

Debugging const with classes assignment with objects

OOP – Lektion 3

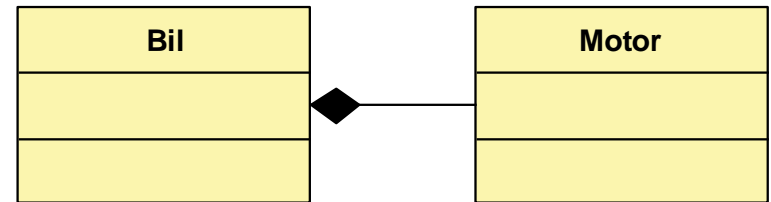
**Først lidt repetition
fra sidste gang**

Klasserelationer

- ▶ Der findes 4 typer klasserelationer
 - Komposition
 - Aggregering
 - Association
 - Arv

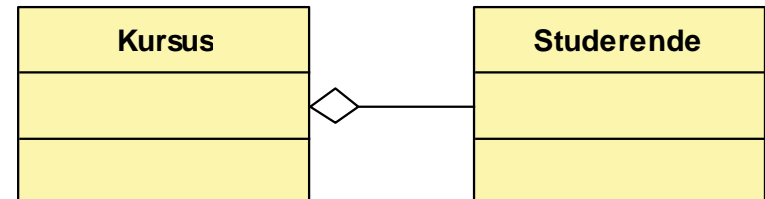
Komposition

- Er en "*har* en/et"-relation (eller "består af") *med* ejerskab
- Eksempel:
 - En Bil *har* en Motor
 - En Bil *består af* en Motor
 - En Motor *er en del af* en Bil
- Bilen *ejer* motoren – motoren kan ikke bruges af andre biler
- De har samme levetid – når bilen nedlægges, nedlægges motoren også
- Komposition er den stærkeste relation
- Implementeringen kender du fra 1. semester ("Komposition.pdf")



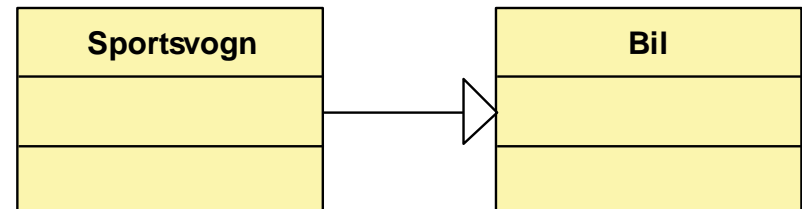
Aggregering

- Er en "*har* en/et"-relation (eller "består af") *uden* ejerskab
- Eksempel:
 - Et Kursus *har* en Studerende
 - Et Kursus *består af* Studerende
 - En Studerende *er en del af* et Kursus
- Kurset *ejer ikke* den studerende – en studerende kan deltage i flere kurser
- De har *ikke* samme levetid – hvis kurset nedlægges, nedlægges den studerende ikke
- Aggregering er en svagere relation end komposition
- Implementeringen foretages vha. pointere – du kan se det i dokumentet "Aggregering.pdf"



Arv

- En "**er** en/et"-relation
- Eksempel: En Sportsvogn **er** en Bil



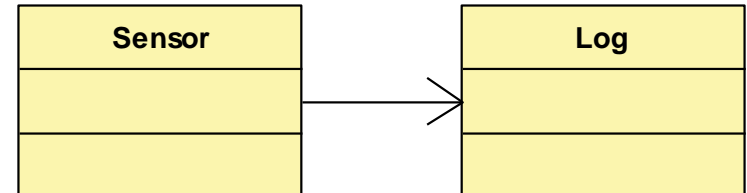
- Lærer du mere om senere i dette kursus

Association – 1

- En "anvender/bruger/aflæser/osv."-relation

- Eksempel:

- En Sensor *skriver til* en Log
- Et Sensor *består ikke* af en Log
- En Log er *ikke* en *del af* en Sensor



- Sensoren *ejer ikke* logen – flere sensorer kan skrive til logen
- De har *ikke* samme levetid – hvis sensoren nedlægges, nedlægges logen ikke
- Association er en svagere relation end aggregering

Association – 2

- Flere eksempler:
 - En Sensor *skriver* til en Log (andre kan skrive til samme log)
 - En Monitor *anvender* et Videokamera (billedet fra kameraet kan vises på flere monitorer)
 - En Tid *vises* på et Display (der kan vises andre info på displayet – f.eks. Dato)
- Altså.....der er *ingen* ejerskab ved association – og objekterne har *ikke* samme levetid !!!
- Det er en svag relation

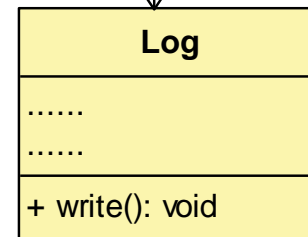
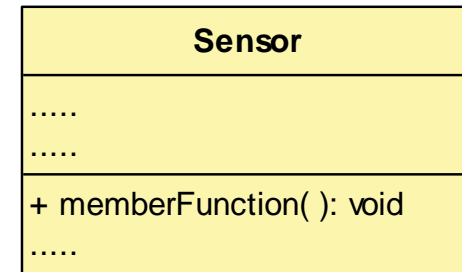
Association – 3

- ▶ I modsætning til komposition er Log objektet derfor *ikke* medlem af Sensor klassen
- ▶ Log objektet og relationen til Sensor objektet oprettes derfor *ikke* automatisk
- ▶ Derimod er en Log *pointer* medlem af Sensor klassen
- ▶ *Begge* objekter oprettes eksplicit (af dig) i koden (kunne være i main()) og relationen skabes via en parameter til constructoren som bruges til at initialisere member–pointeren

Association – 4

► Eksempel:

```
class Sensor
{
public:
    Sensor( Log * );
    .
    .
private:
    Log *myLogPtr_;
    .
};
```



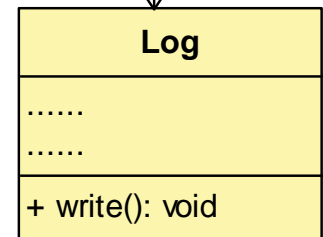
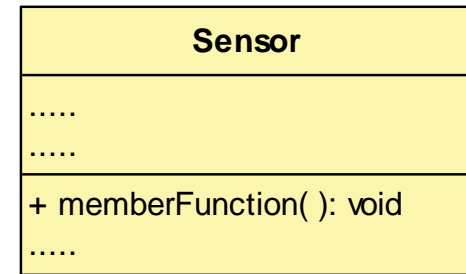
association

Association – 5

► Eksempel – fortsat:

```
Sensor::Sensor( Log *LPtr )  
{  
    myLogPtr_ = LPtr;  
}
```

```
void Sensor::memberFunction()  
{  
    myLogPtr_ -> write();  
}
```



Kald af metode
via POINTER

Skaber forbindelsen
Til Log-objektet

Association – 6

▶ NB!

- Dot-operatoren kan *kun* bruges sammen med et *objekt*
- Da myLogPtr *ikke* er et objekt men en *pointer* til et objekt kan du *ikke* skrive

```
myLogPtr_.write() // Error!!!
```

- Derimod er "det som pointeren peger på" et objekt. Du er derfor nødt til at skrive

```
(*myLogPtr_).write()
```

- *Men...* der findes en speciel operator der *erstatte* dette:

```
myLogPtr_->write()
```

Association – 7

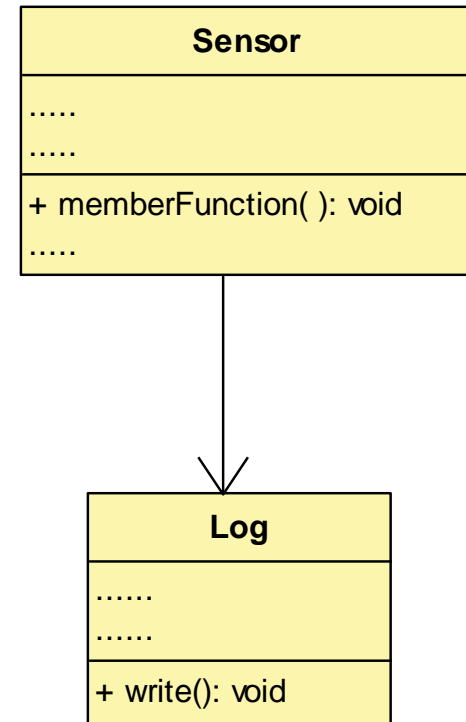
► Eksempel – fortsat:

```
int main()
{
    Log myLog;
    Sensor mySensor( &myLog );

    .
    .
    .

    return 0;
}
```

Adressen på Log-objektet sendes til constructoren



Dagens lektion

Debugging
const with classes
assignment with objects

Debugging – 1

- ▶ Debugging er et *meget* nyttigt redskab til
 - at forstå eksekveringen af et program
 - at lokalisere run-time fejl i programmer
 - det kan være konkrete fejl, så som at programmet crasher
 - eller at programmet ikke "gør som forventet"
- ▶ Du kan *ikke* skrive programmer professionelt uden at anvende debugging!!!
- ▶ Du lærer at debugge ved at gøre det.....OG du lærer af at debugge
- ▶ Du *skal* derfor bruge debugging fra nu af
- ▶ Alle hjælpelærere er informeret om, at du *skal* debugge ved run-time fejl *inden* du spørger os

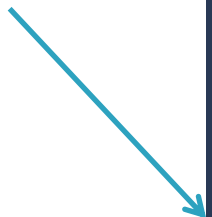
Debugging – 2

- ▶ Princippet er, at du indsætter et breakpoint ud for den kodelinie, hvorfra du vil begynde at debugge:
 - fra det sted hvor du vil forstå programmet
 - før det sted, hvor du ved/tror at fejlen opstår
- ▶ Herefter startes debugging
- ▶ Programmet eksekverer så frem til den linje hvor du har sat dit breakpoint
- ▶ Herfra single-stepper du gennem programmet – dvs. at du eksekverer *en* linie ad gangen
- ▶ Du kan herved følge alle variables værdier og hvordan de ændres undervejs

Breakpoint

- ▶ Breakpoint indsættes ved at klikke yderst til venstre

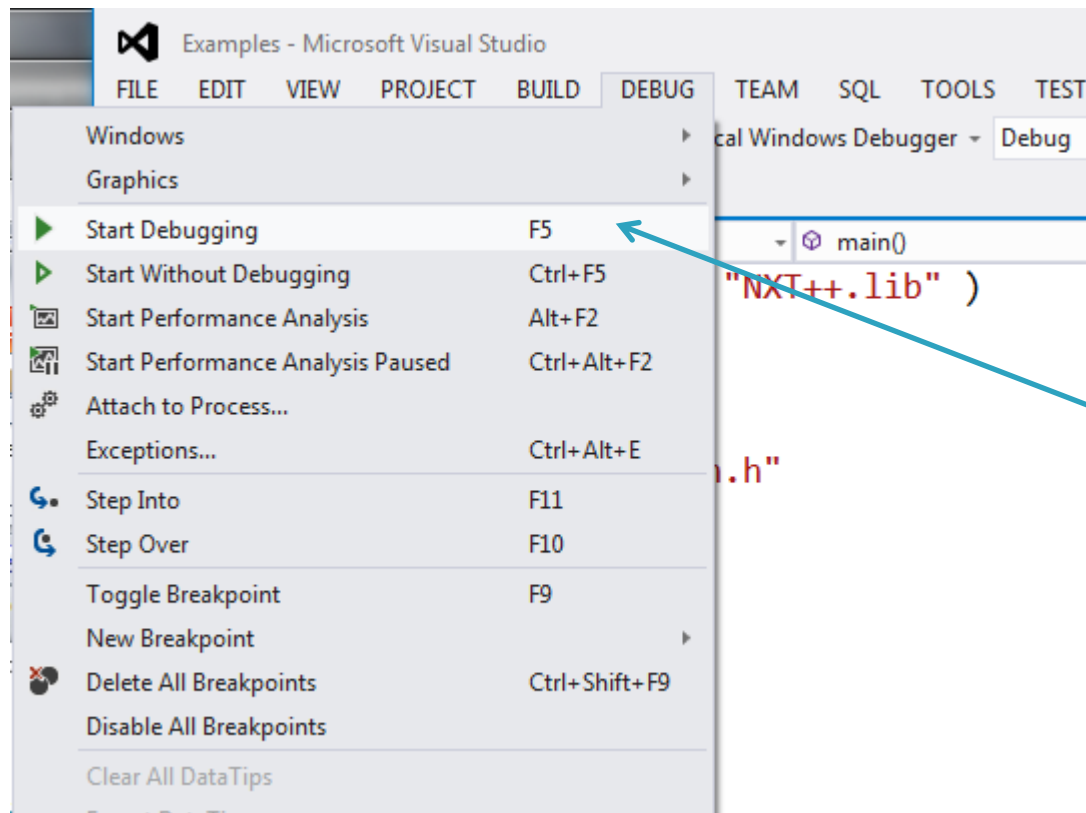
Klik her



```
26         string2 = "kort ",
27         string3 = "laaang ",
28         string4 = "tekststreng";
29
30     cout << "Size of tekst: " << text.size() << endl;
31     cout << "Size of streng1: " << string1.size() << endl;
32
33     text = string1 + string2 + string4;
34
35     cout << endl << text << endl;
36     cout << "Size of tekst: " << text.size() << endl;
37
38     cout << endl << text.substr( 6, 5 ) << endl;
```

Start debugging

- ▶ Vælg "Debug" og "Start debugging" eller tryk F5



Klik her

Single-step koden – 1

- ▶ Programmet eksekverer så al kode hertil:

```
string2 = "kort ",  
string3 = "laaang ",  
string4 = "tekststreng";
```

```
cout << "Size of tekst: " << text.size() << endl;  
cout << "Size of streng1: " << string1.size() << endl;
```

```
text = string1 + string2 + string4;
```


```
cout << endl << text << endl;  
cout << "Size of tekst: " << text.size() << endl;
```

```
cout << endl << text.substr( 6, 5 ) << endl;
```

Hertil
(gul pil)

Single-step koden – 2

- ▶ Hvert tryk på F10 vil nu eksekvere *en* linje



```
string3 = "laaang ",  
string4 = "tekststreng";  
  
cout << "Size of tekst: " << text.size() << endl;  
cout << "Size of streng1: " << string1.size() << endl;  
  
text = string1 + string2 + string4;  
  
cout << endl << text << endl;  
cout << "Size of tekst: " << text.size() << endl;  
  
cout << endl << text.substr( 6, 5 ) << endl;
```

Et tryk på
F10 fører
hertil

Single-step koden – 3

- ▶ Samtidig kan du i vinduet "Locals" følge med i hvilke værdier dine variable har

Denne har ikke fået en værdi endnu

```
text.replace( 12, 5, string3 );  
cout << endl << text << endl;  
cout << "Size of tekst: " << text.size() << endl;
```

Denne har lige fået ny Værdi (rød)

Locals

Name	Value	Type
std::basic_string<char>	"Dette er en laaang tekststreng"	std::basic_string<char>
string3	"laaang "	std::basic_string<char>
myName	<Error reading characters of string.>	std::basic_string<char>
string1	"Dette er en "	std::basic_string<char>
string4	"tekststreng"	std::basic_string<char>
string2	"kort "	std::basic_string<char>
text	"Dette er en laaang tekststreng"	std::basic_string<char>

Locals

Single-step koden – 4

- ▶ Du har følgende muligheder når du single-stepper:
 - **Step over – F10**
 - Eksekverer en programlinje uden at gå ind i eventuelle funktioner der kaldes i linjen (funktionerne eksekveres)
 - **Step into – F11**
 - Eksekverer ikke linjen men hopper ind i den første funktion, som kaldes i linjen (pas på hvis der kaldes biblioteksfunktioner og/eller operatorer i linjen !!!)
 - **Step out of – shift + F11**
 - Hopper ud af den funktion, den er i lige nu og tilbage til den linje, hvor funktionen blev kaldt

Eksempler

eksempel – string.pdf

eksempel – run-time fejl 1.cpp

eksempel – run-time fejl 2.cpp

const med klasser/objekter

- ▶ `const` anvendes 5 steder i fbm. klasser/objekter
 - `const` parametre i metoder
 - `const` returværdi fra metoder
 - `const` metoder
 - `const` objekter
 - `const` attributter

const parametre i metoder

- ▶ Som nævnt tidligere anvender man ***const parametre*** hvis den information, der "sendes" til metoden, skal skrive beskyttes (read-only)

- Eksempler:

```
bool myFunction( const int *locPtr );
```

```
int myOtherFunction( const Sensor &locSensor );
```

- Giver **const** mening her:

```
bool myThirdFunction( const int locVar ); ?
```

Svar: Nej det giver ***ikke*** mening. Dette er call-by-value – altså, det er en ***kopi*** af et tal der modtages, så der tilgås ***ikke*** noget hukommelse, som er allokeret et andet sted.

const returværdi fra metoder

- ▶ Dette vil du møde i forbindelse med operator-overloading – hvor du også kommer til at se meget mere til const parametre
- ▶ Men.....**HVIS** du får brug for at returnere en pointer (eller reference) til en privat member i en klasse **SKAL** du gøre det med const returtypeellers er den jo ikke privat !!!!!

const metoder

- ▶ Dette lærte du allerede om på 1. semester
 - Men du lærte ikke hvorfor – det gør du nu ☺
- ▶ **ALLE** metoder som ikke kan/må ændre på objektets tilstand – dvs. på attributternes værdier **SKAL** erklæres const
 - Fordi du fremover får brug for const objekter...og...
 - "almindelige" (ikke-const) objekter kan kalde *alle* public metoder...men...
 - const objekter kan *kun* kalde public *const* metoder

Giver dette mening?

Ja, i høj grad. Et const objekt er jo netop konstant – så det ville være meningsløst, hvis det var muligt, at kalde en metode, der ændrer på objektets tilstand (attributter).

const objekt

- ▶ Dette er selvfølgelig et objekt, som er erklæret const

- ▶ Eksempler:

- konstant variabel:

```
const int myInt = 9;
```

- Konstant objekt:

```
const Date birthday( 12, 3, 1973 );
```

Eksempel

Time - ver1.0

const attributter – 1

- ▶ Dette er selvfølgelig en attribut (medlemsdata), som er erklæret const

```
class MyClass
{
public:
    MyClass( int, int );    //constructor
    .
    .
private:
    int x_;
    const int y_;
};
```

const attributter – 2

- ▶ En konstant variable SKAL initialiseres

```
const int myInt;                // ERROR !!!
```

```
const int myInt = 9;           // Correct
```

- ▶ Dette kan gøres på to måder for en const attribut
 - I klassedefinitionen (ligesom ovenfor) – i så fald får attributten *samme værdi* i alle objekter
 - Med en *member initializer*, som kaldes af constructoren *inden* constructoren eksekveres – i så fald kan attributten tildeles *forskellige værdier* i forskellige objekter
- ▶ NB! Du brugte også en member initializer i forbindelse med klasserelationen komposition

const attributer – 3

- ▶ Eksempel – samme værdi i alle objekter:

```
class MyClass
{
public:
    MyClass( int, int );
    .
    .
private:
    int x_;
    const int y_ = 7;
};
```

Denne const attribut
får samme værdi i
alle objekter



const attributer – 4

► Eksempel:

```
MyClass::MyClass( int x, int y ):y_(y)
{
    x_ = x;
}
```

► Eller:

```
MyClass::MyClass( int x, int y ):x_(x),y_(y)
{
}
```

```
class MyClass
{
public:
    MyClass( int, int );
    .
    .
private:
    int x_;
    const int y_;
};
```

Member initializer.
y_ tildeles en værdi
når objektet oprettes

const attributer – 5

- ▶ Dvs. at member initializers *skal* bruges til alle const attributter (som skal tildeles en værdi når objektet oprettes) og de *kan* bruges til alle andre attributter.
- ▶ Det sidste gør man f.eks. ofte hvis constructoren skal gøre noget mere end blot initialisere objektet (f.eks. åbne en fil, en port eller lign.) – så holdes de to ting adskilt.
- ▶ Men hvad med validering ??????

const attributer – 5

- ▶ Validering er selvfølgelig stadig ligeså vigtigt. Hvis f.eks. x og y skal være positive, gøres således:

```
MyClass::MyClass( int x, int y )  
:y_( y>0 ? y : 0 ), x_( x>0 ? x : 0 )  
{  
}
```

- ▶ Eller:

```
MyClass::MyClass( int x, int y )  
:y_( y>0 ? y : 0 )  
{  
    x_ = ( x>0 ? x : 0 );  
}
```

Eksempler

Eksempel – const attribut 1

Eksempel – const attribut 2

Date – Employee

Assignment operator/objekter

- ▶ Hvis et objekt assignes til et andet objekt kopieres hver enkelt attribut fra det ene objekt til det andet. Dette kaldes memberwise assignment.

```
Time t1(10, 32, 45);  
Time t2;
```

```
t2 = t1;
```

Her sker dette:

```
t2.hour = t1.hour  
t2.minute = t1.minute  
t2.second = t1.second
```

- ▶ Dette skyldes, at compileren *altid* genererer en assignment operator (=) i klassen. I visse situationer virker denne assignment operator IKKE – og du vil være nødt til at lave din egen udgave.

Det vil du lære meget mere om senere 😊

Eksempler

TestClass
TestClass2

Kør de to eksempler og overvej hvorfor det første eksempel virker mens det næsten tilsvarende andet eksempel ikke virker