

TDT4102

Prosedyre- og objektorientert programmering,

The image features the C++ logo, which consists of a large blue 'C' followed by two blue '+' signs. Behind the logo is a snippet of C++ code in a monospaced font, tilted at an angle. The code is:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!\n";
    return 0;
}
```

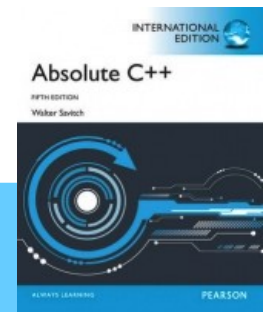
Overlagring av operatorer



Dagens forelesning

- Unære operatorer
- Operatorer som globale funksjoner
- Operatorer som medlemsfunksjoner
- Referanser og bruk av `const`
- Insertion operator (`<<`)
- Bruk av `friend` nøkkelordet i klasser

Kap. 8





Oppsummering: operatorer

- Operatorer er en del av programmeringsspråket
 - `+ / * % == < > []` ...
- Operatorer er som funksjoner
 - `2 + 3` kunne like gjerne vært skrevet som `+(2, 3)`
- Motivasjon: fin syntaks, lettest kode
 - Kan skrive `(d1 == d2)`
istedenfor `(d1.year == d2.year && ...)`

```
bool operator==(Date &d1, Date &d2);
```

Overlagring av unære operatorer



- C++ har også *unære* operatorer

Operatorer med EN operand

```
x = -y;    // minus-operatoren returnerer negative verdien av y
++x;      // ++ (prefiks) returnerer inkrementert verdi
```

- Av og til nyttig å overlagre de unære operatoren ++ og –
- Men kun hvis det gir mening for datatypen!

Inkrementering og dekrementering



- To versjoner av hver operator
- *Prefix* notasjon:

```
++x; --x;
```

- *Postfix* notasjon:

```
x++; x--;
```

Prefiks inkrementering

- "Prefiks" betyr at ++ skrives **foran** variabelen som skal inkrementeres

```
Date x = {1982, 12, 12};
Date y = ++x; // y og x er nå 1982-12-13
```

- Inkrementer verdien og returner den nye verdien

```
Date operator++(Date &d){
    d.day++;
    return d;
}
```

Postfiks inkrementering

- "Postfiks" betyr at ++ skrives **etter** variabelen som skal inkrementeres

```
Date x = {1982, 12, 12};
Date y = x++; // y blir 1982-12-12, x blir 1982-12-13
```

- Inkrementer verdien og returner den "gamle" verdien

```
Date operator++(Date &d, int){
    Date temp = d;
    d.day++;
    return temp;
}
```

Dummy int
parameter
så kompilatoren
skjønner
at det er postfiks

Referanse som returtype

- Av og til deklarerer også **returtypen** med **&**

- Unngår kopiering av store datatyper (klasser, structs)

```
Date& operator++(const Date &d){
    d.day++;
    return d;
}
```

- **NB: Call-by-reference ut krever call-by-reference inn!**
 - IKKE returner en referanse til en variabel som er lokal i funksjonen
 - Lokale variabler frigjøres når funksjonen returnerer og referansen vil da være ugyldig!

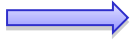

Returtype som er const

- Av og til deklarerer også **returtypen** med **const**
- I praksis mest for å unngå feil bruk av operatoren
 - fordi vi ikke ønsker at returnert verdi ikke skal kunne endres før tilordning (husk at tilordning er kopiering)
 - gjør at uttrykket ikke kan brukes på venstre side i en tilordning

- **Uten const** returtype er dette lov;

Date& operator ++(Date &d);  Date x,y; ++x = y;

- **Med const** returtype er dette **ikke** lov:

const Date& operator ++(Date &d);  Date x,y;  ++x = y;

L-value og R-value

$A = B;$ A er en L (left) value og B er en R (right) value

$A = B = C;$

B er L-value i ett uttrykk og resultatet fra $(B = C)$ er R-value i et annet

- Også kalt “lhs” og “rhs”
- L-value er noe vi kan tilordne
 - Dvs: L-value kan stå på venstre side
- Generell regel: hvis et resultat fra en funksjon eller operator-kall skal kunne brukes på venstre side i en tilordning, MÅ du bruke return-by-reference

Et greit eksempel er overlagring av indeksoperatoren (eks. er denne allerede overlagret for vector og string).

```
myObject[1] = verdi;
```



Insertion-operatoren: <<

- Nyttig å kunne overlagre << brukt til
 - `cout << d;`
- Litt spesiell fordi venstresiden er `&ostream`
- i tillegg må vi returnere &ostream slik at vi kan bruke operatoren flere ganger i samme statement

```
//insertion operatoren
ostream& operator << (ostream& out, const Date &d){
    out << setfill('0') << setw(4) << d.year << "-";
    out << setfill('0') << setw(2) << d.month << "-";
    out << setfill('0') << setw(2) << d.day;
    return out;
}

cout << x << " er før " << y << endl;
```

Overlagring av andre operatører



- Vi kan i praksis overlagre de fleste operatører
 - boolske operatører (<, >, <=, >=, &&, ||, !)
 - aritmetiske operatører (+, -, *, /, %)
 - indeks-operatoren ([])
 - << og >> (insert og extraction)
 - m.m.
- Men vi kan ikke overlagre
 - Scope-operatoren ::, sizeof-operatoren, if-else-operatoren ?:
- Finnes også noen flere begrensinger
 - Vi kan heller ikke ha default verdier for operatører
 - Og ikke alle kan overlagres for struct (kun for klasser)

Overlagring med mening

- Overlagring av operatorer er en nyttig mekanisme
 - gir oss mulighet for å skrive mer "lettelest" kode
- VIKTIGE PRINSIPP!
 - Ikke implementer operatorer som avviker fra den tradisjonelle meningen til operatoren
 - Ikke implementer operatorer som er meningsløse for verdiene (eks: dato + dato = ?????)

Operatoroverlagring og klasser



- I de fleste tilfellene er det for **klasse** vi ønsker å overlagre operatører
- Vi kan **velge** om vi vil lage operatoren **som medlem** av klassen **eller ikke-medlem**
- Hvis medlem defineres overlagret operator som **public medlem** i klassen
 - for binære operatører deklarerer **kun høyresiden** som parameter da kallende objekt implisitt er **venstre operand** til operatoren
 - for unære blir det **ingen** parameter

```
class Date{
private:
    int year;
    int month;
    int day;
public:
    Date();
    Date(int y, int m, int d);
    bool operator == (const Date &d);
};
```

```
bool Date::operator ==(const Date &d) {
    return ((year == d.year) &&
            (month == d.month) &&
            (day == d.day));
}
```

Bruk av this i operatorimplementasjoner

- For binære operatorer er venstreoperand objektet selv og høyreoperand parameter
- this er peker til objektet selv
- For unær operator er parameterlista tom

```
bool Date::operator>(const Date &d) const{
    //må derefere this for å få en type som
    //passer som venstreoperand
    return *this < d;
}
```

```
Date& Date::operator++(){
    //vi inkrementerer og
    //returnerer objektet selv
    day++;
    return *this;
}
```

```
Date Date::operator++(int){//med dummy int
    //Her skal vi selvsagt returnere
    //kopien og returtypen kan ikke
    //være referanse
    Date temp = *this;
    day++;
    return temp;
}
```



Hvorfor deklarere operator som medlem?

- Fordelen med å deklarere som **medlem** er at vi har automatisk tilgang til klassens **private variabler**
 - Slipper dermed å bruke get-funksjoner
- OG her er det viktig å huske at **private** gjelder for klassen
 - I praksis har objekter av samme klasse tilgang til alle private variabler/funksjoner!
- Samler også all relatert funksjonalitet på ett sted
 - i objektorienteringens “ånd”
- Andre mener det ikke er god objektorientering å ha operatorer som medl. hvis disse ikke endrer objektene

Et nytt sted å skrive `const`

- En medlemsfunksjon kan potensielt endre på alle medlemsvariabler
- Hvis vi skal markere at medlemsfunksjonen IKKE endrer noen medlemsvariabler
- Kan vi sette `const` etter funksjonsheaderen

```
bool operator == (const Date &d) const;
```

```
bool Date::operator ==(const Date &d) const{
    return ((year == d.year) &&
            (month == d.month) &&
            (day == d.day));
}
```

Merk at vi kan ha `const` flere plasser i en funksjonsheader
Hver med forskjellig betydning

private og objekter av samme klasse

- Merk at private gjelder implementasjonen
 - Og ikke for instanser
- Funksjonskall på ett objekt kan lese/skrive medlemsvariabler til andre objekt av samme type

Vi leser medlemsvariabelene til objektet anotherDate

```
bool Date::operator ==(const Date &anotherDate) const{  
    return ((year == anotherDate.year) &&  
            (month == anotherDate.month) &&  
            (day == anotherDate.day));  
}
```



Oppgave

- Hva er problemet hvis vi ønsker å overlagre << slik at vi kan skrive:

```
Date d(1986, 12, 12);
cout << d << endl;
```

– venstresiden er ostream og høyresiden er Date

- Kan vi implementere operatoren << som medlem av Date?

NEI! Fordi riktig venstre operand er ostream, og når vi implementerer som medlem er venstre operand implisitt et objekt av klassen operanden er medlem av

Oppgave (forts)

- Gitt setningen: `cout << d;`
- Kompilatoren “leter først” etter en global operator overlagret for : `ostream& operator <<(ostream& out, const Date &d);`
- Hvis denne ikke finnes vil den se etter en medlemsoperator i klassen ostream (siden ostream er venstre operand) `ostream& ostream::operator <<(const Date &d);`
- Siden denne ikke finnes (og du ikke kan legge til noe til ostream-klassen) vil du få feilmelding ved kompilering



Friend funksjoner

- Funksjoner som ikke er medlemmer av en klasse har ikke tilgang til private medlemsvariabler
- For overlagring av +, - etc kan vi bruke public get-metoder
 - ikke bestandig egnet
 - ikke videre effektivt
- I stedet kan vi deklarere metoder som er “venner” av klassen
 - metoder som er venn (friend) med klassen får eksklusiv tilgang til private variabler



Friend funksjoner

- Friend funksjoner er vanlige funksjoner
 - Er ikke medlemsfunksjoner
 - Men har likevel tilgang til private variabler
- Defineres med nøkkelordet *friend* før funksjonsdeklarasjonen
- Listes sammen med medlemsfunksjoner i klassedeklarasjonen
- syntaks helt som for ikke-medlem operatorer

Friend eksempel

```
class Date{
private:
    int year;
    int month;
    int day;
public:
    Date();
    Date(int y, int m, int d);
    bool operator == (const Date &d) const;

    friend ostream& operator<<(ostream& out, const Date &d);
};
```

```
ostream& operator <<(ostream& out, const Date &d){
    out << setfill('0') << setw(4) << d.year << "-";
    out << setfill('0') << setw(2) << d.month << "-";
    out << setfill('0') << setw(2) << d.day;
    return out;
}
```

Deklarert som venn av klassen ved at funksjonen blir listet opp inne i klassen med friend-nøkkeordet, men merk at det er en global funksjon (som er implementert utenfor klassen - uten bruk av klassen som scope)



Bruk av friend funksjoner

- For det meste brukt til overlagring av operatorer
 - Øker effektiviteten siden vi kan unngå metodekall og lese variabler direkte, forenkler programmeringen litt
- NB! Kun en metode for å overstyre public/private –mekanismen.
- Men i prinsippet kan alle slags funksjoner være venner...

Også klasser kan være venner av hverandre...



- En klasse kan deklarereres som venn av en annen
 - hvis klasse F er venn av klasse C
 - så er alle medlemsfunksjoner i F venn med C
- Syntax: **friend** class F

```
class C {
    ...
    friend class F;
    ...
};
```

```
class F {
    ...
};
```



Når er det nyttig å overlagre?

- Overlagre kun operatorer som er meningsfulle
- Gitt en klasse for datoer er det ikke alle operatorer som er meningsfulle å bruke
 - `<`, `>`, `==` osv. er nyttige og har en "logisk mening"
 - `<<` er nyttig brukt sammen med `cout`
 - `++`, `--` tja? gitt at klassen er i stand til å endre verdi uten å ende opp med en ugyldig dato

`(dato + dato)` eller `!dato` er derimot heller meningsløst!

mens `(dato - dato)` kunne vært relevant hvis vi var interessert i antallet dager mellom to datoer



Oppsummering

- De fleste operatorer kan overlages
- Noen av disse er svært hendige å kunne bruke i forbindelse med egendefinerte typer
 - aritmetiske operatorer (+, -) , boolske operatorer (==), ++, --, <<
 - andre operatorer kan også være viktige, men mer for spesielle tilfeller
 - å kunne overstyre tilgangsmodifikatorer vha. friend deklarasjoner er nyttig men brukes mest for operatorer
- const og & er noe vi må kjenne til når vi implementerer operatorer
 - MEN BRUKES PÅ SAMME MÅTE FOR ANDRE FUNKSJONER OGSÅ!