|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Coding Style** | | | | |
| Version | 0.1 (Draft) | | Kessener | 19.04.18 |
| Index | 0  1  2  3  3.1  3.2  3.3  4  5 | Nicht verwendete IDs  Klassen  Funktionen und Methoden  Variablen  Lokale Variablen  Instanzvariablen  Konstanten  Typen  Kontrollfluß | | |

### **0 Nicht verwendete IDs**

Nicht verwendet werden IDs die mit einem Underscore (“\_”) anfangen **oder** enden. IDs dürfen auch keine doppelten Underscores enthalten.

### **1 Klassen**

Klassennamen fangen mit einem Großen Buchstaben an und werden in CamelCase geschrieben. public und private Label werden einen Tab intendiert. Code innerhalb dieser Blöcke wird ein weiteren Tab intendiert.

Klassen bestehen aus bis zu Sechs Blöcken:

* typedef s, Helper-Klassen und Konstanten (Dieser Block kann public und/oder private sein; er wird als einziger **nicht** indendiert.)
* public Methods
* protected Methods
* private Methods
* Instanzvariablen (**Alle** Instanzvariablen sind **immer** private. Keine Ausnahmen!)
* delete t Methods (zB Copy-Constructor, etc.)

Jeder Block beginnt mit dem entsprechenden public, protected bzw private Label, selbst wenn der vorherige Block das selbe Label hat. Zwischen allen Blöcken außen den Methoden-Blöcken (Blöcke 2, 3 und 4) liegt eine leere Zeile. Leerzeilen dürfen überall eingefügt werden, wo man meint, dass es der Lesbarkeit dient.

|  |
| --- |
| **class** **MyAwesomeClass** {  **public**: // Block #1  **typedef** **int** **id\_t**;  **static** **constexpr** **id\_t** INVALID\_ID = 0;   **class** **Helper**  {  };   **public**: // Block #2  MyAwesomeClass( );  **virtual** ~MyAwesomeClass( );  **void** **doSomething**( );  std::string& **name**( ) { **return** mName; } // Getter & Setter  **protected**: // Block #3  **virtual** **void** **doSomethingImpl**( ) = 0;  **private**: // Block #4  **void** **calculateSomething**( );   **private**: // Block #5  **id\_t** mID;  std::string mName;   **private**: // Block #6  MyAwesomeClass& **operator**=(**const** MyAwesomeClass&) = **delete**; }; |

### **2 Funktionen und Methoden**

Methoden werden in camelCase (mit kleinem Anfangsbuchstaben) geschrieben. Funktionen werden klein\_mit\_underscores geschrieben.

|  |
| --- |
| **class** **MyClass** {  **public**:  **void** **computeSomething**( ); // method };  **void** **compute\_something\_else**( ); // function |

### **3 Variablen**

#### **3.1 Lokale Variablen**

Lokale Variablen fangen mit einem kleinen Buchstaben an. Sie können sowohl als camelCase wie auch klein\_mit\_underscore geschrieben werden.

|  |
| --- |
| **int** **main**(**int** argc, **char** \*argv[]) {  **int** aLocalVariable;  **double** another\_local\_variable;   **return** 0; } |

#### **3.2 Instanzvariablen / Member variables**

Instanzvariablem werden in camelCase geschrieben und fangen immer mit einem kleinen ‘m’ (für “Member”) an.

|  |
| --- |
| **class** **MyClass** {  **private**:  **int** mMemberVariable;  **double** mAnotherVariable; }; |

#### **3.3 Konstanten**

Konstanten werden GROSS\_MIT\_UNDERSCORE geschrieben. Konstanten, die in einem Header definiert werden, werden wenn möglich in eine Klassendklaration geschrieben.

|  |
| --- |
| **class** **MyClass** {  **public**:  **static** **constexpr** **int** ANSWER\_TO\_LIFE\_THE\_UNIVERSE\_AND\_EVERYTHING = 42; }; |

### **4 Typen**

Typen, die durch die Schlüsselwörter class oder struct definiert werden, werden in CamelCase (mit großem Anfangsbuchstaben) geschrieben. Typen, die durch ein typedef definiert werden, werden klein\_mit\_underscore geschrieben und enden immer mit einem \_t (für “Typ”) oder \_fn (für “Funktion”, also wenn der definierte Typ ein Funktor beschreibt).

|  |
| --- |
| **class** **MyClass** {  **public**:  **typedef** uint **id\_t**;  **typedef** std::function<**void**(**void**)> callback\_fn; }; |

### **5 Kontrollfluß**

“{“ und “}” werden immer auf eine eigene Zeile geschrieben. Außnahmen sind leere Blöcke “{ }”, die auf eine Zeile geschrieben werden **dürfen**. Kleine Funktionen dürfen auch in einer einzelnen Zeile untergebracht werden. In einem for Statement werden Leerzeichen vor **und** nach den “;” eingefügt.

|  |
| --- |
| **class** **MyClass** {  **public**:  MyClass( ) { }  };  **struct** **MyException** : **public** std::exception { };  **int** **square\_number**(**int** a) { **return** a \* a; }  **int** **main**(**int** argc, **char** \*argv[]) {  **try**  {  **for**(**int** i = 0 ; i < 10 ; ++i)  {  **if**(a)  {  do\_a();  }  **else**  {  do\_b();  }  }  }  **catch**(**const** std::exception& e)  {  }   **return** 0; } |