

8. Funzioni (parte prima)

Andrea Marongiu

(andrea.marongiu@unimore.it)

Paolo Valente

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)

Prima idea

- Dividere N per tutti i numeri i tali che
- $2 \leq i \leq N-1$
 - 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 ;  
}
```

Miglioramenti 1/3

- 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, perché, per essere divisibile per un numero pari, N stesso deve già essere un numero pari
- 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 $i > \sqrt{N}$, allora $i^2 > N$?

Miglioramenti 2/3

- Il fatto che $i^2 > N$ quando $i > \sqrt{N}$, si può riscrivere 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 j tale che
 - $N/i=j$, cioè 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, se ci siamo messi a provare tutti gli i a partire da 1, allora, siccome $j < i$, ci deve essere già stata una iterazione in cui i era uguale a j
 - E questo vuol dire che j lo avremmo già provato come potenziale divisore nel nostro algoritmo
 - **Prima** di arrivare a provare un i così grande che $i * i > N!$

Miglioramenti 3/3

- Quindi, se è vero che $i * j = N$ con $j < i$, allora avremmo già scoperto che il numero 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
- $3 \leq i \leq \sqrt{N}$
 - 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 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`

Programma numero primo

```
main()  
{  
    int n ; cin>>n ;  
  
    if (n>=1 && n<=3) { cout<<"primo"<<endl ; return ; }  
  
    if (n%2 == 0) return 0;    /* 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 0; /* no, perché è stato  
                                trovato  
                                un divisore */  
  
    cout<<"primo"<<endl ;  
}
```


Prova e verifica questi numeri

- 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

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 **riutilizziamo sempre il codice già disponibile**
 - Però sempre con spirito critico
- Esempio: non ci siamo riscritti in ogni nostro programma il codice di invio di caratteri su *stdout*, ma abbiamo (ri)utilizzato 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 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

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 ;
}
```

Prova questi valori

- Quali delle seguenti coppie di numeri è costituita da primi gemelli?
 - 11057 e 11059
 - 11059 e 11061
- 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
- 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 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:
 - $f: \mathbf{A} \times \mathbf{B} \times \dots \times \mathbf{Q} \rightarrow \mathbf{S}$
 - **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

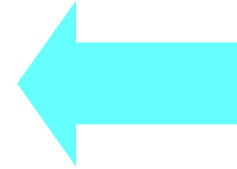
- **S**: insieme dei possibili valori dell'uscita ritorno
- Infine, il nome della funzione è tipicamente una parola
- Uno dei modi di definire una funzione è mediante la notazione matematica
- Esempi:
 - $\text{fun}(x) = x + 3$
 - Quindi, per esempio, $\text{fun}(3) = 3 + 3 = 6$
 - $\text{fattoriale}(n)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

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



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 Nome della Lista dei parametri
risultato funzione 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**

Significato del valore di ritorno

- Cosa vuol dire che una funzione ritorna un dato valore?
- **Non vuol dire che lo stampa su *stdout*!**
- Vuol dire che, se il nome della funzione appare in una espressione, allora quando l'espressione viene valutata, il nome della funzione sarà sostituito dal valore di ritorno della funzione stessa
- Esempio, se *fattoriale(n)* è una funzione che ha per valore di ritorno *n!*, allora:
 - `cout<<fattoriale(3)*2<<endl;`
 - stampa 12

Esempi di intestazioni 1/3

`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`

Esempi di intestazioni 2/3

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

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( )
```

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

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

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

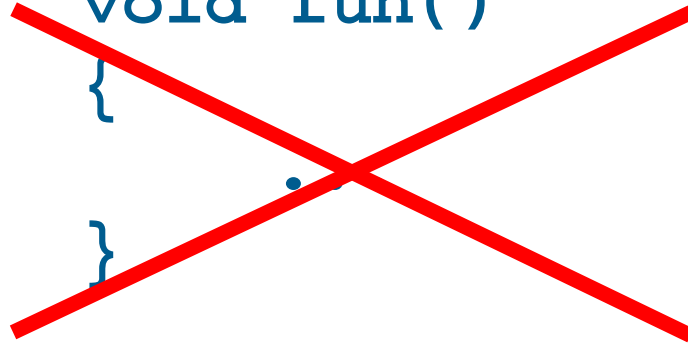
Esempio di definizione

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

Posizione definizioni

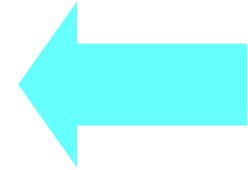
- Una funzione non può essere definita all'interno di un'altra funzione

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



Elementi fondamentali

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

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() ;  
}
```

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**

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;
```

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**
- 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` 1/2

- 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` 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

Invocazione 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
 - 4) Se la funzione ritorna un valore e la sua invocazione fa parte di una espressione, si sostituisce l'invocazione della funzione con il valore ritornato dalla funzione

Numero di parametri attuali

- In una chiamata di funzione si dovranno inserire tanti parametri attuali quanti sono i parametri formali della funzione
- In particolare non si inserirà nessun parametro attuale nel caso di funzioni che non hanno nessun parametro formale (ingresso)
- Vediamo un semplice esempio completo di definizione ed invocazione di una funzione

Esempio

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

Definizione

```
int main()
{
    Int i, k, l
    Fun(3, i*k, l--) ;
}
```

Invocazione

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) ;
}
```

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)
```

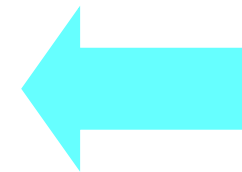
```
{
```

```
    cout<<a<<endl ;
```

```
}
```


Elementi fondamentali

- Definizione e dichiarazione della funzione
 - Prima parte
- Uso della funzione: **chiamata o invocazione**
- 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 ;
}
```

i++ ;

Istruzione legale

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

int 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