Corso sul linguaggio C++

Modulo 5

3 – Proprietà della OOP

Prerequisiti

- Programmazione elementare ad oggetti
- Proprietà della OOP

Introduzione

Lo scopo di questa Unità è quello di mostrare in pratica l'applicazione delle proprietà di ereditarietà e di polimorfismo.

Si vedrà come sfruttare la possibilità di riusare software già collaudato e funzionante, allo scopo di diminuire i tempi di sviluppo.

Nel riuso del software si possono espandere le funzionalità di quello esistente, o mascherarne gli effetti.

Ereditarietà

Supponiamo di voler scrivere una classe *Rettangolo*, con attributi *base* e *altezza* e metodi *area()* e *perimetro()* e di voler ricavare per ereditarietà da essa la classe *TriangoloRettangolo*.

Implementiamo la classe *Rettangolo* e facciamo alcune osservazioni.

Ereditarietà

```
class Rettangolo
{  protected:
     int base, altezza;
   public:
     Rettangolo() { base=0; altezza=0; }
     Rettangolo (int b, int a): base(b), altezza(a) {}
     ~Rettangolo() {…}
     int getBase() {return base;}
     int getAltezza() {return altezza;}
     void setBase (int b) {base=b;}
     void setAltezza (int a)  {altezza=a;}
     int area() {return base*altezza; }
     int perimetro() { return 2*(base+altezza); }
};
```

Lo specificatore **protected** consente di rendere visibili gli attributi alle sole classi derivate, ma non all'esterno

Ereditarietà

La classe *TriangRettangolo* è derivata dalla classe base *Rettangolo* e ne eredita tutti i membri pubblici, ma aggiunge *colore*

```
class TriangRettang: public Rettangolo
{  private:
     int colore;
     float ipotenusa() {return (float) sqrt (pow (base,2) +
                                        pow (altezza,2)); }
   public:
     TriangRettang () : Rettangolo()  {colore = 0;}
     TriangRettang (int b, int a, int c) : Rettangolo (b, a) {colore=c;}
     ~TriangRettang() {…}
     float perimetro() {return base+altezza+ipotenusa();}
     float area() {return (base*altezza/2);}
};
```

Il costruttore TriangRettabg() eredita direttamente dal costruttore senza parametri di Rettangolo, ma deve inizializzare il nuovo attributo *colore*.

Ereditarietà

Il costruttore con parametri deve inizializzare tre attributi. I primi due li inizializza ereditando il costruttore con parametri della superclasse, il terzo attributo lo inizializza nel codice tra le graffe

```
class TriangRettang: public Rettangolo
{  private:
    int colore;
    float ipotenusa() {return (float) sqrt (pow (base,2) +
                                             pow (altezza,2)); }
   public:
    TriangRettang () : Rettangolo()  {colore = 0;}
    TriangRettang (int b, int a, int c) : Rettangolo (b, a) {colore = c;}
    ~TriangRettang() {…}
    float perimetro() {return base+altezza+ipotenusa();}
    float area() {return (base*altezza/2);}
};
```

I metodi *area*() e *perimetro*() sono ridefiniti rispetto alla classe base Rettangolo. Il metodo *ipotenusa*() specifico per questa classe, può essere considerato privato.

Ereditarietà

Viene creato un oggetto di classe *Rettangolo* di cui si stampa area e perimetro.

```
int main()
{   Rettangolo R1(3,5), R2;
    TriangRettang T(2,6);
    cout<<"Area di R1: "<<R1.area()<<endl;
    cout<<"Area di R2: "<<R2.area()<<endl;
    cout<<"Perimetro di R1: "<<R1.perimetro()<<endl;
    cout<<"Perimetro di R2: "<<R2.perimetro()<<endl;
    cout<<"Area di T: "<<T.area()<<endl;
    cout<<"Perimetro di T: "<<T.perimetro()<<endl;
    cout<<"Base:    "<<T.getBase()<<endl;
    cout<<"Altezza: "<<T.getAltezza()<<endl;
    system("Pause");
    return EXIT_SUCCESS;
}
```

Successivamente, viene creato un oggetto di classe *TriangRettangolo* di cui si stampano area e perimetro (mediante i metodi ridefiniti) e *base* e *altezza* mediante i metodi ereditati direttamente dalla classe *Rettangolo*.

Ereditarietà

class TriangRettang: **public** Rettangolo
{ …

};

Lo specificatore di accesso **public** consente l'**ereditarietà pubblica**. In questo caso, dalla classe *TriangRettangolo* (che eredita tutto ciò che nella classe base *Rettangolo* è dichiarato come **public** e **protected**) si possono eventualmente ereditare altre classi con lo stesso procedimento.

class TriangRettang: **private** Rettangolo
{ …

};

In questo esempio si avrebbe **ereditarietà privata**, per cui dalla classe *TriangRettang* non si possono derivare altre classi.

Ereditarietà

```
#include <iostream>
using namespace std;
class Numero
{
  protected:
     int num;
  public:
       void setNum (int n) {num=n;}
       int getNum() {return num;}
};
```

File Numero.h

L'ereditarietà può essere applicata anche ai <u>file esterni</u>, salvati come *header file* (con estensione **.h**.

Ereditarietà

```
#include <iostream>
#include "numero.h"
using namespace std;
class Coppia : public Numero
{   private:
      char car;
   public:
      void setChar (char c) {car=c;}
      char getChar() {return car;}
      void setNumChar (int n, char c)
      {   setNum(n);
         car=c;
      }
};
```

La classe *Coppia* specializza la classe *Numero* e quindi un oggetto di classe *Coppia* prevede un numero ed un carattere.

Coppia eredita tutti e soli i membri pubblici;

I metodi *setChar()* e *getChar()* di *Coppia* possono usare l'attributo *car*. Per accedere all'attributo **protected** *num* di *Numero* però, il metodo *setNumChar()* necessita del metodo *setNum()*

Ereditarietà

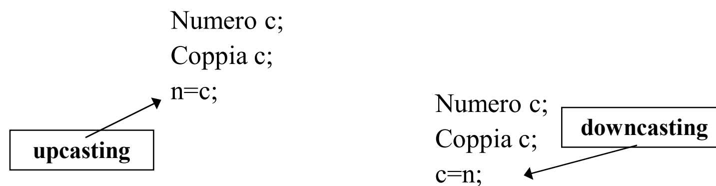
```
int main()
{   Numero n;
    Coppia c;
    n.setNum(3);
    c.setNumChar(5, 'B');
    cout<<"n.num = "<<n.getNum()<<endl;
    cout<<"c.num = "<<c.getNum()<<endl;
    cout<<"c.car = "<<c.getChar()<<endl;
    system("Pause");
    return EXIT_SUCCESS;
}
```

Nel **main**() si istanzia la classe *Numero* e si crea l'oggetto *n* contenente 3.

Si istanzia un oggetto *c* di classe *Coppia* contenente i valori 5 e 'B'. Il metodo *setNumChar*() della classe *Coppia* usa il metodo *setNum*() della classe *Numero*.

Upcasting e downcasting

Spesso capita che un oggetto di una classe debba essere convertito in un oggetto di una superclasse o viceversa. Considerate le due classi precedenti *Numero* e *Coppia* potremmo voler creare oggetti del tipo:



Upcasting e downcasting

L'**upcasting** (*promozione alla superclasse*) non crea problemi in quanto nella conversione

n=c;

al limite, si perde l'uso di alcuni attributi.

Il **downcasting** invece richiede l'operatore *cast* come indicato

(Numero) c=n;

In qualunque altro modo, si hanno errori in compilazione o in esecuzione.

Polimorfismo

Come sappiamo, un stesso metodo può assumere varie forme (*stesso nome, ma diversa firma*). Questa proprietà della **OOP** prende il nome di **polimorfismo**.

Il **polimorfismo** si presenta quando

- In una gerarchia di classi un metodo viene ridefinito all'interno di una classe, coprendo quello originario (*overriding* dei metodi)
- all'interno di una medesima classe, un metodo è presente più volte, con lo stesso nome, ma con interfaccia diversa, come numero e/o tipo dei parametri (*overloading* dei metodi)

Overloading

La classe *Rettangolo* presenta due costruttori con interfaccia diversa. In questo caso il **polimorfismo** si presenta come **overloading**.

```
class Rettangolo
{  protected:
    int base, altezza;
   public:
    Rettangolo() { base=0; altezza=0; }
    Rettangolo (int b, int a): base(b), altezza(a) {}
    ~Rettangolo() {…}
    int getBase() {return base;}
    int getAltezza() {return altezza;}
    void setBase (int b) {base=b;}
    void setAltezza (int a)  {altezza=a;}
    int area() {return base*altezza; }
    int perimetro() { return 2*(base+altezza); }
};
```

Overriding

Le due classi *Rettangolo* e *TriangRettang*, presentano ciascuna proprie versioni dei metodi *area()* e *perimetro()*.

In questo caso il **polimorfismo** si manifesta come **overriding**.

```
class Rettangolo
{  protected:
     int base, altezza;
   public:
     …
     int area() {return base*altezza; }
     int perimetro() { return 2*(base+altezza); }
};

class TriangRettang: public Rettangolo
{  …
   public:
     …
     float perimetro() {return base+altezza+ipotenusa();}
     float area() {return (base*altezza/2);}
};
```


Istanza di metodi in una gerarchia

Il polimorfismo, sia come *overloading* che come *overriding*, pone alcune questioni:

Se abbiamo due costruttori in overloading, come riconoscere quando viene istanziato l'uno o l'altro?

La scelta avverrà al momento della creazione dell'oggetto, in base all'interfaccia

Se abbiamo due metodi in overriding, come riconoscere quando viene istanziato uno o l'altro?

In base all'oggetto che lo istanzia.

Metodi friend

Quando si dichiara un metodo, normalmente si accetta che:

- possa accedere ai membri privati della classe;
- sia utilizzabile in modo pubblico;
- sia incapsulato negli oggetti (grazie al puntatore **this**).

Abbiamo visto che con i metodi statici è possibile fornire alla funzione le prime due proprietà, ma non la terza.

Vediamo ora un particolare tipo di metodi, detti **metodi friend**, nei quali invece si ha solo la prima proprietà.

Metodi friend

I **metodi friend**:

- sono dichiarati all'interno della classe con la sintassi
  friend *tipo ident* (*lista_par*)
- sono definiti esternamente alla classe stessa con la sintassi
  tipo ident **(***lista_par***)**{ *istruzioni* }
- accordano l'accesso ai dati membro della classe come se fossero pubblici
- non possiedono il puntatore **this**, quindi per operare necessitano dell'oggetto passato come parametro
- non sono considerati metodi membro della classe e quindi sono utili quando devono accedere a più classi, per cui non è conveniente farli appartenere a una classe piuttosto che a un'altra.

Metodi friend

class Classe1
{ **friend void** Funzione(Classe1 &Oggetto, *altri parametri eventuali*);
public: …
private: …
};

La **funzione friend** accede all'*Oggetto* mediante un puntatore by reference

class Classe2
{ **friend void** Funzione(Classe2 &Oggetto, *altri parametri eventuali*);
public: …
private: …
};

Ogni classe deve dichiarare la **funzione friend**

La dichiarazione può avvenire nella sezione private o public indifferentemente.

Metodi friend

Esternamente alle classi si definiscono i metodi *friend*:

void Funzione (Classe1 &Oggetto, *eventuali altri parametri*)
{*istruzioni sui dati membro* }

Le istruzioni possono accedere ad attributi pubblici, privati e protetti

void Funzione (Classe2 &Oggetto, *eventuali altri parametri*)
{*istruzioni sui dati membro* }

Le istruzioni possono accedere ad attributi pubblici, privati e protetti

Metodi friend

Il **main**() istanzia un oggetto *Istanza* di classe *Classe1*

```
int main()
{   Amica1 Istanza;
    Funzione(Istanza);
    system("Pause");
   return EXIT_SUCCESS;
}
```

Funzione() opera sull'oggetto *Istanza* e le eventuali modifiche agli attributi possono essere verificate stampandoli.

Classi friend

Analogamente ai metodi friend si possono dichiarare e definire le **classi friend.**

Una **classe friend**:

- accorda l'accesso ai suoi dati membro da parte di altre classi
- si dichiara con:

  friend class *nomeclasse*;

  dove *nomeclasse* è la classe a cui si consente l'accesso
- può essere acceduta dall'esterno mediante la sintassi

  tipo nomemetodo (*lista_par*) { *istruzioni* }

Classi friend

Quando tutti i metodi di una classe B sono **friend** di una classe A, anziché dichiarare ciascun metodo di B come *friend*, si può indicare che tutta la classe B è friend in A, nel seguente modo:

```
class A
{    ..........
     friend class B;.  // A accorda l’amicizia a B
     .........
};
```


Classi friend

La classe *Classe1* si rende disponibile per la *Classe2*.

```
class Classe1
{    friend class Classe2;
     private:    int dato1;
     public:     void leggi() {cout<<"dato 1="; cin>>dato1;}
                 void stampa() {cout<<"dato 1="<<dato1<<endl;}
};
class Classe2
{    private:    int dato2;
     public:     void leggi() {cout<<"dato 2="; cin>>dato2;}
                 void stampa() {cout<<"dato 2="<<dato2<<endl;}
                 void stampa (Classe1 c)
                    {cout<<"dato 1 tramite Classe 2="<<c.dato1<<endl;}
};
```

Il metodo *stampa (Classe1 c)* consente l’accesso di *Classe2* al dato membro *dato1* di *Classe1*, di cui effettua la stampa

Classi friend

```
int main()
{     Classe1 c1;
      Classe2 c2;
      c1.leggi();
      c1.stampa();
      c2.leggi();
      c2.stampa();
      c2.stampa(c1);
      system("Pause");
      return EXIT_SUCCESS;
}
```

Stampa dato privato di *Classe1*

Stampa del dato privato di *Classe2*

Il metodo *stampa* (c1) appartenente alla *Classe2*, stampa il dato membro *dato1* di *Classe1*

Classi friend

L'uso di metodi e classi friend permette al C++ di aggirare l'**information hiding** ogni volta che classi diverse necessitano di:

- interagire
- condividere dati

pur restando distinte.

Classi friend

Osserviamo le 3 proprietà fondamentali dell'amicizia:

- l'amicizia è accordata e non pretesa;
- l'amicizia non è simmetrica (se A è friend di B non è detto che B sia friend di A);
- l'amicizia non è transitiva (se A è friend di B e B è friend di C, non è detto che A sia friend di C).

Ogni relazione tra clessi, deve essere esplicitamente dichiarata.

Argomenti

- Ereditarietà
- Upcasting e downcasting
- Polimorfismo
- Overloading
- Overriding
- Istanza di metodi in una gerarchia
- Metodi friend
- Classi friend

Altre fonti di informazione

- P.Gallo, F.Salerno – Informatica Generale 1, ed. Minerva Italica
- M.Romagnoli, P.Ventura – Linguaggio C/C++, ed. Petrini
- M. Bigatti – Il linguaggio Java, ed. Hoepli