Ein kleiner C++ Style Guide

Prof. Dr. Reinhold Kröger

Sven Bauer

Stand 15.3.2004

Fachhochschule Wiesbaden

Fachbereich Informatik

Inhaltsverzeichnis

1 Wozu dient ein Style Guide ?

Ein Style Guide legt Regeln fest, die vor allem dazu dienen sollen, die Lesbarkeit und Wartbarkeit des Codes zu erhöhen. In Unternehmen entstehen häufig für die Wartung von Software größere Kosten als für deren Erstellung, und der weitaus größte Teil des Codes wird öfter gelesen als geschrieben. In Unternehmen werden daher i.d.R. umfangreiche Style Guides für die verschiedenen Programmiersprachen angewandt. Mit diesem kleinen Style Guide können Sie sich einige Aspekte eines verbreiteten Stils angewöhnen und die Einhaltung von Vorgaben üben, wie sie in der beruflichen Praxis vorkommen. Beachten Sie die zu diesem Style Guide gehörende Datei `StyleGuide-Beispiele.txt`, die verdeutlichende Beispiele zu den Regeln enthält und gleichzeitig zu diesem Text betrachtet werden sollte, aus der Sie aber auch mit dem Editor Teile kopieren können.

2 Grundregeln

2.1 Regeln werden nicht definiert, um gebrochen zu werden

Ein guter Style Guide kann auch dazu beitragen, die Qualität des Codes zu erhöhen. Wichtig ist allerdings, dass man das Ziel im Auge behält: Höhere Lesbarkeit und Wartbarkeit. Regelverstöße können mitunter erforderlich sein – aber sie müssen grundsätzlich ausführlich dokumentiert und begründet werden. Selbst wenn die Regeln nicht jedem gefallen, sind sie dennoch einzuhalten, da das angestrebte Ziel Vorrang hat vor persönlichen Vorlieben.

2.2 Keine Fehler oder Warnungen

Im Idealfall soll der Code sauber kompiliert werden können – das bedeutet, ohne Fehler oder Warnungen von einem beliebigen Compiler auf einer beliebigen Plattform zu erzeugen. Selbstverständlich ist dies schwer zu erreichen, da die Unterschiede zwischen Compilern und erst recht zwischen verschiedenen Plattformen oft zu groß sind.

Fehlermeldungen des Compilers weisen eindeutig auf Fehler im Code hin; aber auch Warnungen lassen in vielen Fällen darauf schließen, dass der bemängelte Code fehlerhaft oder zumindest verbesserungswürdig ist. Daher sollte man jeder Warnung auf den Grund gehen und sie beseitigen.

3 Dateien

3.1 Namenskonventionen

Dateinamen sollen „sprechend“ sein, d.h. man soll aus dem Namen auf den Inhalt schließen können. Beispielsweise soll eine Header-Datei nach der enthaltenen Klasse (C++) oder Modul (C) benannt sein. Die zugehörige Implementierungsdatei soll möglichst den gleichen Namen tragen bei unterschiedlicher Dateiendung. Folgende Konventionen für die Dateiendungen sind einzuhalten:

Dateityp	**Endung**
C++ Implementierungsdatei	.cpp
C++ Header-Datei (Schnittstellen)	.h
C++ Header-Datei (implementierungsspezifisch)	P.h (z.B. „VektorP.h“) (P für private)
C Implementierungsdatei	.c
C Header-Datei	.h
C Header-Datei (implementierungsspezifisch)	P.h
Windows Batchdatei	.bat

3.2 Dateiinhalte

Eine Datei soll eine logische Einheit darstellen wie eine Klasse oder eine Sammlung zusammengehöriger Klassen (Deklarationen oder Implementierungen). Falls die Implementierung einer einzelnen Klasse zu umfangreich ist, soll die Aufspaltung nachvollziehbar sein, z.B. alle privaten Funktionen in einer eigenen Datei.

Eine Datei soll nicht beliebig lang sein. (In Industrieprojekten kann eine maximale Länge von ca. 2000 – 2500 Zeilen festgelegt sein, diesen Umfang erreichen unsere Beispiele nie). Darüber hinaus wird der Code zu unübersichtlich, und damit leidet die Wartbarkeit. Weiterhin ist die Zeilenlänge auf 80 Zeichen zu limitieren. Längere Zeilen sind schlecht lesbar und sorgen zudem für Probleme, falls Ausdrucke erforderlich sind.

3.2.1 Autor

In jeder Datei ist zu Beginn in einer Kommentarzeile der Programmierer des Codes mit vollem Namen und gegebenenfalls mit Matrikelnummer zu nennen.

3.2.2 Inhalt von Header-Dateien

Header-Dateien enthalten Schnittstellenbeschreibungen, in C++ Deklarationen von Klassen und Typen, die eng zusammengehören. Header-Dateien, die Schnittstellen oder öffentliche Klassen deklarieren, im Folgenden mit „öffentliche Header-Dateien“ bezeichnet, sind in separaten Verzeichnissen abzulegen, getrennt von den Implementierungsdateien, während „private“ (implementierungsspezifische) Header-Dateien in den selben Verzeichnissen abzulegen sind wie die zugehörigen Implementierungen. Öffentliche Header-Dateien sollen niemals private Header-Dateien einschleusen.

3.2.2.1 „Macro Guards“

Jede Header-Datei ist durch einen sogenannten „Macro Guard“ vor Mehrfacheinschleusung zu schützen. Der Name des Makros setzt sich in größeren Systemen zusammen aus dem Verzeichnisnamen und dem Dateinamen, getrennt durch „_“; für die Praktikumsaufgaben reicht hier der Dateiname. Das erste und letzte Zeichen ist ebenfalls ein „_“. Alle Zeichen, die in Präprozessor-Direktiven ungültig sind, sind gleichfalls durch „_“ zu ersetzen, so beispielsweise der Punkt im Dateinamen.

```
#ifndef _VEKTOR_H_
#define _VEKTOR_H_
// Dateiinhalt
#endif /*!_VEKTOR_H_*/
```

Die `#ifndef`-Anweisung hat immer am Anfang der Datei zu stehen, die erforderliche abschließende `#endif`-Anweisung in der letzten Zeile der Datei. Diese ist mit einem Kommentar zu versehen, der den Namen des „Macro Guard“ enthält.

3.2.2.2 Include-Anweisungen

Alle Header-Dateien sollen ihrerseits ausschließlich die Header-Dateien einschleusen, die unbedingt erforderlich sind, um den Inhalt der Header-Datei korrekt deklarieren zu können. Die Reihenfolge der Include-Anweisungen ist festgelegt: die öffentlichen Header-Dateien stehen ganz oben, die privaten aus dem selben Verzeichnis ganz unten in der Liste. Als erstes stehen also beispielsweise System-Header-Dateien wie `<iostream.h>`.

3.2.2.3 Deklarationen

Unmittelbar auf die Include-Anweisungen folgen die Deklarationen der Header-Datei. Diese haben nach Möglichkeit diese Reihenfolge einzuhalten:

- `const`-Deklarationen.
- Vorwärtsdeklarationen von `class`, `struct`, `union`, `typedef`.
- Deklarationen von `struct` und `union`.
- `typedef`-Deklarationen.
- Klassendeklarationen. Innerhalb der Klasse sind die Sektionen `public`, `protected` und `private` in der genannten Reihenfolge zu deklarieren.
- Deklaration globaler Variablen (keine Definitionen). Globale Variablen sind jedoch möglichst zu vermeiden; stattdessen sind Konstanten oder Enumerationen zu verwenden, definiert in einer Klasse oder einem Namespace. Wirkliche globale Variablen sind mittels des Schlüsselwortes `extern` zu deklarieren.
- Externe Deklaration von Funktionen, die zu keiner Klasse gehören.

3.2.3 Inhalt von Implementierungsdateien

Eine Implementierungsdatei soll alles implementieren, was in der zugehörigen Header-Datei deklariert ist. Dies soll im Normalfall genau eine Klasse mitsamt eventuell erforderlichen weiteren Datentypen sein; in Ausnahmefällen können es auch mehrere stark zusammenhängende Klassen sein.

Private Klassen und Strukturen einer Implementierung sollen möglichst innerhalb der Implementierungsdatei deklariert werden anstatt in der Header-Datei; im Sinne einer besseren Lesbarkeit kann hierfür auch eine separate private Header-Datei vorgesehen werden.

Globale Variablen und Variablen, deren Gültigkeitsbereich auf die Datei beschränkt ist, stehen grundsätzlich vor den Funktionsdefinitionen. Vor jeder Funktionsdefinition ist ein doxygen-konformer Blockkommentar einzufügen (siehe Dokumentations-Richtlinien). Die einzelnen Funktionen können durch einen Kommentar getrennt werden.

3.3 Präprozessor

Pfadnamen von Header-Dateien sollen niemals absolute Pfade angeben, sondern immer relative Pfade. Hierbei ist die Verzeichnisstruktur im Home-Verzeichnis zu berücksichtigen.

3.3.1 Makros

Makros sollen nicht dazu verwendet werden, Konstanten zu definieren. Stattdessen sind `const`-Deklarationen oder Enumerationen zu benutzen. Dies dient vor allem der Typsicherheit des Wertes.

4 Namenskonventionen für Bezeichner

4.1 Allgemeine Regeln

- Bezeichner sollen aussagekräftig sein, d.h. sie müssen leicht verständlich und selbsterklärend sein. Auf diese Weise ist auch der Code leichter zu verstehen.

- Abkürzungen sind möglichst zu vermeiden, da sie von anderen Personen nicht in jedem Fall auf Anhieb verstanden werden. Gebräuchliche Bezeichner können abgekürzt werden; in diesen Fällen ist auf aussagekräftige Abkürzungen zu achten.
- Variablennamen, die aus lediglich einem einzelnen Zeichen bestehen, sind zu vermeiden, da hierunter die Wartbarkeit des Codes leidet. Ausnahmen sind Schleifenzähler und temporäre Zeigervariablen mit kurzer Lebensdauer.
- Bezeichner, die sowohl die Zeichen „0“ (null) als auch „O“ bzw. sowohl die Zeichen „l“ als auch „1“ (eins) enthalten, sind aufgrund der schlechten Unterscheidbarkeit der Zeichen unbedingt zu vermeiden.

4.2 Stil der Namensgebung

- Typnamen (dies schließt `class`, `enum` und `struct` ein) sowie Namen von Namespaces beginnen mit einem Großbuchstaben. Daraus folgt, dass auch Konstruktoren und Destruktoren grundsätzlich mit einem Großbuchstaben beginnen.
- Bei Bezeichnern, die aus mehreren Wörtern zusammengesetzt sind, ist zusätzlich zur Verbesserung der Lesbarkeit der erste Buchstabe jedes neuen Wortes groß zu schreiben. Die Verwendung von „_“ ist nicht zulässig.
- Variablennamen beginnen mit einem Kleinbuchstaben. Dies schließt auch `const`-Variablen ein.
- Funktionsnamen beginnen ebenfalls grundsätzlich mit einem Kleinbuchstaben.
- Bezeichner, die sich nur durch Groß- und Kleinschreibung unterscheiden, sind nicht zulässig.
- Bezeichner sollen nicht mit einem „_“ beginnen oder enden.

4.3 Reservierte Bezeichner

Bei den nachfolgend aufgelisteten Bezeichnern handelt es sich um reservierte Bezeichner, die nicht in selbstgeschriebenem Code verwendet werden sollen.

- Alle Bezeichner, die mit einem „_“ beginnen, sind reserviert.
- C++ reserviert alle Bezeichner, die mit zwei oder mehr „_“ beginnen.

5 „Whitespace“

Leerzeilen und Leerzeichen erhöhen die Lesbarkeit erheblich, sofern sie vernünftig angewandt werden, d.h. um logische Abschnitte im Code sichtbar zu trennen. Leerzeilen sind grundsätzlich in folgenden Situationen zu verwenden:

- Vor und nach dem Abschnitt der `#include`-Anweisungen.
- Beim Wechsel von Präprozessor-Direktiven zu Code und umgekehrt.
- Vor und nach jeder Klassen- oder Strukturdeklaration.
- Vor und nach jeder Funktionsdefinition.

- Vor einem Satz von `case`-Anweisungen in einer `switch`-Klausel, die logisch zusammengehören, also beispielsweise dasselbe Verhalten auslösen.

Leerzeichen sind folgendermaßen zu verwenden:

- Wird ein Schlüsselwort von einer öffnenden runden Klammer gefolgt, so sind sie durch ein Leerzeichen zu trennen.
- Zwischen einem Funktionsnamen und der Parameterliste steht dagegen kein Leerzeichen.
- Innerhalb einer Parameterliste steht nach jedem Komma ein Leerzeichen.
- Alle binären Operatoren mit Ausnahme von „.“ und „->“ sind von ihren Operanden durch Leerzeichen zu trennen. Dies betrifft also Zuweisungs-, arithmetische, Vergleichs- und logische Operatoren.
- Leerzeichen sollen niemals unäre Operatoren wie z.B. die Negation oder den Adressoperator von ihrem Operanden trennen.
- Die Anweisungen in einer `for`-Schleife sind durch Leerzeichen zu trennen.
- Zusätzliche Leerzeichen können verwendet werden, um mehrere aufeinanderfolgende Anweisungen auszurichten.

5.1 Einrückungen

Die Einrückung beträgt je Stufe 4 Zeichen. Die Verwendung von Tabulatoren zur Einrückung ist zulässig; die Tabulatorweite beträgt 8 Zeichen.

5.2 Lange Zeilen

In manchen Fällen ist eine Zeile länger als die zulässigen 80 Zeichen, beispielsweise ein Funktionsaufruf mit langer Parameterliste.

Falls eine lange Zeile in mehrere Zeilen zerlegt werden muss, ist darauf zu achten, dass der Code lesbar und verständlich bleibt. Auf die öffnende Klammer folgt ein Zeilenumbruch, die schließende Klammer steht in einer eigenen Zeile.

5.3 Kommentare

Kommentare sind von zentraler Bedeutung für die Verständlichkeit und Wartbarkeit des Codes. Sie sollen einen Überblick über den Zweck des Codes sowie zusätzliche Informationen geben, die nicht unmittelbar aus dem Code selbst hervorgehen. Verschachtelte Kommentare sind nicht zulässig.

Kommentare im C++-Stil (eingeleitet durch `//`) sind solchen im C-Stil (eingeleitet durch `/*`, abgeschlossen durch `*/`) vorzuziehen. Beide Formen sind aber erlaubt.

5.3.1 Blockkommentare

Diese Kommentare stehen in separaten Zeilen. Der Text wird um mindestens ein Zeichen eingerückt; die letzte Zeile des Kommentarblocks ist eine Leerzeile. Blockkommentare stehen bei Bedarf vor der kommentierten Anweisung.

C-Kommentare, die sich über mehrere Zeilen erstrecken, beginnen in jeder neuen Zeile mit einem „*“; diese sind unter dem einleitenden „/*“ auszurichten. Das abschließende „*/“ ist entsprechend auszurichten.

Globale Blockkommentare beginnen in der ersten Spalte. Blockkommentare innerhalb von Klassen oder Funktionen werden auf die gleiche Stufe eingerückt, wie Code einzurücken wäre.

5.3.2 Abschließende Kommentare

Diese sind nach Variablendeklarationen und einzeiligen Typdeklarationen zulässig. Hierbei ist die doxygen-Notation `///<` zu verwenden.

Die Kommentare sind weit genug nach rechts zu rücken, um sie deutlich sichtbar vom Code zu trennen. Falls mehrere aufeinanderfolgende Zeilen solche Kommentare erhalten, sollten diese linksbündig untereinander ausgerichtet werden.

5.4 Kommentierung von Funktionen

Vor jeder Funktion steht ein doxygen-konformer Kommentarblock (siehe Dokumentations-Richtlinien).

6 Typen

6.1 Konstanten

- Numerische Konstanten sollen an genau einer Stelle definiert werden, um möglichst einfach geändert werden zu können.
- Die Verwendung von `#define` zur Konstantendefinition wird in C++-Programmen untersagt. Stattdessen ist das Schlüsselwort `const` oder eine Enumeration zu verwenden. Dies dient vor allem der Typsicherheit des Wertes.
- Konstanten wie 0, 1 oder –1 können ohne Konstantendefinition verwendet werden.
- Größenangaben sind, wenn möglich, über den `sizeof`-Operator auszudrücken. Zur Erhöhung der Lesbarkeit ist der Operand des `sizeof`-Operators stets in Klammern zu setzen.
- Der `sizeof`-Operator ist dabei, wenn möglich, auf Objekte anzuwenden, nicht auf Typen. (Auf diese Weise sind `sizeof`-Operationen auch dann noch korrekt, wenn sich der Typ des Objekts geändert hat).

6.2 Enumerationen

Der Bezeichner einer Enumeration beginnt immer mit einem Großbuchstaben (siehe auch Abschnitt 4.2). Das Layout einer Enumeration gleicht dem einer Struktur, wenn die Enumeration sich über mehrere Zeilen erstreckt oder explizite Initialisierungen enthält.

6.3 Strukturen und Unions

Der Bezeichner einer Struktur oder Union beginnt grundsätzlich mit einem Großbuchstaben (siehe auch Abschnitt 4.2). Die Deklarationen der Member sind um eine Stufe einzurücken.

6.4 Klassen

6.4.1 Allgemeines

- Nur Funktionen sollen `public` oder `protected` sein; Member-Variablen sind in der Regel als `private` zu deklarieren. Für den Lese- und Schreibzugriff auf die einzelnen Member-Variablen sind entsprechende Funktionen zu definieren, die als `public` oder `protected` deklariert werden.
- Die Klasse soll als „Black Box“ agieren, d.h. ihre Schnittstelle soll genau die Informationen bzw. Operationen zur Verfügung stellen, die unbedingt benötigt werden, um die Klasse verwenden zu können, alles andere soll verborgen werden.
- Die Member-Funktionen sind immer dann als `const` zu deklarieren, wenn sie den Zustand des Objekts unverändert lassen.
- Die Sektionen `public`, `protected` und `private` sind in der genannten Reihenfolge zu deklarieren, wobei diese Schlüsselwörter eine halbe Stufe (entspricht 2 Leerzeichen) eingerückt werden. Jede Sektion erscheint genau einmal in jeder Klasse.

6.4.2 Automatisch verfügbare Member-Funktionen

- Man sollte immer einen Konstruktor, einen Copy-Konstruktor und einen Destruktor definieren, selbst wenn die Implementierung leer ist. Andernfalls werden sie von C++ automatisch zur Verfügung gestellt (in der `public`-Sektion), was in bestimmten Fällen unerwünscht sein kann.
- Falls eine Klasse den Copy-Konstruktor oder den Zuweisungsoperator nicht benötigt, sollte man eine Implementierung in der `private`-Sektion vornehmen. Auf diese Weise verhindert man eine Verwendung durch den Benutzer, da sie andernfalls von C++ zur Verfügung gestellt werden.

6.4.3 Operator-Overloading

- Die Deklarationen überladener Funktionen sollen aus Gründen der Übersichtlichkeit innerhalb der Klassendeklaration gruppiert werden.
- Wird ein Operator eines logisch zusammengehörigen Satzes überladen, so sind die verbleibenden Operatoren dieses Satzes ebenfalls zu überladen. Wird beispielsweise der ==-Operator überladen, so ist auch der `!=`-Operator zu überladen. Andernfalls wird der Benutzer verwirrt.
- Das Überladen von Funktionen ist nur dann zu verwenden, wenn die verschiedenen Versionen prinzipiell das gleiche Verhalten implementieren.
- Man soll niemals explizite Casts verwenden, um einen Zeiger oder eine Referenz auf ein Objekt einer abgeleiteten Klasse in einen Zeiger oder eine Referenz auf eine öffentliche Basisklasse zu konvertieren. Dies geschieht implizit; explizite Casts können im ungünstigsten Fall eine Konvertierung zwischen vollkommen verschiedenen Typen durchführen.

7 Variablen

Variablennamen beginnen mit einem Kleinbuchstaben, dies gilt auch für Namen von Konstanten. Globale Variablen sollen mit dem Präfix `global_` beginnen (siehe auch Abschnitt 4.2 zur Namensgebung).

- Die Typnamen sind auf die aktuelle Stufe einzurücken.
- In jeder Zeile soll nur eine Variablendefinition stehen. Eine Ausnahme sind stark zusammenhängende Variablen wie beispielsweise Koordinatenpaare o.ä. In diesen Fällen dürfen mehrere Variablen in einer Zeile deklariert werden. Dies erleichtert auch spätere Typänderungen.
- Lokale Variablen sollen nicht den gleichen Namen haben wie Variablen, die bereits auf höheren Ebenen definiert wurden, um Missverständnisse zu vermeiden.

7.1 Initialisierungen

- Wenn eine Initialisierungsliste – für Arrays oder Strukturen – nicht vollständig auf eine Zeile passt, so sind die Initialisierungswerte in separaten Zeilen zu platzieren und eine Stufe einzurücken.
- Initialisierungslisten dürfen kein abschließendes Komma beinhalten.
- Objekte, die nur einen Initialisierungswert benötigen, sollen keine geschweiften Klammern verwenden.

8 Funktionen

Funktionsnamen beginnen mit einem Kleinbuchstaben (siehe auch Abschnitt 4.2).

8.1 Funktionsdeklarationen

- Der Rückgabetyp steht in der selben Zeile wie der Funktionsname.
- Funktionsparameter werden wie folgt aufgelistet: Jeder Parameter steht in einer eigenen Zeile und wird um eine Stufe eingerückt. Die schließende Klammer steht in einer eigenen Zeile und wird nicht eingerückt. Dieser Stil ermöglicht zum einen die Kommentierung der Parameter, zum anderen trägt er im Falle einer längeren Parameterliste entscheidend zur Lesbarkeit bei. Zudem können die Parameter einfacher umsortiert, ergänzt oder entfernt werden.
- In der Funktionsdeklaration sind auch die Parameternamen, nicht nur die Datentypen, zu nennen. Sämtliche Parameternamen in Deklaration und Definition einer Funktion müssen übereinstimmen.
- Reine Eingabeparameter, die von der Funktion nicht verändert werden, müssen entweder als Wert oder als `const&` übergeben werden.
- Parameter, die von der Funktion verändert werden dürfen, sollen als Referenzen übergeben werden. Die Übergabe als Zeiger ist nicht verboten, sollte aber nach Möglichkeit vermieden werden.

8.2 Funktionsdefinitionen

- Vor jeder Funktionsdefinition steht ein doxygen-konformer Kommentarblock. Dieser enthält insbesondere eine kurze Beschreibung des Zwecks der Funktion (siehe hierzu die Dokumentations-Richtlinien).
- Der Rückgabetyp der Funktion ist explizit zu spezifizieren.
- Der Klassenname steht in der selben Zeile wie der Funktionsname, wenn es sich um eine Member-Funktion einer Klasse handelt.
- Alle lokalen Deklarationen sowie der Code innerhalb des Funktionsrumpfes wird um eine Stufe eingerückt.

9 Anweisungen

Jede Zeile enthält höchstens eine Anweisung. Dies trägt zur Lesbarkeit des Codes bei.

9.1 Anweisungsblöcke

- Anweisungsblöcke in geschweiften Klammern sind um eine weitere Stufe einzurücken. Die schließende geschweifte Klammer steht in einer eigenen Zeile.
- Geschweifte Klammern sind möglichst auch für Einzelanweisungen zu verwenden, wenn diese Teil einer Kontrollstruktur wie z.B. einer `if/else`-Anweisung sind. Dies erleichtert das Hinzufügen oder Löschen von Anweisungen erheblich.

9.2 for-Schleifen

- Wenn die drei Anweisungen einer `for`-Schleife nicht in eine Zeile passen, so sind sie in drei separate Zeilen zu schreiben. Alternativ kann auch die Initialisierung der Zählvariablen vor die Schleife gezogen werden.
- Die Deklaration der Zählvariablen direkt in der Initialisierungsanweisung der Schleife ist akzeptabel, sofern die Variable nicht außerhalb der Schleife verwendet wird.

9.3 Endlosschleife

Die Endlosschleife ist als `for`-Schleife in der folgenden Form zu implementieren:

```
for (;;)
{
    ...
}
```

```
for (;;) {
    ...
}
```

Beide Schreibweisen sind zulässig (siehe Abschnitt 10.1 zur Klammernsetzung).

9.4 switch-Anweisungen

- Die `case`-Klausel und die von ihnen kontrollierten Anweisungen stehen in separaten Zeilen.
- Die `case`-Klauseln werden um eine halbe Stufe eingerückt, die von ihnen kontrollierten Anweisungen eine volle Stufe.

- Das Durchlaufverhalten der `switch`-Anweisung ist möglichst selten zu benutzen. Einzige Ausnahme bilden Mehrfach-Labels (siehe Beispiel). Falls das Verhalten in anderen Fällen genutzt wird, so ist dies an der Stelle, an welcher normalerweise die `break`-Anweisung zu erwarten wäre, entsprechend zu kommentieren.
- Die `break`-Anweisung kann, sofern möglich und sinnvoll, durch eine `return`-Anweisung ersetzt werden.
- `switch`-Anweisungen, die eine Variable auf `enum`-Werte prüfen, sollen immer einen `default`-Fall enthalten. Dieser soll über das `assert`-Makro einen Fehler melden. Dies verhindert, dass einer Enumeration ein neuer Wert hinzugefügt wird, der aber in der `switch`-Anweisung nicht abgefragt wird.
- `switch`-Anweisungen, die nicht auf Enumerationen prüfen, sollen immer dann einen `default`-Fall enthalten, wenn angenommen wird, dass die geprüfte Variable nur bestimmte Werte annehmen kann. Der `default`-Fall muss in diesem Fall zumindest die Aufmerksamkeit auf den unerwarteten Wert lenken.

9.5 Die goto-Anweisung

Die Verwendung der `goto`-Anweisung ist möglichst zu vermeiden. In vielen Fällen reicht es aus, ein anderes Sprachkonstrukt zu verwenden oder eine Funktion in mehrere kleine Funktionen zu trennen. Falls die Verwendung von `goto` unumgänglich ist, sind folgende Regeln zu beachten:

- Der Sprung zu einem Label innerhalb eines inneren Blocks ist nicht zulässig. Ein solcher Sprung kann Variableninitialisierungen umgehen.
- Das Label muss in einer separaten Zeile stehen; der Name beginnt in der ersten Spalte. Es muss in jedem Falle eine Anweisung folgen, selbst wenn dies die leere Anweisung ist (nur ein Semikolon).

10 Verschiedenes

10.1 Allgemeines

- Globale Variablen sind nach Möglichkeit zu vermeiden. An ihrer Stelle können oft statische Variablen mit Dateigültigkeit oder Klassen-Membervariablen verwendet werden.
- Die Boole'sche Negation darf nicht für nicht-Boole'sche Ausdrücke verwendet werden; dies betrifft insbesondere die Prüfung eines Zeigers auf `NULL`.
- Man darf sich nicht darauf verlassen, dass der Wertebereich eines `int` über den Wertebereich eines `short int` hinausgeht. In Fällen, in denen ein `short int` ausreichend wäre, sollte dennoch `int` verwendet werden, da dies die Wortbreite des Prozessors ausnutzt und somit für höhere Effizienz sorgen kann. Falls ein größerer Wertebereich benötigt wird, so ist ein `long int` zu verwenden.

10.2 Klammernsetzung

Für das Setzen geschweifter Klammern um Anweisungsblöcke von Kontrollstrukturen sind zwei „Stile“ zugelassen: Die öffnende Klammer kann entweder in einer Zeile mit der

Kontrollanweisung stehen oder in einer separaten Zeile. Innerhalb eines Programms hat man sich für einen dieser Stile zu entscheiden; ein Gemisch ist nicht akzeptabel.

10.3 Vergleich mit Null

- Die normale Form eines Boole'schen Vergleichs ist die folgende:

  ```
  if (boolescheVariable)
  ```

  ```
  if(!boolescheVariable)
  ```

  Diese Form ist ausschließlich für Boole'sche Ausdrücke zulässig, alle anderen Datentypen (int, char, Zeiger etc.) müssen explizit mit einem entsprechenden Wert verglichen werden.

- Vergleich für Datentyp `char`:

  ```
  if (charVariable != '\0')
  ```

- Ganzzahlige Werte:

  ```
  if (ganzzahlVariable == 0)
  ```

- Gleitkommazahlen:

  ```
  if (gleitkommaVariable >= 0.0)
  ```

  Im Falle von Gleitkommazahlen ist Vorsicht geboten. Die Verwendung des Gleichheits- bzw. Ungleichheitsoperators ist generell nicht zu empfehlen. Stattdessen sind die Operatoren <= und >= zu verwenden. Um einen Test auf „Gleichheit" zweier Werte durchzuführen, ist eine minimale Abweichung zuzulassen.

- Prüfung eines Zeigers auf `NULL`: Diese Prüfung sollte durch Vergleich des Zeigers mit dem Zahlwert 0 geschehen. Das ANSI-C-Makro `NULL` ist in C++ nicht portabel und soll daher nicht verwendet werden.

10.4 Gebrauch und Missbrauch von inline

- Die Verwendung von Inline-Funktionen in öffentlichen Klassendeklarationen ist nicht zulässig.
- Inline-Funktionen sollen generell nicht in Klassendeklarationen implementiert werden. Stattdessen soll die Funktion in der Klasse normal deklariert werden, anschließend wird eine Inline-Implementierung am Ende der Header-Datei zur Verfügung gestellt. Dies erleichtert den Wechsel zwischen Inline- und normaler Implementierung.
- Die Implementierung von Funktionen als Inline soll grundsätzlich erst dann geschehen, wenn das Programm implementiert und getestet wurde.
- Eine Funktion sollte nur dann als Inline implementiert werden, wenn dieser Schritt definitiv einen Performance-Gewinn bringt.