iha.dk

C++ kodestandard

for E- og IKT-linierne på
Ingeniørhøjskolen i Århus

INDHOLD

REVISIONSHISTORIE

Rev.	Dato/Init.	Beskrivelse
1.0	09.01.2007/TFJ	Indledende revision
1.1	02.12.2013/TFJ	Rettet navngivning af variable (tilføjet '_'-prefix) + formatering.

1 Indledning

1.1 Introduktion

Dette dokument beskriver den kodestandard, der skal benyttes i C++ på IKT-linien på Ingeniørhøjskolen i Århus. Kodestandarden består af en række skabeloner og retningslinier, der skal højne læseligheden af kode og sikre en ensartet måde at skrive kode på.

Denne kodestandard er ikke udtømmende. Der findes industrielle kodestandarder, som er stort set udtømmende, men disse er uanvendelige til det formål, denne standard skal opfylde. Derfor skal der udvises sund fornuft. I tilfælde, som denne kodestandard *ikke* beskriver, skal den anvendes som retningslinie for hvordan kode skal skrives.

Denne kodestandard skal anvendes i alle tilfælde hvor der skrives C++ - i programmeringsfag og –opgaver, i semesterprojekter og i afgangsprojekter.

1.2 Hvorfor anvende en kodestandard?

At anvende en kodestandard kan synes blot at tilføje flere ting, man skal huske på, når man skriver kode – og det gør jo ikke programmet hurtigere eller mere syntaktisk korrekt. Det er for så vidt sandt, men det gør programmet *bedre*.

Tag for eksempel en review-situation, hvor en gruppes medlemmer udveksler kode indbyrdes til gennemsyn. Hvis hvert gruppemedlem har sin individuelle ”kodestandard”, vil der uværgelig blive brugt tid på at forstå, og måske endda diskutere, den anvendte kodestandard. Det betyder, at diskussionen kan komme til at dreje sig om *form* frem for *indhold* - en situation der meget ofte opstår ved reviews, og som er meget uproduktiv. Hvis gruppens medlemmer derimod anvender den samme kodestandard, vil kodens indhold træde tydeligere frem, og dermed blive emnet for diskussionen.

Situationen kan sammenlignes med læsningen af et dokument: Hvis alle afsnit er formateret ens, vil indholdet være lettere at læse, end hvis hvert afsnit er med forskellig skrifttype, indrykning, linieafstand osv.

En kodestandard tjener også til at minimere antallet af fejl: Hvis en for-løkke er specificeret (korrekt) i en kodestandard, og denne anvendes konsekvent, vil der ikke længere optræde fejl i for-løkker i ens program.

2 Generelle konventioner

Følgende konventioner er generelle for hele kodestandarden:

- Det anvendte sprog er engelsk. Det gælder for navngivning af filer, klasser, objekter etc., og for kommentarer. Der anvendes *kun* engelsk i C++ kode.

3 Filer

3.1 Fil-headers

Fil-headers er kommentar-blokke i starten af en fil, der angiver filens navn, forfatter, oprettelsesdata, beskrivelse, samt en revisionshistorie for filen. Fil-headers benyttes i *alle* filer i et givet projekt.

```
//===========================================================================
// FILENAME   : <filename>
// CREATED    : <Creation date>
// AUTHOR     : <Author>
// DESCR.     : <Description of file contents>
//
//---------------------------------------------------------------------------
//
// REV.    DATE/AUTHOR           CHANGE DESCRIPTION
// 1.0     <rev. date/author>    <Change description>
//===========================================================================
```

3.2 Navngivning

Header-filer navngives med `.h` som extension. Source-filer navngives med `.cpp` som extension. Filer skal så vidt muligt navngives efter den klasse, de indeholder. Indeholder en header- og source-fil således definitionen hhv. implementationen af klassen `Vector`, skal filerne hedde `Vector.h` og `Vector.cpp`.

Hvis en fil indeholder andet end en klassedefinition/implementation, skal filens funktion fremgå af filnavnet (for eksempel `constants.h` eller `main.cpp`).

3.3 Formatering

3.3.1 Indrykning

Indrykning anvendes til at visualisere sammenhængen i et kodefragment. Korrekt anvendt er indrykning et meget kraftfuldt værktøj til at opdage ”tuborg-fejl”, hvor en ’{’ eller ’}’ er glemt.

Der anvendes fire mellemrum som indrykning for hvert indrykningsniveau. Der startes på niveau 0, og for hver kodeblok (typisk for hver ’{’) øges indrykningsniveauet, indtil kodeblokken er slut (den tilsvarende ’}’ mødes). Eksempel herunder:

```
bool MyClass::find(int x, unsigned int size, int* pArray)
{
    for(int i=0; i<size; i++)
    {
        if(pArray[i]==x)
        {
            return true;
        }
    }
    return false;
}
```

3.3.2 Mellemrum

Som i almindelig tekst skal mellemrum (spaces) anvendes i kode i det omfang det øger overskueligheden. Eksempler på korrekt anvendelse af mellemrum er efter kommaer i parameterlister og mellem argumenter i for-løkker. Eksempler på korrekt brug af mellemrum følger herunder:

```
bool MyClass::find(int x, unsigned int size, int* pArray)
{
    for(int i=0; i<size; i++)
    {
        if(pArray[i]==x)
        {
            return true;
        }
    }

    return false;
}
```

3.3.3 Tomme linier

Tomme linier anvendes for at adskille logiske segmenter af kode, for eksempel erklæring af metode-lokale variable fra metodens kode. Anvendelsen af tomme linier skal begrænses til et omfang hvor det tjener til at fremme kodens overskuelighed. Eksempel på korrekt anvendelse af tomme linier:

```
unsigned MyClass::countOccurrencesOf(int x, unsigned int size, int* pArray)
{
    unsigned count=0;

    for(int i=0; i<size; i++)
    {
        if(pArray[i]==x)
        {
            ++count;
        }
    }

  return count;
}
```

4 Kommentarer

Der skelnes mellem strategiske og taktiske kommentarer. Strategiske kommentarer beskriver en række efterfølgende sourcekode-linier og beskriver deres overordnede funktion. Taktiske kommentarer beskriver typisk en enkelt linie sourcekode.

Kommentarer tjener til dokumentation af koden. Alle ikke-trivielle dele af kode skal dokumenteres med kommentarer – kode uden kommentarer betragtes som udokumenteret!

4.1 Strategiske kommentarer

Strategiske kommentarer placeres på egne linier før en blok af kode som indeholder en sammenhængende funktionalitet. Strategiske kommentarer beskriver funktionaliteten på overordnet niveau, ikke detaljeret – hertil benyttes taktiske kommentarer.

4.2 Taktiske kommentarer

Taktiske kommentarer placeres på samme linie som den kode, de dokumenterer, og beskriver i detaljer funktionaliteten i netop denne linie.

4.3 Kommentarers indhold

Kommentarer, strategiske som taktiske, skal kort forklare funktionaliteten af den sourcekode, de hører til.

4.4 Eksempler

```
//============================================================
// METHOD : convertPolarToCartesianCoordinates
// DESCR. : Converts polar coordinates (r, theta) to Cartesian
//          coordinates (x,y) and returns them in an XYPoint object
//============================================================
XYPoint MyClass::convertPolarToCartesianCoordinates(double r, double theta)
{
    double tempX;
    double tempY;
    XYPoint result;

    // Calculate Cartesian coordinates from polar ones
    x = r * cos(theta);  // Calculate x from r and theta
    y = r * sin(theta);  // Calculate y from r and theta
    result.set(x, y);    // Update result with calculated values

    return result;
}
```

5 Klasser

5.1 Navngivning

Klasser navngives med et navneord i ental der beskriver klassens funktion eller ansvar. Klassen navngives efter InfixCaps style, dvs. klassers navne starter med stort begyndelsesbogstav, for eksempel `Vector`, `Point` eller `Car`. Sammensatte klassenavne begynder med et stort bogstav, og hvert efterfølgende ord er med stort, for eksempel `ComputedPatientValue`.

5.2 Formatering

5.2.1 Klassedefinitioner

Definitionen af en klasse skal finde sted i en korrekt formateret og navngivet headerfil (`.h`-fil). Klasser defineres med `public`, `protected` og `private` sektioner i den rækkefølge. Enummerationer, metoder, konstanter og medlemsvariable defineres i den rækkefølge inden for hver sektion. Før klassedefinitionen placeres en klasseheader med en kort beskrivelse af klassen

Nedenfor følger et eksempel på definitionen af en klasse:

```
.
.

//====================================
// CLASS  : Car
// DESCR. : Represents a car
//====================================
class Car
{
    public:
        Car();
        Car(string model);
        void printInfo() const;
        static const unsigned int MAX_CYLINDERS;

    private:
        string makeCarString();
        string licensePlate;
        string model;
        unsigned int productionYear;
};
```

5.2.2 Klasseimplementeringer

Implementationen af en klasse skal finde sted i en korrekt formateret og navngivet source-fil (`.cpp`-fil). Klasseimplementationen indeholder instantieringen af statiske medlemsvariable og metoder, i den rækkefølge. Før hver metode placeres en metodeheader med en kort (evt. tom) beskrivelse af metoden.

```
.
.
unsigned int Car::MAX_CYLINDERS = 14;
.
.

//===============================================================
// METHOD : Default constructor
// DESCR. :
//===============================================================
Car::Car()
{
    ...
}


//===============================================================
// METHOD : Explicit constructor
// DESCR. :
//===============================================================
Car::Car(string _lp, string _m, unsigned int _p)
{
}

//===============================================================
// METHOD : printInfo
// DESCR. : Prints information pertaining to a car to console
//===============================================================
void Car::printInfo() const
{
    cout << "License plate : " << licensePlate << endl
         << "Model         : " << model << endl
         << "Prod. year    : " << productionYear << endl;
}
```

5.2.2.1 Constructorer med member initializers

Hvis en constructor anvender en member initializer liste, skal skrives efter constructorens første linie. Hvis antallet af initializers overstiger hvad der kan stå på en linie, anbringes disse på separate linier under constructorens hoved, før implementationen. Se eksempel nedenfor:

```
//============================================================
// METHOD : Default constructor
// DESCR. :
//============================================================
Car::Car() : licensePlate(“”), model(“”), productionYear(0)
{
    ...
}


//============================================================
// METHOD : Explicit constructor
// DESCR. :
//============================================================
Car::Car(string _lp, string _m, unsigned int _p)
 : licensePlate(_l),
   model(_m),
   productionYear(_p)
{
    ...
}
```

6 Objekter og variable

6.1 Navngivning

Objekter og variable navngives med et navneord, der beskriver objektets funktion. Objektet navngives efter infixCaps style, dvs. med lille begyndelsesbogstav, og prefixes med en underscore '_', for eksempel `_weight`, `_acceleration` eller `_currency`. Sammensatte objektnavne skal begynde med et lille bogstav, og begyndelsesbogstavet for hvert efterfølgende ord skal være stort, for eksempel `_initialForceVector`, `_originalPosition` eller `_myFirstCar`. Bemærk, at det *ikke* er nødvendigt at typen (f. eks. `Vector`, `Point` eller `Car`) er en del af objektets navn, men ofte er det hensigtsmæssigt at inkludere dette alligevel.

7 Pointere

Pointere navngives efter samme retningslinier som objekter og variable, bortset fra at de skal afsluttes med "`Ptr`". Operatoren '`*`' placeres ved variabelnavnet, ikke typenavnet. Eksempler på rigtig og forkert navngivning af pointere:

```
    Car *_myCar2Ptr;  // GOOD FORMAT
    Car* _pMyCar;     // BAD FORMAT
    Car *_pMyCar;     // BAD FORMAT
```

8 Konstanter

Konstanter skal anvendes i alle de tilfælde i sourcekode, hvor man ellers vil anvende tal – forekomsten af såkaldte "magic numbers" i kode er en fejl og et brud på denne kodestandard. Anvendelsen af konstanter erstatter ligeledes alle anvendelser af makroer (`#define`'s) til definition af konstanter.

8.1 Navngivning

Konstanter navngives med navneord, der beskriver konstantens værdi. Konstantens navn skrives med stort. Hvis en konstants navn er sammensat, adskilles de enkelte ord af en underscore, for eksempel `NUMBER_OF_WORK_HOURS_PER_DAY`.

8.2 Eksempel

Nedenstående vises eksempler på forkert brug af "magic numbers" og makroer:

```
#define SECONDS_PER_DAY (24*60*60)  // BAD FORMAT - DO NOT USE MACROS!

//============================================================
// METHOD : convertHrsMinsSecsToDecimalDay()
// DESCR. : Converts a number of hours, minutes, and seconds
//          into a decimal [0..1[ of a 24-hour day
//          *** NOTE: BAD FORMAT - USES MAGIC NUMBERS! ***
//============================================================
double MyClass::convertHrsMinsSecsToDecimalDay(char hrs, char mins, char secs)
{
    double tempSeconds;

    tempSeconds = secs + min*60 + hrs*60*60; // BAD FORMAT - MAGIC NUMBERS!

    return tempSeconds/SECONDS_PER_DAY;      // BAD FORMAT - MACRO USE!
}
```

Nedenstående viser, hvordan ovenstående implementeres korrekt med brug af konstanter. Disse erklæres i klassen, initialiseres i klassens constructor og anvendes i metoden:

MyClass.h

```
class MyClass
{
  ...
    const unsigned int SECONDS_PER_MINUTE;  // Definition of class constants
    const unsigned int MINUTES_PER_HOUR;
    const unsigned int HOURS_PER_DAY;
};
```

MyClass.cpp

```
//============================================================
// METHOD : Constructor
// DESCR. :
//============================================================
MyClass::MyClass()
  : SECONDS_PER_MINUTE(60),  // Initialization of constants
    MINUTES_PER_HOUR(60),
    HOURS_PER_DAY(24)
{
}


//============================================================
// METHOD : convertHrsMinsSecsToDecimalDay()
// DESCR. : Converts a number of hours, minutes, and seconds
//             into a decimal [0..1[ of a 24-hour day
//============================================================
double MyClass::convertHrsMinsSecsToDecimalDay(char hrs, char mins, char secs)
{
    double tempSeconds;

    tempSeconds = secs +
                  min * SECONDS_PER_MINUTE +
                  hrs * SECONDS_PER_MINUTE * MINUTES_PER_HOUR;

   return tempSeconds/(SECONDS_PER_MINUTE * MINUTES_PER_HOUR * HOURS_PER_DAY);
}
```

9 Metoder

9.1 Navngivning

Metoder navngives med et udsagnsord, der beskriver metodens funktion. Metoder navngives jf. infixCaps style, dvs. med lille begyndelsesbogstav, for eksempel `make()`, `print()` eller `increment()`. Sammensatte metodenavne skal begynde med et lille bogstav, og begyndelsesbogstavet for hvert efterfølgende ord skal være stort, for eksempel `makeInfoString()`, `calculateUpdatedPosition()` eller `deleteArray()`.

9.2 Argumentnavne

Argumenter til metoder navngives efter deres indhold, for eksempel temperature, `heighAboveGround` eller `course`. Navne som `temp`, `x`/`y`/`z`, i etc. må ikke anvendes, med mindre disse netop beskriver variablens indhold (for eksempel de tre komposanter af en 3D-vektor, x, y, og z).

9.3 Formatering

Metoder implementeres i sourcefilen for den klasse, metoden tilhører. Er der tale om globale funktioner, implementeres disse i en source-fil, der er navngivet passende.

Implementationen af en metode foregås altid af en metodeheader, der indeholder metodens navn og evt. en kort beskrivelse af metodens funktion. Mellem to metodeimplementationer placeres 2 tomme linier.

10 Kontrolsætninger

Kontrolsætninger skal formateres, så de er overskuelige og ideelt set selvforklarende. De følgende afsnit giver eksempler der definerer kontrolsætningers formatering.

10.1 if – else-if – else

`If`- (`else-if`, `else`) testen placeres på en linie for sig. den tilhørende kodeblok indrykket og omkranset af et {…}-sæt. Et eksempel på en korrekt formateret `if – else-if – else` sætning ses nedenfor:

```
if(index==0)
{
    cout << "index is now zero" << endl;
}
else if(index==3)
{
    cout << "index is now three" << endl;
}
else
{
    cout << "index is not zero or three" << endl;
}
```

Lange, nestede, komplicerede boolske udtryk i `if`-sætninger skal undgås – benyt i disse tilfælde midlertidige bool-variable til at holde "mellemresultater". Nedenstående er et eksempel på manglende anvendelse af midlertidlige variable:

```
// Bad format!
if(IO686.getPortA() & 0x0FF0) && (PV2019.getVal(A) < PV2019.getVal(B))  p=4;
else if(4* IO686.getPortB().getBit(3) == (1 << 3))  p=3;
else p=-1;
```

Samme som ovenfor, men med korrekt anvendelse af midlertidige variable:

```
// Good format (does the same as above)!
bool io686PortAStatus = IO686.getPortA() & 0x0FF0;
bool pv2019Status = PV2019.getVal(A) < PV2019.getVal(B);
bool io686PortBBit3Status = IO686.getPortB().getBit(3);

if(io686PortAStatus && pv2019Status)
{
    p=4;
}
else if(4* io686PortBBit3Status == (1 << 3))
{
    p=3;
}
else
{
    p=-1;
}
```

10.2 for-løkker

`For`-løkker formateres med en separat linie til initialisering/test/inkrementering og den tilhørende kodeblok indrykket og omkranset af et {…}-sæt

```
for(unsigned int i=0; i<SIZE; i++)
{
    cout << myArray[i] << endl;
}
```

10.3 switch-case

`Switch-case` sætninger formateres med en linie til `switch`-statement og de tilhørende cases indrykket og omkranset af et {…}-sæt. I hver enkelt case indrykkes den tilhørende kodeblok. Den sidste case i en blok er default casen. I de (sjældne) tilfælde hvor en case-blok *ikke* afsluttes af et break skal dette klart dokumenteres, så det er helt tydeligt at der her ikke er tale om en kodefejl.

```
switch(carType)
{
    case STATION_CAR:
        cout << "A station car" << endl;
        break;

    case MPV:
        cout << "A Multi-Purpose Vehicle (MPV)" << endl;
        break;

    case SUZUKI_SWIFT:
        cout << "A Suzuki Swift" << endl;
        // INTENTIONAL CASE FALL-THROUGH - NO BREAK INTENDED!!

    default:
        cout << "That's not a car!" << endl;
        break;
}
```

10.4 While

`While`-løkker formateres med en separat linie til `while`-testen, efterfulgt af den tilhørende kodeblok omkranset af et {…}-sæt.

```
while(io686.getPortA().getBit(4) == true)
{
    cout << "Port A bit 4 is still true" << endl;
}
```

10.5 do-while

`Do-while` sætninger formateres med en separat linie til `do`-delen, efterfulgt af den tilhørende kodeblok omkranset af et {…}-sæt. `While`-delen på samme linie som den afsluttende tuborg-parentes.

```
do
{
    cout << "Port A bit 4 is still true" << endl;
} while(io686.getPortA().getBit(4) == true);
```