Unik - Engineers Programmier-Styleguide

Kevin Kappelmann

28. Juni 2015

Bemerkung

Dieses Dokument ist work in progress. Die Regeln sollen mit Rücksprache des Engineering-Teams fix festgelegt werden und werden vor allem im frühen Stadium des Projektes laufend erweitert.

Inhaltsverzeichnis

1 Vorwort

Dieser Styleguide stellt wichtige Regeln für die Entwicklung des C++-Quellcodes für das Spieleprojekt zusammen. Alle Punkte dieses Styleguides müssen bei jeder Datei vor dem Upload in das git-Repository erfüllt sein. Ziel des Styleguides ist es eine konsistente, standardisierte und vor allem sehr gut wartbare Codebasis zu garantieren, weshalb es wichtig ist, dass sich wirklich jeder an diese hier festgehaltenen Regeln hält.
Mir ist klar, dass bei einigen Punkten (vor allem Formatierungsregeln) unterschiedliche Präferenzen bestehen, allerdings ist es wichtig, einen einheitlichen Style durchzuziehen, um Konsistenz zu garantieren und das Merging von Dateien einfach zu ermöglichen.

2 Allgemeine Regeln

2.1 Git-Commits

- Code, der commitet wird, muss kompilierbar sein (Testen!).
- Commits sollen in kleinen Stücken und häufig geschehen. Ebenfalls soll jeder Commit für sich eine geschlossene Einheit darstellen. Commits à la "Added new ghost, added path finding algorithm, fixed 1000 bugs, cleaned up code and did some more magic." sind zu vermeiden.
- Commit-Nachrichten sind auf Englisch und beschreibend zu verfassen.
- Nach den Commits pushen nicht vergessen!

2.2 Dateiendungen

- .cpp für C++-Implementierungs-Dateien
- .hpp für C++-Header-Dateien
- .tpp für C++-Template-Implementierungen

2.3 Include-Guards

Es werden ausschließlich ifndef, define, endif und keine pragma once Include-Guards verwendet. Pragma once wird zwar von fast jedem Compiler unterstützt, es ist aber kein offizieller Standard und um Konsistenz zu bewahren, wird daher strikt ifndef, define, endif verwendet. Die Include-Guards haben der folgenden Namenskonvention zu folgen:
PROJECTNAME_SUBDIRECTORY1_SUBSUBDIRECTORY2_NAMEOFFILE_HPP_
Beispiel:

```
1 #ifndef HORRORGAME_ENEMIES_GHOSTS_ONEARMEDGHOST_HPP_
2 #define HORRORGAME_ENEMIES_GHOSTS_ONEARMEDGHOST_HPP_
3 //code in hpp file
4 #endif
```

2.4 Namenskonventionen

- Dateien und Klassen starten mit einem Groß-, Funktionen und Variablen mit einem Kleinbuchstaben. Dabei wird immer CamelCase (`https://en.wikipedia.org/wiki/CamelCase`) verwendet.
  Ausnahmen stellen Konstanten und Enums dar. Diese bestehen nur aus Großbuchstaben und Unterstrichen an Stelle von CamelCase.

```
1 std::string playerName; //richtig
2 std::string PlayerName; //falsch
3 std::string player_name; //falsch
4
5 constexpr double SUPER_AWESOME_CONSTANT = 2502.1994; //richtig
6 constexpr double superAwesomeConstant = 2502.1994; //falsch
```

- Namespaces bestehen aus ausschließlich Kleinbuchstaben.

```
1 namespace enemyutils{//code} //richtig
2 namespace enemyUtils{//code} //falsch
```

- Abkürzungen und Akronyme (z.B. HTML oder DVD) bestehen nicht aus nur Großbuchstaben bei der Verwendung in einem Namen.

```
1 std::string outputHtmlAddress; //richtig
2 std::string outputHTMLAddress; //falsch
```

2.5 Code-Dokumentierung

- Dokumentiert eure Dateien (siehe weiter unten), Klassen, Funktionen und Variablen sinnvoll, beschreibend und klar.

- Es wird ausschließlich in Englisch dokumentiert.

- Funktionen, Klassenvariablen und ähnliches werden in der Header-Datei, in welcher sie definiert werden, dokumentiert.

- Alle Dokumentationen müssen dem JavaDoc-Style folgen (`https://en.wikipedia.org/wiki/Javadoc`).

- Jede neue Datei muss einen führenden Kommentar beinhalten, der folgendem Style folgt: (Anm.: Die Lizenz steht noch nicht fest, seht den Text hier als Beispiel)

```
1 /**
2  * @file ghostEnemy.hpp
3  * Header file which contains the definitions for the ghost enemy.
4  * @author Kevin Kappelmann
5  * @license gnu general public license version 3
6  * @version 0.1
7  * @date 2015-04-04
8 */
```

bzw.

```
/**
 * @file ghostEnemy.cpp
 * Implementation of the functions definied in ghostEnemy.hpp
 * @author Kevin Kappelmann
 * @license gnu general public license version 3
 * @version 0.1
 * @date 2015-04-04
*/
```

bzw.

```
/**
 * @file ghostEnemy.tpp
 * Implementation of the template functions defined in ghostEnemy.hpp
 * @author Kevin Kappelmann
 * @license gnu general public license version 3
 * @version 0.1
 * @date 2015-04-04
*/
```

Wenn jemand einen größeren Part einer existierenden Datei ändert, so soll die Versionsnummer erhöht und der Modifizierer zusätzlich als Autor vermerkt werden.
Bei kleineren Veränderungen/Fixes soll die Versionsnummer um eine Nachkommastelle, bei großen, tiefgreifenden Änderungen die führende Versionsziffer erhöht werden.

2.6 Code-Formatierung

- Jede C++-Datei wird mit Allman-Style (`https://en.wikipedia.org/wiki/Indent_style#Allman_style`) formatiert. Beim Allman-Style werden geschweifte Klammern, die einen neuen Scope definieren, in eine neue Zeile geschrieben. Ausgenommen hiervon sind leere Scopes und List-Initialisierungen.

- Ausdrücke, bei denen die Reihenfolge der Ausführung nicht klar ersichtlich ist, sind zusätzlich zu klammern.
  Beispiel:

```
//someVal will hold 3.0
double someVal = 3/2+2*3-7?2.0:3.0; //schlecht
double someVal = (3/2+2*3-7)?2.0:3.0; //besser
```

- Es wird mit Tabs und nicht mit Whitespaces eingerückt.

Beispiel für ein Header-File:

```
/**
 * @file CountingSort.hpp
 * Header file for the counting sort algorithm.
 * The counting sort algorithm is a relatively fast sorting algorithm.
 * However, in order to use the counting sort algorithm
```

```
 6  * one needs to know the range of values which can appear in
 7  * the vector which should be sorted (lower and upper limit).
 8  * The counting sort algorithm performs in O(n+(upperLimit-lowerLimit)
 9  * best, average and worst-case.
10  * Depending on the implementation, the counting sort algorithm
11  * can either be stable or unstable.
12  * The counting sort algorithm has a space complexity of
13  * O(n+(upperLimit-lowerLimit)).
14  * @author Kevin Kappelmann | kappelmann.me | github.com/kappelmann
15  * @license GNU General Public License Version 3
16  * @version 0.2
17  * @date 2015-03-26
18  */
19 #ifndef ALGORITHMS_SORTING_COUNTINGSORT_COUNTINGSORT_HPP_
20 #define ALGORITHMS_SORTING_COUNTINGSORT_COUNTINGSORT_HPP_
21
22 #include "../SortingAlgorithm.hpp"
23
24 class CountingSort final : public SortingAlgorithm<int>
25 {
26   public:
27     /**
28      * @param vector the vector which should be sorted.
29      * @param lowerLimit the lowest value which can
30      *  possibly appear in the vector.
31      * @param upperLimit the highest value which can
32      *  possibly appear in the vector.
33      */
34     CountingSort(std::vector<int> & vector, int const lowerLimit, int const
          upperLimit);
35     using SortingAlgorithm<int>::sort;
36     /**
37      * Sorts the passed vector of the given size which can hold
38      * values between the passed upper and lower limit.
39      *
40      * @param vector the vector which should be sorted.
41      * @param lowerLimit the lowest value which can
42      *  possibly appear in the vector.
43      * @param upperLimit the highest value which can
44      *  possibly appear in the vector.
45      */
46     static void sort(std::vector<int> & vector, int lowerLimit, int upperLimit
          );
47   protected:
48     /**
49      * @copydoc SortingTest#run()
50      */
51     void run();
52   private:
53     int const lowerLimit;
54     int const upperLimit;
55 };
56 #endif
```

Und dazugehöriges Cpp-File:

```
1
2  /**
3   * @file CountingSort.cpp
4   * Implementation of the functions defined in CountingSort.hpp.
5   * This implementation produces an unstable sort.
6   * @author Kevin Kappelmann | kappelmann.me | github.com/kappelmann
7   * @license GNU General Public License Version 3
8   * @version 0.2
9   * @date 2015-03-26
10  */
11
12 #include "CountingSort.hpp"
13 #include <vector>
14
15 CountingSort::CountingSort(std::vector<int> & vector, int const lowerLimit,
       int const upperLimit):SortingAlgorithm<int>(vector, new std::string("
       counting sort")),lowerLimit(lowerLimit),upperLimit(upperLimit) {}
16
17 void CountingSort::run()
18 {
19   CountingSort::sort(this->vector, this->lowerLimit, this->upperLimit);
20 }
21
22 void CountingSort::sort(std::vector<int> & vector, int lowerLimit, int
       upperLimit)
23 {
24   //create the vector which counts the occurence of each value
25   auto countSize = upperLimit-lowerLimit+1;
26   std::vector<unsigned int> * count = new std::vector<unsigned int>(countSize)
         ;
27   std::fill(count->begin(), count->end(), 0);
28   for(auto & element : vector)
29   {
30     //count the occurence of each value in the passed vector.
31     ++((*count)[element-lowerLimit]);
32   }
33   decltype(vector.size()) curIndex = 0;
34   for(decltype(vector.size()) i=0; curIndex<vector.size(); ++i)
35   {
36     for(unsigned int j = 0; j<(*count)[i]; ++j)
37     {
38       /*each time we counted an occurence of the current value,
39         we create an entry for the current value in the sorted vector.*/
40       vector[curIndex++]=i+lowerLimit;
41     }
42   }
43   delete count;
44 }
```

2.7 Arrays und Strings

C-Style Arrays - das sind Arrays der Form "T array[]" - sollen nie verwendet werden. Stattdessen soll der von Unreal gegebenen dynamische Arraytyp TArray, der sich analog zu std::vector verhält, verwendet werden.
C-Style Arrays sind wohl einer der häufigsten Fehlerquellen und bieten fast nie einen Performancegewinn. Aus diesem Grund sollen auch Strings nicht in Form von char-Arrays, sondern String-Klassen von Unreal benützt werden.

2.8 Using-Direktiven

Using-Direktiven sollten nie in Header-Files verwendet werden, da ansonsten alle Files, die dieses Header-File inkludieren, plötzlich auch den angegebenen Namespace bzw. Member inkludieren und es dazu schnell zu Namenskonflikten kommen kann. Des Weiteren sollten ganze Namespaces, vor allem externe Namespaces, nicht komplett inkludiert werden.

```
1 using namespace std; //Schlecht! ganzer Namespace wird inkludiert.
2 int main()
3 {
4   foo();
5   cout << "Nur ein Test" << endl:
6   return 0;
7 }
8
9 void foo(){//code that does not use cout}
```

```
1  int main()
2  {
3    foo();
4    /*
5     * Besser! 1. Nur der verwendete cout Member wird inkludiert und
6     * 2. der Scope der Using-Direktive wird auf den folgenden Teil
7     * der main-Methode begrenzt.
8     */
9    using std::cout;
10   cout << "Nur ein Test" << endl:
11   return 0;
12 }
13
14 void foo(){//code that does not use cout}
```

3 Spezifische Regeln

Hier werden evtl. zusätzliche, für einen bestimmten Codebereich spezifische Regeln eingefügt.

Last modified: 2015/06/28