EDAF30 – Programmering i C++

Klasser. Operatorer.

Sven Gestegård Robertz
Datavetenskap, LTH

2015

Innehåll

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() const;
private:
  int seconds;
};
Clock& Clock::tick() {
    ++seconds;
    return *this; // this är en pekare
}
```

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

Konstanta objekt

- **const** betyder "jag lovar att inte ändra denna"
- Objekt (variabler) kan vara deklarerade **const**
  - "jag lovar att inte ändra värdet på variabeln"
- Medlemsfunktioner kan vara deklarerade **const**
  - "jag lovar att funktionen inte ändrar objektets tillstånd"

Konstanta objekt
Exempel

`const` objekt och `const` funktion

```
class Counter {
private:
    int cnt{0};
public:
    int inc() {return ++cnt;}
    int get() const {return cnt;}
};
void test_const()
{
    Counter c;
    c.inc();
    cout << c.get() << endl;

    const Counter cc;
    cc.inc();                  // fel, cc är const
    cout << cc.get() << endl; // OK, get() är const
}
```

```
passing 'const Counter' as 'this' argument discards qualifiers
```


Konstanta objekt
Exempel

Notera **const** i deklarationen (och i definitionen!) av medlemsfunktionen `get()`: ("**const** *ingår i namnet*")

```
class Counter {
public:
    //...
    int get() const;      // funktionsdeklaration
    //...
};


int Counter::get() const  // funktionsdefinition
{
    return cnt;
}
```

Konstanta objekt
Exempel: överlagring med **const**

Notera **const** i deklarationen (och i definitionen!) av medlemsfunktionen `get()`: (“**const** *ingår i namnet*”)

Funktionerna `inc()` och `inc()` **const** *är olika funktioner*.

Exempel

```
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;}
private:
    int cnt{0};
};
```


Statiska medlemmar

Statiska medlemmar: Delas mellan alla objekt i klassen

- *deklareras* i klassdefinitionen
- *definieras* utanför klassdefinitionen (om inte **const**)
- kan vara **public** eller **private**

Statiska medlemmar
Exempel: räkna allokeringar och avallokeringar

```
class Foo {
private:
    static int created;
    static int alive;
public:
    Foo() {++created; ++alive;}
    ~Foo() {--alive;}

    static void print_counts();
};
```

Definitioner: *NB! inte* **static**

```
int Foo::created{0};
int Foo::alive{0};

void Foo::print_counts()
{
    cout << alive << " / ";
    cout << created << endl;
}
```

```
void test_static_member()
{
    {
        Foo a;
        a.print_counts();

        Foo b;
        b.print_counts();
    }

    {
        Foo c;
        Foo::print_counts();
    }
    Foo::print_counts();
}
```

```
1 / 1
2 / 2
1 / 3
0 / 3
```


Vänner (**friend**)

Funktioner eller klasser med full tillgång till medlemmar i en klass utan att själv vara medlem

Deklaration i klassen Vektor

```
class Vektor{
    //...
    friend void kvadrera(Vektor& v);
};
```

Definition utanför klassen Vektor

```
void kvadrera(Vektor& v) {
    for (int i=0; i<v.ant; i++)
      v.p[i] *= v.p[i];
}
```

Även medlemsfunktioner från andra klasser eller hela klasser kan vara vänner (jfr. package-synlighet i Java).

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

Fundera över – Objekt som returvärden

- Vi kan inte returnera referenser till objekt som allokerats (på stacken) i en funktion.
- Hur ineffektivt är det att istället returnera kopior av objekt?

Fundera över – Objekt som returvärden

- Vad skrivs ut av programmet nedan?

```
class C {
public:
  C() : a(0), b(0) { cout << "A C was made.\n"; }
  C(const C& aC) : a(aC.a),b(aC.b) {cout << "A copy was made.\n";}
private:
  int a, b;
};

C f() {
  return C();
}

int main() {
  cout << "Hello World!\n";
  C obj = f();
}
```

Fundera över – Objekt som returvärden
Svar

Som programmet är skrivet anropas

- default-konstruktorn en gång och
- copy-konstruktorn två gånger
  1. vid initialiseringen av den anonyma returvariabeln i `f()` och
  2. vid initialiseringen av variabeln `obj`.
- Vad som faktiskt skrivs ut beror på hur bra kompilatorn är på att optimera!
- Både Visual Studio och gcc (CodeBlocks) optimerar bort bägge anropen av copy-konstruktorn och ger följande utskrift:

```
Hello World!
A C was made.
```

- Andra kompilatorer kan ge andra svar

Överlagring av operatorer
Annan variant av + som använder +=

Klassdefinition

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

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
Exempel: `Vektor`, Jämförelseoperatorn ==

Deklarationen (i klassdefinitionen av Vektor)

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

Definitionen (utanför klassdefinitionen)

```
bool Vektor::operator==(const Vektor& v) const {
    if (ant!=v.ant)
        return false;
    for (int i=0; i<ant; i++)
        if (p[i] != v.p[i])
            return false;
    return true;
}

// v1 == v2 tolkas som v1.operator==(v2)
```


Överlagring av operatorer
Operatorn +=

Deklarationen (i klassdefinitionen av Vektor)

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

Definitionen (utanför klassdefinitionen)

```
const Vektor& Vektor::operator+=(const Vektor& v) {
    assert(ant == v.ant);
    for (int i=0; i<ant; i++)
        p[i] += v.p[i];
    return *this;
}
```

Returnerar **const**-referens för att inte tillåta t.ex.
`(v1 += v2) = v3;`

Överlagring av operatorer
Operatorn +

Deklarationen (i klassdefinitionen av Vektor)

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

Definitionen (utanför klassdefinitionen)

```
Vektor Vektor::operator+(const Vektor& v) const {
    assert(ant == v.ant);
    Vektor temp(*this);
    temp += v;
    return temp;
}
// Ingen referens som returvärde (för att undvika
// kvardröjande pekare (dangling pointers))
```


Överlagring av operatorer

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

Deklarationen (i klassdefinitionen av Vektor)

```
const Vektor& operator+=(int d);
```

Definitionen (utanför klassdefinitionen)

```
const Vektor& Vektor::operator+=(int d) {
        for (int i=0; i<ant; i++)
                p[i] += d;
        return *this;
}
```

Överlagring av operatorer

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

Deklarationen (i klassdefinitionen av Vektor)

```
Vektor operator+(int d) const;
```

Definitionen (utanför klassdefinitionen)

```
Vektor Vektor::operator+(int d) const {
        Vektor temp(*this);
        temp += 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 av Vektor)

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

Definitionen (Obs! Ingen medlemsfunktion!)

```
Vektor operator+ (int d, const Vektor& 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 av Vektor)

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

Definitionen (Obs! Ingen medlemsfunktion)

```
ostream& operator<<(ostream& o, const Vektor& v) {
    o << '{';
    if (v.ant > 0)
        o << v.p[0];
    for (int i=1; i<v.ant; ++i)
        o << ", " << v.p[i];
    o << '}';
    return o;
}
```


Överlagring av operatorer

Unär operator: Teckenskifte –

Deklarationen (i klassdefinitionen av Vektor)

```
Vektor operator- () const;
```

Definitionen (utanför klassdefinitionen)

```
Vektor Vektor::operator- () const {
        Vektor temp(*this); //Temp. kopia
        for (int i=0; i<ant; i++)
                temp.p[i] = -temp.p[i];
        return temp;
}
```


Överlagring av operatorer

Unära operatorer: Ökningsoperatorerna ++

Deklarationen (i klassdefinitionen av Vektor)

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

Dummy-parameter för att markera postinkrement-varianten

Definitionen (utanför klassdefinitionen)

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


Överlagring av operatorer
Indexeringsoperatorn []: **const** och icke-**const**

Deklarationen (i klassdefinitionen av Vektor)

```
int& operator[] (int i);
int  operator[] (int i) const; // Se nedan!
```

Definitionen (utanför klassdefinitionen)

```
int& Vektor::operator[] (int i) {
        assert(i>=0 && i<ant);
        return p[i]; // Returnerar en referens
}
```

lvalue&: Kan användas som vänsterled i en tilldelning

Överlagrad av variant för konstanta objekt:

```
int Vektor::operator[] (int i) const {
        assert(i>=0 && i<ant);
        return p[i]; // Returnerar en kopia
}
```

rvalue: Kan inte tilldelas (temporärt värde)

Funktionsanrops-operatorn
Exempel: `Rand_int` från föreläsning 1

Deklaration

```
class Rand_int {
public:
    Rand_int(int low, int high);
    int operator()() {return dist(re);}
private:
    std::default_random_engine re;
    std::uniform_int_distribution<> dist;
};
```

Användning:

```
int main()
{
  Rand_int dice{1,6};

  cout << dice() << endl;
}
```


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) {     // inte tilldelning till sig själv
     delete[] p;       // Städa bort gamla arrayen
     ant = v.ant;
     p = new int[ant]; // allokera en ny array
     for (int i=0; i<ant; i++)
             p[i] = v.p[i];
 }
 return *this;
}
```


"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`

"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 (C++11)
Exempel: Vektor

Copy-konstruktor

```
Vektor::Vektor(const Vektor& v) : ant(v.ant)
{
  p = new int[ant];     // Allokera ny array
  for (int i=0; i<ant; i++)
    p[i] = v.p[i];      // Kopiera elementen
}
```

Move-konstruktor

```
  Vektor::Vektor(Vektor&& v) : ant(v.ant)
{
    p = v.p;            // Flytta pekaren till data
    v.p = nullptr;
    v.ant = 0;
}
```

`std::move` gör om till en *rvalue-referens* (`T&&`)

Move semantics (C++11)
Exempel: Vektor

Copy-assignment

```
const Vektor& Vektor::operator=(const Vektor& v) {
  if (this != &v) {    // inte tilldelning till sig själv
     delete[] p;       // Städa bort gamla arrayen
     ant = v.ant;
     p = new int[ant];           // allokera en ny array
     for (int i=0; i<ant; i++)   // kopiera data
        p[i] = v.p[i];
  }
  return *this;
}
```

Move-assignment

```
const Vektor::Vektor& Vektor::operator=(Vektor&& v) {
  if (this != &v) {   // inte tilldelning till sig själv
    delete[] p;       // Städa bort gamla arrayen
    ant = v.ant;
    p = v.p;          // "flytta" arrayen från v
    v.p = nullptr;    // markera att v är ett "tomt skrov"
    v.ant = 0;
}
```


Referens- eller värdeanrop
Tumregler

Om du skickar ett objekt som parameter till en funktion och

- du vill kunna *ändra* värdet på objektet
  - referens: `void f(T&);` eller
  - pekare: `void f(T*);`
- du vill *inte ändra* objektet, men det *är stort*
  - konstant referens: `void f(const T&);`
- annars, *använd värdeanrop*
  - `void f(T);`

Referens- eller värdeanrop
Tumregler

- hur stort är "stort"?
  - mer än ett par *ord*
- pekare eller referens?
  - om *inte ett objekt* (`nullptr`) behövs eller är rimligt: pekare
  - notera: medlemsfunktioner innebär (i princip) referensanrop av objektet
- alternativ till utparametrar
  - skriv program som är lätta att förstå

Exempel: två funktioner:

```
void incr1(int& x);
int  incr2(int x);
```

Här är det tydligare att `v = incr2(v)` ändrar `v`

Användning:

```
int v = 0;
//...

incr1(v);
v = incr2(v);
```


Sammanfattning

Vi har talat om

- **const** för objekt och medlemmar
- Statiska medlemmar
- pekaren **this**
- **friend**
- Överlagring av operatorer

Nästa föreläsning:

Vi kommer att studera

- Klasser och arv. Virtuella funktioner.