Funzioni

Parte I
Definizione,
Dichiarazione o Prototipo,
Chiamata

Passaggio dei parametri

- Per “passaggio dei parametri” si intende l’associazione fra parametri attuali e parametri formali che avviene al momento della chiamata di una funzione (o procedura).

- ***L'unico meccanismo adottato in C, si chiama PASSAGGIO PER VALORE***

- Come vedremo, in C++ disponiamo anche del passaggio per **riferimento**

Blocco

- Sequenza di definizioni e istruzioni racchiuse tra parentesi graffe

```
{
        <dichiarazione1 o istruzione1>
        <dichiarazione2 o istruzione2>
        ...
}
```

- Avevamo definito l'istruzione composta come caso particolare di blocco
- In C, parte dichiarativa ed esecutiva devono essere separate:

```
{
        <dichiarazioni>
        <istruzioni>
        ...
}
```

Indice Lezioni Linguaggio C/C++

- √ Tipi di dato primitivi (focus su **int**)
- √ Dichiarazioni e Definizioni
- √ Espressioni
- √ Istruzioni
- • **Funzioni**

Concetto di funzione

- **L'astrazione di *funzione* è presente in tutti i linguaggi di programmazione di alto livello.**

- Una **funzione** è un *componente software* che cattura l'astrazione matematica di funzione:

$$f : A \times B \times ... \times Q \rightarrow S$$

  - **molti ingressi possibili** (corrispondenti ai valori su cui operare)
  - **una sola uscita** (corrispondente al risultato)

- Altri linguaggi (**ma non il C/C++!**) introducono anche l'astrazione di **procedura** che cattura l'astrazione dell'esecuzione di un insieme di azioni, senza ritornare risultati (**comunque, è IMMEDIATO DA EMULARE IN C/C++**)

Vantaggi delle funzioni

- Testo del programma *suddiviso in unità significative*
- Testo di ogni unità *più breve*
  - *minore probabilità di errori*
  - *migliore verificabilità*
- Riutilizzo di codice
- Migliore leggibilità
- Sviluppo top-down del software

I tre elementi fondamentali per comprendere le funzioni

- **Definizione e dichiarazione della funzione**
- **Uso della funzione (chiamata)**
- **Esecuzione della funzione (record di attivazione)**
  - **Si vedrà in seguito**

Definizione di funzione

- Una definizione di funzione è costituita da una *intestazione* e da un *corpo,* definito mediante un *blocco*

  <def-funzione> ::= <intestazione-funzione> <blocco>

  <intestazione-funzione> ::=
  <nomeTipo> <nomeFunzione> (<lista-parametri>)

  <lista-parametri> ::=
  void | <def-parametro> { , <def-parametro> }

  <def-parametro> ::= <nomeTipo> <identificatore>

Definizione di funzione (*cont.*)

L'intestazione specifica nell'ordine:

- **Tipo del risultato** (**void** se non c'è risultato); corrisponde alla *procedura* di altri linguaggi
- **Nome della funzione**
- Lista dei **parametri formali** (***in ingresso***)
  - **void** se la lista è vuota (non ci sono parametri)
    - in questo caso può anche essere omessa la lista stessa
  - una sequenza di definizioni di parametri, se la lista non è vuota

Definizione di funzione: esempi

int — Tipo risultato

fattoriale — Nome della funzione

(int n) — Lista dei parametri formali

float — Tipo risultato

somma3 — Nome della funzione

(float x, float y, float z) — Lista dei parametri formali

void — Tipo risultato

stampa_n_volte — Nome della funzione

(int n) — Lista dei parametri formali

void — Tipo risultato

stampa_2_volte — Nome della funzione

(void) — Lista dei parametri formali vuota (*void* può essere omesso).
LE PARENTESI SERVONO SEMPRE

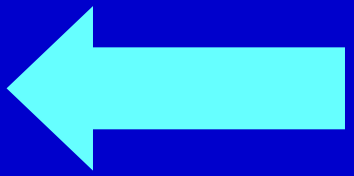
Definizione di funzione (*cont.*)

Il corpo della funzione è costituito da un blocco:

- Delimitato da parentesi graffe **{ }**
- Dichiarazione/definizione di variabili **locali** al blocco di funzione
- Lista di istruzioni

I tre elementi fondamentali per comprendere le funzioni

- Definizione e dichiarazione della funzione
- Uso della funzione (chiamata)
- Esecuzione della funzione (record di attivazione)



Chiamata di funzione

- **Una chiamata di funzione è costituita dal nome della funzione e dalla lista dei parametri attuali tra parentesi tonde:**

  <chiam-funzione> ::=

  <nomeFunzione> (<lista-parametri-attuali>)

- **La chiamata di una funzione è una espressione**

Lista dei parametri formali e attuali

- **Parametri formali: argomenti dichiarati nella definizione di funzione**

  →Devono essere **variabili**

- **Parametri attuali: argomenti inseriti al momento della chiamata di funzione**

  →**Espressioni separate da virgole**:
  - *costanti*
  - *variabili*
  - *espressioni aritmetiche*
  - …

Dinamica parametri formali e attuali

- **La corrispondenza tra parametri formali e attuali è sia posizionale sia di tipo**. Ovvero si presume che la lista dei parametri formali e la lista dei parametri attuali abbia lo stesso numero e tipo di elementi (usare il **casting** se necessario → *si vedrà in seguito*)

- **I nomi dei parametri attuali e formali non hanno importanza**. Possono essere gli stessi o diversi. L'importante è la posizione ed il valore che assume un parametro attuale al momento della chiamata

- Non appena l'ambiente della funzione chiamata viene attivato, i parametri formali vengono **definiti** (**come variabili locali all'ambiente della funzione**) ed **inizializzati** al valore del corrispondente parametro attuale

Proviamo ...

- ... a scrivere, compilare ed eseguire un programma in cui
  - Si definisce una funzione di nome *fun*, che
    - non prende alcun parametro in ingresso
    - non ritorna alcun valore
    - Stampa sullo schermo un messaggio
  - Si invoca tale funzione all'interno della funzione *main* e si esce

Soluzione

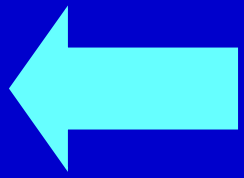
```
#include <iostream>

using namespace std ;

void fun()
{
    cout<<"Saluti dalla funzione fun"<<endl ;
}

int main()
{
    fun() ;
    return 0 ;
}
```

I tre elementi fondamentali per comprendere le funzioni

- Definizione e dichiarazione della funzione 
- Uso della funzione (chiamata)
- Esecuzione della funzione (record di attivazione)
  - Si vedrà in seguito

Collocazione definizioni di funzione

Una funzione <u>non può</u> essere definita <u>all'interno</u> di un'altra funzione

```
int main()
{
    void fun()
    {
        cout<<"Saluti dalla funzione fun"<<endl ;
    }

    fun() ;
    return 0 ;
}
```

Ancora sui parametri formali 1/2

Un parametro formale non è altro che una variabile locale alla funzione

Definizioni (quasi) equivalenti di una variabile locale i

```
void fun(int i)
{
    i++ ;
    cout<<i ;
}
```

Istruzione legale

```
void fun(int j)
{
    int i ;
    i++ ;
    cout<<i ;
}
```

Ancora sui parametri formali 2/2

Unica differenza (ma molto importante)

i è inizializzata col valore del parametro attuale

```
void fun(int i)
{
        i++ ;
        cout<<i ;
}
```

```
void fun(int j)
{
        int i ;
        i++ ;
        cout<<i ;
}
```

i ha un valore iniziale casuale

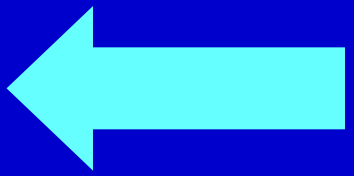
Istruzione *return*

- Viene usata per *restituire il controllo all'ambiente chiamante* e nel contempo *restituirgli il valore calcolato dalla funzione* (risultato)
  $\rightarrow$ Eventuali istruzioni della funzione successive all'esecuzione del ***return*** non saranno eseguite!
- Se la funzione ritorna un risultato, istruzione
  return (<espressione>)
  oppure semplicemente
  return <espressione>

  <espressione> deve essere del **tipo di ritorno** specificato nell'intestazione della funzione

I tre elementi fondamentali per comprendere le funzioni

- Definizione e dichiarazione della funzione
- Uso della funzione (chiamata)
- Esecuzione della funzione (record di attivazione)



Esecuzione di istruzioni e valore di ritorno

- **La chiamata di una funzione è una espressione**
- **Due casi:**
  - **Funzioni *void*:** la chiamata di funzione va considerata solamente come un insieme di istruzioni
  - **Funzioni che ritornano un valore:** la chiamata di funzione va considerata come un'espressione il cui valore di ritorno è di tipo uguale al tipo di ritorno della funzione, e può essere utilizzato come fattore all'interno di espressioni composte
    Es. Supponendo che *fun()* ritorni un valore di tipo *int*:
    int b = fun(a) + 1 ;

Esercizi

- *funz_max.cc*
- *funz_fattoriale.cc*

Definizione di funzione: esempio

```
int fattoriale (int n)
```

- **int** — Tipo ritorno
- **fattoriale** — Nome della funzione
- **(int n)** — Lista dei parametri formali

INTESTAZIONE

```
{
    int i, fatt;

    if (n==0) return (1);
        else fatt=1;

    for (i=1; i<=n; i++)
        fatt = fatt*i;

    return fatt;
}
```

- `int i, fatt;` — **Definizioni locali alla funzione**
- `return fatt;` — **Restituzione del valore**

CORPO

Progetto di una funzione

- **Scegliere un nome significativo per la funzione**

- **La funzione deve ricevere qualche dato dalla funzione chiamante?**

  Se sì, elencare ed identificare tutti i tipi di dato da passare alla funzione (questa è chiamata la **lista dei parametri** o la **lista degli argomenti**).

  Se no, la lista dei parametri è **void**

- **La funzione deve restituire un dato dalla funzione chiamante?**

  Se sì, identificare il tipo di dato → corrisponderà al tipo di ritorno della funzione

  Se no, il tipo della funzione è **void**

Dichiarazione della funzione

- Nel linguaggio C/C++ si possono utilizzare solo le funzioni che sono state precedentemente “dichiarate”
  - Finora abbiamo visto solo le definizioni
  - Sono un caso di dichiarazione: quello in cui si definisce anche il corpo della funzione stessa
    - Comportano l'allocazione di spazio in memoria per le variabili e le istruzioni
- Ci sono diverse situazioni in cui è utile o necessario utilizzare una funzione definita altrove o successivamente
- Ad esempio, nel caso di ...

Chiamate incrociate

```
void fun1()
{
    cout<<"fun1"<<endl ;
    fun2();
}

void fun2()
{
    cout<<"fun2"<<endl ;
    fun1();
}
```

- *Ancora non è stata dichiarata!*
- *Invertire l'ordine di definizione delle funzioni non risolve il problema*

Dichiarazione della funzione

- Una **dichiarazione** (senza definizione) o **prototipo** è costituita dalla sola intestazione di una funzione

<dich-funzione> ::= <intestazione-funzione> ;

<intestazione-funzione> ::=
<nomeTipo> <nomeFunzione> (<lista-parametri>)

<lista-parametri> ::=
void | <dich-parametro> { , <dich-parametro> }

<dich-parametro> ::= <nomeTipo> [<identificatore>]

Opzionale !

Soluzione chiamate incrociate ...

```
void fun2() ;

void fun1()
{
    cout<<”fun1”<<endl ;
    fun2();
}

void fun2()
{
    cout<<”fun2”<<endl ;
    fun1();
}
```

Esempi di prototipo

```
int  fattoriale (int);   /* della funzione precedente */
…
int  fattoriale   (int n)
  {
    int i, fatt=1;
    for (i=1; i<=n; i++)
          fatt = fatt*i;
    return(fatt);
  }
```

```
float  max (float, float, float) ;   /* calcola il max di 3 float */
```

Prototipi e definizioni

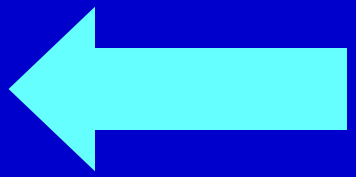
- Il prototipo:
  - è un puro “avviso ai naviganti”
  - ***non produce alcun byte di codice***
  - quindi, se lo si ripete non succede niente di male (basta che non ci siano due dichiarazioni in contraddizione)
  - può comparire anche dentro un'altra funzione

- La definizione, invece:
  - ***contiene il vero codice della funzione***
  - **non può essere duplicata!!**
    (altrimenti ci sarebbero due codici per la stessa funzione)
  - il nome dei parametri formali, *non necessario* in un *prototipo*, è più importante in una *definizione*

QUINDI: il prototipo di una funzione può comparire più volte, ma la funzione deve essere *definita* una sola volta.

I tre elementi fondamentali per comprendere le funzioni

- Definizione e dichiarazione della funzione
- Uso della funzione (chiamata)
- Esecuzione della funzione (record di attivazione)



Sintesi

- Nel testo di un programma:
  - Il **prototipo** di funzione può comparire una o più volte, anche dentro un'altra funzione, ma sempre prima dell'eventuale chiamata di funzione
  - La **definizione** di funzione può comparire una sola volta e non dentro altre funzioni
  - La **chiamata** di funzione può comparire una o più volte, ma sempre dopo il prototipo o la definizione, e dentro una funzione

Esempio: programma (errato)

```
#include <iostream>

using namespace std ;

int main()
{
      int a, b;
      cin>>a>>b ;
      cout<<"Il massimo tra "<<a<<" e "<<b<<" e' "<<max(a,b)<<endl ;
      return 0 ;
}

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

Esempio: programma (corretto) 1/2

```
#include <iostream>

using namespace std ;

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

int main()
{
      int a, b;
      cin>>a>>b ;
      cout<<"Il massimo tra "<<a<<" e "<<b<<" e' "<<max(a,b)<<endl
         ;
      return 0 ;
}
```

Esempio: programma (corretto) 2/2

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

int max(int, int) ;

int main()
{
      int a, b;
      cin>>a>>b ;
      cout<<"Il massimo tra "<<a<<" e "<<b<<" e' "<<max(a,b)<<endl ;
      return 0 ;
}

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

Tipo dei parametri
Ad esempio, scrivere
int max(int a, int c) ;
sarebbe stato equivalente

Parametri attuali
(*espressioni*)

Parametri formali
(*variabili*)

Esercizio 1 (*Specifica*)

- **Scrivere una funzione che verifichi se un numero naturale fornito dal *main()* è primo**
- **La funzione deve restituire *false* se il numero non è primo, *true* se il numero è primo**

Esercizio 1 (*prime idee*)

- Un numero è primo se è divisibile solo per 1 e per se stesso. Quindi, occorre provare a dividere ***N*** per tutti i numeri $2 \leq i \leq N-1$. Se nessun numero *i* risulta essere un divisore di *N*, allora *N* è primo. **Funziona? Sì, ma si può fare meglio**
- IDEA 1: Poiché i numeri pari non sono primi, eliminiamo subito la possibilità che *N* sia pari
- IDEA 2: Non c'è bisogno di provare tutti i numeri fino a N-1, ma è sufficiente provare a dividere ***N*** per tutti i numeri dispari $3 \leq i \leq (N/2)$: se nessun numero *i* risulta essere un divisore di *N*, allora *N* è primo
- IDEA 3: Non c'è bisogno di provare tutti i numeri fino a N/2, ma è sufficiente provare a dividere ***N*** per tutti i numeri dispari $3 \leq i \leq \text{sqrt}(N)$: se nessun numero *i* risulta essere un divisore di *N*, allora *N* è primo

Esercizio 1 (*Algoritmo*)

- Se *N* è 1, 2 o 3, allora senz'altro *N* è un numero primo
- Altrimenti, se è un numero pari, certamente *N* non è primo
- Se così non è (quindi se *N* è dispari e >3), occorre tentare tutti i possibili divisori dispari da 3 in avanti, fino a **sqrt**(*N*)

Esercizio 1 (*Rappresentazione informazioni*)

- Nome della funzione: **isPrime**
- Elenco dei parametri formali (nome e tipo): **int n**
- Tipo (di ritorno) della funzione: **bool** (*valore booleano* dove: false → non è numero primo, true → è numero primo

- All'interno della funzione:
  - Serve una variabile **int max_div** che contenga la parte intera della radice quadrata del numero
  - Serve, poi una variabile ausiliaria (**int i**) come indice per andare da 3 a max_div

- Occorre includere **<cmath>** per utilizzare la funzione **sqrt()**

Esercizio 1 (*Programma*)

```
#include <iostream>


bool isPrime(int n)
 {
   int max_div, i;

   if (n>=1 && n<=3) return true;     /* 1,2,3 → sì */

   if (n%2==0) return false;          /* no perché numeri pari */

   max_div = (int) sqrt(n);
   for(i=3; i<=max_div; i=i+2)
       if (n%i==0) return false;      /* no perché è stato trovato
                                         un divisore */
  return true;
 }
```

Esercizio 2 (*Specifica*)

- **Scrivere una funzione radice che calcoli la radice quadrata (intera) di un valore naturale *N*.**

Esercizio 2 (*Idea e Algoritmo*)

int **radice**(int n);

< restituisce il massimo intero **x** tale che **x*x ≤ n** >

- Considera un naturale dopo l'altro a partire da 1 e calcolane il quadrato
- Fermati appena tale quadrato supera ***n***
- Il risultato corrisponde al valore dell'ultimo numero tale per cui vale la relazione: **x*x ≤ n**

Esercizio 2 (*Programma*)

```
int radice_intera(int n)
  {
    int i, radice;
    for (i=1; i <= n;  i++)
          if (i*i>n) radice=i-1;
    return (radice);
  }
```

Funziona?
Si può fare di meglio?

Esercizio 2 (*Programma*)

```
int radice_intera(int n)
  {
    int i, radice=1;
    for (i=1; i*i <= n;  i++)
          { radice=i-1;
            break;
          }
    return (radice);
  }
```

Funziona?

Esercizio 2 (*Programma –versione OK*)

```
int radice_intera(int n)
  {
    int i, radice=1;
    for (i=1; i*i <= n;  i++)
          ;          // istruzione vuota
    radice=i-1;
    return (radice);
  }
```

Nota sul passaggio dei parametri in C/C++

Lista dei parametri

- **Parametri attuali**: argomenti inseriti al momento della chiamata di funzione.

  Possono essere **espressioni** (*costanti*, *variabili*, *espressioni aritmetiche*, …)

- **Parametri formali**: argomenti definiti/dichiarati nella definizione/dichiarazione di funzione.

  Devono essere **variabili**.

Lista dei parametri

- Non appena l'ambiente della funzione chiamata viene attivato, i parametri formali vengono **definiti** (come **variabili locali** all'ambiente della funzione) ed **inizializzati** al valore del corrispondente parametro attuale

- La corrispondenza tra parametri formali e attuali è sia posizionale sia di tipo. Ovvero si presume che la lista dei parametri formali e la lista dei parametri attuali abbia lo stesso numero e tipo di elementi

- I nomi dei parametri attuali e formali non hanno importanza. Possono essere gli stessi o diversi. L'importante è la posizione ed il valore che assume un parametro attuale al momento della chiamata

Passaggio dei parametri

- Per “passaggio dei parametri” si intende l’associazione fra parametri attuali e parametri formali che avviene al momento della chiamata di una funzione

- ***L'unico meccanismo adottato in C, si chiama PASSAGGIO PER VALORE***

- Come vedremo, in C++ disponiamo anche del passaggio per **riferimento**

Passaggio dei parametri per valore

Le locazioni di memoria corrispondenti ai parametri formali:

- Sono allocate al momento della chiamata della funzione
- Sono inizializzate con i valori dei corrispondenti parametri attuali trasmessi dalla funzione chiamante
- Vivono per tutto il tempo in cui la funzione è in esecuzione
- Sono deallocate quando la funzione termina

QUINDI

- La funzione chiamata riceve copia dei valori dei parametri attuali passati dalla funzione chiamante
- Tali copie sono *sue copie private che servono solo per inizializzare i parametri formali*
- Ogni modifica ai parametri formali è *strettamente locale* alla funzione
- **I parametri attuali della funzione chiamante non saranno mai modificati!**

Esempio

- **Prototipo di funzione**

```
double CalcDistance (int, int, int, int);
```

- **Definizione di funzione**

```
double CalcDistance (int px1, int py1, int px2, int py2)
{
    px1 = pow (px1 - px2, 2);
    py2 = pow (py1 - py2, 2);
    return sqrt(px1 + py2) ;
}
```

- **Chiamata di funzione**

```
int a= 9, b= 5, c=4, d=12; double dist;
cout<<a<<b<<c<<d<<endl;
dist = CalcDistance (a, b, c, d);
cout<<a<<b<<c<<d<<dist<<endl;
```

Il collegamento tra parametri formali e parametri attuali si ha solo al momento della chiamata. Sebbene *px1 e py2* vengano modificati all'interno della funzione, i valori dei corrispondenti parametri attuali (*a*, *d*) rimangono inalterati. Quindi gli stessi valori di *a* e *d* sono stampati prima e dopo

Esempio 2

```
int  fattoriale   (int n)
  {
    int i, fatt=1;                    // con variabile ausiliaria i
    for (i=1; i<=n; i++)
          fatt = fatt*i;
    return fatt;
  }

main()  {
  int risultato, n = 4 ;
  risultato = fattoriale(n);
  cout<<"fattoriale("<<n<<") = "<<risultato<<endl ;
  }
```

Stampa

```
fattoriale(4) = 24
```

Esercizio

- Provare a realizzare il calcolo del fattoriale mediante una funzione senza utilizzare la precedente variabile ausiliaria

Esempio 2bis

```
int  fattoriale   (int n)
 {
   if (n == 0) return 1;
   int fatt = n;                    // senza variabile ausiliaria i
   for (n--; n > 0; n--)
         fatt = fatt*n;
   return(fatt);
 }

main()  {
  int risultato, n = 4 ;
  risultato = fattoriale(n);
  cout<<"fattoriale("<<n<<") = "<<risultato<<endl ;
 }
```

Anche se il parametro formale **n** viene modificato, la variabile **n** definita nel main *non viene alterata!* E' il suo valore (4) che viene passato alla funzione.

Stampa

```
fattoriale(4) = 24
```

Commenti al passaggio per valore

- E' sicuro: le variabili del chiamante e del chiamato sono *completamente disaccoppiate*
- Consente di ragionare per componenti e servizi: *la struttura interna dei singoli componenti è* ***irrilevante***

  (la funzione può anche modificare i parametri ricevuti senza che ciò abbia impatto sul chiamante)

- **LIMITI**
  - ***impedisce a priori*** **di scrivere funzioni che abbiano come scopo quello di *modificare* i dati passati dall'ambiente chiamante;**
  - **come vedremo può essere costoso per dati di grosse dimensioni.**

Esercizio 3 (*Specifica*)

- **Scrivere una funzione che scambi due valori A e B (di tipo int)**

Esercizio 3 (*Idea e Algoritmo*)

- Effettuare una “triangolazione” fra A, B e una variabile ausiliaria T

- Salvare in T il valore di A, poi assegnare ad A il valore di B, quindi assegnare a B il valore di T (cioè il vecchio valore di A)

Esercizio 3 (*Programma*)

```
void scambia(int A, int B)
  { int t;
    t = A;   A = B;   B = t;
  }

main()
{
  int  x = 12, y = 27;
  cout<<"x="<<x<<" y="<<y<<endl ;
  scambia(x, y);
  cout<<"x="<<x<<" y="<<y<<endl ;
}
```

Stampa voluta	***Stampa vera***
`x=12  y=27`	`x=12  y=27`
`x=27  y=12`	`x=12  y=27`

MOTIVO: Semantica del **passaggio di parametri per valore.**

La funzione **scambia** ha scambiato A e B **al suo interno, ma questa modifica non si è propagata** ai parametri attuali

Passaggio dei parametri per riferimento

Per superare i limiti della semantica per copia, occorre consentire alla funzione di far riferimento alle variabili effettive del chiamante

IL CONCETTO DI RIFERIMENTO

- Una funzione deve poter dichiarare, nella sua intestazione, che un parametro costituisce un riferimento. In tal caso:
  - il parametro formale non è più una variabile locale inizializzata al valore del parametro attuale, ma è un riferimento alla variabile originale (*parametro attuale*) nell'ambiente della funzione chiamante
  - quindi, ogni modifica fatta al parametro formale in realtà viene effettuata sul parametro attuale della funzione chiamante (le modifiche si propagano)

- **Il passaggio per riferimento è disponibile in molti linguaggi (Pascal, C++), ma NON è disponibile in C, nel quale deve essere simulato tramite puntatori**

- La versione corretta del programma C/C++ che realizza la funzione **scambia()** si vedrà successivamente, dopo aver studiato i riferimenti