EDAF30 – Programmering i C++

9. Polymorfism och arv

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

Typomvandlingar (casting)
Implicita Typomvandlingar

Automatiska typomvandlingar

- Uttryck av typen $x \odot y$, för någon binär operator $\odot$
  Ex: `double + int ==> double`
  `float + long + char ==> float`
- Tilldelningar och initieringar: Värdet i högerledet konverteras till samma datatyp som i vänsterledet
- Konvertering av typen hos aktuella parametrar till typen för de formella parametrarna
- Villkor i `if`-satser, etc. ⇒ **`bool`**
- C-array ⇒ pekare (*array decay*)
- `0` ⇒ `nullptr` (tom pekare, i C++11, tidigare definierade man ofta konstanten `NULL`)

Typomvandlingar (casting)
Implicita Typomvandlingar

Automatiska typomvandlingar

- Uttryck av typen $x \odot y$, för någon binär operator $\odot$
  Ex: `double + int ==> double`
  `float + long + char ==> float`
- Tilldelningar och initieringar: Värdet i högerledet konverteras till samma datatyp som i vänsterledet
- Konvertering av typen hos aktuella parametrar till typen för de formella parametrarna
- Villkor i `if`-satser, etc. ⇒ **`bool`**
- C-array ⇒ pekare (*array decay*)
- `0` ⇒ `nullptr` (tom pekare, i C++11, tidigare definierade man ofta konstanten `NULL`)

Typomvandlingar (*casting*)
Explicita, namngivna typomvandlingar (C++-11)

- **`static_cast`**`<new_type> (expr)`
  - omvandlar mellan kompatibla typer (*kollar inte talområden*)
- **`reinterpret_cast`**`<new_type> (expr)`
  - inget skyddsnät, samma som C-stil
- **`const_cast`**`<new_type> (expr)` - lägger till eller tar bort **`const`**
- **`dynamic_cast`**`<new_type> (expr)` - används för pekare till klasser. Gör typkontroll vid *run-time*, som i Java.

Exempel

```
char c;              // 1 byte
int *p = (int*) &c;  // pekar på int: 4 bytes

*p = 5; //fel vid exekvering, stack-korruption

int *q = static_cast<int*> (&c); // kompileringsfel
```


Typomvandlingar (*casting*)
Explicita typomvandlingar, C-stil

Syntax i C och i C++, som i Java

`(typnamn)uttryck` , t ex `(float) 10`

- Stor risk att göra fel - använd namgivna typomvandligar
  - blir tydligare i koden, t ex **`const_cast`** kan bara ändra **`const`**
  - lätt att söka efter: casts är bland det första man tittar på när man letar fel
- Varning i GCC: `-Wold-style-casts`
- Vanlig i äldre kod

Alternativ syntax i C++

```
typnamn(uttryck)
```

`typnamn` måste vara *ett ord*,
d v s **`int`**`*(...)` eller **`unsigned long`**`(...)` är inte OK.

Typomvandlingar (*casting*)
Varnande exempel

```
struct Point{
    int x;
    int y;
};

struct Point3d :public Point{
     int z;
};
```

Point:

x:
y:

Point3d:

x:
y:
z:

Typomvandlingar (*casting*)
Varnande exempel

```
struct Point{
    int x;
    int y;
};

Point ps[3];

struct Point3d{
     int x;
     int y;
     int z;
};

Point3d* foo = (Point3d*) ps;
```

ps:

x:
y:
x:
y:
x:
y:

$foo[0]$

$foo[1]$

Med *named casts* måste man använda **reinterpret_cast**<Point3d*>

med **static_cast** fås felet

`invalid static_cast from type 'Point[3] to type 'Point3d*'`

specialfall: **void**-pekare

En **void*** kan peka på vad som helst (objekt av godtycklig typ.)

- I C omvandlas **void*** implicit till/från varje pekartyp.
- I C++ omvandlas T* implicit till **void***. Åt andra hållet krävs en explicit *type cast*.

Konstruktorer vid arv
Regler för basklassens konstruktor

- Basklassens default-konstruktor anropas implicit
  - om den finns!
- Argument till basklassens konstruktor
  - ges i *initierar-listan* i subklassens konstruktors .
  - *basklassens namn* måste används.
    (super() som i Java finns inte p g a multipelt arv.)

Konstruktorer vid arv

Initieringsordning i en konstruktor (för härledd klass)

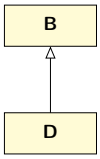
1 *Basklassen initieras*: Basklassens konstruktor anropas
2 *Subklassen initieras*: Datamedlemmar (i subklassen) initieras
3 Funktionskroppen i subklassens konstruktor exekveras

Explicit anrop av basklassens konstruktor i initieringslistan

```
D::D(param...) :B(param...), ... {...}
```

Notera:

- Konstruktorer ärvs inte
- *Anropa inte virtuella funktioner från en konstruktor.*: I basklassen B är **this** av typen B*.



Konstruktorer vid arv

Konstruktorer ärvs inte

```
class Base{
public:
    Base(int i) :x{i} {}
    virtual void print() {cout << "Base: " << x << endl;}
private:
    int x;
};

class Derived :public Base {
};

void test_ctors()
{
    Derived b(5); //no matching function for call to
                  //Derived::Derived(int)
    Derived b2; //use of deleted function Derived::Derived()
}
```

Konstruktorer vid arv

`using`: gör basklassens konstruktor synlig (C++11)

```
class Base{
public:
    Base(int i) :x{i} {}
    virtual void print() {cout << "Base: " << x << endl;}
private:
    int x;
};

class Derived :public Base {
  using Base::Base;
};

void test_ctors()
{
    Derived b2(5); // OK!
    Derived b; //use of deleted function Derived::Derived()
    b.print();
}
```


Konstruktorer vid arv

Nu med default-konstruktor

```
class Base{
public:
    Base(int i=0) :x{i} {}
    virtual void print() {cout << "Base: " << x << endl;}
private:
    int x;
};

class Derived :public Base {
  using Base::Base;
};

void test_ctors()
{
    Derived b;     // OK!
    b.print();
    Derived b2(5); // OK!
    b2.print();
}
```


Ärvda konstruktorer
regler

- **`using`** gör att alla superklassens konstruktorer ärvs, utom
  - de som döljs av subklassen (har samma parametrar)
  - default- copy- och move-konstruktorer ärvs inte
    ⇒ *om de inte definieras, syntetiseras de som vanligt*
- default-parametrar i superklassen ger flera ärvda konstruktorer

Kopiering och arv

- Kopieringskonstruktorn ska kopiera *hela objektet*
  - typiskt: anropa superklassens kopierings-konstruktor
- Samma sak gäller för **`operator=`**
- Skillnad mot destruktorn
  - En destruktor i en subklass ska bara avallokera det som allokerats i subklassen. Basklassens destruktor anropas implicit.
- En subklass kan inte kopieras om superklassen saknar (d v s är **`private`** eller **`=delete`**)
  - default-konstruktor,
  - copy-konstruktor,
  - copy-assignment operator eller
  - destruktor
- Basklasser bör definiera dessa **`=default`**

Destruktorer vid arv

Destruktorn görs i omvänd ordning:

Exekveringsordning i en destruktor

1. Funktionskroppen i subklassens destruktor exekveras
2. Subklassens medlemmar destrueras
3. Basklassens destruktor anropas

I basklassen ska destruktorn vara virtuell

Tillgänglighet

De olika nivåerna av tillgänglighet

```
class C {
public:
      // Medlemmar åtkomliga från godtyckliga funktioner
protected:
      // Medlemmar åtkomliga från medlemsfunktioner
      // i klassen eller härledda klasser
private:
      // Medlemmar åtkomliga endast från
      // klassens egna medlemsfunktioner
};
```

Tillgänglighet

Tillgänglighet vid arv

```
class D1 : public B { // Publikt arv
    // ...
};

class D2 : protected B { // Skyddat arv
    // ...
};

class D3 : private B { // Privat arv
    // ...
};
```


Tillgänglighet

Tillgänglighet vid arv

	Tillgänglighet i `B`	Tillgänglighet via `D`
Publikt arv	**public** **protected** **private**	**public** **protected** **private**
Skyddat arv	**public** **protected** **private**	**protected** **protected** **private**
Privat arv	**public** **protected** **private**	**private** **private** **private**

Tillgängligheten inuti `D` påverkas *inte* av typen av arv

Funktionsöverlagring och arv

Funktionsöverlagring fungerar ej som vanligt mellan olika nivåer i arvshierarkin

```
class C1 {
public:
    void f(int) {cout << "C1::f(int)\n";}
};

class C2 : public C1 {
public:
    void f(); {cout << "C2::f(void)\n";}
};

C1 a;
C2 b;
a.f(5); // Ok, anropar C1::f(int)
b.f();  // Ok, anropar C2::f(void)
b.f(2)  // Fel! C1::f är dold!
b.C1::f(10); // Ok
```


Funktionsöverlagring och arv
Gör namn i superklass synliga med **using**

Funktionsöverlagring mellan olika nivåer i arvshierarkin

```
class C1 {
public:
    void f(int); {cout << "C1::f(int)\n";}
};

class C2 : public C1 {
public:
    using C1::f;
    void f(); {cout << "C2::f(void)\n";}
};

//...
C1 a;
C2 b;
a.f(5); // Ok, anropar C1::f(int)
b.f();  // Ok, anropar C2::f(void)
b.f(2)  // Ok, anropar C1::f(int)
```


Arv och *scope*

- En subklass *scope* är kapslat (eng: *nested*) inuti superklassens
  - Namn i superklassen syns i subklasser
  - *om de inte döljs* av samma namn i subklassen
- Använd *scope-operatorn* `::` för att komma åt dolda namn
- Namnuppslagning sker vid kompilering
  - *Statisk typ* för en pekare eller referens styr vilka namn som syns (som i Java)
  - Virtuella funktioner måste ha samma parametertyper i subklasser

Arv utan virtuella funktioner

I C++ är medlemsfunktioner *inte virtuella om det inte anges*.
(Skillnad från Java)

- Man kan ärva från en klass och *dölja* dess funktioner.
- Man kan explicit anropa funktioner i superklassen.
- Kan användas för att "utöka" en funktion. (Lägga till saker före och efter funktionen.)

Arv utan virtuella funktioner
Exempel

```
struct Clock{
    Clock(int h, int m, int s) :seconds{60*(60*h+m) + s} {}
    Clock& tick();      // OBS! Inte virtuell
    int get_ticks() {return seconds;}
private:
    int seconds;
};
struct AlarmClock : public Clock {
    using Clock::Clock;
    void setAlarm(int h, int m, int s);
    AlarmClock& tick(); // döljer Clock::tick()
    void soundAlarm();
private:
    int alarmTime;
};

AlarmClock& AlarmClock::tick()
{
    Clock::tick();  // explicit anrop av funktion i superklassen
    if(get_ticks() == alarmTime)  soundAlarm();
    return *this;
}
```


Fallgropar

- Typomvandling
- Tilldelning eller kopiering av objekt av konkreta typer

Typomvandling

- Se upp med typomvandlingar
  - Framför allt (Subklass*) BasklassPekare
  - Inget skyddsnät, ingen ClassCastException
- Använd **dynamic_cast** (returnerar nullptr om ej OK)

```
Vektor v;

Container* c = &v;

if(dynamic_cast<Vektor*>(c)) {
    cout << " *c instanceof Vektor\n";
}
```

- **typeid** motsvarar .getClass() i Java

```
if(typeid(*c) == typeid(Vektor)) {
    cout << " *c is a Vektor\n";
}
```


Object slicing
Exempel

```
class Point {...};
class Point3d : public Point {...};

Point3d b;
Point a = b;
```

Inte farligt, men a innehåller bara Point-delen av b

```
Point3d b1;
Point3d b2;

Point& point_ref = b2;
point_ref = b1;
```

Fel! b2 innehåller nu Point-delen av b1 och Point3d-delen av sitt gamla värde.

Object slicing
Exempel

```
struct Point{
    Point(int xi, int yi) :x{xi}, y{yi} {}
    virtual void print() const; // prints Point(x,y)
    int x;
    int y;
};

struct Point3d :public Point{
    Point3d(int xi, int yi, int zi) :Point(xi,yi), z{zi} {}
    virtual void print() const; // prints Point3d(x,y,z)
    int z;
};

void test_slicing() {
    Point3d q1{1,2,3};
    Point3d q2{3,4,5};

    q2.print();     Point3d(3,4,5)
    Point& pr = q2;
    pr = q1;                    Lösning: virtuell operator=
    q2.print();     Point3d(1,2,5)
}
```


Object slicing
Lösning med virtuell operator=

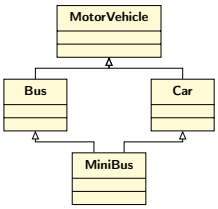
```
struct Point {
  ...
  virtual Point& operator=(const Point& p) =default;
};

struct Point3d :public Point{
  ...
  virtual Point3d& operator=(const Point& p) noexcept;
};

Point3d& Point3d::operator=(const Point& p) noexcept
{
  Point::operator=(p);
  auto p3d = dynamic_cast<const Point3d*>(&p);
  if(p3d){
    z = p3d->z;
  } else {
    z = 0;
  }
  return *this;
}
```

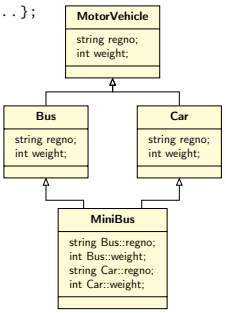
Multipelt arv

- En klass kan ärva från flera basklasser
- Jfr. implementera flera `interface` i Java
  - Som i Java om max en av basklasserna har medlemsvariabler
  - Kan bli besvärligt annars
- *The diamond problem*
  - Hur många `MotorVehicle` ingår i en `MiniBus`?



Multipelt arv
Hur många `MotorVehicle` ingår i en `MiniBus`?

```
class MotorVehicle {...};
class Bus : public MotorVehicle {...};
class Car : public MotorVehicle {...};
class MiniBus : public Bus, public Car {...};
```



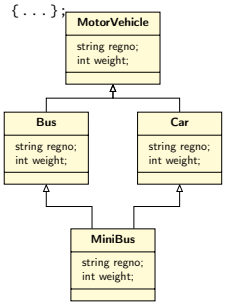
Multipelt arv
The diamond problem

- Den gemensamma basklassen tas med flera gånger
  - Flera upplagor av medlemsvariabler
  - Medlemsfunktioner måste användas som `Basklass::funktion()` för att undvika tvetydighet
- om inte *virtuellt arv* används

Multipelt arv
Virtuellt arv

Virtuellt arv : Subklasser delar instans av basklassen.
(Basklassen tas bara med en gång)

```
class MotorVehicle {...};
class Bus : public virtual MotorVehicle {...};
class Car : public virtual MotorVehicle {...};
class MiniBus : public Bus, public Car {...};
```



Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Dynamisk poymorfism och arv kapitel 15 – 15.4
Tillgänglighet och scope kapitel 15.5 – 15.6
Typomvandling och arv kapitel 15.2.3
Arv och resurshantering 15.7
Polymorfa typer och containers 15.8
Multipelt arv 18.3
Virtuella basklasser 18.3.4 – 18.3.5