

# Pointere

# Referencer

# Statiske klassemedlemmer

OOP – Lektion 4

**Først lidt repetition  
fra sidste gang**

# 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

- ▶ ***const parametre*** anvendes i fbm. – call-by-reference, 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 returtype .....ellers 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 );
```

# 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
```

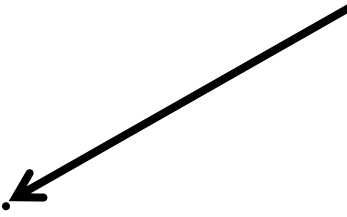
- ▶ Dette kan gøres på to måder for en const attribut
  - I klassedefinitionen (ligesom ovenfor) – i så fald får attributten *samme værdi* i alle objekter
  - Med en *member initializer*, som kaldes af constructoren *inden* constructoren eksekveres – i så fald kan attributten tildeles *forskellige værdier*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

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

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

Member initializer.  
y\_ tildeles en værdi  
når objektet oprettes

- ▶ Eller:

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

# 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 );  
}
```

# 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 😊

# Dagens lektion

Pointere

Referencer

Statiske klassemedlemmer

# Først lidt repetition af pointere fra 1. semester



# Pointere – 1

## ► Datatyper:

- int – 16/32 bit – kan indeholde hele tal
- float – 32 bit – kan indeholde decimaltal
- double – 64 bit – kan indeholde decimaltal
- char – 8 bit – kan indeholde karakterer (ASCII)
- pointer – 32 bit – kan indeh. lageradr. (hextal)

# Pointere – 2

- ▶ MEN.....lageradressen er kun 1. byte af det den refererer til.
- ▶ DERFOR..... SKAL en pointer have en *type* (for at "vide" hvor meget den *ialt* refererer til).
- ▶ Dvs.

```
char myChar = 'a';
```

```
char *myCharPtr = &myChar;
```

# Pointere – 3

## ▶ ALTSÅ:

- i en *erklæring* betyder \* at det er en *pointer*.

```
char *myCharPtr = &myChar;
```

- andre steder betyder \* "*det som pointeren peger på*".

```
cout << *myCharPtr << endl;
```

```
char c = *myCharPtr;
```

# Pointere – 4

- ▶ *Denne* pointer er const og kan ikke flyttes:

```
int * const myPtr = &x;  
myPtr = &y;           // ERROR  
myPtr++;              // ERROR
```

- ▶ Her er det pointeren referere til const og kan ikke ændres via pointeren (read-only):

```
const int *myPtr = &x;  
*myPtr = 4;           // ERROR
```

- ▶ Her er *begge dele* const:

```
const int * const myPtr = &x;
```

# Pointere – 5

- ▶ Hvad bruges pointere til?
  - Til at "sende" information til en funktion (call-by-reference) – altså som parametre i funktioner og metoder.
    - Bl.a. i forbindelse med arrays (1.semester)
  - Til at gennemløbe arrays
  - Til at implementere klasserelationen association
  - Til at implementere klasserelationen aggregering
  - Til at referere til dynamisk hukommelse

# Arrays og pointere

- ▶ NB! En pointer kan tælles "en" adresse frem med ++ operatoren.
- ▶ "en" adresse svarer automatisk til størrelsen af den type data som pointeren refererer til.
- ▶ Dvs.
  - Hvis pointeren er af typen char tælles 1 byte frem med ++
  - Hvis pointeren er af typen double tælles 8 bytes frem med ++
  - Osv.

smart ☺

# Funktionskald – "call-by-value"

```
int triple( int );           // "call-by-value"-funktion
```

```
int main()
{
    int x = 4, y;

    y = triple( x );         // der sendes en KOPI af x
                             // y ændres
}
```

```
int triple( int a )         // a = 4
{
    a *= 3;                  // x ændres IKKE
    return a;                // 12 returneres
}
```

# Funktionskald – "call-by-reference"

```
void triple( int * );           // "call-by-reference"-funktion

int main()
{
    int x = 4;

    triple( &x );               // der sendes ADRESSEN af x
                                // x er ændret !!!
}

void triple( int *aPtr )       // aPtr = &x
{
    (*aPtr) *= 3;              // x ændres !!!
}
```



# VIGTIGT !!!

- ▶ Funktionen `f` ændrer altså i noget hukommelse som "ejes" af "en anden"
- ▶ Hvis den *ikke* må gøre det – altså hvis den kun må *læse* på denne hukommelsesplads *skal* man erklære parameteren for `const` (jvf. `const` parametre i sidste uge)

```
bool f( const int *locPtr )
```

`const` sikrer således "read-only-access"

# Referencer – 1

- ▶ En reference "minder om" en pointer...MEN
- ▶ En reference er *IKKE* en ny variabel
- ▶ En reference er bare et alias (nyt navn) for en allerede eksisterende variabel (lagerplads)
  - En reference optager altså *ikke* ekstra plads i hukommelsen
- ▶ Dvs. når du erklærer en reference laver du blot et *navn mere* til en eksisterende variabel
- ▶ De to navne gælder typisk i hvert sit scope !!!
- ▶ En reference er pr. default const – dvs. den refererer til det samme i hele sin levetid
  - En reference kan altså – modsat en pointer – ikke flyttes

# Referencer – 2

## ► Syntax:

```
int x = 3;
```

```
int &y = x;
```

Den lagerplads, der er allokeret til `x`, har nu ***både*** navnet `x` ***og*** navnet `y`

Dvs. at følgende programlinier er ***nøjagtig*** ens:

```
cout << x << endl;
```

```
cout << y << endl;
```

(men vil typisk tilhøre hvert sit scope)

# Referencer – 3

- ▶ Dette er selvfølgelig bedre:

```
int myInt = 3;
```

```
int &myIntRef = myInt;
```

Ligesom du bør tilføje Ptr bag navnet på en pointer, bør du tilføje Ref bag navnet på en reference

# Referencer – 4

## ▶ BEMÆRK:

- i en *erklæring* betyder & at det er en *reference*.

```
int &myIntRef = myInt;
```

- andre steder betyder & *adresseoperatoren*

```
char *myCharPtr = &myChar;
```

Dvs. at *både* \* og & har *to* betydninger

# Referencer – 5

- ▶ Men hvorfor har man brug for to navne til samme variabel ??????
- ▶ Fordi det ene navn bruges i et scope og det andet i et andet scope (nøjagtig ligesom ved pointere)
- ▶ Det er altså endnu en måde at tilgå samme lagerplads fra forskellige scopes
- ▶ NB! En reference er *altid* const – dvs. at den sættes til at referere til noget, når den oprettes og dette refererer den til indtil den nedlægges – den kan altså *ikke* sættes til at referere til noget andet undervejs i programmet (husk at en pointer kan "flyttes" undervejs)

# Referencer – 6

- ▶ Hvad bruges referencer til?
  - Til at "sende" information til en funktion (call-by-reference) – altså som parametre i funktioner og metoder.
    - Objekter kan være store (bruge meget hukommelse) og det er derfor ikke hensigtsmæssigt at kopiere objekter – så når et objekt skal "sendes" til en funktion gøres det *altid* med en reference
    - Dette kaldes "call-by-reference" (i modsætning til "call-by-value")
  - Til at "returnere" mere end en værdi fra en funktion

# Eksempler

Eksempel – referencer.cpp

Eksempel – call-by-ref.cpp

Eksempel – call-by-value or ref.cpp

Time – ver 1.1



# statisk attribut – 1

- ▶ Normalt har hvert objekt sine *egne* udgaver af alle klassens attributter
- ▶ Hvis man har brug for, at alle objekter *deles* om en attribut, skal man blot erklære den `static`
- ▶ Dette kan f.eks. bruges til at have en tæller, der tæller antallet af objekter, der eksisterer
- ▶ Eller til at tælle "noget andet" der er fælles for alle objekter (f.eks. summen af alle solgte enheder for alle sælgere i en virksomhed)

# statisk attribut – 2

## ► Eksempel:

```
class MyClass
{
public:
    MyClass( int );    //constructor
    .
    .
private:
    int myInt_;
    static int myStatic_;
};
```

Statisk  
attribut



# statisk attribut – 3

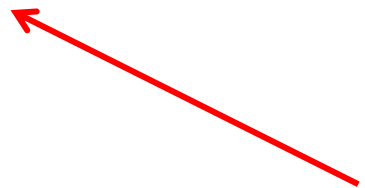
- ▶ En static attribut initialiseres i sourcefilen således (som en "privat global variabel"):

```
int MyClass::myStatic_ = 0;
```

```
MyClass::MyClass( int a )  
{  
    myInt_ = a;  
}
```

```
.  
.
```

Initialisering  
af statisk  
attribut



# statisk metode – 1

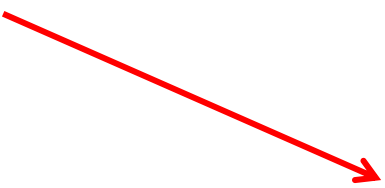
- ▶ En metode der kun skal tilgå en (eller flere) statiske attribut(ter) skal også erklæres static
- ▶ En statisk metode kan *kun* tilgå statiske attributter, da den ikke knytter sig til et bestemt objekt
- ▶ En statisk metode kan kaldes *uden* brug af et objekt
- ▶ En ikke-statisk metode kan (selvfølgelig) tilgå alle attributter – også de statiske

# Statisk metode – 2

## ► Eksempel:

Statisk metode

```
class MyClass
{
public:
    MyClass( int ); //constructor
    int getMyInt() const;
    static int getMyStatic();
    .
private:
    int myInt_;
    static int myStatic_;
};
```



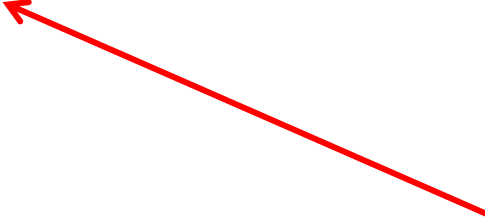
# statisk metode – 3

## ► Eksempel:

```
int main()
{
    MyClass myObj1(7), myObj2(30);

    cout << myObj1.getMyInt() << endl;

    cout << MyClass::getMyStatic() << endl;
    .
    .
    .
```



Kald af  
statisk  
metode

# Eksempel

Test static