

Lektion 4 – Polymorfism och dynamisk bindning

Objektorienterad programmering i C++

Begreppet polymorfism och hur det implementeras i C++ med arv, virtuella funktioner och dynamisk bindning.

Lektion 5 – Polymorfism och dynamisk bindning

Krav på OOP-språk

OOP, Objektorienterad programmering, bygger på tre hörnstenar som måste tillhandahållas av varje språk som gör anspråk på att vara ett riktigt OOP-språk.

1. Inkapsling (encapsulation)

Separation av interface och implementation uppdelning av klasser i klassdefinition och implementationsfil.

2. Arv (inheritance)

Skapa nya klasser genom tillägg/förändringar gentemot befintliga klasser. En ny **deriverad klass** skapas från en eller flera **basklasser**.

3. Polymorfism (polymorphism)

Polymorfism betyder bokstavligen 'flera former'. Detta avspeglas i C++ i flera mekanismer.

- *Överlagring (overloading)* av funktioner. Flera funktioner kan ha samma namn så länge de skiljer sig åt i signaturen, dvs antalet parametrar och parametrarnas typer. Funktionsöverlagring hanteras av kompilatorn under '*compile-time*'. Finns inte i alla OOP-språk.

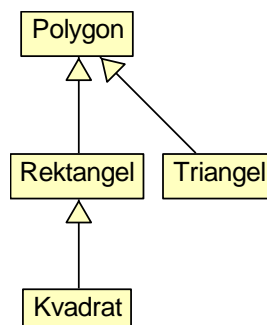
- *Dynamisk bindning (late binding)*

I en arvshierarki kan funktioner som ärvts från någon basklass ges en ny implementation i en deriverad klass, en sk *override*. Vilken implementation som ska bindas till ett visst funktionsanrop avgörs först under exekveringen dvs under *run-time*. I C++ implementeras detta mha av *virtuella funktioner* och fungerar i samband med *pekare eller referenser* till objekt.

Dynamisk bindning

Bindning av pekare / referens till rätt implementation av en virtuell funktion görs vid runtime dvs. under exekveringen .

Ex. Klassen Polygon kan vara basklass för mer specialiserade klasser:



Alla dessa klasser kan innehålla operationer för beräkning av t.ex. area, omkrets och tyngdpunkt.

Area, omkrets och tyngdpunkt måste implementeras olika i de olika klasserna. Funktioner med samma namn och signatur kan omdefinieras (*override*) om de görs virtuella mha nyckelordet *virtual*. Dynamisk bindning i C++ aktiveras alltså genom nyckelordet *virtual*.

```
class Polygon {
public:
    Polygon() { }; // tom implementation
    virtual int area() { };
    virtual int omkrets() { };
    virtual int tp() { };
};

class Triangel : public Polygon {
public:
    Triangel(Point, Point, Point);
    virtual int area(); // Herons formel
    virtual int omkrets();
    virtual int tp();
private:
    Point p1, p2, p3;
};

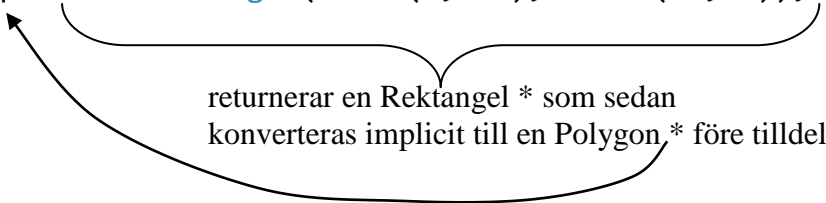
class Rektangel : public Polygon
private:
    Point upperLeft, lowerRight

public:
    Rektangel(Point, Point);
    virtual int area();
    .
    .
};
```

Area-funktionen implementeras olika i de två klasserna. Deriverade klassers implementation ersätter basklassens implementation, man talar om omdefiniering eller *override* av funktioner. Obs, *override* är *inte* samma sak som *overloading*.

En pekare eller referens till en deriverad klass konverteras *implicit* till pekare/referens av basklassens typ:

```
Polygon *pp = new Rektangel(Point(1, 10), Point(10, 1));
```



returnerar en Rektangel * som sedan konverteras implicit till en Polygon * före tilldelning

Anropet `pp -> area();` binds till `Rektangel::Area` trots att `pp` är en pekare av typen `Polygon*` eftersom samtliga följande villkor är uppfyllda:

1. `Rektangel` är deriverad från `Polygon`
2. `area` är en virtuell funktion i `Polygon`
3. `area` är omdefinierad i klassen `Rektangel`
4. `pp` pekar på en `Rektangel` och inte en `Polygon` !!

Om något av villkoren 2,3 eller 4 inte uppfylls binds i stället anropet till `Polygon:Area` (statisk bindning). Vi fortsätter föregående exempel:

```
delete pp; // Deallokera Rektangel-objektet
pp = new Triangel(Point(1, 1), Point(3, 10), Point(10, 5));
pp->area();
```

Anropet `pp -> area();` binds nu till `Triangel::area()` eftersom villkoren ovan är uppfyllda:

- `Triangel` är deriverad från `Polygon`
- `area` är en virtuell funktion i `Polygon`
- `area` är omdefinierad i klassen `Triangel`
- `pp` pekar på en `Triangel`

Bindningen görs vid exekveringen, alltså en *dynamisk bindning*. Exemplet visade dynamisk bindning av virtuella funktioner genom pekare. Även referenser kan användas enligt följande exempel:

```
class A {
public:
    virtual void id() { cout << "Base" << endl; }
};

class B : public A {
public:
    virtual void id() { cout << "B" << endl; }
};

class C : public B { };

class D : public C {
public:
    virtual void id() { cout << "D" << endl; }
};

void showId(A &ref) {
    ref.id();
}

int main() {
    A aObj; B bObj; C cObj; D dObj;
    showId(aObj);      // Base
    showId(bObj);      // B
    showId(cObj);      // B
    showId(dObj);      // D
}
```

Utskrifterna blir

Base

B

B

D

En första slutsats vi kan dra från exemplet är att om en deriverad klass inte själv omdefinierar en ärvd virtuell funktion så används den närmast föregående implementationen uppåt i arvshierarkin (`showId(cObj); // B`).

Vilka konverteringar görs och var finns den dynamiska bindningen i exemplet?

T.ex. vid anropet `showId(cObj);` kommer en `C&` (referensen till `cObj`) att konverteras till en `A&`, en basklass-referens. Värdet tilldelas sedan den formella parametern `ref`. Den dynamiska bindningen verkar sedan vid anropet `ref.id();`. Runtime-systemet vet att `ref` refererar till ett C-objekt och väljer den version av `id()` som gäller för klassen C. Med dynamisk bindning är det alltså *typen (klassen) för objektet som pekars ut eller refereras*, som avgör vilken funktion som ska bindas.

Statisk vs. Dynamisk bindning

Statisk bindning är bindning av pekare/referens till funktioner som sker redan vid kompileringen (compile time).

Statisk bindning...

- görs alltid för icke-virtuella medlemsfunktioner och fristående funktioner
- görs alltid med anrop direkt till ett objekt via 'member selection operator'. t.ex. `myClass.myfunc();`
- görs alltid vid kvalificerade anrop av typen `MyClass::func();`

Vid statisk bindning är det *pekarens/referensens typ* som avgör vilken funktion som ska bindas.

Ett exempel, vi utgår från hierarkin

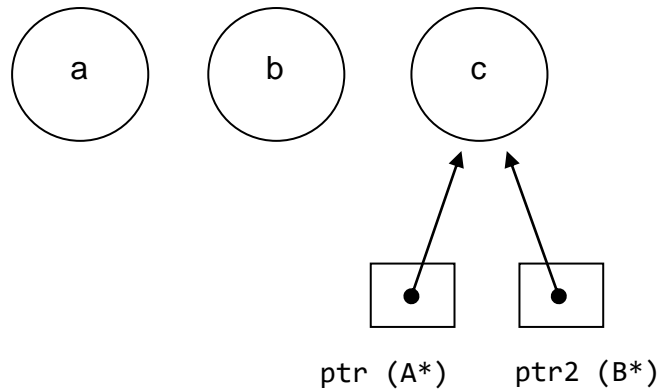
```
class A {
public:
    void func();
};

class B : public A {           // Innehåller A::func och
func                           // B::func
public:
    void func();
};

class C : public B {          // Innehåller A::func,
B::func och func
public:
    void func();
};
```

Vi gör några definitioner:

```
A a;  
B b;  
C c;  
A *ptr = &c;  
B *ptr2 = &c;
```



Anropet `ptr -> func()`; binds statiskt_till `A::func()` eftersom

- `func` inte är virtuell, och
- `ptr` är av typen `A*`

Anropet `ptr2 -> func()`; binds statiskt_till `B::func()` eftersom

- `func` inte är virtuell, och
- `ptr2` är av typen `B*`

Viktig slutsats:

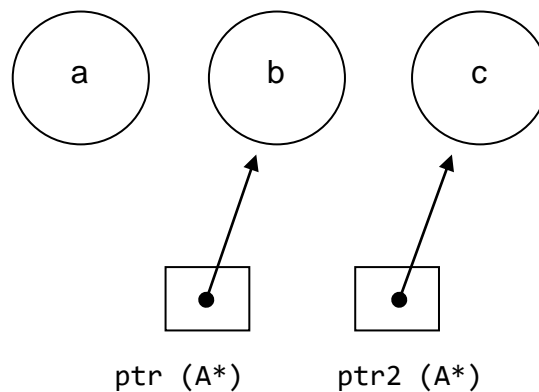
- Vid *statisk* bindning är det pekarens/referensens typ som avgör vilken funktion som binds.

Vi gör nu funktionerna virtuella och ser hur situationen förändras.

```
class A {  
public:  
    virtual void func();  
};  
  
class B : public A { // Innehåller A::func och func  
public:  
    virtual void func();  
};  
  
class C : public B { // Innehåller A::func, B::func och func  
public:  
    virtual void func();  
};
```

Några definitioner:

```
A a;  
B b;  
C c;  
A *ptr = &b;  
A *ptr2 = &c;
```



Anropet `ptr -> func()`; binds till `B::func()` eftersom

- `func` är virtuell, och
- `ptr` pekar på ett B-objekt

Anropet `ptr2->func()`; binds till `C::func` eftersom

- `func` är virtuell, och
- `ptr2` pekar på ett C-objekt

Viktig slutsats:

- Vid *dynamisk* bindning är det typen för det utpekade/refererade objektet som avgör vilken funktion som ska bindas till ett anrop.

Ett större exempel:

Geometriskta figurer med dynamisk bindning (exGeometry)

För att beskriva två-dimensionella geometriska figurer behöver vi en klass för punkter i planet. Klassen `Point`:

Point
-xc: float -yc: float
+Point(newx:float, newy:float) +x(): float +x(newx:int): float +y(): float +y(newy:int): float +operator+(p:const Point&): Point +operator+=(p:const Point&): void +printOn(strm:ostream&): void

Utnyttjande av dynamisk bindning kräver en gemensam basklass för alla geometriska figurer.

- Basklassen *Shape* får innehålla attribut och operationer som ska vara gemensamma för alla deriverade klasser.
- Operationer som ska implementeras olika i deriverade klasser ska finnas deklarerade som virtuella funktioner i *Shape*.
- Klassen *Shape* ska inte själv kunna implementeras utan endast utgöra en basklass. Klassen ska göras till en *abstrakt* basklass. Detta innebär att minst en av medlemsfunktionerna ska sakna implementation. En sådan funktion kallas rent virtuell (pure virtual) och markeras kodmässigt med `= 0` i klassdefinitionen.

<i>Shape</i>
-org: Point
#setOrg(no:Point&): void
+Shape()
+Shape(o:const Point)
+~Shape()
+origin(): Point
+move(d:const Point): void
+draw(): void=0

Datamedlemmar:

- alla Shapes har en utgångspunkt i planet – Point org.
- alla Shapes ska kunna tala om sin utgångspunkt – Point origin(), ej virtuell, behöver inte omdefinieras i definierade klasser.
- alla Shapes ska kunna flyttas – virtual void move(), kan omdefinieras
- alla Shapes ska kunna beskriva sig själva – virtual void draw() = 0.

Beteckningen = 0; efter funktionshuvudet i klassdefinitionen anger alltså att den virtuella funktionen saknar implementation i klassen och därför måste implementeras i deriverade klasser för att de ska kunna instansieras. En sådan funktion kallas *rent virtuell* (pure virtual) . Om en klass har en eller flera rent virtuella funktioner så är klassen *abstrakt* och kan inte instansieras.

Avsikten med en sådan klass är att den ska utgöra en basklass och definiera ett gemensamt interface för ett antal deriverade klasser. En basklass kan även ha datamedlemmar och implementationer. En vanlig missuppfattning är att en 'nollad' medlemsfunktion inte kan ha en implementation i den klassen men så är inte fallet. En 'nollad' funktion kan ha en implementation som kan användas i deriverade klasser (som då *kan*, men inte *behöver*, omdefiniera denna funktion). Alltså: en klass med en eller flera 'nollade' virtuella funktioner kan inte instansieras utan bara utgöra en basklass för deriverade klasser. De virtuella funktioner som saknar implementation i basklassen måste implementeras i deriverade klasser för att dessa ska bli *konkreta* klasser som kan instansieras.

```
class Shape {
private:
    Point org;        // origin
protected:
    void setOrg(Point &no) { org=no;}
public:
    Shape() {}
    Shape(const Point o) : org(o) { } //origin
    virtual ~Shape() {}
    Point origin() const { return org;}
    virtual void move(const Point d) { org += d; }

    // draw is pure virtual ==> Shape abstract class
    virtual void draw() const =0; // pure virtual
};
```

Destruktorn i en basklass ska alltid vara virtuell för att samtliga destruktorer ska exekveras när delete anropas med en basklass-pekare (eller referens) som pekar på ett objekt av en deriverad klass.. Detta garanterar att deriverade klassers destruktorer kommer att exekveras. Om destruktorn i basklassen inte görs virtuell och ett objekt av deriverad typ allokeras med new så kommer endast *basklassens* destruktör att exekveras vid delete.

En linje 'är en sorts figur', vi deriverar Line från Shape



```
class Line: public Shape {
private:
    Point endp;          // end point
public:
    Line(const Point a, const Point b);
    virtual ~Line() {}
    virtual void move(const Point d);
    virtual void draw() const;
};
```

- `Point endp;` är linjens slutpunkt. Var finns startpunkten?
Jo, i arvet från `Shape`!
- Klassen omdefinierar virtuella `move`
- Klassen implementerar rent virtuella `draw`

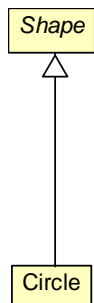
Strategi för `move`:

1. Flytta startpunkten genom `Shape::move`
2. Flytta slutpunkten

```
void Line::move(const Point d) {
    Shape::move(d);
    endp += d;
}

void Line::draw() const {
    cout << "Line from " << origin()
    << " to " << endp << endl;
}
```

En cirkel är också en sorts figur:

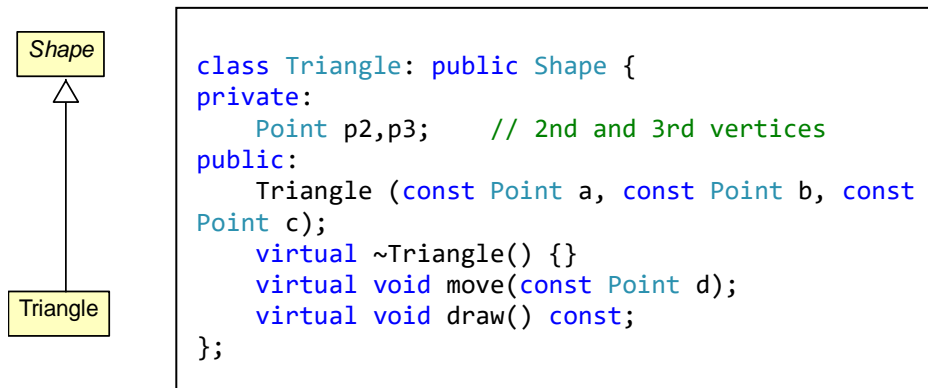


```
class Circle: public Shape {
private:
    int rad;          // radius of circle
public:
    Circle(const Point c, int r) : Shape(c)
    { rad = r; }
    virtual ~Circle() {}
    virtual void draw() const;
};
```

```
void Circle::draw() const {
    cout << "Circle with center " << origin()
    << " and radius " << rad << endl;
}
```

- Cirkeln beskrivs av medelpunkten samt radien.
- Implementerar draw.
- Omdefinierar *inte* virtuella move. Varför?

Även en triangel är en sorts figur:



- Två Point-objekt som hörn (var finns det tredje?).
- Implementerar rent virtuella draw
- Omdefinierar virtuella move.

```
void Triangle::move(const Point d) {
    Shape::move(d);    // move origin and tp
    p2 += d;
    p3 += d;
}

void Triangle::draw() const {
    cout << "Triangle with corners " << origin()
         << ", " << p2 << " and " << p3 << endl;
}
```

Klassen Picture

En bild kan innehålla ett antal figurer → En Picture kan betraktas som en kontainer för ett antal geometriska figurer, i vårt fall ett antal *Shape-objekt*. Med Shape-objekt menas objekt av någon klass som är deriverad från Shape. Substitutionsprincipen säger ju att alla sådana klasser är "ett slags Shape" och följaktligen kan ersätta Shape.

Attribut i Picture blir

```
vector<Shape*> shapes;
```

→ en Picture innehåller pekare till diverse Shape-objekt som kan vara linjer, trianglar osv.

Picture
-shapes: vector<Shape*> -n: int
+Picture() +~Picture() +add(:Shape&): void +draw(): void +move(d:const Point): void

```
class Picture {
private:
    vector<Shape*> shapes; // pointers to shapes
    int n;                // number of shapes in
this Picture
public:
    Picture( )
    :n(0) { } // constructor

    virtual ~Picture(){} // destructor
    void add(Shape&);     // add Shape to Picture
    void draw() const;    // draw picture;
    void move(const Point d);
};
```

Att 'rita ut' en Picture innebär att alla ingående Shape-objekt får rita ut sig själva. Alla har sin egen implementation av draw som binds dynamiskt.

```
void Picture::draw() const // draw a Picture
{
    for (int i=0; i<shapes.size(); i++)
        shapes[i]->draw(); // Let every shape draw itself
}
```

Dynamisk bindning

Att flytta en Picture innebär att alla ingående objekt får flytta sig själva. Alla har sin egen implementation av move som binds dynamiskt.

```
void Picture::move(const Point d)
{
    for (int i=0; i<n; i++)
        shapes[i]->move(d); // Let every shape move itself
}
```

Dynamisk bindning

Till en Picture kan vi alltså addera ett godtyckligt antal Shape-objekt och utnyttja polymorfism för att rita ut dem och flytta dem. Exempel på klient kod följer här nedan. Fullständig kod finns i exGeometry.

```

Line li1(Point(10,20),Point(100,200)); // create a Line
Circle c1(Point(100,200),100);        // create a Circle
li1.draw();                            // draw the line
c1.draw();                             // draw the circle

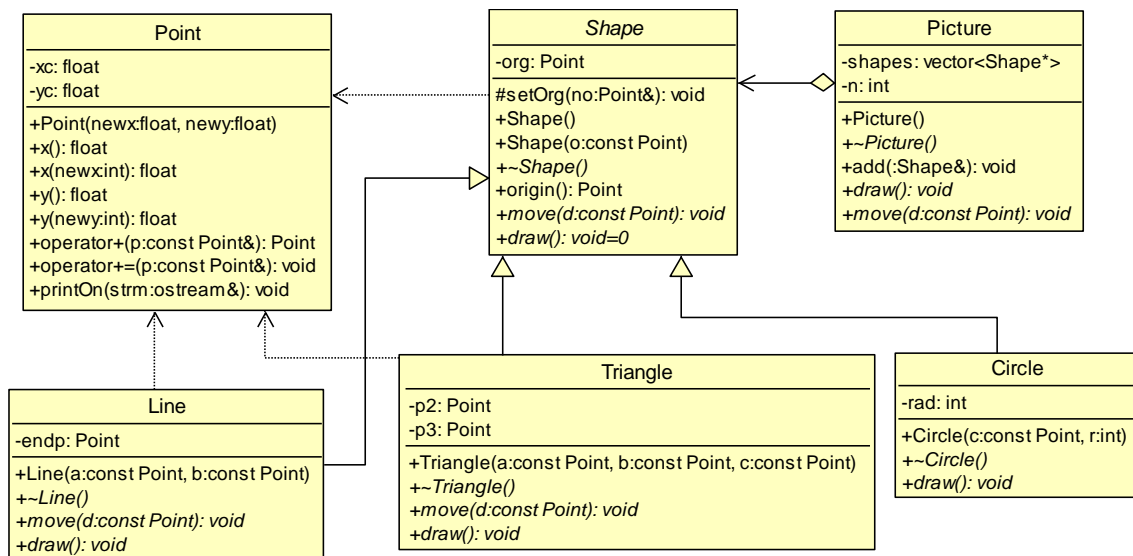
Picture pic1; // create a Picture
pic1.add(li1); // add the line to the picture
pic1.add(c1);  // ..and the circle
cout << "=>Picture 1:" << endl;
pic1.draw(); cout << endl; // draw the picture
pic1.move(Point(-10,-5)); // move the picture
cout << "=>Picture 1 moved (-10,-5):" << endl;
pic1.draw(); cout << endl; // draw the picture

Triangle t1(Point(-10,0),Point(20,30),Point(40,60));
Circle c2(Point(50,75),80);

// define another picture
Picture pic2;
pic2.add(t1); // add t1 to the picture
pic2.add(c2); // ...and c2
cout << "=>Picture 2" << endl;
pic2.draw(); cout << endl; // draw
cin.get();
pic2.move(Point(20,30)); // move the picture
cout << "=>Picture 2 moved (20,30):" << endl;
pic2.draw(); cout << endl; // draw the picture
...

```

Klassdiagram



"Down-cast" i arvshierarkier

Ett problem som kan uppstå i en arvshierarki är att man ibland vill införa nya funktioner i deriverade klasser. T.ex. kan man prata om area när det gäller en triangel eller en cirkel men inte när det gäller en punkt eller en linje. Om vi inför funktionen `getArea()` i klassen `Circle` och försöker kompilera koden

```
Shape *shapePtr = new Circle(Point(10, 30), 50);  
double area = shapePtr->getArea();
```

så får vi ett kompileringsfel av typen "Error: `getArea` is not a member of `Shape`".

Med en `Shape`-pekare kan vi bara anropa funktioner som finns deklarerade i `Shape` men samtidigt måste vi använda `Shape`-pekare för att den dynamiska bindningen ska fungera i hela hierarkin. Detta kan tyckas vara motsägelsefullt men är en konsekvens att principerna för objektorienterad programmering. Man kan dock hantera detta problem på olika sätt.

Metod 1

Deklarera alla funktioner redan i basklassen och låt dessa ha en "tom" implementation i de klasser där de inte är relevanta. Detta kan kombineras med funktioner för att testa om ett anrop är meningsfullt eller inte.

I `Shape`-hierarkin:

```
class Shape {  
...  
public:  
...  
  
    virtual double getArea() = 0;  
    virtual bool hasArea() = 0;  
  
};  
  
class Line: public Shape {  
...  
public:  
...  
    virtual double getArea() { return 0.0; }  
    virtual bool hasArea() { return false; }  
};  
  
class Circle: public Shape {  
...  
public:  
...  
    virtual double getArea() { return PI*rad*rad; }  
    virtual bool hasArea() { return true; }  
  
};
```

Klientkoden

```
Shape *shapePtr[2];
shapePtr[0]= new Circle(Point(10, 30), 10);
shapePtr[1] = new Line(Point(5, 5), Point(20, 20));
for (int i = 0; i<2;++i)
    if (shapePtr[i]->hasArea())
        cout << shapePtr[i]->getArea() << endl;
```

ger utskriften 314,159 dvs. arean av en cirkel med arean 10.

Metoden bygger alltså på att försöka införa alla virtuella funktioner redan i basklassen och sedan testa om man kan anropa dem.

Metod 2

Metod 2 bygger på att man utnyttjar operatoren `dynamic_cast` för att under runtime undersöka om en basklasspekare pekar på ett objekt av en viss deriverad klass. Då är det ofarligt att göra en typkonvertering neråt i hierarkin för att få en pekare till exakt den klassen. Detta är ett s.k. ”down-cast” som normalt *inte* ska göras eftersom det inte finns garantier för att alla funktioner som finns i den deriverade klassen också finns i basklassen. Substitutionsprincipen säger att motsatsen (”upcast”) alltid ska fungera, dvs. att en deriverad klass alltid ska kunna användas istället för en klass högre upp i hierarkin. C++-kompilatorn gör också sådana ofarliga konverteringar implicit. Det sker t.ex. i definitionen

```
Shape* shapePtr = new Circle(Point(10, 30), 10);
```

där Circle-pekaren som returneras från new konverteras till en Shape-pekare före tilldelningen.

Antag att basklassen Shape inte innehåller någon av funktionerna `getArea()` eller `hasArea()` och att funktionen `getArea()` finns införd först i de deriverade klasserna Circle och Triangle. En Shape-pekare kan peka på vilket Shape-objekt som helst så innan vi anropar `getArea()` måste vi veta att den faktiskt pekar på ett objekt som har den funktionen definierad. Vi utnyttjar operatoren `dynamic_cast` för att testa detta.

Exempel:

Vi utgår från vår ursprungliga Shape-klass som inte deklarerar någon `getArea`-funktion. Däremot antar vi att `Circle::getArea()` finns definierad. Vi definierar två Shape-objekt:

```
Shape *shapePtr[2];
shapePtr[0]= new Circle(Point(10, 30), 10);
shapePtr[1] = new Line(Point(5, 5), Point(20, 20));
```

och skriver ut arean om någon av dessa pekare pekar på en Circle:

```
for (int i = 0; i<2; ++i) {  
    if (Circle *cptr = dynamic_cast<Circle*>(shapePtr[i]))  
        cout << cptr->getArea() << endl;  
    else  
        cout << "Conversion failed"<< endl;  
}
```

Utskriften blir
314.159
Conversion failed

Om konverteringen lyckas så returnerar `dynamic_cast` en pekare av den önskade typen och anropet kan göras genom den pekaren. Om konverteringen inte kan genomföras (ingen `Line::getArea` finns) så returneras en `nullptr`.

Operatören `dynamic_cast` kan även användas med referenser till objekt. Om den önskade konverteringen inte kan genomföras kastas ett exception av typen `bad_cast`. Mer om detta i lektionen som behandlar exceptions (undantag).