

# Lezione 5

---

Funzioni:  
Definizione,  
Prototipo,  
Chiamata

# Istruzione composta 1/2

---

- Come abbiamo visto, una istruzione composta è una sequenza di istruzioni racchiuse tra parentesi graffe

```
{  
    <istruzione>  
    <istruzione>  
    ...  
}
```

- Quindi in una istruzione composta vi possono essere anche istruzioni definizione

```
{  
    int a, c ;  
    cin>>a ;  
    ...  
}
```

# Istruzione composta 2/2

---

- Completiamo la discussione delle istruzioni composte aggiungendo che, in C, parte dichiarativa ed esecutiva devono essere separate:

```
{  
    <dichiarazione>  
    <dichiarazione>  
    ...  
    <istruzione diversa da dichiarazione>  
    <istruzione diversa da dichiarazione>  
    ...  
}
```

- Una istruzione composta viene anche chiamata **blocco**

# Introduzione

---

- Scriviamo 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)

# Analisi ed idee 1/3

---

- Un numero è primo se è divisibile solo per 1 e per se stesso
- Quindi, per scoprire se un numero  $N$  è primo, occorre provare a dividere  $N$  per tutti i numeri  $2 \leq i \leq N-1$ .
  - Se nessuno di questi numeri  $i$  risulta essere un divisore di  $N$ , allora  $N$  è primo.
- Funziona?
  - Sì, ma si può fare meglio ...

# Analisi ed idee 2/3

---

- IDEA 1: Poiché i numeri pari non sono primi, possiamo controllare subito se  $N$  è pari. Poi non sarà più necessario provare a dividere  $N$  per un numero pari
- IDEA 2: Non c'è bisogno di provare tutti i numeri dispari fino ad  $N-1$ , ma è sufficiente provare a dividere  $N$  per tutti i numeri dispari  $3 \leq i \leq N/2$ : se nessuno di questi numeri  $i$  risulta essere un divisore di  $N$ , allora  $N$  è primo
- IDEA 3: Ci si può fermare anche prima!  
Per capirlo consideriamo le seguenti domande
  - Quanto fa  $(\sqrt{N})^2$  ?
  - Se  $i > \sqrt{N}$ , allora  $i^2 > N$  ?

# Analisi ed idee 3/3

---

- Ma il fatto che  $i^2 > N$  quando  $i > \sqrt{N}$ , si può riscrivere come  $i * i > N$
- Ora, se tale numero  $i > \sqrt{N}$  fosse un divisore di  $N$ , significherebbe che esiste un numero  $j$  tale che  $i * j = N$
- Però, siccome  $i * i > N$ , allora, affinché  $i * j = N$ , si deve avere  $j < i$ 
  - Ma questo vuol dire che  $j$  lo avremmo già provato come potenziale divisore nel nostro algoritmo prima di arrivare ad  $i$ !
- In definitiva è sufficiente provare a dividere  $N$  per tutti i numeri dispari  $3 \leq i \leq \sqrt{N}$ : se nessuno di tali numeri risulta essere un divisore di  $N$ , allora  $N$  è primo

# Possibile 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  $N > 3$ ), occorre tentare tutti i possibili divisori dispari da 3 in avanti, fino a  $\sqrt{N}$ 
  - Ma  $\sqrt{N}$  può non essere un numero intero, mentre invece per ora noi sappiamo lavorare solo con i numeri interi
  - Per fortuna ci sta bene utilizzare la parte intera di  $\sqrt{N}$  perché il potenziale divisore deve essere necessariamente un numero intero!
- La parte intera di  $\sqrt{N}$  si può ottenere valutando l'espressione `static_cast<int>(sqrt(N))`

- Per utilizzare la funzione `sqrt()` occorre:
  - includere anche `<cmath>` (`<math.h>` in C)  
Esempio: 

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

```
g++ -lm -o nome nomefile.cc
```

# Struttura dati

---

- 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
- Servirebbe poi una variabile ausiliaria  
`int i`  
come indice per andare da 3 a `max_div`

- Utilizzando il programma dire quali dei seguenti numeri sono primi
  - 161531
  - 419283
  - 971479

# Programma numero primo

---

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

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

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

    int 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 ;
}
```

# Risposte

---

- 161531 Primo
- 419283 Non primo
- 971479 Primo

# Primi gemelli

---

- Due numeri **primi** si definiscono **gemelli** se differiscono per esattamente due unità
  - Esempi: 5 e 7, 11 e 13
- Scriviamo un programma che
  - legga in ingresso due numeri interi non negativi e, se e solo se sono entrambi primi, comunichi all'utente se si tratta di due numeri primi gemelli

# Riutilizzo del codice

---

- Scrivere codice costa fatica
- Come vedremo meglio in seguito, man mano che un programma diventa più complicato la componente di lavoro dovuta al collaudo ed alla correzione degli errori aumenta
- Quindi, con spirito critico, **riutilizziamo sempre il codice già disponibile**
- Esempio: non ci siamo riscritti in ogni nostro programma il codice di invio di caratteri su *stdout*, ma abbiamo usato l'oggetto `cout` e l'operatore `<<`

# Applicazione al programma

---

- Mettiamo in pratica questo approccio anche nel programma che stiamo per scrivere
  - Cerchiamo quindi di riutilizzare il codice già scritto per verificare se un numero è primo
- Con le conoscenze attuali possiamo riutilizzare tale codice senza doverlo riscrivere (o incollare) nel nuovo programma?

# Primo problema

---

- Purtroppo no
- Ci manca prima di tutto un meccanismo per dare un nome ad un pezzo di codice e
  - richiamarlo in qualsiasi punto di un programma,
  - senza doverlo riscrivere in quel punto
- Cerchiamo comunque di fare del nostro meglio con le nostre conoscenze e scriviamo il programma

- Quali delle seguenti coppie di numeri è costituita da primi gemelli?
  - 11057 e 11059
  - 11059 e 11061

# Programma 1/2

---

```
main()
{
    int n1, n2 ; cin>>n1>>n2 ;
    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 ;
    }
    // continua nella prossima slide ...
}
```

# Programma 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 ;
}
```

- La prima coppia

# Leggibilità e manutenibilità

---

- Quanto è leggibile il programma?
  - Non molto
- Come mai?
  - Fondamentalmente perché c'è codice molto simile ed abbastanza lungo ripetuto due volte
- Inoltre il codice replicato rende più difficile anche la manutenzione del programma per i motivi precedentemente discussi
- Riusciamo a trovare un modo, con le nostre conoscenze, per eliminare la replicazione?

# Miglioramento leggibilità

---

- Purtroppo no!
- Proviamo almeno a rendere più leggibile il programma cercando di spiegare l'obiettivo di ciascuna parte del programma
  - Come possiamo fare?

- Aggiungendo dei commenti
- Cosa si scrive nei commenti ad un pezzo di codice?
  - L'obiettivo/significato di un pezzo di codice
  - Un riepilogo di cosa fa un pezzo di codice complesso
- Cosa non si scrive nei commenti?
  - Non si ripete quello che un pezzo di codice fa, perché è già scritto **nel** codice

# Programma commentato 1/2

---

```
main()
{
    int n1, n2 ; cin>>n1>>n2 ;

    // ciascuna delle seguenti due variabili ha valore true se e solo
    // se il corrispondente valore intero (n1 o n2) è primo;
    // le inizializziamo a false e lasciamo ai seguenti due pezzi di
    // codice il compito di assegnare a ciascuna di loro il valore true
    // quando il corrispondente valore intero è primo
    bool n1_is_prime = false, n2_is_prime = false ;

    // determino se n1 è primo e, nel caso, setto n1_is_prime a true
    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 ;
    }
    // continua nella prossima slide ...
}
```

# Programma commentato 2/2

```
// determino se n2 è primo e, nel caso, setto n1_is_prime a
// true
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 ;
}
```

# Riepilogo

---

- Utilizzando i commenti siamo riusciti ad ottenere un po' più di leggibilità
  - Ma l'ideale sarebbe stato poter dare un significato a quel pezzo di codice NEL LINGUAGGIO DI PROGRAMMAZIONE
  - Ossia dargli un nome **significativo** ed utilizzarlo semplicemente chiamandolo per nome
  - Supponiamo di esserci riusciti in qualche modo, e di averlo trasformato in una funzione `is_prime()` a cui si passa come argomento un numero e ci dice se è primo

# Nuova versione programma

---

```
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 ;
}
```

- Il nome della funzione (se scritto bene) ci fa subito capire a cosa serve la sua invocazione
- Non dobbiamo più stare attenti alle righe di codice da ripetere ovunque vogliamo che venga svolta la funzione `is_prime()`
- Dobbiamo solo scrivere il codice della funzione da qualche parte una volta per tutte

# Replicazione del codice

---

- Ed il problema della replicazione del codice?
  - **Risolto anche quello dalla nostra funzione!**
- A questo punto abbiamo dei validi motivi per iniziare lo studio delle funzioni

---

# Funzioni

# Concetto di funzione

---

- L'astrazione di funzione è presente in tutti i linguaggi di programmazione di alto livello
- Una funzione è un costrutto che rispecchia l'astrazione matematica di funzione:  
$$f : \mathbf{A} \times \mathbf{B} \times \dots \times \mathbf{Q} \rightarrow \mathbf{S}$$
  - **multi 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 (istruzione composta)**, a cui ci si riferisce tipicamente come il **corpo della funzione**

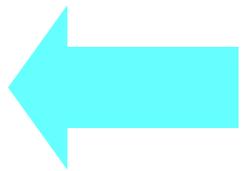
# Procedura

---

- Altri linguaggi (ma non il C/C++!) introducono separatamente anche l'astrazione di **procedura**
  - Esecuzione di un insieme di azioni, senza ritornare risultati
- Comunque, come vedremo una procedura è IMMEDIATA DA EMULARE IN C/C++ mediante una funzione dal valore di ritorno **vuoto**

# Elementi fondamentali

---

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

# Definizione

---

- Una definizione di funzione è costituita da una **intestazione** e da un **corpo**, definito mediante un blocco (istruzione composta)
- Partiamo da alcuni esempi per dare un'idea intuitiva della definizione di una funzione e delle modalità di esecuzione
- Vedremo poi tutti i dettagli formali

# Esempi di intestazioni 1/4

---

`int`

Tipo del  
risultato

`fattoriale`

Nome della  
funzione

`(int n)`

Lista dei parametri  
formali

- Intestazione di una funzione di nome `fattoriale`, che prende in ingresso un valore di tipo `int` e ritorna **un valore** di tipo `int`.
- All'inizio dell'esecuzione della funzione il valore preso in ingresso sarà memorizzato nel **parametro formale** `n`
  - Tale parametro può essere poi utilizzato all'interno del corpo della funzione come una normale variabile
    - Quindi tramite tale parametro il codice della funzione può leggere il valore passato alla funzione stessa

# Esempi di intestazioni 2/4

---

`int`

Tipo del  
risultato

`somma`

Nome della  
funzione

`(int x, int y)`

Lista dei parametri  
formali

- Intestazione di una funzione di nome `somma`, che prende in ingresso **due** valori di tipo `int` e ritorna **un** valore di tipo `int`.
- All'inizio dell'esecuzione della funzione i due valori presi in ingresso saranno memorizzati nei **due parametri formali** `x` ed `y`, e tramite tali parametri potranno essere utilizzati all'interno della funzione stessa

# Esempi di intestazioni 3/4

---

`void`

Tipo del  
risultato

`stampa_n_volte`

Nome della  
funzione

`(int n)`

Lista dei parametri  
formali

- Intestazione di una funzione di nome `stampa_n_volte`, che prende in ingresso un valore di tipo `int` e **non ritorna nulla** (tipo di ritorno vuoto)
- All'inizio dell'esecuzione della funzione il valore preso in ingresso sarà memorizzato nel parametri formale `n`, e tramite tale parametro potrà essere utilizzato all'interno della funzione stessa

# Esempi di intestazioni 4/4

---

`void`

Tipo del risultato

`stampa_2_volte`

Nome della funzione

`(void)`

Lista dei parametri formali

- Intestazione di una funzione di nome `stampa_2_volte`, che **non prende in ingresso nulla** (tipo di ingresso vuoto) e **non ritorna nulla** (tipo di ritorno vuoto)
- L'intestazione si poteva equivalentemente scrivere così:

```
void stampa_2_volte ()
```

# Sintassi definizione funzione

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

$\langle \text{definizione-funzione} \rangle ::=$   
 $\langle \text{intestazione-funzione} \rangle \langle \text{blocco} \rangle$

$\langle \text{intestazione-funzione} \rangle ::=$   
 $\langle \text{nomeTipo} \rangle \langle \text{nomeFunzione} \rangle ( \langle \text{lista-parametri} \rangle )$

$\langle \text{lista-parametri} \rangle ::=$   
 $\text{void} \mid \langle \text{def-parametro} \rangle \{ \text{ , } \langle \text{def-parametro} \rangle \}$



O l'uno o l'altro



$\langle \text{def-parametro} \rangle ::= [ \text{const} ] \langle \text{nomeTipo} \rangle \langle \text{identificatore} \rangle$

# Intestazione

---

- 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 (ossia non ci sono parametri)
      - può anche essere semplicemente omessa la lista senza scrivere `void`
    - una sequenza di definizioni di parametri, se la lista non è vuota

# Elementi fondamentali

---

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

# Sintassi chiamata funzione

---

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

*<chiam-funzione> ::=*

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

- Un **parametro attuale** è una **espressione**, il cui risultato è il valore da assegnare, quando inizia l'esecuzione della funzione, al parametro formale che si trova nella stessa posizione del parametro attuale
  - Ci torneremo sopra a breve

# Uso chiamata funzione

---

- L'istruzione più semplice che contenga una chiamata di funzione, è la seguente:

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

- Si tratta quindi della chiamata di funzione, seguita dal ;
- L'effetto di tale istruzione è quello di far partire l'esecuzione della funzione
  - Una volta terminata la funzione, l'esecuzione del programma riprende dall'istruzione successiva a quella in cui la funzione è stata invocata
- In generale però una chiamata di una funzione è una **espressione**
  - e si può quindi inserire a sua volta in una espressione

# Esempio

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

*Definizione*

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

*Invocazione*

# Definizione e chiamata

---

- Una funzione può essere invocata solo da un punto del programma successivo, nel testo del programma stesso, alla definizione della funzione
  - In verità, come vedremo fra qualche slide, basta che sia successivo ad un punto in cui la funzione è stata dichiarata
- Esempio di **programma scorretto**:

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

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

# Posizione definizioni

---

- Una funzione **non può essere definita all'interno di un'altra funzione**

```
main ()  
{  
    void fun ()  
    {  
        ...  
    }  
  
    fun () ;  
}
```

# 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() ;  
}
```

# Parametri formali ed attuali

---

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

- La seguente funzione:

```
int fun(int a, int b)
{
    ...
}
```

- Può essere invocata, ad esempio, in tutti i modi mostrati nel seguente pezzo di programma:

```
main()
{
    int d = 3, k = 5 ;
    fun(k, d) ;
    fun(2, k) ;
    fun(k - 5, 2 * d + 7) ;
}
```

# Associazione parametri

---

- 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 oggetto locale alla funzione) ed **inizializzato** al valore del corrispondente parametro attuale

# Oggetti locali 1/2

---

- Definiamo come **locale ad una funzione** un oggetto che si può utilizzare solo all'interno della funzione
- Un parametro formale è un oggetto locale di una funzione
  - Come si è visto può essere variabile oppure una costante
    - Nel caso sia variabile, il suo valore può essere modificato all'interno della funzione
    - Nel caso sia costante, il suo valore, inizializzato all'atto della chiamata della funzione, non può più essere cambiato.

Esempio di intestazione di funzione con parametro formale costante:

```
int fun(const int a)
```

# Oggetti locali 2/2

---

- Anche le variabili e le costanti con nome definite **all'interno del corpo** di una funzione sono locali alla funzione
- Se non inizializzate, le variabili locali hanno valori casuali

# Confronto 1/2

*Definizioni (quasi) equivalenti di una variabile locale  $i$*

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

*Istruzione legale*

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

# Confronto 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 deve restituire un valore di ritorno

# Uso del valore di ritorno

---

- Come si è detto, la chiamata di una funzione è una espressione
- Due casi:
  - 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` si può scrivere:  

```
int b = fun(a) + 1 ;
```
  - Funzioni `void`: la chiamata di funzione va considerata solamente come l'invocazione di un insieme di istruzioni (procedura) che non ritornano alcun valore

# Esercizi e consigli

---

- Svolgere *funz\_max.cc* e *funz\_fattoriale.cc* della sesta esercitazione
- Già da qualche esercizio non vi sto più dando suggerimenti su ogni fase di sviluppo (analisi del problema, idee, algoritmo, scrittura programma)
  - Dovete però sempre seguire lo schema corretto se volete fare un buon lavoro
  - Partire direttamente dalla scrittura di codice confuso porta quasi sempre ad un **cattivo risultato** ed uno **scarso miglioramento delle proprie capacità**

# Ripasso anatomia funzione

int                      fattoriale                      (int n)                      ] **INTESTAZIONE**  
Tipo                      Nome della                      Lista dei parametri  
ritorno                      funzione                      formali

```
{  
  int fatt;                      ] Definizione di oggetto  
  locale alla funzione  
  
  if (n==0) return 1;                      ] Restituzione  
  del valore  
  
  for (int i=1; i<=n; i++)  
    fatt = fatt*i;  
  
  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
  - Altrimenti la lista dei parametri è vuota (`void`)
- La funzione deve restituire un dato dalla funzione chiamante?
  - Se sì, identificare il tipo di dato e definirlo come tipo di ritorno della funzione
  - Se no, il tipo di ritorno della funzione è `void`

# Chiamate incrociate

```
void fun1 ()  
{  
    ...  
    fun2 ();  
    ...  
}
```

```
void fun2 ()  
{  
    ...  
    fun1 ();  
    ...  
}
```

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

# Dichiarazione 1/2

---

- Come abbiamo già detto a suo tempo, una definizione è un caso particolare di dichiarazione
- In particolare:
  - Una definizione di variabile o costante con nome è una dichiarazione che causa l'allocazione di spazio in memoria quando viene incontrata
  - Una definizione di funzione è un caso particolare di dichiarazione in cui si definisce il corpo della funzione

# Dichiarazione 2/2

---

- In generale, una dichiarazione è una istruzione in cui si introduce un nuovo identificatore e se ne dichiara il tipo
- In C/C++ ogni identificatore si può utilizzare solo dopo essere stato dichiarato
- Quindi le definizioni sono delle dichiarazioni in cui non solo si introduce un nuovo identificatore ed il tipo associato, ma
  - nel caso delle variabili e costanti con nome si alloca anche memoria
  - nel caso delle funzioni si definisce anche il corpo
- Vediamo quindi la dichiarazione senza definizione di una funzione

# Dichiarazione funzione

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

*<dichiarazione-funzione> ::= <intestazione-funzione> ;*

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

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

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

**Opzionale !**

# Soluzione chiamate incrociate

---

```
void fun2 () ; // dichiarazione di fun2
```

```
void fun1 ()  
{  
    ...  
    fun2 () ;  
    ...  
}
```

```
void fun2 ()  
{  
    ...  
    fun1 () ;  
    ...  
}
```

# Altri esempi di prototipi

---

```
int fattoriale (int);
```

```
main()
```

```
{
```

```
    ...
```

```
}
```

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

```
{
```

```
    int fatt=1;
```

```
    for (int 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 causa la produzione di alcun byte di codice**
  - può essere ripetuto più volte nel programma (basta che non ci siano due dichiarazioni in contraddizione)
  - può comparire anche dentro un'altra funzione
- La definizione, invece:
  - contiene il codice della funzione
  - **non può essere duplicata!!** (altrimenti ci sarebbero due codici per la stessa funzione)
  - non può essere inserita in un'altra 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**

# Esempio di programma errato

---

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

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

# Versione corretta 1

---

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

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

# Versione corretta 2

```
int massimo(int, int) ;
```

```
main()
```

```
{
```

```
    int a, b;
```

```
    cin>>a>>b ;
```

```
    cout<<"Il massimo tra "<<a<<" e "<<b<<" e' "
```

```
        <<massimo(a,b)<<endl ;
```

```
}
```

```
int massimo(int a, int b)
```

```
{
```

```
    if (a > b)
```

```
        return a ;
```

```
    return b ;
```

```
}
```

Tipo dei parametri.

Scrivere, ad esempio,

```
int max(int a, int c) ;
```

sarebbe stato equivalente

Parametri attuali  
(*espressioni*)

Parametri formali  
(*variabili*)

# Esercizio

---

- Scrivere una funzione che verifichi se un numero naturale passato in ingresso sia primo
  - Il numero **non viene letto da *stdin* da parte della funzione!**
- La funzione deve restituire falso se il numero non è primo, vero se il numero è primo
  - Attenzione al tipo di ritorno ...
- E' proprio la funzione che ci serviva per completare il programma che abbiamo usato per introdurre l'utilità delle funzioni all'inizio di questa presentazione

# Soluzione

---

```
bool isPrime(int n)
{
    if (n>=1 && n<=3) return true; // 1,2,3: sì

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

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

    // non è stato trovato alcun divisore
    return true;
}
```

# Istruzione vuota

---

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

```
// ciclo che si ferma quando num è primo
int num = 10001
for(; ! isPrime(num) ; num += 2)
    ; // non fa nulla
// usando un'istruzione vuota nel precedente ciclo,
// abbiamo ottenuto che in questo punto del programma
// num contenga il piu' piccolo numero primo a partire
// da 10001
```

# Esercizio

---

- Scrivere una funzione radice che calcoli la radice quadrata intera di un valore naturale  $N$ 
  - Ossia il più grande intero  $r$  tale che  $r * r \leq N$
  - In altri termini, bisogna calcolare `static_cast<int>(sqrt(N))`
- Approfittiamo di questo esercizio per tornare ad evidenziare la giusta sequenza di fasi di sviluppo
  - La fase di analisi è abbastanza immediata e non sembrano esserci problemi sottili da evidenziare

# Prototipo ed idea/algoritmo

---

```
int radice(int n); // restituisce il massimo intero  
                  // x tale che x*x <= N
```

- Bozza di algoritmo

- 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:  
 $x*x \leq N$

# Proposta programma

---

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

Funziona?

# Soluzione corretta

---

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

---

# Introduzione alle tipologie di passaggio dei parametri in C/C++

# Passaggio dei parametri

---

- Per *passaggio dei parametri* si intende l'inizializzazione dei parametri formali di una funzione mediante i parametri attuali, che avviene al momento della chiamata della funzione
- L'unico meccanismo adottato in C, è il **PASSAGGIO PER VALORE**
- Come vedremo in lezioni successive, in C++ disponiamo anche del **passaggio per riferimento**

# Passaggio 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 effettua una **copia** dei valori dei parametri attuali passati dalla funzione chiamante
  - Tali copie sono sue copie private
  - Ogni modifica ai parametri formali è **strettamente locale alla funzione**
  - **I parametri attuali della funzione chiamante non saranno mai modificati!**

# Esempio 1

---

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

main()
{
    int a= 9, b= 9, c= 7, d= 12;

    cout<<a<<b<<c<<d<<endl;

    int dist =
        distanza_al_quadrato(a, b, c, d);

    cout<<a<<b<<c<<d<<dist<<endl;
}
```

Cosa viene stampato prima e dopo dell'invocazione di *CalcDistanza*?

# Esempio 1

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

```
main()
{
    int a= 9, b= 9, c= 7, d= 12;

    cout<<a<<b<<c<<d<<endl;

    int dist =
    distanza_al_quadrato(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;
    for (int i = n; i > 0; i--)
        fatt = fatt * i;
    return fatt;
}

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

Cosa viene stampato?

# Esempio 2

---

```
int fattoriale (int n)
{
    int fatt=1;
    for (int i = n; i > 0; i--)
        fatt = fatt * i;
    return fatt;
}

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

`fattoriale(4) = 24`

# Esercizio

---

- Provare a scrivere una funzione che calcoli il fattoriale utilizzando una sola variabile locale
  - In particolare definendo solo la variabile locale `fatt` utilizzata nel precedente esempio, e senza definire l'altra variabile `i`
- Per riuscirci bisogna utilizzare una tecnica sconsigliata
  - Facciamo questo esercizio solo per capire bene di cosa si tratti

# Soluzione

```
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

- 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: chi legge non capisce più se si tratta di parametri di ingresso (solo da leggere) o altro
    - Rischioso nel caso di parametri passati per riferimento (che vedremo nelle prossime lezioni)

- 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 più avanti come implementare i parametri di uscita mediante il passaggio per riferimento

# Commenti passaggio per valore

---

- E' sicuro: le variabili del chiamante e del chiamato sono *completamente disaccoppiate*
- *Consente di ragionare per componenti isolati*: 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 il passaggio per valore 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++ ;
}
```

- 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 COMPILA AFFATTO!
}
```

# Conclusione 1/2

---

- 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

# Conclusione 2/2

---

- Come capiremo meglio in seguito, il vantaggio più grande è che le funzioni forniscono il **primo strumento per gestire la complessità**
  - Sono il meccanismo di base con cui, dato un problema più o meno complesso, lo si può spezzare in sotto-problemi distinti più semplici
  - Questa è di fatto l'unica via per risolvere problemi molto complessi

- Completare gli esercizi della sesta esercitazione