UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI **MATEMATICA E INFORMATICA**

Programmazione ad Oggetti

Carmine Dodaro

Anno Accademico 2019/2020

Programmazione ad oggetti

I concetti fondamentali

- Incapsulamento
- Composizione
- Ereditarietà
- Polimorfismo

Programmazione ad oggetti

I concetti fondamentali

- ~~Incapsulamento~~
- ~~Composizione~~
- ~~Ereditarietà~~
- Polimorfismo

Ereditarietà e polimorfismo

Professore.h

```
class Professore : public Persona {
  public:
    Professore();

    float getStipendio() const;
    void setStipendio(float);
  private:
    float stipendio;
};
```

Studente.h

```
class Studente : public Persona {
  public:
    Studente();

    float getIsee() const;
    void setIsee(float);
  private:
    float isee;
};
```

Persona.h

```
class Persona {
  public:
    Persona();
    string getCodiceFiscale() const;
    string getNome() const;
    string getCognome() const;
    void setCodiceFiscale(string);
    void setNome(string);
    void setCognome(string);

  private:
    string codiceFiscale;
    string nome;
    string cognome;
};
```

Ereditarietà e polimorfismo

Professore.h

```
class Professore : public Persona {
  public:
    Professore();
    float getStipendio() const;
    void setStipendio(float);
    void stampa() const;

  private:
    float stipendio;
};
```

Studente.h

```
class Studente : public Persona {
  public:
    Studente();
    float getIsee() const;
    void setIsee(float);
    void stampa() const;

  private:
    float isee;
};
```

Persona.h

```
class Persona {
  public:
    Persona();
    string getCodiceFiscale() const;
    string getNome() const;
    string getCognome() const;
    void setCodiceFiscale(string);
    void setNome(string);
    void setCognome(string);

    void stampa() const;

  private:
    string codiceFiscale;
    string nome;
    string cognome;
};
```

Ereditarietà e polimorfismo

Implementazione

```
void Persona::stampa() const {
    cout << codiceFiscale << " " << nome << " " << cognome; }

void Studente::stampa() const {
  Persona::stampa();
  cout << " " << isee; }

void Professore::stampa() const {
  Persona::stampa();
  cout << " " << stipendio; }
```

Ereditarietà e polimorfismo

Implementazione

```
void Persona::stampa() const {
    cout << codiceFiscale << " " << nome << " " << cognome; }

void Studente::stampa() const {
  Persona::stampa();
  cout << " " << isee; }

void Professore::stampa() const {
  Persona::stampa();
  cout << " " << stipendio; }
```

Output?

```
Persona* p1 = new Persona();
Persona* p2 = new Professore();
Persona* p3 = new Studente();

p1->stampa();      p2->stampa();      p3->stampa();
```

Quale metodo di stampa è chiamato nei 3 casi?

Ereditarietà e polimorfismo

Implementazione

```
void Persona::stampa() const {
    cout << codiceFiscale << " " << nome << " " << cognome; }

void Studente::stampa() const {
  Persona::stampa();
  cout << " " << isee; }

void Professore::stampa() const {
  Persona::stampa();
  cout << " " << stipendio; }
```

Output?

```
Persona* p1 = new Persona();
Persona* p2 = new Professore();
Persona* p3 = new Studente();

p1->stampa();      p2->stampa();      p3->stampa();
```

Quale metodo di stampa è chiamato nei 3 casi? Sempre quello di persona!

Override

Come possiamo ridefinire il comportamento di stampa in modo da stampare le informazioni di studente e professore?

Override

Come possiamo ridefinire il comportamento di stampa in modo da stampare le informazioni di studente e professore?

Effettuando l'override del metodo stampa! Quindi, sostituendo l'implementazione della classe base con l'implementazione della classe derivata. Questo consente a porzioni di codice di essere:

- indipendenti dagli specifici oggetti utilizzati, ma
- specifici nel comportamento in fase di esecuzione.

In c++, possiamo usare la keyword virtual per effettuare l'override dei metodi della classe base.

Ereditarietà e polimorfismo

Persona.h

```
class Persona {
  public:
    ...
    virtual void stampa() const;
    ...
};
```

Output?

```
Persona* p1 = new Persona();
Persona* p2 = new Professore();
Persona* p3 = new Studente();

p1->stampa();      p2->stampa();      p3->stampa();
```

Quale metodo di stampa è chiamato nei 3 casi?

Ereditarietà e polimorfismo

Persona.h

```
class Persona {
  public:
    ...
    virtual void stampa() const;
    ...
};
```

Output?

```
Persona* p1 = new Persona();
Persona* p2 = new Professore();
Persona* p3 = new Studente();

p1->stampa();      p2->stampa();      p3->stampa();
```

Quale metodo di stampa è chiamato nei 3 casi?
Nel caso di p1, il metodo stampa di Persona.
Nel caso di p2, il metodo stampa di Professore.
Nel caso di p3, il metodo stampa di Studente.

Ereditarietà e polimorfismo

Attenzione

```
Persona* p1 = new Professore();
p1->stampa(); //Invoca il metodo di stampa di Professore

Professore p2;
p2.stampa(); //Invoca il metodo di stampa di Professore

Persona p3 = p2;
p3.stampa(); //Invoca il metodo di stampa di Persona

Persona& p4 = p2;
p4.stampa(); //Invoca il metodo di stampa di Professore
```

Alcune considerazioni

Il polimorfismo indica la capacità di richiamare su vari oggetti uno stesso metodo che agisce in modo diverso in base al tipo di oggetto su cui è richiamato.

In c++, il polimorfismo si può attuare effettuando l'override delle funzioni virtual della classe base.

Il metodo giusto da invocare è stabilito a tempo di esecuzione (late binding o dynamic binding).

Classi e metodi astratti

Classi astratte

Una classe astratta:

- Lo scopo di una classe astratta è quello di essere usata come classe base di qualche altra classe
- Gli oggetti della classe astratta non possono essere istanziati!
- Possono essere utilizzati puntatori e referenze
- Una classe è astratta se almeno uno dei suoi metodi è astratto (o virtuale puro)

Classi e metodi astratti

FiguraGeometrica.h

```
#ifndef FIGURA_GEOMETRICA_H
#define FIGURA_GEOMETRICA_H

class FiguraGeometrica {
  public:
    FiguraGeometrica() {}
    unsigned int area();//Come si implementa questo metodo?
    unsigned int perimetro();//Come si implementa questo metodo?
};

#endif
```

Classi e metodi astratti

FiguraGeometrica.h

```
#ifndef FIGURA_GEOMETRICA_H
#define FIGURA_GEOMETRICA_H

class FiguraGeometrica {
  public:
    FiguraGeometrica() {}
    unsigned int area();//Come si implementa questo metodo?
    unsigned int perimetro();//Come si implementa questo metodo?
};

#endif
```

Osservazioni

- La classe FiguraGeometrica non ha un modo per implementare i metodi area() e perimetro()
- L'implementazione di area() e perimetro() viene demandata alle classi derivate

Classi e metodi astratti

Metodi virtuali puri

```
#ifndef FIGURA_GEOMETRICA_H
#define FIGURA_GEOMETRICA_H

class FiguraGeometrica {
  public:
    FiguraGeometrica() {}
    virtual unsigned int area() = 0;
    virtual unsigned int perimetro() = 0;
};
#endif
```

- I metodi area() e perimetro() sono virtuali puri
- L'implementazione di area() e perimetro() viene demandata alle classi derivate
- Le classi che ereditano da FiguraGeometrica devono implementare i metodi area() e perimetro() oppure renderli virtuali puri
- Non è possibile istanziare oggetti di tipo FiguraGeometrica. Ad esempio, FiguraGeometrica* f = new FiguraGeometrica(); oppure FiguraGeometrica f;
- È possibile istanziare con classi derivate. Ad esempio, è possibile fare FiguraGeometrica* f = new Quadrato();

Implementazione classi derivate

Quadrato.h

```
#ifndef QUADRATO_H
#define QUADRATO_H

class Quadrato : public FiguraGeometrica {
  public:
    Quadrato(unsigned int l) : FiguraGeometrica(), lato(l) {}
    unsigned int area() { return lato*lato; }
    unsigned int perimetro() { return lato*4; }

  private:
    int lato;
};
#endif
```

Implementazione classi derivate

Rettangolo.h

```
#ifndef RETTANGOLO_H
#define RETTANGOLO_H

#include "FiguraGeometrica.h"

class Rettangolo : public FiguraGeometrica {
  public:
    Rettangolo(unsigned int b, unsigned int a) : FiguraGeometrica() {
      base = b;
      altezza = a;
    }
    unsigned int area() { return base*altezza; }
    unsigned int perimetro() { return (base+altezza)*2; }

 private:
    unsigned int base;
    unsigned int altezza;
};

#endif
```

Classe astratta: esempio

main.cpp

```
#include <iostream>
#include <vector>
using namespace std;
#include "Rettangolo.h"
#include "Quadrato.h"
#include "FiguraGeometrica.h"

int main()  {
  Rettangolo* r1 = new Rettangolo(2,3);
  Rettangolo* r2 = new Rettangolo(4,5);
  Quadrato* q1 = new Quadrato(4);
  Quadrato* q2 = new Quadrato(5);

  vector<FiguraGeometrica*> figure = {r1, r2, q1, q2};
  for(auto f : figure) {
    cout << "Perimetro: " << f->perimetro() << endl;
    cout << "Area: " << f->area() << endl;
  }

  delete r1; delete r2; delete q1; delete q2;
  return 0;
}
```

Distruttori virtuali

main.cpp

```
int main()  {
  FiguraGeometrica* r1 = new Rettangolo(2,3);
  delete r1;
}
```

In questo esempio, cancellare r1 ha un comportamento indefinito. Per ovviare a questa situazione, il distruttore di FiguraGeometrica deve essere virtual. In generale, è opportuno che le classi che si prevedono verranno ereditate devono aggiungere la keyword virtual al distruttore.

Distruttori virtuali

main.cpp

```
int main()  {
  FiguraGeometrica* r1 = new Rettangolo(2,3);
  delete r1;
}
```

In questo esempio, cancellare r1 ha un comportamento indefinito. Per ovviare a questa situazione, il distruttore di FiguraGeometrica deve essere virtual. In generale, è opportuno che le classi che si prevedono verranno ereditate devono aggiungere la keyword virtual al distruttore.

```
#ifndef FIGURA_GEOMETRICA_H
#define FIGURA_GEOMETRICA_H

class FiguraGeometrica {
  public:
    FiguraGeometrica() {}
    virtual ~FiguraGeometrica() {}
    virtual unsigned int area() = 0;
    virtual unsigned int perimetro() = 0;
};
#endif
```