# Capitolo 7

## Funzioni: Motivazione, Definizione, Prototipo, Chiamata


UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

# 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>
        ...
}
```

# Blocco

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

# Introduzione

- Per introdurre il concetto e l'importanza delle funzioni, partiamo dai problemi che si hanno se **non** si dispone delle funzioni
- In particolare, i problemi che metteremo in evidenza saranno
  - replicazione del codice
  - scarsa leggibilità del programma
- Iniziamo dallo scrivere un programma che risolve un dato problema
  - Cercheremo poi di riutilizzare il codice di tale programma per risolvere un problema più complesso

# Domanda

- Per arrivare al primo problema da risolvere, troviamo la risposta alla seguente domanda
- Cosa vuol dire che un numero intero positivo *N* è divisibile per un numero intero positivo *i*?

# Divisibilità

- Vuol dire che dividendo *N* per *i* (divisione reale), si ottiene un numero intero *j*
  - $N / i = j$
- Cioè *N* è un *multiplo* di *i*
  - $N = i * j$
- Esempi:
  - 30 è divisibile per 3: 30/3 = 10
  - 30 è divisibile per 5: 30/5 = 6
  - 9 non è divisibile per 2: 9/2 = 4.5

# Numero primo

- Un numero è **primo** se è **divisibile solo per 1 e per se stesso**
  - Esempi: 2, 3, 5, 7, 11, 13, 17, 19, …
- 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)

# Proposte?

- Qualche idea per risolvere il problema?

# Prima idea

- Dividere *N* per tutti i numeri *i* tali che 2 <= *i* <= *N*-1
  - Se nessuno di questi numeri *i* risulta essere un divisore di *N*, allora *N* è primo
- Quest'idea è praticamente già un algoritmo
- Proviamo ad implementarlo

# Programma inefficiente

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

   for(int i=2; i < n; i++)
         if (n%i == 0)
            return ; /* non primo: é stato
                        trovato
                        un divisore */
   cout<<"primo"<<endl ;
 }
```

# Domanda

- Come mai il titolo della precedente slide era “Programma inefficiente”?

# Discussione

- Perché forse, per scoprire se *n* è primo, non è necessario provare a dividere per tutti i numeri da 2 ad *N-1*
- Proposte?

# Miglioramenti 1/4

- Poiché, tranne il numero 2, i numeri pari non sono primi, possiamo controllare subito se *N* è pari
  - Se *N* è pari (e diverso da 2), allora abbiamo già scoperto che *N* non è primo
  - Invece, se *N* non è pari, allora non sarà più necessario provare a dividere *N* per un numero pari, perché, per essere divisibile per un numero pari, *N* stesso deve già essere pari

# Miglioramenti 2/4

- Inoltre, non c'è bisogno di provare tutti i numeri dispari fino ad *N*-1, ci si può fermare anche prima!
  Per capirlo consideriamo le seguenti domande
  - Quanto fa $(\sqrt{N})^2$ ?
  - Se per un possibile divisore *i*, vale la relazione $i > \sqrt{N}$, allora $i^2 > N$ ?
- Scriviamo l'ultima relazione con la sintassi del C++, ossia scriviamo $i^2 > N$ (che è vera quando $i > \sqrt{N}$), come *i*i>N*
- E' anche vero però che, se tale numero $i > \sqrt{N}$ è un divisore di *N*, significa che esiste un numero intero *j* tale che *N/i=j*, cioè tale che *i*j=N*
  - Quindi anche *j* è un divisore di *N*

# Miglioramenti 3/4

- Riassumendo, supponiamo che *i* sia un divisore con la proprietà *i***i*>N, e che quindi esista un secondo divisore *j* tale che *i***j*=*N*
- Però, siccome *i***i*>N, allora, affinché *i***j*=*N*, si deve avere *j* < *i* (altrimenti, se *j*≥*i*, allora *i***j* ≥ *i***i* > *N*)
- Ma questo vuol dire che, siccome ci siamo messi a provare tutti i possibili divisori a partire da 2, allora, siccome *j* < *i*, ci deve essere già stata una iterazione in cui abbiamo provato col valore *j* come divisore
- Quindi *j* lo avremmo già trovato come potenziale divisore **prima** di arrivare a provare un *i* così grande che *i* * *i* > *N*!

# Miglioramenti 4/4

- Quindi, se è vero che $i * j = N$ con $j < i$, allora avremmo già scoperto che il numero $N$ non era primo
  - senza bisogno di arrivare a provare con un $i$ tale che $i * i > N$
  - Ossia prima di provare con un $i > \sqrt{N}$
- In definitiva, abbiamo scoperto che è sufficiente provare a dividere $N$ per tutti i numeri dispari $3 <= i <= \sqrt{N}$
  - se nessuno di tali numeri risulta essere un divisore di $N$, allora $N$ è primo

# Problema numerico

- √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 √N perché il potenziale divisore deve essere necessariamente un numero intero!
- La parte intera di √N si può ottenere inserendo l'espressione **`static_cast<int>(sqrt(N))`**

# sqrt

- 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**

# Provate

- Solo a definire un algoritmo

# 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 √N
  - In particolare, come abbiamo detto, proviamo fino alla parte intera di √N

# 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`**

# Sfida

- 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 += 2)
         if (n%i==0) return ; /* no, perché è stato
                                 trovato
                                 un divisore */
   cout<<"primo"<<endl ;
 }
```

# Risposte

- 161531 Primo
- 419283 Non primo
- 971479 Primo

# Nota conclusiva

- Prima di procedere con l'argomento principale di questa presentazione, notiamo per l'ennesima volta la grande differenza di risultato tra
  - scrivere subito un programma inefficiente
  - fermarsi prima un po' di più a riflettere su una soluzione migliore ed arrivare ad un programma molto più efficiente

# Problema più complesso

- Proviamo ora a risolvere un problema più complesso
- Di cui il problema di “determinare se un numero è primo” è un sotto-problema

# 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

# Replicazione del codice

- Cerchiamo quindi di riutilizzare il codice già scritto per verificare se un numero è primo
- Con le conoscenze attuali possiamo ottenere il seguente risultato?
  - Riutilizzare tale codice
    - senza doverlo scrivere (o incollare) due volte nel nuovo programma
    - ottenendo un programma chiaro da capire

# Problema

- Purtroppo no
  - Per non scrivere due volte il codice potremmo utilizzare soluzioni basate su costrutti iterativi, che porterebbero però ad una ridotta chiarezza
- Ci mancano conoscenze per fare di meglio
- Prima di tutto non conosciamo nessun meccanismo per dare un nome ad un pezzo di codice e
  - richiamarlo (per farlo eseguire) da qualsiasi punto di un programma,
  - senza doverlo scrivere per intero in quel punto

# Tentativo

- Cerchiamo comunque di fare del nostro meglio con le nostre conoscenze e scriviamo il programma come meglio riusciamo

# Sfida

- 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 +=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 ;
}
```

# Risposta

- 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**
- Il codice replicato rende più difficile anche la manutenzione del programma per i motivi precedentemente discussi

# Miglioramento leggibilità

- A meno di adottare soluzioni ancora meno leggibili mediante le istruzioni iterative, non riusciamo ad eliminare la replicazione
- Proviamo almeno a rendere più leggibile il programma cercando di spiegare l'obiettivo di ciascuna parte
  - Come possiamo fare?

# Commenti

- Aggiungendo dei commenti
- Proviamo ...

# 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 n2_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 scelto bene) ci fa subito capire a cosa serve la sua invocazione
  - **Miglioramento della leggibilità**
- Dobbiamo scrivere il codice della funzione da qualche parte, ma una volta sola
  - **Eliminata la replicazione**

# Funzioni

# Concetto di funzione 1/2

- 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:

  $f$ : **A** x **B** x ... x **Q** → **S**

  - **molti ingressi, anche detti parametri, possibili** corrispondenti ai valori su cui operare
  - **una sola uscita** corrispondente al risultato o valore di ritorno
  - **A**: insieme dei possibili valori del primo parametro
  - **B**: insieme dei possibili valori del secondo parametro

    ...
  - **Q**: insieme dei possibili valori dell'ultimo parametro

# Concetto di funzione 2/2

- **S**: insieme dei possibili valori di ritorno
- Infine, il nome della funzione è tipicamente una parola
- Uno dei modi di definire una funzione è mediante la notazione matematica
- Esempi:
  - *fun(x) = x + 3*
    - Quindi, per esempio, *fun*(3) = 3 + 3 = 6
  - *fattoriale(n) = n!*
    - Quindi, per esempio, *fattoriale*(3) = 6
  - *g(x, y) = x – y*
    - Quindi, per esempio, *g*(2, 5) = 2 – 5 = -3

# Funzioni in C/C++

- Per implementare una funzione in C/C++, bisogna scrivere la sequenza di istruzioni che calcola il valore di ritorno della funzione
(ora vediamo come si fa)
- Ma in generale attraverso le istruzioni del C/C++ possiamo fare anche di più di calcolare semplicemente dei valori
  - Possiamo per esempio stampare su *stdout*
- A differenza delle funzioni matematiche, le funzioni in C/C++ sono delle parti di un programma che possono non limitarsi al semplice ritorno di un valore

# Elementi fondamentali

- Uso della funzione: **chiamata** o **invocazione**
  - Prima parte
- Definizione e dichiarazione della funzione
  - Prima parte
- Uso della funzione: **chiamata** o **invocazione**
  - Seconda parte
- Definizione e dichiarazione della funzione
  - Seconda parte
- Esecuzione della funzione (e relativo *record di attivazione*)
  - Si vedrà in una lezione successiva

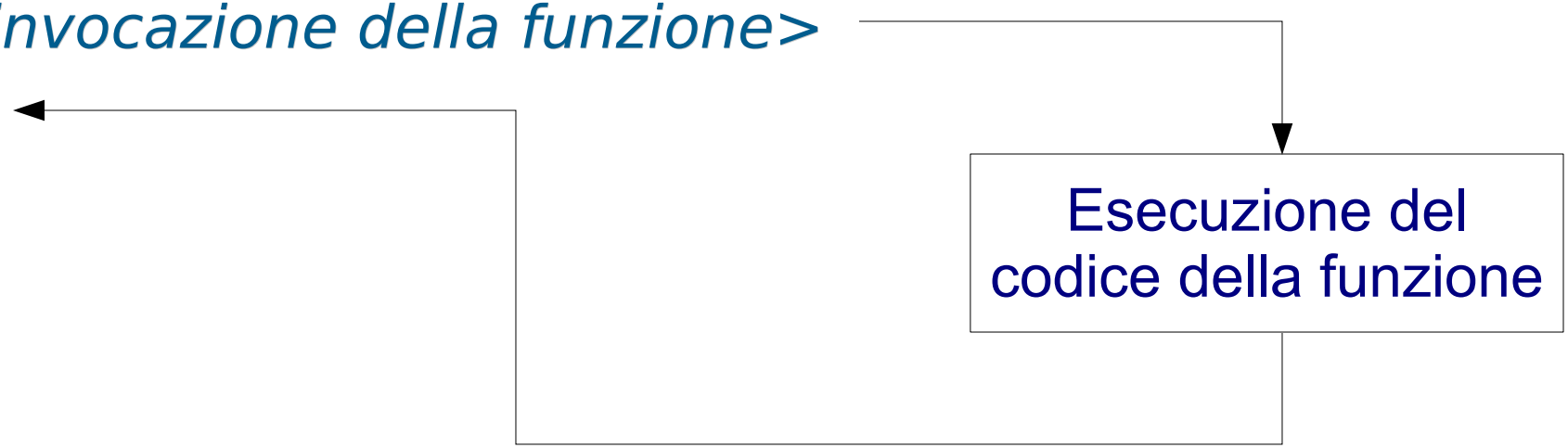
# Invocazione o chiamata

- Come vedremo meglio nelle prossime slide, l'esecuzione di una funzione consiste fondamentalmente nell'esecuzione di un frammento di codice
- Per far partire l'esecuzione di una funzione bisogna eseguire una **invocazione** o **chiamata** della funzione

# Schema esecuzione

- Lo schema di esecuzione di una funzione è il seguente:

*...*
*<invocazione della funzione>*
*...*




- Ossia, a seguito dell'invocazione si esegue il codice della funzione, dopodiché il controllo torna all'istruzione successiva all'invocazione stessa

# Elementi fondamentali

- Uso della funzione: **chiamata** o **invocazione**
  - Prima parte
- Definizione e dichiarazione della funzione
  - Prima parte
- Uso della funzione: **chiamata** o **invocazione**
  - Seconda parte
- Definizione e dichiarazione della funzione
  - Seconda parte

# 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 dell'intestazione
- Vedremo poi tutti i dettagli formali

# Primo esempio di intestazione

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

**Tipo del risultato** — **Nome della funzione** — **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`**

# Parametro formale

- Cosa vuol dire che la funzione ha un parametro formale?
  - Che nel nostro esempio è di tipo **int** ed è chiamato **`n`**
- Vuol dire che quando la funzione viene invocata, le dovrà obbligatoriamente essere passato un valore
  - Nel nostro esempio, un valore di tipo **int**
- Tale valore sarà memorizzato nel parametro formale
  - Nel nostro esempio, nel parametro formale di nome **`n`**

# Uso del valore di ritorno 1/3

- Cosa vuol dire che una funzione ritorna un dato valore?
- **Non vuol dire che lo stampa su *stdout*!**
- La semantica del valore di ritorno è invece la seguente
- L'invocazione di una funzione è una espressione. Ad esempio, l'invocazione di funzione
  `fattoriale(3)`
  è una espressione
- Il valore di tale espressione è dato dal valore di ritorno della funzione

# Uso del valore di ritorno 2/3

- In particolare l'invocazione di una funzione costituisce un *fattore*, che si può a sua volta scrivere anche all'interno di espressioni più complesse
- Consideriamo quindi una espressione che contiene l'invocazione della funzione
- La funzione viene invocata quando, in base all'ordine di valutazione degli operatori e dei fattori presenti nell'espressione, è necessario utilizzare il valore del fattore rappresentato dall'invocazione della funzione

# Uso del valore di ritorno 3/3

- Il valore del fattore sarà uguale al valore di ritorno della funzione

  Ad esempio, se *fattoriale(n)* è una funzione che ha per valore di ritorno *n!*, allora:

```
cout<<fattoriale(3)*2<<endl;
```

  stampa `12`

# Continuiamo

- Vediamo ora esempi più complessi

# Esempi di intestazioni 1/3

```
int    somma    (int x, int y)
```

**Tipo del risultato** — **Nome della funzione** — **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`**

# Esempi di intestazioni 2/3

```
void   stampa_n_volte   (int n)
```

**Tipo del risultato** — **Nome della funzione** — **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, non si potrà usare in una espressione)
- All'inizio dell'esecuzione della funzione il valore preso in ingresso sarà memorizzato nel parametri formale **`n`**

# Procedura

- Altri linguaggi (ma non il C/C++!) introducono separatamente anche l'astrazione di **procedura**
  - Esecuzione di un insieme di azioni, senza ritornare esplicitamente un risultato
  - Esempio: semplice stampa di valori su *stdout*
- In C/C++ le procedure sono realizzate mediante le funzioni con tipo di ritorno **vuoto**
  - Come le funzioni **`stampa_n_volte`** e **`stampa_2_volte`** riportate nelle slide precedenti e nella prossima slide

# Esempi di intestazioni 3/3

```
void   stampa_2_volte   (void)
```

**Tipo del risultato** — **Nome della funzione** — **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**, quest'ultimo definito mediante un blocco

*<definizione-funzione>* ::=
*<intestazione-funzione>* *<blocco>*

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

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

Una delle tre opzioni

Zero o più volte

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

# Intestazione

- L'intestazione specifica nell'ordine:
  - **Tipo del valore di ritorno**
    - **`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

# Valore di ritorno

- Vedremo a breve come si stabilisce il valore ritornato da una funzione

# Corpo di una funzione

- Il blocco che definisce il corpo di una funzione è di fatto una istruzione composta
- I parametri formali sono visibili, possono cioè essere utilizzati, all'interno del corpo della funzione
  - come normali variabili
- Quindi, come stiamo per vedere in dettaglio, tramite i parametri formali il codice della funzione può leggere i valori passati alla funzione stessa

# Esempio di definizione

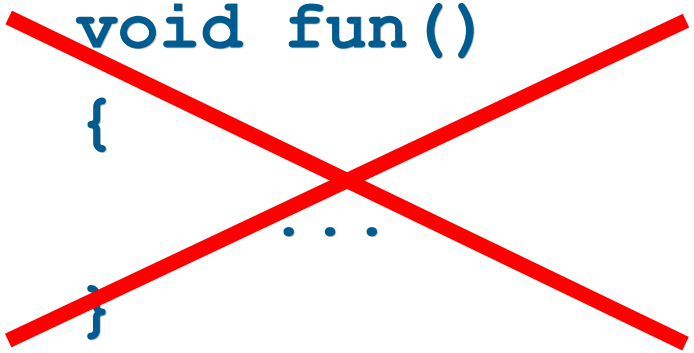
```
// la seguente funzione stampa il
// valore che le viene passato
void fun(int a)
{
    cout<<a<<endl ;
}
```

# Posizione definizioni

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

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

    fun() ;
}
```



# Elementi fondamentali

- Uso della funzione: **chiamata** o **invocazione**
  - Prima parte
- Definizione e dichiarazione della funzione
  - Prima parte
- Uso della funzione: **chiamata** o **invocazione**
  - Seconda parte
- Definizione e dichiarazione della funzione
  - Seconda parte

# 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>* )
*<lista-parametri-attuali>* ::=
  *<nessun parametro>* | *<parametro>* { , *<parametro>* }

- Un parametro attuale può essere una qualsiasi **espressione**
- I parametri attuali sono utilizzati per inizializzare i parametri formali della funzione ...

# Esempi

```
…
// quando eseguita, la seguente riga di
// codice invoca una funzione fun1, che
// supponiamo abbia unico parametro
// formale, e passa come unico parametro
// attuale il valore 7
fun1(7) ;

int n ;
cin>>n ;
fun1(n - 3) ; // ora si passa invece il
              // valore dell'espressione
              // n - 3
```

# Uso più semplice

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

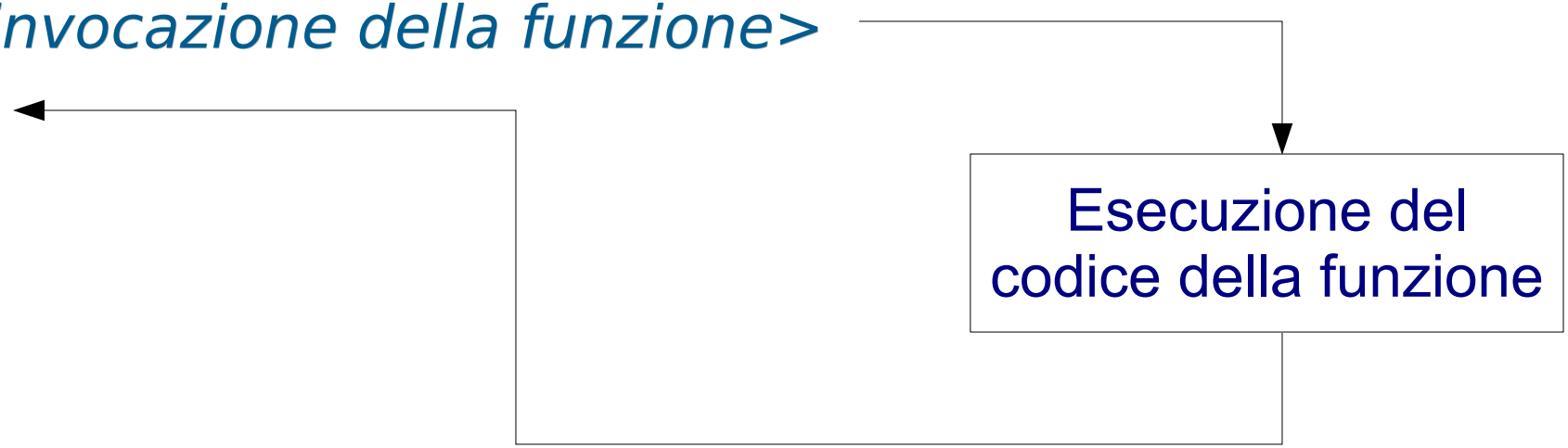
*<nomeFunzione>* ( *<lista-parametri-attuali>* ) ;

- 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
  - Come nello schema già visto, e mostrato di nuovo nella prossima slide

# Schema esecuzione

- Lo schema di esecuzione di una funzione è il seguente:

  ...
  *<invocazione della funzione>*
  ...




- Ossia, a seguito dell'invocazione si esegue il codice della funzione, dopodiché il controllo torna all'istruzione successiva all'invocazione stessa

# Domanda

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

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

*Invocazione corretta?*

# Risposta

- No, l'invocazione è costituita dal **solo nome** della funzione, con tra parentesi i valori che vogliamo passare alla funzione

# 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() ;
}
```

# 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 ;
}
```

# Esecuzione funzione

- L'invocazione di una funzione comporta i seguenti passi (definiti più in dettaglio nelle prossime slide):

1) Si **calcola** il valore di ciascuno dei parametri attuali (che sono in generale espressioni)

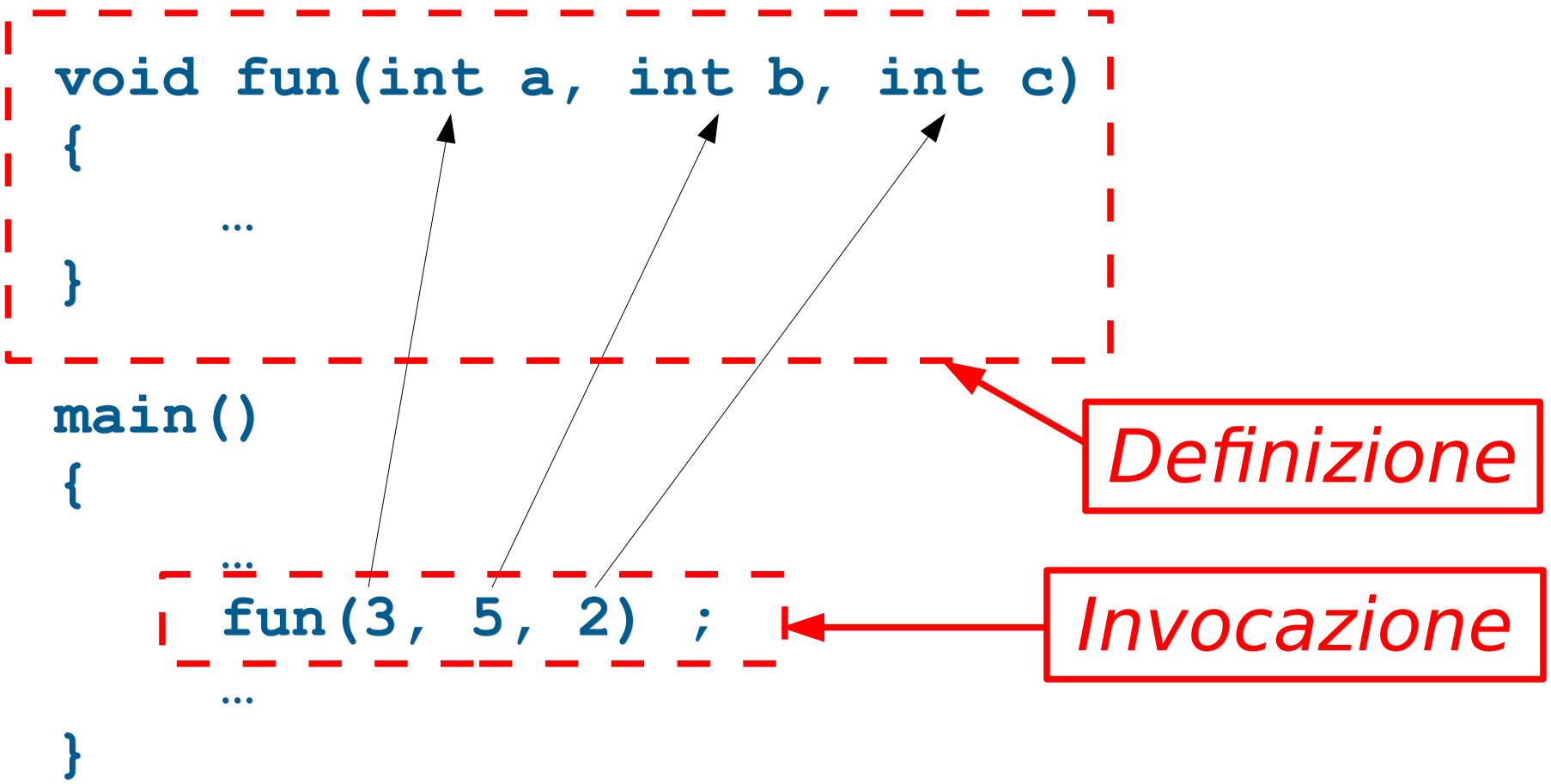
2) Si **definisce** ciascun parametro formale della funzione e lo si **inizializza** con il valore del parametro attuale che si trova nella stessa posizione

3) Si esegue la funzione

# Parametri formali ed attuali

- In una chiamata di funzione si dovranno inserire tanti parametri attuali quanti sono i parametri formali della funzione
- Ogni parametro formale della funzione sarà **inizializzato** con il parametro attuale *nella stessa posizione* nella chiamata, prima di iniziare ad eseguire la funzione stessa
- Visualizziamo la cosa con due esempi ...

# Esempio 1


void fun(int a, int b, int c)
{
…
}
main()
{
…
fun(3, 5, 2) ;
…
}
Definizione
Invocazione

# Esempio 2

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

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

*Definizione*

*Invocazione*

- La funzione **fun**, e quindi l'intero programma, stampano **3**

# 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 di elementi, e che il tipo di ogni parametro attuale sia compatibile con il tipo del corrispondente parametro formale (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 da quelli dei parametri formali. **Conta solo la posizione all'interno della chiamata**

# Esempi

- 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) ;
}
```

# Funzioni non void

- Nel caso di funzione con tipo di ritorno diverso da void, lo schema dell'esecuzione della funzione va completato come segue

1) Si **calcola** il valore di ciascuno dei parametri attuali (che sono in generale espressioni)

2) Si **definisce** ciascun parametro formale della funzione e lo si **inizializza** con il valore del parametro attuale che si trova nella stessa posizione

3) Si esegue la funzione

4) Se l'invocazione della funzione fa parte di una espressione, si sostituisce l'invocazione della funzione con il valore ritornato dalla funzione

   - Vediamo in dettaglio ...

# Funzioni ed espressioni 1/2

- In generale, una chiamata di una funzione è una **espressione**
- Si può inserire a sua volta in una espressione
  - ma solo a patto che la funzione ritorni effettivamente un valore
  - ossia che il suo tipo di ritorno non sia **void**
- Riepiloghiamo e completiamo quanto già spiegato sull'uso di una invocazione di funzione all'interno di una espressione

# Funzioni ed espressioni 2/2

- Nel caso in cui una chiamata di funzione sia effettivamente presente in una espressione e la funzione non sia di tipo **void**
  - Il valore di ritorno della funzione costituisce un fattore dell'espressione
  - La funzione è invocata quando bisogna calcolare il valore di tale fattore
  - Il valore del fattore sarà uguale al valore di ritorno della funzione
- In particolare, al termine dell'esecuzione della funzione e dopo aver quindi calcolato il fattore corrispondente alla chiamata della funzione, riprende il calcolo del valore dell'espressione in cui la chiamata di funzione è inserita

# Esempio

```
…
// se la funzione invocata nella
// seguente istruzione ritorna,
// per esempio, 4, allora
// l'istruzione stampa 8

cout<<(2*fun(3))<<endl;
```

# Valore di ritorno

- Ma come si stabilisce il valore di ritorno di una funzione?

# Istruzione `return` 1/3

- 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**
- Se la funzione ha tipo di ritorno diverso da `void`, è mediante l'istruzione `return` che si determina il valore di ritorno della funzione

# Istruzione **`return`** 2/3

- Sintassi nel caso di funzioni con tipo di ritorno **`void`**:
  **`return ;`**

- Sintassi nel caso di funzioni con tipo di ritorno diverso da **`void`**:

  **`return`** <*espressione*> **`;`**

  - Il tipo del valore dell'espressione deve coincidere col tipo del valore di ritorno specificato nell'intestazione della funzione

    - O essere perlomeno compatibile, come vedremo in seguito parlando delle conversioni di tipo

# Istruzione **`return`** 3/3

- 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

# Esercizio su valore di ritorno

- Scrivere un programma in cui
  - Si definisce una funzione di nome **`ritorna_2`**, che
    - non prende alcun parametro in ingresso
    - ritorna il valore 2
  - Da dentro la funzione **`main`**, si invoca la funzione **`ritorna_2`** e si stampa il valore ritornato da tale funzione

- Domande a supporto della scrittura della soluzione
  - Di che tipo deve essere il valore di ritorno della funzione?
  - Mediante quale istruzione si stabilisce il valore ritornato dalla funzione?

# Risposte

- Di tipo **int**
- Mediante l'istruzione **`return`**
- Nella prossima slide la soluzione dell'esercizio

# Soluzione

```
int ritorna_2()
{
     return 2;
}

int main()
{
     cout<<ritorna_2()<<endl ;
}
```

# Elementi fondamentali

- Uso della funzione: **chiamata** o **invocazione**
  - Prima parte
- Definizione e dichiarazione della funzione
  - Prima parte
- Uso della funzione: **chiamata** o **invocazione**
  - Seconda parte
- Definizione e dichiarazione della funzione
  - Seconda parte

# 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 costante
    - Esempio di intestazione di funzione con parametro formale costante:
      `int fun(const int a)`
  - 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.

# 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 ;
}
```

# Domanda

- Qual è l'unica differenza tra le due definizioni?

# 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*

# Esercizi e consigli

- Svolgere la sesta esercitazione fino all'esercizio *somma_quadrati.cc* escluso
- Link alla videoregistrazione: https://drive.google.com/file/d/1BITkPbgE3OQoyQGdLE38co4llrgV3eFn/view?usp=sharing
- Da qualche esercizio non vi stiamo più dando ogni volta 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**

# Fine dell'introduzione degli elementi fondamentali di una funzione

# Progetto di una funzione

# Domanda

- Se una funzione
  - lavora su un certo valore in ingresso
  - e, quando si progetta la funzione, si può scegliere tra
    - Far leggere tale valore alla funzione da *stdin*
    - Far ricevere tale valore in ingresso dalla funzione attraverso un parametro formale
- Quale delle due soluzioni è migliore?

# Soluzione migliore ingresso 1/2

- La soluzione migliore è la seconda
  - La funzione può essere utilizzata ovunque all'interno del programma passandole il valore che si preferisce
  - Non è necessario dover leggere obbligatoriamente qualcosa da *stdin*
  - Eventuali letture da *stdin* si possono effettuare semplicemente nel main o in generale in altre funzioni il cui scopo è proprio quello di leggere qualcosa da stdin

# Soluzione migliore ingresso 2/2

- Pensate ad  esempio all'uso della funzione sqrt nel programma che verifica se un numero è primo
- Se la funzione non avesse avuto un parametro formale tramite il quale passarle il valore su cui lavorare
  - Ma lo avesse letto da *stdin*
- Come saremmo riusciti a scrivere il programma in maniera tale che chiedesse, correttamente, una sola volta il numero su cui lavorare?
  - Ossia il numero del quale stabilire se fosse primo oppure no

# Risposta

- Non ci sarebbe stato **alcun modo**

# Domanda

- Se una funzione
  - fornisce un certo valore in uscita
  - e, quando si progetta la funzione, si può scegliere tra
    - Far scrivere tale valore su *stdout*
    - Far restituire tale valore alla funzione attraverso l'istruzione *return*
- Quale delle due soluzioni è migliore?

# Soluzione migliore uscita 1/2

- La soluzione migliore è la seconda
  - La funzione può essere utilizzata ovunque all'interno del programma, leggendo ed eventualmente memorizzando in una variabile il valore da essa ritornato
  - Non è necessario dover necessariamente stampare qualcosa su *stdout*
  - Eventuali scritture su *stdout* si possono effettuare semplicemente nel *main* o in generale in altre funzioni il cui scopo è proprio quello di scrivere qualcosa su *stdout*

# Soluzione migliore uscita 2/2

- Pensate di nuovo all'uso della funzione *sqrt* nel programma che verifica se un numero è primo
- Se la funzione non avesse ritornato la radice quadrata del numero passato in ingresso
  - Ma avesse stampato il risultato su *stdout*
- Saremmo riusciti a scrivere il programma?

# Risposta

- No

# Dichiarazione di una funzione

# Il main è una funzione

- La prima istruzione della funzione **main** è la prima istruzione dell'intero programma
- Le variabili definite nella funzione **main** hanno valori casuali
- Quando la funzione **main** termina, tutto il programma termina
- In un programma corretto, la funzione **main** ha tipo di ritorno **int**
- Il valore intero ritornato dalla funzione **main** coincide col valore restituito dal processo quando termina

# Chiamate incrociate 1/2

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

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

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

# Chiamate incrociate 2/2

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

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

- *Purtroppo no ...*

# Dichiarazione 1/2

- Come abbiamo già detto a suo tempo, una definizione è una 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 della funzione
- 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 un punto e virgola

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

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

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

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

***Opzionale !***

# Esempi di prototipi

```
int  fattoriale (int);

main()
{
       … // invocazione funzione fattoriale
}

int  fattoriale   (int n)
{
       int fatt=1;
       for (int i=1; i<=n; i++)
              fatt = fatt*i;
       return(fatt);
 }
```

---

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

# Soluzione chiamate incrociate

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

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

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

# Prototipi e definizioni

- Il prototipo:
  - è un puro “avviso ai naviganti”
  - **non causa la produzione di alcun byte di codice eseguibile**
  - può essere ripetuto più volte nel programma (basta che non ci siano due dichiarazioni in contraddizione)
  - può comparire anche dentro un'altra funzione (non usiamolo in questo modo)

- 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 importante in una definizione

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

# Utilizzo di intestazione e corpo di una funzione da parte del compilatore

# Domanda

- Quali elementi di una chiamata di funzione deve controllare il compilatore per essere sicuro che la funzione sia invocata in modo sintatticamente corretto?

# Risposta

- Numero di parametri attuali
  - Deve essere uguale al numero di parametri formali
- Tipo di ciascun parametro attuale
  - Il tipo di ciascun parametro attuale deve essere compatibile col tipo del parametro formale nella posizione corrispondente
- Tipo del valore atteso nel punto del programma in cui si utilizza il valore di ritorno della funzione
  - Tale valore di ritorno non si può utilizzare affatto se la funzione ha tipo di ritorno *void*

# Domanda

- Cosa è sufficiente conoscere, da parte del compilatore, per accertarsi che l'invocazione di una funzione sia sintatticamente corretta in merito agli aspetti evidenziati nella precedente slide?

# Risposta

- Tutte e sole le informazioni contenute nell'intestazione, ossia nella dichiarazione, della funzione!
  - Numero dei parametri formali
  - Tipo di ciascun parametro formale
  - Tipo di ritorno della funzione

# Controllo sintattico invocazione

- La precedente risposta è il motivo fondamentale per cui l'unico vincolo posto dal compilatore per poter inserire correttamente l'invocazione di una funzione in un dato punto del programma è che la dichiarazione della funzione preceda, nel testo del programma, il punto in cui la funzione è invocata
- Tale vincolo è imposto per aumentare la capacità del compilatore di **trovare subito** errori *sintattici* commessi dal programmatore
  - Spesso tali errori sintattici scaturiscono da errori concettuali, che vengono così rilevati immediatamente

# Definizione corpo

- Al compilatore interessa in prima battuta di controllare solo la correttezza sintattica delle invocazioni, e per questo bastano solo le intestazioni delle funzioni come abbiamo visto
- In quanto al corpo, ossia al codice vero e proprio di una funzione, al compilatore basta solo trovare prima o poi, nel testo del programma, la definizione di tale corpo (che verrà tradotto in linguaggio macchina)
  - Quando lo trova, traduce ciascuna invocazione della funzione in un *salto* all'esecuzione di tale corpo (ed altre operazioni accessorie)
  - Se non lo trova, allora segnala un errore
- Si può quindi definire il corpo di una funzione dove si vuole nel testo del programma

# 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 massimo(int a, int c);`
sarebbe stato equivalente

Parametri attuali
(*espressioni*)

Parametri formali
(definizioni di *variabili*)

# Esercizio

- Scrivere una funzione che verifichi se un numero naturale passato in ingresso come parametro attuale è 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 ...
- Utilizzando tale funzione, riscrivere il programma che controlla se due numeri primi sono gemelli

# Soluzione funzione

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

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

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

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

# Resto del 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 ;
}
```

- Utilizzando la funzione abbiamo scritto in modo leggibile e senza replicazione del codice il nostro programma di verifica se due numeri sono primi gemelli

# Esercizio da fare assieme

- 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* <= *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 <= 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 1/2

```
int distanza_al_quadrato(int px1, int py1, int px2, int py2)
{
  px1 = pow (px1 - px2, 2); // pow(x, y) = x^y
  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 *distanza_al_quadrato*?

# Esempio 2/2

```
int 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**

# Esercizio

- Provare a scrivere una funzione che prenda in ingresso (come parametro formale) un numero naturale *n*, e nessun altro parametro, e ritorni la somma dei numeri da 1 ad *n*
  - Definendo **una sola variabile locale** nella funzione
  - Senza utilizzare la chiusura della sommatoria
  - Quella che conterrà il risultato
- Per riuscirci bisogna utilizzare una tecnica sconsigliata
  - Facciamo questo esercizio solo per capire bene di cosa si tratti

# Soluzione

```
int somma (int n)
{
   int somma = 0;
   // utilizzo decremento del parametro formale
   for (; n > 0; n--)
      somma += n;
   return somma;
}
```

Cosa viene stampato?

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

# Soluzione

```
int somma (int n)
{
   int somma = 0;
   // utilizzo decremento del parametro formale
   for (; n > 0; n--)
      somma += n;
   return somma;
}


main()  {
   int risultato, n = 4 ;
   risultato = somma(n);
   cout<<"somma("<<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:*

```
somma(4) = 10
```

# Nota 1/2

- Abbiamo visto la modifica di un parametro formale variabile all'intero di una funzione solo per capire: 1) che la cosa si può fare, e 2) 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
    - Crea effetti collaterali 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 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 persino modificare i parametri ricevuti senza che ciò abbia alcun impatto sul chiamante)

- LIMITI
  - Impedisce *a priori* di scrivere funzioni che abbiano come scopo proprio quello di modificare variabili utilizzate poi nella funzione da cui sono invocate
  - 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++ ;
}
```

# 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 SI COMPILA AFFATTO!
}
```

# Domanda

- Il seguente programma è corretto?

```
void fun(int);

int main()
{
      fun(3) ;
      return 0;
}

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

# Risposta

- No, perché il prototipo della funzione **`fun`** e l'intestazione della funzione **`fun`** nella definizione non coincidono
  - Nel prototipo manca il qualificatore **`const`**

# 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

# Esercizi

- Completare la sesta esercitazione
- Link alla videoregistrazione: https://drive.google.com/file/d/1ABGG3RG3p8M9j2Ypxglxep_Bu2enZtLv/view?usp=sharing