Coding Conventions

for C++

Reto Carrara
Gyrenstrasse 17
CH-8967 Widen
+4179/328’17’05
www.carrara.ch
info@carrara.ch

Inhalt

Revision

Datum	Version	Autor	Änderungen
17.01.02	1.0	rc	Basisversion
15.02.02	1.1	rc	Layout

Einführung

Transparente Kodierrichtlinien sind ein Muss in jedem grossen Projekt, um die nötige Softwarequalität garantieren zu können. Die vorliegenden Coding-Conventions für C++ sind simpel aber vollständig.

Klassen- und Struct-Namen

Klassen- und Struct-Namen werden mit grossem Anfabgsbuchstaben geschrieben. Die Gross-/Kleinschreibung im Namen selber darf variieren.

Beispiel:

```
class ThisIsMyFirstClass{};
```

Underscores sind nur erlaubt wenn aus technischen Gründen Innerclasses nicht möglich waren, diese aber mit einem typedef nachgeahmt werden möchten.

```
class SList_Iterator{....};

class SList
{
  public:
    typedef SList_Iterator Iterator;
};
```

anstelle von

```
class SList
{
  public:
    class Iterator{....};
};
```

Makros

Die Anwendung von Makros ist grundsätzlich untersagt. Makros sind erlaubt, falls Pre-Processor-Informationen wie __FILE__, __LINE__, __DATE__ ausgewertet werden müssen.

Makro-Namen werden gross geschrieben. Blanks sind erlaubt. Ein B_ am Anfang des Makros zeichnet ein metaFrame Makro aus.

```
#define B_ASSERT(Assertion) if (!(Assertion)[1]) \
 cout << __FILE__ << “:” << __LINE__ << “:” << #Assertion << endl;
```

[1] Makro-Parameter werden zur Verwendung immer in Klammern gesetzt. Ansonsten ergeben sich Probleme mit dem Präzedenz-Regeln des Compilers

Funktions-Namen

Funktions-Namen werden gross geschrieben. Die Gross-/Kleinschreibung im Namen selber darf variieren

Leading Underscores sind nur erlaubt, wenn es sich um interne (private) Helper-Funktionen[2] handelt.

Beispiel:

```
class MyClass
{
  public:
    void Fn(){....;_Fn();....;}
  private:
    void _Fn(){....}
};
```

Underscores sind des weiteren erlaubt, wenn mit dem Konkatinations-Operator ## des C-Pre-Processors neue Symbole generiert werden. Mit Hilfe der Underscores kann in manchen Fällen die Übersicht erhöht werden.

```
#define B_STUB(Symbol) class NetworkStub_##Symbol {…};

// Generates a class NetworkStub_MyClass
B_STUB(MyClass)
```

[2] Das public Interface enthält zum Beispiel oftmals eine Mutex-Semaphore zur Sicherstellung des gegenseitigen Ausschlusses. Eine private Funktion sollte keinen Schutz enthalten, da sie so auf demselben Stackframe mehrmals aufgerufen werden kann. Es gibt Betriebs-Systeme, welche diese Funktionalität bereits in den Semaphoren anbieten. Diese Calls sind jedoch um einiges langsamer.

Globale Funktionen

Globale Funktionen sind ausnahmslos verboten. Der Ersatz bietet eine statische Klassenfunktion[3].

[3] In einem objekt-orientierten Design kann ausnahmslos jede Funktionalität einer Klasse zugeordnet werden.

Enum

Enum Typen werden durch ein grosses **E** gekennzeichnet. Zusätzlich werden enums mit einem Pre-Qualifier versehen, da der Namespace leider nicht durch den Enum, sondern die enthaltende Klasse definiert ist.

Die Werte selber werden mit **k** markiert.

Fehlerhaftes Beispiel:

```
class MyClass
{
  public:
    enum ETrafficLight {kRed,kYellow,kGreen};
    enum ESunLight {kRed,kYellow,kWhite};
};
```

In obigem Beispiel kollidieren die Deklarationen für kRed und kYellow, da der Namespace von MyClass relevant ist.

Korrektes Beispiel:

```
class MyClass
{
  public:
    enum ETrafficLight {kTrf_Red, kTrf_Yellow, kTrf_Green};
    enum ESunLight { kSun_Red, kSun_Yellow, kSun_White};
};
```

Variablen-Namen

[Allocation][Parameter Type]_[Reference Type](Variable Type)VarName[Array]
[s][m|i|o|io]_[r|p](a|n|f|e|b|c)VarName[Arr];

[Eckige] Klammern bedeuten, dass die Angaben bei nicht zutreffen weggelassen werden können.
(Runde) Klammern bedeuten, dass das Vorhandensein der Angaben zwingend ist.

[Allocation]	
s	static

[Parameter Type]	
m	member
i	input parameter
o	output parameter (always a reference or a pointer)
io	input/output parameter (always a reference or a pointer)

[Reference Type]	
r	reference
p	pointer

(Variable Type)		
a	object of class or struct	
n	integer number	Ex: int, long, const int, short etc.
f	floating point	Ex: float double, const float
e	enum	
b	bool	
c	char	

Variablen Name	
VarName	Capital and small Letters, Leading capital letters, no underscores, at least 3 characters

[Array]	
Arr	The variable represents an array

Beispiele:

```
char* pcGreetings = ”Hello World”;
char cGreetingsArr[100];

float fValueArr[100];
for (int nIdx=0;nIdx<100;nIdx++) fValueArr[nIdx]=nIdx*3.4;
```

Enum Hack

Der Enum Hack[4] wird verwendet, um Konstanten im Header zu deklarieren und definieren.

```
class MyClass
{
  public:
    enum {s_nNumOfElements = 100};
    MyClass()
    {
      for (int nIdx=0;nIdx<s_nNumOfElements;nIdx++) m_paUserClassArr[nIdx]=0;
    }
  private:
    UserClass* m_paUserClassArr[s_nNumOfElements];
};
```

[4] Obwohl der Standard eigentlich das Initialisieren von const Primitives im Header-File erlaubt, wird dies von den wenigsten Compilern akzeptiert.

const correctness

Const wird wo immer möglich[5] verwendet. Objekte und Strukturen werden nie als Kopie, sondern immer als const&[6] übergeben.

[5] Const sowohl bei Variablen als auch bei Funktionen. Const Correctness erlaubt es, viele konzeptionelle Fehler bereits zur Kompilierzeit zu entdecken.

[6] Diese Regel gilt vor allem bei templates als absolut unumgänglich. Ansonsten resultieren ernsthafte Performance Probleme. Ausserdem ist das schreiben eines Copy Constructors und eines Assignment Operators nötig.

Pointer vs. Reference als Übergabe-Parameter

Es gilt die folgende Regel[7]:

- ?? Wird die Verantwortlichkeit für die Lebenszeit eines Parameters abgegeben, so wird dieser per Pointer übergeben (die Aufgerufene Funktion ist für den Aufruf des delete operators zuständig).
- ?? Wird ein Parameter nur zur Verwendung übergeben, erfolgt die Übergabe per Reference.

Ein Abweichen von der obigen Regeln muss im Kommentar explizit erwähnt werden.

```
class Test
{
  public:
    Test(User& i_raUser)
    : m_paUser  (&i_raUser)
    , m_bDelete (false)
    {}

    Test(User* i_paUser)
    : m_paUser  (i_paUser)
    , m_bDelete (true)
    {}

    ~Test() {if (m_bDelete) delete m_paUser;

    User&       Get()       {return *m_paUser);
    const User& Get() const {return *m_paUser);

  private:
    User* m_paUser;
};
```

[7] Die Pointer vs. Reference Regel ist nötig, um Memory Leaks bereits in der Codierphase zu verhindern.

Copy Constructor und Assignment Operator

Copy Constructor und Assignment Operator müssen per default in allen Klassen und Structs private deklariert (aber nicht implementiert) werden[8].

Dies aus dem einfachen Grund, weil das standardmässige Bit-Wise-Copy meist nicht den gewünschten Effekt hat.

```
class Test
{
  public:
    ....
  private:
    Test(const Test&);
    Test& operator=(const Test&);
};
```

Man achte auch auf die korrekte Signatur[9] der Operatoren!

[8] siehe auch: Scott Meyers: Effective C++
[9] siehe auch: Scott Meyers: Effective C++

Kommentar

Jede Klasse und jede Funktion wird kommentiert, so dass dieser Kommentar zum Generieren[10] eines User-Manuals verwendet werden kann.

[10] Zum Beispiel mit Object Outline von http://www.bbeesoft.com

Interfaces

Die Verwendung von Multiple-Inheritance unter C++ schafft einige Probleme, vor allem Punkto Wartbarkeit. So erhöhen wir die Komplexität des Codes in schier unverantwortlicher Weise.

Keine Multiple-Inheritance zu verwenden resultiert hingegen in sogenannten Fat-Bases. Das heisst, sämtliche gemeinsame Funktionalität wandert in die Basisklasse.

Nun stellt sich aber das folgende Problem. In der Basis-Klasse befindet sich sämtliche denkbare Funktionalität in Form von virtuellen Funktionen. Es liegt aber auf der Hand, dass nicht jede abgeleitete Klasse alle Funktionalität braucht oder brauchen wird. Somit wird es nötig, leere Implementationen für diese Funktionen zur Verfügung zu stellen. Dies widerspricht aber dem Gesetz der Vererbung.

Eine elegante Lösung, um diesem Problem zu begegnen ist die Einführung von Interfaces. Ein Interface[11] enthält per Definition nur Funktionsdeklarationen, aber keine Implementationen. Des weiteren ist es untersagt, Membervariablen anzulegen.

> Def: In C++ wird ein Interface durch eine Klasse repräsentiert, welche nur *Pure Virtual Functions* enthält. Falls ein Objekt über den polymorphen Interface Pointer gelöscht werden soll (operator delete), so ist ein virtueller Destruktor[12] nötig.

Um obigen Forderungen Nachdruck zu verleihen, werden zwei neue Keywords definiert. Durch diese an Java angelehnte Syntax wird es einfacher, die oben genannten Richtlinien durchzusetzen.

```
#define B_INTERFACE class
#define B_IMPLEMENTS public

B_INTERFACE StreamableIf
{
  public:
    virtual ~StreamableIf(){}
    virtual void Write(std::istream&) = 0;
    virtual void Read(std::ostream&) = 0;
};

class MyClass : public       Base
              , B_IMPLEMENTS StreamableIf
{
  public:
    ….
  private:
    virtual void Write(std::istream&){…}
    virtual void Read(std::ostream&){…}
};
```

[11] siehe auch Java Language Specification
[12] Der virtuelle Destruktor besitzt als einzige Ausnahme ein leere Implementation ✍ {}

Def: Wird auf eine Funktion nur über ein Interface (virtuelle Funktion) zugegriffen, so kann diese in der Sub-Klasse private deklariert werden.