EDAF30 – Programmering i C++

8. Klasser; resurshantering och polymorfism

Sven Gestegård Robertz
Datavetenskap, LTH

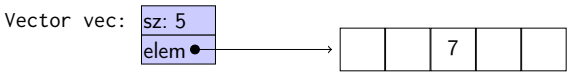
2016

Innehåll

Klasser
Resurshantering, representation

```
struct Vector {
  Vector(int s) :sz{s},elem{new double(sz)} {}
  ~Vector() {delete[] elem;}
  double& operator[](int i) {return elem[i];}
  int sz;
  double* elem;
};

void test()
{
   Vector vec(5);
   vec[2] = 7;
}
```



- *Resource handle* – Vector äger sin `double[]`
- objektet: pekare + storlek, arrayen ligger på heapen

Pekaren `this`
Självreferens

I en medlemsfunktion finns den implicita *pekaren* **`this`**, som pekar på objektet som funktionen anropades för. (jfr. **`this`** i Java).

Exempel på användning

```
struct Clock {
  Clock();
  Clock& set(int h, int m, int s);
  Clock& tick();
  Clock& print();
private:
  int seconds;
};
Clock& Clock::tick() {
    ++seconds;
    return *this;
}
```

Implicit deklarerad `Clock* const this`

Om vi har en variabel `Clock c;` kan vi nu "kedja" anrop:
`c.set(12,15,0).tick().tick().print();`

Överlagring av operatorer

Kan göras för de flesta operatorer, utom

```
sizeof   .   .*   ::   ?:
```

T ex kan dessa operatorer överlagras

```
=
+ - * / %
^ & | ~
<< >>
&& || !
!= == < >
++ -- += *= .......
() []
-> ->*
&
new delete new[] delete[]
```


Överlagring av operatorer

Överlagring av operatorer görs med syntaxen

```
returtyp operator⊗ (parametrar...)
```

för någon operator ⊗ t.ex. == eller +

Kan, för klasser, göras på två sätt:

- som medlemsfunktion
  - om ordningen på operanderna är lämplig
- som *fri* funktion
  - om det publika gränssnittet räcker, *eller*
  - om funktionen deklareras **`friend`**

Överlagring av operatorer
som medlemsfunktioner och fria funktioner

Exempel: deklaration som medlemsfunktioner

```
class Komplex {
public:
    Komplex(float r, float i) : re(r), im(i) {}
    Komplex  operator+(const Komplex& rhs) const;
    Komplex  operator*(const Komplex& rhs) const;
    // ...
private:
    float re, im;
};
```

Exempel: deklaration `operator+` som `friend`

Deklaration inuti klassdefinitionen för `Komplex`:

```
friend Komplex operator+(const Komplex& l, const Komplex& r);
```

Notera antalet parametrar

Överlagring av operatorer

Överlagrade operatorer i användning:

Exempel: Komplexa tal

```
Komplex a = Komplex(1.2, 3.4);
Komplex b = Komplex(2.3, 1);
Komplex c = b;

a = b + c;        // a = b.operator+(c);
b = b + c * a;
c = a * b + Komplex(7, 4.5);
```


Överlagring av operatorer

Definition av operatorn + på två sätt

- Som medlemsfunktion

```
Komplex Komplex::operator+(const Komplex& rhs)const{
    Komplex temp;
    temp.re = re + rhs.re;
    temp.im = im + rhs.im;
    return temp;
}
```

- Som fri funktion

```
Komplex operator+(const Komplex& l, const Komplex& r){
        Komplex temp;
        temp.re = l.re + r.re;
        temp.im = l.im + r.im;
        return temp;
}
```

Att denna är friend syns bara i friend-deklarationen i klassen

Överlagring av operatorer

Definition av operatorn + på två sätt

- Som medlemsfunktion

```
Komplex Komplex::operator+(const Komplex& rhs)const{
    Komplex temp;
    temp.re = re + rhs.re;
    temp.im = im + rhs.im;
    return temp;
}
```

så att högra operanden inte kan ändras

så att vänstra operanden inte kan ändras

- Som fri funktion

```
Komplex operator+(const Komplex& l, const Komplex& r){
        Komplex temp;
        temp.re = l.re + r.re;
        temp.im = l.im + r.im;
        return temp;
}
```

Att denna är friend syns bara i friend-deklarationen i klassen

Överlagring av operatorer
Annan variant av + som använder +=

Klassdefinition

```
class Komplex {
public:
    const Komplex& operator+=(const Komplex& z) {
        re += z.re;
        im += z.im;
        return *this;
    }
    // ...
};
```

Returnerar **const**-referens för att inte tillåta t.ex. `(a += b) = c;`

Fri funktion, behöver inte vara `friend`

```
Komplex operator+(Komplex a, Komplex b) {
    return a+=b;
}
```

NB! *värdeanrop*: vi vill returnera *en kopia*.

Överlagring av operatorer

Binära operatorer: Variant av + med *heltal* som *höger* operand

Deklarationen (i klassdefinitionen av Komplex)

```
Komplex Komplex::operator+(int d) const;
```

Definitionen (utanför klassdefinitionen)

```
Komplex Komplex::operator+(int d) const {
        Komplex temp(*this);
        temp.re += d;
        return temp;
}
```

Överlagring av operatorer

Binära operatorer: Variant av + med heltal som *vänster* operand

- Problem: Kan inte använda medlemsfunktion! (varför?)

Deklarationen (Obs! *Utanför* klassdefinitionen)

```
Komplex operator+ (int d, const Komplex& v);
```

Definitionen (Obs! Ingen medlemsfunktion!)

```
Komplex operator+ (int d, const Komplex& v) {
        return v + d; // Utnyttjar andra +-op.!
}
```

Behöver inte vara **friend**: använder bara det publika gränssnittet.

Överlagring av operatorer
Exempel: <<

Exempel på friend-deklarerad operator: `<<` (`#include <ostream>`)

Deklarationen (i klassdefinitionen)

```
friend ostream& operator<<(ostream& o, const Komplex& v);
```

Definitionen (Obs! Ingen medlemsfunktion)

```
ostream& operator<<(ostream& o, const Komplex& v) {
    o << v.re << '+' << v.im << 'i';
    return o;
}
```


Överlagring av operatorer

Unära operatorer: Ökningsoperatorerna ++

Deklarationen (i klassdefinitionen)

```
const Komplex& operator++ ();    // preinkrement  (++v)
      Komplex  operator++ (int); // postinkrement (v++)
```

Dummy-parameter för att markera postinkrement-varianten

Definitionen (utanför klassdefinitionen)

```
const Komplex& Komplex::operator++ () { // prefix
        return (*this) += 1; // Returnera inkrementerad
}
Komplex Komplex::operator++ (int) {     // postfix
        Komplex temp(*this); // Kopia av detta objekt
        (*this) += 1;
        return temp; // Returnera oinkrementerad kopia
}
```


Typomvandlings-operatorer
Exempel: `Counter`

Konvertering till `int`

```
struct Counter {
    Counter(int c=0) :cnt{c} {};
    Counter& inc()       {++cnt; return *this;}
    Counter  inc() const {return Counter(cnt+1);}
    int get() const {return cnt;}
    operator int() const {return cnt;}
private:
    int cnt{0};
};
```

Notera: **operator** `T()`.

- returtyp anges inte
- kan deklareras **explicit**

Överlagring av operatorer
Tilldelningsoperatorn: **operator=** (*copy assignment*)

Deklarationen (i klassdefinitionen av Vektor)

```
const Vektor& operator=(const Vektor& v);
```

Definitionen (utanför klassdefinitionen)

```
const Vektor& Vektor::operator=(const Vektor& v)
{
  if (this != &v) {
      delete[] elem;
      sz = v.sz;
      elemp = new int[sz];
      for (int i=0; i<sz; i++)
          elem[i] = v.elem[i];
  }
  return *this;
}
```

1. kolla "self assignment"
2. Frigör gamla resurser
3. Allokera nya resurser
4. Kopiera värden

För felhantering bättre att allokera och kopiera först och bara göra **delete** *om allokeringen lyckades.*

"Rule of three"
Canonical construction idiom

Om en klass äger någon resurs, ska den ha en egen

1. Destruktor
2. Copy constructor
3. Copy assignment operator

för att inte läcka minne. T ex. klassen `Vektor`

Regel: Om du definerar *någon*, ska du definiera *alla*.

Varnande exempel
Default-kopiering

För klasser som äger resurser fungerar inte default-kopiering.

- värdeanrop
- kopiering
- destruktorn körs vid **`return`**
- *dangling pointer*

Exempel: `Vector`

"Rule of ~~three~~ five"
Canonical construction idiom, fr o m C++11

Om en klass äger någon resurs, ska den ha en egen

1. Destruktor
2. Copy constructor
3. Copy assignment operator
4. *Move* constructor
5. *Move* assignment operator

Move semantics

- Onödigt att kopiera om man vet att källan ska förstöras direkt
- Bättre att "stjäla" innehållet
- Gör *resource handles* ännu effektivare
- Vissa objekt får/kan inte kopieras
  - `std::move` gör om till en *rvalue-referens* (`T&&`)

Move semantics (C++11)
Exempel: Vektor

Copy-konstruktor

```
Vector::Vector(const Vector& v) :elem{new double[v.sz]}, sz{v.sz}
{
    for(int i=0; i < sz; ++i) {
        elem[i] = v[i];
    }
}
```

Move-konstruktor

```
Vector::Vector(Vector&& v) : elem{v.elem}, sz{v.sz}
{
    v.elem = nullptr;
    v.sz = 0;          // v har inga element
}
```


Move semantics (C++11)
Exempel: Vektor

Copy-assignment

```
Vector& Vector::operator=(const Vector& v) {
   if(this != &v) {
       double* tmp = new double[v.sz];
       for(int i = 0; i < v.sz; ++i) {
           tmp[i] = v[i];
       }
       delete[] elem;
       elem = tmp;
       sz = v.sz;
   }
   return *this;
}
```

Move-assignment

```
Vector& Vector::operator=(Vector&& v) {
    if(this != &v) {
        elem = v.elem;       // "flytta" arrayen från v
        v.elem = nullptr;    // markera att v är ett "tomt skrov"
        sz = v.sz;
        v.sz = 0;
    }
    return *this;
}
```


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

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

- Representationen av objekt (*inkl. storleken!*) är okänd
- Kan bara refereras via pekare eller referenser
- Måste allokeras på heapen[2]

[1] kan vara privat, men är känd
[2] Egentligen: objekt av en abstrakt typ kan inte skapas, men objekt på stacken blir inte polymorfa

Konkreta och abstrakta typer
En konkret typ: Vektor

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

Generalisering: *extract interface*

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


Konkreta och abstrakta typer
Generalisering: en abstrakt typ, Container

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

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

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

- `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
Användning av abstrakta klassen

```
  void fill(Container& c, int v)
{
    for(int i=0; i!=c.size(); ++i){
        c[i] = v;
    }
}
void print(Container& c)
{
    for(int i=0; i!=c.size(); ++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::size()` och `Container::operator[]()`

Konkreta och abstrakta typer
Variant, utan att ändra Vektor

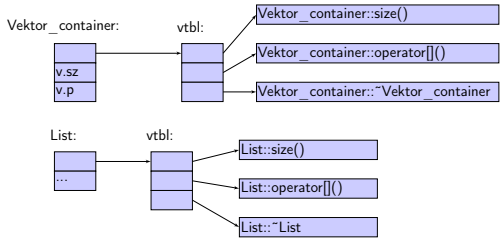
Om vi inte kan (eller vill) ändra klassen `Vektor` kan vi använda den för att skapa en ny klass:

```
class Vektor_container :public Container {
public:
    Vektor_container(int l = 10) :v{l} {}
    ~Vektor_container() {}
    int size() const override {return v.size();}
    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

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



Nästa föreläsning

Dynamisk poymorfism och arv kapitel 15 – 15.4
Typomvandling och arv kapitel 15.2.3
Arv och resurshantering 15.7
Klassmallar 16.1.2

Läsanvisningar

Referenser till relaterade avsnitt i Lippman
Klasser, resurshantering 13.1, 13.2
Move semantics 13.6
Operatorer och typomvandling kapitel 14
Dynamisk poymorfism och arv kapitel 15 – 15.2.2
Pekaren `this` s. 257–260