

Programmazione  
orientata agli oggetti  
e cenni al C++

M. Falda\*

A.A. 2004/2005

Programmazione orientata agli oggetti e C++

1 / 30

\* [marco.falda@unipd.it](mailto:marco.falda@unipd.it).

## Prospettiva

- Introduzione e motivazione.
- Oggetti e classi.
- Struttura dei sistemi ad oggetti.
- Astrazioni.
- Panoramica su C++ e Java.

Dapprima cercherò spiegarvi perché viene usata la OOP e come può essere utile nella programmazione.

Poi introdurrò gli oggetti e le classi, che sono i fondamenti della OOP.

Gli oggetti interagiscono tra loro in un sistema, per cui vi farò vedere i meccanismi ideati per far comunicare gli oggetti tra loro e di metterli in relazione.

Un altro aspetto caratteristico sono le astrazioni, che permettono di costruire correttamente gli oggetti e di derivarne di nuovi dagli esistenti.

Infine qualche codice d'esempio per farvi vedere come si applica la teoria in pratica.

## La strada dell'*hardware*

- Gli ultimi processori presentano più di 100 milioni di *transistors*;
- enormi progressi nell'*hardware* grazie anche a componenti flessibili e riutilizzabili.

Si sa che l'*hardware* ha fatto passi da gigante negli ultimi anni. Questo è certamente dovuto alle nuove tecniche che hanno permesso di miniaturizzare i circuiti.

Tuttavia non si sarebbero potuti creare e gestire sistemi così complessi senza adottare una metodologia di progettazione basata su componenti autonomi. Oggi esistono programmi che permettono ai progettisti di lavorare sulle unità costitutive di un processore senza occuparsi dei singoli *transistor*, i quali vengono disposti automaticamente sul *chip*. Se si osserva un processore moderno al microscopio si nota una struttura regolare indotta dai componenti.

## Applicazione al *software*

- Nella programmazione strutturata spesso si incontrano situazioni di analogia:
  - si possono astrarre le parti in comune e costruire delle librerie di codice;
  - una volta scritte tali librerie possono essere riusate molte volte applicandole a casi specifici.

Anche il progetto di sistemi software può beneficiare di un approccio a componenti. Il concetto chiave da cui partire è che durante la stesura di programmi spesso ci si rende conto di scrivere codice ripetuto.

Si potrebbero risparmiare tempo ed errori se si organizzasse tale codice in librerie per poi poterlo riusare più volte. Il problema è che non basta strutturare tali librerie mediante funzioni tradizionali, perché è facile che le funzioni includano dipendenze a livello globale.

## Utilità della OOP

È un diverso modo di progettare i programmi:

- si pensa meglio al problema mettendo in secondo piano il calcolatore;
- robusto al variare dei requisiti nel tempo perché modulare;
- è il metodo più usato nella moderna progettazione.

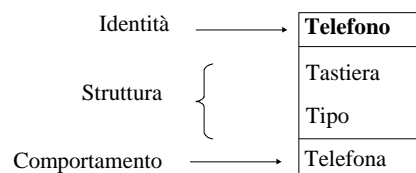
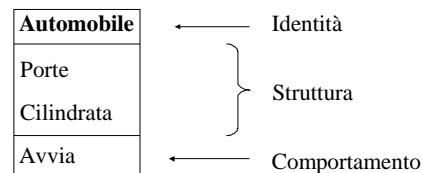
Un metodo più recente rispetto alla programmazione procedurale è quello della progettazione orientata agli oggetti.

L'intento di questo metodo è quello di spostare l'attenzione dal calcolatore al problema da modellare. Un secondo scopo è quello di strutturare il sistema in modo che sia facilmente adattabile al variare dei requisiti, che cambiano spesso soprattutto nel caso di sistemi complessi.

Per questi motivi la programmazione orientata agli oggetti è il metodo più usato per costruire i moderni *software*.

# Oggetti e realtà

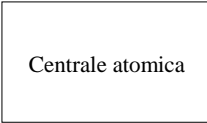
Gli oggetti aiutano a modellare il mondo reale



Iniziamo con il dire che noi stessi tendiamo a pensare tramite oggetti, ad esempio un telefono si distingue da un modem perché ha una tastiera (ed una cornetta), può essere di vari tipi, per esempio tradizionale o ISDN. L'entità telefono ha poi un caratteristica azione che è quella di poter telefonare (e ricevere telefonate).

## Divide et impera

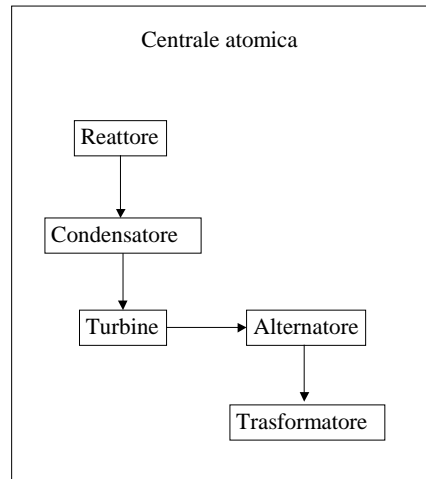
- Un sistema complesso è difficilmente trattabile come una sola unità;
- si decompone il sistema in oggetti compiuti in sè stessi.



Centrale atomica

La nota frase di Cesare “divide et impera” ben si presta ad essere applicata alla OOP, infatti è difficile modellare un sistema complesso, quale ad esempio la simulazione di una centrale atomica, come un’unica entità. È meglio decomporlo in oggetti autonomi che interagiscono tra loro.

## Divide et impera



Programmazione orientata agli oggetti e C++

8 / 30

Per completare l'esempio citato, una centrale atomica potrebbe essere scomposta in cinque oggetti che racchiudono cinque fasi della produzione energetica: il reattore riscalda l'acqua e il condensatore ritrasforma il vapore prodotto in acqua. Il vapore fa girare le turbine; il movimento delle turbine si trasforma in elettricità grazie all'alternatore. Infine l'elettricità è predisposta per la distribuzione tramite un trasformatore.



## Utilità degli oggetti

- Mascheramento: nascondono i particolari algoritmi usati.
- Indipendenza: definiscono facilmente le parti riusabili.
- Modularità: possono essere sostituiti.
- Località: adatti ad uno sviluppo iterativo (gestione migliore del codice e degli errori).

Perché gli oggetti sono utili allora? Le caratteristiche positive si possono riassumere in quattro punti.

Gli oggetti permettono di nascondere al loro interno gli algoritmi che realizzano il loro comportamento, così che ad esempio anche persone meno esperte possano farne uso.

La loro indipendenza fa sì che si possano costruire librerie di codice da usare quando ce ne sia la necessità.

Viceversa, si possono facilmente individuare e sostituire gli algoritmi negli oggetti e questo permette una più semplice gestione del codice.

Modularità ed indipendenza rendono il codice localizzato, vale a dire che i singoli oggetti racchiudono in sé stessi tutto ciò che serve per funzionare. Per questo motivo è più facile individuare gli errori, perché questi dipenderanno solo dall'oggetto che contiene il codice problematico.

## Gli oggetti e le classi

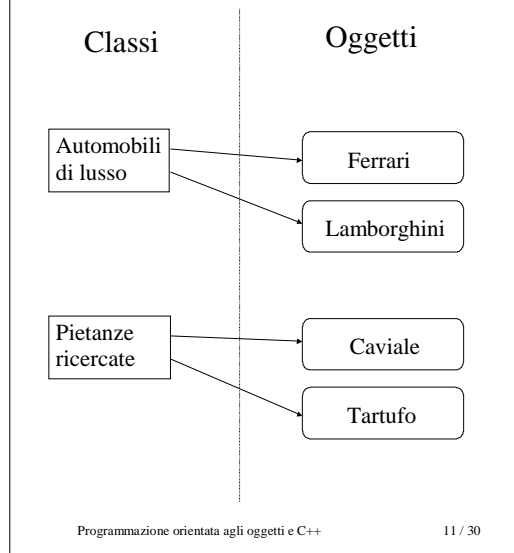
- Gli oggetti sono scatole nere:
  - l'unica parte visibile è la loro interfaccia.
- Ogni oggetto deriva da una classe (è una sua istanza).
- La classe descrive la struttura e il comportamento di un oggetto.

Gli oggetti sono come scatole nere: non possiamo guardare al loro interno, ma solo comunicare con esse tramite una interfaccia.

Una classe è una entità che descrive in maniera generale un oggetto ed in particolare la sua struttura e il suo comportamento.

Una classe è una entità generale e viene “istanziata” in un oggetto specifico.

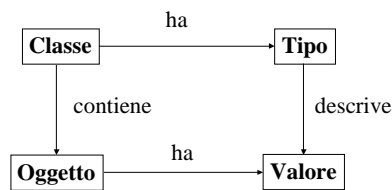
## Esempi di classe



Abbiamo detto che le classi descrivono le proprietà ed il comportamento degli oggetti. In questo esempio vediamo a destra alcuni oggetti che hanno qualcosa in comune. I primi due sono automobili di classe, veloci e costose e possiamo raggrupparle insieme come automobili di lusso. Analogamente gli ultimi due oggetti si possono accomunare come pietanze ricercate.

## Gli oggetti e le classi

- Gli oggetti sono caratterizzati da:
  - proprietà statiche: struttura;
  - proprietà dinamiche: comportam.
- Una classe raccoglie oggetti dello stesso tipo.



Programmazione orientata agli oggetti e C++

12 / 30

Ciò che caratterizza un oggetto è la sua struttura e il suo comportamento.

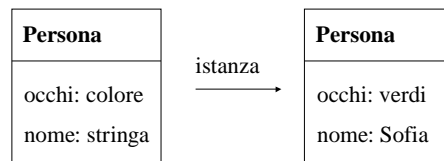
La struttura è costituita da proprietà statiche, ad esempio se diciamo che una persona si distingue per il colore degli occhi e per la sua altezza, qualsiasi persona avrà occhi di un certo colore e misurerà una certa altezza.

Il comportamento è formato dalle proprietà dinamiche, perché varia nel tempo a seconda dello stato interno e delle informazioni che si comunicano.

Riassumendo, abbiamo detto che una classe contiene un insieme di oggetti. Ogni oggetto ha valori specifici che lo caratterizzano. Questi valori appartengono ad un certo tipo, ad esempio gli occhi hanno un colore. Quindi le proprietà di una classe sono un insieme di tipi.

## Proprietà

- Le proprietà modellano le caratteristiche di un oggetto;
- negli oggetti esse acquisteranno dei valori (verranno istanziate).



Le proprietà caratterizzano un oggetto e gli oggetti sono istanze di classi. Così un oggetto specifico si distingue per i valori che le sue proprietà assumono: le proprietà di una classe vengono istanziate in un oggetto assumendo dei valori. L'insieme dei valori di un oggetto ne descrive lo stato.

Ritornando all'esempio precedente si vede che la classe “Persona” ha due proprietà: il colore degli occhi che è un colore e il nome che è una stringa. L'oggetto “Persona” deriva dalla classe “Persona” (ne è una istanza) e i valori unici che assumono le sue proprietà ci permettono di riconoscerlo e di dargli un significato (ad esempio occhi “verdi” e nome “Sofia”).

## Metodi

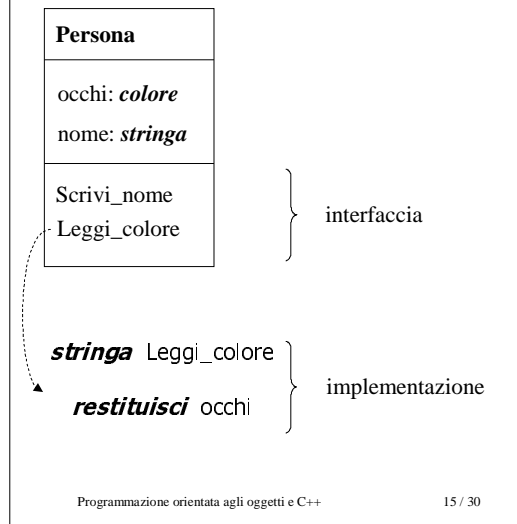
- Qualunque scambio di informazioni tra oggetti avviene tramite messaggi;
- la reazione ad un messaggio è gestita da un metodo;
- i metodi e le proprietà sono comuni a tutti gli oggetti della stessa classe;
- l'insieme dei metodi pubblici è detto interfaccia.

Gli oggetti comunicano tra di loro e con il mondo esterno tramite messaggi; un messaggio può essere visto come uno scambio di informazioni.

Un oggetto però reagisce solo ai messaggi che sono stati previsti e programmati nei cosiddetti metodi.

Evidentemente i metodi, come le proprietà, devono essere codificati all'interno di una classe, dato che sono comuni a tutti gli oggetti da essa istanziati; per questo motivo anche i metodi caratterizzano una classe, non entrano però a far parte dello stato.

## Esempio

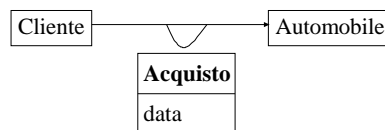


Per riprendere il caso di prima, la classe “Persona” può essere completata ad esempio con i metodi “Scrivi\_nome” e “Leggi\_colore” che servono per memorizzare il nome nell’oggetto e per conoscere il colore degli occhi. Questi due metodi definiscono l’interfaccia della classe, cioè dicono come si può comunicare con essa; inoltre hanno l’importante ruolo di mantenere integro lo stato interno degli oggetti.

Ciascun metodo avrà poi una sua implementazione interna alla classe e mascherata; si nota che un metodo potrà avere varie implementazioni, l’importante è che sia rispettata l’interfaccia verso l’esterno (nel caso in esame che venga restituito il colore degli occhi).

## Associazioni

- Quando due oggetti comunicano tra loro si instaura una relazione;
- una associazione può avere delle proprietà.



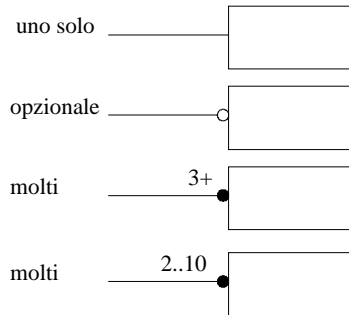
Quando gli oggetti comunicano tra loro vengono stabilite delle relazioni ed il fatto che queste comunicazioni sono fissate una volta per tutte dai metodi rende tali relazioni uniche.

Se ad esempio abbiamo una classe “Cliente” ed una “Automobile” possiamo costruire un legame tra le due dicendo che ogni cliente a cui siamo interessati ha acquistato una automobile. Una relazione può anche avere delle proprietà, ad esempio la data d’acquisto. Potrebbe anche darsi che la relazione “Acquisto” diventi tanto importante per noi da assumere una sua autonomia: in tal caso dovremmo allora modellarla come una classe ulteriore.



## Molteplicità

Una relazione può collegare  
anche più oggetti:



Vi sono vari tipi di relazioni: univoche (uno ed un solo legame), opzionali (anche nessuno) oppure multiple. Si può anche specificare l'intervallo di legami ammessi, ma è solo una notazione.

## Panoramica

- Un sistema viene modellato tramite un insieme di oggetti;
- gli oggetti comunicano tramite messaggi;
- gli oggetti sono definiti dalle classi (proprietà e operazioni);
- le classi sono collegate mediante relazioni.

Riassumendo, un sistema complesso è modellato meglio come un insieme di oggetti. Questi oggetti comunicano tra loro tramite messaggi. Gli oggetti sono istanze di classi, e queste ne stabiliscono le caratteristiche ed il comportamento. Infine le classi sono in relazione tra loro e le relazioni sono stabilite dalle interfacce, cioè dal modo in cui le classi possono comunicare tra loro.

## Astrazioni

Per aumentare la flessibilità degli oggetti si ricorre alle astrazioni:

1. Modularità (incapsulamento).
2. Generalizzazione (ereditarietà).
3. Parametrizzazione (polimorfismo):
  - a. sovraccaricamento;
  - b. modelli.

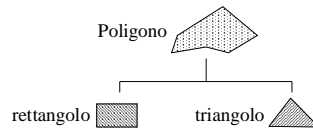
Ricordiamo che l'obiettivo della OOP è quello di favorire il riuso del codice. Ci sono tre meccanismi ideati per questo scopo; abbiamo già parlato del primo ovvero della modularità che nella OOP è conosciuta come incapsulamento. Gli altri due sono l'ereditarietà e il polimorfismo che si basano rispettivamente sul concetto di generalizzazione e di parametrizzazione. Il polimorfismo ha tre “sfaccettature”, noi vedremo solo il sovraccaricamento e i modelli.

## Astrazioni

- Incapsulamento



- Ereditarietà



- Sovraccaricamento

–  $1/2 + 1/3 = 5/6$ , “a” + ”b” = “ab”

- Modelli

– Ordina(“a”, “b”), Ordina(1,2)

Polimorfismo

Programmazione orientata agli oggetti e C++

20 / 30

L’incapsulamento abbiamo detto che si basa sul principio di modularità, gli oggetti sono cioè visti come delle scatole nere con le quali si può comunicare solamente attraverso la loro interfaccia.

L’ereditarietà permette di derivare classi più specializzate sfruttando l’esistenza di classi generali; ad esempio possiamo derivare le classi “Triangolo” e “Rettangolo” dalla classe “Poligono” ed evitare di dover riscrivere le proprietà in comune, come ad esempio i lati, l’area o il perimetro. Ci basterà fissare il numero di lati e scrivere nuove implementazioni per i metodi “Calcola\_area” o “Calcola\_perimetro” (magari più veloci). Volendo si possono poi aggiungere nuove proprietà come ad esempio “quadrato” per “Rettangolo” e “isoscele” per “Triangolo”.

Del polimorfismo vediamo i primi due aspetti:

- il sovraccaricamento consiste nel definire operatori che mantengono la sintassi ma cambiano di significato al variare degli operandi o del tipo di ritorno, ad esempio l’operatore di somma potrebbe comportarsi normalmente con i tipi predefiniti, sommare le frazioni se applicato ad una classe “Frazione” e concatenare i caratteri se applicato ad una classe “Stringa”.
- i modelli permettono di definire un unico codice di implementazione che funziona per più tipi di dato sfruttando il sovraccaricamento. In questo caso l’esempio non è molto chiaro perché la funzione “Ordina” potrebbe essere semplicemente sovraccaricata, in realtà il codice sottostante è unico.

## Incapsulamento

- Ogni componente deve rivelare solo ciò che è utile per gli altri;
- l'interfaccia deve rivelare il meno possibile sulla struttura interna;
- i compartimenti stagni facilitano il riuso del codice.

L'incapsulamento è il principio della scatola nera: non si sa cosa ci sia all'interno e si può comunicare solo attraverso l'interfaccia, la quale rivela l'essenziale. In questo modo si facilita il riuso del codice, perché si danno tutte e sole le istruzioni che servono per mettere insieme i vari pezzi.

## Ereditarietà

- Le sottoclassi permettono di specializzare il codice di una classe e di riusarlo molte volte;
- si possono definire classi “astratte” che definiscono un comportamento generale;
- le classi astratte assicurano uniformità nella gerarchia.

Anche l’ereditarietà permette di sfruttare il codice esistente e di specializzarlo secondo le necessità.

In più c’è la possibilità di definire classi astratte che definiscono un comportamento generale, ad esempio una classe che modella un *modem* deve prevedere nella sua interfaccia almeno un metodo “Ricevi” ed uno “Trasmetti” per potersi interfacciare con il resto del sistema.

Le classi astratte poi introducono una certa uniformità, perché ad esempio permettono di imporre che tutte le classi derivate da “Poligono” abbiano un metodo per calcolare l’area.

In Java e C# esiste un concetto analogo noto con il nome di “interfaccia”.

## Polimorfismo

- Il sovraccaricamento delle funzioni permette di ottenere un codice più pulito e semplice;
- le classi modello forniscono uno strumento per scrivere codice generalizzato.

I vantaggi del polimorfismo sono la possibilità di scrivere meno codice e quindi di renderlo più pulito, inoltre il codice può essere generalizzato e quindi riusato ancora in più occasioni.

## C / C++

- Miglioramento del C:
  - controllo sui tipi;
  - dichiarazioni più flessibili;
  - valori predefiniti;
  - riferimenti.
- Supporto alla OOP:
  - mascheramento dell'informazione;
  - funzioni e classi modello;
  - sovrapposizione;
  - ereditarietà;
  - legame dinamico.

Il C++ estende il C in due maniere: lo migliora e lo predispone alla OOP.

Tra i miglioramenti rispetto al C citiamo: un maggiore controllo sui tipi di dato usati, la dichiarazione di variabili ovunque nel codice, i valori predefiniti per gli argomenti di funzione, i riferimenti.

C'è poi il supporto agli oggetti e alle astrazioni. Il legame dinamico, di cui non abbiamo parlato, si può vedere come un sovraccaricamento delle funzioni esteso alla catena di generalizzazioni realizzate tramite l'ereditarietà.



## Miglioramento del C

```
int main()
{
    ...
    if (...) {
        int a = 5, b = 10;
        Scambia(a, b);
    }
}

void Scambia(int &a, int &b, int c = 1)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

In C: &a e &b

In C: \*a e \*b

Programmazione orientata agli oggetti e C++

25 / 30

Questo frammento di codice concentra tre delle migliorie del C++.

Innanzitutto vediamo che nel “main” le variabili “a” e “b” sono dichiarate nel bel mezzo del programma, cosa che in C è proibita; in C++ questo permette di rendere visibili “a” e “b” solo all’interno del blocco in cui sono dichiarate, cioè dentro l’”if”.

La seconda novità è che la funzione “Scambia” viene richiamata con soli due argomenti all’interno del “main”, ma in realtà questa si aspetta tre argomenti; in effetti nella definizione è scritto “c=1” a significare che possiamo tralasciare “c” e che se viene omissso il suo valore è pari a “1”.

La terza particolarità riguarda gli argomenti “a” e “b” della funzione “Scambio”. Tale funzione, come si osserva, non restituisce niente, perché non fa altro che scambiare i valori dei suoi due primi argomenti. Se volessimo realizzarla in C dovremmo passare dei puntatori come riportato nei riquadri. In C++ esiste anche il passaggio per riferimento, che permette di snellire la scrittura (non si devono più dereferenziare i puntatori).

## Supporto alla OOP

```
class Pila {  
  private:                                ← incapsulamento  
    int elementi[50];  
  public:  
    Impila(int elem);  
    int Spila();  
};
```

} interfaccia

In C:  
**struct** Pila {  
 **int** elementi[50];  
};

La classe Pila ha due sezioni: la sezione privata nasconde le proprietà interne, mentre quella pubblica espone l'interfaccia. Per scrivere in C un codice che si avvicini a questo dovremmo usare una struttura, ma non avremmo il controllo sulla visibilità né la possibilità di salvaguardare i dati.

## Supporto alla OOP

```
class Pila {  
    ...  
    Impila(int elem);  
    int Spila();  
};  
  
class PilaOrdinata: Pila {  
    ...  
    Impila(int elem);  
    Impila(float elem);  
    ...  
};
```

ereditarietà

sovrapposizione

PilaOrdinata deriva da Pila e ne eredita le proprietà e i metodi. In questo codice essa definisce anche una funzione sovrapposta per gestire i valori in virgola mobile.

## Supporto alla OOP

```
template < class T >
class Pila {
private:
    T elementi[50];
public:
    virtual Impila(T &elem);
    virtual T &Spila();
};

template < class T >
class PilaOrdinata: Pila< T > {
...
}
```

parametrizzazione

legame dinamico

ereditarietà con parametro

Infine questo è un esempio di classe modello in cui viene parametrizzato il tipo di dati su cui si lavora, cioè T potrà essere di volta in volta un intero, un numero in virgola mobile o addirittura una classe a sua volta. Una classe modello può anche far parte di una gerarchia.

## Standard Template Library

- Nello *standard* del C++ è presente anche la STL.
- Sfrutta tutte le caratteristiche del C++ e funziona sui sistemi compatibili.
- Comprende ad esempio:
  - Strutture dati (stringhe, liste, ...);
  - Algoritmi (ricerca, ordinam., ...);
  - Eccezioni.

Tutti i concetti precedenti sono usati nella STL, una libreria di funzionalità comunemente usate in programmazione.

## Java / C++

- **Lentezza:**
  - 5÷10 volte più lento del C++.
- **Semplicità / Robustezza:**
  - senza puntatori; prevede istruzioni per programmi paralleli.
- **Portabilità / Sicurezza:**
  - compila in *byte-code* interpretato e verificato a *run-time*.
- **Dinamicità / Internet-aware:**
  - risorse locali e distribuite.

## Confronto con il Java

Esempio di  
programmazione  
orientata agli oggetti  
in C++

A.A. 2003/2004

Programmazione orientata agli oggetti e C++

31 / 30

Inserisco le diapositive della lezione tenuta l'anno scorso nel caso qualcuno fosse interessato ad una applicazione funzionante del C++.

## Specifiche per l'esempio

1. Moltiplicazione di matrici rettangolari (massimo 10x10);
2. si vuole che siano introdotte da tastiera;
3. in uscita stampare il risultato.



## Una classe per le matrici

### **Matrice**

r: **intero**

c: **intero**

elem: **vettore di interi**

Leggi ( )

**Matrice** Moltiplica (**Matrice**,**Matrice** )

Scrivi ( )

interfaccia ←

(Questo schema è in UML)

## Organizzazione dei codici sorgente

matrice.h

dichiarazioni  
della classe

matrice.cpp

definizioni  
della classe

prova\_matr.cpp

programma  
principale

## Il *file* di intestazione matrice.h

```
#include <iostream>
```

```
class Matrice {  
    private:  
        int elem[10][10];  
        int r;  
        int c;  
    public:  
        void Leggi();  
        Matrice operator*(const  
                           Matrice& m);  
        void Scrivi();  
};
```

La classe in UML corrisponde al seguente codice in C++.

## Uso di una classe

Dichiarazione di un oggetto:

**Matrice m;**

Accesso ad una proprietà:

**m.r**

Invocazione di un metodo:

**m.Leggi()**

## Il programma principale prova\_matr.cpp

```
#include "matrice.h"

int main(int argc, char **argv)
{
    Matrice a, b, c;

    a.Leggi();
    b.Leggi();
    c = a * b;
    c.Scrivi();
}
```

Questo è tutto il codice che serve per moltiplicare le matrici una volta definita la classe Matrice.

## Lo spazio dei nomi

- In C non è possibile dichiarare più variabili con lo stesso nome;
- in C++ esistono i *namespace*:

```
namespace Tavolo {  
    int n_gambe;  
}  
namespace Persona {  
    int n_gambe;  
}
```

Tavolo::n\_gambe  $\neq$  Persona::n\_gambe

## La gestione dell'I/O

- Il C++ sfrutta una classe specifica “`iostream`”;
- “`iostream`” fa parte della Standard Template Library (*namesp. std*).
- L’*output* è gestito tramite un oggetto chiamato “`cout`” (“`cin`”);
- i vari “pezzi” di un testo si concatenano tramite “<<” (“>>”).

## Il metodo “Scrivi” in matrice.cpp

```
#include "matrice.h"
#include <iostream>

using namespace std;

void Matrice::Scrivi()
{
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            cout<<elem[i][j]<<" ";
        }
        cout<<"\n";
    }
}
```

Programmazione orientata agli oggetti e C++

40 / 30

Con l’istruzione “using namespace std” indichiamo al compilatore che vogliamo tener conto dello spazio dei nomi “std”, omettendolo dovremmo preporre “std::” a tutti i “cin” e “cout” (sono oggetti).



## Il metodo “Leggi” in matrice.cpp

```
void Matrice::Leggi()
{
    int i = 0, j = 0;

    cout<<"Inserire il numero di righe  
di M1 (massimo 10): ";
    cin>>r;
    cout<<"Inserire il numero di colonne  
di M2 (massimo 10): ";
    cin>>c;
    ...
}
```

## Il metodo “Leggi” continuazione

```
...
do {
    cout<<"Inserire l'elemento ("<<(i
    + 1)<<" "<<(j + 1)<<" ): ";
    cin>>elem[i][j];
    j++;
    if (j >= c) {
        i++;
        j = 0;
    }
} while (i < r);
}
```

## La moltiplicazione di matrici (A \* B)

- Si usa il prodotto “riga per colonna”;
- le dimensioni devono essere compatibili:  
Colonne di A = Righe di B

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 15 \\ 19 & 26 & 33 \end{bmatrix}$$

## Il metodo “Moltiplica” in matrice.cpp

```
Matrice Matrice::operator*(const  
    Matrice& m)  
{  
    Matrice risultato;  
  
    if (c != m.r) {  
        cout<<"Le matrici non sono  
            compatibili!";  
        exit(1);  
    }  
    risultato.r = r;  
    risultato.c = m.c;
```

## Il metodo “Moltiplica” continuazione

```
...  
for (int i = 0; i < r; i++) {  
    for (int j = 0; j < m.c; j++) {  
        risultato.elem[i][j] = 0;  
        for (int k = 0; k < c; k++) {  
            risultato.elem[i][j] +=  
                _elem[i][k] * m.elem[k][j];  
        }  
    }  
}  
return risultato;  
}
```

## Compilazione con g++

Per compilare l'esempio  
scrivere il codice nei *file* indicati e  
poi digitare:

```
g++ -o matr.exe matr.cpp  
prova_matr.cpp
```