Elementi di OOP in C++

(Object Oriented Programming)

prof. *Claudio Maccherani*

2016

Introduzione

La programmazione **OOP** (*Object oriented Programming*) "orientata agli oggetti", nata negli anni settanta e sviluppatasi negli anni ottanta, pone l'attenzione sui dati da manipolare, piuttosto che sulle procedure che li manipolano (come avviene per la programmazione "imperativa" tradizionale) e impone che siano questi ultimi alla base del Modello Orientato ai Dati, un sistema costituito da un insieme di entità e oggetti che interagiscono tra loro. Il dato viene visto come un tipo di dato astratto (**ADT**) caratterizzato da un ***insieme di valori*** che lo caratterizzano e da un ***insieme di operazioni*** che possono essere applicate a esso. La OOP garantisce modularità e "riusabilità" del software, facile gestione e manutenzione di progetti di grandi dimensioni, riduce la dipendenza del programma dalla rappresentazione dei dati ai quali accede mediante un'interfaccia.

Classi e Oggetti

L'elemento principale della OOP è la **CLASSE**, una descrizione astratta di un tipo di dato (ADT) che descrive una famiglia di oggetti con caratteristiche e comportamenti simili. Un **OGGETTO** è una **ISTANZA** della classe, cioè la rappresentazione concreta di una classe. Quando si instanzia una variabile definendola di una classe si crea un oggetto di quella classe rappresentato dal nome della variabile istanziata. La differenza tra classe e oggetto è la stessa che c'è tra tipo di dato e dato.
Ad esempio gli oggetti Airbus A330, Boeing 474 e Antonov 124 appartengono alla classe Aerei.

Attributi e Metodi

Una classe è costituita da **Attributi** (campi che specificano le caratteristiche o proprietà che tutti gli oggetti della classe debbono avere, i cui valori in un certo istante determinano lo *stato* del singolo oggetto della classe) e da **Metodi** (funzioni che specificano le azioni o i comportamenti ammissibili che un oggetto della classe è in grado di compiere; tali operazioni possono comunicare all'esterno lo stato dell'oggetto o modificarlo). I metodi di una classe sono l'**Interfaccia** della classe, l'unico strumento tramite il quale è possibile interagire con gli oggetti della classe.
Durante l'elaborazione un oggetto viene creato, utilizzato e infine distrutto.

	classe
	Aereo
attributi	peso velocità passeggeri carburante fase di volo
metodi	Decolla() Accelera() Rallenta() Prendi_quota() Perdi_quota() Stabilizzati() Atterra() Rifornisci()

oggetto
AirbusA330
peso = 230 t velocità = 910 km/h passeggeri = 330 carburante = 120000 lt fase di volo = crociera

oggetto
Boeing474
peso = 300 t velocità = 950 km/h passeggeri = 500 carburante = 180000 lt fase di volo = decollo

Gli oggetti di una classe vengono creati da uno specifico metodo **Costruttore** che deve avere lo stesso nome della classe che quando viene eseguito (in C++ con la definizione "*classe oggetto;*", in Visual Basic e altri linguaggi con l'operatore **New** "*oggetto* = New *classe*") alloca la memoria necessaria a contenere l'oggetto e ne inizializza gli attributi.

Per poter stabilire quali attributi e quali metodi occorre prevedere per una classe occorre conoscere il **contesto** nel quale la classe sarà utilizzata. Nella classe Aereo dell'esempio attributi e metodi saranno diversi se il contesto è di volo oppure di manutenzione oppure di costruzione, etc.

UML

Per rappresentare graficamente classi e oggetti la **UML** (*Unified Modeling Language*, linguaggio di modellizzazione unificato) mette a disposizione il **diagramma delle classi** (*class diagram*), simile al sintetico formalismo utilizzato per l'esempio della classe Aereo precedente.
La classe si rappresenta con un rettangolo diviso in tre sezioni: nella prima c'è il nome della classe, nella seconda l'elenco degli attributi ciascuno con il relativo tipo di dato, nella terza c'è l'elenco dei metodi ciascuno specificato con il nome, la lista dei parametri e il tipo.
Per ogni attributo e per ogni metodo occorre anche specificare se è *Pubblico* (accessibile e visibile dall'esterno) o *Privato* (non accessibile e invisibile dall'esterno).
L'oggetto si rappresenta con un rettangolo con gli angoli arrotondati diviso in due sezioni: la prima contiene il nome dell'oggetto e il riferimento alla classe di appartenenza, la seconda l'elenco degli attributi e dei rispettivi valori.

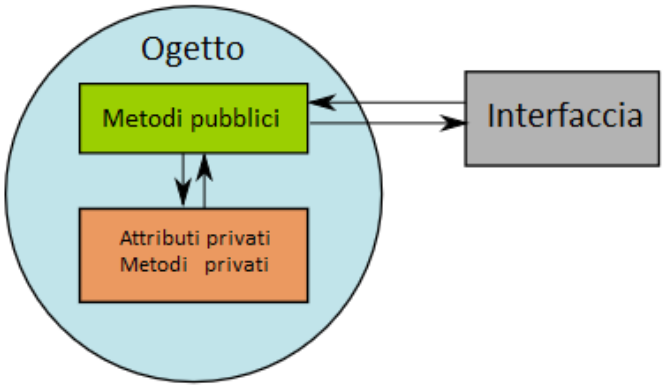
Tipo del metodo:	*costruttore*	crea un oggetto e inizializza il suo stato
	modificatore	cambia lo stato di un oggetto
	query	comunica lo stato di uno o più attributi di un oggetto

Interfaccia

Si accede agli oggetti tramite un'**interfaccia** che consente di utilizzare l'oggetto e interagire con esso senza doverne conoscere la struttura interna e l'implementazione. Tale interfaccia è costituita dagli attributi e dai metodi definiti **pubblici** della classe di appartenenza dell'oggetto.

La comunicazione e l'interazione tra oggetti avviene attraverso lo scambio di messaggi che richiedono di conoscere o modificare il valore di un attributo o di eseguire un dato metodo.
Gli attributi dell'oggetto vengono identificati dalla cosiddetta "notazione punto" ***oggetto.attributo***.
Ad esempio *AirbusA330.passeggeri* identifica l'attributo "passeggeri" dell'oggetto "AirbusA300".

Gli attributi si possono modificare direttamente - esempio: AirbusA300.velocità = 800 -, ma è meglio prevedere appositi metodi attraverso i quali accedervi per conoscerne il valore, **get()**, o per modificarne il valore, **set()**. Questo permette di controllare la correttezza dei dati, impedendo, per esempio, di impostare la velocità ad un valore superiore a quella massima consentita.



La OOP è alla base della programmazione dei **videogiochi**. Ad esempio tutti i soldati debbono rispondere secondo le loro caratteristiche al segnale "attacca": l'arciere scaglia la freccia, il fante colpisce di spada, il cavaliere lancia il giavellotto. Se sono "oggetti" ciascuno avrà il suo metodo "attacca" immediatamente attivabile con una semplice istruzione del tipo `soldato.Attacca()` che lo farà *attaccare* in base alla categoria di appartenenza.

Incapsulamento e Information Hiding

L'**incapsulamento**, che è la proprietà degli oggetti di mantenere al loro interno attributi e metodi (variabili e funzioni), è uno dei vantaggi della OOP. L'oggetto è all'interno di una "capsula" che lo isola dal mondo esterno e lo protegge.

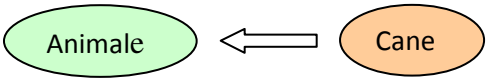
L'**information hiding** (*mascheramento dell'informazione*), legato all'incapsulamento, è la proprietà dell'oggetto di nascondere all'esterno i dettagli implementativi dei propri metodi.

Ai programmi che usano l'oggetto non serve sapere come esso è implementato, ma conoscere solo la sua interfaccia, sapere come utilizzare i suoi metodi, non come essi sono realizzati.

vedi esempio **1** (Pila)

Attributi e metodi possono essere dichiarati **private** (default – accessibili solo all'interno della classe dove compaiono), **public** (accessibili da tutti) e **protected** (accessibili dalla classe ove compaiono e dalle classe derivate – vedi sotto "ereditarietà" e "polimorfismo")

Ereditarietà



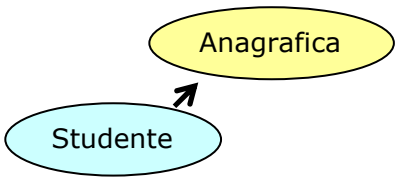
vedi esempio **2** (Animali)

L'**ereditarietà** è uno strumento molto potente nella programmazione a oggetti che consente di definire una nuova classe che mantiene le proprietà (attributi e metodi) di una classe esistente (detta "classe base" o "superclasse") ma aggiunge alla nuova classe (detta "classe derivata" o "sottoclasse") nuovi attributi e metodi.

Con l'ereditarietà viene introdotto il concetto di "gerarchia di classi": una data classe potrà avere classi che la precedono e classi che la seguono nella gerarchia.
La gerarchia si stabilisce con la dichiarazione "`classe derivata : classe base`".
Esistono due tipi di ereditarietà: quella ***singola*** (dove una classe deriva da un'unica superclasse, come nell'esempio 2 più avanti riportato) e quella ***multipla*** (dove una classe deriva da due o più superclassi, ereditarietà questa che non tutti i linguaggi di programmazione OOP supportano).
In C++ è prevista la **funzione virtuale** che può essere ridefinita nelle sottoclassi solo con lo stesso prototipo della sua definizione, così da impedire alle sottoclassi di ridefinire tale funzione con diverso numero o tipo di parametri, costringendo l'implementatore della sottoclasse a usare la stessa interfaccia (prototipo) della funzione definita la prima volta come "`virtual`".

Polimorfismo

Il **polimorfismo** è la capacità degli oggetti di assumere comportamenti diversi, "più forme", di risposta in base al contesto (cioè è la capacità degli oggetti di adattarsi al contesto in cui si trovano). Questo meccanismo fa si che un singolo nome di un metodo o di un attributo possa essere definito su più di una classe e assumere diverse implementazioni in ciascuna di quelle classi.



vedi esempio **3** (Studenti)

Esistono due tipi di polimorfismo:
1) per **overloading** (*sovraccarico* dei metodi, metodi con lo stesso nome e con numero diverso di parametri e/o tipo diverso dei parametri – *firma* differente - e solitamente con comportamenti diversi). Un esempio si questo tipo di polimorfismo sono i *metodi costruttori*, che possono essere più di uno per la stessa classe, necessariamente con lo stesso nome, ma con diversa lista di parametri.
2) per **overriding** (*sovrapposizione* dei metodi, nella classe derivata il metodo ereditato viene sovrascritto cambiandone le istruzioni, e quindi il comportamento, ma mantenendo invariato il numero e il tipo dei parametri – stessa *firma*).
In C++ il polimorfismo è generalmente implementato con le funzioni virtuali tramite overriding.

Esempi

1) **Incapsulamento** - programma C++ che implementa l'ADT PILA con le operazioni PUSH e POP.

```
//"PilaOOP.cpp" - incapsulamento - dato astratto Pila - (Claudio Maccherani)

#include<iostream>
using namespace std;

// definizione classe del dato astratto pila
class Pila
   { static const int N=11
             int p[N]; ;     // attributi privati, non visibili all'esterno
             int top;
     public: Pila();         // metodi pubblici, accessibili dall'esterno
             void Visualizza();
             int Pop();
             void Push(int elemento);  };

// definizione metodi della classe pila
Pila::Pila()                  // costruttore
   { for(int i=0;i < N;i++) p[i]=0;
     top=0; cout << "Pila costruita ("<<N-1<<" elementi)\n\n"; }
void Pila::Visualizza()        // mostra il contenuto della pila, query
   {  if (top == 0) { cout << "\nPila VUOTA !\n"; }
      else { cout << "\nContenuto dell pila: (Top="<<top<<")\n\n";
             for(int i=top;i>0;i--)
                { cout << "|\t"<< p[i] << "\t|";
                  if (i!=top) cout<<"\n";
                  else        cout<<"  <-- Top\n"; }
             cout<<"+---------------+\n";  } }
void Pila::Push(int elemento) // PUSH - inserimento in pila, set()
   { if (top == N-1) { cout << "\nPila PIENA, elemento NON inserito !\n"; }
     else            { top++; p[top] = elemento; }  }
int Pila::Pop()               // POP - estrazione dalla pila, get()
   { int dato;
     if (top == 0) { dato=-1; cout << "\nPila VUOTA, nessun elemento!\n"; }
     else          { dato = p[top]; p[top] = 0; top--; }
     return dato; }

int main(){ // programma ------------------------------------------------
   int scelta,dato;
   Pila P; // dichiarazione istanza di pila (e costruzione oggetto P)
   do { cout << "\n\n    Gestione PILA (oop)\n\n";
        cout << "1 - inserimento elemento  in    pila (PUSH)\n";
        cout << "2 - estrazione  elemento  dalla pila (POP) \n";
        cout << "3 - visualizzazione stato della pila       \n";
        cout << "    scegli (0=fine) "; cin>>scelta;
        if (scelta==1) { cout<<"\ndato: "; cin>>dato; P.Push(dato); }
        if (scelta==2) { dato = P.Pop();
                         if (dato!=-1) { cout<<"\ndato:"<<dato<<"\n";} }
        if (scelta==3) { P.Visualizza(); }
      }
   while(scelta!=0);
   return 0; }
```

2) **Ereditarietà** - programma C++ con la classe base Animale e la classe derivata Cane.

```
//"AnimaleOOP.cpp" - ereditarietà - (HTML.it, Claudio Maccherani)

#include <iostream>
#include <string>
using namespace std;

class Animale {               // "Animale", classe base o superclasse
    public:    Animale();
    protected: string specie; // attributi protected, visibili anche
               int  eta;      // alle classi derivate (Cane)
               char sesso;
               void Mangia();
               void Beve();
    public:    void Inserimento_dati();  };
Animale::Animale() { specie = " ";  cout << "\nCostruito Animale\n"; }
void Animale::Mangia() { cout << "MANGIA\n"; }
void Animale::Beve()   { cout << "BEVE\n"; }
void Animale::Inserimento_dati() {
    cout << "\nEta'   "; cin >> eta;
    cout << "Sesso  ";   cin >> sesso; }

class Cane : public Animale { // "Cane", classe_derivata o sottoclasse
    public:  Cane();
             string razza;
             void Esegui_azioni();
             void Stampa_dati();
    private: void Abbaia(); };
Cane::Cane() { cout << "\nCostruito Cane\n"; specie = "cane"; razza = " ";}
void Cane::Abbaia() { cout << "ABBAIA\n"; }
void Cane::Stampa_dati() {
    cout << "\nSpecie " << specie << endl;
    cout << "Eta'   " << eta << endl;
    cout << "Sesso  " << sesso << endl;
    cout << "Razza  " << razza << endl << endl; }
void cane::Esegui_azioni() {
    Mangia();
    Beve();
    Abbaia(); }

int main() { // programma ------------------------------------------------
    cout << "Animale e Cane: esempio di Ereditarieta'\n";
    Cane c;
    c.Inserimento_dati();
    cout << "Razza  "; cin >> c.razza;
    c.Stampa_dati();
    c.Esegui_azioni();
    cout<<endl; system("pause");
    return (0); }
```

3) **Polimorfismo** - programma C++ con la classe base Anagrafica e la classe derivata Studente.

```
// "StudenteOOP.cpp" - polimorfismo - (Lorenzi/Govoni, Claudio Maccherani)

#include <string>
#include <iostream>
using namespace std;

class Anagrafica {  // "Anagrafica", superclasse/classe base
   public:    int    codice;
              string cognome;
              string nome;
              void   Registra();
              void   Mostra();
   protected: bool   registrato; }; // protected, visibile a classi derivate
void Anagrafica::Registra() { registrato = true; }
void Anagrafica::Mostra() {
     if (registrato) { cout<<codice<<endl; cout<<cognome<<endl; cout<<nome<<
endl; }
     else { cout << "Non registrato" << endl; } }

class Studente:public Anagrafica { //"Studente", sottoclasse/classe derivata
           int  anno;
           char sezione;
           bool promosso;
   public: void Registra(int,char);
           void Promuovi();
           void Mostra();
           void Controlla();
           Studente() { promosso=false; anno=0; sezione=' '; } //costruttore
   };
void Studente::Promuovi() { promosso = true; }
void Studente::Controlla()
      { if (promosso) cout<<"PROMOSSO\n"; else cout<<"NON promosso\n"; }
// OVERLOADING del metodo "Registra" della classe base Anagrafica:
void Studente::Registra(int a,char s)
      { anno = a; sezione = s; registrato = true;}
// OVERRIDING del metodo "Mostra" della classe base Anagrafica:
void Studente::Mostra()
      { Anagrafica::Mostra(); cout << anno <<endl; cout << sezione <<endl; }

int main() { // programma ------------------------------------------------
   Studente stud1; char risp,sez; int cla;
   cout<<"\nRegistrazione STUDENTE\n\n";
   cout<<"Codice:   "; cin >> stud1.codice;
   cout<<"Cognome:  "; cin >> stud1.cognome;
   cout<<"Nome:     "; cin >> stud1.nome;
   cout<<"Anno:     "; cin >> cla;
   cout<<"Sezione:  "; cin >> sez;
   stud1.Registra(cla,sez); stud1.Controlla();
   cout<<"Promosso: "; cin >> risp; if (risp=='s') { stud1.Promuovi(); }
   stud1.Mostra(); stud1.Controlla();
   system("pause");
   return 0; }
```