Objektorientering - Arv och polymorfi

Eric Elfving
Institutionen för datavetenskap

LINKÖPING UNIVERSITY

- Med hjälp av arv kan vi bryta ut saker som är gemensamt hos flera klasser.
- Vi får också möjlighet att referera till en grupp av klasser.
- Vi studerar ett par exempel:

Listing 1: Rectangle.h

```
class Rectangle
{
public:
    Rectangle(double h, double w)
        : height{h}, width{w}
    {}
    double area() const
    {
        return height * width;
    }
    double get_height() const
    {
        return height;
    }
    double get_width() const
    {
        return width;
    }
private:
    double height;
    double width;
};
```

Listing 2: Triangle.h

```
class Triangle
{
public:
    Triangle(double h, double w)
        : height{h}, width{w}
    {}
    double area() const
    {
        return height * width / 2.0;
    }
    double get_height() const
    {
        return height;
    }
    double get_width() const
    {
        return width;
    }
private:
    double height;
    double width;
};
```

- De två klasserna är helt olika typer, men har väldigt mycket gemensamt.
- Upprepning av kod bör undvikas - ger risk för buggar
- I C++ (och alla andra objektorienterade språk) kan vi skapa en gemensam basklass som får hantera de delar som klasserna delar.

Listing 3: Shape.h

```
class Shape
{
public:
    Shape(double width, double height)
        : width{width}, height{height}
    {}
    double get_height() const
    {
        return height;
    }
    double get_width() const
    {
        return width;
    }
private:
    double width;
    double height;
};
```

- Sedan låter vi `Rectange` och `Triangle` ärva från `Shape`:

Listing 4: Rectangle.h

```
class Rectangle: public Shape
{
public:
    Rectangle(double h, double w)
        : Shape{w,h}
    {}
    double area() const
    {
        return height * width;
    }
};
```

Listing 5: Triangle.h

```
class Triangle : public Shape
{
public:
    Triangle(double h, double w)
        : Shape{w,h}
    {}
    double area() const
    {
        return height * width / 2.0;
    }
};
```

- Detta fungerar dock inte!

```
Triangle.h: In member function ‘double Triangle::area() const’:
Triangle.h:10:16: error: ‘double Shape::height’ is private within this context
         return height * width / 2.0;
                ^~~~~~
In file included from Triangle.h:1:0:
Shape.h:17:12: note: declared private here
     double height;
            ^~~~~~
```

- Medlemmarna är deklarerade som privata i Shape!

- `Rectangle` är en `Shape`.
- Vi kan anropa de publika medlemsfunktionerna i `Shape` som om de vore våra:

Listing 6: Rectangle.h

```
class Rectangle: public Shape
{
public:
    Rectangle(double h, double w)
        : Shape{w,h}
    {}
    double area() const
    {
        return get_height() * get_width();
    }
};
```

- Hittills har vi använt medlemsåtkomst `public` och `private`.
- Det finns också en tredje variant, `protected` vilket gör medlemmar tillgängliga som om de vore publika från subklasser men privata i övrigt.
- Det är viktigt att alltid fundera på gränssnittet hos en klass - vilka medlemmar ska vi ge tillgång till och till "vem"?

Listing 7: Shape.h

```
class Shape
{
public:
    Shape(double width, double height)
        : width{width}, height{height}
    {}
    double get_height() const
    {
        return height;
    }
    double get_width() const
    {
        return width;
    }
protected:
    double width;
    double height;
};
```

- Nu kommer vi åt dem som om de vore deklarerade direkt i vår subklass!

- Utifrån kommer vi åt de publika medlemmarna från båda klasserna:

```
int main()
{
    Triangle t{12,4};
    cout << t.get_height() << " " << t.area() << endl;
}
```

- Om vi skapar en funktion som tar emot en referens till `Shape` kan vi även skicka in `Triangle` och `Rectangle`

```
void foo(Shape const & s)
{
    cout << s.get_height() << endl;
}

int main()
{
    Triangle t{12,4};
    foo(t);
}
```

- Problemet är att vi endast når saker som finns publikt i `Shape`.
- Hur får vi reda på arean?

Polymorfi!

- Med nyckelordet `virtual` kan vi deklarera en medlem i en basklass som subklasser kan skriva över.

Listing 8: Shape.h

```
class Shape
{
public:
    // ...
    virtual double area() const;
protected:
    double width;
    double height;
};
```

- Vi behöver inte göra någon ändring i subklasserna, men lägger vi till `override` kommer kompilatorn kontrollera att vi inte råkat skapa en ny funktion (dvs en överlagring).

Listing 9: Rectangle.h

```
class Rectangle: public Shape
{
public:
    Rectangle(double h, double w)
        : Shape{w,h}
    {}
    double area() const override
    {
        return width * height;
    }
};
```

- Detta kommer dock inte kompilera -- vi måste implementera `Shape::area`!
- Frågan är hur vi gör det?

- Vi kan säga till kompilatorn att vi inte vill implemetera en virtuell funktion, den kallas då pure virtual"
- Klassen kallas då abstrakt - vi kan inte skapa objekt av den typen.
- Subklasser måste implementera funktionen för att även de inte ska vara abstrakta

```
class Shape
{
    // ...
    virtual double area() const = 0;
    // ...
};
```

```
void foo(Shape const & s)
{
    cout << s.area() << endl;
}

int main()
{
    Triangle t{12,4};
    foo(t);
    // Shape s{12,3};  // Inte tillåtet, Shape är abstrakt
}
```

- För att få tillgång till polymorfi krävs två saker i C++:
  - En referens (eller pekare) till basklass
  - Anrop till en virtuell funktion

- I C++ brukar man tala om ett uttrycks statiska och dynamiska typ.
- Antag följande deklaration:

```
Triangle t{12,5};
Shape & s { t };
```

- Den statiska typen av s är alltid `Shape &`
- Den dynamiska typen beror på vad s refererar till, i detta fall `Triangle`

- När vi anropar en medlemsfunktion kommer kompilatorn göra följande:
  - Om funktionen inte är virtuell $\Rightarrow$ Anropa funktionen hos den statiska typen
  - Annars, om funktionen är virtuell och den statiska typen är av pekare- eller referenstyp $\Rightarrow$ Anropa funktionen hos den dynamiska typen

- Hittills har vi jobbat med referenser till basklass. Detta fungerar ofta bra, men vi har såklart problem:
  - Vi måste initiera en referens med en existerande variabel
  - Vi kan aldrig ändra på en referens

- Ofta används istället pekare:

```
int main()
{
    Shape * s;
    s = new Triangle{4,7};
    cout << s->area() << endl;
}
```

- Problemet är så klart de vanliga pekarproblemen:
  - Minnesläckor
  - Avreferera felaktiga adresser
  - …
- Senare pratar vi om några hjälpmedel (smartpekare)

- Även om vi kommer ihåg att köra `delete` kommer vi få problem...

```
int main()
{
    Shape * s;
    s = new Triangle{4,7};
    cout << s->area() << endl;
    delete s;
}
```

- Destruktorn är inte virtuell ⇒ endast basklassens destruktor körs

- Vi har ingen speciell resurs att hantera här - den automatgenererade destruktorn fungerar bra.
- Markera den som `= default`

```
class Shape
{
    // ...
    virtual ~Shape() = default;
    // ...
};
```

- Vi har även tillgång till `= delete` om vi vill förbjuda användning av (och generering av) vissa funktioner.

Eric Elfving
Institutionen för datavetenskap

www.liu.se

LINKÖPING UNIVERSITY