Was ist ein Beispiel für schlecht geschriebenen C ++ - Code?

Hans-Peter Beständig, Embedded C++, C++14
Aktualisiert 24.06.2019

Schlechten C++ Code? Ach, da gibt es sooooo viele Möglichkeiten, schlechten Code zu schreiben. Leider hat C++ daher auch oft den Ruf, inperformant zu sein.

Hier sind meine "Top 5“, die ich leider allzu häufig in fremdem Socurcecode sehe. Bedauerlicherweise gibt es auch Kollegen, die unten genannte Fehler wiederholt machen und sich dann wundern, dass deren Code crasht oder flasche Ergebnisse liefert.

Ironischerweise bekomme ich dann (vor allem beim Thema Copy Konstruktoren, unten) oft die Anwort "*Ach weißt Du, mir ist die Schreiberei zu viel, außerdem bläht das ja alles meinen Code auf*".

Ich kann da nur als Antwort aus Ratlosigkeit und Verzweiflung den Kopf schütteln.

Also genug geschwafelt - los geht's hier meine "**Top-5**“:

1) Der Klassiker bei "Hasenfüßen": Call by Value anstatt Call by Reference.

Viele Programmierer - vor allem solche, die von C auf C++ umsteigen - trauen sich nicht Referenzen zu benutzen und benutzen "*aus Sicherheit*" (hat wirklich mal ein Kollege so zu mir gesagt ;-)) gerne "*Copy by Value*".

Gegeben sei als Beispiel folgender (C++11 >=) Code:

```
1. /*!
2.  * \brief Counts the occurences of a certain character
   within a
3.  *        given String.
4.  * \param inChar The character to count within the string
5.  *        \p inString.
6.  * \param inString The string to count the occurences of
   the
7.  *        character \p inChar.
8.  * \return The count of occurence of the character \p
   inChar within
9.  *         \p inString.
10.  */
11. unsigned int countCharsInString(
12.     char inChar,
13.     std::string inString)
14. {
```

```
15.     unsigned int cnt = 0;
16.     for (auto thisChar : inString)
17.     {
18.         cnt += (inChar == thisChar);
19.     }
20.     return cnt;
21. }
```

Nun, dieser Code funktioniert, denn er liefert das Korrekte Ergebnis:

```
1. assert(3 == countCharsInString('l', "Hello, World!"));
```

-> Correct Assertion -> Test Passed.

Also, was gibt es denn dann zu beanstanden?

Dann fragt mal die *arme* CPU! Die muss bei jedem Aufruf zunächst den String 'inString' kopieren. Zudem wird temporär eine Kopie des Strings auf dem Stack gehalten. Und das alles nur, weil die Werteübergabe '*by Value*' ist.

Dabei ist diese Kopieraktion vollkommen **unnötig**, denn innerhalb der Funktion '`countCharsInString`' wird lediglich **LESEND** auf den String zugegriffen!

Ändert man die Funktionssignatur so …

```
1. int countCharsInString(
2.     char inChar,
3.     std::string inString)
```

…in…

```
1. int countCharsInString(
2.     char inChar,
3.     const std::string& inString)
```

… verwendet also statt dem tatsächlichen Inhalt des String-Objektes nur dessen Referenz (implizit ist das eigentlich nur ein elegant verpackter Pointer auf das Objekt), dann wird der vom Compiler erzeugte Code schon viel kompakter und auch performanter.

Erstens wird der String nicht mehr temporär kopiert (CPU Copy-Loop fällt weg -> Performancegewinn); und zweitens wird auch nicht unnötig Stackspace verschwendet. Der Code verwendet nun **lesend** direkt den OriginalString als **Referenz** (*implizit als Pointer*)

Überhaupt wird meiner Beobachtung nach dem Schlüsselwort 'const' eine viel zu geringe Beachtung geschenkt! Dieses Zauberwort wird viel zu wenig verwendet, verrät es dem Compiler doch viel genauer, was man vor hat, nämlich dass man nur lesen möchte, und erlaubt ihm somit Typ-Prüfungen (die z.B.).

Somit wird eine Zuweisung an konstante Elemente unmöglich - der Compiler wird solchen Code nicht übersetzen und als Fehler melden.

Man sieht sehr schnell, ob ein Programmierer erfahren ist oder nicht, nur alleine dass man seinen Code auf das Vorhandensein von const überprüft.

2) Falsche / unvollständige Initialisierungslisten.

Ein Übel, das ich oft beobachte, sind nicht vollständige oder (noch schlimmer) falsch angelegte Initialisierungslisten bei C++ Konstruktoren.

Beispiel:

```
1. #include <stdint.h>
2. #include <string>
3.
4. ...
5.
6. class StringBuffer
7. {
8.     char* m_Buffer;
9.     size_t m_StringLength;
10.
11. public:
12.     StringBuffer(const std::string& inString)
13.     : m_StringLength(inString.length())
14.     , m_Buffer(new char[m_StringLength+1])
15.     {
16.         strcpy(m_Buffer, inString.c_str());
17.     }
18.
19.     ~StringBuffer()
20.     {
21.         delete [] m_Buffer;
22.     }
23.
24.     const char* buffer() const
25.     {
26.         return m_Buffer;
27.     }
28.
29.     size_t length() const
30.     {
31.         return m_StringLength;
32.     }
33.
34. }; // class StringBuffer
```

Verwendet man diesen Code z.B. via 'StringBuffer stringBuffer("Hello, World!");' dann wird er crashen!

Warum? Weil es wir hier mit einer falschein Initialisierungsliste zu tun haben:

Die Merkregel ist, dass die Member in einer Intitlaisierungliste **immer** in der Reihenfolge initialisiert werden, wie sie **deklariert** wurden und **nicht** in der Reihenfolge **wie sie in der Initialisierungsliste genannt werden**.

Moderne Compiler spucken bei einem solchen Fehler eigentlich eine Warning aus, welcher aber gerne übersehen / oder unterschätzt wird.

Oben wird also effektiv zuerst ein Buffer mit der Länge 'm_StringLength+1' alloziert und erst dann m_StringLength die Länge des ursprünglichen Strings zugewiesen. Da es sich hierbei um die falsche Reihenfolge handelt - m_StringLength ist beim Allozieren noch nicht zugewiesen und damit undefiniert - Wird das Programm im Konstruktor irgend etwas machen, aber nichts definiertes. Die Größe des allozierten Speichers ist im Moment der Allozierung nicht definiert. Ein absolutes "**No Go**"!

Die Korrektur ist einfach - die Reihenfolge in der Initialisierungsliste muss mit der in der Deklaration übereinstimmen! Da wir im Beispiel oben wollen, dass die Länge vor der Anforderung des Speichers feststeht, würde ich hier die Reiehnfolge in der Deklaration ändern – sprich: die Länge wird zugewiesen, bevor diese Variable für die Allozierung verwendet wird.

```
1. #include <cstdlib>
2. #include <cstdint>
3. #include <string>
4.
5. ...
6.
7. class StringBuffer
8. {
9.     size_t m_StringLength;
10.     char* m_Buffer;
11.
12. public:
13.     StringBuffer(const std::string& inString)
14.     : m_StringLength(inString.length())
15.     , m_Buffer(new char[m_StringLength+1])
16.     {
17.         strcpy(m_Buffer, inString.c_str());
18.     }
19.     ...
```

3) Vergessene Copy-Konstruktoren / Assignment Operatoren.

Ein Übel, das viele Klassen aufweisen, ist, dass deren "Rule of three" bzw. "Rule of five" nicht beherzigt wurde. Die "Rule of three" besagt, dass eine Klasse, die Pointer verwendet, zudem einen Copy-Construktor und einen assigment Operator implementieren sollten. Die "Rule of five" erweitert diese Regel um den "move copy constructor" und "move assignment operator"

Nehmen wir doch gleich mal das Beispiel der Klasse '`StringBuffer`' von oben an.

Diese Klasse verwendet intern einen ***Pointer*** als Member. Somit herrscht ein *potentielles* Problem, denn wenn jemand nun schreibt:

```
1. StringBuffer buffer1("Hello, World");
2. StringBuffer buffer2 = buffer1;
```

…was wird dann wohl passieren, wenn die Objekte zerstört werden?

Das Programm wird aller Voraussicht nach crashen, und zwar weil **zwei mal** versucht wird **ein und denselben** Buffer im *Destruktor freizugeben*.

Zuerst wird das passieren, wenn das *erste Objekt* zerstört wird - dabei crasht das Programm noch nicht, bis da hin ist ja noch alles in Ordnung. Dann aber, wenn das *zweite Objekt* zerstört wird, dann wird in dessen Destruktor versucht **genau den Selben Speicher freizugeben**!

Warum? Nun, weil die Klasse **keinen** Assignment Operator definiert hat, wendet der C++ Compiler das *Standard* Verhalten für *imlizite Copy-Konstruktoren und Assignment Operatoren* an, nämlich er kopiert einfach alle Members *bit-identisch* auf das *andere* Objekt.

Übrigens gilt das Gleiche, wenn man ein Objekt versucht via Copy-Construktor zuzuweisen. Man also schreiben würde:

```
1. StringBuffer buffer1("Hello, World");
2. StringBuffer buffer2(buffer1);
```

Auch dieser Code crasht bei obiger Klasse aus den bereits genannten Gründen.

Daher ist es **dringends** anzuraten bei Klassen, die selbstverwaltete Pointer verwenden, sich um die Copy-Konstruktoren **und** Assignment Operatoren zu **kümmern**!

Das heißt, entweder man verbietet diese explizit, wenn man z.B. gar nicht möchte, dass Objekte einer Klasse kopiert werden sollen.

Verbieten kann man die entweder ab C++11 indem man in der Klasse oben schreibt:

```
1.     StringBuffer(const StringBuffer&) = delete; // Forbid
   copy ctor
2.     StringBuffer(StringBuffer&&) = delete; // Forbid move
   copy ctor
3.     StringBuffer& operator=(const StringBuffer&) =
   delete; // Forbid assignment operator
4.     StringBuffer& operator=(StringBuffer&&) = delete; //
   Forbid move assignment operator
```

…oder man formuliert diese *ordentlich semantisch korrekt* aus. z.B.

```
1.     StringBuffer(const StringBuffer& inStringBuffer)
2.     : m_StringLength(inStringBuffer.m_StringLength)
3.     , m_Buffer(new char[m_StringLength+1])
4.     {
5.         strcpy(m_Buffer, inStringBuffer.m_Buffer);
```

```
6.     }
7.
8.     StringBuffer(StringBuffer&& inStringBuffer)
9.     : m_StringLength(inStringBuffer.m_StringLength)
10.     , m_Buffer(inStringBuffer.m_Buffer)
11.     {
12.         inStringBuffer.m_StringLength = 0;
13.         inStringBuffer.m_Buffer = nullptr;
14.     }
15.
16.     StringBuffer& operator=(const StringBuffer& inRHS)
17.     {
18.         if (this != &inRHS)
19.         {
20.             m_StringLength = inRHS.m_StringLength;
21.
22.             delete[] m_Buffer; // Dispose old buffer
23.             m_Buffer = new char[m_StringLength+1];
24.             strcpy(m_Buffer, inRHS.m_Buffer);
25.         }
26.         return *this;
27.     }
28.
29.     StringBuffer& operator=(StringBuffer&& inRHS)
30.     {
31.         m_StringLength = inRHS.m_StringLength;
32.
33.         delete[] m_Buffer; // Dispose old buffer
34.              m_Buffer = inRHS.m_Buffer;
35.
36.         inRHS.m_StringLength = 0;
37.         inRHS.m_Buffer = nullptr;
38.         return *this;
39.     }
```

4) Gebrochene Versprechen

Schauen wir uns mal folgenden fiktiven Code an (*die Sinnhaftigkeit sei da mal dahingestellt* ;)):

```
1. #include <cstdlib>
2. #include <new>
3.
4. ...
5.
6. /*!
7.  * \brief Tries to allocate a Buffer with a given capacity in
8.  *        characters by using new.
9.  * \param inBuffSize The Amount of characters to allocate a buffer
10.  *        to.
11.  * \return A Pointer to a storage that is \p inBuffSize characters
12.  *        big.
13.  *        In the case of a failure (Strage of \p inBuffSize byte
14.  *        not possible) a nullptr will be returned.
```

```
15.   * \note  The caller has to call delete[] to dispose this storage
16.   *        if it is not needed any longer in order to regain the
17.   *        allocated storage!
18.   */
19.  char* allocBuffer(size_t inBuffSize) noexcept
20.  {
21.      char* newBuffer = new char[inBuffSize];
22.      assert(nullptr != newBuffer);
23.      return newBuffer;
24.  }
```

Diese Implementation ist mangelhaft! Aber warum?

Nun, es wird hier das Versprechen '*noexcept*' gebrochen, nämlich, dass diese Funktion keine Exception werfen wird. Aber ist dem wirklich so?

Viele Programmierer gehen *leider* davon aus, dass ein '`new char[inBuffSize]`' beim Fehlschlag - sprich die Anforderung des Speichers nicht erfolgreich ist, weil z.B. nicht mehr genügend Speicher zur Verfügung steht, einen `nullptr` zurückgibt, aber diese Annahme ist **FALSCH**!

In Wirklichkeit wird aber im Falle eines Fehlschlages lt. Definition des '`operator new`' passieren, dass eine `std::bad_alloc` exception geworfen wird. Und das Werfen der Exception in einer Funktion die als `noexcept` vereinbart wurde, *bricht* dann dieses *Versprechen*!

Und was hat das dann zur Folge? Interessanterweise passiert dann durch die Angabe von '`noexcept`' nun, dass beim Fehlschlag das Programm **sofort** durch abort beendet wird! Selbst wenn der Aufrufer schreiben würde...

```
1. ...
2. char* myBuffer;
3. try {
4.     myBuffer = allocBuffer(size_t(-1));
5. }
6. catch (...)
7. {
8.     myBuffer = nullptr;
9. }
```

...würde ihn das nicht retten, weil durch die Angabe von 'noexcept', der Compiler sich dann auf dieses Versprechen der Implementation verlässt und für diese Funktion **keinen Code fürs Exception Handling** erzeugt! Das hat zur Folge, dass der Prozess sofort schon durch ein `abort()` beendet wird, wenn '`new char[inBuffSize]`' eine `std::bad_alloc` exception wirft!

Oder anders gesagt: Das oben genannte Exception Handling wird **nie** wirksam!

Aber wie macht man das besser?

Nun. Entweder man kümmert sich innerhalb der als `noexcept` deklarierten Funktion um das Exception Handling und sorgt dafür, dass eben **keine** Exception nach außen dringt - *erfüllt also sein gegebenes Versprechen*:

```
1. #include <cstdlib>
2. #include <new>
3.
4. ...
5.
6. char* allocBuffer(size_t inBuffSize) noexcept
7. {
8.     char* newBuffer;
9.     try
10.     {
11.         newBuffer = new char[inBuffSize];
12.     }
13.     catch (const std::bad_alloc&)
14.     {
15.         newBuffer = nullptr;
16.     }
17.
18.     assert(nullptr != newBuffer);
19.     return newBuffer;
20. }
```

ODER - noch Eleganter - man verwendet die '`nothrow`' Variante des `operator new`:

```
1. #include <cstdlib>
2. #include <new>
3.
4. ...
5.
6. char* allocBuffer(size_t inBuffSize) noexcept
7. {
8.     char* newBuffer = new(std::nothrow) char[inBuffSize];
9.     assert(nullptr != newBuffer);
10.     return newBuffer;
11. }
```

Der Witz am '`new(std::nothrow)`' ist nämlich der, dass der sich dann so verhält, wie es viele Programmierer (eigentlich) *denken*, dass sich ein '`new`' verhält. Nämlich der '`new(std::nothrow)`' gibt nun **wirklich** einen `nullptr` zurück, wenn die Specheranforderung fehlschlägt. So wird das gegebene Versprechen eingehalten.

5) Array New does not match array delete

Die Regel ist, dass Objekte, die durch einen *Array* ***new*** erzeugt werden, auch wieder durch einen *Array* ***delete*** zerstört werden müssen.

Beispiel:

```
1. char* tmpBuffer = new char[1234];
2.
3. // Do something with the buffer...
4.
5. // WRONG CODE!
6. delete tmpBuffer; // Time to release the storage
7.
8. // Good Code: Storage allocated using new[] demands to
   release it by using 'delete[]'!
9. delete[] tmpBuffer; // Time to release the storage
```

Ich hoffe, das hilft Dir oder einem Anderen weiter!

In diesem Sinne - bleibt dran und macht Katas im Coding Dojo - denn nur Übung und "Hands-On" machen Meister! :-)