Programmering i C++
EDA623
Mer om klasser

Mer om klasser

Innehåll

- Konstanta objekt
- Statiska medlemmar
- Pekaren `this`
- Vänner (friends)
- Överlagring av operatorer

Konstanta objekt

Exempel: Bankkonto

Hantering av konstant objekt

```
const Bankkonto std_konto("0",0), *pk;

std_konto.insattning(1000); // Fel
std_konto.kontobesked();    // Fel !?
std_konto.uttag(250);       // Fel
pk = &std_konto;            // pk pekare till konstant objekt!
pk->uttag(4500);            // Fel
```

Konstanta objekt

Exempel: Bankkonto

Lägg till `const` i deklarationen (och i definitionen!) av medlemsfunktionen `kontobesked()`:

```
class Bankkonto {
public:
    //...
    void kontobesked() const;
    //...
};

//... dekl. som på föregående bild
std_konto.kontobesked();    // Ok nu!
```

Statiska medlemmar

Statiska medlemmar: Delas mellan alla objekt i klassen

Exempel: Bankkonto

```
class Bankkonto {
public:
    // ...
    static void andra_ranta(float r) {ranta=r;}
    private:
    // ...
    static float ranta;
}

// I .cpp
float Bankkonto::ranta = 2;

// Ändra ränta på klass-nivå (utan ngt obj.)
Bankkonto::andra_ranta(2.5);
```

Pekaren this

Självreferens

För att referera till sig själv kan ett objekt använda den ”dolda” pekaren `this` (motsvarande referensen `this` i Java).

Typiskt exempel när denna behövs

```
// I klassdefinitionen
Vektor& inc();

// Medlemsfunktion som returnerar det
// aktuella objektet inkrementerat
Vektor& Vektor::inc() {
    for (int i=0; i<ant; i++)
        p[i]++;
    return *this; // this är ju pekare
}
```

Vänner

Funktioner med ”VIP-kort” dvs funktioner med full access in i en klass utan att vara medlem

Deklaration i klassen 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 t.o.m. hela klasser kan vara vänner

Överlagring av operatorer

Kan göras för de flesta operatorer som är definierade för de inbyggda (primitiva) datatyperna. Undantag:

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

kan inte överlagras men det kan t.ex.

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

Överlagring av operatorer

Överlagring av operatorer görs med syntaxen
returtyp `operator`$\otimes$`(`*parametrar*`)`
för någon operator $\otimes$ t.ex. == eller +

Exempel: Komplexa tal

```
class Komplex {
public:
    Komplex(float r, float i) : re(r), im(i) {}
    Komplex operator+(const Komplex& rhs) const;
    Komplex operator*(const Komplex& rhs) const;
    // De andra medlemsfunktionerna som t.ex.
    // operator-, operator/, get_re, set_re ...
private:
    float re, im;
};
```

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

Alternativ implementering mha `friend`

Exempel: Komplexa tal

```
//Deklaration i klassen Komplex
friend Komplex operator+(const Komplex& l, const Komplex& r);
friend Komplex operator*(const Komplex& l, const Komplex& r);
//...

a = b + c; // a = operator+(b,c);
```

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 vänfunktion

```
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 deklarationen
```

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 vänfunktion

```
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 deklarationen
```

Puzzle Corner – Objekt som returvärden

Vi kan inte returnera referenser till objekt som skapas i en funktion.
Hur ineffektivt är det att istället returnera kopior av objekten?
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();
}
```

Puzzle Corner – Svar

- Som programmet är skrivet anropas default-konstruktorn en gång (`C()`) och copy-konstruktorn två gånger (vid initialiseringen av den anonyma returvariabeln i `f()` och vid initialiseringen av variablen `obj`).
- Vad som faktiskt skrivs ut beror på hur bra kompilatorn är på 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

Binära operatorer: 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

Binära 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;
}
// Referens till konstant som retur för att
// t.ex. (v1 += v2) += v3 ej ska kunna ske
```

Överlagring av operatorer

Binära 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.!
}
```

Alt. dekl. med `friend` i klassdefinitionen

```
friend Vektor operator+ (int d, const Vektor& v);
// Detta behövs dock ej i detta fall!
```

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++ (); // prefix  (++v)
Vektor operator++ (int);     // postfix (v++)
// Dummy-parameter i postfix för att se skillnad
```

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

Tilldelningsoperatorn =

Deklarationen (i klassdefinitionen av Vektor)

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

Definitionen (utanför klassdefinitionen)

```
const Vektor& Vektor::operator=(const Vektor& v) {
    if (this != &v) {//Uteslut tilld. till sig själv
        delete[] p; // Städa bort gammal bråte
        ant = v.ant;
        p = new int[ant]; // Fixa plats för ny vektor
        for (int i=0; i<ant; i++)
            p[i] = v.p[i];
    }
    return *this;
}
```

Överlagring av operatorer

Indexeringsoperatorn []

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
} // Kan anv. i v.l. 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
} // Kan ej anv i v.l. i en tilldelning!
```

Överlagring av operatorer

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;
}
```