Corso sul linguaggio C++

Modulo 5

4 – Polimorfismo

Prerequisiti

- Programmazione elementare ad oggetti
- Proprietà della OOP

Introduzione

Il polimorfismo ed il binding dinamico si ritrovano nel linguaggio C++ nel concetto di **funzioni virtuali**.

Inoltre, vedremo come con il polimorfismo sia possibile ridefinire i simboli di operazioni, facendo in modo che un operatore (ad es. "=", "+") possa operare con dati di tipo diverso rispetto a quello originario.

Classi astratte

Per trarre dall'ereditarietà il massimo vantaggio, si potrebbe pensare di progettare una superclasse *che menzioni proprietà e comportamenti generali senza fornire nessuna implementazione specifica.*

Queste classi si dicono **classi astratte**.

Le classi derivate da una classe astratta, che forniscono l'implementazione di tutti i metodi mancanti, si dicono **classi concrete.**

Ad esempio, pensiamo alla classe *Alimento*: non esiste un alimento in generale, esistono solo alimenti specifici (acqua, cioccolata, carne); ma tutti gli alimenti possiedono *caratteristiche comuni* (provenienza, potere calorico, tenore in grassi, carboidrati, proteine, ecc).

Quindi la classe *Alimento* può essere considerata astratta perché raccoglie in sé proprietà simili a vari tipi di alimento;

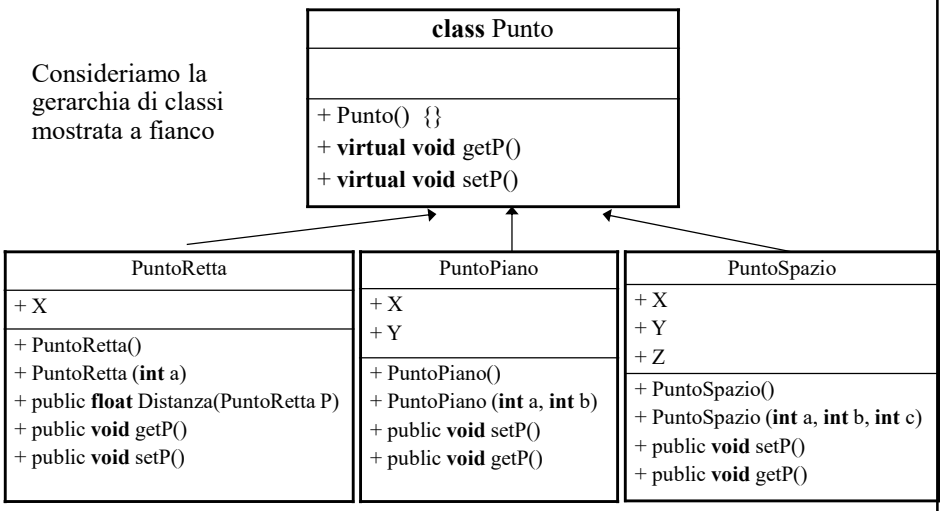
Classi astratte e metodi virtuali

E' ovvio che una classe astratta:

- *non può essere istanziata*: non ha senso parlare di un oggetto di classe *Alimento*, poiché va specificato di che alimento si tratti;
- *non può essere la classe terminale di una gerarchia*: la classe astratta, per definizione, serve a derivare classi concrete e non può quindi terminare una gerarchia di classi;
- *deve possedere almeno un* **metodo virtuale**, ossia un metodo di cui non si fornisce l'implementazione.

La classe astratta *Punto*

Consideriamo la gerarchia di classi mostrata a fianco



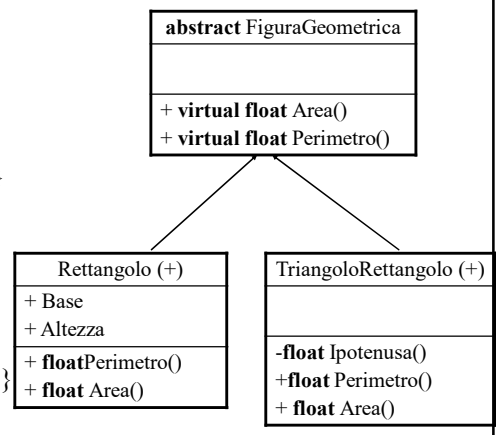
La classe astratta *Punto*

Osservazioni:

- come si nota, la classe *Punto* è astratta poiché contiene i **metodi virtuali** *setP*() e *getP*();
- i **metodi virtuali** vengono espansi in modo specifico nelle singole sottoclassi;
- ciascuna sottoclasse ridefinisce il proprio costruttore in forma senza parametri e con parametri;
- la classe *PuntoRetta* definisce il metodo *Distanza (PuntoRetta P)* per il calcolo della distanza tra due punti sulla retta.

Classe astratta *FiguraGeometrica*

```
class FiguraGeometrica
{ ….
  public:
  virtual float Area () {return 0;}
  virtual float Perimetro() {return 0;}
}
class Rettangolo : FiguraGeometrica
{  float Base, Altezza;
   public
      float Perimetro()
           {return (Base+Altezza)*2;  }
      float Area()
           {return (Base*Altezza);}
}
```



Metodi virtuali

La classe base *FiguraGeometrica* definisce i **metodi virtuali** *Area*() e *Perimetro*() come interfacce comuni a tutte le figure, ottenute come classi derivate.

Ciascuna delle classi derivate *Rettangolo* e *TriangoloRettangolo* ridefinisce personalizzandole, le funzioni *Area*() e *Perimetro*().

Durante l'uso degli oggetti derivati, viene invocata la funzione appropriata a seconda dell'oggetto, grazie al **binding dinamico**.

Metodi virtuali puri

Se si desidera impedire la definizione di oggetti della classe *FiguraGeometrica*, consentendone la creazione solo nelle classi derivate, allora è possibile rendere *Area*() e *Perimetro*() **funzioni virtuali pure**, definendole nella classe *FiguraGeometrica* con la sintassi:

virtual float Area() = 0;
virtual float Perimetro() = 0;

Funzioni virtuali pure

Classi astratte pure

Se in una classe esiste una funzione virtuale pura, la classe si dice **classe astratta pura**.

Il compito delle **classi astratte pure** è solo di fungere da interfaccia comune per altre classi derivate.

Se una classe derivata non ridefinisce una funzione virtuale pura, è anch'essa una classe astratta pura.

Il polimorfismo

Come abbiamo visto, il polimorfismo interviene in diverse situazioni (costruttori, overloading e overriding).

Vediamo un altro caso in cui esso si presenta in modo assai interessante sugli operatori.

Sovraccarico di operatori

Il **sovraccarico di operatori** (*operator overloading*)è una tecnica che consente di ridefinire operatori (ad es. "+", "=",ecc) in modo poter operare con essi su dati complessi con la stessa sintassi dei tipi semplici

Il sovraccarico di operatori già esiste nei tipi di dato semplice:

- il compilatore distingue se l'operatore "+" agisce tra tipi interi o reali);
- l'operatore di assegnazione di un oggetto ad un altro della stessa classe, implica l'assegnazione uno ad uno di tutti i dati membro del primo al secondo.

Sovraccarico di operatori

Tuttavia, l'overloading di operatori ha delle limitazioni, in quanto NON consente:

- la creazione di **nuovi** operatori
- la modifica dell'arità di un operatore
- l'alterazione della precedenza degli operatori
- la modifica dell'associatività degli operatori
- la presenza di argomenti di default

Sovraccarico di operatori

Limiti dell'uso dell'operator overloading:

Operatori C++ su cui si puo' fare overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operatori C++ su cui NON si puo' fare overloading

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Sovraccarico di operatori

L'*operator overloading* può essere realizzato con funzioni membro o con funzioni friend.

In generale, per sovraccaricare operatori come ( ), [ ], -> e tutti gli assegnamenti, si devono usare funzioni membro (ossia definite nella classe) poiché il valore di uscita *deve essere un oggetto della classe*.

Se un operatore in sovraccarico *deve modificare membri privati o protetti* della classe o se *l'operando più a sinistra non appartiene alla classe*, il sovraccarico si implementa con una funzione friend.

Sovraccarico di operatori

Iniziamo a mostrare un caso particolare di *operator overloading* (lettura e stampa di un vettore).

Successivamente, applicheremo questo casi di polimorfismo ad altri operatori.

Ridefinizione di "<<" e ">>"

```
class vettore
{   friend istream &operator>> (istream &s, vettore &vett);
    friend ostream &operator<<(ostream &s, vettore &vett);
    private:
        int v[MAX];
    public:
    vettore()
    {   int i;
        for (i=0;i<MAX;i++)   v[i]=0;
    }
};
```

Questa classe dichiara il tipo *vettore* ed introduce due metodi **&operator>>** e **&operator<<** che ridefiniscono le operazoni di lettura e stampa su un oggetto di classe *vettore*.

Notare che sono implementati come metodi friend.

Notare che sono implementati come metodi friend, poiché **istream** e **ostream** non sono oggetti della classe.

Ridefinizione di “<<“ e “>>”

```
istream &operator>>(istream &s, vettore &vett)
{
        for (int i=0;i<MAX; i++)
            s>>vett.v[i];
    return s;
}

ostream &operator<<(ostream &s, vettore &vett)
{
        for (int i=0;i<MAX; i++)
            s<<vett.v[i]<<endl;
    return s;
}
```

L’espansione dei due metodi ridefinisce le operazioni di lettura e di stampa di un vettore.

Ridefinizione di “<<“ e “>>”

```
int main()
{   vettore v;
    cout<<"Lettura vettore "<<MAX<<" elementi"<<endl;
    cin>>v;
    cout<<"Stampa vettore "<<MAX<<" elementi"<<endl;
    cout<<v;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Dopo aver definito un oggetto *v* di classe *vettore*, lo si legge e lo si stampa come fosse una variabile semplice.

Sovraccarico di operatori

La sintassi per definire l'overloading di un operatore mediante una funzione membro è:

tipo **operator** *simbolo_op* (*lista_par*)
{ *istruzioni* }

dove

- *tipo* è il tipo del valore di ritorno
- *simbolo_op* è il simbolo dell'operatore ridefinito
- *lista_par* è l'elenco degli operandi usati come parametri

Ridefinizione di "+"

In tal caso, l'operazione verrà istanziata con una sintassi del tipo:

risultato = op1 simbolo_op op2

In questi casi si può scrivere come funzione friend all'interno della classe, l'interfaccia

friend *classe* &simbolo_op (*classe* &op1, *classe* &op3)

che va implementata all'esterno della classe stessa.

Ad esempio, nella classe compare:

friend vettore &operator+(vettore &a, vettore &b);

Ridefinizione di “+”

```
vettore operator+(vettore &a, vettore &b)
{   vettore somma;
      for (int i=0;i<MAX; i++)
            somma.v[i]=a.v[i]+b.v[i];
    return somma;
}
int main()
{   vettore v, a, b;
    cin>>a;
    cin>>b;
    v=a+b;
    cout<<"Vettore somma"<<endl;
    cout<<v<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Il metodo definisce esternamente l’operatore “+”

L’assegnazione calcola prima la somma tra *a* e *b*, mediante l’operatore “+” sovraccaricato.

Terminologia

- Metodi virtuali
- Metodi virtuali puri
- Classi astratte
- Classi astratte pure
- Sovraccarico degli operatori

Argomenti

- Classi astratte
- Classi astratte e metodi virtuali
- La classe astratta *Punto*
- Classe astratta *FiguraGeometrica*
- Metodi virtuali
- Metodi virtuali puri
- Classi astratte pure
- Il polimorfismo
- Sovraccarico di operatori
- Ridefinizione di “<<“ e “>>”
- Ridefinizione di “+”

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