

# Repetition fra 1. semester

OOP – Lektion 1

# Objektorienterede sprog – 1

- ▶ Hvad kan man så i C++ som man ikke kan i C?

**MEGET !!!**

- ▶ Mange (men ikke alle) af de ekstra og smarte features vil I møde i dette semester
- ▶ *Først og fremmest* kan koden struktureres MEGET bedre i et objektorienteret sprog
- ▶ Princippet hedder *indkapsling* og implementeres vha. begreberne *klasser* og *objekter*

# Objektorienterede sprog – 2

- ▶ Hvad var det nu begreberne betød?
  - Indkapsling ?
  - Information hiding?
  - Klasse ?
  - Objekt ?

# indkapsling

- ▶ I C++ kan vi med klasser indkapsle data *og* funktionalitet – en struct med benefits 😊!

```
class Rectangle
{
public:
    Rectangle();
    Rectangle( double, double );
    void setSides( double, double );
    double area() const;
    double circumference() const;
private:
    double sideA_;
    double sideB_;
};
```

Dette kode skrives  
I header-filen  
Rectangle.h

# class – object

- ▶ En **class** er en erklæring om hvordan vores **data type** skal opføre sig
- ▶ **class** == blue print == en tegning
- ▶ Et **objekt** er en variabel, en parameter el. lign der opfører sig sådan som vi har beskrevet
- ▶ **object** == et færdigt hus
- ▶ Først når vi laver et **objekt** – kaldet en **instans** af klassen – afsættes der hukommelse og vi kan kalde member funktioner

# Objektorienterede sprog – 3

- ▶ I objektorienterede sprog kan man altså vha. klasser:
  - Definere sine *egne* datatyper (Rektangel, Motor, Sensor, Person osv. osv.).
  - Definere den funktionalitet der skal hører til vores nye datatype.
  - Strukturere sin kode langt bedre.
- ▶ Det skal lære i meget mere om i dette kursus

# Definition af en klasse – 1

## ► Klassens "skelet":

```
class Circle  
{
```

```
    // Heri defineres de variable som definerer en  
    // cirkel og den funktionalitet vi ønsker at  
    // tilknytte
```

```
};
```

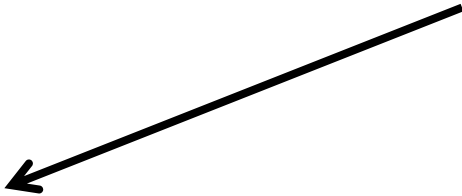
# Definition af en klasse – 2

## ► Klassens medlemsdata/attributer

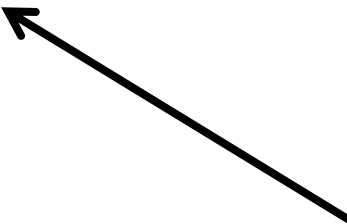
```
class Circle  
{
```

```
private:  
    double radius_  
};
```

Disse er ALTID  
private (skjulte)



Dette kaldes en  
medlemsdata  
eller en attribut





# Definition af en klasse – 3

## ► Klassens medlemsfunktioner/metoder

```
class Circle
{
public:
    double getRadius( );
    void setRadius( double );

private:
    double radius_;
};
```

Disse er normalt offentlige (kan være private)

Dette kaldes medlemsfunktioner eller metoder

# Definition af en klasse – 4

## ► Klassens constructors

```
class Circle
{
public:
    Circle();
    Circle( double );
    double getRadius( );
    void setRadius( double );
private:
    double radius_;
};
```

Dette er en  
default constructor

Dette er en  
explicit constructor

Constructorer hedder  
**ALTID** samme som  
klassen og er offentlige

# Definition af en klasse – 5

## ► Eller:

```
class Circle
{
public:
    Circle( double = 1 );
    double getRadius( );
    void setRadius( double );
private:
    double radius_;
};
```

Dette er en  
kombineret  
default/specifik  
constructor



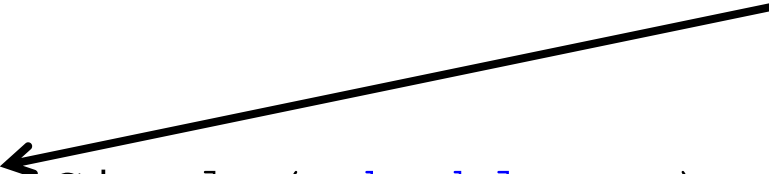
Nu er klassen  
defineret

Dette kode skrives  
I header-filen Circle.h

# Implementering af en klasse – 1

- ▶ Dette gøres i source-filen Circle.cpp
  - Constructoren:

Dette er  
vigtigt !!!



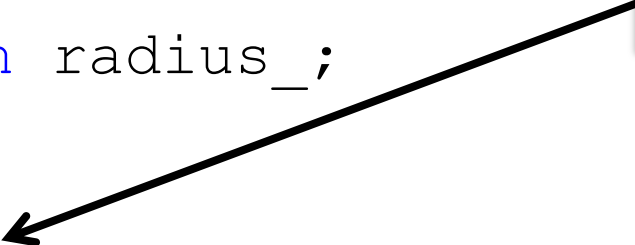
```
Circle::Circle( double r )  
{  
    if( r > 0 )  
        radius_ = r;  
    else  
        radius_ = 1;  
}
```

# Implementering af en klasse – 2

De øvrige metoder:

```
double Circle::getRadius( )  
{  
    return radius_;  
}
```

Her er det igen!!!



```
void Circle::setRadius( double r )  
{  
    if( r > 0 )  
        radius_ = r;  
    else  
        radius_ = 1;  
}
```

# Objekter – initialisering

- ▶ Der er flere måder at oprette og initialisere objekter på
  1. Default constructor
  2. Eksplicit constructor med parameterliste
  3. Eksplicit constructor med initialiseringsliste
  4. ... og flere andre, som vi først ser på senere

# Objekter – initialisering

## ► Default constructor

```
#include "Circle.h"
```

```
int main()
```

```
{
```

```
    // Default constructor  
    Circle c1;
```

```
    // Dette er en funktionsprototype!  
    Circle funktion();
```

Default constructor  
har ingen parenteser  
og parameterliste!

Dette er ikke et objekt  
skabt med default  
constructor pga.  
kompatibilitet med C!

- OBS! Tomme parenteser betyder at det er en funktion uden parametre – ikke et objekt!

# Objekter – initialisering


## ► EksPLICIT constructor med parametre

```
#include "Circle.h"
```

```
int main()
```

```
{
```

```
    // EksPLICIT constructor
```

```
    Circle c2(3.5); 
```

Objekt skabt med den eksplícitte  
constructor  
Circle(double x);



# Objekter – initialisering

- ▶ Eksplicit constructor med initialiseringsliste (C++11)

```
#include "Circle.h"
```

```
int main()  
{
```

```
    // Initialiseringsliste  
    // matcher en eksplicit constructor  
    Circle c3 = { 4.5 };
```

```
    // Alternativ notation  
    Circle c4 { 3 };
```

Disse er IKKE en struct initialisering!  
Der skal være en constructor, der  
matcher!

# Validering – 1

- ▶ "information hiding" er MEGET vigtigt
- ▶ Vores medlemsdata er "usynlige" udefra
  - (ved at erklære dem private)
- ▶ Så kan **VI** styre hvordan de tildeles værdier
- ▶ Så de **KUN** kan tildeles gyldige værdier
- ▶ Vi **SKAL** derfor VALIDERE modtagne værdier
- ▶ Dette gælder **ALLE** metoder som tildeler værdier til vores data (constructorer, set-metoder)

# Validering – 2

## ► Eksempel:

```
void Circle::setRadius( double r )
{
    if( r > 0 )
        radius_ = r;
    else
        radius_ = 1;
}
```

## ◦ Alternativ:

```
void Circle::setRadius( double r )
{
    radius_ = ( r > 0 ? r : 1 );
}
```



# Validering – 3

## ► Eksempel:

```
Circle::Circle( double rad )  
{  
    if( rad > 0 )  
        radius_ = rad;  
    else  
        radius_ = 1;  
}
```

## ◦ Alternativ:

```
Circle::Circle( double rad )  
{  
    setRadius( rad );  
}
```

# Forskellige typer metoder – 1

## ▶ Constructors

- Metoder som kun kaldes når et objekt erklæres – constructors kaldes automatisk.
- Vi har set på default- og explicit-constructors
- **VALIDERING!**

## ▶ Destructors

- Disse kaldes ligeledes automatisk når et objekt nedlægges – hører I nærmere om i dette kursus

## ▶ Mutators

- Metoder som modificerer/ændrer på værdierne af medlemsdata
- Eksempel: alle set-metoder
- **VALIDERING!**

# Forskellige typer metoder – 2

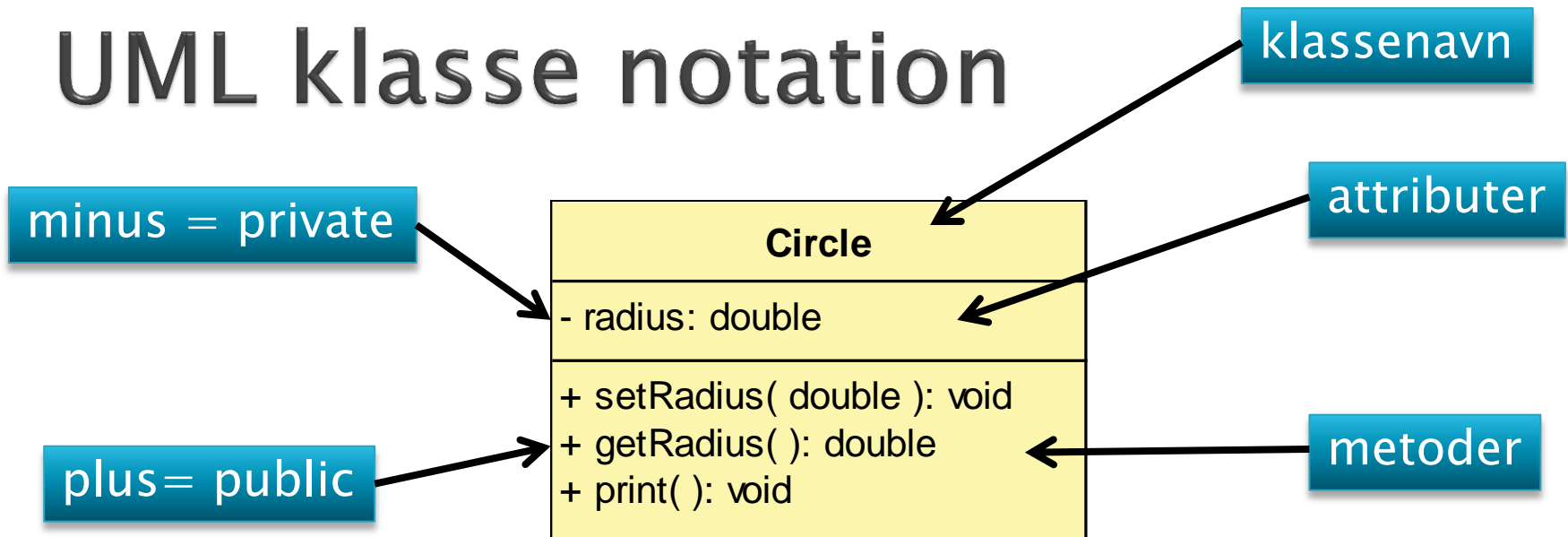
## ▶ Accessors

- Metoder som kun læser værdier af medlemsdata
- Eksempel: alle get-metoder og print-metoder
- Disse skal **ALTID** defineres som **const** metoder!

## ▶ Utilities

- Hjælpe-metoder til andre metoder
- Disse er normalt private, da de kun bruges af de andre metoder

# UML klasse notation



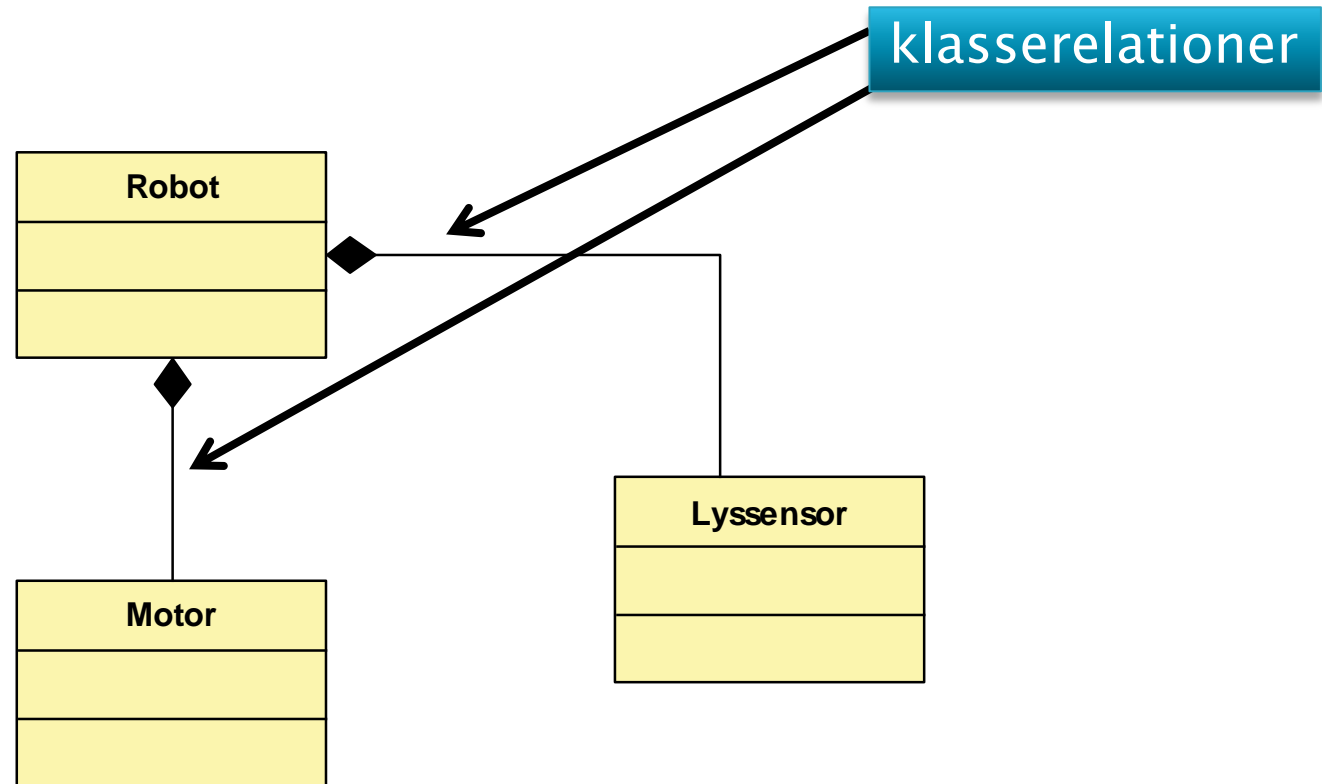
- ▶ **Bemærk:** constructorer vises normalt *ikke* i klassenotationen – fordi *alle* klasser har *minimum* en constructor
- ▶ **Men:** constructorer *skal* beskrives i den efterfølgende klassebeskrivelse

# Eksempel

Time
- hour: int - minute: int - second: int
+ setTime( int, int, int ): void + print( ): void



# UML klasse diagram



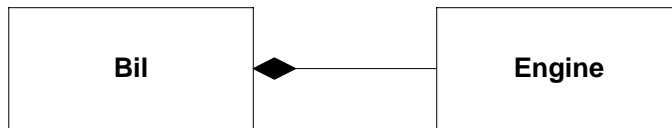
- ▶ De her viste relationer kaldes *komposition*

# Klasserelationer

- ▶ De 3 vigtigste typer klasserelationer
  - **Komposition**
    - En "**har** en/et/flere"– relation med ejerskab
    - Eksempel: En Bil **har** en Motor
  - **Association**
    - En "**anvender/aflæser/osv.** en/et/flere"–relation
    - Eksempel: En Sensor **skriver til** en Log
  - **Arv**
    - En "**er** en/et"–relation
    - Eksempel: En Sportsvogn **er** en Bil
- ▶ Og en 4. klasserelation
  - **Aggregering**
    - En "**har** en/et/flere" – relation uden ejerskab
    - Fx. Et Kursus **har** flere Studerende

# Relationer og deres symboler

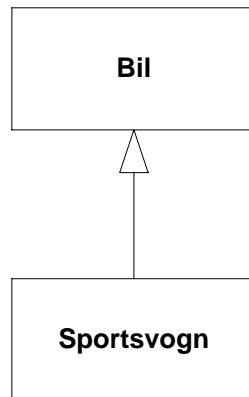
Komposition



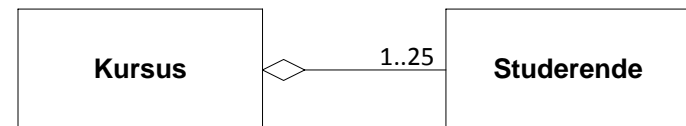
Association



Arv



Aggregering



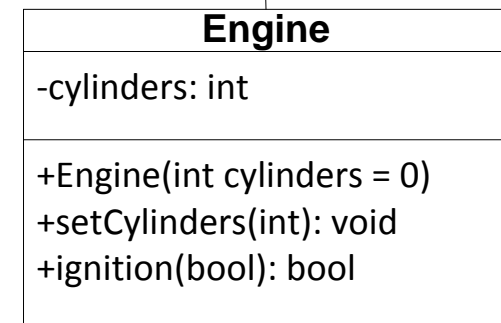
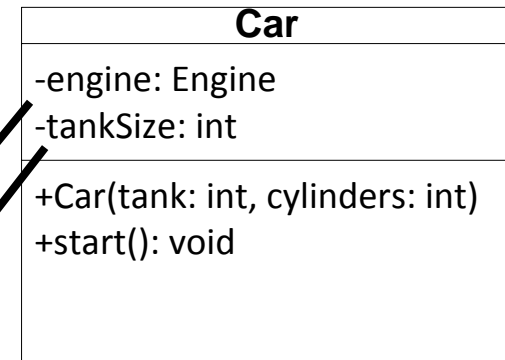
# Komposition – 1

- ▶ Komposition er en "*har* en/et/flere"-relation
  - Eksempler:
    - En Bil *har* en Motor
    - En Cirkel *har* et Punkt (som centrum)
    - Et Kontrolpanel *har* en (eller flere) Knap
- ▶ Komposition implementeres præcis som "forventet"

# Komposition – 2

## ► Eksempel:

```
class Car
{
public:
    Car( int ts=0, int cyl=0 );
    void start();
private:
    Engine engine_;
    int tankSize_;
};
```



komposition

# Eksempel

CarEngine files

# Eksempel

Point-Circle-Trouble

# Eksempel

Point-Circle-Initializer



# Klasser – initialisering

- ▶ Der er 3 måder at initialisere memberdata
  1. Explicit assignment i constructor
  2. Member initialiseringsliste i constructor
  3. in-class initialization i klasseerklæring
- ▶ i faldende prioritet!
- ▶ Disse kan også bruges til at initialisere **memberobjekter!**

# Klasser – initialisering

- ▶ Assignment i constructor

```
class Circle
{
public:
    Circle();
    Circle(double x, double y, double r);

private:
    double x_, y_; //center
    double radius_;
};

Circle::Circle()
{
    x_ = 0.0;
    y_ = 0.0;
    radius_ = 1.0;
}

Circle::Circle(double x, double y, double r)
{
    x_ = x;
    y_ = y;
    radius_ = (r > 0 ? r : 1.0);
}
```

# Klasser – initialisering

- ▶ Member initialiseringsliste i constructor

```
class Circle
{
public:
    Circle();
    Circle(double x, double y, double r);

private:
    double x_, y_; //center
    double radius_;
};

Circle::Circle()
: x_(0.0), y_(0.0), radius_(1.0)
{
}

Circle::Circle(double x, double y, double r)
: x_(x), y_(y), radius_(r > 0 ? r : 1.0)
{
}
```

Member initializer list



Skriver inden selve koden udføres!

# Klasser – initialisering

## ► in-class initialization (C++11)

```
class Circle
{
public:
    Circle();
    Circle(double x, double y, double r);
```

```
private:
    double x_ = 0, y_ = 0; //center
    double radius = 1.0;
};
```

```
Circle::Circle()
{
}
```

```
Circle::Circle(double x, double y, double r)
: x_(x), y_(y), radius_(r > 0 ? r : 1.0)
{
}
```

in-class initialization

Default constructor  
behøver ikke at gøre  
noget

Ingen ændringer her

Sker inden member initialization!

# Objekter – initialisering ved komposition

- ▶ Member initialiseringsliste i constructor
- ▶ Eneste/bedste mulighed ved objekt members uden default constructor

```
class Circle
{
public:
    Circle(double x, double y, double r);

private:
    Point center_;
    double radius_;
};
```

```
class Point
{
public:
    Point(double x, double y);

private:
    double x_,y_;
}
```

```
Circle::Circle(double x, double y, double r)
:
center_(x, y),
radius_((r > 0 ? r : 1.0))
{
}
```

# Eksempel

Point-Circle-Initializer

# const objekter og metoder

- ▶ **const** metoder har betydning for **const** objekter
- ▶ Et **const** objekt:
  - ▶ `const Dato Nytaar(1, 1, 2015);`
  - ▶ `Nytaar.setDato(1, 1, 2016);`
  - ▶ `// Forbudt - setDato er IKKE en`  
`// const metode !!`
- ▶ `cout << "Nytaar er " << Nytaar.getDag()`  
`<<...;`
- ▶ `// OK - for getDag er en const metode`

# const parametre og metoder

- ▶ const metoder har også betydning for:
  - readonly call-by-reference parametre
- ▶ Funktionsprototype:  
`int erEnsDatoer(const Dato *, const Dato *);`
- ▶ Implementation:  

```
int erEnsDatoer(  
    const Dato *Dato1Ptr,  
    const Dato *Dato2Ptr)  
{  
    return Dato1Ptr->getDag() == Dato2Ptr->getDag()  
        &&  
        ...;  
}
```
- ▶ `// OK, fordi getDag er const metode.`



# Default argumenter – 1

- ▶ Alle funktioner kan tildeles default argumenter.
- ▶ Dette gøres i funktionens *prototype* – og *kun* der.
- ▶ Eksempel:

- Prototype:

```
void myFunction( string = "Empty", int = 1);
```

- Implementering:

```
void myFunction( string s, int x )  
{  
    .  
    .  
}
```

# Default argumenter – 2

- Dette betyder, at funktionen kan kaldes således:

```
myFunction( "Hej", 7 );
```

- eller således:

```
myFunction( "Hej med dig" );
```

- eller således:

```
myFunction( );
```

- men *ikke* således:

```
myFunction( 3 );
```

Hvorfor ikke?

**Fordi default parametre starter bagfra!**

# Default argumenter – 4

- ▶ Dette kan bl.a. bruges til at lave en kombineret default- og eksplicit defineret constructor

- ▶ Eksempel:

- Prototyperne:

```
Time( );
```

```
Time( int, int, int );
```

- ▶ Erstattes af:

```
Time( int=0, int=0, int=0 );
```

# Default argumenter – 5

- Implementeringerne:

```
Time::Time( )  
{  
    setTime( 0, 0, 0 );  
}
```

```
Time::Time( int h, int m, int s )  
{  
    setTime( h, m, s );  
}
```

- "Erstattes" af (m.a.o. default-constructoren slettes):

```
Time::Time( int h, int m, int s )  
{  
    setTime( h, m, s );  
}
```

# Default argumenter – 6

- ▶ Dette betyder, at objekter kan oprettes således:

```
Time t1;                // default objekt 00:00:00
Time t1( 7 );           // 07:00:00
Time t1( 21, 13 );      // 21:13:00
Time t1( 15, 34, 6 );   // 15:34:06
```

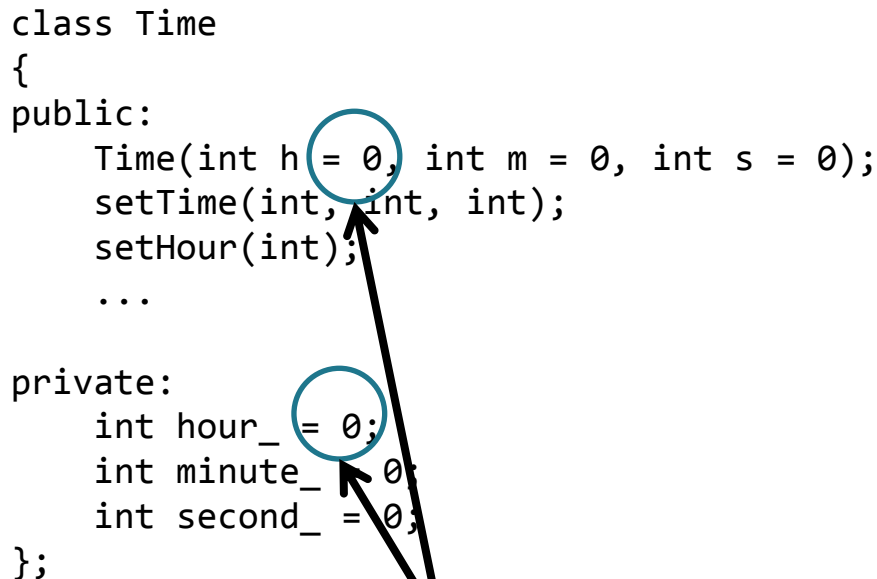
# Default værdier

- ▶ Default værdier er et specifikationsbegreb
- ▶ -- ikke et C++ begreb!
- ▶ Men vi bruger C++ mekanismer
  - initialisering
  - default constructor
  - default argumenter
  - validering i mutators

# Default værdier – samspil

```
class Time
{
public:
    Time(int h = 0, int m = 0, int s = 0);
    setTime(int, int, int);
    setHour(int);
    ...

private:
    int hour_ = 0;
    int minute_ = 0;
    int second_ = 0;
};
```

A diagram with two blue boxes. The left box contains the text "Default værdier, når der IKKE specificeres noget". Two arrows originate from this box: one points to the "0" in the public constructor "Time(int h = 0, ...)" and the other points to the "0" in the private member variable "int hour\_ = 0;".

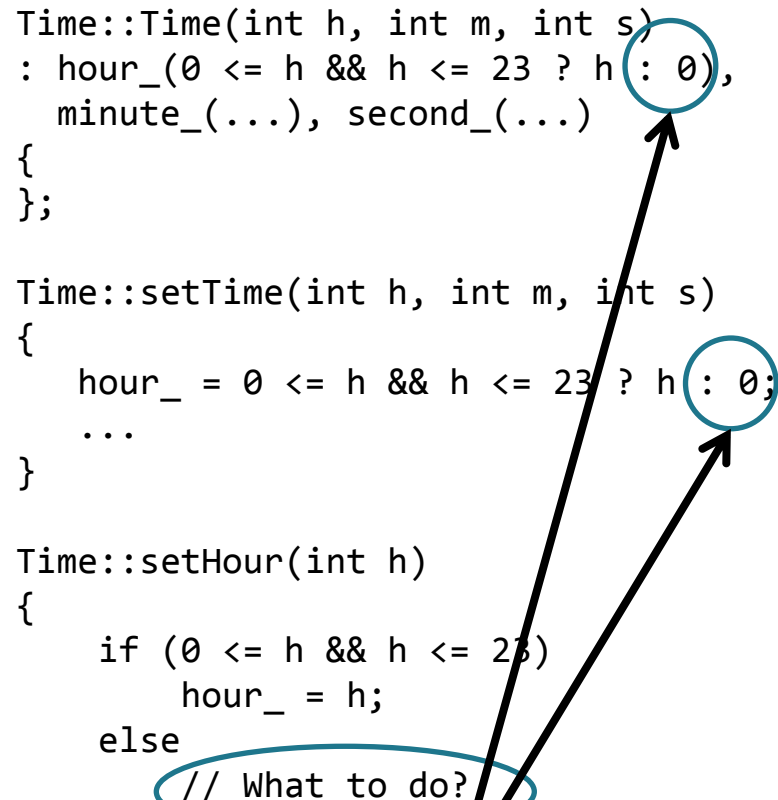
Default værdier, når der IKKE specificeres noget

SKAL de være de samme?  
Specifikation og dokumentation!

```
Time::Time(int h, int m, int s)
: hour_(0 <= h && h <= 23 ? h : 0),
  minute_(...), second_(...)
{
};

Time::setTime(int h, int m, int s)
{
    hour_ = 0 <= h && h <= 23 ? h : 0;
    ...
}

Time::setHour(int h)
{
    if (0 <= h && h <= 23)
        hour_ = h;
    else
        // What to do?
}
```

A diagram with two blue boxes. The right box contains the text "Overvej default værdier, når der specificeres noget FORKERT!". Two arrows originate from this box: one points to the "0" in the ternary expression "0 <= h && h <= 23 ? h : 0" in the Time constructor, and the other points to the "0" in the ternary expression "0 <= h && h <= 23 ? h : 0" in the setTime method. The "What to do?" comment in the setHour method is circled in blue.

Overvej default værdier, når der specificeres noget FORKERT!