menu(){
int scelta;

//stampa menù
cout<<"Digita la tua scelta\n";
cin>>scelta;


return scelta;
}
```





# Necessità del passaggio di parametri

- Si osservi la seguente istruzione:
  - `radice = sqrt(num);`
- Se non si usa il passaggio di parametri (quindi si usano solo variabili globali) come si può scrivere una libreria di funzioni?
- Risposta ...
- Ogni sottopgm deve essere indipendente dagli altri, cioè si deve fare in modo che ciascun sottopgm contenga tutto e solo quello che gli serve.
- In generale è meglio definire una variabile nel blocco/sottopgm che la utilizza.



## Necessità del passaggio di parametri (2)

- Abbiamo visto che in C/C++ una funzione ritorna al più un valore.
- Come ci si deve comportare quando il sottopgm deve ritornare più di un valore (è una procedura)?
- Esistono due soluzioni:
  - usare le variabili globali (sconsigliato!)
  - passaggio dei parametri per riferimento (by reference)

# Metodologie di progettazione

