Università della Calabria
Corso di Laurea in Ingegneria Elettronica

Programmazione Orientata agli Oggetti

A.A. 2006/2007

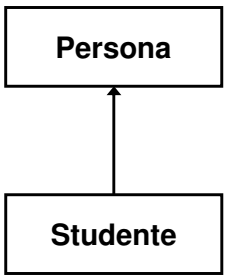
Ereditarietà e polimorfismo

Paolo Trunfio *

* DEIS, Università della Calabria – http://si.deis.unical.it/~trunfio

Derivazione di classi (1)

```
class Persona {
  public:
    string nome;
    string cognome;
    int annoDiNascita;
};

class Studente: public Persona {
  public:
    int matricola;
    string corsoDiStudio;
};
```



La classe Persona è detta *superclasse* (o *classe base*). La classe Studente, che *eredita* i membri della superclasse, è detta *sottoclasse* (o *classe derivata*). Le classi sono rappresentate con un rettangolo e la derivazione è rappresentata con una freccia dalla sottoclasse alla superclasse.

Derivazione di classi (2)

```
int main() {
  Studente s;
  s.nome = "Mario";
  s.cognome = "Rossi";
  s.annoDiNascita = 1990;
  s.matricola = 12345;
  s.corsoDiStudio = "Laurea in Ingegneria Elettronica";
}
```

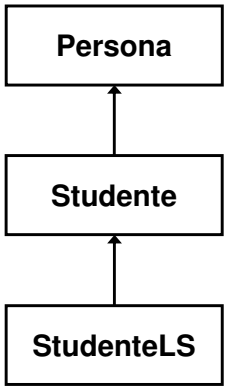
nome	“Mario”
cognome	“Rossi”
annoDiNascita	1990
matricola	12345
corsoDiStudio	“Laurea in Ingegneria Elettronica”

Persona

Studente

Derivazione di classi (3)

```
// Studente di un corso di Laurea Specialistica:
class StudenteLS: public Studente {
  public:
    string laurea; // laurea triennale già conseguita
    int votoDiLaurea; // voto di laurea triennale
};
```



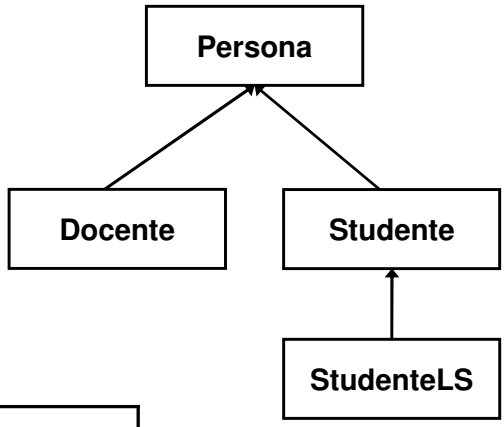
nome	“Mario”
cognome	“Rossi”
annoDiNascita	1990
matricola	54321
corsoDiStudio	“Laurea Specialistica in Ingegneria Elettronica”
laurea	“Laurea in Ingegneria Elettronica”
votoDiLaurea	105

Studente

StudenteLS

Derivazione di classi (4)

```
class Docente: public Persona {
  public:
    string facolta;
    string dipartimento;
};
```



nome	“Guglielmo”
cognome	“Marconi”
annoDiNascita	1874
facolta	“Ingegneria”
dipartimento	“DEIS”

Persona

Docente

Assegnamenti ad oggetti

```
int main() {
  Persona p;
  Studente s1;
  StudenteLS s2;
  Docente d;
  ...inizializzazione oggetti...
  p = d; // ok
  p = s1; // ok
  p = s2; // ok
  s1 = s2; // ok
  s1 = p; // errore
  s1 = d; // errore
  s2 = s1; // errore
}
```

nome	“Guglielmo”
cognome	“Marconi”
annoDiNascita	1874
facolta	“Ingegneria”
dipartimento	“DEIS”

d

p = d; // conversione implicita

nome	“Guglielmo”
cognome	“Marconi”
annoDiNascita	1874

p

Un oggetto di una sottoclasse può essere convertito in un oggetto di una superclasse, ma non viceversa.

Assegnamenti a puntatori

```
int main() {
  Persona p;
  Studente s1;
  StudenteLS s2;
  ...inizializzazione oggetti...
  Persona *ptp;
  Studente *pts;
  ptp = &p; // ok
  ptp = &s1; // ok
  ptp = new Persona; // ok
  ptp = new Studente; // ok
  ptp = &s2; // ok
  pts = new Studente; // ok
  pts = &p; // errore
}
```

Il riferimento ad un oggetto di una sottoclasse può essere assegnato ad un puntatore di una superclasse, ma non viceversa.

Funzioni membro (1)

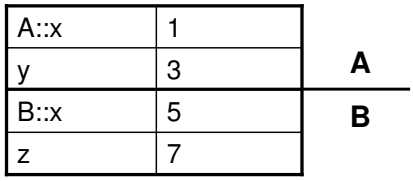
```
class Persona {
  public:
    string nome;
    string cognome;
    int annoDiNascita;
    void stampaDatiPersonali() {
      cout << nome << " " << cognome << " " << annoDiNascita << endl;
    }
};

class Studente: public Persona {
  public:
    int matricola;
    string corsoDiStudio;
    void stampaDatiUniversitari() {
      cout << matricola << " " <<  corsoDiStudio << endl;
    }
};
```

Funzioni membro (2)

```
int main() {
  Studente s;
  ...inizializza s...
  s.stampaDatiPersonali(); // ok, stampa: nome, cognome e anno
  s.stampaDatiUniversitari(); // ok, stampa: matricola e corso di studio
  Persona p;
  ...inizializza p...
  p.stampaDatiPersonali(); // ok, stampa: nome, cognome e anno
  p.stampaDatiUniversitari(); // errore: metodo non disponibile
  p = s;
  p.stampaDatiPersonali(); // ok, stampa: nome, cognome e anno
  p.stampaDatiUniversitari(); // errore: metodo non disponibile
}
```


Membri con lo stesso nome

```
class A {
  public:
    int x;
    int y;
};
class B: public A {
  public:
    int x;
    int z;
};
int main() {
  B b; // b è un'istanza della classe B
  b.x = 3; // equivale a: b.B::x = 3
  b.y = 5;
  b.z = 7;
  b.A::x = 1; // per accedere alla x
              // definita nella classe A
}
```



Se sono presenti due campi con lo stesso nome nella gerarchia, di default viene acceduto il campo definito nella classe più vicina a quella dell'oggetto. Per accedere al campo non di default si utilizza il risolutore di visibilità (operatore ::).

La stessa regola si applica nel caso di funzioni membro con lo stesso nome.

Costruttori (1)

```
class Persona {
  …come nei lucidi precedenti…
  Persona(string n, string c, int a) {
    nome = n; cognome = c; annoDiNascita = a;
  }
};

class Studente: public Persona {
  …come nei lucidi precedenti…
  Studente(string n, string c, int a, int m, string cs): Persona(n, c, a) {
    matricola = m;
    corsoDiStudio = cs;
  }
};
```

lista di inizializzazione

Quando si invoca il costruttore della sottoclasse viene eseguita prima la **lista di inizializzazione**, e dopo il costruttore della sottoclasse.

Costruttori (2)

Se il costruttore della sottoclasse non invoca il costruttore della superclasse nella lista di inizializzazione, viene invocato il costruttore di default della superclasse.

```
class Persona {
  …
  Persona(string n, string c, int a) {
    nome = n;
    cognome = c;
    annoDiNascita = a;
  }
  Persona() { // costruttore di default
    nome = ""; cognome = "";
    annoDiNascita = 1900;
  }
};
```

```
class Studente: public Persona {
  …
  Studente(string n, string c, int a,
           int m, string cs) {
    // manca la lista di inizializzazione:
    // invoca automaticamente il
    // costruttore di default di Persona
    matricola = m;
    corsoDiStudio = cs;
  }
};
```

NOTA: in questo caso, se non fosse presente il costruttore di default nella classe Persona, si produrrebbe un errore in fase di compilazione.

Distruttori

```
class A {
  public:
    A() {
      cout << "crea A\n";
    }
    ~A() {
      cout << "distrugge A\n";
    }
};

class B: public A {
  public:
    B() {
      cout << "crea B\n";
    }
    ~B() {
      cout << "distrugge B\n";
    }
};
```

```
void f() {
  B b;
}
```

L'invocazione della funzione **f** produce il seguente output:

crea A
crea B
distrugge B
distrugge A

Mentre il costruttore della superclasse viene eseguito prima di quello della sottoclasse, il distruttore della superclasse viene eseguito dopo di quello della sottoclasse.

Qualificatori di accesso (1)

```
class Persona {
  private:
    string nome;
    string cognome;
  public:
    Persona(string n, string c) {
      nome = n;
      cognome = c;
    }
    void stampaDati() {
      cout << "Persona: "
           << nome << " "
           << cognome << endl;
    }
};
```

```
class Studente: public Persona {
    int matricola;
  public:
    Studente(string n, string c, int m):
      Persona(n,c) {
      matricola = m;
    }
    void stampaDati() {
      cout << "Studente: "
           << nome << " " // errore
           << cognome << " " // errore
           << matricola << endl;
    }
};
```

Il metodo **stampaDati** della classe Studente non funziona perché i campi **nome** e **cognome** della superclasse sono privati e quindi inaccessibili.

Qualificatori di accesso (2)

Per rendere i campi non pubblici accessibili anche alle sottoclassi è necessario fare uso del qualificatore d'accesso "**protected**":

```
class Persona {
 protected:
   string nome;
   string cognome;
 public:
   …come nel lucido precedente…
}
```

```
class Studente: public Persona {
   …come nel lucido precedente…
   void stampaDati() {
     cout << "Studente: "
        << nome << " " // ok
        << cognome << " " // ok
        << matricola << endl;
   }
}
```

I campi protected sono accessibili dalle sottoclassi, ma inaccessibili dall'esterno.

Qualificatori di accesso (3)

```
class A {
   int x; // private
 protected:
   int y;
 public:
   int z;
};
class B: public A {
 public:
   void setX(int n) {
     x = n; // errore: x privato in A
   }
   int getY() {
     return y; // ok
   }
   void setZ(int n) {
     z = n; // ok
   }
};
```

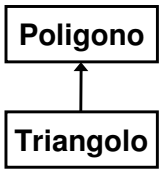
```
int main() {
  A a;
  a.x = 1; // errore
  cout << a.x; // errore
  a.y = 3; // errore
  a.z = 5; // ok
  B b;
  b.x = 2; // errore
  b.y = 4; // errore
  cout << b.y; // errore
  cout << b.getY(); // ok
  b.z = 6; // ok
}
```

Funzioni virtuali (1)

Supponiamo di avere le seguenti classi base e derivata, entrambe con una funzione **stampaInfo**:

```
class Poligono {
   int numLati;
 public:
   Poligono(int n) {
     numLati = n;
   }
   int numeroLati() {
     return numLati;
   }
   void stampaInfo() {
      cout << "Poligono con "
           << numLati << " lati\n";
   }
};
```

```
class Triangolo: public Poligono {
   float a, b, c;
 public:
   Triangolo(float a, float b, float c):
   Poligono(3) {
   this->a = a;
   this->b = b;
   this->c = c;
 }
 void stampaInfo() {
     cout << "Triangolo con i "
     << "seguenti lati: " << a << ", "
     << b << " e " << c << "\n";
   }
};
```



Funzioni virtuali (2)

```
int main() {

  Poligono *p = new Poligono(5);
  p->stampaInfo(); // Poligono con 5 lati

  Triangolo *t = new Triangolo(3, 4, 5);
  t->stampaInfo(); // Triangolo con i seguenti lati; 3, 4 e 5

  Poligono *p2 = t; // punta ad un Triangolo
  p2->stampaInfo(); // Poligono con 3 lati
}
```

La scelta della funzione da eseguire avviene “a tempo di compilazione” **in base al tipo del puntatore**.

Funzioni virtuali (3)

Modifichiamo le classi precedenti in modo che la funzione **stampaInfo** sia definita **"virtual"**:

```
class Poligono {
   …come nei lucidi precedenti…
   void virtual stampaInfo() {
     cout << "Poligono con " << numLati << " lati\n";
   }
};

class Triangolo: public Poligono {
   …come nei lucidi precedenti…
   void virtual stampaInfo() { // "virtual" qui si può omettere
     cout << "Triangolo con i seguenti lati: "
          << a << ", "  << b << " e " << c << "\n";
   }
};
```


Funzioni virtuali (4)

```
int main() {

  Poligono *p = new Poligono(5);
  p->stampaInfo(); // Poligono con 5 lati

  Triangolo *t = new Triangolo(3, 4, 5);
  t->stampaInfo(); // Triangolo con i seguenti lati; 3, 4 e 5

  Poligono *p2 = t; // punta ad un Triangolo
  p2->stampaInfo(); // Triangolo con i seguenti lati; 3, 4 e 5

}
```

La scelta della funzione da eseguire avviene "a tempo d'esecuzione" **in base al tipo dell'oggetto effettivamente puntato** (*polimorfismo*).

Funzioni virtuali (5)

Le funzioni virtuali hanno effetto solo con i puntatori. Quindi, anche assumendo che **stampaInfo** sia una funzione virtuale, la sua invocazione su oggetti ha il seguente effetto:

```
int main() {
  Poligono p(5);
  p.stampaInfo(); // Poligono con 5 lati

  Triangolo t(3, 4, 5);
  t.stampaInfo(); // Triangolo con i seguenti lati; 3, 4 e 5

  Poligono p2 = t; // conversione ad un oggetto Triangolo
  p2.stampaInfo(); // Poligono con 3 lati
}
```


Funzioni virtuali (6)

```
void stampaLista(const list<Poligono*> &poligoni) {
  list<Poligono*>::const_iterator it = poligoni.begin();
  while (it != poligoni.end()) {
    (*it)->stampaInfo(); // invocazione di funzione virtuale
    it++;
  }
}

int main() {
  Triangolo *t = new Triangolo(3, 4, 5);
  Poligono *p1 = new Poligono(4);
  Poligono *p2 = new Poligono(5);
  list<Poligono*> poligoni;
  poligoni.push_back(t);
  poligoni.push_back(p1);
  poligoni.push_back(p2);
  stampaLista(poligoni);
}
```

Output:

```
Triangolo con i seguenti lati: 3, 4 e 5
Poligono con 4 lati
Poligono con 5 lati
```

Funzioni virtuali pure e classi astratte (1)

```
class Poligono {
  private:
    int numLati;
  public:
    Poligono(int n) {
      numLati = n;
    }
    int numeroLati() {
      return numLati;
    }
    virtual float perimetro() = 0;
    virtual float area() = 0;
};
```

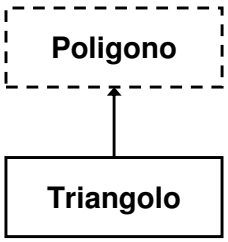
funzioni virtuali pure

Una classe è **astratta** se contiene una o più **funzioni virtuali pure**.

Non è possibile istanziare oggetti di una classe astratta.

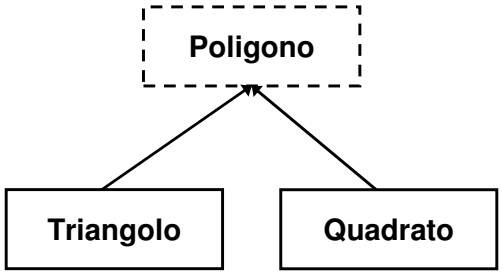
Funzioni virtuali pure e classi astratte (2)

```
#include<math.h>

class Triangolo: public Poligono {
  private:
    float a, b, c;
  public:
    Triangolo(float a, float b, float c): Poligono(3) {
      this->a = a; this->b = b; this->c = c;
    }
    float perimetro() {
      return a+b+c;
    }
    float area() {
      float s = perimetro()/2;
      return sqrt(s*(s-a)*(s-b)*(s-c)); // formula di Erone
    }
};
```



Funzioni virtuali pure e classi astratte (3)

```
class Quadrato: public Poligono {
  private:
    float lato;
  public:
    Quadrato(float lato): Poligono(4) {
      this->lato = lato;
    }
    float perimetro() {
      return lato * 4;
    }
    float area() {
      return lato * lato;
    }
};
```



Funzioni virtuali pure e classi astratte (4)

```
float areaTotale(const list<Poligono*> &poligoni) {
  float at = 0;
  list<Poligono*>::const_iterator it = poligoni.begin();
  while (it != poligoni.end()) {
    at += (*it)->area(); // invocazione di funzione virtuale pura
    it++;
  }
  return at;
}

int main() {
  Triangolo *t = new Triangolo(3, 4, 5);
  Quadrato *q = new Quadrato(5);
  list<Poligono*> poligoni;
  poligoni.push_back(t);
  poligoni.push_back(q);
  cout << areaTotale(poligoni); //  stampa: 31
}
```