Università degli Studi di Cassino e del Lazio Meridionale

Corso di Fondamenti di Informatica

Sottoprogrammi

Anno Accademico 2016/2017

Francesco Tortorella

Nuove operazioni ?

- In qualunque linguaggio di programmazione il tipo di dati non specifica solo l'insieme dei valori che a questo appartengono, ma anche le operazioni su questi definite.
- Solo alcune di tali operazioni sono però rese disponibili dal linguaggio; le altre devono essere implementate dal programmatore.

Come inserirle ?

- Definito l'algoritmo che realizza una particolare operazione, i costrutti visti finora ne permettono la codifica.
- Il problema è che il codice va replicato ogni qual volta quell'operazione è richiesta nel programma.
- Conseguenze:
  - Minore leggibilità del codice
  - Maggiore probabilità di errori
  - Minore manutenibilità del codice

Sottoprogrammi

- L'ideale sarebbe un meccanismo che permetta di arricchire il linguaggio con una istruzione che realizzi quell'operazione.
- Tale meccanismo dovrebbe dare al programmatore la possibilità di:
  - specificare le istruzioni che realizzano l'operazione richiesta;
  - specificare i dati coinvolti;
  - specificare il nome con cui identificare l'operazione.
- Questo meccanismo è realizzato tramite i **sottoprogrammi**.

Sottoprogrammi

- Un sottoprogramma è una particolare unità di codice che non può essere eseguita autonomamente, ma soltanto su richiesta del programma principale o di un altro sottoprogramma.
- Un sottoprogramma viene realizzato per svolgere un compito specifico (p.es. leggere o stampare gli elementi di un array, calcolare il valore di una particolare funzione matematica, ecc.) per il quale implementa un opportuno algoritmo.

Sottoprogrammi

- Per questo scopo, il sottoprogramma utilizza variabili proprie, alcune delle quali sono impiegate per scambiare dati con il programma dal quale viene attivato.
- Un sottoprogramma può essere attivato più volte in uno stesso programma o anche utilizzato da un programma diverso da quello per cui era stato inizialmente progettato.

Sottoprogrammi: definizione

- Nel definire un sottoprogramma è quindi necessario precisare:
  - Quale operazione esso realizza
  - Qual è il flusso di dati tra il sottoprogramma ed il codice che lo ha attivato ed, in particolare:
    - Quali sono i dati in ingresso al sottoprogramma
    - Quali sono i dati in uscita dal sottoprogramma

Sottoprogrammi: attivazione

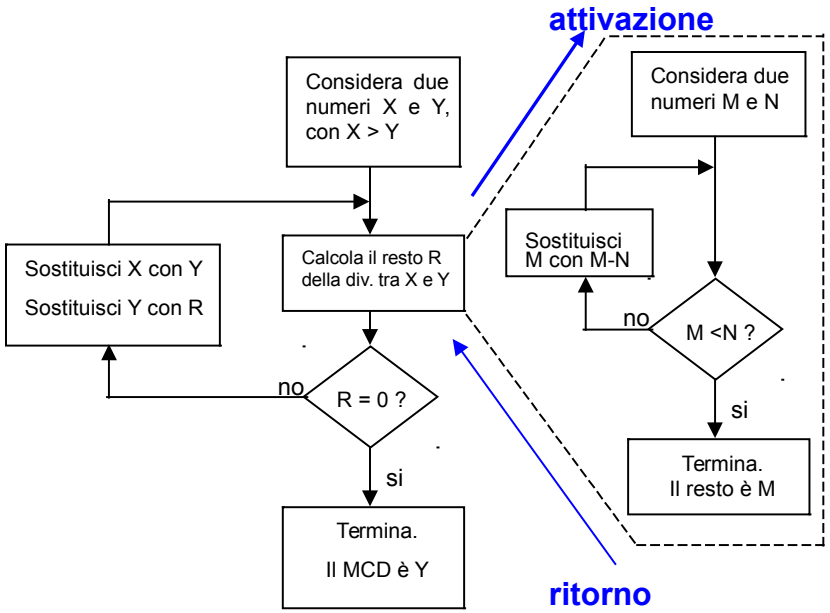
- L'esecuzione delle istruzioni di un sottoprogramma è provocata da una particolare istruzione del programma che lo attiva (istruzione di *chiamata*, per cui il programma è anche detto *chiamante*). Ciò determina la sospensione dell'esecuzione delle istruzioni del programma chiamante, che riprenderà dopo l'esecuzione dell'ultima istruzione del sottoprogramma (tipicamente, un'istruzione di *ritorno*).

Sottoprogrammi: flusso di dati

- Il programma chiamante ed il sottoprogramma scambiano dati attraverso una lista di variabili, definite all'interno del sottoprogramma, dette ***argomenti*** o ***parametri formali*** del sottoprogramma. Esse sono destinate ad ospitare i dati di ingresso e/o di uscita del sottoprogramma.
- Con la istruzione di chiamata, il programma chiamante fornisce al sottoprogramma una lista di ***parametri effettivi***, costituiti dai valori effettivi di ingresso su cui il sottoprogramma deve operare e dalle variabili del programma chiamante in cui i valori di uscita del sottoprogramma dovranno essere memorizzati.
- La corrispondenza tra parametri effettivi e formali è fissata per ordine.

Sottoprogrammi: proprietà

- L'esecuzione del chiamante viene sospesa all'atto della chiamata e ripresa al ritorno dal sottoprogramma
- Prima dell'esecuzione del sottoprogramma, i suoi parametri formali (M ed N) vengono inizializzati con i valori dei parametri effettivi (X e Y) forniti dal chiamante.
- Al ritorno dal sottoprogramma, il chiamante vede modificate alcune sue variabili (R)



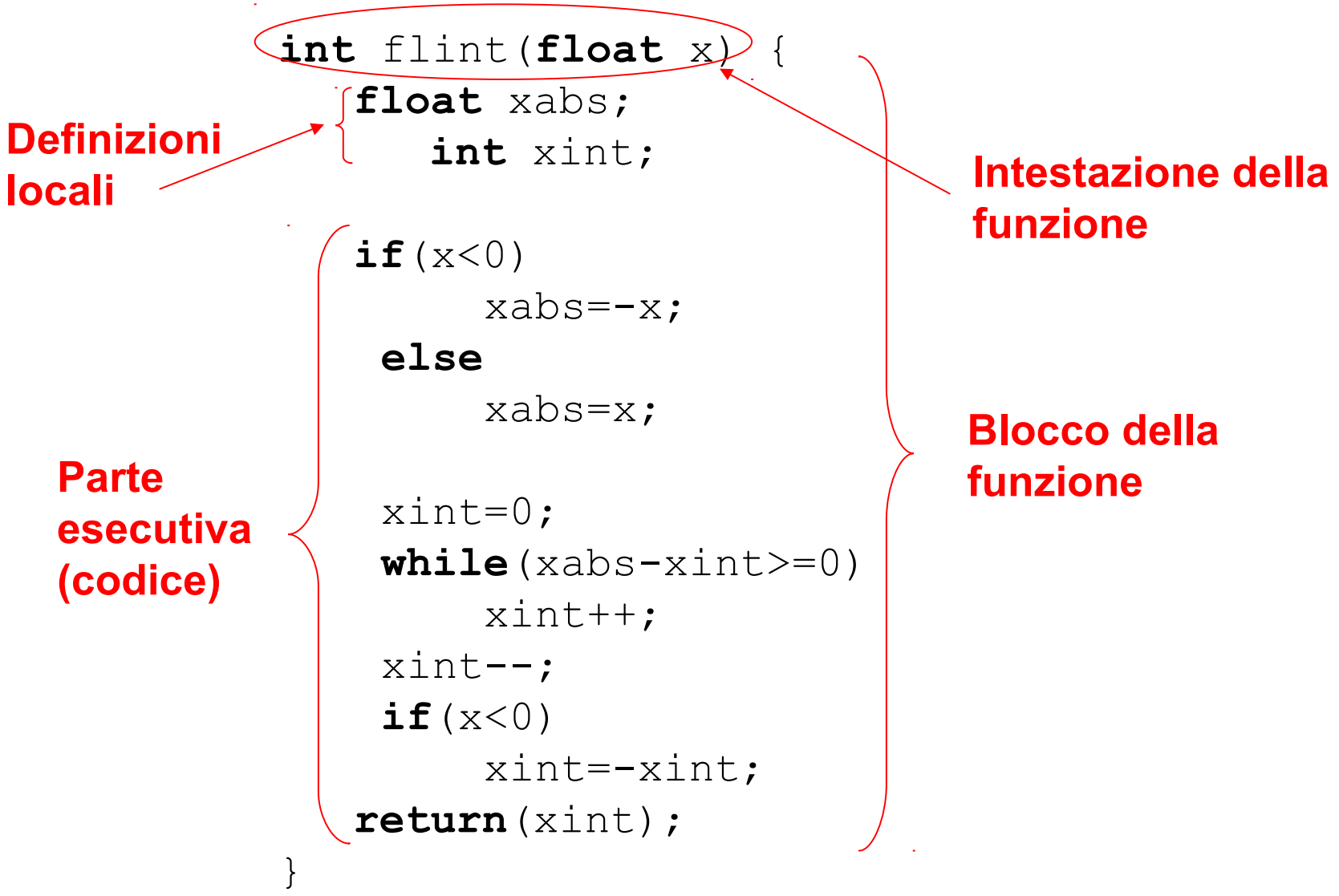
Sottoprogrammi

- Nella maggior parte dei linguaggi sono presenti due tipi fondamentali di sottoprogrammi:
  - le **funzioni**
  - le **procedure**

Funzioni

- Una funzione è un particolare sottoprogramma che produce in uscita un valore il quale non è assegnato ad uno dei parametri, ma viene attribuito al nome stesso della funzione.
- La chiamata della funzione non avviene mediante una esplicita istruzione di chiamata, ma inserendo il nome della funzione seguito dalla lista dei parametri effettivi direttamente in altre istruzioni (p.es. in istruzioni di assegnazione).
- Alcune funzioni sono già disponibili all'interno di librerie fornite con il compilatore e quindi non richiedono una definizione esplicita da parte dell'utente. Es.: sqrt(x)

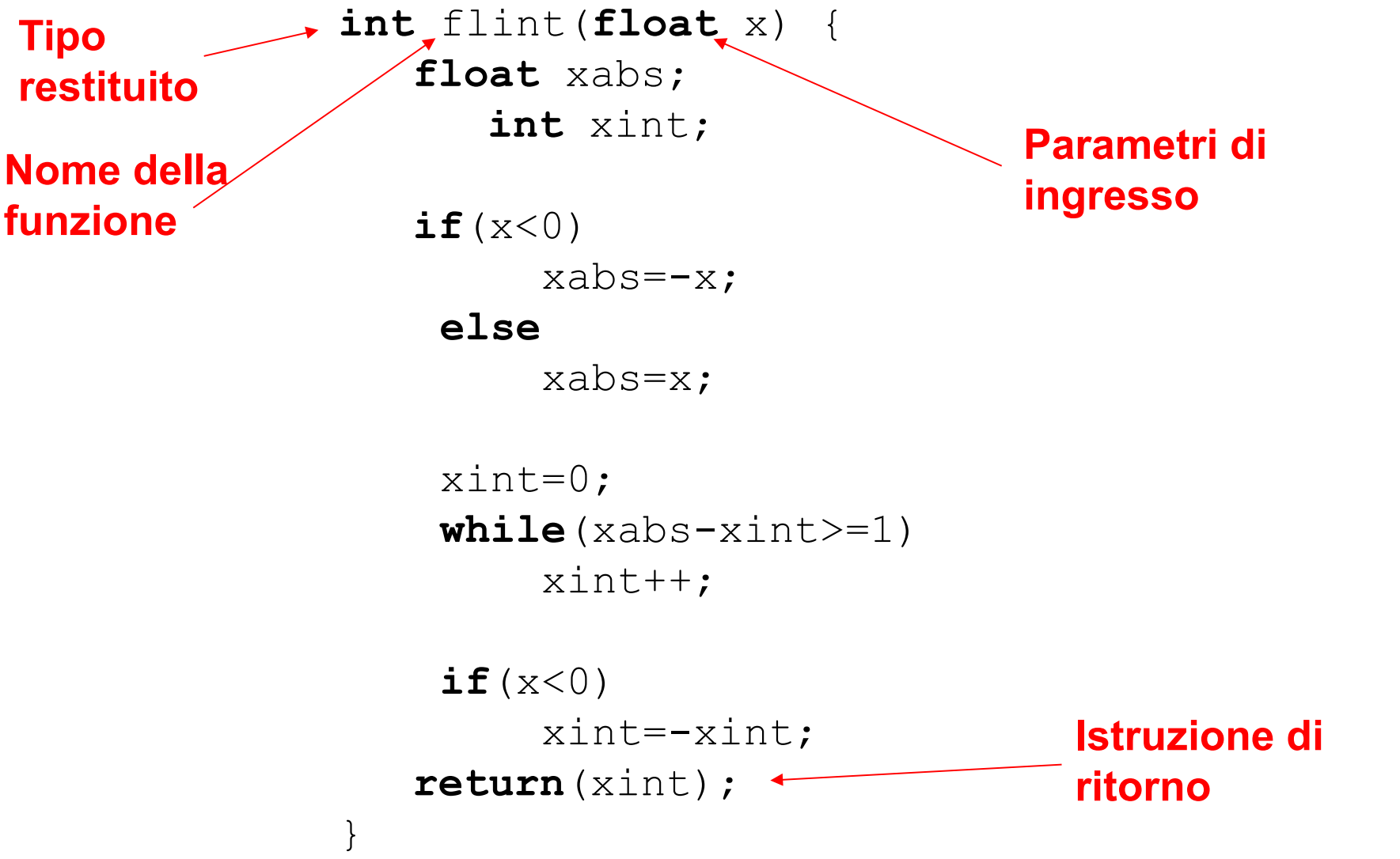
Struttura di una funzione



Struttura di una funzione

- Sono riconoscibili due parti:
  - l'**intestazione**
  - il **blocco**
- L'intestazione della funzione riporta le informazioni principali relative alla funzione: nome, tipo restituito, parametri di ingresso.
- Il blocco è costituito da:
  - una parte dichiarativa (variabili locali)
  - una parte esecutiva (istruzioni)

Struttura di una funzione




```
int flint(float x) {
    float xabs;
       int xint;

    if(x<0)
         xabs=-x;
     else
         xabs=x;

     xint=0;
     while(xabs-xint>=1)
         xint++;

     if(x<0)
         xint=-xint;
    return(xint);
}
```

Nome della funzione

- Il nome identifica univocamente la funzione
- I nomi delle funzioni hanno gli stessi vincoli dei nomi delle variabili. Il nome deve cominciare con una lettera che può essere seguita da una combinazione di lettere, cifre, underscore.

Parte esecutiva

- La parte esecutiva contiene l'insieme di istruzioni che implementa l'operazione che la funzione deve realizzare.
- Le istruzioni lavorano sull'insieme formato dai parametri di ingresso e dalle variabili definite all'interno.
- Le istruzioni possono essere costrutti di qualunque tipo (calcolo e assegnazione, I/O, selezioni, cicli, commenti, linee vuote, chiamate di altre funzioni).
- Al termine c'è una istruzione di **`return`** il cui scopo è di:
  - terminare l'esecuzione della funzione;
  - restituire il valore tra parentesi come valore della funzione.

Chiamata di una funzione

```
int main() {
    float a,b;
    int ai,bi;

    cout << "a: "; cin >> a;
    cout << "b: "; cin >> b;

    ai=flint(a);
    bi=flint(b);

    cout<<"La parte intera di "<<a<<" e' "<<ai<<endl;
    cout<<"La parte intera di "<<b<<" e' "<<bi<<endl;

    cout<<"Somma delle parti intere: "<<ai+bi<<endl;
    cout<<"Parte intera della somma : "<<flint(a+b)<<endl;

    return(EXIT_SUCCESS);
}
```

Chiamata di una funzione

- La chiamata di una funzione avviene all'interno di una espressione in cui compare il nome della funzione seguito dai parametri effettivi tra ().
- Nell'espressione la funzione partecipa fornendo un valore del tipo restituito.
- La valutazione dell'espressione avvia l'esecuzione della funzione.
- Come parametri effettivi possono essere presenti anche delle espressioni (ovviamente, del tipo assegnato al parametro formale corrispondente).

Esecuzione di una funzione

1. Nel programma chiamante, la valutazione di un'espressione attiva la chiamata della funzione;
2. All'atto della chiamata, i parametri effettivi vengono valutati ed assegnati ai rispettivi parametri formali;
3. L'esecuzione del programma chiamante viene sospesa e il controllo viene ceduto al sottoprogramma;
4. Inizia l'esecuzione della funzione: i parametri formali sono inizializzati con i valori dei parametri effettivi;
5. Le istruzioni della funzione sono eseguite;
6. Come ultima istruzione viene eseguito un **`return`** che fa terminare l'esecuzione della funzione e restituire il controllo al programma chiamante;
7. Continua la valutazione dell'espressione nel programma chiamante sostituendo al nome della funzione il valore restituito.

Organizzazione del programma

```
#include<iostream>
using namespace std;

int flint(float x) {
     float xabs;
     int xint;

     if(x<0)
        xabs=-x;
     else
        xabs=x;
     xint=0;
     while(xabs-xint>=1)
        xint++;

     if(x<0)
        xint=-xint;
     return(xint);
}

int main() {
     float a,b;
     int ai,bi;

     cout << "a: "; cin >> a;
     cout << "b: "; cin >> b;
     ai=flint(a);
     bi=flint(b);
     cout<<"La parte intera di "<<a<<" e' "<<aint<<endl;
     cout<<"La parte intera di "<<b<<" e' "<<bint<<endl;
     cout<<"Somma delle parti intere: "<<ai+bi<<endl;
     cout<<"Parte intera della somma : "<<flint(a+b)<<endl;
     return(EXIT_SUCCESS);
}
```

La definizione della funzione `flint` deve precedere il `main`

La funzione main

- Anche il blocco identificato da **`main`** è una funzione a tutti gli effetti.
- Particolarità di **`main`**:
  - viene chiamata dal Sistema Operativo all'atto dell'esecuzione del programma;
  - il flusso di dati avviene con il S.O., sia per i parametri effettivi in ingresso, sia per il valore restituito da **`return`**.

Prototipo di una funzione

- Come per le variabili, anche le funzioni devono essere definite prima di essere usate.
- Nel caso ci siano più funzioni, il main andrebbe in fondo al file sorgente, rendendo meno leggibile il codice.
- In effetti, per poterle gestire correttamente, il compilatore ha bisogno solo delle informazioni presenti nell'intestazione della funzione.
- E' quindi possibile anticipare al main solo le intestazioni delle funzioni (**prototipi**) e inserire dopo il main le definizioni delle funzioni.
- Il prototipo è formato dall'intestazione della funzione terminato con ';': `int flint(float x);`

Organizzazione del programma con i prototipi

```
#include<iostream>
using namespace std;

int flint(float x);

int main() {
     float a,b;
     int ai,bi;

     cout << "a: "; cin >> a;
     cout << "b: "; cin >> b;
     ai=flint(a);
     bi=flint(b);
     cout<<"La parte intera di "<<a<<" e' "<<aint<<endl;
     cout<<"La parte intera di "<<b<<" e' "<<bint<<endl;
     cout<<"Somma delle parti intere: "<<ai+bi<<endl;
     cout<<"Parte intera della somma : "<<flint(a+b)<<endl;
     return(EXIT_SUCCESS);
}
int flint(float x) {
     float xabs;
     int xint;

    if(x<0)
        xabs=-x;
    else
        xabs=x;
    xint=0;
    while(xabs-xint>=1)
        xint++;

    if(x<0)
        xint=-xint;
    return(xint);
}
```

Prototipo della funzione

```
int flint(float x);
```

Definizione della funzione

Procedure

- A volte le operazioni da implementare non richiedono la *produzione di un valore*, ma l'*esecuzione di un'azione*, come la stampa di valori o la modifica di variabili.
- In questi casi si può utilizzare un tipo diverso di sottoprogramma: la **procedura**.
- La chiamata della procedura avviene mediante una esplicita istruzione di chiamata, costituita dal nome della procedura seguito dalla lista dei parametri effettivi tra ().

Realizzazione di procedure in C++

- In C++ una procedura viene definita come una funzione che non restituisce valori.
- Questo si realizza tramite il tipo **`void`**.
- Può essere presente l'istruzione **`return`**, che in questo caso ha solo la funzione di terminare l'esecuzione della funzione.

```
void stampa3int(int a,int b,int c) {
   int s;

   cout<<"Primo valore: "<<a<<endl;
   cout<<"Secondo valore: "<<b<<endl;
   cout<<"Terzo valore: "<<c<<endl;
   s=a+b+c;
   cout<<"Somma: "<<s<<endl;
   return;
}
```

Chiamata di una procedura

- La chiamata di una procedura avviene con un'istruzione apposita costituita dal nome della procedura seguito dalla lista dei parametri effettivi tra ().
- L'attivazione viene realizzata nelle stesse modalità viste per la funzione.

```
void stampa3int(int,int,int);

int main() {
    int p,q,r;

    p=2; q=12; r=6;
    stampa3int(p,q,r);

    return(EXIT_SUCCESS);
}
```

Tecniche di scambio di parametri

- Esistono due tecniche principali per lo scambio di parametri:
  - **Scambio per valore**
  - **Scambio per riferimento**

Scambio per valore

- Nello scambio per valore (o *by value*), il valore del parametro effettivo viene copiato nel parametro formale.
- Il parametro formale costituisce quindi una copia locale del parametro effettivo.
- Ogni modifica fatta sul parametro formale non si riflette sul parametro effettivo.

Scambio per valore

- Tutti gli esempi visti finora hanno utilizzato lo scambio per valore

```
void stampa3int(int a,int b,int c) {
     int s;

     cout<<"Primo valore: "<<a<<endl;
     cout<<"Secondo valore: "<<b<<endl;
     cout<<"Terzo valore: "<<c<<endl;
     s=a+b+c;
     cout<<"Somma: "<<s<<endl;
     return;
}
```

```
int flint(float x) {
     float xabs;
     int xint;

     if(x<0)
          xabs=-x;
     else
          xabs=x;

     xint=0;
     while(xabs-xint>=1)
          xint++;

     if(x<0)
          xint=-xint;
     return(xint);
}
```

Scambio per riferimento

- Nello scambio per riferimento, al parametro formale viene assegnato l'indirizzo del parametro effettivo.
- In questo modo, al sottoprogramma è possibile accedere al registro che ospita il parametro effettivo e fare delle modifiche che saranno poi visibili al programma chiamante.
- In altre parole, qualunque modifica effettuata sul parametro formale avrà effetto sul parametro effettivo corrispondente .

Scambio per riferimento

Lo scambio per riferimento (o *by reference*) lo si definisce anteponendo un & al nome del parametro formale.

```
#include <iostream>
using namespace std;

void raddoppia(int& a, int& b, int& c){
    a*=2;
    b*=2;
    c*=2;
}

int main() {
    int x,y,z;

    x=0; y=1; z=2;
    raddoppia(x,y,z);
    cout<<x<<" "<<y<<" "<<z<<endl;
    return(EXIT_SUCCESS);
}
```

… già visto da qualche parte ?

Scambio per riferimento

- Lo scambio per riferimento fornisce un modo efficace per realizzare una funzione che deve restituire più di un valore

```
void precsucc(int x, int& prec, int& succ){
   prec=x-1;
   succ=x+1;
}
```

Problemi

- Scrivere un sottoprogramma che effettua lo swap di due variabili.
- Scrivere un sottoprogramma che restituisce parte intera e parte frazionaria di un numero reale.

Array come parametri

- Gli array sono passati unicamente per riferimento (perché ?).
- Come parametro formale, un array si indica con tipo, identificatore e parentesi quadre: **`int vet[]`**. Non si inserisce l'&.
- Come parametro effettivo va fornito il nome dell'array, senza specificare altro.

Array come parametri

```
void stamparray (int vet[], int num) {
  for (int i=0; i<num; i++)
    cout << vet[i] << " ";
  cout << "\n";
}

int main ()
{
  int vet1[] = {5, 10, 15};
  int vet2[] = {2, 4, 6, 8, 10};

  stamparray(vet1,3);
  stamparray(vet2,5);

  return (EXIT_SUCCESS);
}
```

Array come parametri: errori frequenti

```
void stamparray (int vet[], int num) {
  for (int i=0; i<num; i++)
    cout << vet[i] << " ";
  cout << "\n";
}

int main ()
{
  int vet1[] = {5, 10, 15};
  int vet2[] = {2, 4, 6, 8, 10};

  stamparray(vet1[3],3);
  stamparray(vet2,5);

  return (EXIT_SUCCESS);
}
```

ACHTUNG !!

Si passa un intero e non un array

korrekt !!

Il ruolo dei sottoprogrammi nella progettazione dei programmi

- L'uso dei sottoprogrammi permette di organizzare in modo particolarmente efficace la progettazione di un programma. Infatti, con l'uso dei sottoprogrammi è possibile:
  - articolare il programma complessivo in più sottoprogrammi, ognuno dei quali realizza un compito preciso e limitato, rendendo più semplice la comprensione e la manutenzione del programma
  - progettare, codificare e verificare ad uno ad uno i singoli sottoprogrammi
  - riutilizzare in un programma diverso un sottoprogramma già codificato e verificato
  - limitare al minimo gli errori dovuti ad interazioni non previste tra parti diverse del programma (effetti collaterali)

Modularità e compilazione separata

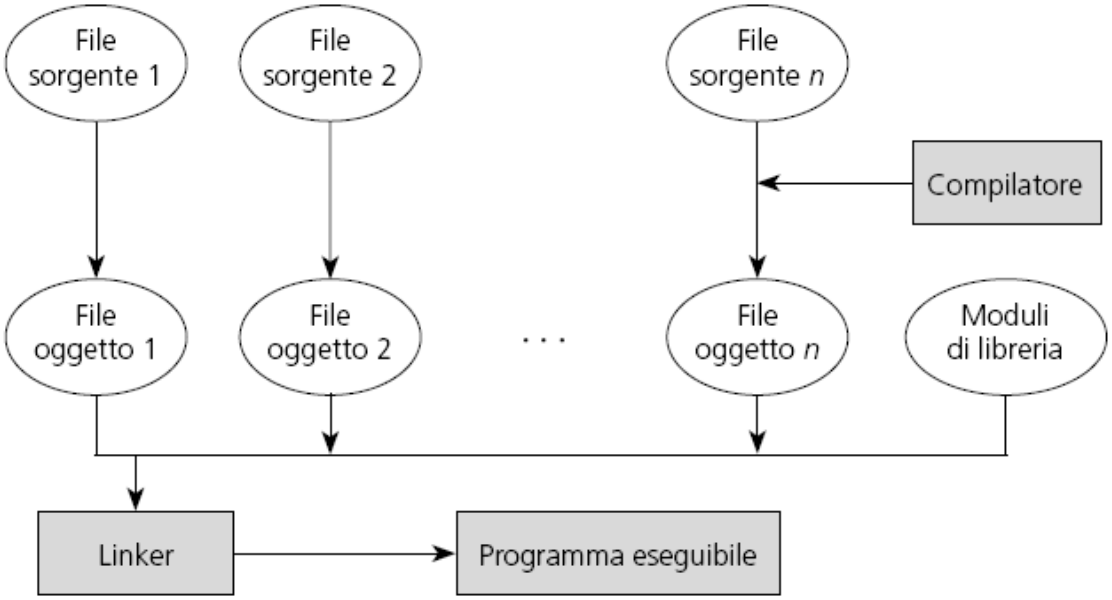
- Questi aspetti permettono di creare programmi caratterizzati da **modularità**.
- Il programma complessivo risulta da un insieme di sottoprogrammi (o moduli) che:
  - Risolvono ognuno un proprio compito specifico e ben definito;
  - Sono progettati, implementati e verificati autonomamente rispetto al resto del programma con il solo vincolo di rispettare l'interfaccia verso il resto del programma (nome della funzione, compito, flusso dei dati).

Modularità e compilazione separata

- Questa caratteristica è resa ancor più vantaggiosa dal fatto che il C++ permette la **compilazione separata.**
- Le funzioni che costituiscono un programma non devono necessariamente risiedere in un unico file, ma possono essere distribuite su più file sorgente.
- Esempio: un programma di calcolo strutturale può essere diviso in
  - main.cc - contiene la f. main e altre ff. specifiche del programma
  - inpout.cc - contiene le ff. di I/O
  - calcol.cc - contiene le ff. di calcolo vero e proprio

Modularità e compilazione separata

- All'atto della compilazione, tutti i moduli sono compilati separatamente.
- Il programma eseguibile viene costruito con una successiva fase di **collegamento** (**linking**).



Modularità e compilazione separata

- La compilazione separata presenta alcuni vantaggi:
  - Eventuali modifiche ad un modulo richiedono solo la ricompilazione di quel modulo e il linking ☺
  - Eventuali errori o richieste di modifica sono facilmente identificabili e limitate al solo modulo interessato ☺ ☺
  - Un modulo realizzato e verificato per un programma può essere riutilizzato facilmente per la costruzione di un nuovo programma ☺ ☺ ☺ ☺

Compilazione separata

- Per organizzare un programma su file sorgenti diversi è necessario che in ogni modulo siano definite o dichiarate le funzioni che vengono utilizzate
- Questo significa che se il modulo A utilizza la funzione `funz` definita nel modulo B, il modulo A deve contenere il prototipo della funzione `funz`.

Compilazione separata

- Quando si definisce un modulo (es. calcol.cpp) tipicamente si definisce anche un file header (es. calcol.h) che contiene i prototipi delle funzioni definite nel modulo.
- In questo modo, per i moduli che useranno le funzioni di calcol.cpp sarà sufficiente includere il relativo file header:

```
#include "calcol.h"
```

Esempio: main.cpp

```
#include <iostream>
#include "calcol.h"
using namespace std;

int main() {
    const float prec=1e-8;
    float x0,y0,x,y;
    float d;

    cout<<"Fornire le coordinate del punto P0\n";
    cout<<"x0: "; cin >>x0;
    cout<<"y0: "; cin >>y0;

    cout<<"\nFornire le coordinate del punto P\n";
    cout<<"x: "; cin >>x;
    cout<<"y: "; cin >>y;

    d=radq((x-x0)*(x-x0)+(y-y0)*(y-y0),prec);

    cout<<"\nPunto P a distanza "<<d<<endl;
    return (EXIT_SUCCESS);
}
```

Esempio: calcol.cpp e calcol.h

calcol.cpp

```
float assol(float x){
    if(x<0)
        return(-x);
    else
        return(x) ;
}


float radq(float x,float p){
    float yn,yv;

    yn=1.0;

    do{
        yv=yn;
        yn=.5*(yv+x/yv);
    } while (assol(yn-yv)>=p);

    return(yn);
}
```

calcol.h

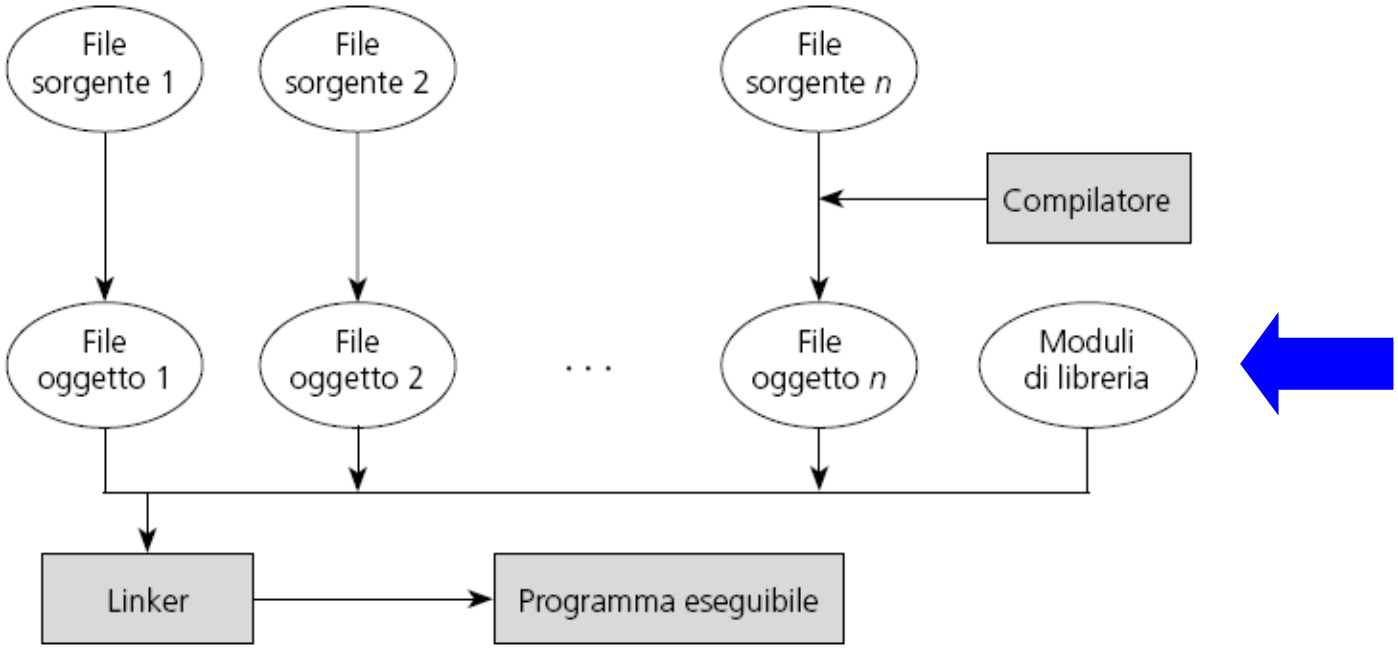
```
float radq(float x,float p);
float assol(float x);
```

Il project

- I file (.cpp e .h) che costituiscono il programma sono organizzati nell'IDE in una struttura definita **project**.
- Insieme ai file source e header (prodotti dal programmatore), l'IDE produce e aggiunge al project un file (**make file**) che contiene le informazioni per costruire l'eseguibile.
- Ogni volta che si richiede la produzione dell'eseguibile, un programma apposito (**make**) esegue le opportune operazioni (compilazioni, link) per lo scopo.

Librerie di funzioni

- Molte funzioni utili sono già rese disponibili dal compilatore in file (**libreria**) che le raccolgono secondo la tipologia.
- Alcune tipologie di funzioni di libreria:
  - **cmath** libreria di funzioni matematiche
  - **cstdio** libreria di funzioni di Input/Output
  - **cstdlib** libreria di funzioni di uso generale
  - **cstring** libreria di funzioni per gestione stringhe
  - **ctime** libreria di funzioni per gestione ora e data
- Per utilizzare le funzioni di una certa libreria va inserita la direttiva **`include`**.
  Es.: **`#include <cmath>`**

Librerie di funzioni

- All'atto della produzione dell'eseguibile, le funzioni di libreria sono disponibili in forma compilata.