

# Funzioni

Parte I  
Definizione,  
Dichiarazione o Prototipo,  
Chiamata

# Blocco

- Sequenza di definizioni e istruzioni racchiuse tra parentesi graffe

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

- In C, parte dichiarativa ed esecutiva devono essere separate:

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

# Blocco ed istruzione composta

- Per semplicità in precedenza abbiamo definito l'istruzione composta come un caso particolare di blocco in cui c'erano solo istruzioni
  - In verità un'istruzione composta si esprime esattamente mediante un blocco
  - Quindi si possono inserire dichiarazioni anche in un'istruzione composta
- Inoltre con i blocchi non si esprimono solo le istruzioni composte, ma anche ad esempio i corpi delle funzioni, come vedremo a breve ...

# Utilità delle funzioni

- Per evidenziare l'utilità delle funzioni, scriviamo
  - 1) un programma che legga in ingresso un numero intero non negativo e dica all'utente se il numero è primo (o altrimenti non stampi nulla)
  - 2) un programma che legga in ingresso due numeri interi non negativi e, se e solo se sono entrambi primi, dica all'utente se si tratta di due numeri *primi gemelli* (ossia che differiscono esattamente di due unità l'uno dall'altro)  
Es.: 5 e 7 sono primi gemelli
- Cominciamo dal primo programma

# Programma 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  $3 \leq i \leq \sqrt{N}$ : se nessun numero  $i$  risulta essere un divisore di  $N$ , allora  $N$  è primo

# Programma 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  $static\_cast<int>(sqrt(N))$ 
  - Ove  $sqrt(N)$  è una funzione che ritorna  $\sqrt{N}$ , mentre l'espressione  $static\_cast<int>(sqrt(N))$  è uguale al più grande numero intero non maggiore di  $\sqrt{N}$
  - Nota: ci sta bene utilizzare la parte intera di  $sqrt(N)$  perché il potenziale divisore sarà senz'altro un numero intero

# Programma 1 (*Rappresentazione informazioni*)

- Variabile per contenere il numero: **int n**
- Può tornare poi utile 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
- Per utilizzare la funzione **sqrt()** occorre:
  - includere anche **<cmath>** (**<math.h>** in C)  
Es.: 

```
#include <iostream>
#include <cmath>
```
  - aggiungere l'opzione **-lm** nell'invocazione del **g++**

Es.: `g++ -lm nomefile.cc`

# Programma 1

```
main()
{
    int max_div, n ; cin>>n ;

    if (n>=1 && n<=3) { cout<<"primo"<<endl ; return ; }

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

    max_div = static_cast<int>(sqrt(n));
    for(int i=3; i<=max_div; i=i+2)
        if (n%i==0) return ;    /* no, perché è stato trovato
                                   un divisore */
    cout<<"primo"<<endl ;
}
```



# Secondo programma

- Veniamo ora al secondo programma, che ricordiamo era:
- Un programma che legge in ingresso due numeri interi non negativi e, se e solo se sono entrambi primi, comunica all'utente se si tratta di due numeri primi gemelli (ossia che differiscono esattamente di due unità l'uno dall'altro)
- Riutilizzando le righe di codice già scritte, potremmo scrivere ad esempio:

# Programma 2 1/2

```
main()
{
    int n1, n2 ; cin>>n1>>n2 ;
    int max_div; bool n1_is_prime = false,
        n2_is_prime = false ;

    if (n1>=1 && n1<=3) n1_is_prime = true ;

    else if (n1%2 != 0) {
        int i; max_div = static_cast<int>(sqrt(n1));
        for(i=3; i<=max_div; i=i+2)
            if (n1%i==0) break ;
        if (i > max_div)
            n1_is_prime=true ;
    }
}
```

# Programma 2 2/2

```
if (n2>=1 && n2<=3) n2_is_prime = true ;  
else if (n2%2 != 0) {  
    int i ; max_div = static_cast<int>(sqrt(n2));  
    for(i=3; i<=max_div; i=i+2)  
        if (n2%i==0) break ;  
    if (i > max_div)  
        n2_is_prime=true ;  
}  
if (n1_is_prime && n2_is_prime)  
    if (n1 == n2 - 2 || n2 == n1 - 2)  
        cout<<"n1 ed n2 sono due primi gemelli"<<endl ;  
}
```

# Considerazioni

- Quanto è leggibile il programma?
  - Non molto
- Come mai?
- Fondamentalmente perché c'è codice abbastanza lungo ripetuto due volte
- Quale sarebbe stato il livello di leggibilità se avessimo avuto a disposizione una funzione *is\_prime()* a cui passavamo come argomento un numero e ci diceva se era primo?

# Programma 2 Altra versione

```
main()
{
    int n1, n2 ; cin>>n1>>n2 ;
    if (is_prime(n1) && is_prime(n2))
        if (n1 == n2 - 2 || n2 == n1 - 2)
            cout<<"n1 ed n2 sono due primi gemelli"<<endl ;
}
```

- Molto più breve, leggibile e facile da capire
- Non contiene codice duplicato
- Per arrivare a questo risultato dobbiamo imparare a definire ed utilizzare le funzioni ...

# 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 rispecchia l'astrazione matematica di funzione:

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

- **molti ingressi possibili** (corrispondenti ai valori su cui operare)
- **una sola uscita** (corrispondente al risultato o valore di ritorno)
- Per calcolare il valore di ritorno della funzione dovranno essere eseguite una serie di istruzioni
  - In C/C++ tali istruzioni sono specificate mediante un blocco, a cui ci si riferisce tipicamente come il **corpo della funzione**

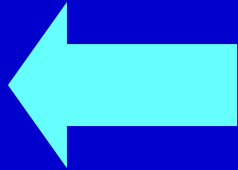
# Procedura

- Altri linguaggi (**ma non il C/C++!**) introducono anche l'astrazione di **procedura**, che cattura la sola astrazione dell'esecuzione di un insieme di azioni, senza ritornare risultati
- Comunque, come vedremo è **IMMEDIATO DA EMULARE IN C/C++** mediante funzioni dal valore di ritorno **vuoto**



# 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*

# Esempi di intestazioni

**int**  
Tipo risultato

**fattoriale**  
Nome della funzione

**(int n)**  
Lista dei parametri formali

**int**  
Tipo risultato

**somma3**  
Nome della funzione

**(int x, int y, int 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 omissa).  
**LE PARENTESI SERVONO SEMPRE**

# Sintassi definizione di funzione

- Come già detto, 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)  
o in questo caso può anche essere omessa la lista stessa
  - una sequenza di definizioni di parametri, se la lista non è vuota

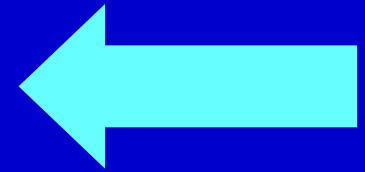
# Definizione di funzione (*cont.*)

**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
- Un parametro attuale è una espressione il cui risultato è il valore da assegnare al parametro formale che si trova in quella posizione
  - Ci torneremo sopra a breve



# Esempio

```
void fun(int a)
{
    cout<<a<<endl ;
}
```

Definizione



```
int main()
{
    fun(3);
}
```

Invocazione



# 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

```
void fun()
```

```
{
```

```
    cout<<"Saluti dalla funzione fun"<<endl ;
```

```
}
```

```
main()
```

```
{
```

```
    fun() ;
```

```
}
```

# Lista dei parametri formali e attuali

- **Lista parametri formali: argomenti dichiarati nella definizione di funzione**
  - Devono essere variabili o costanti con nome
- **Lista parametri attuali: argomenti inseriti al momento della chiamata di funzione**
  - Espressioni separate da virgole, ove ciascuna espressione può essere una:
    - *costante*
    - *variabile*
    - *espressione aritmetica o logica*
    - ...

# Dinamica parametri formali e attuali

- La corrispondenza tra parametri formali e attuali è posizionale, con in più il controllo di tipo. Si presume che la lista dei parametri formali e la lista dei parametri attuali abbiano lo stesso numero e tipo di elementi (l'uso della conversione di tipo si vedrà in seguito)
- La corrispondenza tra i nomi dei parametri attuali e formali non ha nessuna importanza. Gli eventuali nomi di variabili passate come parametri attuali possono essere gli stessi o diversi. Conta solo la posizione all'interno della chiamata
- Non appena parte l'esecuzione di una funzione, ciascuno dei parametri formali viene **definito (come variabile locale alla funzione)** ed **inizializzato** al valore del corrispondente parametro attuale

# Collocazione definizioni di funzione

Una funzione non può essere definita all'interno 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  $i$*



# 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 i;
    i++;
    cout<<i;
}
```

*i ha un valore iniziale casuale*



# Istruzione *return* 1/2

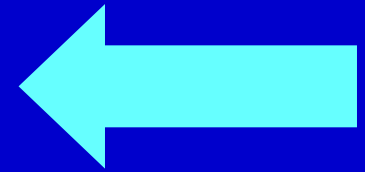
- Viene usata per far terminare l'esecuzione della funzione e far proseguire il programma dall'istruzione successiva a quella con cui la funzione è stata invocata, ossia per *restituire il controllo alla funzione chiamante*, e, se la funzione ha tipo di ritorno diverso da *void*, *restituire il valore calcolato dalla funzione*, ossia il risultato della funzione
- Sintassi nel caso di funzioni con tipo di ritorno diverso da *void*:  
**return (<espressione>);**  
oppure semplicemente  
**return <espressione>;**  
  
    **<espressione>** deve essere del **tipo di ritorno** specificato nell'intestazione della funzione
- Sintassi nel caso di funzioni con tipo di ritorno *void*:  
**return ;**

# Istruzione *return* 2/2

- Eventuali istruzioni della funzione successive all'esecuzione del *return* non saranno eseguite!
- Nel caso della funzione *main* l'esecuzione dell'istruzione *return* fa uscire dall'intero programma
- Una funzione con tipo di ritorno *void* può terminare o quando viene eseguita l'istruzione *return* o quando l'esecuzione giunge in fondo alla funzione
- Al contrario, una funzione con tipo di ritorno diverso da *void* deve sempre terminare con una istruzione *return*, perché bisogna specificare il valore di ritorno

# 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 l'invocazione di un insieme di istruzioni (procedura)
  - **Funzioni che ritornano un valore**: la chiamata di funzione va considerata come un'espressione il cui valore di ritorno (calcolato eseguendo le istruzioni della funzione) è di tipo uguale al tipo di ritorno della funzione, e può essere utilizzato come fattore all'interno di espressioni composte

Esempio:

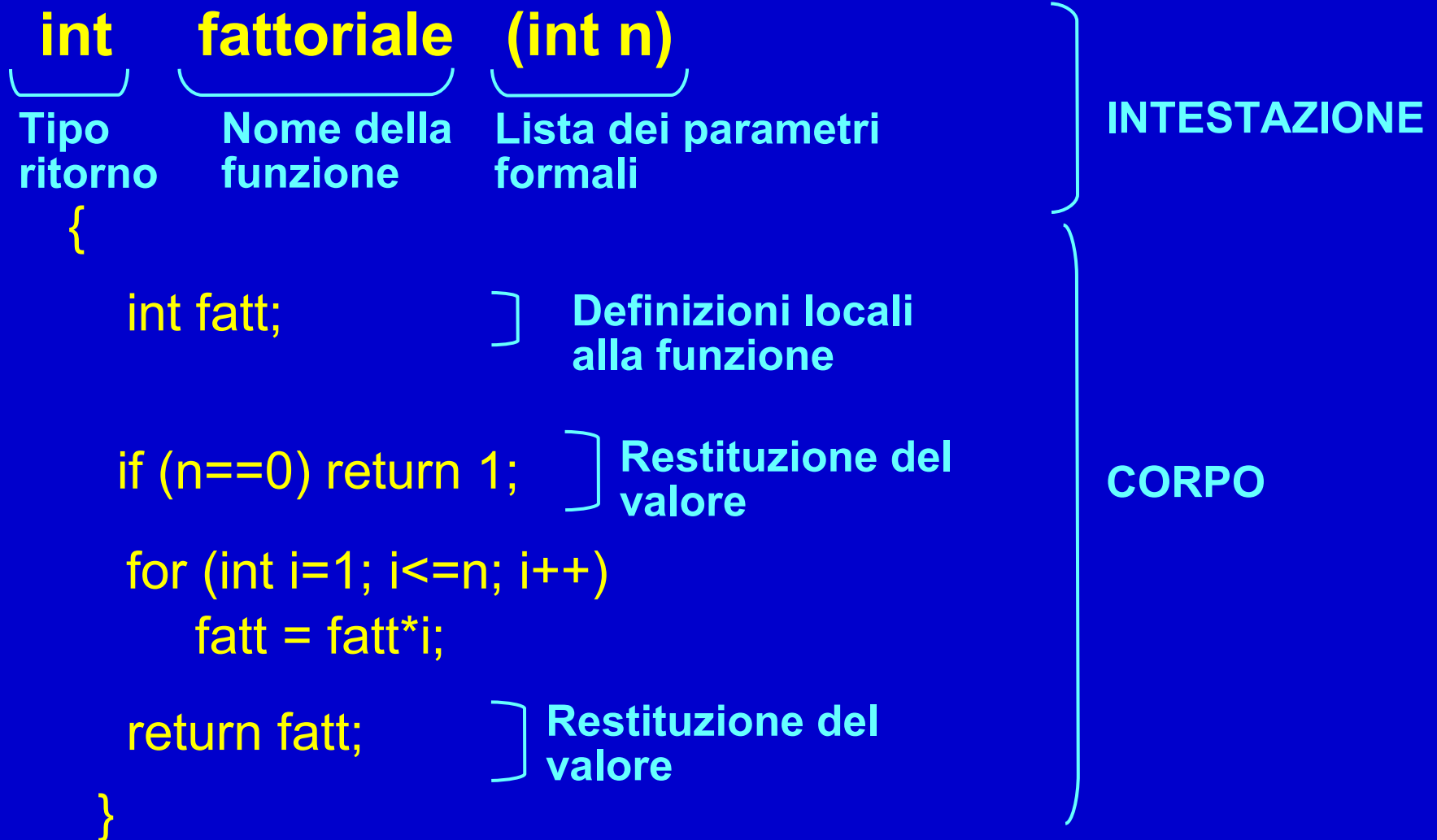
Supponendo che *fun()* ritorni un valore di tipo *int*:

```
int b = fun(a) + 1 ;
```

# Esercizi

- *funz\_max.cc*
- *funz\_fattoriale.cc*

# Anatomia funzione fattoriale



# 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 oggetti precedentemente dichiarati
- Questo vale anche per le funzioni, che quindi possono essere invocate solo dopo essere state precedentemente dichiarate
  - Finora abbiamo visto solo definizioni di funzione
  - Sono in effetti un caso di dichiarazione: quello in cui si definisce anche il corpo della funzione stessa
    - o 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();
```

```
}
```

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

```
void fun2()
```

```
{
```

```
    cout<<"fun2"<<endl ;
```

```
    fun1();
```

```
}
```

# Dichiarazione della funzione

- Una dichiarazione (senza definizione) o prototipo è costituita dalla sola intestazione di una funzione seguita da ;  
**<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();
```

```
}
```

# Altri esempi di prototipi

```
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);
```

```
}
```

---

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

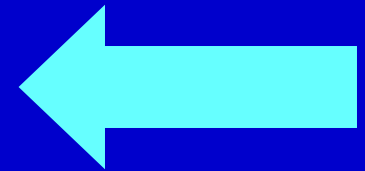
# Prototipi e definizioni

- 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)**



# 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
- E' proprio la funzione che ci serve per completare il programma che abbiamo usato per introdurre l'utilità delle funzioni all'inizio di questa presentazione

# Esercizio 1 (*Programma*)

```
bool isPrime(int n)
{
    int max_div ;

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

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

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

    return true;
}
```

# Istruzione vuota

- E' un semplice ;
- Non fa nulla
- Sintatticamente è trattata come una qualsiasi altra istruzione
- Esempio:

```
// ciclo che si ferma quando num è primo
for(int num = 10001 ; ! isPrime(num) ; num += 2)
    ; // non fa nulla
```

## 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 radice;
    for (int i=1; i <= n; i++)
        if (i*i>n) radice=i-1;
    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++

# 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, è il **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 fatt=1;          // con variabile ausiliaria i
    for (int 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;  
  for (n--; n > 0; n--)  
    fatt = fatt*n;  
  return(fatt);  
}
```

// senza variabile ausiliaria i

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.

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

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 proprio quello di *modificare* i dati passati dall'ambiente chiamante
  - come vedremo può essere costoso per dati di grosse dimensioni



# Domanda

- Se un parametro formale è dichiarato di tipo *const*, lo si può poi modificare all'interno della funzione?

- Esempio:

```
int fun(const int j)
{
    j++;
}
```

# Risposta

- Ovviamente no
- Il parametro è inizializzato all'atto della chiamata della funzione, e da quel momento non potrà più essere modificato
- Quindi:

```
int fun(const int j)
{
    j++; // ERRATO !! NON VERRA' COMPILATO
}
```

# Nota 1/2

- Abbiamo visto la modifica di un parametro formale variabile all'intero di una funzione solo per capire che la cosa si può fare, e che tale parametro è perfettamente equivalente ad una variabile locale
- Tuttavia, è fondamentale avere presente che
  - In generale è una **cattiva abitudine** modificare i parametri formali per utilizzarli come variabili ausiliarie
    - Poca leggibilità
    - Rischioso nel caso di parametri passati per riferimento (che vedremo nelle prossime lezioni)

# Nota 2/2

- L'unico caso in cui è necessario ed appropriato modificare i parametri formali è quando tali parametri sono intesi come **parametri di uscita**, ossia parametri in cui devono essere memorizzati valori che saranno poi utilizzati da chi ha invocato la funzione
- Questo non può però accadere nel caso di passaggio per valore, perché i parametri formali sono oggetti locali alla funzione, e saranno quindi eliminati alla terminazione della funzione stessa
- Vedremo invece più avanti come implementare i parametri di uscita mediante il passaggio per riferimento

# Nota conclusiva sui 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
- Supporto allo sviluppo top-down del software
  - Si può progettare prima quello che c'è da fare in generale, e poi si può realizzare ogni singola parte