EDAF30 – Programmering i C++

Arv. Polymorfism.

Sven Gestegård Robertz
Datavetenskap, LTH

2015

Innehåll

1. Kort om dynamisk polymorfism
   - Konkreta och abstrakta typer
   - Virtuella funktioner
2. Arv i C++
   - Konstruktorer och destruktorer
   - Tillgänglighet
   - Arv utan polymorfism
   - Fallgropar
3. Multipelt arv
4. Något om statisk polymorfism
   - Parametriserade klasser
   - Parametriserade funktioner

Polymorfism och dynamisk bindning

Polymorfism (mångformighet)

Överlagring	*Statisk bindning*
Generiska programenheter (templates)	*Statisk bindning*
Virtuella funktioner	*Dynamisk bindning*

Statisk bindning: Betydelsen hos en viss konstruktion avgörs vid *kompilering*

Dynamisk bindning: Betydelsen hos en viss konstruktion avgörs vid *exekvering*

Arv. Generalisering och specialisering

- Generalisering: abstrahera gränssnitt
- Specialisering: återanvändning och utökning
- Relationen *är*: En bil *är ett* fordon

Konkreta och abstrakta typer

Konkreta typer uppför sig "precis som inbyggda typer":

- *Representationen* ingår i *definitionen* [1]
- Kan placeras på stacken, och i andra objekt
- Kan refereras till direkt (och inte bara genom pekare eller referenser)
- Kod som använder den *måste kompileras om* om typen ändras
- *Detta är en kort introduktion, mer om mallar kommer i senare föreläsning.*

Abstrakta typer isolerar användaren från implementationsdetaljer och *skiljer gränssnittet från representationen*:

- Representationen av objekt (*inkl. storleken!*) är okänd
- Måste allokeras på heapen
- Kan bara refereras via pekare eller referenser

[1] kan vara privat, men är känd

Konkreta och abstrakta typer
En konkret typ: Vektor

```
class Vektor {
public:
    Vektor(int l = 10) :p{new int[l]},ant{l} {}
    ~Vektor() {delete[] p;}
    int lngd() const {return ant;}
    int& operator[](int i) {assert(i<ant); return p[i];}
private:
    int *p;
    int ant;
};
```

Generalisering: *extract interface*

```
class Container {
public:
    int lngd() const;
    int& operator[](int o);
};
```

Konkreta och abstrakta typer
Generalisering: en abstrakt typ, Container

```
class Container {
public:
    virtual int lngd() const  =0;
    virtual int& operator[](int o) =0;
    virtual ~Container() {}
};

class Vektor :public Container {
public:
    Vektor(int l = 10) :p{new int[l]},ant{l} {}
    ~Vektor() {delete[] p;}
    int lngd() const override {return ant;}
    int& operator[](int i) override {assert(i<ant); return p[i];}
private:
    int *p;
    int ant;
};
```

- *pure virtual* funktion
- Abstrakt klass
- eller `interface` i Java

- `extends` (eller `implements`) `Container` i Java
- `override` motsvarar `@Override` i Java (C++11)
- En polymorf typ måste ha en virtuell destruktor

Konkreta och abstrakta typer
Variant, utan att ändra Vektor

Om vi inte kan (eller vill) ändra klassen `Vektor` kan vi använda den för att skapa en ny klass:

```
class Vektor_container :public Container {
public:
    Vektor_container(int l = 10) :v{l} {}
    ~Vektor_container() {}
    int lngd() const override {return v.lngd();}
    int& operator[](int i) override {return v[i];}
private:
  Vektor v;
};
```

- `Vektor` är en konkret klass
- Notera att `v` är ett `Vektor`-objekt, inte en referens
  - Skillnad från Java
- `Vektor`s destruktor (för `v`) anropas implicit

Konkreta och abstrakta typer
Användning av abstrakta klassen

```
  void fill(Container& c, int v)
{
    for(int i=0; i!=c.lngd(); ++i){
        c[i] = v;
    }
}
void print(Container& c)
{
    for(int i=0; i!=c.lngd(); ++i){
        cout << c[i] << " " ;
    }
    cout << endl;
}
void test_container()
{
    Vektor v(10);

    print(v);
    fill(v,3);
    print(v);
}
```


Konkreta och abstrakta typer
Användning av abstrakta klassen

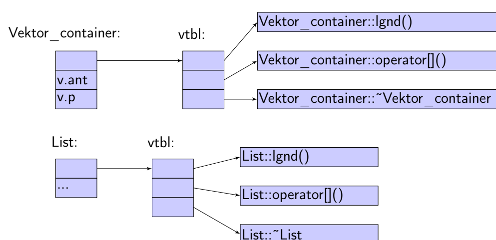
Anta nu att vi har två andra subklasser till `Container`

```
class MyArray : public Container { ...};
class List : public Container { ...};

void test_container()
{
    Vektor v(10);
    print(v);
    fill(v);
    print(v);

    MyArray a(5);
    fill(a);
    print(a);

    List l{1,2,3,4,5,6,7};
    print(l);
}
```

- Dynamisk bindning av `Container::lngd()` och `Container::operator[]()`

Dynamisk bindning

- virtuell funktions-tabell(*vtbl*)
  - innehåller pekare till objektets virtuella funktioner
  - varje klass med någon virtuell medlemsfunktion har en vtbl
  - varje objekt har en pekare till klassens vtbl
  - anrop av en virtuell funktion (typiskt) $< 25\%$ dyrare



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 användas.
    (`super()` som i Java finns inte p g a multipelt arv.)

Konstruktorer vid arv

Initieringsordning i en konstruktor (för härledd klass)

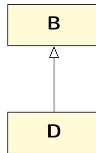
1 *Basklassen initieras*: Basklassens konstruktor anropas
2 *Subklassen initieras*: Datamedlemmar (i subklassen) initieras
3 Funktionskroppen i subklassens konstruktor exekveras

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();
}
```


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

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**	**public**
	protected	**protected**
	private	**private**
Skyddat arv	**public**	**protected**
	protected	**protected**
	private	**private**
Privat arv	**public**	**private**
	protected	**private**
	private	**private**

Tillgängligheten inuti D påverkas *inte* av typen av arv

Funktionsöverlagring och arv

Funktionsöverlagring fungerar ej mellan olika nivåer i arvshierarkin

```
class C1 {
public:
        void f(int); // C1::f
};

class C2 : public C1 {
public:
        void f(); // C2::f (döljer C1::f)
};

//...
C1 a; C2 b;
a.f(5); // Ok
b.f(); // Ok
b.f(2) // Fel! C1::f är dold!
```


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`
  - Se exempel i C++ direkt, kap 9.1 (s 307)
- 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

```
class A {...};
class B : public A {...};

B b;
A a = b;
```

Inte farligt, men `a` innehåller bara `A`-delen av `b`

```
B b1;
B b2;
A& a_ref = b2;
a_ref = b1;
```

Fel! `b2` innehåller nu `A`-delen av `b1` och `B`-delen av sitt gamla värde.

Object slicing

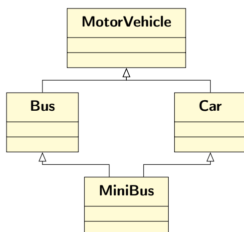
```
struct A
{
    A(int a) : a_var(a) {}
    int a_var;
    virtual void print() {cout << "A("<<a_var<<")\n";}
};

struct B : A
{
    B(int a, int b) : A(a), b_var(b) {}
    int b_var;
    void print() {cout << "B("<<a_var<<","<<b_var<<")\n";}
};

B b1(1,2);
B b2(98,99);

A& ar = b2;

b2.print();       B(98,99)
ar = b1;
b2.print();       B(1,99)
```


Object slicing
Lösning med virtuell `operator=`

```
struct A {
    // ...
    virtual A& operator=(const A& a) {
        assign(a);
        return *this;
    }
protected:
    void assign(const A& a); // assign As members;
};
struct B : public A {
    // ...
    virtual B& operator=(const A& a) {
        if(const B* b = dynamic_cast<const B*>(&a)){
            assign(*b);
        } else // handle bad assignment to B
        return *this;
    }
protected:
    void assign(const B& b) {
        A::assign(b);
        // assign Bs members
    }
};
```

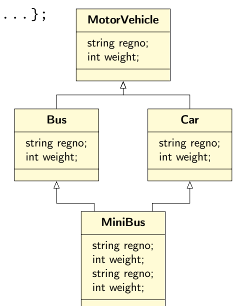

Multipelt arv

- En klass kan ärva från flera basklasser
- Jfr. implementera flera `interface` i Java
- *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 {...};
```



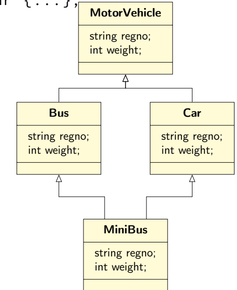
Multipelt arv
The diamond problem

- 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 {...};
```



Parametriserade typer och funktioner

- Templates (mallar)
  - Klassmallar
  - Funktionsmallar
- Kompilatorn skapar nya typer vid *instansiering*
- *Detta är en kort introduktion, mer om mallar kommer i en senare föreläsning.*

Parametriserade typer och funktioner
Exempel: parametriserad `Vektor`

```
template <typename T>
class Vektor {
public:
    Vektor(int l = 10) :p{new T[l]},ant{l} {}
    ~Vektor() {delete[] p;}
    int lngd() const {return ant;}
    T& operator[](int i) {return p[i];}
private:
    T* p;
    int ant;
};
```

Användning:

```
Vektor<int> iv;
Vektor<Foo> fv;

// ...

int i = iv[4];
Foo& F = fv[2];
```


Från tidigare: Konkreta och abstrakta typer
Användning av abstrakta klassen `Container`

```
void print(Container& c)
{
    for(int i=0; i!=c.lngd(); ++i){
        cout << c[i] << " " ;
    }
    cout << endl;
}

void test_container()
{
    Vektor v(10);

    //...

    print(v);
}
```


Parametriserade typer och funktioner
Exempel: parametriserade funktioner

```
template <typename T>
void print(T& t)
{
    for(int i=0; i != t.lngd(); ++i){
        cout << t[i] << " ";
    }
    cout << endl;
}
```

Användning:

```
Vektor<int> iv;
Vektor<Foo> fv;
// ...
print(iv);
print(fv);
```

Kompilatorn skapar *(instansierar)* funktionerna `print(Vektor<int>)` och `print(Vektor<Foo>)`.

Vi har talat om

- Virtuella fuktioner
- Konkreta och abstrakta typer
- Arv i C++

Nästa föreläsning:

Vi kommer att

- introducera exceptions och
- studera standard-klasserna för I/O