Corso di Laurea triennale in Ingegneria Chimica

in condivisione con

Corso di Laurea triennale in

Ingegneria Navale e Scienze dei Materiali

Elementi di Informatica

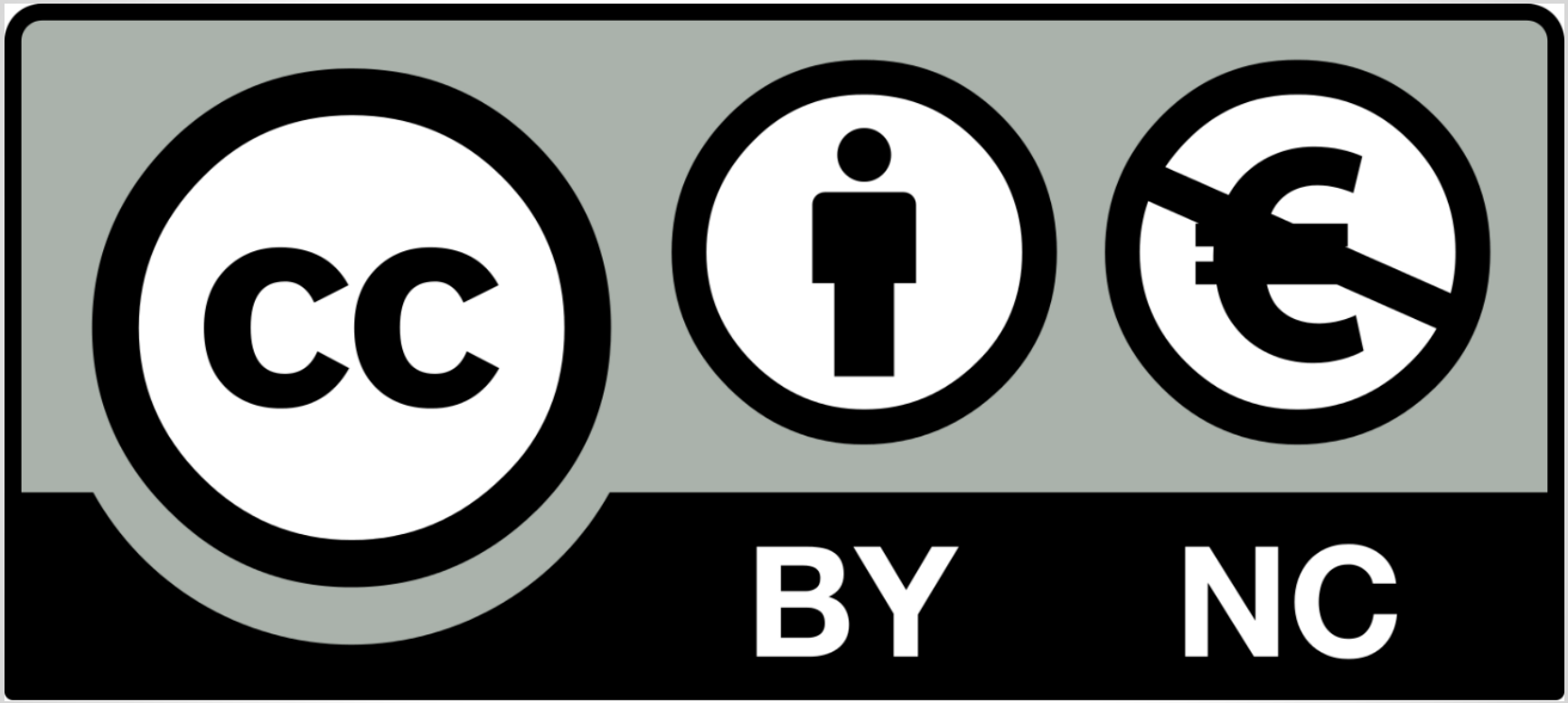
A.A. 2016/17

prof. Mario Barbareschi

Sottoprogrammi

Informazioni di Licenza

- Questo lavoro è licenziato con la licenza Creative Commons BY-NC



- Per consultare una copia della licenza visita: http://creativecommons.org/licenses/by-nc/3.0/legalcode

L'astrazione

- **L'astrazione** è il processo che porta ad identificare le proprietà rilevanti di un'entità (o di un fenomeno), ignorando i dettagli inessenziali
  - Le proprietà così astratte definiscono una **vista dell'entità** (o del fenomeno)
  - Una stessa entità può dar luogo a viste diverse

- Esempio: un'automobile
  - vista dal venditore: prezzo, durata della garanzia, colore, …
  - vista dal meccanico: tipo di motore, cilindrata, tipo di olio, …

- L'astrazione è un caso particolare di **separazione degli interessi:**
  - Separa aspetti rilevanti da dettagli considerati secondari per un dato fine
  - È uno strumento per dominare la complessità di problemi.

La modularità

- La **modularità** è **l'organizzazione in parti** (per moduli) di un modello o di un sistema, in modo che esso risulti **più semplice da comprendere e manipolare**.
  - Gran parte dei sistemi complessi sono modulari

- Esempio:
  - Un'automobile è suddivisa in più sottosistemi: Motore, Trasmissione, …, ed ogni parte si concentra sul risolvere un determinato problema.

Il concetto di modulo

- Un **modulo di un sistema software** è un componente che:
  - Realizza una astrazione
  - È dotato di netta separazione tra:
    - Interfaccia
    - Corpo
- **L'interfaccia specifica il cosa** (ovvero l'astrazione realizzata dal modulo).
- Il corpo descrive come è realizzata l'astrazione

Un modulo può ad es. ***realizzare una funzionalità***, ***fornire servizi*** o ***gestire risorse***

Meccanismi di astrazione

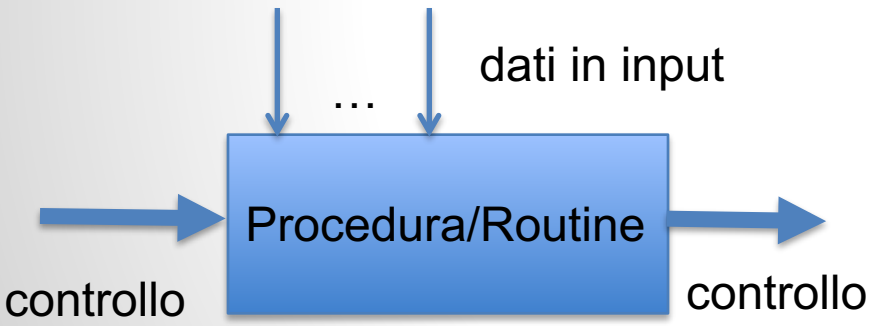
- *I tipi di astrazione realizzati da un modulo possono essere:*
- **ASTRAZIONE SUL CONTROLLO**
  - Consiste nell'astrarre una data **funzionalità** dai dettagli della sua implementazione;
  - È ben supportata dai linguaggi di programmazione tradizionali tramite il concetto di **sottoprogramma**.
- **ASTRAZIONE SUI DATI**
  - Consiste nell'astrarre le ***entità*** (oggetti) costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa;
  - Può essere realizzata con un uso opportuno delle tecniche di **programmazione modulare** nei linguaggi tradizionali;
  - È supportata da appositi costrutti nei linguaggi di **programmazione ad oggetti.**

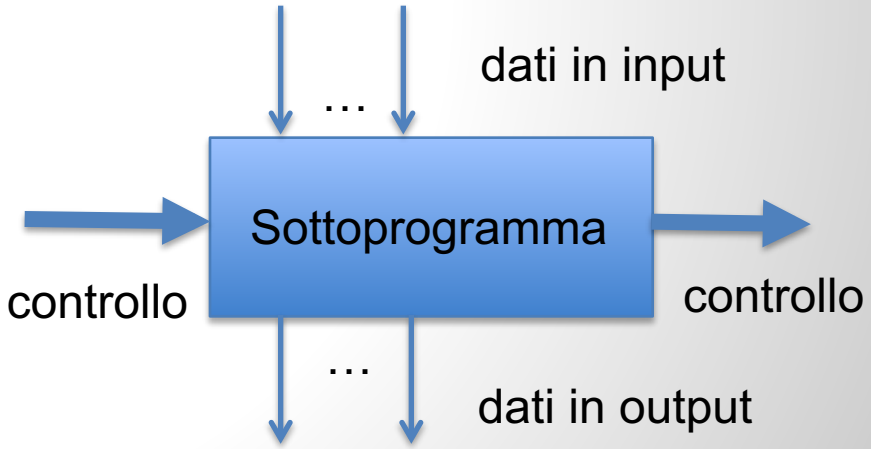
La modularità: sottoprogrammi (1/2)

- Per gestire la complessità dei programmi è utile individuare al loro interno dei **moduli funzionali**, detti **sottoprogrammi**:
  - Ai moduli è passato il flusso di controllo in esecuzione: essi hanno un solo punto in ingresso e di uscita (secondo i principi della programmazione strutturata).
- Ai **sottoprogrammi** si assegnano determinate **responsabilità**, che includono la risoluzione di un sotto-problema:
  - I sottoprogrammi hanno un **nome** e possono essere attivati durante l'esecuzione del programma zero, uno o più volte.

La modularità: sottoprogrammi (2/2)

- Un sottoprogramma scambia informazioni con l'esterno, in ingresso e/o in uscita.
  - Se il sottogramma non scambia informazioni in uscita, si chiama anche **procedura** o **routine**.
  - Se il sottoprogramma fornisce informazioni in uscita, allora si chiama anche **funzione**.





Indicazione di un sottoprogramma

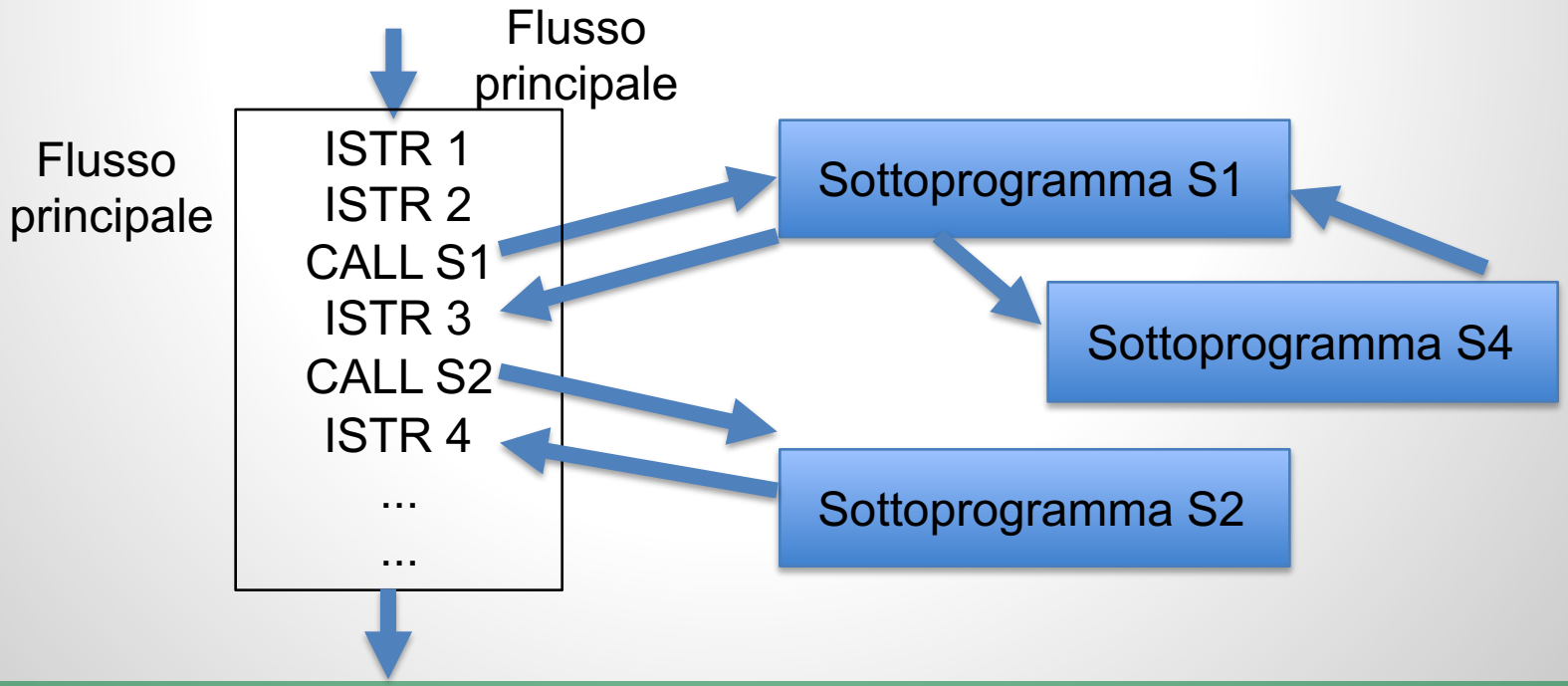
- La **definizione** di un sottoprogramma si compone di **titolo** e il **corpo**.
  - Il **titolo** comprende:
    - il **nome** del sottoprogramma
    - i **parametri di ingresso** con il loro tipo
    - i **dati output con il suo tipo**
  - Il **corpo** è tipicamente un blocco sequenziale che comprende una sequenza di dichiarazioni e di istruzioni.

C++

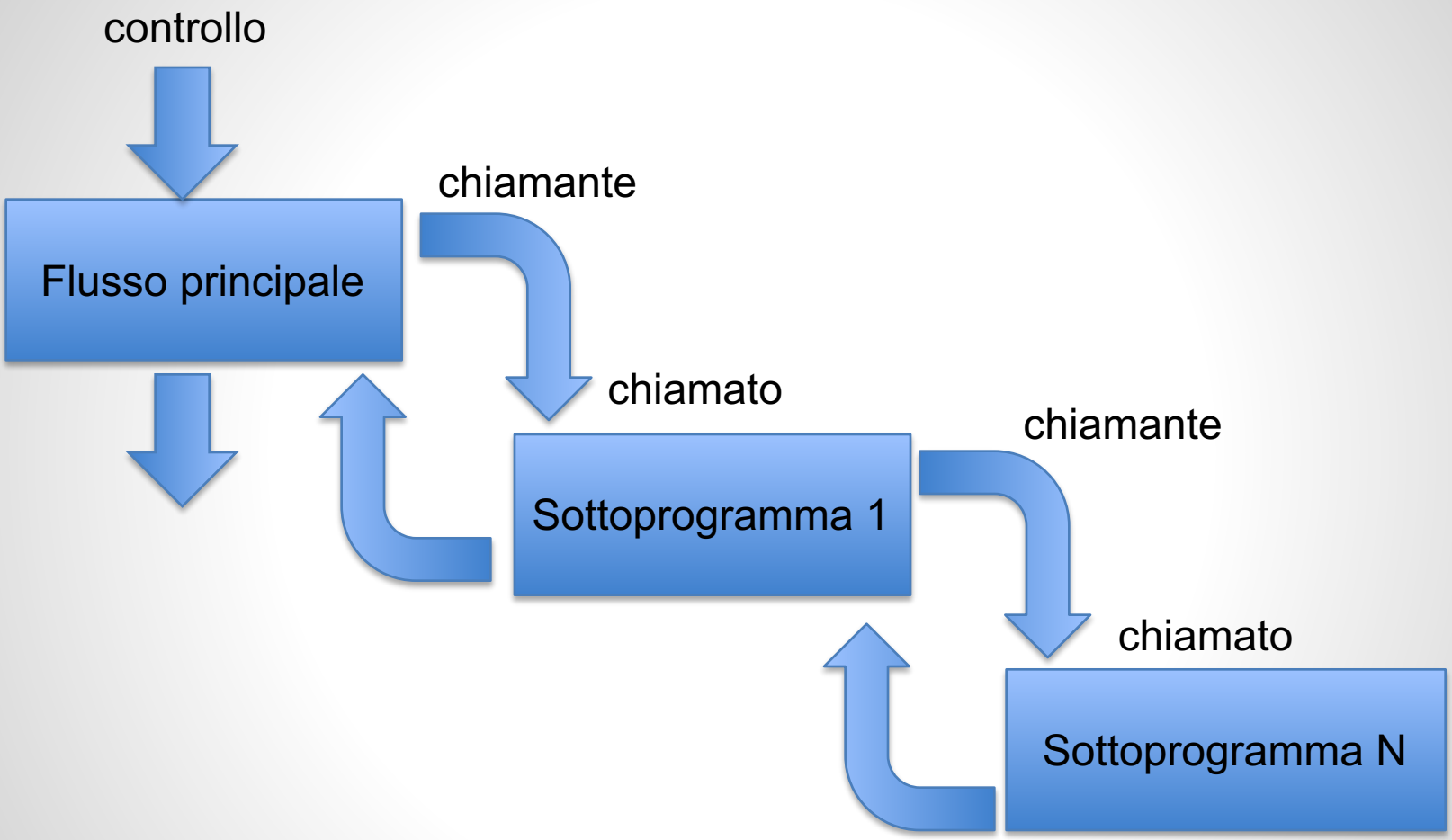
```
returnType functionName(paramType param_1, …, paramType param_n)
{
    <function body>
}
```

Invocazione di un sottoprogramma

- Una volta definito, è possibile richiamare il sottoprogramma utilizzando il suo nome e fornendo i parametri di ingresso, se presenti.
  - L'invocazione del sottoprogramma è un'istruzione come le altre
  - Evita di riscrivere più volte lo stesso insieme di istruzioni laddove queste vadano ripetute
  - Consente di concentrarsi su un problema di piccole dimensioni rispetto al programma principale



Distinzione chiamante-chiamato



Parametrizzazione del codice

- Si definiscono **parametri formali** i parametri **dichiarati** nell'interfaccia della sottoprogramma:
  - essi sono **definiti** (gli si assegna un valore) all'atto dell'invocazione dal **chiamante**
- Quando il sottoprogramma viene invocato ai parametri formali si assegnano i valori dei **parametri effettivi (ovvero quelli usati per attivare il sottoprogramma).**
  - **I parametri effettivi e quelli formali devono essere dello stesso tipo e devono essere specificati nello stesso ordine!**

Esempio: Parametri formali ed effettivi (1/2)

Invocazione con parametri effettivi

Parametri formali (dichiarati ma non definiti!)

C++

```
// Parametri effettivi due valori costanti
somma(0.9, 0.1);
```

C++

```
double somma(double x, double y)
{
    <body>
}
```

```
double somma() {
    double x = 0.9;
    double y = 0.1;
    <body>
}
```

Nell'invocazione ai parametri formali è assegnato il **VALORE dei parametri effettivi, al momento dell'invocazione!**

Esempio: Parametri formali ed effettivi (2/2)

Invocazione con parametri effettivi

Parametri formali (dichiarati ma non definiti!)

C++

```
double k = 10.0;
somma(k, 0.1);
k = -24;
somma(k, 0.5);
```

C++

```
double somma(double x, double y)
{
    <body>
}
```

```
double somma() {
    double x = -24;
    double y = 0.5;
    <body>
}
```

```
double somma() {
    double x = 10.0;
    double y = 0.1;
    <body>
}
```

Nell'invocazione ai parametri formali è assegnato il **VALORE dei parametri effettivi, al momento dell'invocazione!**

L'istruzione return

- Per terminare il sottoprogramma e restituire il controllo al **chiamante** esistono speciali istruzioni, definite nei linguaggi di programmazione:
  - Il C++ adotta la keyword **return**

C++

```
double somma(double x, double y)
{
    double z = x + y;
    return z;
}
```

return termina l'esecuzione del sottoprogramma
e restituisce al più **un valore** al **chiamante**,
del tipo dichiarato nell'interfaccia (returnType)

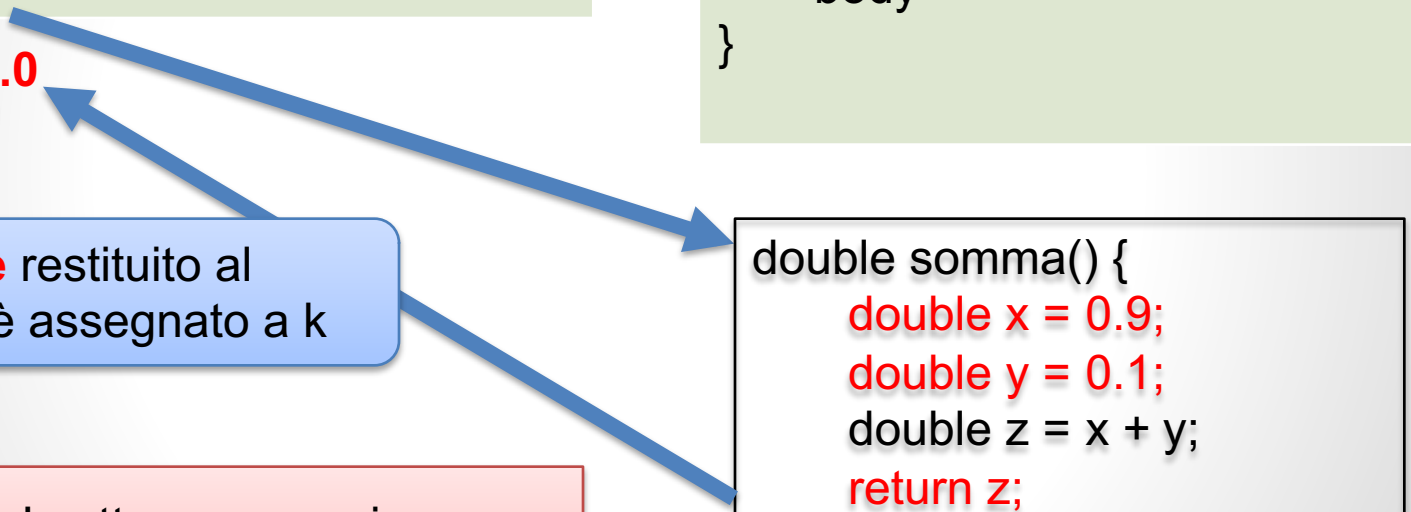
Esempio: istruzione return

C++

```
// Parametri effettivi due valori costanti
double k = somma(0.9, 0.1);
```

C++

```
double somma(double x, double y)
{
    <body>
}
```



N.B.: I sottoprogrammi trasferiscono **VALORI, NON VARIABILI!**

Sottoprogrammi senza output: tipo void

- Siccome in C++ è sempre necessario specificare un tipo di ritorno per un sottoprogramma, nel caso in cui si voglia scrivere una routine (sottoprogramma senza dati in output), si può usare il tipo **void**

C++

```
void stampaNumero(int numero)
{
    cout << “Numero = “ << numero;
}
```

Nel caso di funzioni void è comunque possibile
usare l’istruzione **return** per restituire il controllo al chiamante.

In ogni caso, il controllo è restituito al chiamante
al termine del blocco sequenziale del sottoprogramma “{…}”

Esercizio 12

- Scrivere un programma che:
  - Chiede all'utente due estremi a virgola mobile "a, b"
  - Chiede all'utente un numero a virgola mobile "passo"
- Tabella la funzione matematica $y = x^2$ stampando a schermo il valore della funzione tra gli estremi a e b secondo il passo specificato dall'utente.
- Esempio input: "0.5 1.5 0.25"
- Esempio output:

  f(0.5) = 0.25
  f(0.75) = 0.5625
  f(1) = 1
  f(1.25) = 1.5625
  f(1.5) = 2.25
- Si realizzino i sottoprogrammi:
  - double leggiDouble();
  - double calcolaFunzione(double x);
  - void tabellaFunzione(double a, double b, double dx);

Visibilità (1/2)

- Possiamo dichiarare variabili sia nel programma principale, sia nei sottoprogrammi.
- Le **regole di visibilità** (o di **scope**) indicano l'ambito in cui una variabile (in generale un oggetto) è dichiarata in un programma.
  - Le regole di visibilità sono importanti per evitare **interferenze** con variabili che non rientrano nel contesto di una trasformazione!

C++

```
double x = 3.1;
double y = 0.9;
double z = 0.0;
double k = somma(x, y);
```

C++

```
double somma(double x, double y)
{
    // posso usare z??
    return x + y + z;

}
```

Il compilatore segnala errore perché **z** non è visibile nel sottoprogramma!

Visibilità (2/2)

- Un oggetto (una costante, una variabile, …) può avere visibilità:
  - **locale al sottoprogramma,** se definito al suo interno **(interfaccia e/o corpo)**.
  - **non locale al sottoprogramma**, se è dichiarato esternamente.

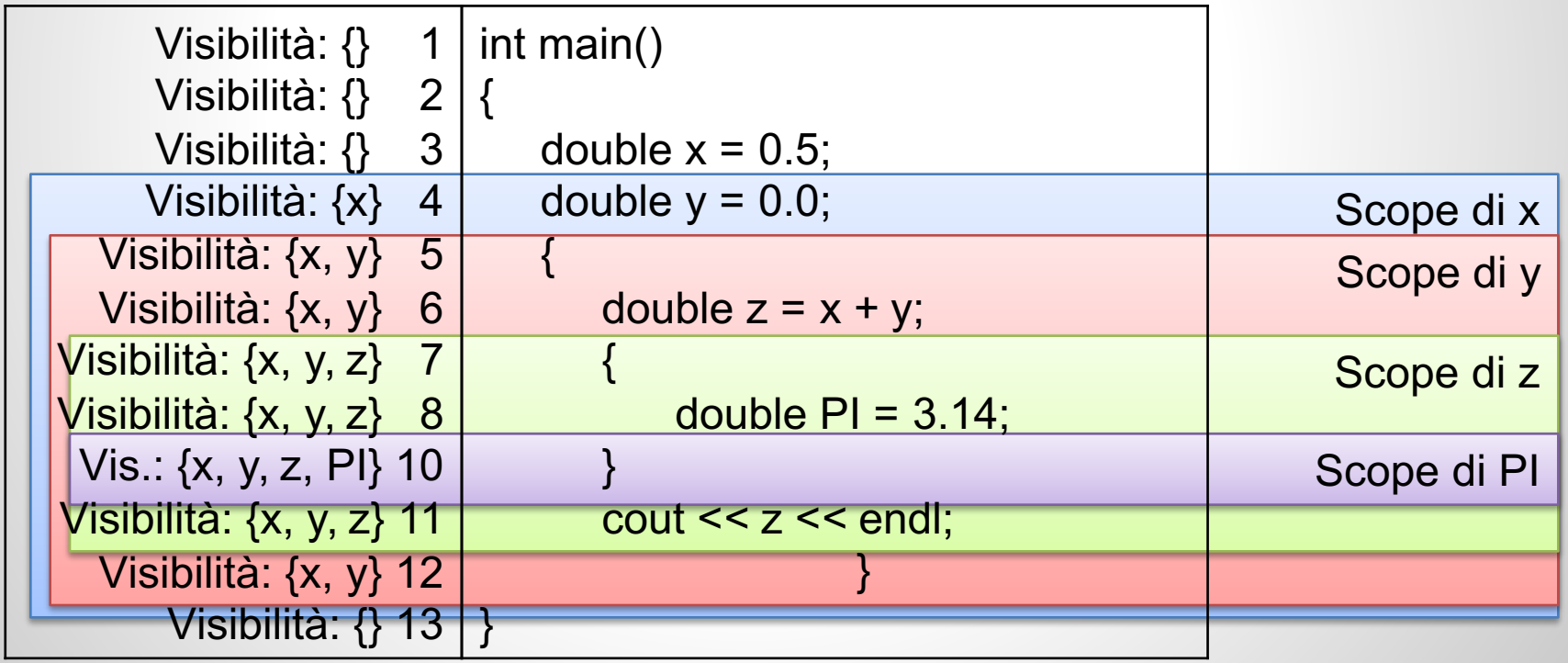
- Lo **scope degli oggetti locali** è limitato al sottoprogramma. Gli oggetti:
  - **non sono visibili** in nessun altro sottoprogramma chiamato.
  - **non sono visibili** al programma chiamante, dopo la terminazione del sottoprogramma.

- Gli **oggetti non locali al sottoprogramma**:
  - risultano visibili all'interno del sottoprogramma, e nei sottoprogrammi chiamati.
  - Rimangono visibili al termine del sottoprogramma.

Oggetti non locali ad un sottopr. si indicano anche come **globali**, oppure **esterni**.

Visibilità e C++ (1/2)

- In C++ la visibilità di un oggetto è limitata al **blocco sequenziale** più interno nella quale è definita:
  - I blocchi sequenziali più interni hanno visibilità degli oggetti di tutti i blocchi nei quali sono contenuti.
  - Un blocco esterno non ha visibilità degli oggetti nei blocchi interni.



Visibilità e C++ (2/2)

- In C/C++ Le variabili dichiarate in un blocco sono **variabili locali al sottoprogramma nel quale il blocco è inserito**:
  - Non sono visibili ai sottoprogrammi chiamati.
  - Non sono visibili dopo la terminazione del sottoprogramma.
- Per creare variabili **non locali ad un programma** (variabili globali) in C/C++ occorre definirle **esterne ad ogni blocco (es.: in testa al sorgente).**

```
const double PI = 3.1416;

double areaCerchio(double raggio) {
 …
}

double circonferenza(double raggio) {
...
}
int main() { ... }
```

Scope di PI:
tutto il resto del file

Visibilità C/C++: osservazioni (1/3)

- La visibilità di un oggetto inizia dalla sua dichiarazione, sino alla chiusura del blocco sequenziale che lo contiene (per le variabili locali), o del file sorgente (per le variabili esterne).
  - Questo vale, oltre che per i sottoprogrammi, anche per i costrutti di controllo!

Ad ogni iterazione si apre e chiude il blocco del for che dichiara y, dunque si riassegna la variabile "y" **(non è persistente tra le iterazioni)**

```
for (int i = 0; i < 10; i++) {
    double y = f(i);

}
cout << y << endl;
```

y non è visibile dopo la chiusura del blocco che la dichiara!

Visibilità C/C++: osservazioni (2/3)

- La visibilità di un oggetto inizia dalla sua dichiarazione, sino alla chiusura del blocco sequenziale che lo contiene (per le variabili locali), o del file sorgente (per le variabili esterne).
  - Questo vale, oltre che per i sottoprogrammi, anche per i costrutti di controllo!

Essendo ora il blocco del for "interno" al blocco che dichiara y, allora y è la stessa variabile ad ogni iterazione (**persistente**)

```
double y;
for (int i = 0; i < 10; i++) {
    y = f(i);

}
cout << y << endl;
```

Ora lo scope di y include l'istruzione di stampa.

Visibilità C/C++: osservazioni (3/3)

- Se un blocco dichiara una variabile che ha lo stesso nome di una variabile già visibile nel suo ambito, si ha un fenomeno di **aliasing**.
  - La variabile con lo stesso blocco più interna maschera tutte le variabili con lo stesso nome dichiarate nei blocchi più esterni.

Le dichiarazioni di **i** e **k** nel blocco del for mascherano le variabili i e k dichiarate in un blocco esterno

L'ultima stampa vede le variabili i, k del blocco esterno (che non sono state modificate nel for, perchè mascherate).

C++

```
int i = 55;
int k = -8;
for (int i = 0; i < 3; i++) {
    int k = 100;
    cout << i << " " << k << endl;
}
cout << i << " " << k << endl;
```

Output:

```
0 100
1 100
2 100
55 -8
```

Riepilogo definizione funzioni in C++

I sottoprogrammi che non restituiscono alcun valore hanno tipo **void**

C++

```
void nomeRoutine(tipo1 param1, tipo2 param2, …)
{ … }
```

C++

```
tipoReturn nomeSottoprogramma(tipo1 param1, tipo2 param2, …)
{ … }
```

I sottoprogrammi che non restituiscono **un solo valore** hanno un tipo diverso da **void**

E sottoprogrammi che restituiscono più valori in output?

Il linguaggio C/C++ non supporta la scrittura di sottoprogrammi con più valori in output, ma è possibile comunque realizzarli ...

Sostituzione dei parametri formali (1/3)

- I parametri effettivi possono essere sostituiti a quelli formali in C/C++ in due diverse modalità: *per valore, per riferimento .*
- **Passaggio per valore**: al parametro formale viene assegnato il valore del parametro effettivo
  - Il parametro effettivo può essere un'espressione, una costante o una variabile. Se è un'espressione (es.: "x+4*y)", se ne calcola il valore e lo si assegna al corrispondente parametro formale.
  - Nel caso di variabili, la sostituzione per valore garantisce che la variabile passata come parametro effettivo non venga alterata (il sottoprogramma lavora sulla sua copia)
  - All'atto della chiamata viene fatta la copia del parametro effettivo con conseguente consumo di tempo e spazio

Passaggio per valore: esempio

Nel passaggio per valore **il valore dei parametri attuali al momento dell'invocazione è copiato nei parametri formali**

C++

```
double s = 10.0;
double t = 5.0;
double u = Somma(s, t);
```

C++

```
double somma(double x, double y)
{
    double r = x + y;
    x = 1;
    y = 2;
    return r;
}
```

s = 10.0
t = 5.0
u = **15.0**

```
double somma() {
    double x = 10.0;
    double y = 5.0;
    double r = x + y;
    x = 1;
    y = 2;
    return z;
}
```

I parametri formali sono **locali** al sottoprogramma, e non modificano i parametri effettivi

Sostituzione dei parametri formali (2/3)

- **Passaggio per riferimento**: al parametro formale viene assegnato l'indirizzo del parametro effettivo
  - Il parametro effettivo deve essere una variabile
  - Il sottoprogramma lavora direttamente sulla variabile in memoria
  - Il parametro effettivo occupa solo lo spazio necessario per contenere un indirizzo

Passaggio per riferimento: esempio

Nel passaggio per riferimento
ai parametri formali è assegnato
un **riferimento dei parametri attuali**
al momento dell'invocazione

C++

```
double s = 10.0;
double t = 5.0;
double u = Somma(s, t);
```

C++

```
double somma(double& x, double& y)
{
    double r = x + y;
    x = 1;
    y = 2;
    return r;
}
```

s = 1;
t = 2;
u = 15.0

```
double somma() {
    double x = Rif. a "s"
    double y = Rif. a "t"
    double r = x + y;
    x = 1;
    y = 2;
    return z;
}
```

Modificare un riferimento
altera il valore del parametro attuale!

Sostituzione dei parametri formali (3/3)

- Tramite un passaggio per riferimento possiamo creare funzioni C/C++ che restituiscono più di un valore in uscita:
  - Sempre e al più un valore più essere restituito al chiamante tramite "return"
  - Tutti gli altri valori in output possono essere scambiati con passaggio dei parametri per riferimento.

- Il passaggio per riferimento non necessariamente ha l'obiettivo di aggiungere ulteriori dati in output dalla funzione:
  - Possiamo scambiare per riferimento anche per variabili in input ai sottoprogrammi, invece che per valore.
  - Il passaggio per riferimento risulta più efficiente quando si scambiano dati di grandi dimensioni (es.: dati strutturati).

Scambio di parametri per riferimento: C++

- Per scambiare un parametro per riferimento in C++:
  - Si aggiunge una “&” dopo il tipo del parametro.

C++

```
double somma(double& x, double& y);
```

  - I parametri passati per rifermento sono a tutti gli effetti di “input-output”.
  - Se si vuole proteggere la modifica di un parametro passato per riferimento (e dunque usarlo esclusivamente per input) si può usare il qualificatore **const**

C++

```
double somma(const double& x, const double& y);
```

Funzioni a più parametri di output

- In generale, se un sottoprogramma restituisce più parametri di output si può adottare la seguente convenzione:
  - Si usa come returnType della funzione il tipo **void**
  - Si indicano i parametri in ingresso alla funzione (passandoli per valore, o per riferimento "const")
  - Si indicano i parametri in output alla funzione (o input-output) passandoli per riferimento

C++

```
void complesso_coniugato(double p_i, double p_j, double& y_i, double& y_j);
```

Due parametri passati per valore (di input alla funzione)

Due parametri passati per rif. e non costanti, ovvero di output (o in-out) alla funzione

Il programma principale

- Si chiamerà **programma principale** quella unità di programma che si interfaccia direttamente con il sistema operativo.
- In molti linguaggi il programma principale si chiama **main**
  - La funzione main è il punto da cui comincia l'esecuzione, indipendentemente dalla posizione nel listato
  - La funzione main può prendere dei parametri in ingresso (che servono per interagire col sistema operativo) e restituisce solitamente un valore intero, che indica l'esito dell'esecuzione (codice di errore)
    - Per convenzione "0" indica che il programma ha completato la sua esecuzione correttamente.
  - Tutti i programmi C/C++ contengono sempre e solo una funzione main

Librerie del linguaggio C/C++

- Insiemi di sottoprogrammi sono organizzati in librerie, ovvero moduli di funzionalità che possono essere facilmente riusati in più applicazioni.
- Per includere una libreria nel linguaggio C/C++ si usa l'istruzione (da aggiungere generalmente in testa al file sorgente):

  #include <nomeLibreria>

  Oppure (nel caso in cui non sia installata nel path predefinito dal compilatore):

  #include "pathLIbreria"
- Inclusa una libreria si può far uso degli oggetti che sono esportati da essa.