TDIU20

Arv, Polymorfi

Eric Ekström

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 UML
6 Typkonvertering

LINKÖPING UNIVERSITY

Specialisering - Två snarlika klasser

```
class Rectangle
{
public:
  Rectangle(int x, int y,
            int w, int h);

  float area() const;
  void print_pos() const;

private:
  int xpos;
  int ypos;
  int w;
  int h;
};
```

Specialisering - Två snarlika klasser

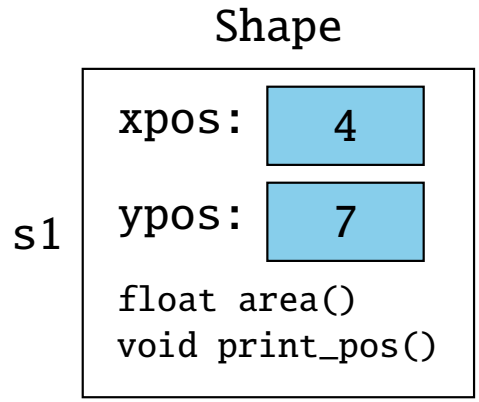
```
class Rectangle
{
public:
  Rectangle(int x, int y,
            int w, int h);

  float area() const;
  void print_pos() const;

private:
  int xpos;
  int ypos;
  int w;
  int h;
};
```

```
class Circle
{
public:
  Circle(int x, int y,
         float r);

  float area() const;
  void print_pos() const;

private:
  int xpos;
  int ypos;
  float r;

};
```

Specialisering - Basklass

```
class Shape
{
public:
  Shape(int x, int y)
    : xpos {x}, ypos {y}
  {}

  float area() const;
  void print_pos() const;

private:
  int xpos;
  int ypos;
};
```

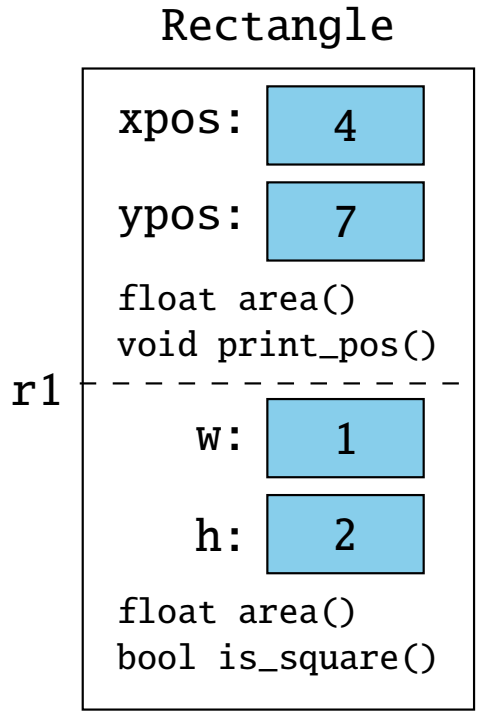
```
int main()
{
  Shape s1 {4, 7};
}
```



Specialisering - Basklass

```
class Rectangle : public Shape
{
public:
  Rectangle(int x, int y,
            int w, int h)
    : Shape {x, y}, w {w}, h {h}
  {}

  float area() const;
  bool is_square() const;

private:
  int w;
  int h;
};
```

```
int main()
{
  Rectangle r1 {4, 7, 1, 2};
}
```



Specialisering - Härledd klass

- Härledda klassen ska alltid initiera sin basklassdel genom att anropa basklassens konstruktor

```
Rectangle(int x, int y, int w, int h)
  : Shape {x, y}
{}
```

- Vi kan göra explicita anrop till basklassens version av en funktion.

```
Shape::area()
```

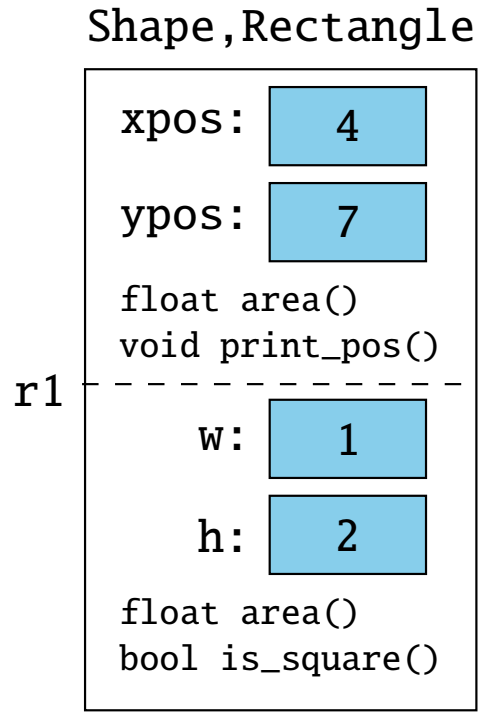
- Basklassens privata medlemmar går inte att komma åt från den härledda klassen.

1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 UML
6 Typkonvertering

Statisk bindning

Hur vet kompilatorn vilken funktion som ska anropas?

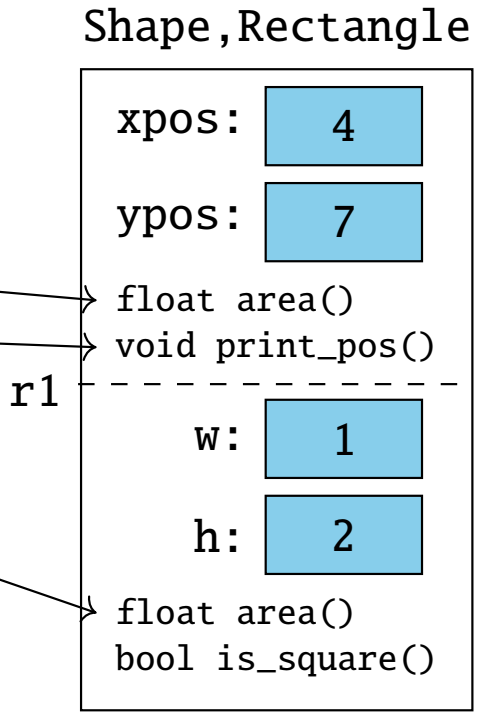
```
Rectangle r1 {4, 7, 1, 2};

Shape& s1 {r1};

cout << s1.is_square()
     << s1.area()
     << r1.is_square()
     << r1.print_pos()
     << r1.area();
```



Statisk bindning

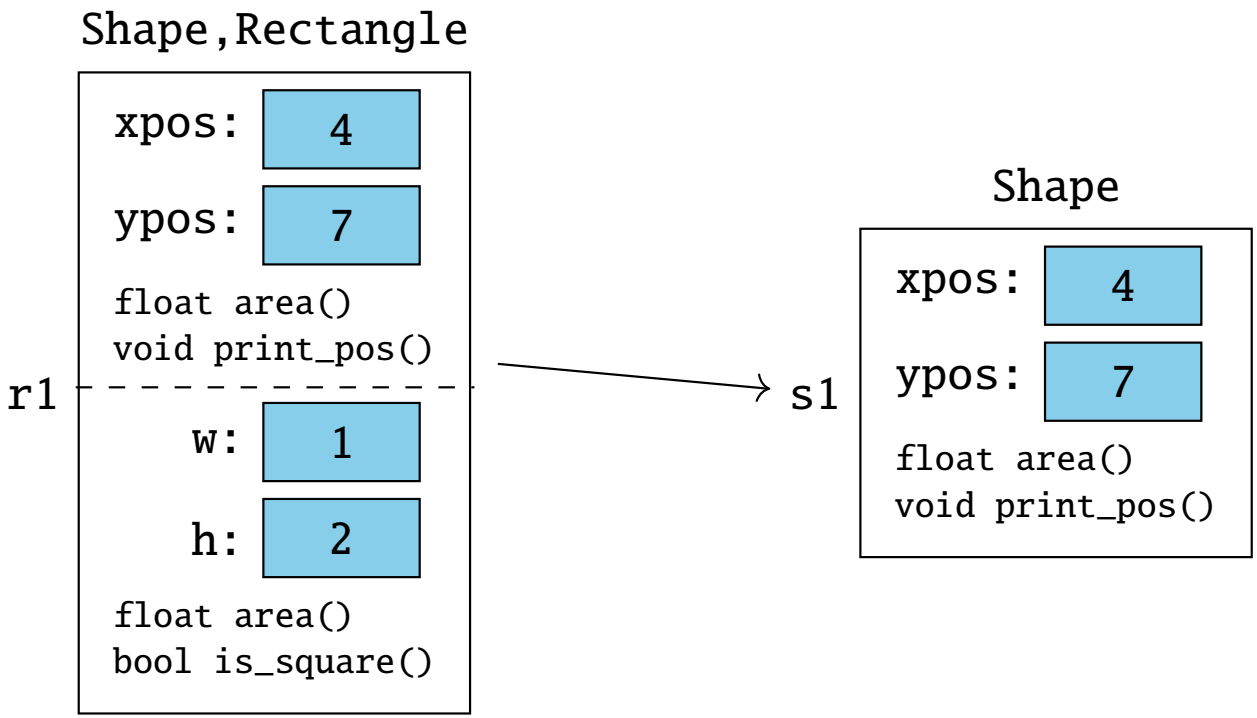
Hur vet kompilatorn vilken funktion som ska anropas?

```
Rectangle r1 {4, 7, 1, 2};

Shape& s1 {r1};

cout << s1.is_square()  error
     << s1.area()
     << r1.is_square()
     << r1.print_pos()
     << r1.area();
```



Slicing

```
Rectangle r1 {4, 7, 1, 2};

Shape s1 {r1}; // Kopia!
```



1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 UML
6 Typkonvertering

Härledda klasser

Vilken funktion kommer anropas (med statisk bindning)?

```
vector<Shape*> v {};

v.push_back(new Shape{...});
v.push_back(new Rectangle{...});
v.push_back(new Circle{...});

for (Shape* s : v)
{
  cout << s->area() << endl;
}
```

Polymorfi

Vi vill att varje härledd klass ska ha sin egen specialiserade implementation av en funktion.

Vi vill att den härledda klassens implementation anropas automatiskt istället för basklassens.

Dynamisk bindning

```
class Shape
{
public:
  ...

  virtual float area() const
  {

  }
};
```

```
class Rectangle : public Shape
{
public:
  ...

  float area() const override
  {
    return x * y;
  }
};
```

Härledda klasser

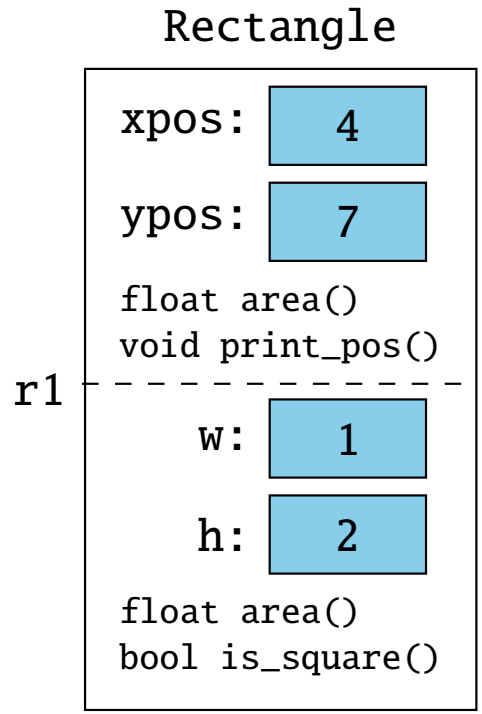
Vilken funktion kommer anropas (nu med dynamisk bindning)?

```
vector<Shape*> v {};

v.push_back(new Shape{...});
v.push_back(new Rectangle{...});
v.push_back(new Circle{...});

for (Shape* s : v)
{
  cout << s->area() << endl;
}
```

Virtuell destruktor

```
Shape*     s {new Rectangle{...}};
Rectangle* e {new Rectangle{...}};

delete s;
delete e;
```

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



Abstrakt klass

Problem:

- Ibland har en basklass ingen rimlig implementation av en funktion.

```
class Shape
{
public:
  Shape(int x, int y);

  virtual double get_area() const
  {
    // ???
  }

private:
  int x, y;
};
```

Abstrakt klass

Lösning:

- Sätt implementationen i basklassen till noll (pure virtual).
- En härledd klass måste implementera funktionen.
- Klassen är nu abstrakt. Det går inte att skapa instanser av en abstrakt klass.

```
class Shape
{
public:
  Shape(int x, int y);

  virtual double get_area() const = 0;



private:
  int x, y;
};
```

Interface

- En abstrakt klass med endast “pure virtual”-funktioner.
- Ett interface är som att ärva en designspecifikation, eftersom härledda klasser måste implementera alla “pure virtual”-funktioner.
- Samma princip som en abstrakt datatyp.

```
class Stack_Interface
{
public:
  virtual int top() const = 0;
  virtual void pop() = 0;
  virtual void push(int i) = 0;
  virtual int size() const = 0;
  virtual bool is_empty() const = 0;
};
```

using och delete

Det går att välja vilka medlemsfunktioner från basklassen som ska finnas i den härledda klassen.

- using - använd en implementation från basklassen.
- delete - ta bort en implementation.

```
class Point : public Shape
{
public:
  using Shape::Shape;

  float area() const = delete;
};
```

Exempel

- `std::ostream` är i själva verket basklassen till
  - `std::ostringstream`
  - `std::ofstream`
- `std::istream` är i själva verket basklassen till
  - `std::istringstream`
  - `std::ifstream`
- Möjliggör att återanvända samma kod för olika typer av strömmar.

Läs mer på https://en.cppreference.com/w/cpp/io

1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 UML
6 Typkonvertering

LINKÖPING UNIVERSITY

Synlighet

I en klass kan vi sätta synlighet på medlemmarna.

- public - tillgänglig utanför klassen
- private - endast tillgänglig inom klassen.

Hur blir det med arv?

Ibland vill vi att härledda klasser ska komma åt medlemmar, men ingen utanför klasshierarkin ska komma åt dem.

Synlighet

I en klass kan vi sätta synlighet på medlemmarna.

- public - tillgänglig utanför klassen
- private - endast tillgänglig inom klassen.
- protected - som private men tillgänglig från härledda klasser.

Synlighet

```
class Person
{
public:

  Person(string const& n,
         string const& p);
  string get_phone() const;
  void print_info(ostream& os) const;

protected:
  string name;
  string phone;

};
```

```
class Employee : public Person
{
public:

  Employee(string const& n,
           string const& p,
           string const& w,
           int s);

  string get_phone() const
  {
    return phone + work_phone;
  }

private:
  string work_phone;
  int salary;

};
```

Synlighet

Varför behövs `public` vid arv?

```
class Employee : public Person
```

Vad händer om vi skriver något annat?

- `public` - medlemmar i basklassen behåller deklarerat skydd i härledd klass.
- `protected` - publika medlemmar blir skyddade i härledd klass.
- `private` - alla medlemmar i basklassen blir privata i härledd klass.

Om inget deklareras väljs `private`.

1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 UML
6 Typkonvertering

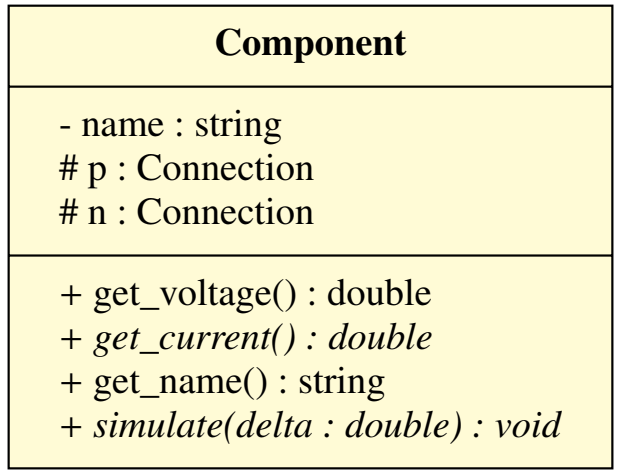
LINKÖPING UNIVERSITY

UML-diagram

- Visuell representation av projektets design
- Oberoende av programmeringsspråk
- Standardiserad representation
- Innehåller:
  - Klasser med alla medlemmar
  - Relationer mellan klasser

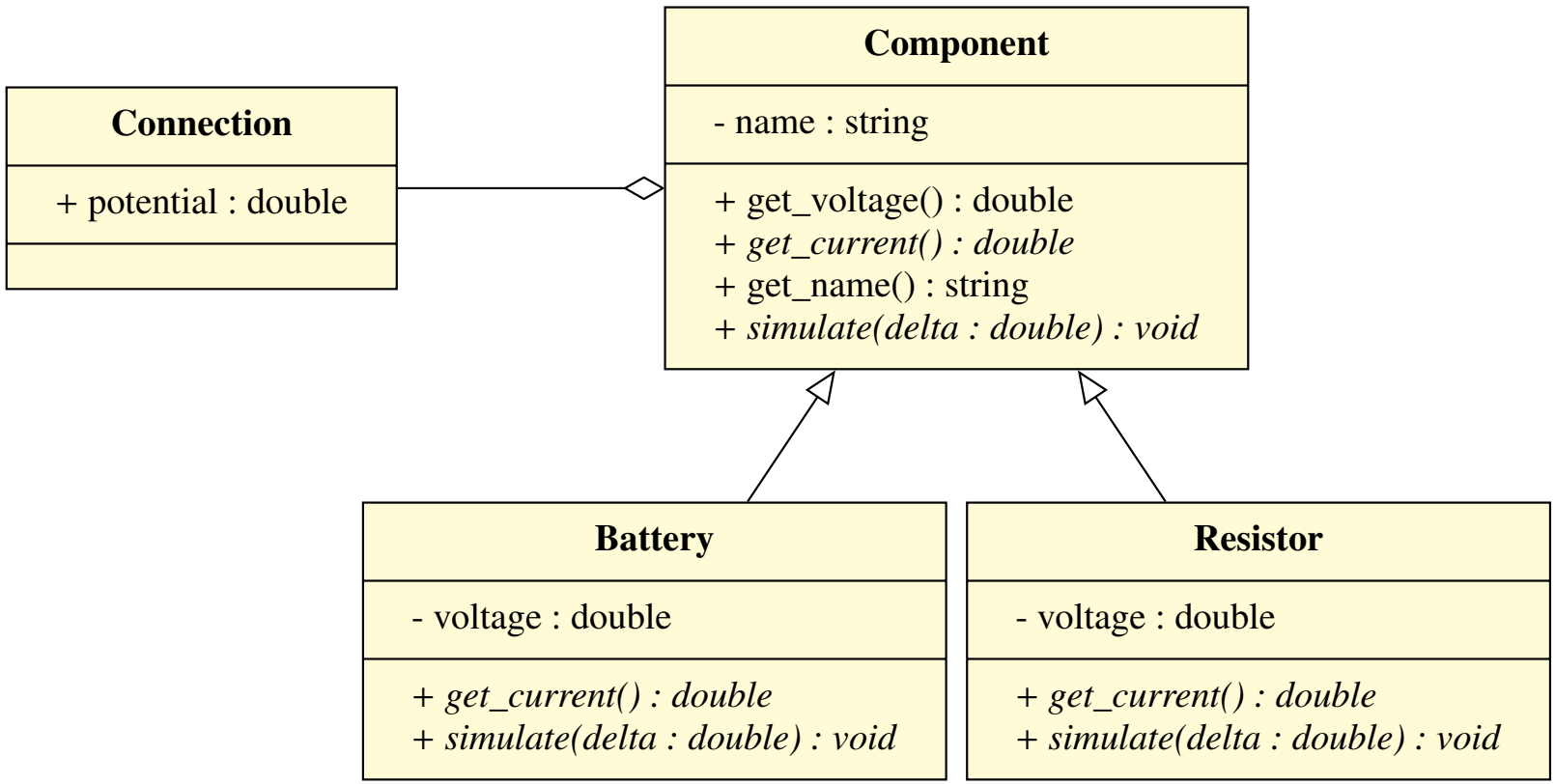
UML-diagram

Hur representeras en klass?

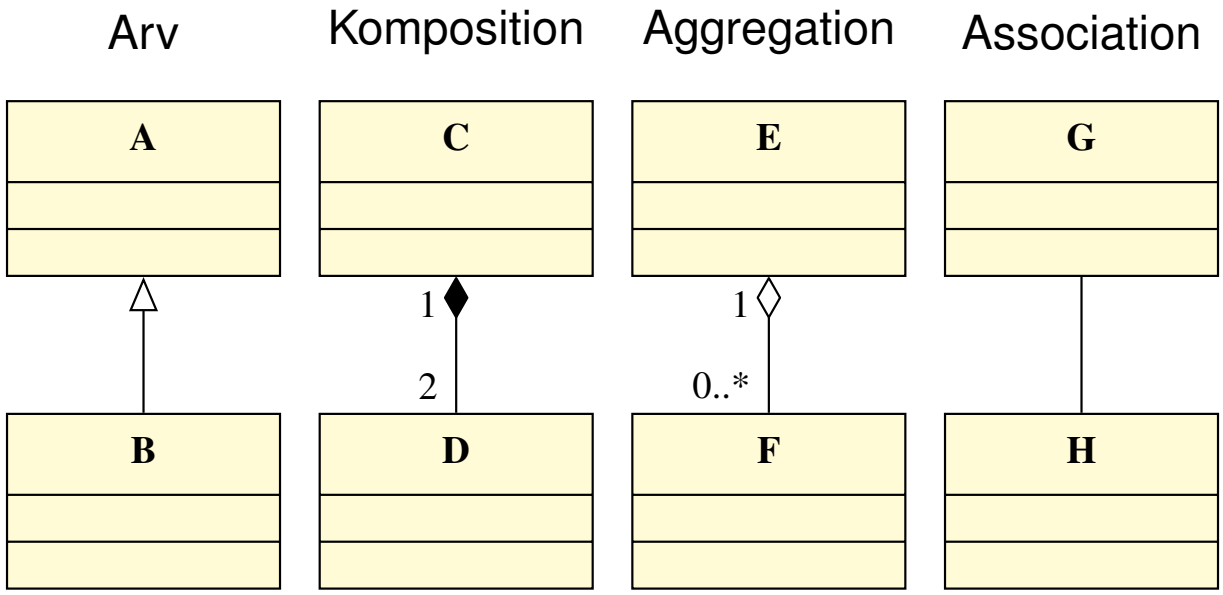


- + public
- – private
- # protected

UML-diagram


Component
- name : string
+ get_voltage() : double
+ get_current() : double
+ get_name() : string
+ simulate(delta : double) : void
Connection
+ potential : double
Battery
- voltage : double
+ get_current() : double
+ simulate(delta : double) : void
Resistor
- voltage : double
+ get_current() : double
+ simulate(delta : double) : void

UML-diagram



“B är en A”
“C består av två D”
“E har av noll eller flera F”
“G känner till H”

UML-diagram i kod

```
class Component
{
public:
  Component() = default;

private:
  vector<Connection*> connections; // Aggregation
  std::string name;                // Komposition
};

class Battery : public Component   // Arv
{
  //...
};
```

1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 UML
6 Typkonvertering

LINKÖPING UNIVERSITY

Explicit typkonvertering i C++

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`
- (C-style cast)

Statisk typkonvertering

```
int main()
{
  char c {'i'};
  int i { static_cast<int>(c) };

  float f { 3.14 };
  double d { static_cast<double>(f) };
}
```

Dynamisk typkontroll

När du behöver komma åt en medlemsfunktion som enbart finns i en viss härledd klass.

Funktionen är olämplig att konvertera till en virtuell funktion i basklassen.

Vi kan kontrollera om en basklasspekare i själva verket pekar på ett objekt av en härledd typ.

```
vector<Shape*> v {
  new Rect   {1,2,3,4},
  new Circle {1,2,5},
  new Rect   {1,1,2,2}
};

for (Shape* s : v)
{
  Rect* r { dynamic_cast<Rect*>(s) };
  if (r != nullptr)
  {
    cout << r->is_square() << endl;
  }
}
```

C-style cast

```
float f {};
int i { (int) f };
```

Använd ej!!!!!

Använd inte C-style casts!

Typkonverterande konstruktorer

En konstruktor som endast tar ett argument kan användas för typkovertering.

```
class Time
{
public:
  Time(std::string const& str);
};

void foo(Time const& t)
{
  std::cout << "En typkonvertering hände!";
}

int main()
{
  std::string str {};
  foo(str);
}
```

Typkonverterande konstruktorer

En konstruktor som endast tar ett argument kan användas för typkovertering.

```
class Time
{
public:
  explicit Time(std::string const& str);
};

void foo(Time const& t)
{
  std::cout << "En typkonvertering hände!";
}

int main()
{
  std::string str {};
  foo(str); // kompilerar ej!
}
```

Typkonverterande operator

```
class Time
{
public:
  operator int()
  {
    return 5;
  }
};

void bar(int i)
{
  std::cout << "En annan typkonvertering hände!";
}

int main()
{
  Time t {};
  bar(t);
}
```

www.ida.liu.se/~TDIU20/