C++ Coding Style

LANKE

2. Mai 2017

Um die Garantie zu geben, dass die Qualität des Quellcodes eigehalten wird, ist eine Richtlinie für den Code zu schreiben, unabdingbar. Auf Basis von den Codebeispielen der Professoren, unseren Erfahrungen und allgemeinen Vereinbarungen in der Programmierer Szene ist hier für das Praktikum ESE eine solche Richtline erstellt worden.

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument legt die Konventionen für den Quellcode fest. Zuerst wird das Sprachbild (die Namensgebung, das Layout und die Struktur im Code) festgelegt. Danach wird über das Dokumentieren gesprochen. Das Testen eines Modules wird abschließend besprochen.

2 C++ Sprachbild

2.1 Namengebung

Alle Namen sind Englisch.
Typen: **C**amel**C**ase
- Klasse (class)
- Struktur (struct)
- Aufzählung (enum)
  - Singular (Status statt Stati)
- typedef

Variablen: **c**amel**C**ase
private Attribute: **c**amel**C**ase
- suffix_

Konstanten: **UPPER_CASE**
Funktionen: Verb + Nomen, **c**amel**C**ase
Flags: **c**amel**C**ase
- **is**Prefix,
- keine Negation
- > isRunning anstatt isNotRunning

```
#define UNIVERCITY_NAME "HAW Hamburg"
class UniAccount
{
  public:
    Status getStatus(void) ;

  private:
    Status status_ ;
    bool isStudent_ ;
}

enum Status
{
   STUDENT
  ,PROF
}
```

2.2 Layout

white space

Einrückung: für jeden Block
- Leerzeichen statt Tabs
- 2 oder 3 Leerzeichen

Zeilen zwischen... :

Funktionen: eine Leerzeile

logische Aufteilung: Zeilenkommentar statt Leerzeile

Lesbarkeit: ist das oberste Ziel
- **Klammern:** je nach Lesbarkeit
- **Operantionen:** zwischen Operanten und Operatoren ein Leerzeichen
- **Listen:** hinter jedem Komma ein Leerzeichen (Ausnahme: wenn Liste unteinander geschrieben wird)
- **Zuweisungen:** es dürfen mit Leerzeichen Tabellen gemimt werden.

Blöcke

öffnende Blockklammer: {
- **Typen:** in neuer Zeile
- **Schleifen:** direkt dahinter oder neue Zeile
- **Bedingungen:** direkt dahinter oder neue Zeile

schließende Blockklammer: }
- in der Einrückebene, in der sie geöffnet wurden
- sind einziges Zeichen in Zeile

Zeilenumbruch

Zeilenzeichenlimit: 80 Zeichen

Deklaration: je eine Zeile
- wenn das Zeichenlimit pro Zeile nicht reicht...
  - **Parameter-/Argumentenliste:** wo die Klammer sich öffnet
  - **Zuweisung, Berechnung:** wo = beginnt

```
1  class UniAccount
2  {
3    public:
4      void work(int time){
5        // get infos
6        int freeTime      = freeTime_ ;
7        Status motivation = getMotivation();
8        // test if she/he is capable to work
9        if (freeTime > time && motivation => OK){
10         workTime_ += time;
11         freeTime_ -= time;
12       }
13       return;
14     }
15
16     int calcSomthing ( int a, int b, int c,
17                        int d, int e, inf f
18     ){
19       int thisReturnValueNameLong = a + b + c
20                                    + d + e + f;
21       return thisReturnValueNameLong;
22     }
23
24   private:
25     workTime_ ;
26     freeTime_ ;
27 }
28
29 enum Status
30 {
31    DEPRESSED
32   ,OK
33   ,MOTIVATED
34 }
```

3 Struktur

Module: Teile Implementation vom Interface
- **Module.h:** Interface
  - verwende include guards (MODULE_H_)
  - Deklarationen
  - Templates
  - inline function Definitionen
- **Module.cpp:** Implementation
  - Funktionsdefinitionen
  - strikte innere Klassen
- **Ausnahme:** Eigene Bibliotheken sind in einem Header definiert.

Weiteres: Was soll noch eingehalten werden.
- keine magic numbers (lieber Konstanten)
- vermeide namespaces
- order of includes: most spacific first
- wenn du einen Header nur im cpp file benötigst, füge diesen auch nur dort ein
- wenn möglich, verwende forward declaration
- erstelle den Ctor
- verwende RAII (Ressource hängt von Lebzeit des Objekts ab)
- versuche die rule of three
  - Konstruktor und passender Destruktor
  - Kopierkonstruktor
  - assigment operator

Module.h

```
#ifndef MODULE_H_
#define MODULE_H_

class SomeClass;

class Module
{
  public:
    // rule of three
    Module(SomeClass) ; // ctor
    virtual ~Module() ; // destructor
    Module(const Module&) ; // copy constructor
    Module& operator= (const Module&); //
    assigment operator

    void printSomething(void) const;

  private:
    SomeClass &handle_ ;
}
#endif /* MODULE_H_ */
```

Module.cpp

```
#include "Module.h"
#include "SomeClass.h"

#include <iostream>

using namespace std;

// RAII begin
Module::Module(SomeClass cl)
: handle_(cl)
{
  //ctor
}

Module::~Module(){
  release (handle_);
}
// RAII end

Module::printSomething(void){
  cout << handle_->getSomthing() << endl;
}
```

4 Dokumentation

CODE DOKUMENTATION

Die Dokumentation wird mit dOxygen erzeugt. Dafür wird mit Tags auf Informationen gesammelt. DOxygen holt sich die Informationen aus nebenstehenden Kommentarstrukturen. Funktionen und Typen werden mit dem Block erklärt. Variablen, Attribute und Konstanten werden mit /**< ...*/ beschrieben. Die Beschreibung erfolgt fast ausschließlich in der Headerdatei des Modules. Erstellt wird die Dokumentation mit den Befehlen:

```
$ doxygen <config-file>
```

```
/** @file <filename>
 *  @brief <short description of module>
 *
 *  <detailes, longer explaination, pattern, componente>
 *  @author <author 1>
 *  @author <author 2>
 */

/**
 * @class <description of class>
 */
class Module
 {
 ...
/**
 *  @details description of function
 *  @param a <description of 1st parameter>
 *  @param b
 *       <description of 2nd parameter>
 *  @return <description of retrunvalue>
 */
int  func(int a, int b);

/// single line

int x_; /**< decription of x */
```

Die verwendetet Tags für dieses Projekt sind:

tag	Rendering
@file	Name des Files
@brief	kurze Beschreibung des Moduls
@author	Name des Authors
@class @enum @struct	Beschreibung des Typen
@details	kurze Beschreibung der Funktion
@param <par>	Beschreibung des Parameters <par> in Funktionen
@return	Beschreibung des Rückgabewertes

LIZENZ

Unser Project läuft unter der MIT Lizenz, welche in der Textdatei `LICENSE.txt` beschrieben ist. Jedes File muss den folgenden Text im header haben.

```
/**
 *   ...
 *  Embedded System Engineering SoSe 2017
 *  Copyright (c) 2017  LANKE devs
 *  This software is licensed by MIT License.
 *  See LICENSE.txt for details.
 */
```

5 Testen

Wie ein Unittest aussieht, ist jedem Entwickler freigestellt. Er hat somit die alleinige Verantwortung, dass sein Code richtig funktioniert. Die Testfiles müssen in dem Unterverzeichnis `test` abgelegt werden, und sollten im Team veröffentlicht werden. Die Tests sollten alle Ausnahmefälle und die Funktion des Modules Testen. Eventuell werden auch mehrere Komponenten gleichzeitig getestet. Bevor es an die Hardware geht, sollte der Code in Software reibungslos laufen.

Mögliche Testwerkzeuge sind einmal ein Testmodul mit Mainfunktion oder man verwedet C++Unit.

kurze Empfehlungen

Wenn Pattern verwendet werden, sollten die Namen der Pattern bzw. deren Komponenten in den Namen der Klassen wiederzufinden sein.