ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

EREDITARIETÀ

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI (SAVITCH CAPITOLO 15.1)

1. ereditarietà
2. sottotipaggio
3. costruttori

EREDITARIETÀ

spesso non si vuole definire una classe a partire dal nulla, ma vogliamo **definirla a partire da** un'altra classe

* uno `studente` è anche una `persona`
* se abbiamo già definito una classe `persona` vorremmo riusare il codice già scritto

EREDITARIETÀ – ESEMPIO

- uno `studente` è una **sottoclasse** di `persona`
  - ha **tutti i campi** di persona
  - ha **tutti i metodi** di persona, eventualmente modificati (tranne il costruttore)
  - può avere altri metodi e altri campi
- come `persona`, uno `studente` ha `nome`, `cognome`, `indirizzo`, `telefono`, …
- in aggiunta uno `studente` ha una lista di esami fatti e una media voti, che una `persona` generica non ha

PERCHÈ L'EREDITARIETÀ?

perchè si desidera riusare il codice già scritto

- non si riscrivono tutti i campi e i metodi di una classe
- si scrivono solamente quelli **nuovi** e/o **modificati**

il riuso del codice aiuta a risparmiare

- **tempo di scrittura**: meno codice da scrivere
- **tempo di debugging**: meno codice = meno errori e quindi meno debugging
- **tempo di modifica**: meno codice da modificare
- **denaro**: se pago qualcuno per scrivere il codice

senza ereditarietà i linguaggi orientati agli oggetti avrebbero avuto meno successo

TERMINOLOGIA

- `studente` si dice classe derivata, o **sottoclasse**, o classe figlio di `persona`
- viceversa, `persona` si dice classe base, o classe padre, o **superclasse** di `studente`
- la modifica di un metodo della classe padre nella sottoclasse si chiama **overriding**

ESEMPIO

```
class persona {
    protected:
        char nome[50];
        int eta;
    public:
        persona(char n[]="",int e=0){
            strcpy(nome,n); eta=e;
        }
        void presentati(){
            cout << "Sono " << nome << " e ho " << eta << anni\n " ;
            }
};
class studente: public persona {
    protected:
        double media;
    public:
        studente(char n[], int e, double m){
            strcpy(nome,n); eta=e; media=m;
        }
        void presentati(){
            cout << "Sono "<< nome << ", ho "<< eta<<" anni e media"
            << media ;
        }
};
```

ALTRO ESEMPIO

```
class rectangle{
    protected:
        double base ;
        double altezza ;
    public:
        double area(){ return(base*altezza) ;
        }
        double perimeter(){ return(2*base + 2*altezza) ;
        }
        rectangle(int m =0 , int n=0){ base = abs(m); altezza = abs(n);
        }
} ;
class triangle_rect : public rectangle {
    public:
        double area(){ return((base*altezza)/2) ;
        }
        double perimeter(){
            return(base + altezza + sqrt(base*base + altezza*altezza));
        }
        triangle_rect(int m,int n){ base = abs(m); altezza = abs(n); }
} ;
```

SINTASSI

questo è l'`access-level`: definisce il tipo della derivazione (in questo corso sarà sempre `public`)

```
class nomeFiglio: public nomePadre{
    campi nuovi e metodi nuovi o modificati
}
```

- **`nomeFiglio`** è il nome della nuova classe, **`nomePadre`** il nome della classe da cui si eredita
- la nuova classe **ha tutti i campi della classe vecchia**, più quelli nuovi
- la nuova classe **ha tutti i metodi della classe vecchia** (tranne il costruttore), eventualmente modificati, più quelli nuovi
- il prototipo del metodo modificato deve essere uguale a quello del corrispondente metodo nel padre

I COSTRUTTORI NON SONO EREDITATI

prima di invocare il costruttore di una sottoclasse viene invocato il costruttore senza parametri della superclasse

- se il costruttore della superclasse **non esiste si ha un errore**
- in questo modo il costruttore della sottoclasse può assumere che i campi della superclasse siano già stati inizializzati

SOTTOTIPAGGIO

una sottoclasse è un SOTTOTIPO della sua superclasse

- quindi è possibile usare un oggetto della sottoclasse in qualunque contesto serva un oggetto della superclasse
  - un oggetto della sottoclasse **si può assegnare** ad una variabile della superclasse
  - un oggetto della sottoclasse **si può passare come parametro** ad una funzione che si aspetta un oggetto della superclasse
  - un oggetto della sottoclasse **si può ritornare come valore di ritorno** se il tipo di tale valore deve essere una superclasse
  - un oggetto della sottoclasse **si può farlo puntare** da un puntatore alla superclasse
- Il contrario non è vero

SOTTOTIPAGGIO: ESEMPIO

```
class rectangle{
    protected:
        double base ;
        double altezza ;
    public:
        double area(){ return(base*altezza) ; }
        double perimeter(){ return(2*base + 2*altezza) ; }
        rectangle(int m =0 , int n=0){ base = abs(m); altezza = abs(n); }
} ;

class triangle_rect : public rectangle {
    public:
        double area(){ return((base*altezza)/2) ;}
        double perimeter(){
             return(base + altezza + sqrt(base*base + altezza*altezza));}
        triangle_rect(int m,int n){ base = abs(m); altezza = abs(n); }
} ;

int main() {
     triangle_rect t = triangle_rect(3,4) ;
     rectangle r = rectangle(2,3) ;
     r = t ;
     cout << r.perimeter() << ' ' << t.perimeter() ;
 }
```

static dispatch in C++

classe `triangle_rect` sottotipo di `rectangle`

`r = t` è possibile perchè `triangle_rect` è sottotipo di `rectangle`

output: 14 12

SOTTOTIPAGGIO: ALTRO ESEMPIO

```
class rectangle{
     protected:
          double base ;
          double altezza ;
     public:
          double area(){ return(base*altezza) ; }
          double perimeter(){ return(2*base + 2*altezza) ; }
          rectangle(int m =0 , int n=0){ base = abs(m); altezza = abs(n); }
} ;

class triangle_rect : public rectangle {
     public:
     double area(){ return((base*altezza)/2) ;}
     double perimeter(){ return(base + altezza + sqrt(base*base + altezza*altezza)); }
     triangle_rect(int m,int n){ base = abs(m); altezza = abs(n); }
} ;

rectangle give_me_a_rect(rectangle x){
     double p = x.perimeter() ;
     triangle_rect z = triangle_rect(p/2,p/2) ;
     return(z) ;
}

int main() {
     triangle_rect q = triangle_rect(3,4) ;
     rectangle r = give_me_a_rect(q) ;
     cout << r.perimeter() ;
}
```

static dispatch in C++

classe `triangle_rect` sottotipo di `rectangle`

ritorna uno `triangle` che è tipato come `rectangle`

`give_me_a_rect` è invocata con un `triangle_rect`

output: 28

ESERCIZI

1. definire una classe **`contoBanca`** con un **`saldo`** e metodi **`versa`** e **`preleva`**. Definire una sottoclasse **`contoInteressi`** con un metodo addizionale che aumenta il saldo del 2%.

2. definire una classe **`persona`** con campi opportuni, un costruttore e metodo **`presentati`**. Definire una sottoclasse **`impiegato`** con in più campi **`reparto`** e **`stipendio`**, un metodo **`presentati`** opportunamente modificato e un metodo **`aumento`** che aumenta lo stipendio del 5%.

ANCORA SUI COSTRUTTORI

una sottoclasse può dichiarare quale costruttore della classe base vuole usare

```
class persona{
    ...
    persona(char n[], int e){ strcpy(nome,n); eta=e; }
    ...
};
class studente: public persona{
    ...
    studente(char n[], int e, double m):persona(n,e){
          media = m ;
          }
    ...
};
```

RICHIAMARE I METODI RIDEFINITI

- quando un metodo è ridefinito nella sottoclasse (**overriding**), si può ancora accedere all'omonimo metodo nella superclasse
- lo si richiama con `superclasse::metodo`

```
class persona{
    ...
    void presentati(){
        cout << "Sono " << nome << " e ho " << eta <<
                " anni\n " ;
        }
    ...
};
class studente: public persona{
    ...
    void presentati(){
        persona::presentati();
        cout << "Ho una media voti " << media << "\n “;
        }
    ...
};
```

GERARCHIA DI CLASSI

- una sottoclasse può essere a sua volta la superclasse di un'altra classe
- `studente` è una sottoclasse di `persona`, e una superclasse di `laureando`
- un `laureando` ha i metodi e campi di `persona`, quelli di `studente` più altri, ad esempio `relatore` e `titoloTesi`
- in questo modo è possibile definire anche gerarchie complicate

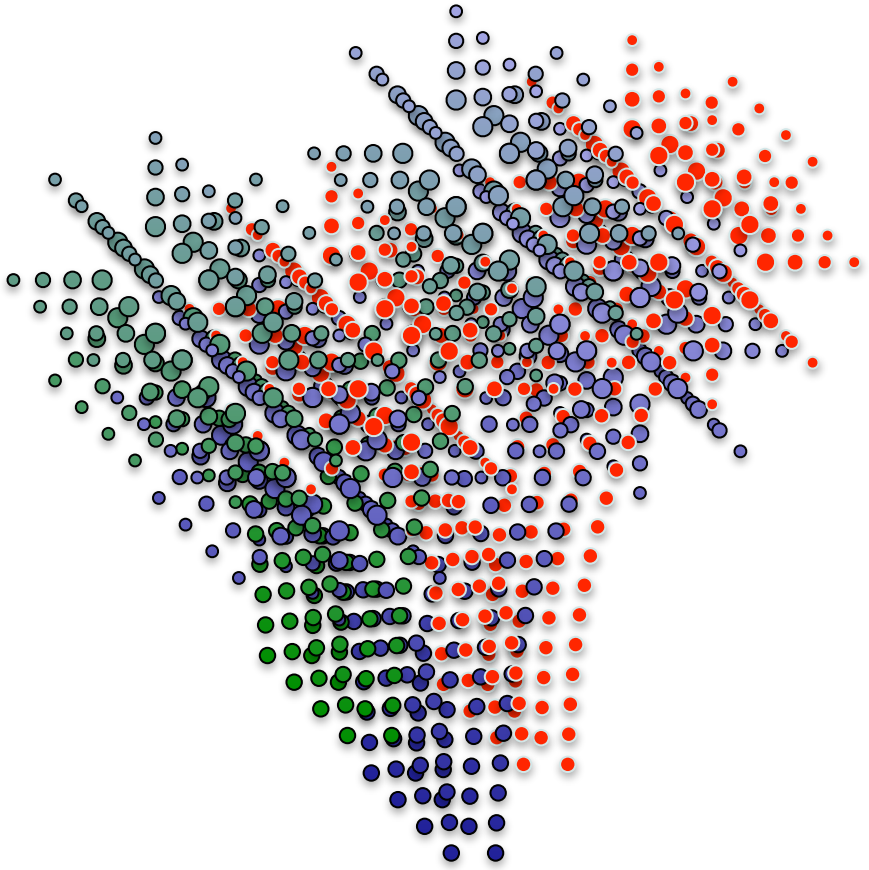
ESERCIZI

1. Definire una classe **`citta`** con campi **`nome`** e **`numeroAbitanti`**, con un opportuno costruttore e metodi **`descrizione`** (che stampa le informazioni sulla città) e **`cambiaAbitanti`**, che cambia il numero di abitanti.

2. Definire una classe **`capoluogo`** che è una **`citta`** con in più l'informazione sulla **`regione`** di cui è capoluogo.

3. Definire una classe **`capitale`** che è un **`capoluogo`** con in più l'informazione sulla **`nazione`** di cui è capitale.

4. Creare gli oggetti per **`Cesenatico`**, **`Bologna`**, **`Torino`** e **`Roma`** e stamparne le informazioni.



FINE