ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

DICHIARAZIONI, FUNZIONI E PASSAGGIO DEI PARAMETRI

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI (CAP 4 E 5 DI SAVITCH)

1. dichiarazioni e definizioni
2. portata di una dichiarazione
3. funzioni
4. passaggio dei parametri
5. namespaces e la direttiva `using`

ESERCIZIO: cercate i vari argomenti nei capitoli 4 e 5 di Savitch

PORTATA DI UNA DICHIARAZIONE (SCOPE)

la portata di una dichiarazione è la parte di programma in cui una dichiarazione di un identificatore è valida

tra i comandi c' è il blocco: { *declaration-part statement* }

che di solito si usa per *raggruppare comandi*

osservazione: in un blocco è possibile dichiarare nuovi identificatori, *o ridichiararne di vecchi* !

non è mai possibile dichiarare due volte lo stesso identificatore

esempio: cosa stampano i programmi seguenti?

```
int x ;  x = 1 ;
{
   int y ; y = 2 ;
   cout << x << y ;
}
```

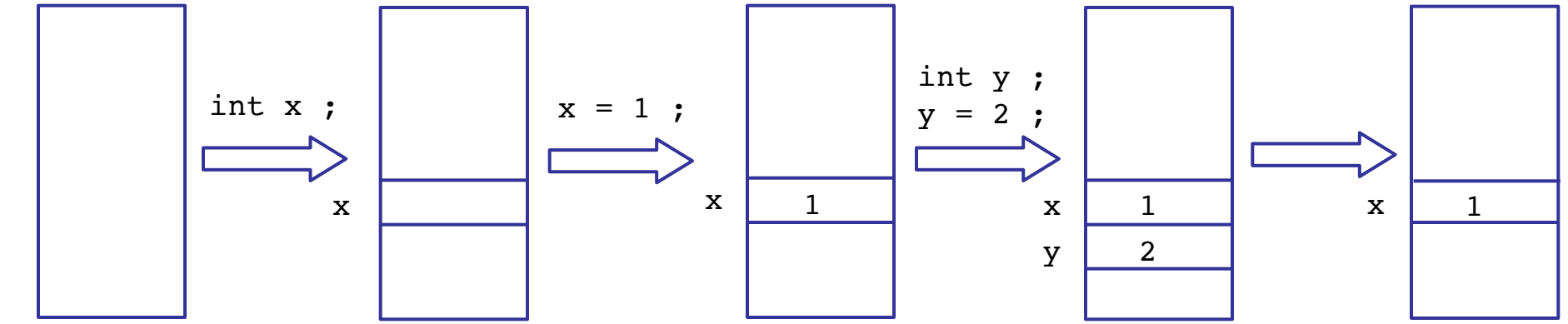
```
int x ;  x = 1 ;
{
   int y ; y = 2 ;
}
cout << x << y ;
```

l'esempio di destra è erroneo…

EVOLUZIONE DELLA MEMORIA

evoluzione della memoria per il programma

```
int x ;  x = 1 ;
          {  int y ; y = 2 ;  }
cout << x << y ;
```



memoria iniziale (vuota)

memoria dopo la dichiarazione di x

memoria dopo la dichiarazione di y

memoria dopo il blocco

quindi "`cout << x << y ;`" verrebbe eseguito senza che `y` sia dichiarata

il compilatore riconosce questo tipo di errori!

PORTATA DI UNA DICHIARAZIONE: LE REGOLE

1. **la portata di una dichiarazione di un identificatore è il blocco in cui occorre la dichiarazione**, *dal momento in cui occorre*, e tutti i blocchi interni a tale blocco, con l'eccezione di 2.
2. se un identificatore dichiarato nel blocco A, è **ridefinito** nel blocco B, interno ad A, allora **B ed ogni blocco interno a B non fanno parte della portata della dichiarazione di A**

esempio: la stampa dei programmi seguenti:

```
int x ;  x = 1 ;
int y ;  y = 2 ;
{
        int y ; y = 3 ;
        cout << x << y ;
}
```

B

A

```
int x ;  x = 1 ;
int y ;  y = 2 ;
{
        int y ; y = 3 ;
}
cout << x << y ;
```

B

A

identificatore ridefinito!

PORTATA DI UNA DICHIARAZIONE: LE REGOLE/CONT.

commenti:

- la portata di una dichiarazione di identificatori è dal punto in cui occorre alla fine del blocco corrispondente

- all'interno dello stesso blocco non è possibile definire due volte lo stesso identificatore

```
{    int x ;
     int x ;
}
```

è sbagliato!

- gli identificatori introdotti con **`const`** (le costanti) seguono le regole di portata degli altri identificatori

LE FUNZIONI E LO SVILUPPO TOP-DOWN

problema: *introdurre un meccanismo che consente di riutilizzare codice già esistente*

soluzione: *le funzioni (o sottoprogrammi)*

definizione: una funzione è un blocco (dichiarazioni + comandi) a cui è associato un nome

esempi: `main` è una funzione

```
void radice_quadrata(){ int m, i ;
                cin >> m ;
                i = 0 ;
                while (i*i <= m) i= i+1 ;
                cout << i-1 ;
}
```

osservazione: *le funzioni possono avere degli argomenti e restituire un risultato*

le funzioni matematiche di **`cmath`**: **`sqrt`**, **`log`**, **`abs`**

LE FUNZIONI: BENEFICI

1. **evitare ripetizioni dello stesso codice**: se `radice_quadrata` devo calcolarlo in molti punti del programma, posso incapsulare il calcolo in una funzione e poi richiamarla
    a. la lunghezza del codice diminuisce
    b. il codice è più leggibile (meglio vedere scritto radice_quadrata, piuttosto che la sequenza di comandi nel suo blocco
    c. si evitano errori: riscrivendo il codice si rischia di sbagliare
2. **il codice è decomposto in moduli disgiunti**:
    a. facilita la comprensione del programma (astrazione dai dettagli, vedi le funzioni di libreria)
    b. facilita l'implementazione del codice: funzioni differenti possono essere implementate da persone differenti
    c. facilita le modifiche (upgrades) poichè si localizza il codice da modificare

FUNZIONI: ASPETTI SINTATTICI

bisogna distinguere **due momenti**:

la definizione della funzione

il suo uso

questi due momenti sono differenziati da C++ da *due categorie sintattiche differenti* :

function-definition *(definizione)*

function-call e *void-function-call* *(invocazione)*

in altri termini:

nei due momenti si utilizzano costrutti linguistici differenti

FUNZIONI/ASPETTI SINTATTICI

function-definition ***::=*** ***type*** ***identifier*** **(** ***formal-parameters*** **) {**

declaration-part ***statement-part***

}

::= si legge "può essere"

// la prima linea è detta ***intestazione della funzione***, la parte restante

// è detta ***corpo della funzione***)

// *type* indica il tipo di ritorno – eventualmente *type* = void

function-call ***::=*** ***identifier*** **(** ***actual-parameters*** **)**

// le *function-call* ***sono le funzioni che ritornano valori*** e che devono essere

// usate *all' interno di espressioni*

void-function-call ***::=*** ***identifier*** **(** ***actual-parameters*** **) ;**

// le *void-function-call* **sono le funzioni che non ritornano niente** e che

// si usano *come comandi* -- notare il ";" alla fine di una *void-function-call*

FUNZIONI/DEFINIZIONI

definizione delle funzioni

- le funzioni sono definite allo stesso livello della funzione **`main`** (non è possibile definire una funzione all'interno di **`main`**, o all'interno di un'altra funzione)
- la portata della definizione della funzione è dal momento in cui si trova in tutto il resto del programma

esempio:

```
int m, n;
void radice_quadrata(){ cin >> m ; n = 0 ;
      while (n*n <= m) n= n+1 ;
      cout << n-1 ; }
void min(){ int tmp ; cin >> m >> n ;
      if (m > n) { tmp = m ; m = n ; n = tmp ; } }
int main() { min(); radice_quadrata();return(0);}
```

DOVE SI DICHIARANO LE FUNZIONI?

le funzioni vanno dichiarate allo stesso livello del **`main`**

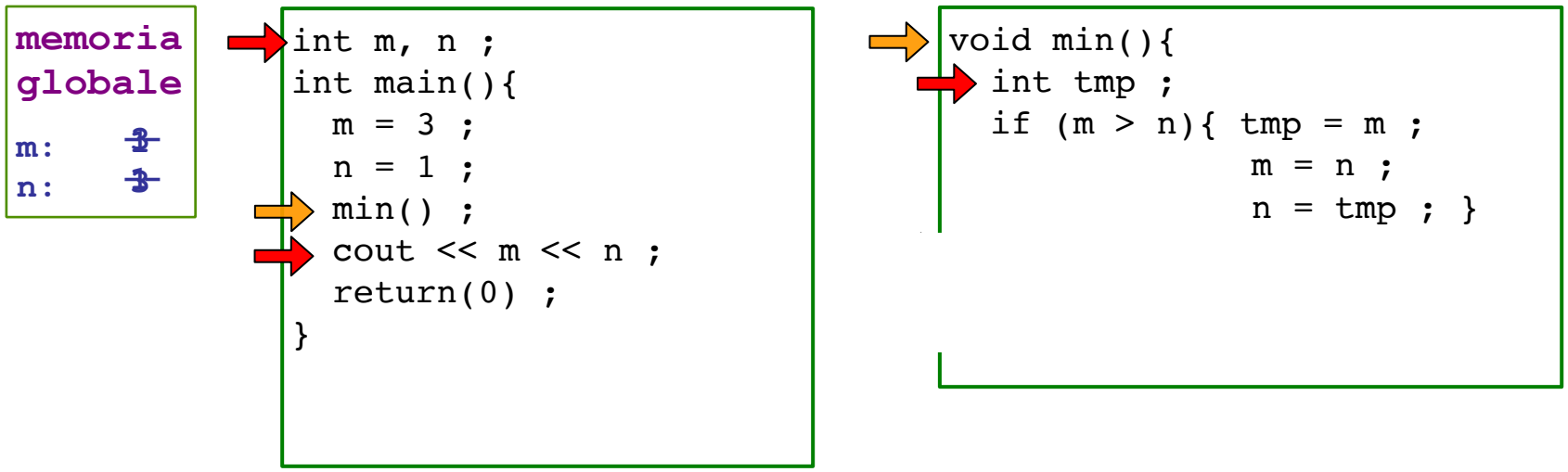
```
int m, n;
int main(){
      void min(){
          int tmp ; if (m > n) { tmp = m ; m = n ; n = tmp ; }
      }
      cin >> m >> n ;
      min() ;
      cout << m << n ;
      return(0) ;
}
```

è sbagliata!

FUNZIONI/ASPETTI SEMANTICI

semantica:

- ogni chiamata a funzione viene tradotta in **un trasferimento del controllo** al codice della funzione
- al termine del corpo di ogni funzione, **il compilatore aggiunge un'istruzione in linguaggio macchina che trasferisce il controllo all'istruzione che segue la chiamata alla funzione**



FUNZIONI: ESERCIZI

1. scrivere una funzione che legge un numero binario da tastiera e lo stampa in decimale

2. scrivere una funzione che legge due numeri binari da tastiera e ne stampa la somma e il prodotto (sempre in binario)

3. scrivere una funzione che dati due numeri interi `x` e `y` calcoli `x²+y² e x²*y`

4. scrivere una funzione che prende in input un intero e stampa l'intero con le cifre invertite

FUNZIONI: PROBLEMI CON LA PORTATA DELLE DICHIARAZIONI

cosa stampa?

portata della dichiarazione `n` **a linea** 2

```
1     int m ;
2     int n ;
3     void min(){
4        int tmp ;
5        if (m > n) { tmp = m ; m = n ; n = tmp ; }
6     }
7     int main() {
8        cin >> m >> n ;    // m = 10 ; n = 5
9        {  int m = 7 ;
10          min() ;
11          cout << m << n ;
12       }
13       return(0) ;
14    }
```

FUNZIONI: PROBLEMI CON LA PORTATA DELLE DICHIARAZIONI

cosa stampa?

```
1     int m ;
2     int n ;
3     void min(){
4        int tmp ;
5        if (m > n) { tmp = m ; m = n ; n = tmp ; }
6     }
7     int main() {
8        cin >> m >> n ;    // m = 10 ; n = 5
9        {  int m = 7 ;
10          min() ;
11          cout << m << n ;
12       }
13       return(0) ;
14    }
```

portata della dichiarazione `m` **a linea** `9`

FUNZIONI: PROBLEMI CON LA PORTATA DELLE DICHIARAZIONI

cosa stampa?

portata della dichiarazione `m` **a linea** `1`

```
1     int m ;
2     int n ;
3     void min(){
4        int tmp ;
5        if (m > n) { tmp = m ; m = n ; n = tmp ; }
6     }
7     int main() {
8        cin >> m >> n ;    // m = 10 ; n = 5
9        {  int m = 7 ;
10          min() ;
11          cout << m << n ;
12       }
13       return(0) ;
14    }
```

FUNZIONI: PROBLEMI CON LA PORTATA DELLE DICHIARAZIONI

cosa stampa?

```
1     int m ;
2     int n ;
3     void min(){
4        int tmp ;
5        if (m > n) { tmp = m ; m = n ; n = tmp ; }
6     }
7     int main() {
8        cin >> m >> n ;    // m = 10 ; n = 5
9        {  int m_9  = 7 ;
10          min() ;
11          cout << m_9 << n ;
12       }
13       return(0) ;
14    }
```

(cf. legame statico -- static binding)

FUNZIONI CON PARAMETRI

lo svantaggio delle funzioni **`min`** e **`radice_quadrata`** è che si applicano sempre alle stesse variabili (nel caso di sopra, sono *globali*)

- ogni volta che si usano, occorre sapere a quali variabili esse fanno riferimento

conseguenza: *poca flessibilità*, *possibilità di errore*, *codice poco leggibile*

esempio: `void somma(){  c = a+b ; }`

richiede la presenza di 3 variabili globali `a`, `b` e `c`

per calcolare la somma di `3` e `10` si deve fare:

```
a = 3 ; b = 10 ; somma() ; cout << c ;
```

desiderata: vorremmo invocare **`somma`** in questo modo

`somma(a, b)` oppure `somma(a+3, b*5 + 1)`

FUNZIONI CON PARAMETRI

quando si **definisce una funzione** occorre specificare la **lista dei parametri formali** che saranno usati

```
void somma(int a, int b){   c = a+b ;
}
void radice_quadrata(int n){
      int i = 0 ;
      while (i*i <= n) i = i+1 ;
      cout << i-1 ;
}
```

sintassi:

function-definition ::= *type identifier* **(** *formal-parameters* **) {**
declarations
statements
}

formal-parameters ::= *type identifier* , *type identifier* , . . .

FUNZIONI CON PARAMETRI

i parametri formali sono identificatori a cui è associato un tipo

quando si ***chiama una funzione*** occorre elencare gli **operandi** con cui saranno istanziati i parametri formali

gli operandi sono detti: **parametri attuali**

sintassi:

function-call ::= identifier (actual-parameters)

void-function-call ::= identifier (actual-parameters) **;** ← notare il ;

actual-parameters ::= expression , expression, . . .

osservazione: i parametri attuali sono espressioni

FUNZIONI CON PARAMETRI: ESEMPIO

```
void stampa_int (int x){
            cout <<  x+1 ;
}
int main(){
      stampa_int (6) ;
      return(0) ;
}
```

stampa 7

PORTATA DI UNA DICHIARAZIONE: LE REGOLE/CONT.

- i parametri formali sono visibili solo nel corpo della funzione in cui appaiono

FUNZIONI CON PARAMETRI/CONSIDERAZIONI

vincoli sintattici

- la lunghezza della lista dei parametri attuali deve essere uguale alla lunghezza della lista dei parametri formali
- ogni parametro attuale corrisponde al parametro formale che occupa la stessa posizione nella lista dei parametri formali
- il tipo del parametro attuale deve essere uguale al tipo del parametro formale corrispondente
  - quando possibile avvengono conversioni

`int` → `real` `real` → `int` `char` → `int`

considerazione semantica

l'ordine di valutazione dei parametri attuali è arbitrario (rilevante per funzioni che ritornano un valore e che possono aver modificato una variabile globale)

ORDINE DEGLI ARGOMENTI NELLE CHIAMATE

il compilatore verifica che i tipi degli argomenti siano corretti e nella corretta sequenza

il compilatore non può verificare che siano nella stessa sequenza logica

esempio: data la funzione

```
int dividi(int dividendo, int divisore){
        . . . // ritorna dividendo/divisore
}
```

se viene chiamata con

```
int numeratore, denominatore ;
cin >> numeratore >> denominatore ;
x = dividi(denominatore,numeratore) ;
```

si rischia un errore (divisione per 0) perchè gli argomenti non sono nell'ordine corretto!

FUNZIONI CON RITORNO DI VALORI

se una funzione ritorna un valore al chiamante (si pensi alle funzioni aritmetiche) occorre

- specificare un tipo non-`void` nell'intestazione della funzione (*il valore ritornato è un valore di quel tipo*)
- *inserire nel corpo* l'istruzione `return`:

statement ::= ... | `return(`*expression*`) ;`

vincolo sintattico: il tipo dell' "*expression*" deve essere lo stesso di quello specificato nell'intestazione (quando possibile ci sono conversioni)

IL COMANDO `RETURN`

esempi:

```
int sum(int m, int n) {  return(m+n) ;  }
int abstract(int m){
      if (m<0) return(-m) ; else return(m) ;
}
```

regola semantica: quando viene raggiunta una istruzione `return`:

1. l'espressione che è argomento di `return` viene calcolata e il valore viene ritornato al chiamante
2. **l'esecuzione della funzione si conclude ed il controllo ritorna al chiamante (i comandi dopo il `return` non vengono mai eseguiti)**

FUNZIONI CON RITORNO DI VALORI/COMMENTI ED ERRORI

commento: le funzioni con tipo **`void`** possono anche avere "**`return ;`**" al loro interno

- valgono le stesse considerazioni per le funzioni con ritorno di valori

errori: le funzioni con ritorno di valori senza **`return`** nel corpo o in qualche alternativa sono da considerare erronee (il risultato dipende dal compilatore):

```
int abstract(int m){
      if (m<0) return(-m) ;
      else if (m>0) return(m) ;
}
```

è indefinita quando l'input è **`0`**

FUNZIONI/PASSAGGIO DEI PARAMETRI

C++ offre **tre** modalità di passaggio dei parametri

1. *per **valore*** (è la modalità che abbiamo sempre usato finora)
2. *per **riferimento***
3. *per **costante***

nel ***passaggio per valore***:

- i parametri attuali sono valutati
- il loro valore è memorizzato in variabili locali alla funzione che corrispondono ai parametri formali
- ogni modifica all'interno del corpo della funzione riguarderà le variabili locali

esempio:

```
void scambia (int x, int y){ int tmp ;
        tmp = x ; x = y ; y = tmp ;
}
int main (){ int a = 1, b = 2 ;
        scambia (a, b) ; cout << a << b ;   return(0) ;
}
```

stampa 1 2 (e non 2 1)

FUNZIONI/PARAMETRI PER RIFERIMENTO

nel ***passaggio per riferimento***:

- i parametri formali devono essere dichiarati con "`&`" dopo il tipo (`int&` è l'indirizzo di una cella di memoria che contiene un intero, `char&` è l'indirizzo di una cella di memoria che contiene un carattere, etc.)
- i parametri attuali devono essere *variable* (identificatori o similari)
- ogni modifica all'interno del corpo della funzione riguarderà i parametri attuali

FUNZIONI/PARAMETRI PER RIFERIMENTO

esempio:

```
void scambia (int& x, int& y){
   int tmp ; tmp = x ; x = y ; y = tmp ;
}
int main (){ int a = 1, b = 2 ;
             scambia (a, b) ; cout << a << b ;
             return(0) ;
}
```

esecuzione:

x ed y sono degli alias

```
int main (){ int a = 1, b = 2 ;
   scambia (a, b) ;
   cout << a << b ;
   return(0) ;
}
```

memoria

a 1 2

b 2 1

```
void scambia(int& x, int& y){
   int tmp ;
   tmp = x; x = y; y = tmp;
}
```

memoria

tmp 1

&x

&y

stampa 2 1 (e non 1 2)

FUNZIONI/PARAMETRI PER RIFERIMENTO

esempio:

```
void scambia (int& x, int& y){ int tmp ;
                tmp = x ; x = y ; y = tmp ;
}
int main (){ int a = 1, b = 2 ;
                scambia (a+b, b) ;
                cout << a << b ;
                return(0) ;
}
```

è errore!

FUNZIONI/PARAMETRI PER RIFERIMENTO/COMMENTI

il ***passaggio per riferimento***:

- ***ottimizza l'uso della memoria***: consente di passare dati molto grandi senza doverli copiare nelle variabili locali delle funzioni (cf. arrays)
- ha lo svantaggio di rendere i programmi **incomprensibili** (va usato con estrema cautela)

esempio: cosa stampa?

```
int f(int& a){  a = a+1 ; return(a+3) ; }
int main(){ int x = 1 ; cout << f(x) + x ;
      return(0) ;
}
```

FUNZIONI/PARAMETRI PER COSTANTE

nel ***passaggio per costante***:

- i parametri formali devono essere dichiarati con "const" prima del tipo (const int è un intero usato come costante all'interno del corpo, const char è un usato come costante , etc.)
- all'interno del corpo della funzione i parametri passati per costante NON possono essere modificati

esempio:

```
int foo (const int x){
          int tmp ; tmp = x * x ; return(tmp) ;
}
int main (){
          int a = 2 ; cout << foo(a+a) ; return(0) ;
}
```

osservazione:

```
int foo (const int x){
        int tmp ; tmp = x * x ; x = x ; return(tmp) ;
 }
```

è errore!

DICHIARAZIONI DI FUNZIONI

la regola generale di `C++` stabilisce che

ogni identificatore debba essere definito prima del suo uso

e gli identificatori di funzione devono seguire questa regola!

esempio:

```
int main (){
        int a, b ; a = 1 ; b = 2 ; scambia (a, b) ;
        return(0) ;
}
void scambia (int& x, int& y){
        int tmp ; tmp = x ; x = y ; y = tmp ;
}
```

è sbagliato perchè `scambia` è usata **prima** della sua definizione

problema: non sempre è possibile definire le funzioni prima del loro uso (cf. *mutua ricorsione*)

DICHIARAZIONI DI FUNZIONI

soluzione: *dichiarazioni di funzione*, cioè si specifica solamente l'intestazione della funzione

le dichiarazioni di funzione servono a informare il compilatore

- del nome della funzione
- del numero e tipo dei parametri che devono essere passati alla funzione
- del tipo del valore da essa restituito

sintassi: *function-declaration ::= type identifier (type, type, . . .)* **;**

- **si può anche inserire l'identificatore dopo il tipo**
- **l'identificatore non deve coincidere col parametro formale**

DICHIARAZIONI DI FUNZIONI

esempio:

```
void scambia (int& x, int& y) ;
// Postcondition: scambia i valori di x e y

int main (){
      int a, b ; a = 1 ; b = 2 ;
      scambia (a, b) ;
      return(0) ;
}

void scambia (int& x, int& y){
      int tmp ; tmp = x ; x = y ; y = tmp ;
}
```

è corretto

FUNZIONI/REGOLE DI PROGRAMMAZIONE

evitare sempre funzioni che modificano variabili globali perchè

i. **complicano la comprensione del programma**

ii. **rendono *ambiguo il significato***

esempio:

```
int m, n;
void min(){
    int tmp ;
    if (m > n) { tmp = m ; m = n ; n = tmp ; }
}
int main() {
    cin >> m >> n ;   // m = 10 ; n = 5
    {   int m = 7 ; min() ; cout << m << n ;
    }
    return(0) ;
}
```

altro esempio:

```
int x ;
 int f(){  x = x+1 ; return(x) ; }
 int main(){  x = 1 ; cout << f() + x ; return(0) ; }
```

FUNZIONI/REGOLE DI PROGRAMMAZIONE

come evitare funzioni che modificano variabili globali:

1. ***se una funzione deve accedere in lettura ad una variabile globale***, aumentare il numero di parametri formali e, ogni volta che la si chiama, si passa la variabile globale come parametro
2. ***se una funzione deve modificare una variabile globale***, aumentare il numero di parametri formali con un parametro passato per reference e, ogni volta che la si chiama, si passa la variabile globale come parametro [POSSIBILMENTE DA EVITARE!]

FUNZIONI/REGOLE DI PROGRAMMAZIONE

evitare sempre di <u>interrompere una iterazione</u> *(for, while)*

con un `return`

- la leggibilità del programma aumenta quando l'unica condizione di terminazione di una iterazione è la guardia del comando iterativo

BLACK BOX

l'analogia col black box

- indica che si utilizza qualcosa perchè si conosce **come usarla**, senza sapere **come e fatta** (cellulari, automobili, etc.)
- una persona che usa un programma può non sapere come e fatto: necessita solo di sapere cosa fa

funzioni e l'analogia black box

- un programmatore che usa una funzione deve sapere cosa fa la funzione non **come** lo fa
- un programmatore che usa una funzione deve **soltanto** conoscere come invocarla
- è detta **procedural abstraction**

NASCONDERE L'INFORMAZIONE

(o **information hiding**) l'analogia black box consente di "nascondere" in un altro file o in un'altra parte del file l'implementazione delle funzioni

- in questo modo è possibile modificare (upgrade) la funzione senza che gli utilizzatori ne siano a conoscenza
- il codice diventa più leggibile: l'utente sa cosa fa la funzione leggendo la **dichiarazione e i relativi commenti**

PROCEDURAL ABSTRACTION

- i commenti vanno scritti nel modo seguente

```
int foo(int x, double& z) ;

// Precondition: x > 0
// Postcondition: z memorizza la metà di x
```

gli identificatori `x` e `z` servono per poter scrivere Pre e Postcondizioni

- le precondizioni devono dire quali sono le condizioni per gli argomenti delle funzioni
- le postcondizioni devono descrivere il valore ritornato
- tutte le altre informazioni (variabili locali, dettagli sull'algoritmo, etc.) devono essere nascoste nel corpo

INCLUSIONE DI LIBRERIE

vogliamo utilizzare identificatori presenti in altri file, possibilmente sviluppati da altri programmatori

```
// file library.h
void min (int& x, int& y) ;
```

```
// file library.cpp
#include "library.h"
void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = a ; a = b ; b = tmp ; } ;
}
```

```
// file working.cpp
#include <iostream>
#include "library.h"
using namespace std;

int main (){
      int a=2, b=1;
      min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

in **Eclipse**: creare i file **library.h** e **library.cpp** ad esempio nella stessa cartella del **main** cliccando col tasto destro sul nome della cartella e selezionando l'opzione **new**

si possono creare file in progetti diversi, ad esempio nel progetto **Library** e accedere da **working.cpp**

* cliccando su `propreties/"C/C++ General"/"Paths and Symbols"`
* in `"Includes"`, opzione `"Gnu C++"`, aggiungere la cartella della libreria
* in `"Source Location"/"Add Folder"`, aggiungere cartella della Libreria
* a questo punto in working.cpp scrivere `#include "library.h"`

INCLUSIONE DI LIBRERIE/PROBLEMI

include di librerie può causare errori:

```
// file library.h
void min (int& x, int& y) ;
```

```
// file librarybis.h
void min (int& x, int& y) ;
```

```
// file library.cpp
#include "library.h"
void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = a ; a = b ; b = tmp ; } ;
}
```

```
// file librarybis.cpp
#include "librarybis.h"
void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = 2*a ; a = 2*b ; b = tmp ; } ;
}
```

```
// file working.cpp
#include <iostream>
#include "library.h"
#include "librarybis.h"
using namespace std;

int main (){
      int a=2, b=1;
      min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

errore perché `min` è definito in due librerie

(presenza di dichiarazioni multiple)

i **namespace** consentono di ovviare a questo problema

I NAMESPACE

i namespace consentono di risolvere questi problemi:

```
// file library.h
namespace one {
   void min (int& x, int& y) ;
}
```

```
// file library.cpp
#include "library.h"
namespace one {
   void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = a ; a = b ; b = tmp ; } ;
   }
}
```

```
// file librarybis.h
namespace two {
   void min (int& x, int& y) ;
}
```

```
// file librarybis.cpp
#include “librarybis.h"
namespace two {
   void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = 2*a ; a = 2*b ; b = tmp ; } ;
   }
}
```

```
// file working.cpp
#include <iostream>
#include "library.h"
#include "librarybis.h"
using namespace std;

int main (){
      int a=2, b=1;
      one::min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

- un **namespace** è uno scope “etichettato” da un identificatore
- la sintassi “**: :**” è usata per **specificare** quale namespace si sta usando, tra tutti quelli possibili

I NAMESPACE E USING

- per evitare di scrivere
  `one::min(a,b)`
  uno può usare la dichiarazione
  `using`

```
// file working.cpp
#include <iostream>
#include "library.h"
#include "librarybis.h"
using namespace std;

using one::min;

int main (){
      int a=2, b=1;
      min(a,b) ;
      two::min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

per accedere al namespace `two`

- in alternativa uno può utilizzare la direttiva **`namespace`**

```
// file working.cpp
#include <iostream>
#include "library.h"
#include "librarybis.h"
using namespace std;

using namespace one ;

int main (){
      int a=2, b=1;
      min(a,b) ;
      two::min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

ESERCIZI

1. scrivere una libreria che contiene le funzioni
   - `bin2dec`: prende un numero binario e ritorna il suo valore in base 10
   - `dec2bin`: prende un numero decimale e ritorna il suo valore in base 2
   - `bin_sum`: prende due numeri in binario e ritorna lo loro somma in binario
   - `bin_prod`: prende due numeri binari e ritorna il loro prodotto in binario

2. scrivere una funzione "`ln`" che prende in input un double `1+x, -1<x<=1,` e ritorna il logaritmo naturale definito dalla formula (serie di Mercator)

   (iterarla fino a 50) $$\ln(1+x) = \sum_{n=1}^{+\infty} \frac{(-1)^{n+1}}{n} x^n = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$$

3. scrivere una funzione che risolve le equazioni di 2^ grado $ax^2+bx + c = 0$, cioè prende in input a, b e c e ritorna $$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   problema: come facciamo a tornare 2 valori?