Mängel der Programmierregeln des Eisenbahn-Bundesamtes

Martin Lottermoser
Greifswaldstraße 28
38124 Braunschweig
http://home.vrweb.de/martin.lottermoser

2009–07–09 (Version 1.15)

Gewidmet einem Gutachter, der sich sehr echauffierte, als er hörte, dass mein Code nicht strikt den Regeln des EBAs folgte: ohne ihn hätte ich dies nicht aufgeschrieben, da ich bis dahin die Mängel für offensichtlich hielt.

Inhaltsverzeichnis

1 Einleitung

1.1 Änderungsübersicht

KM-Version	Datum	Änderungen
1.7	2003–10–13	Neuerstellung und erste Weitergabe
1.8	2003–12–06	Korrekturen und Klarstellungen
1.12	2004–03–03	Eingeschränkt auf die EBA-Vorgaben, Vertraulichkeitsgrad heruntergestuft, kleinere Umformulierungen.
1.15	2009–07–09	Abgeglichen mit den EBA-Regeln von 2003, projektspezifische Hinweise entfernt, inhaltliche Ergänzungen und Korrekturen.

1.2 Zielgruppe, Inhalt und Zweck

Dieses Dokument richtet sich an Personen, die im Rahmen der Erstellung sicherheitsrelevanter Software für Eisenbahnen mit Programmierregeln für C oder C++ zu tun haben, die auf den Regeln des deutschen Eisenbahn-Bundesamtes basieren [11, 12]. Dies betrifft Anwender solcher Regeln (Entwickler, Verifizierer und Validierer) sowie Personen, die mit der Aufstellung oder Änderung firmenspezifischer Regeln zu tun haben.

Der Inhalt des vorliegenden Dokumentes kann als Review-Bericht für die Vorgaben des Eisenbahn-Bundesamtes aufgefasst werden.

Der Hauptzweck dieses Dokumentes ist es aber, die Qualität von Quelldateien zu verbessern, indem vor allem weniger erfahrene Mitarbeiter auf qualitätsmindernde Eigenschaften der EBA-Regeln aufmerksam gemacht werden. Langfristig kann es auch wünschenswert sein, das Dokument als Ausgangspunkt für eine Überarbeitung der eigenen Regeln zu nutzen. Bis dahin soll es zumindest als Kommentar zu den EBA-Regeln einsetzbar sein und es dadurch erleichtern, missverständliche oder fehlerhafte Aussagen in den Regeln zu erkennen; allerdings ist es unwahrscheinlich, dass ich alle Mängel aufgedeckt habe.

1.3 Literatur

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Series in Object-Oriented Software Engineering. Redwood City (California, USA), etc.: The Benjamin/Cummings Publishing Company, Inc., zweite Auflage, 1994. ISBN 0-8053-5340-2.

[2] Alan Bridger, Jim Pisano. *C++ Coding Standards.* Atacama Large Millimeter Array, August 2001. Dokument ALMA-SW-0010, Revision 5.

[3] L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington, Henry Spencer. *Indian Hill C Style and Coding Standards — as amended for U of T Zoology UNIX*, Mai 1988. Dies ist eine von H. Spencer kommentierte Fassung der *Indian Hill C Style and Coding Standards*, ursprünglich entwickelt in den Indian Hill Laboratories von AT&T.

[4] L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington, Henry Spencer, David Keppel, Mark Brader. *Recommended C Style and Coding Standards*, Juni 1990. Version 6.0. Dies ist eine erweiterte Fassung von [3].

[5] James O. Coplien. *Advanced C++.* Reading (Massachusetts, USA), etc.: Addison-Wesley, 1992.

[6] The Corelinux Consortium. *C++ Coding Standards*, September 2000. Revision 1.6.

[7] Jerry Doland, Jon Valett. *C Style Guide.* Software Engineering Laboratory (SEL) am Goddard Space Flight Center (GSFC) der National Aeronautics and Space Administration (NASA), Greenbelt (Maryland, USA), August 1994. SEL-94-003.

[8] Eisenbahn-Bundesamt, München. *Technische Grundsätze für die Zulassung von Sicherungsanlagen (Mü 8004).*

[9] Eisenbahn-Bundesamt. *Technische Anforderungen an Sicherungsanlagen der Elektronik — Vorgaben zum Einsatz der objektorientierten Programmiersprache C++*, Januar 1999. Teil 42 730 in [8]. Dieser Stand wird als Entwurf bezeichnet.

[10] Eisenbahn-Bundesamt. *Technische Anforderungen an Sicherungsanlagen der Elektronik — Vorgaben zum Einsatz der problemorientierten Programmiersprache C*, Januar 1999. Teil 42 720 in [8]. Dieser Stand wird als Entwurf bezeichnet.

[11] Eisenbahn-Bundesamt. *Technische Anforderungen an Sicherungsanlagen der Elektronik — Vorgaben zum Einsatz der problemorientierten Programmiersprache C*, August 2003. Teil 42 720 in [8].

[12] Eisenbahn-Bundesamt. *Technische Anforderungen an Sicherungsanlagen der Elektronik — Vorgaben zum Einsatz der problemorientierten Programmiersprache C++*, August 2003. Teil 42 730 in [8].

[13] European Laboratory for Particle Physics, Genf. *ATLAS C++ Coding Standard — Specification*, Juli 2003. Version 1.2, Issue 1, ID ATL-SOFT-2002-001.

[14] Flight Software Branch am Goddard Space Flight Center (GSFC) der National Aeronautics and Space Administration (NASA), Greenbelt (Maryland, USA). *C++ Coding Standard*, Dezember 2003. Dokument 582-2003-004, Version 1.0.

[15] Flight Software Branch am Goddard Space Flight Center (GSFC) der National Aeronautics and Space Administration (NASA), Greenbelt (Maryland, USA). *C Coding Standard*, November 2005. Dokument 582-2000-005, Version 1.1.

[16] Keith Gabryelski. *Wildfire C++ Programming Style — With Rationale.* Wildfire Communications, Inc., 1997.

[17] Mats Henricson, Erik Nyquist. *Programming in C++ — Rules and Recommendations.* Ellemtel Telecommunication Systems Laboratories, Älvsjö (Schweden), April 1992. Document No. M 90 0118 Uen, Rev. C.

[18] Mats Henricson, Erik Nyquist. *Industrial Strength C++.* Prentice Hall PTR, 1997. ISBN 0-13-120965-5.

[19] International Organization for Standardization/International Electrotechnical Commission. *Programming languages — C++*, erste Auflage, September 1998. International Standard ISO/IEC 14882:1998(E).

[20] International Organization for Standardization/International Electrotechnical Commission. *Programming languages — C*, zweite Auflage, Dezember 1999. International Standard ISO/IEC 9899:1999(E).

[21] International Organization for Standardization/International Electrotechnical Commission. *Programming languages — C, Technical Corrigendum 1*, September 2001. Reference Number ISO/IEC 9899:1999/Cor.1:2001(E).

[22] International Organization for Standardization/International Electrotechnical Commission. *Programming languages — C++*, zweite Auflage, Oktober 2003. International Standard ISO/IEC 14882:2003(E).

[23] International Organization for Standardization/International Electrotechnical Commission. *Programming languages — C, Technical Corrigendum 2*, November 2004. Reference Number ISO/IEC 9899:1999/Cor.2:2004(E).

[24] International Organization for Standardization/International Electrotechnical Commission. *Programming languages — C, Technical Corrigendum 3*, November 2007. Reference Number ISO/IEC 9899:1999/Cor.3:2007(E).

[25] Andrew Koenig. *C Traps and Pitfalls.* AT&T Bell Laboratories, Murray Hill (New Jersey, USA), 1989.

[26] Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Dezember 2005. Document Number 2RDU00001 Rev C.

[27] Richard Stallman, et al. *GNU Coding Standards.* Free Software Foundation, Oktober 2001.

1.4 Abkürzungen

ANSI	American National Standards Institute
EBA	Eisenbahn-Bundesamt
IEC	International Electrotechnical Commission
ISBN	International Standard Book Number
ISO	International Organization for Standardization
POD	Plain old data [22, Abschnitt 1.8]
XPG	X/Open Portability Guide

1.5 Konventionen

Zur leichteren Bezugnahme und Bewertung werden die gefundenen Mängel durchnummeriert und klassifiziert:

> **M42 (*K*): Kurze Charakterisierung des Mangels Nummer 42.**
>
> Gegebenenfalls folgt hier eine detaillierte Beschreibung oder eine Begründung, warum es sich um einen Mangel handelt.

Die Angabe *K* dient einer groben Klassifikation, wobei Großbuchstaben einen nach meiner Meinung schweren Mangel bezeichnen sollen und Kleinbuchstaben einen leichten:

- L/l: logische Fehler (Widersprüche oder Lücken)
- P/p: unzureichender Praxisbezug oder Diskrepanz zu bekanntlich guter Programmierpraxis
- S/s: Verstoß gegen oder Lücken in Bezug auf die Sprachdefinition
- ü: überflüssige Regeln oder Aussagen
- V/v: Verständlichkeit (missverständlich oder unverständlich)

Auf syntaktische Größen der Sprachdefinitionen wird den Standards folgend *auf-diese-Weise* Bezug genommen.

1.6 Copyright und Lizenz

2 Wünschenswerte Eigenschaften von Programmierregeln

Um den Kontext zu erläutern, in dem ich Programmierregeln beurteilen möchte, und um nicht im Folgenden an mehreren Stellen die gleiche Begründung geben zu müssen, stelle ich hier zunächst einige Anforderungen an Programmierregeln auf, die meines Erachtens für gute Regeln zumindestens wünschenswert sind. Abweichungen von diesen Forderungen behindern oft indirekt die Verbesserung der Qualität der Quelldateien (beispielsweise durch Verunsicherung der Entwickler) oder haben sogar direkt qualitätsmindernden Einfluss auf den erstellten Code.

Die meisten dieser Anforderungen werden von in Industrie, Forschung und offener Software-Entwicklung üblichen Programmierregeln erfüllt [2, 3, 4, 6, 7, 13, 14, 15, 16, 17, 18, 26, 27].

Klarheit. Die Regeln müssen so formuliert sein, dass erkennbar ist, was gefordert wird. Insbesondere muss dazu formal eindeutig zwischen Forderungen, Empfehlungen und Ratschlägen (o. ä.) getrennt werden.

Verständlichkeit. Die Regeln müssen so formuliert sein, dass sie von den Entwicklern verstanden werden können. Für komplexere Regeln sollten daher Beispiele angegeben werden.

Zurückhaltung. Man kann leicht in den Irrtum verfallen, dass es für ein Problem nur eine einzige Lösung gibt, nämlich die, die man selber gerade gefunden hat. Oft gibt es aber mehrere, die sich nicht einmal in ihrer Qualität (vor allem bezüglich der Vermeidung von Fehlern) unterscheiden müssen aber trotzdem anders aussehen.

Autoren von Programmierregeln sollten sich dieses Sachverhaltes bewusst sein und nicht unnötig detaillierte Regeln aufstellen. Im Allgemeinen wird dies bedeuten, dass sie versuchen müssen, inhaltliche statt formaler Regeln vorzugeben, die den Anwendern einen gewissen Gestaltungsspielraum lassen.

Auf keinen Fall dürfen Autoren in dieser Situation zu dem Schluss kommen, dass irgendeine Regel immer noch besser ist als gar keine. Dieser Weg der Vereinheitlichung um jeden Preis stößt Entwickler unnötig vor den Kopf, falls eine Regel willkürlich ihren persönlichen Präferenzen widerspricht, und führt insgesamt zu einer verringerten Akzeptanz der Regeln.

Vereinheitlichung ist auch meines Erachtens kein eigenständig sinnvolles Ziel sondern nur ein Mittel zum Erreichen von Zielen wie Verständlichkeit oder Wartbarkeit. Der Glaube, dass alles besser wird, wenn alle im Gleichschritt marschieren, gehört auf den Müllhaufen der Geschichte aber nicht in die Software-Entwicklung: gute Software erhält man von guten Entwicklern, nicht über Programmierregeln, die festlegen, wann um wieviele Leerzeichen eingerückt werden muss.

Begründung. Nicht unmittelbar einsehbare Regeln müssen begründet werden.

Diese Forderung hat mehrere Ursachen:

- Ein autoritärer Stil ist unhöflich, wenn nicht gar beleidigend („Die Leser sind sowieso zu dumm, die Gründe zu verstehen.“), und in Deutschland erfreulicherweise auch schon lange nicht mehr zeitgemäß.
- Eine Begründung hilft bei der Weitergabe mühsam erworbenen Know-Hows. Diese Möglichkeit sollte man gerade in der Industrie nicht leichtfertig ignorieren, da die Alternative zu höheren Kosten führt.

  Man bedenke dabei auch, dass Programmierregeln aufgrund ihres Themas gezwungen sind, sich auf Aspekte zu konzentrieren, mit denen die Leser direkt zu tun haben. Mitgelieferte Begründungen sind dann besonders eingängig, d. h. Programmierregeln können eine effiziente Art der Weitergabe von Know-How sein.
- Vor allem Entwickler mit geringer Erfahrung werden verunsichert, wenn sie den Zweck einer Regel nicht verstehen.
- Nur begründete Regeln werden ernst genommen.

  Es erhöht die Akzeptanz von Regeln, wenn die Autoren in einer Diskussion nachweisen, dass sie sich ihre Anforderungen überlegt und vor allem auch Alternativen betrachtet haben, statt einfach ihre persönlichen Vorlieben als Regeln zu verhängen.
- Begründungen erleichtern es den Autoren der Regeln, sich von vornherein auf sachlich gerechtfertigte Vorgaben zu beschränken.

- Auch Autoren von Programmierregeln machen Fehler. Eine Begründung erleichtert es dem Leser, die Argumentation des Autors zu überprüfen und zu entscheiden, ob ein vermutetes Problem auf mangelndem eigenen Verständnis beruht oder ob es sich um ein Versehen des Autors handelt, beispielsweise um auf den konkreten Fall nicht zutreffende implizite Annahmen, so dass von einer Regel mit gutem Gewissen abgewichen werden kann.
- Eine Regel kann durch Änderungen an der Sprachdefinition obsolet werden. Falls eine Begründung der Regel vorliegt, ist das sofort erkennbar, so dass die Wartung der Regeln erleichtert wird.

Hintergrundinformation. Gerade für Entwickler mit geringer Erfahrung ist es sehr hilfreich, in den Programmierregeln auch Hinweise auf weiterführende Literatur zu „guter“ Software-Entwicklung zu finden.

Entscheidbarkeit. Die Regeln müssen so formuliert sein, dass für jedes in Frage kommende Programm entscheidbar ist, ob es die Regeln erfüllt oder nicht.

Dies erfordert vor allem eine Charakterisierung des Begriffes „in Frage kommendes Programm“, d. h. die Angabe einer Grundmenge (möglichst mit eindeutiger lexikalischer und syntaktischer Spezifikation), aus der dann durch zusätzliche Auflagen die nach den Regeln zulässigen Programme ermittelt werden.

Regelwerke, die nur mit Verboten bestimmmter syntaktischer Konstrukte arbeiten, haben es beim zweiten Schritt leichter als solche, die erlaubte Konstrukte aufzählen, da letztere korrekt in die Restsyntax eingebettet werden müssen. (Wenn man etwas verbietet, ist es überall verboten; wenn man etwas erlaubt, ist es dagegen meist nur an Stellen erlaubt, die nicht einfach charakterisierbar sind.)

Erfahrung. Programmierregeln sollten nur von Entwicklern aufgestellt werden, die mehrjährige Erfahrung mit der betroffenen Programmiersprache in größeren Projekten haben. Diese Erfahrung sollte vor allem auch die Wartung nicht selbst geschriebenen Codes umfassen. Es ist dann fast selbstverständlich, dass die Autoren bereits andere Programmierregeln kennengelernt haben.

Die folgenden Punkte sind Indizien dafür, dass diese Anforderungen bei einem vorliegenden Regelwerk nicht erfüllt sind:

- fehlerhafte Aussagen zur Programmiersprache
- fehlende Aussagen zu bekanntermaßen problematischen Konstrukten der Programmiersprache
- unnötige Erläuterungen von elementaren Eigenschaften der Programmiersprache
- logisch überflüssige Regeln für Vorgaben, die die Sprachdefinition bereits verhängt, auch wenn sie ein Compiler nicht immer überprüfen kann, so dass eine Verletzung erst zur Laufzeit offenbar wird[1]
- praxisferne Regeln (zuviel Aufwand bei zuwenig oder gänzlich ohne Nutzen für die Qualität des Quell-Codes, Verbot manchmal notwendiger Funktionalität)
- Betonung formaler gegenüber inhaltlichen Regeln (beispielsweise ausführliche Vorgaben zur Bildung von Bezeichnern aber dafür keine Forderung nach Konsistenz von Code und Kommentaren)
- fehlende Begründungen für Vorgaben
- keine Literaturhinweise auf andere Programmierregeln, die bei der Erstellung zu Rate gezogen wurden

Strukturierung. Unter anderem für eine wenigstens teilweise automatisierte Überprüfung ist es wünschenswert, bei den Regeln klar nach lexikalischer, syntaktischer und semantischer Ebene zu trennen, wie es auch bei der Beschreibung von Programmiersprachen üblich ist.

Eine weitere nützliche Klassifizierung, vor allem bei einem Regelwerk mit dem Anspruch eines großen Anwendungsbereiches, ist die Einteilung nach mit einer Regel angestrebten Zielen. Dies vereinfacht eine Auf- oder Abstufung der Dringlichkeit von Regeln in Abhängigkeit von projektspezifischen Zielen; bei der Entwicklung eines Betriebssystems oder eines Compilers sind beispielsweise Regeln, deren Ziel eine Verbesserung der Portabilität des erstellten Codes ist, oft wenig nützlich oder sogar prinzipiell nicht umsetzbar.

[1] Es kann allerdings zur Unterstützung guter Praxis sinnvoll sein, so etwas in eine Zusammenstellung häufig gemachter Fehler oder in Checklisten für Inspektionen aufzunehmen. Eigene Programmierregeln dafür sind aber meist nicht gerechtfertigt, da entsprechende Regeln bereits in der Sprache vorhanden sind. Wenn man sie trotzdem anführen will (um beispielsweise den Programmierern das Lesen des Standards zu ersparen), sollte man sie auf jeden Fall besonders kennzeichnen.

Empfehlenswert ist auch, Regeln besonders zu kennzeichnen, die zu einem gewissen Grade willkürlich sind, beispielsweise Namenskonventionen und Layout-Regeln. Diese Regeln erbringen für ein einzelnes Code-Stück selten eine direkte Qualitätsverbesserung sondern versuchen, eine bessere Verständlichkeit durch projektweite Vereinheitlichung zu erreichen. In projektübergreifenden Programmierregeln müsste dafür eigentlich sogar die Forderung reichen, dass ein Projekt solche Konventionen aufstellen sollte.

Für Inspektionen ist es ferner hilfreich, jede Regel mit einem eindeutigen Identifikator zu versehen.

Ausweichregel. Autoren von Programmierregeln können nicht damit rechnen, alle Anwendungsfälle voraussehen zu können. Realistisch denkende Autoren werden daher eine Ausweichregel aufstellen, die Abweichungen von den Regeln unter genau spezifizierten Bedingungen gestattet.

3 Vorgaben des Eisenbahn-Bundesamtes zum Einsatz von C

Dieser Abschnitt befasst sich mit den Vorgaben des Eisenbahn-Bundesamtes zum Einsatz der Programmiersprache C [11]. Da diese Vorgaben nahezu identisch zu einem vorausgegangenen Entwurf [10] sind, treffen die hier angeführten Aussagen auch auf die ältere Fassung zu.

3.1 Inhalt der Vorgaben

Die folgenden Unterabschnitte enthalten jeweils Aussagen zu den gleichnamigen Abschnitten in [11].

3.1.1 Vorbemerkung

M1 (S): Der zugrundeliegende Sprachstandard ist nicht eindeutig spezifiziert.

Die Vorgaben sagen:

> Der zugelassene Sprachumfang von C ist eine Untermenge des ANSI-Standards.

Die Bezeichnung „ANSI-Standard“ oder „ANSI-C“ war Ende der achtziger und Anfang der neunziger Jahre eine gebräuchliche Charakterisierung für den vom ANSI-Komitee X3J11 entwickelten amerikanischen Standard für C. Daraus wurde 1990 der internationale Standard ISO/IEC 9899; seitdem spricht man stattdessen von „Standard-C“ oder „ISO-C“. Zu diesem Standard gab es zwei Korrekturen und eine Ergänzung, bis es 1999 zu einer zweiten Auflage kam [20], die die erste vollständig ersetzt hat. Zu dieser zweiten Auflage existieren auch schon mehrere Korrekturen [21, 23, 24]. Es ist nicht erkennbar, welcher dieser Stände in den Vorgaben gemeint ist.

Ferner wäre es wünschenswert, eine Aussage zur Nutzung eventueller Compiler-spezifischer Erweiterungen von C zu haben, wie beispielsweise die Schlüsselwörter `near` und `far`. (Es gibt ein paar Fälle, wo solche Erweiterungen sogar bereits im Standard vorgesehen sind.)

Ich nehme für die folgenden Betrachtungen den Stand von 2007 [20, 21, 23, 24] als Grundlage.

Für zukünftige Regeln beachte man, dass ein Regelwerk mit Bezug auf einen bestimmten Stand der Sprachdefinition auch Vorsorge für spätere Änderungen am Sprachstandard sowie für unzureichende Unterstützung des geltenden Standards in den eingesetzten Entwicklungswerkzeugen treffen sollte.

M2 (v): Die Ausnahmeregel ist unklar formuliert.

Die Ausnahmeregel zu Abweichungen von den Vorgaben besagt, dass Abweichungen ausschließlich dann zulässig sind, wenn sie zu einer „Qualitätsverbesserung der Software gegenüber den Regelvorgaben“ führen. Aber was ist eine „Qualitätsverbesserung“?

Die allgemein akzeptierte Definition von Qualität ist „Erfüllung der Anforderungen“. Wenn ein Hersteller also eine Anforderungsspezifikation erstellt, in die er die Forderung aufnimmt, dass die resultierende Software jede EBA-Regel mindestens einmal brechen muss, ist das dann eine legitime Abweichung von den Regeln oder nicht?

Ernster zu nehmen sind aber Fälle, bei denen die Anforderungen eines Programms keinen Bezug auf die EBA-Vorgaben nehmen, das Programm jedoch im Rahmen der Regeln die Anforderungen nicht erfüllen kann.[2] Ich nehme an, dass die Ausweichregel Abweichungen in diesem Fall legitimiert.

[2] Ein typischer Fall ist der Einsatz von Compiler-spezifischen Steueranweisungen für das Layout von Datenstrukturen, die nicht unter Kontrolle des Programmierers stehen sondern extern vorgegeben wurden.

3.1.2 Voraussetzungen für den Einsatz

M3 (P): Es fehlt eine Aussage, wie Fehler und insbesondere Warnungen des Compilers behandelt werden sollen.

Die Vorgaben sagen nur, dass sämtliche Fehler- und Warnmeldungen des Compilers aktiviert sein müssen.

Eine sinnvolle und auch übliche Forderung ist, dass alle Code-Stellen, die zu Warnungen führen, mit einem Kommentar versehen sein müssen, der erläutert, warum die Warnung harmlos ist. Falls es der Compiler unterstützt, kann man dann zusätzlich Steuerbefehle einsetzen, um die Warnung an dieser Stelle zu unterdrücken.

M4 (L): Es fehlt eine Aussage zum Linker.

Auch der Linker kann Fehler erkennen oder Warnungen ausgeben[3], insbesondere zu fehlenden oder mehrfach definierten Symbolen.

Ferner wäre es übersichtlicher gewesen, Anforderungen an den Entwicklungsprozess (wie Festschreiben der Compiler-Version) oder den Compiler logisch sauber von Anforderungen an den Code zu trennen.

3.1.3 Codierung der Grundstrukturen

M5 (S): Der Bezug zur Syntax von C fehlt.

Da das Vorgehen ein positives statt eines negativen sein soll (Vorgabe erlaubter statt Verbot untersagter Elemente), hätte man sagen sollen, welche der syntaktischen Elemente von C ersetzt werden sollen und man hätte für sie eine vollständige eigene Syntaxdefinition geben müssen.

Ich gehe im Folgenden davon aus, dass es in diesem Abschnitt hauptsächlich um *statement* [20, Abschnitt 6.8] und seine Teile geht. Allerdings wird hier auch auf *function-definition* [20, Abschnitt 6.9.1] Bezug genommen, dafür aber nicht auf *declaration* [20, Abschnitt 6.7], obwohl es in *external-declaration* [20, Abschnitt 6.9] parallel zu *function-definition* auftritt.

M6 (S): Es fehlt eine Definition für „Anweisung“ und der Begriff wird widersprüchlich verwendet.

Ein unvoreingenommener Leser würde erwarten, dass *statement* [20, Abschnitt 6.8] gemeint ist, aber diese Interpretation stimmt nicht mit der Verwendung des Begriffes überein:

- Vergleicht man die Einleitung des Abschnittes (3) mit dem Inhalt, so muss es sich um einen Begriff handeln, der mindestens *statement* umfasst.
- Nach dem Abschnitt „Einzelne Verarbeitungsschritte“ ist es wahrscheinlich, dass „Anweisung“ dort nur für *expression-statement* steht. (Siehe dazu die Diskussion in M8.)
- Nach dem Abschnitt „Zusammengefasste Verarbeitungsschritte“ sieht es so aus, als wäre *block-item* [20, Abschnitt 6.8.2] gemeint, also *statement* oder *declaration*. (Sollte tatsächlich nur *statement* gemeint sein, wäre die Deklaration lokaler Variablen verboten.)
- Die Darstellung in „Einfach-Verzweigung“ ist nur dann mit der Syntax von C konform, wenn man „Anweisung(en)“ dort als Synonym von *expression* betrachtet.
- Der Abschnitt „Mehrfachverzweigung“ legt nahe, dass „Anweisung“ dort *statement* bedeutet.

[3] Nicht ohne Grund spricht der Standard von „translation environment“ statt von „compiler“ [20, Abschnitt 5.1.1].

M7 (V): Es bleibt fast immer unklar, inwieweit die Vorgaben nur die Syntax oder auch das textuelle Layout betreffen.

Der einzige Fall dieses Abschnittes, der einigermaßen klar ist, ist die „Einzelne Anweisung, die in einer eigenen Zeile steht“ (siehe dazu aber die nächsten Punkte). Folglich wollen die Vorgaben auch Regeln zum textuellen Layout aufstellen.

Man kann argumentieren, dass dies der einzige Fall ist, wo etwas zum textuellen Layout gefordert wird, denn nur an dieser Stelle steht eine entsprechende Aussage im Text. Die Behauptung, dass das textuelle Layout auch der anderen Code-Auszüge bindend sei, lässt sich dagegen nicht logisch konsistent aufrechterhalten, da die Vorgaben und das Beispiel für `switch` dann nicht miteinander in Einklang zu bringen sind.

Aus persönlicher Erfahrung weiß ich jedoch, dass es Personen gibt, die das textuelle Layout weiterer Code-Auszüge dieses Abschnittes als verbindlich ansehen. Ich interpretiere dies als Beleg dafür, dass dieser Teil der Vorgaben zumindest nicht eindeutig verständlich formuliert ist. In erster Linie bleibt völlig unklar, welche textuellen Eigenschaften der angeführten Beispiele bindend sein sollten. Die Art und Größe der Einrückung? Die Zeilentrennung? Muss man beispielsweise wirklich den Zeilenumbruch in

```
if (j > 0)
    k++;
```

nach der Bedingung einfügen und dafür dann in einem `case`-Zweig sämtliche Anweisungen in derselben Zeile unterbringen wie das `case`-Label?

Speziell zum Layout von `if` beachte man auch, dass ein striktes Befolgen des Vorgaben-Beispiels bei kaskadierten `if`-Statements ein Layout der Form

```
if (...)
    ...
else
    if (...)
        ...
    else
        ...
```

statt

```
if (...)
    ...
else if (...)
    ...
else
    ...
```

erzwingt.[4] Ich halte den zweiten Stil dagegen für übersichtlicher; das ist auch die gängige Ansicht [4, 6, 7, 13, 16, 27].

M8 (V/P): Die Vorgabe zur Zeilentrennung von einzelnen Anweisungen ist nicht präzise genug formuliert und hat praxisferne Folgen.

Die Vorgabe lautet:

```
Anweisung
  /* Einzelne Anweisung, die in einer eigenen Zeile steht */
```

Nach den Regeln der deutschen Grammatik ist dies natürlich keine Aussage, da dem Hauptsatz das Verb fehlt. Es ist aber aus der Einleitung des Abschnittes (3) ergänzbar:

[4] Das ist kein theoretisches Beispiel: ich musste schon mal auf Forderung eines Validierers hin ein konkret bezeichnetes Statement, dessen Layout der zweiten Variante folgte, so ändern, dass es der ersten entsprach.

> Zur Codierung der in Teil 42 530 dargestellten Grundstrukturen sind für die Programmiersprache C nur die folgenden Anweisungen zugelassen: ...

Die Aussage der Vorgaben ist also, dass eine einzelne „Anweisung“, die in einer eigenen Zeile steht, zugelassen ist.

Falls mit „Anweisung“ ein *statement* gemeint sein sollte, gestattet diese Regel, ein komplettes *compound-statement* (beispielsweise einen Funktionsrumpf) in einer einzigen Zeile unterzubringen.[5] Das kommt mir unwahrscheinlich vor. Wahrscheinlicher ist dagegen, dass „Anweisung“ für *expression-statement* [20, Abschnitt 6.8.3] stehen soll.

Es wäre auch denkbar, dass es statt „einzelner Anweisung“ eigentlich „*statement*, das keine Teile enthalten kann, die wieder ein *statement* sind“ heißen sollte. Abgesehen von der Hinzunahme der *jump-statements* [20, Abschnitt 6.8.6], die später unter „Sonstige zugelassene Sprachelemente“ behandelt werden, ist dies aber mit *expression-statement* identisch.

Unabhängig davon, wie „Anweisung“ zu interpretieren ist, verbieten die Vorgaben, eine längere „Anweisung“ auf mehrere Zeilen zu verteilen: es gibt nirgendwo eine Regel, die dies gestattet, und die Einleitung des Abschnittes sagt ausdrücklich, dass *nur* die genannten Konstrukte zugelassen sind.

M9 (p): Es gibt Fälle, in denen mehr als ein *statement* pro Zeile sinnvoll ist.

Sofern es mir nicht durch Programmierregeln ausdrücklich verboten wird, setze ich die Konvention ein, dass nach dem Freigeben dynamischen Speichers der zugehörige Zeiger noch in derselben Zeile auf einen neuen Wert gesetzt wird:

```
free(p); p = NULL;
```

Diese beiden Anweisungen gehören logisch eng zueinander und bleiben auch beim Kopieren ganzer Zeilen immer zusammen; diese Konvention erleichtert das Aufdecken fehlerhafter Zugriffe auf freigegebenen Speicher beträchtlich.

M10 (s): Die Vorgabe für „Unterprogramm“ ist unklar formuliert.

Immerhin ist nach dem Kontext wahrscheinlich, dass mit „Funktionen“ Funktionsdefinitionen (*function-definition*) und nicht Deklarationen oder Aufrufe gemeint sind, aber das hätte man ruhig ausdrücklich sagen können.

M11 (S): Die Vorgabe für „Einfach-Verzweigung“ enthält Fehler.

Zur Erinnerung hier der Text aus [11]:

```
if (Ausdruck mit boole'schem Ergebnis (z. B. if (x EQ 0))
    Anweisung(en);

if (Ausdruck mit boole'schem Ergebnis)
    Anweisung(en);
else
    Anweisung(en);
```

Es geht also um die beiden `if`-Zweige von *selection-statement* [20, Abschnitt 6.8.4].

Die Fehler in dieser Vorgabe sind folgende:

- In der ersten Zeile fehlt eine schließende Klammer.
- Die *expression* in einem `if`-Statement ist in C nicht von booleschem sondern allgemeiner von skalarem Typ (sie wird mit `0` verglichen und nicht wie in C++ implizit zu `bool` konvertiert).
- Der in Abschnitt (4) geforderte Ersatz für `==` heißt „`eq`“ und nicht „`EQ`“.

[5] In C++ ist das sogar in Einzelfällen übersichtlicher, hier geht es aber um den allgemeinen Fall.

- Nach `if` bzw. `else` folgt in C jeweils ein einziges *statement*, nicht mehrere „Anweisungen“ abgeschlossen mit „`;`“.
- Die Angabe des Semikolons im untergeordneten Teil von `if` verbietet die Verwendung eines *compound-statement* vor `else`.[6]

M12 (s): Der `default`-Zweig eines `switch` muss eine Anweisung enthalten.

Die Formulierung

> Dabei muss der `default`-Zweig nicht zwingend Anweisungen enthalten.

steht im Widerspruch zur Syntax von C, wo nach `default:` immer ein *statement* folgen muss [20, Abschnitt 6.8.1]. Natürlich ist aber auch ein leeres *statement* („`;`“) erlaubt.

M13 (p): Es gibt Situationen, in denen das Weglassen eines `default`-Zweigs zu besser wartbarem Code führt.

Manche Compiler können bei `switch`-Statements mit einem `enum`-Typ eine Warnung ausgeben, wenn es für einen oder mehrere der Werte keinen Zweig gibt. Die Angabe eines `default`-Labels dürfte diese Diagnose unterbinden, so dass eventuelle Fehler (insbesondere bei Hinzufügen neuer Werte zum `enum`-Typ) erst zur Laufzeit erkannt werden können.

Allerdings zögere ich, das Weglassen von `default` bei `enum` generell zu fordern. Falls man den Compiler nicht so konfigurieren kann, dass ein fehlender `case` zum Abbruch der Kompilation führt, würde ein Weglassen von `default` auch bewirken, dass beim Ignorieren der Warnung dieser Fehler zur Laufzeit nicht mehr entdeckt wird.

Eine mögliche Alternative wäre die Forderung, in einem `default`-Zweig immer auf ungültige Werte zu prüfen. Dann darf der `default`-Zweig nie nur aus einem leeren Statement bestehen (siehe M12).

M14 (S): Die Vorgabe für `while` enthält Fehler.

Dies ist eine Teilmenge derselben Probleme, die schon bei `if` betrachtet wurden:

```
while (Ausdruck mit boole'schem Ergebnis)
      Anweisung(en);
```

M15 (V): Es ist nicht klar, ob und ggf. warum bei `do ... while` und `for` immer ein *compound-statement* gefordert wird.

Wenn die Angabe des *compound-statements* tatsächlich Absicht ist, warum dann nicht auch bei `while` oder `if`?

M16 (V): Es ist unklar, welche Varianten der `for`-Anweisung erlaubt sind.

Die Vorgabe sagt:

```
for(Startwert; Abbruchkriterium; Schrittdefinition)
{
     Anweisung(en)
}
```

In der Syntax von C sieht das aber etwas anders aus, insbesondere gestattet sie zwei verschiedene Arten der `for`-Anweisung [20, Abschnitt 6.8.5]:

[6] Bei `switch`, `do ... while` und `for` steht „Anweisung(en)“ dagegen ohne nachfolgendes Semikolon.

`for` (*expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$) *statement*
`for` (*declaration expression*$_{opt}$ `;` *expression*$_{opt}$) *statement*

Ist die zweite verboten? Und bedeutet die Verwendung der Begriffe „Startwert“, „Abbruchkriterium“ und „Schrittdefinition“ in irgendeiner Weise Einschränkungen an die *expressions*?

M17 (P): Das Verbot unendlicher Schleifen ist praxisfern.

Die Vorgabe sagt:

> Bei den Schleifen ist sicherzustellen, dass sie eine Endlichkeit besitzen.

Es gibt aber Schleifen (insbesondere bei der ereignisgesteuerten Programmierung, wie sie gerade für Echtzeitsysteme typisch ist), die ihrem Zweck nach nie oder nur in Ausnahmesituationen verlassen werden sollten.
Was dagegen Sinn gemacht hätte, wäre eine Regel, nach der *absichtlich* unendliche Schleifen auf eine syntaktisch einheitliche Art programmiert werden müssen, damit sie sofort erkennbar sind; in C wählt man dafür meist eine der folgenden Konventionen:

```
while (1) {          for (;;) {
    ...                  ...
}                    }
```

Dass man aber Schleifen, die enden sollen, auch so programmiert, dass sie es tun, ist meines Erachtens in dieser allgemeinen Form keine eigene Regel wert. Anders würde es nur aussehen, wenn besondere Maßnahmen zum Nachweis der Endlichkeit genannt würden.

M18 (v/p): Es fehlt eine Begründung, warum `continue` vermieden werden soll.

Es ist mir nicht erkennbar, warum der Einsatz von `continue` ein nennenswertes Qualitätsproblem darstellt. Eine weitergehende Diskussion zu diesem Punkt ist bei M146 zu finden.

M19 (v): Die Regel für die Verwendung von `return` ist unnötig umständlich formuliert und lässt den Zweck unklar.

Eine bessere und vor allem verständliche Regel wäre gewesen, dass jede Funktion nur einen einzigen regulären Ausgang haben darf (inhaltliche statt formaler Regel).

M20 (S): Die Funktion `exit()` ist kein „Sprachelement“ wie `break`, `continue` und `return` sondern Teil der Standard-Bibliothek.

M21 (P): Die Einschränkung der Verwendung von `exit()` auf Fehlerbehandlung ist praxisfern.

Unter dieser Vorgabe darf kein Programm erfolgreich enden.

M22 (V): Die syntaktischen Regeln des Abschnittes (3) lassen sich kompakter und verständlicher formulieren.

Bei einem Vergleich mit der Syntax von *statement* kann man feststellen, dass sich — abgesehen von den vorstehend angeführten offenen Punkten — fast alle syntaktischen Vorgaben dieses Abschnittes auf zwei Aussagen reduzieren lassen:

- Die Verwendung von `goto` ist nicht erlaubt.
- Ein einfaches *labeled-statement* (nur *identifier* `:` *statement*) ist nicht gestattet.

Das ist akzeptabel; allerdings gibt es auch für `goto` durchaus sinnvolle und die Qualität verbessernde Anwendungen.[7]

[7] Beispielsweise wird die Implementation von Zustandsautomaten dadurch erheblich vereinfacht, vor allem wenn man ihren Code durch ein Programm aus einer übergeordneten Beschreibung erzeugen möchte. Manche Programmierregeln [3, 7, 26] gestatten `goto` ferner für das Verlassen geschachtelter Schleifen, insbesondere im Fehlerfall.

3.1.4 Operatoren

M23 (s): Der Bezug zur Syntax von C fehlt.

Ich gehe im Folgenden davon aus, dass es in diesem Abschnitt um *expression* [20, Abschnitt 6.5.17] und seine Teile geht.

M24 (L): Der letzte Absatz dieses Abschnittes gehört logisch an den Anfang.

Der letzte Absatz („Zeichen, die in Abhängigkeit ihrer Stellung verschiedene Bedeutung haben, ...“) liefert die Begründung für die Forderung nach den Ersetzungen und die zugehörige Erläuterung. Die Spalte „geforderte Ersetzung“ in den Tabellen ist beim sequenziellen Lesen ohne diese Aussage nicht verständlich.

M25 (S): Die Vorgaben verbieten Funktionsaufrufe.

Die für Funktionsaufrufe nötigen Klammern `(...)` sind in den Vorgaben nur zur „Gliederung von Ausdrücken“ gestattet.

Allerdings steht diese Aussage unter „Suffix-Operatoren“, und die Klammern zur Gliederung von Ausdrücken sind keine Suffix-Operatoren während es die Klammern für Funktionsaufrufe sind. Möglicherweise ist hier also je die Hälfte zweier Aussagen entfallen.

M26 (S): Die geschweiften Klammern in Ausdrücken dienen nicht der „Markierung zusammengesetzter Anweisungen“.

Es handelt sich vielmehr um „compound literals“ [20, Abschnitt 6.5.2.5].

M27 (P): Die Anwendung von `sizeof` wird auf Datentypen beschränkt.

Die Vorgabe lautet:

`sizeof( )`	Größe des Datentyps

Die Syntax von C kennt zwei Varianten von `sizeof`:

`sizeof` *unary-expression*
`sizeof (` *type-name* `)`

Nach der Vorgabe ist also nur die zweite Variante gestattet.

Es ist aber in vielen Fällen sinnvoll und wird von manchen Programmierregeln sogar gefordert [16], `sizeof` nach Möglichkeit auf die betroffene Variable und nicht auf ihren Typ anzuwenden.[8] Der erste Weg ist robust gegen Änderungen des Typs der Variablen, der zweite nicht. Die Vorgabe erzwingt also die schlechter wartbare Lösung.

M28 (v): Es wird nicht klar, warum `sizeof` in der *constant-expression* eines `#if...` ausdrücklich verboten wird.

Da der Präprozessor alle nach der Makrosubstitution verbliebenen Bezeichner durch Null ersetzt, sollten zudem alle im normalen Code gestatteten Arten der Verwendung von `sizeof` an dieser Stelle immer zu einem vom Präprozessor erkannten Syntaxfehler führen.

[8] Das ist nicht das gleiche, wie jede *unary-expression* zuzulassen; unerfahrenen Entwicklern wird unter Umständen nicht bewusst sein, dass hier nicht der Wert des Ausdrucks sondern nur sein Typ von Bedeutung ist. Es gibt daher auch mindestens ein Regelwerk [26], dass an dieser Stelle Ausdrücke mit Seiteneffekten untersagt.

M29 (s): Die Header-Datei `<iso646.h>` ist Teil des C-Standards.

Dies gehört zu den 1999 eingeführten Erweiterungen. Damit ist der Verweis auf den C++-Standard überflüssig.

M30 (V): Die Regeln für Postfix-Inkrement und -Dekrement werden nicht begründet.

Die Formulierung der Vorgaben lautet:

> Die Postinkrement- und Postdekrement-Operatoren sind nur zugelassen, wenn sie in einem eigenen Ausdruck stehen, bzw. in `for`-Anweisungen.

Abgesehen von den Unklarheiten der Formulierung (was ist ein „eigener Ausdruck“ und warum ist eine *expression* in einer `for`-Anweisung keiner?) ist auch nicht erkennbar, welcher Zweck mit dieser Regel erreicht werden soll. Mir fallen zwei mögliche Erklärungen ein:

- Man war der Meinung, dass sich ein Programmierer nicht merken kann, ob der Wert dieser Operatoren der Wert des Arguments vor oder nach der Ausführung ist.[9] Das hätte man dann aber klarer mit einer Regel behandelt, die verbietet, diesen Wert zu nutzen.
- Es geht um das Problem fehlender Sequenzpunkte bei Ausdrücken, in denen eine Variable mehrfach auftritt und an mindestens einer Stelle modifiziert wird. Das ist aber ein allgemeineres Problem, das durch diese Regel nur teilweise verhindert wird (siehe Abschnitt (13)) und für das die Regel eine überzogen einschränkende Vorgabe macht.

M31 (V): Die Regel für `mod` ist unverständlich formuliert, wird nicht begründet und ist wahrscheinlich überflüssig.

Die Vorgabe sagt:

> Der Operator mod (Divisionsrest) darf nur auf positive `int`-Werte innerhalb des Zahlenbereichs angewandt werden.

Diese Aussage hat folgende Probleme:

- Was ist der Zweck dieser Einschränkung?
- Welche der Argumente von `mod` sind gemeint: Dividend, Divisor oder beide?
- Warum ist die Anwendung bei `short` oder `long` verboten?

Ich vermute, die Aussage stammt daher, dass sich das Ergebnis der ganzzahligen Division und der Restbildung aufgrund einer Spezifikationslücke bei frühen Implementationen von C nur dann als portabel erwies, wenn der Dividend nicht negativ und der Divisor positiv war [25]. Das träfe dann aber genauso auf das Ergebnis der Division zu und ist zudem überholt: der Standard schreibt die Ergebnisse eindeutig fest, solange der Divisor nicht Null ist [20, Abschnitt 6.5.5].

M32 (p): Der Nutzen der geforderten Makro-Ersetzungen ist in den meisten Fällen fraglich.

Die am Schluss von (4) angegebene Begründung ist meines Erachtens nur eine Behauptung und nicht in der Praxis nachgewiesen. Bloß im Fall des Gleichheitsoperators `==` muss man davon ausgehen, dass es sich um ein bekannt fehleranfälliges Konstrukt von C handelt [25]. Die Ursache dafür ist aber offensichtlich nicht, wie vom EBA behauptet, die positionsabhängige Bedeutung des Symbols sondern die Tatsache, dass man in diesem Punkt bei C von der mathematisch gebräuchlichen Notation abgewichen ist.

In meinen Augen wird der Code durch die geforderten Ersetzungen im Allgemeinen sogar schlechter lesbar.

[9] Das würde auch erklären, warum die Präfix-Inkrement- und -Dekrement-Operatoren nicht erwähnt werden, denn dann benötigt man nur eine der beiden Varianten.

M33 (v/p): Es ist nicht erkennbar, warum Präfix-Inkrement und -Dekrement nicht erwähnt sind.

Allerdings kann man unter Berücksichtigung der zusätzlich verhängten Einschränkungen sicherlich mit dieser Regel leben, solange man nur in C programmiert; es wird lästiger, wenn man mit C++ arbeitet.

M34 (L): Die Präfix-Operatoren `&` und `*` werden nicht erwähnt.

Ich vermute, es handelt sich dabei um ein Versehen. Das wird zumindest teilweise dadurch gestützt, dass in Abschnitt (17) die Verwendung des Adressoperators `&` eingeschränkt wird; folglich ist er im Prinzip zugelassen. Warum der Dereferenzierungsoperator `*` nicht erwähnt wird (insbesondere, da `->` gestattet ist), ist nicht erklärlich.

M35 (L): Casts werden nicht erwähnt, obwohl sie im Prinzip zugelassen sind.

Der Abschnitt (18) gestattet Casts unter bestimmten Bedingungen.

M36 (P): Der konditionale Operator `?:` wird nicht erwähnt.

Es ist nicht erkennbar, warum dieser Operator generell verboten sein soll[10], und es gibt Situationen, in denen er erheblich übersichtlicheren Code liefert. Ein Beispiel dafür ist die Initialisierung lokaler Variablen einer Funktion in Abhängigkeit von ihren Argumenten:

```
int f(int a, int b)
{
    int Minimum =  a < b ? a : b;
    ...
}
```

Ohne den konditionalen Operator muss man das so schreiben:

```
int f(int a, int b)
{
    int Minimum;
    if (a < b)
        Minimum = a;
    else
        Minimum = b;
    ...
}
```

Ähnliches gilt in Zusammenhang mit `return`.

Wenn man das Verbot auf C++ ausdehnt und statt eines `int` ein Objekt mit Konstruktor betrachtet, sieht man, dass die Vorgabe dort auch noch unnötigen Laufzeit-Overhead erzwingt. Möchte man das Objekt zudem als `const` deklarieren, muss man ferner eine überflüssige Hilfsvariable einführen.

M37 (p): Der Komma-Operator wird nicht erwähnt.

Die nützlichen Anwendungsfälle dieses Operators sind zugegebenermaßen nicht besonders zahlreich, aber er ist sinnvoll in `for` einsetzbar und man braucht ihn in seltenen Fällen zwingend zum Setzen eines Sequenzpunktes.

[10] Eine (vielleicht nicht ernsthaft gemeinte) Begründung, die mir mal gegeben wurde, war, dass man sich nicht merken könne, welcher der beiden Zweige unter welchen Umständen ausgeführt wird. Wer damit ein Problem hat, sollte dasselbe Problem aber auch mit `if ... else ...` haben; auch die Reihenfolge ist dort die gleiche.

3.1.5 Präprozessorbefehle

M38 (p): Die explizite Vorgabe zweier bestimmter Formen von `#include` schränkt die Syntax dieser Anweisung unmotiviert ein.

Beispielsweise ist es damit nicht gestattet, als Argument für `#include` einen Makro zu verwenden. Das ist aber in manchen (zugegebenermaßen seltenen) Situationen übersichtlicher.

M39 (P): Die `#undef`-Direktive ist nicht erlaubt.

Da Makro-Definitionen in allen Scopes wirken, ist die Verwendung von `#undef` manchmal unumgänglich, um Namenskonflikte zu vermeiden.

Es macht auch Sinn, diese Direktive bei Makros einzusetzen, die nur in einem begrenzten Code-Stück einer größeren Quelldatei gebraucht werden. Dies erhöht die Wartbarkeit des auf dieses Stück folgenden Codes. (Ein Beispiel findet man in M74.)

M40 (P): Die `#error`-Direktive ist nicht erlaubt.

Die `#error`-Direktive bietet eine verlässliche Möglichkeit, die Kompilation bei Eintritt gewisser Bedingungen mit einer brauchbaren Fehlermeldung abzubrechen. Dies hilft vor allem bei über Präprozessorsymbolen konfigurierbaren Quelldateien, aber auch bei der Portierung.

M41 (ü): Die Aussage zu `#if`/`#elif` und `defined` ist unnötig.

Die Vorgabe sagt:

> Die Präprozessorbefehle `#if` und `#elif` sind auch in Kombination mit dem Operator `define` erlaubt.

Diese Aussage ist unnötig (und es müsste „`defined`“ statt „`define`“ heißen), da keine Einschränkungen an die *constant-expression* von `#if` oder `#elif` verhängt wurden.

M42 (P): Die Vorgaben zu `#pragma` sind unnötig restriktiv und ignorieren Portabilitätsprobleme.

`#pragma`-Direktiven wurden ursprünglich eingeführt, um Compiler-spezifische Eingriffsmöglichkeiten zu haben. Ihr zentrales Problem ist, dass dieselbe Direktive von verschiedenen Compilern unterschiedlich interpretiert werden kann, d. h. ihr fehlt die Portabilität (Gleiches gilt natürlich auch für `_Pragma()`).

Inzwischen gibt es aber sogar vom Standard definierte `#pragma`-Direktiven [20, Abschnitt 6.10.6] und es existiert eine `#pragma`-Direktive, die viele Compiler verstehen und auf die gleiche Weise behandeln:

```
#pragma ident "..."
```

Diese Anweisung hat nicht einmal Einfluss auf den erzeugten ausführbaren Code. Es ist daher nicht erkennbar, warum solche Direktiven nicht gestattet sein sollen.

Ähnliches gilt meines Erachtens aber auch für Compiler-spezifische `#pragma`-Direktiven. Wenn ein Compiler die Möglichkeit bietet, auf diesem Weg Eigenschaften des erzeugten Codes zu beeinflussen, die sonst der Willkür des Compilers unterliegen, ist nicht verständlich, warum das generell verboten sein soll. Es macht zwar Sinn, solche `#pragma`-Direktiven zunächst zu untersagen (schon aus formalen Gründen, da sie den Sprachumfang ändern), man sollte dieses Verbot jedoch nur unter dem Vorbehalt verhängen, dass es durch Compiler-spezifische Ergänzungen der Regeln aufgehoben werden darf.

Viel wichtiger ist aber noch, dass man Compiler-spezifische `#pragma`-Direktiven bei Einsatz mehrerer Compiler *unbedingt* mit einem geeigneten Compiler-spezifischen `#ifdef` kapseln sollte, damit sichergestellt ist, dass eine Direktive nur von Compilern gesehen wird, die sie wie vom Programmierer beabsichtigt interpretieren. Das ist auch gleich ein gutes Beispiel für den Einsatz von `#error`:

```
/* Kompaktes Speicherlayout für S sicherstellen */
#ifdef CADUL
#pragma align (S = 1)
#elif anderer_Compiler
...
#elif !defined(__GNUC__)
#error "Nicht vorgesehener Compiler."
#endif

typedef struct {
    ...
}
#ifdef __GNUC__
    __attribute__((packed))
#endif
    S;
```

M43 (V): Die Aussage zu Namen von Prototypen ist unverständlich.

Die Vorgabe sagt:

> Es ist sicherzustellen, dass die Namen der Prototypen in der Headerdateien [sic] in einem Modul nicht auch für non-static Funktionen verwendet werden (keine automatische Überprüfung möglich).

Unter „Prototypen“ versteht man in C normalerweise Funktionsprototypen (siehe auch den in [11] folgenden Absatz), d. h. Funktions-Deklarationen, die noch keine Definitionen sind und die Typen der Funktionsargumente nennen. Eine solche Deklaration enthält mehrere Bezeichner, darunter den der Funktion selber; letzterer könnte mit „Name des Prototypen“ gemeint sein. Allerdings werden solche Deklarationen meist für Funktionen mit external linkage eingesetzt, d. h. für nicht als `static` deklarierte Funktionen. Anders ausgedrückt: fast jede Funktions-Deklaration würde mit dieser Interpretation einen Regelkonflikt mit sich selber erzeugen.

Eine andere Möglichkeit wäre, dass die Bezeichner der Funktionsparameter gemeint sind. Da sie aber in einem eigenen Scope definiert sind [20, Abschnitt 6.2.1], können sie keinen Konflikt mit anderen Namen produzieren.

Für den Präprozessor, um den es in diesem Abschnitt geht, ist das alles aber zudem ohne Belang, da er diese Struktur nicht analysiert. Die Aussage steht daher (ebenso wie die zwei folgenden) im falschen Abschnitt.

M44 (V/p): Funktionsprototypen in Implementationsdateien werden verboten.

Die Vorgabe sagt:

> Funktionsprototypen sollen ausschließlich in Headerdateien verwendet werden.

Es ist nicht erkennbar, welcher Zweck mit dieser Regel verfolgt wird, und sie verbietet die manchmal sinnvolle Vorwärtsdeklaration innerhalb einer Implementationsdatei.

Sinnvoll wäre dagegen die Forderung, dass es keine lokalen Deklarationen für extern definierte Funktionen geben darf. Man betrachte dazu folgendes Beispiel:

```
extern int f(void);
...
{
    ...
    j = f();
    ...
}
```

Dies soll dann für den Fall, dass `f()` in einer anderen Quelldatei definiert ist, verboten sein. Das ist auch sinnvoll, da beim obigen Code das Risiko besteht, dass sich die Signatur von `f()` ändert, ohne dass die lokale Deklaration nachgezogen wird. Die in der Vorgabe angeführte Regel ist unzureichend, dieses Problem zu verhindern, denn man würde sie immer noch erfüllen, wenn man die Deklaration von `f()` jetzt in eine eigene lokale Header-Datei auslagern würde. Stattdessen wird eine Regel benötigt, die verlangt, dass bei der Kompilation einer Funktionsdefinition auch alle zugehörigen Funktionsprototypen in dieser *translation-unit* enthalten sind, so dass der Compiler die Konsistenz aller Deklarationen dieser Funktion überprüfen kann.

3.1.6 Datentypen

M45 (s): Der Bezug zur Syntax von C fehlt.

Ich gehe im Folgenden davon aus, dass es in diesem Abschnitt vor allem um *declaration* [20, Abschnitt 6.7] und seine Teile geht. Allerdings wird auch auf Ausdrücke Bezug genommen.

M46 (P): Es ist nicht erkennbar, warum so viele der Grundtypen verboten sind.

Die Vorgabe sagt:

> Zur Strukturierung von Daten sind nur die folgenden Datentypen zugelassen: `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `enum`, `void`.

(Selbstverständlich ist `void` nicht zur Strukturierung von Daten tauglich.) Diese Regel verbietet die Nutzung folgender Grundtypen („basic types“):

- `_Bool` (und damit `bool`)
- `signed char`
- `long long int`
- `unsigned long long int`
- eventuell vom Compiler unterstützte „extended integer types“
- `float`, `double` und `long double`
- `float _Complex`, `double _Complex`, `long double _Complex`, `float _Imaginary`, `double _Imaginary` und `long double _Imaginary`.

Zwar kann man im Bereich der Eisenbahnsignaltechnik wohl fast immer auf die Gleitkommatypen verzichten, so dass dieser Teil der Einschränkungen hinnehmbar ist, es wäre aber sicher hilfreich zu wissen, warum dieses Verbot verhängt wurde. Übliche Programmierregeln betrachten beim Umgang mit Gleitkommatypen nur die Vergleiche als problematisch und schränken sie daher ein. Dazu könnte man noch Forderungen zum Umgang mit Floating Point Exceptions hinzufügen.

Das Verbot der oben aufgezählten integralen Typen ist mir dagegen völlig unverständlich. Insbesondere die Nutzung von `_Bool` (besser noch `bool`) sollte man auf jeden Fall zulassen, damit man statt (möglicherweise auch noch mehrerer) anwenderdefinierter Typen für diesen Zweck einen einheitlichen und in der Sprache verankerten Typ verwenden kann.

M47 (P): Die Mischung von `signed` und `unsigned` kann in jeder *expression* problematisch sein, nicht nur bei arithmetischen Operationen.

Insbesondere bei Vergleichsoperationen werden ebenfalls implizite Typ-Konversionen durchgeführt, die den Wert verändern können:

```
int a;
unsigned int b;

a = -1;
b = 100;

printf("Es gilt: %d ", a);
```

```
if (a < b) putchar('<');
else printf(">=");
printf(" %u.\n", b);
```

Die Ausgabe dieses Programmstücks lautet bei einem Standard-konformen Compiler:

```
Es gilt: -1 >= 100.
```

M48 (V): Der Begriff „typkorrekte Konvertierung“ wird nicht erläutert.

Die Vorgabe sagt:

> Die Mischung von `signed`- und `unsigned`-Variablen in arithmetischen Operationen ist zu vermeiden. Soweit erforderlich ist eine explizite, typkorrekte Konvertierung vorzunehmen.

Die Sprache C kennt nur die Begriffe „implicit conversion“, „explicit conversion“, „integer promotions“ und „usual arithmetic conversions“ [20, Abschnitt 6.3]; ein Begriff wie „typkorrekte Konvertierung“ wird dagegen nicht erwähnt.

Im Abschnitt (18) heißt es:

> Explizite Typumwandlungen (**cast**) sind außer bei einer typkorrekten Konvertierung nicht erlaubt.
>
> Die explizite Typumwandlung ist auch dann zulässig, wenn eine Bereichsüberprüfung durch das Programm erfolgt.

Ich interpretiere das „auch“ im zweiten Satz so, dass es sich bei dieser Aussage trotz der Abtrennung in einen eigenen Absatz um eine Einschränkung des Verbots im ersten Satz handelt. Folglich ist gemeint: eine explizite Konversion (also mit Cast) ist entweder „typkorrekt“ oder nicht; im zweiten Fall ist sie verboten, außer wenn das Programm eine geeignete Bereichsüberprüfung durchführt.

Wendet man allerdings diese Interpretation auf das einführende Zitat an, so folgt, dass nicht „typkorrekte“ Konversionen zur Behebung von `signed`/`unsigned`-Konflikten auch dann verboten sind, wenn sie mit Bereichsüberprüfung abgesichert werden: nur eine „explizite, typkorrekte Konvertierung“ wurde erlaubt.

Aber was wäre nun eine sinnvolle Bedeutung von „typkorrekt“? Im Allgemeinen gibt es bei Konversionen dann und nur dann Probleme, wenn der Wert der Quelle außerhalb des Wertebereichs des Zieltyps liegt. Eine mögliche Interpretation wäre daher, dass mit einer „typkorrekten Konvertierung“ eine Konversion gemeint ist, die bereits anhand der beteiligten Typen als werterhaltend erkannt werden kann und daher keine Bereichsüberprüfung benötigt.

Wenn das tatsächlich die beabsichtigte Bedeutung von „typkorrekt“ ist, so ist diese Bezeichnung irreführend gewählt worden, denn das zentrale Problem steckt in den Werten und nicht in den Typen. Eine Forderung nach „werterhaltender Konvertierung“ wäre dagegen nicht nur sofort verständlich gewesen sondern hätte außerdem noch die durch Bereichsüberprüfung abgesicherten Fälle mit erfasst.

M49 (V/l): Die Forderung nach Testläufen auf dem Zielsystem ist nicht hinreichend präzise und passt logisch auch nicht in diesen Abschnitt.

Die Formulierung lautet:

> Es müssen Testläufe auf dem Zielsystem durchgeführt werden, um Fehler auszuschließen, die durch die rechnerspezifische Integer-Darstellung und Busbreite entstehen können.

Diese Aussage hat folgende Unklarheiten:

- Welcher Bus ist gemeint?
- Welche Fehler sind gemeint?
- Warum steht eine Forderung zur Überprüfung der Korrektheit von Maschinenbefehlen des Zielsystems in einer Programmierregel für C? Die Korrektheit der Übersetzung von C in Maschinenbefehle ist bereits mit den Anforderungen in (2) abgedeckt.

M50 (S/L): Die Beschreibung der abgeleiteten Typen ignoriert die Begriffe von C und ist zudem lückenhaft.

Die Typen von C lassen sich wie folgt gliedern [20, Abschnitt 6.2.5]:

- `void`
- basic types
- derived types:
  - aggregate types:
    - array type
    - structure type
  - union type
  - function type
  - pointer type

Der Begriff „komplexe Struktur“ wie in der Vorgabe wird in C nicht verwendet; das gäbe wegen `_Complex` auch beträchtliche Verwirrung.

Die Vorgabe lässt von den „derived types“ nur „structure type“ und „union type“ zu (Bitfelder sind Strukturen), aber offensichtlich sollen auch „array type“ (siehe Beispiel zur Initialisierung in diesem Abschnitt) und „pointer type“ zugelassen sein (Abschnitt (17)). Letzteres gilt unter Auflagen auch für Zeiger auf Funktionen, so dass auch „function type“ im Prinzip erlaubt sein muss.

M51 (s): `typedef` erzeugt keine „komplexen Strukturen“.

`typedef` führt nur einen neuen Namen für einen bereits existierenden Typ ein. Letzteres braucht auch keine Struktur zu sein.

M52 (V/P): Die Forderung zur Initialisierung mit geschweiften Klammern ist missverständlich formuliert oder führt zu unübersichtlichem Code.

Die Vorlage sagt:

> Die Initialisierung soll zur Übersichtlichkeit mit geschweiften Klammern erfolgen.

Die Syntax von C legt fest [20, Abschnitt 6.7.8], dass es für *initializer* drei Varianten gibt: eine *assignment-expression* oder eine in geschweifte Klammern gesetzte *initializer-list*, optional mit hängendem Komma. Nach dem Wortlaut der Vorgabe würde man also erwarten, dass die erste dieser drei Möglichkeiten verboten werden soll.

Angewandt auf Skalare erfordert das im Folgenden die linke statt der rechten Variante:

```
int j = {0};            int j = 0;
```

Ich halte dagegen die rechte Fassung für übersichtlicher als die linke. Dies liegt vor allem daran, dass die geschweiften Klammern in C eine Strukturierungsebene bedeuten: etwas in geschweifte Klammern zu setzen bedeutet fast immer, dass es eine Ebene tiefer liegt. Der Leser wird daher durch die Klammern eher verwirrt als erleuchtet, wie man am folgenden EBA-konformen Beispiel besonders deutlich sieht:

```
int j = {0};
int a[] = {0};
```

Die rechten Seiten suggerieren, dass die linken Seiten vom gleichen Typ sind; das ist jedoch falsch. Es scheint also sogar eher sinnvoll zu fordern, dass die geschweiften Klammern bei der Initialisierung entfallen müssen, falls das möglich ist. Eine solche Regel wird beispielsweise in [16, Abschnitt 8.6] für den Fall verhängt, dass das betroffene Objekt nur einen einzigen Wert zur Initialisierung benötigt.

Nach dem Kontext der Vorgabe ist es auch möglich, dass die Aussage nur auf „komplexe Strukturen“ (genannt sind `struct` und `union`) angewendet werden soll. Allerdings sind die Klammern

in diesen Fällen meist notwendig und nicht optional. Eine von zwei Ausnahmen ist der Fall, dass man bereits ein Objekt des Ziel-Typs hat und es zur Initialisierung verwendet, aber das ist meines Erachtens die gleiche logische Situation wie bei den Skalaren, d. h. man sollte die Klammern dann weglassen. Die zweite Ausnahme betrifft im Objekt enthaltene Aggregate oder `unions`: für die Initialisierung dieser untergeordneten Komponenten darf man in C die inneren geschweiften Klammern auch weglassen. Initialisierungen dieser Art sind nicht einmal robust gegen Hinzufügen eines weiteren Feldes am Ende eines Subaggregats und sind somit in der Tat eine Fehlerquelle. Es ist denkbar, dass die Vorgabe nur diesen Fall geschachtelter Initialisierungen gemeint hat.

Schließlich sei auch noch darauf hingewiesen, dass man mit der 1999 erfolgten Einführung von *designation* in *initializer-list* die Möglichkeit sehr übersichtlicher und robuster Initialisierungen bekommen hat.

M53 (s): Im angegebenen Initialisierungsbeispiel fehlen sowohl das „=“ als auch das abschließende Semikolon.

3.1.7 Speicherklassen

M54 (s): `volatile` ist keine Speicherklasse.

Ich interpretiere den Abschnitt so, dass es hier mit Ausnahme von `typedef` um *storage-class-specifier* und die *type-qualifier* gehen soll.

M55 (p): `register` sollte nicht generell verboten sein.

Nahezu jedes Regelwerk zu C empfiehlt, `register` normalerweise nicht zu verwenden, da Compiler in dieser Hinsicht schon seit längerem bessere Entscheidungen treffen als der durchschnittliche Programmierer. Trotzdem kann es seltene Situationen geben, in denen ein geschickt platziertes `register` zu schnellerem Code führt. (Allerdings muss ich zugeben, dass ich selber `register` noch nie benötigt habe.)

Zudem hat die An- oder Abwesenheit von `register` keinen Einfluss auf das Resultat der Ausführung. Damit kann ein Verbot auch keine Fehler verhindern.

M56 (P): Es fehlen Aussagen zu `const` und `restrict`.

Man beachte, dass das Beispiel in (10) `const` nur für Konstanten im File Scope gestattet; so etwas wie ein Funktionsargument vom Typ `const char *` wird nirgendwo erwähnt und ist damit auch nicht erlaubt.

3.1.8 Kommentare

M57 (ü): Die Forderung, Kommentare „übersichtlich zu gestalten“, ist zu subjektiv, um nützlich zu sein.

Übliche Programmierregeln stellen stattdessen inhaltliche Anforderungen an die Kommentare, beispielsweise nicht in den Kommentaren das zu wiederholen, was bereits aus dem Code ablesbar ist.

M58 (s): //-Kommentare sind Teil des C-Standards.

Dies gehört zu den 1999 eingeführten Erweiterungen.

M59 (v): Die Forderung, Kommentare nicht zu verschachteln, sollte umformuliert werden.

Erstens können Kommentare nach der lexikalischen Struktur von C gar nicht verschachtelt werden und zweitens sollte man die wohlbekannte Lösung (`#if 0`) für das eigentlich dahinterstehende Problem (Ausblenden von Code-Teilen, die Kommentare enthalten) ruhig erwähnen.

3.1.9 Bezeichner

M60 (ü/V/P): Die Regel zur Bildung von Bezeichnern ist zum Teil überflüssig, zum Teil unklar, und gestattet ausdrücklich nicht portablen Code.

Die Vorgabe sagt:

> Zur Bildung von Bezeichnern sind alle Buchstaben des Alphabets in Groß- und Kleinschreibung sowie die Ziffern von 0, 1..9, das Zeichen '$' und '_' zugelassen ('_' darf nicht am Anfang stehen).

Nach der lexikalischen Struktur von C kann man einen *identifier* dagegen aus einem *identifier-nondigit* und einer möglicherweisen leeren Folge von *digit* und *identifier-nondigit* bilden. Zu *identifier-nondigit* gehören die 52 Buchstaben des lateinischen Alphabets (groß und klein), der Unterstrich, *universal-character-name* sowie implementationsdefinierte Zeichen. In diesem Namensraum sind die Schlüsselwörter nicht als Bezeichner verwendbar; darüber hinaus reserviert die Standard-Bibliothek einen Teil des Namensraums für sich [20, Abschnitt 7.1.3], darunter solche Bezeichner, die mit einem Unterstrich beginnen.

Die Aussage der Vorgabe hat also folgende Probleme:

- Es bleibt offen, was genau mit „Alphabet" gemeint ist.
- „`$`" kann nicht portabel in einen *identifier* aufgenommen werden.
- Die Vorgabe wiederholt Aussagen, die bereits von C verhängt werden.
- Die Vorgabe lässt Bezeichner zu, die in C verboten sind (solche, die mit Ziffern beginnen).

Interpretiert man „Alphabet" als Vereinigung der 52 Buchstaben des lateinischen Alphabets mit *universal-character-name* und streicht man alle Anteile aus der Vorgabe, die überflüssig sind, so bleibt also als Rest nur folgende Aussage:

> Man darf „`$`" in *identifier* verwenden, falls der Compiler das akzeptiert.

Der resultierende Code wäre nicht portabel.

M61 (V/p): Die Regeln zur Wahl von Bezeichnern sind unklar formuliert und zum Teil unnötig einschränkend.

Die Vorgabe sagt:

> Namen und Bezeichner müssen so gewählt werden, dass sie sich in den ersten 31 Stellen signifikant unterscheiden. Sie sollten jedoch nach Möglichkeit kurz, unterscheidbar und prägnant sein und müssen projektweit beibehalten werden.
>
> Gleiche Variablen dürfen nicht unterschiedlichen Zwecken dienen.

Diese Aussagen haben folgende Probleme:

- Ein „Name" ist ein „Bezeichner" [20, Abschnitt 6.4.2.1].
- Der C-Standard garantiert eine Unterscheidbarkeit in den ersten 31 Stellen für „external names", bei „internal names" muss eine Standard-konforme Umgebung dagegen sogar mindestens 63 Stellen unterstützen [20, Abschnitt 5.2.4.1]. Es ist damit zu rechnen, dass die Einschränkung für external names auf 255 Stellen erhöht werden wird [20, Abschnitt 6.11.3].

  Zudem handelt es sich bei diesem Punkt um ein reines Portabilitätsproblem; es ist nicht erkennbar, warum ein Hersteller, der einen Compiler verwendet, der mehr als diese Grenzwerte unterstützt, diese Fähigkeit nicht ausnutzen darf.[11]
- Was soll es bedeuten, dass ein Name „projektweit beibehalten" werden muss?
- Normalerweise nennt man zwei Variablen „gleich", wenn sie den gleichen Wert haben. Das ist hoffentlich nicht gemeint; vermutlich soll es dagegen um Variablen mit gleichem Namen gehen. Dann fehlt der Vorgabe aber eine Einschränkung auf den betrachteten Namensbereich (beispielsweise die Quelldatei; eine globale Gültigkeit wäre offensichtlich Unsinn). Außerdem ist unklar, wann zwei Zwecke „unterschiedlich" sind.

[11] Bei der NASA beispielsweise sieht man das erheblich lockerer: „Assign names that are unique (with respect to the number of unique characters permitted on your system)." [7, Abschnitt 2.4].

3.1.10 Konstanten

M62 (P): Das Verbot von Literalen ist nicht erklärlich.

Die Vorgabe sagt:

> Im Code dürfen keine numerischen Werte, String- oder Char-Werte direkt vorkommen, sondern entsprechend definierte Konstanten (Ausnahme 0 und 1).

Diese Aussage hat folgende Probleme:

- „Code“ ist missverständlich. Vermutlich ist *statement* aber nicht *declaration* gemeint.
- Am Beispiel von `char`-Literalen sieht man, dass Literale erheblich besser verständlich sein können als Bezeichner für Konstanten mit entsprechenden Werten:

  ```
  ...                    const char Buchstabe_A = 'A';
  ...                    ...
  char z = 'A';          char z = Buchstabe_A;
  ```

  Eine Ausnahme ist aber der Fall, dass ein Bezeichner die Rolle statt des Wertes ausdrücken kann (s. u.).
  Ähnliches gilt auch für numerische Werte und Strings.
- Es ist unklar, welcher Zweck mit dieser Vorgabe verfolgt wird. Insbesondere beachte man, dass es einem Compiler frei steht, den Wert eines `const`-Bezeichners an der Verwendungsstelle einzusetzen.

Sinnvoll wäre stattdessen eine Regel gewesen, die die Einführung von Namen für Werte fordert, die eine eigenständige Rolle bzw. eine über den Wert hinausgehende Bedeutung haben (z. B. „Pfadseparator“ für „\“ oder „/“).

M63 (V): Es ist unklar, was eine „anwendungsspezifische Konstante“ sein soll.

M64 (ü/s): Die Forderung nach Typisierung von `#define`-Konstanten ist überflüssig.

Die Vorgabe sagt:

> Nicht anwendungsspezifische Konstanten können mittels `#define` definiert werden. Es muss dann aber eine Typisierung der definierten Konstante erfolgen.

Jede literale Konstante hat in C einen Typ [20, Abschnitt 6.4.4], so dass die Forderung in dieser Form (ich will nicht ausschließen, dass eigentlich etwas anderes gemeint war) immer erfüllt ist.

3.1.11 Variablen

M65 (s): Variablen werden in C nicht in einem *statement* sondern in einer *declaration* definiert.

3.1.12 Namen

M66 (ü/s): Die Forderung, dass alle Namen vor Verwendung deklariert werden müssen, ist überflüssig.

Die Vorgabe sagt:

> Alle Namen müssen explizit deklariert werden, bevor sie verwendet werden. Diese Regel gilt auch für Funktions- und Variablen-Namen und Feldnamen.

Historisch bestand die einzige Möglichkeit, in C einen Namen ohne Deklaration zu verwenden, bei Funktionen. Nach der aktuellen Fassung des C-Standards ist selbst das nicht mehr möglich; jede Verwendung eines nicht deklarierten Bezeichners verstößt gegen die Sprachdefinition [20, Abschnitt 6.5.1].

M67 (ü): Die Forderungen zur Initialisierung sind überflüssig.

Die Vorgabe sagt:

> Alle Variablen und Felder sind vor der erstmaligen Benutzung zu initialisieren.
> Bei Schleifenzählern von `for`-Schleifen genügt die Startwertzuweisung im Schleifenkopf als Initialisierung.

Da auch Felder Variablen sind, ist die gesamte erste Aussage eine unnötige Wiederholung der bereits im Abschnitt (11) angeführten Regel. Die Bemerkung zur `for`-Schleife sollte jedem Programmierer auch ohne Hilfestellung klar sein.

M68 (S): Es gibt in C keine „untergeordneten Funktionen“.

Möglicherweise sind untergeordnete Scopes gemeint.

M69 (s): Schlüsselwörter können nicht als Namen verwendet werden.

Das gilt nicht nur für C sondern genauso für jede andere Programmiersprache, die Schlüsselwörter definiert, und ist eine Kerneigenschaft dessen, was „Schlüsselwort“ bedeutet. Eine solche Regel macht also nur für solche Bezeichner Sinn, die in C++ aber nicht in C Schlüsselwörter sind.

3.1.13 Zuweisungen

M70 (V): Es ist unklar, was mit „Mehrfachzuweisungen“ gemeint ist.

Die Vorgabe sagt:

> In einer Anweisung darf es nur eine Zuweisung geben. Gegebenenfalls ist die Zuweisung zur Verdeutlichung aufzuteilen.
> Mehrfachzuweisungen sind verboten.

Es ist zwar nicht unmöglich aber unwahrscheinlich, dass im zweiten Absatz genau das gleiche gemeint ist wie im ersten, also muss es um etwas anderes gehen.

M71 (S/P): Zuweisungen werden verboten.

Die Vorgabe sagt:

> Zuweisungen in Ausdrücken sind verboten, d. h. sie sind nur an der Anweisungspositionen [sic] erlaubt (siehe hierzu Regel 3).

Eine Zuweisung kann nach der Syntax von C nur in einer *assignment-expression* erfolgen; wie schon der Name sagt, ist dies eine *expression*, d. h. ein Ausdruck, und damit verboten.

Die in der Vorgabe folgenden Beispiele machen allerdings plausibel, dass das Ziel dieser Regel ein anderes ist: es geht darum, dass der Wert eines Zuweisungsausdrucks nicht weiterverwendet werden soll. Dieses Verbot ist unmotiviert und erklärungsbedürftig. Beispielsweise ist nicht klar, warum ein praxisbewährtes Idiom wie

```
while ((c = getchar()) != EOF) {
    ...
}
```

durch den unübersichtlicheren Code

```
do {
    c = getchar();
    if (c != EOF) {
        ...
    }
} while (c != EOF);
```

ersetzt werden muss.

Ich könnte mir vorstellen, dass dieser Punkt der Vorgaben auf der Fehlinterpretation eines in manchen Programmierregeln anzutreffenden Verbots basiert, das Ziel einer Zuweisung im betroffenen Ausdruck nochmal zu verwenden. Das bezieht sich aber auf das Problem der fehlenden Sequenzpunkte; siehe dazu die nächste Regel.

M72 (v/ü): Das Verbot von Änderungen zwischen Sequenzpunkten ist missverständlich formuliert und überflüssig.

Die Vorgabe sagt:

> Ein Objekt auf dem [sic] zwischen zwei Sequenzpunkten zugegriffen wird, darf dort nicht verändert werden.

Diese Aussage verbietet *jede* Modifikation, denn auch Zuweisungen sind Zugriffe zwischen zwei Sequenzpunkten.

Gemeint ist vermutlich, dass ein Objekt, auf das zwischen zwei Sequenzpunkten *mehrfach* zugegriffen wird, in diesem Intervall nicht verändert werden darf. Das wird aber vom C-Standard von vornherein verboten [20, Abschnitt 6.5].

3.1.14 Quelldateien

M73 (L): Die Forderungen an Bestandteile von Quelldateien sind zum Teil erklärungsbedürftig, zum Teil praxisfern und zum Teil so detailliert, dass sie nicht mehr universell anwendbar sind.

Erklärungsbedürftig sind folgende Punkte:

- In welchem Rahmen muss ein „Datei-Identifikator“ eindeutig sein?
- Wozu muss der Änderungsautor aus der Datei erkennbar sein?
- Ist mit „Beschreibung der Schnittstelle“ die angebotene Schnittstelle gemeint oder sollte es „Beschreibung der genutzten Schnittstellen“ heißen?

Praxisfern ist das Anführen von Funktionen, die in anderen Quelldateien definiert sind und in der vorliegenden Datei aufgerufen werden; ein Blick auf die `#include`-Direktiven ist meist als Information ausreichend. Für genauere Aussagen sollte man auf automatisch erzeugte Listen und keinesfalls auf Kommentare zurückgreifen, da die Erfahrung zeigt, dass bei gleicher Information in Code und Kommentar die beiden Bestandteile eine Tendenz haben, bei Änderungen inkonsistent zu werden.

Auch die Angabe von Fehlern oder Warnungen im Rahmen der „Kurzbeschreibung“ einer Quelldatei ist nicht besonders sinnvoll, da diese Informationen funktionsspezifisch sind und eine Quelldatei in der Regel mehr als eine einzige externe Funktion enthält (siehe auch Abschnitt (15) in der Vorgabe).

Eine Beschreibung der Ziele, die mit dieser Regel erreicht werden sollen, hätte weniger Platz eingenommen und wäre allgemeiner anwendbar gewesen. Sinnvolle Ziele könnten folgende sein:

- Eine ausgelieferte Quelldatei muss rückverfolgbar sein, d. h. es muss aus dem Inhalt der Datei mit vertretbarem Aufwand ermittelt werden können, um welchen Stand welcher Datei es sich handelt.
- Aus dem Inhalt einer Quelldatei muss direkt oder indirekt (z. B. durch Verweis auf die Dokumentation) eine Beschreibung der angebotenen Schnittstellen entnehmbar sein.

Wichtig wäre auch gewesen, die Rückverfolgbarkeit von Objektdateien zu ihren Quellen mit einer geeigneten Regel sicherzustellen.

M74 (p): Es gibt sinnvolle Anwendungen für `#define` innerhalb einer Quelldatei.

Die Vorgabe fordert, `#define` nur am Beginn einer Quelldatei zu verwenden. Das ist sinnvoll, solange es um in der gesamten Datei genutzte Makros geht. Es gibt aber Fälle, in denen man die Fähigkeiten des Präprozessors nur lokal in einem begrenzten Code-Stück benötigt. (Siehe dazu auch M39.)

Ein Beispiel dafür ist die Konversion von Bezeichnern zu Strings:

```
const char *s;

#define Fall(f)  case f: s = #f; break;

switch(errno) {
Fall(ENOSYS)
...
}
printf("Fehler: %s.\n", s);

#undef Fall
```

Dies eliminiert das Risiko von Inkonsistenzen.

M75 (V): Was sind „globale Definitionen von Strukturen“?

Die Vorgabe sagt:

> Globale Definitionen von Strukturen sollen in Header-Files definiert werden.

Ein kurzer Blick in das Stichwortverzeichnis des C-Standards zeigt, dass es das Wort „global“ in C nicht gibt. Was könnte also gemeint sein?

Die erste Schwierigkeit ist die Interpretation von „Definition von Strukturen“. Sind Variablendefinitionen oder `struct`-Definitionen (also Typdefinitionen) gemeint? Davon hängt ab, wie man „global“ interpretieren kann.

Bei Variablen fallen mir folgende Möglichkeiten ein:

- Es geht um Variablen, die statische Speicherdauer haben.
- Es geht um Variablen, deren Deklarationen File Scope haben.
- Es geht um Variablen, die aus mehr als einer Quelldatei sichtbar sind.

Bei Typdefinitionen (nicht nur für `struct`) scheint mir Folgendes möglich:

- Es geht um Typen, deren Definitionen File Scope haben.
- Es geht um Typen, die in mehr als einer Quelldatei verwendet werden. (Dies soll nicht den Fall einschließen, dass Variablen dieser Typen zwischen den Quelldateien ausgetauscht werden.)

Die einzige dieser möglichen Interpretationen, die mir sinnvoll erscheint, ist die, dass Variablen, die aus mehr als einer Quelldatei sichtbar sind, in einer gemeinsamen Header-Datei deklariert sein müssen. Das sollte allerdings jeder Programmierer sowieso wissen.

M76 (V): Die Forderung, dass die Nutzungsweisen der beiden Arten von Angaben für Include-Dateien definiert werden müssen, ist unverständlich.

Der Unterschied zwischen den Angaben `<...>` und `"..."` besteht darin, dass bei der zweiten die genannte Datei zunächst an weiteren implementationsdefinierten Stellen gesucht werden kann; wird sie dort nicht gefunden oder unterstützt der Compiler diese Suche nicht, ist das Verhalten dasselbe, als hätte man die erste Form verwendet [20, Abschnitt 6.10.2].

Ein Unterschied besteht also nur dann, wenn eine zu inkludierende Datei entweder nur mit `"..."` gefunden wird oder wenn mit `"..."` eine andere Datei gefunden wird als mit `<...>`. Im ersten Fall hat der Programmierer bei der Notation keine Wahl (es sei denn, er kann das Verhalten des Compilers ändern), im zweiten hat man ein so gravierendes potenzielles Konsistenzproblem, dass man eine solche Situation nach Möglichkeit untersagen sollte.

Die Bemerkung, dass bei Standard-Headern `<...>` verwendet werden soll, lässt es aber auch möglich erscheinen, dass die Vorgabe die unterschiedliche Notation für Include-Dateien für einen anderen Zweck nutzen möchte, als von der Funktionalität impliziert wird. Das wäre erstens eine unkluge Vermischung von logisch unabhängigen Eigenschaften und zweitens für die meisten Arten der Strukturierung völlig unzureichend. Sinnvoller wäre stattdessen eine Regel, die eine Reihenfolge und eventuell strukturierende Kommentare bei der Angabe der `#include`-Direktiven vorsieht. In meinen Quelldateien sieht das meist so aus:

```
...
/* Standard-Header */
#include <assert.h>
...

/* Plattform-Header */
#include <unistd.h>
...

/* Anwendungsspezifische Header */
#include "myprog.h"
...
```

In Fällen, wo es zu jeder Implementationsdatei genau eine Header-Datei gibt, in der die Schnittstellen der Implementationsdatei deklariert sind, fordern manche Regeln auch, diese „eigene“ Header-Datei als erste zu inkludieren. Das stellt sicher, dass diese Header-Datei alle nötigen `#include`-Direktiven enthält; allerdings ist diese Art Prüfung nicht vollständig, falls eine Implementationsdatei mehr als eine Schnittstelle exportiert, und sie erfasst auch nicht alleinstehende Header ohne Implementationsdatei. Da Vollständigkeitsprüfungen von Header-Dateien trivial zu automatisieren sind, halte ich eine solche Regel nicht für besonders nützlich.

Für die Portabilität der Quelldateien erheblich wichtiger ist die Frage, welche Zeichen (insbesondere Pfadseparatoren) in `#include`-Direktiven verwendet werden dürfen.

3.1.15 Funktionen

M77 (P): Funktionsdeklarationen ohne Prototyp sind ein vermeidbares Risiko.

Meines Erachtens ist es schon seit Mitte der neunziger Jahre nicht mehr gerechtfertigt, Rücksicht auf Compiler zu nehmen, die keine Funktionsprototypen unterstützen: syntaktisch unpassende Funktionsaufrufe sind einfach ein zu häufiger Fehler.

Eine Sonderregel für die Wartung älteren Codes ist auch überflüssig, da man sich bei der Wartung üblicherweise an die ursprünglich für diesen Code gültigen Regeln hält.

Falls den eigenen Programmierregeln zudem ein Stand des C-Standards zugrundeliegt, der Funktionsprototypen enthält, wird man normalerweise auch einen Compiler benötigen, der erfolgreich gegen diesen Standard geprüft wurde. Dann verliert man durch ein solches Verbot auch keine Funktionalität.

Die einzige realistische Ausnahmesituation scheint mir, wenn man zwar den aktuellen C-Standard für den eigenen Code zugrundelegt, für die Ziel-Plattform aber nur ältere Compiler zur Verfügung hat. In so einer Situation sollte der Einsatz zusätzlicher Werkzeuge für Prüfungen (*lint* ist der Klassiker) zwingend vorgeschrieben sein.

M78 (v): Es ist unklar, was mit „Funktionsrahmen“ gemeint ist.

Ist einfach eine *function-definition* mit einleitenden Kommentaren gemeint?

M79 (v): Es wäre übersichtlicher und klarer gewesen, in der Aufzählung der Teile eines „Funktionsrahmens“ zwischen Code und Kommentaren zu trennen.

Beispielsweise ist nicht ohne Weiteres klar, was der Unterschied zwischen „Parameterliste“ (unter „Funktionskopf“) und „Parameter-Deklaration“ ist.

M80 (p): Es ist nicht klar, welcher Teil der Verwendung von variablen Parameterlisten verboten wird.

Die Vorgabe sagt:

> Es dürfen keine Funktionen verwendet werden, die eine variable Argumentenliste (unbestimmte Anzahl von Argumenten) zulassen.

Der Ausdruck „verwenden“ deutet daraufhin, dass solche Funktionen nicht aufgerufen werden dürfen, aber warum steht diese Aussage dann im Abschnitt (15), wo es um Funktionsdefinitionen geht, statt im Abschnitt (4)?

Der Unterschied ist relevant für diejenigen Funktionen, die von einer standardkonformen Entwicklungsumgebung bereitgestellt werden und eine variable Anzahl von Argumenten zulassen (`printf()` usw.). Diese Funktionen sind so nützlich (darum stehen sie ja im Standard), dass ein kategorisches Verbot überzogen erscheint. Sinnvoll wäre stattdessen eine Forderung, dass Aufrufe solcher Funktionen zentral in typsichere Funktionen mit einer festen Anzahl von Argumenten gekapselt werden müssen und dass diese Funktionen besonderen Prüfungen zu unterwerfen sind.

M81 (p): Es ist unklar, ob `inline` verwendet werden darf.

3.1.16 Rekursive Funktionen

M82 (P): Das Verbot rekursiver Funktionen ist unmotiviert und führt in einigen Fällen zu schlechterem Code.

Es ist bekannt, dass Implementationen mit rekursiven Funktionen immer in Implementationen mit Schleifen überführt werden können und umgekehrt. Also kann das Verbot rekursiver Funktionen nicht aus der damit verbundenen Funktionalität motiviert sein. Insbesondere hat man bei beiden Implementationen das Problem, die Endlichkeit der Verarbeitung sicherstellen zu müssen.

Es kann allerdings auch nicht mit der Klarheit und Verständlichkeit des erstellten Codes zu tun haben, denn bei rekursiv angelegten Algorithmen (beispielsweise Quicksort) sind Implementationen mit rekursiven Funktionen deutlich übersichtlicher und einfacher verifizierbar als solche mit Schleifen.

Die Forderung, dass die maximale Rekursionstiefe eine statische Größe sein muss, deutet darauf hin, dass das diese Forderung motivierende Problem ein möglicher Stack-Überlauf gewesen sein könnte. Dieses Risiko besteht aber auch schon, wenn man keinerlei direkte oder indirekte Rekursion einsetzt, und kann das Verbot somit auch nicht motivieren.

3.1.17 Zeiger

M83 (ü): Die Regel zur Initialisierung von Zeigern ist unnötig.

Die Regel ist bereits in der Regel zur Initialisierung aller Variablen (Abschnitt (11)) enthalten.

M84 (v/l/P): Die Regeln zur Verwendung von Zeigern sind zum Teil unklar formuliert, zum Teil widersprüchlich und zum Teil unnötig einschränkend.

Die Vorgabe sagt:

> Es dürfen keine Zeiger auf lokale Variable verwendet werden, die außerhalb des Gültigkeitsbereiches der Variable liegen.
>
> ...
>
> Eine Funktion darf keinen Zeiger auf eine nicht statische lokale Variable zurückliefern.
>
> ...
>
> Adressen von `auto`-Variablen dürfen nur in `auto`-Variablen mit der gleichen Sichtbarkeit gespeichert werden.
>
> Adressoperatoren sind nur für statische globale Variablen erlaubt.

Das Ziel dieser vier Regeln ist offensichtlich zu verhindern, dass ein Zeiger auf einen Speicherbereich verweist, der bereits wieder anders verwendet wurde. Das kann man mit den in C üblichen Begriffen präziser und umfassender[12] sagen:

> Wenn die Adresse einer Variablen in einem Zeiger gespeichert wird, darf dieser Wert nur während der Lebensdauer („lifetime“) der Variablen verwendet werden.

Ein Verstoß gegen diese Regel ist aber ein klarer Programmierfehler, ähnlich wie die Division durch Null oder der Zugriff über `NULL`, und braucht meines Erachtens genausowenig wie letztere in Programmierregeln aufgezählt zu werden.

Man beachte ferner die Inkonsistenz zwischen den drei letzten Aussagen der Vorgabe: nach der ersten ist es anscheinend gestattet, einen Zeiger auf „statische lokale“ Variablen[13] zu ermitteln (sonst wäre das Verbot, ihn zurückzuliefern, sinnlos), nach der letzten ist das verboten (unter der Annahme, dass „lokal“ und „global“ komplementäre Begriffe sind). Und wie bekommt man im Allgemeinen die Adresse einer skalaren `auto`-Variablen, wenn nicht mit dem Adressoperator?

Die Einschränkung von `&` auf Variable mit „static storage duration“ (ob nun im File Scope oder allgemeiner) scheint mir zudem nicht gerechtfertigt, obwohl sie zugegebenermaßen den Charme hat, eine ganze Klasse von Fehlern weitgehend zu eliminieren. Wenn man sie aufrechterhalten will, sollte man allerdings auch das Freigeben dynamisch allokierten Speichers untersagen, denn damit hat man das gleiche Problem.

M85 (P): Zeiger auf Funktionen sind zu nützlich, um sie zu vermeiden.

Es wäre wünschenswert, eine Begründung dieses Verbotes zu erhalten. Verständlich ist es meines Erachtens nur unter der Annahme, dass die Autoren an Funktionen ohne Prototyp gedacht haben; siehe dazu M77.

M86 (P): Die Ausnahme für das Verbot der Verwendung von `void *` ist unnötig restriktiv.

Das Problem bei `void *` ist, dass der Compiler bei Operationen mit einem solchen Zeiger keine Typprüfungen vornehmen kann. Das ist aber nur dann problematisch, wenn man den Speicherbereich über einen Cast mit einem Typ versieht, der zum tatsächlichen Typ nicht kompatibel ist, und man den Inhalt dann interpretiert. Alle Funktionen, die den Inhalt eines so übergebenen Speicherbereichs gar nicht interpretieren und beispielsweise nur mit seiner Größe operieren (`memcmp()` ist so ein Beispiel) sind dagegen harmlos. Dies umfasst auch die in der Vorgabe genannte Ausnahme der dynamischen Speicherverwaltung.

3.1.18 Typkonvertierung

M87 (P): Die „integer promotions“ sind harmlos aber verboten.

Die Vorgabe verbietet sämtliche impliziten Typumwandlungen und damit auch die „integer promotions“, obwohl diese den Wert einschließlich Vorzeichen erhalten [20, Abschnitt 6.3.1.1].

Dies bedeutet beispielsweise, dass bei einem `int j` und einem `long int l` im Folgenden nur die linke Seite erlaubt ist:

```
l = (long int)j;                l = j;
```

Es ist nicht erkennbar, warum dies zu besserem Code führen soll, und es ist meines Erachtens unübersichtlicher und bei Typänderungen sogar fehlerträchtiger.

[12] Die von mir gegebene Formulierung erfasst auch den Fall des Zugriffs über einen Zeiger, der auf dynamisch allokierten Speicher verweist, der bereits wieder freigegeben wurde. Die Autoren der Vorgabe scheinen diesen Fall für unproblematisch zu halten, denn sie verbieten ihn im Gegensatz zu den anderen Fällen nicht.

[13] Gemeint sind wohl Variablen mit „static storage duration“ und Scope innerhalb eines „function scope“.

M88 (P): Es gibt implizite Umwandlungen zu einem Typ mit geringerem Wertebereich, die offensichtlich harmlos sind und bei denen ein Cast zu verwirrendem Code führt.

In C sind Zeichenliterale vom Typ `int` [20, Abschnitt 6.4.4.4].[14] Daher gestattet das Verbot der impliziten Typumwandlungen im Folgenden nur die linke Variante:

```
char z = (char)'A';          char z = 'A';
```

Das wird den durchschnittlichen Leser eher verwirren als erleuchten.

Die fehlende Definition des Begriffes „typkorrekte Konvertierung“ ist bereits unter M48 behandelt worden und wird hier nicht wiederholt.

M89 (S/P): Das Programmierbeispiel enthält mehrere Fehler.

Das Beispiel lautet in Auszügen:

```
long a;
char b;
...
/* Umwandlung von long nach char mit Bereichsüberprüfung */
if(a <= 255)
    b = (char)a;
else
}
    /* Fehlerbehandlung */
    ...
}
```

Die Probleme dieses Codes sind folgende:

- Die direkte Angabe der Konstanten `255` ist schlechter Stil, da so die Rolle dieses Wertes nicht offenbar wird; zudem ist es nach Abschnitt (10) sogar verboten.
- Es wird vorausgesetzt, dass `char` vorzeichenlos ist.
  Bei einem Compiler, der `char` mit Vorzeichen versieht, muss das Programm für Werte von `a` im Bereich von `CHAR_MAX+1` bis `255` eine Abbildung in den Bereich [`CHAR_MIN`, `CHAR_MAX`] vornehmen. Das Verhalten in dieser Situation ist implementationsdefiniert und kann sogar ein Signal auslösen [20, Abschnitt 6.3.1.3].
- Bei Werten von `a` unterhalb von `CHAR_MIN` wird nicht erkannt, dass eine Bereichsüberschreitung vorliegt.
- Die einleitende Klammer des `else`-Zweigs ist falsch herum.

Eine korrekte `if`-Abfrage könnte dagegen so aussehen:

```
if (CHAR_MIN <= a && a <= CHAR_MAX)
...
```

3.1.19 Datenzuordnung

M90 (L): Der Begriff „projektierungsabhängig“ wird nicht definiert.

Ich habe zudem den Verdacht, dass eine Präzisierung die Forderung der Vorlage trivial werden lassen wird, so dass die Forderung überflüssig werden würde.

[14] In C++ sind Zeichenliterale dagegen vom Typ `char`.

3.1.20 Crossreferenzliste

M91 (L): Die Forderung nach Erstellen solcher Listen ist nutzlos, solange kein Zweck genannt wird.

Es ist beispielsweise nicht erkennbar, ob die Vorgabe fordert, dass die Entwicklungsumgebung so etwas leisten muss (damit der Programmierer es einfacher hat) oder ob solche Listen archiviert werden müssen.

Ferner ist nicht festgelegt, welchen Bereich eine solche Liste abdecken muss (eine Quelldatei, eine *translation-unit*, ein Programm oder ein ganzes Projekt?).

3.1.21 Codierung in Assembler

Zu diesem Abschnitt habe ich keine Anmerkungen.

3.1.22 Wechsel der Programmiersprache

M92 (V): Was ist ein „Modul“?

Ist beispielsweise eine einzelne Quelldatei, eine *translation-unit* oder noch etwas anderes gemeint?

M93 (p): Das Verbot des Mischens von Code aus verschiedenen Sprachen in einem „Modul“ reduziert die Portabilität.

Ich beschränke mich in der Diskussion auf den Fall von Assembler. Der C-Standard nennt dafür als „common extension“ auch das Schlüsselwort `asm` [20, Abschnitt J.5.10], das in C++ zum Standard gehört [22, Abschnitt 7.4]. Ich erwarte daher, dass `asm` in Zukunft auch in C in den Standard aufgenommen werden wird.

Mit `asm` kann man Assembler-Code so in eine C-Funktion einfügen, dass das Ergebnis von Änderungen in den Aufrufkonventionen für Funktionen unabhängig ist. Dies verbessert mindestens die Portabilität zwischen Compiler-Versionen eines einzelnen Herstellers.

3.1.23 Bibliotheksroutinen

M94 (L): Es fehlt die Definition von „Bibliotheksroutine“.

Die Vorgabe sagt:

> Bibliotheksroutinen außerhalb der Standardbibliothek dürfen nur verwendet werden, wenn hierfür ein Funktionsnachweis erbracht wird.

Wann ist eine Funktion aber eine „Bibliotheksroutine“? Wenn sie in einer Bibliotheksdatei (`*.lib`, `*.a` o. ä.) enthalten ist? Das kann auch bei Funktionen der Fall sein, die zur gerade entwickelten Software gehören.

Vermutlich ist beabsichtigt, dass Funktionen nur dann genutzt werden dürfen, wenn sie nachweislich einen hinreichenden Qualitätsstandard haben. Das sollte aber für jede Funktion einschließlich derer in der Standardbibliothek gelten und daher auch so formuliert werden. Aus Sicht von Programmierregeln gehört es zudem eher zu den Voraussetzungen, d. h. in den Abschnitt (2).

M95 (V/P): Es ist nicht klar, ob den Autoren bewusst war, was sie mit dem Verbot der Nutzung von Funktionen ohne explizite Begrenzung von Speicherbereichen bewirken.

Die Vorgabe sagt:

> Andere Funktionen der Standardbibliothek dürfen nur benutzt werden, wenn diese eine explizite Begrenzung des verwendeten Speicherbereiches aufweisen.

Es wird dann darauf hingewiesen, dass aufgrund dieser Forderung die Funktionen `strcpy()`, `strcat()` und `strcmp()` nicht verwendet werden dürfen. Aus der ausdrücklichen Angabe von `strcmp()` folgt, dass die Autoren insbesondere auch den lesenden Zugriff auf Bereiche variabler Länge als problematisch betrachten.

Das muss dann aber genauso auf jede Funktion zutreffen, der beispielsweise ein `'\0'`-terminierter String ohne Längenbegrenzung übergeben wird. In der Standardbibliothek haben unter anderem folgende Funktionen diese Eigenschaft:

```
fopen(), gets(), puts(), atoi(), getenv(), system(), strcspn(),
strlen()
```

Es wird auch nicht klar, warum dieses Verbot ausdrücklich auf Funktionen der Standardbibliothek beschränkt ist, so dass eigener Code sich nicht daran zu halten braucht.

M96 (s): Die Funktion `strncmp()` kann problemlos auch bei überlappenden Speicherbereichen genutzt werden.

M97 (s): Die Funktion `memcpy()` wird durch den Sprachstandard nicht missbilligt.

Der Standard sagt nur, dass ein Aufruf mit überlappenden Speicherbereichen zu undefiniertem Verhalten führt [20, Abschnitt 7.21.2.1] (siehe dazu auch M100). Sofern die Performance-Anforderungen das Ersetzen von `memcpy()` durch `memmove()` gestatten, ist das selbstverständlich sinnvoll, aber für eine solche Regel kann man sich nicht auf den Standard berufen.

3.1.24 Programmprotokoll

M98 (V): Es wird nicht klar, was mit „Programmprotokoll“ gemeint ist und welcher Zweck mit dieser Forderung verfolgt wird.

Nach der Schilderung der Vorgabe ist zu vermuten, dass das vom Compiler erstellte Übersetzungsprotokoll („Listing“) gemeint sein könnte. Das wäre dann aber eine Anforderung an die Übersetzungsumgebung und würde in den Abschnitt (2) gehören.

3.1.25 Optimierung

M99 (V): Es ist unklar, warum die Optimierungsoptionen des Compilers besonders genannt werden und wer die korrekte Funktion sicherstellen muss.

Die Vorgabe sagt:

> Die Optimierungsoptionen des Compilers dürfen verwendet werden, wenn
> - dies bei der Validierung/Genehmigung des verwendeten Compilers als zulässig erklärt wurde und
> - vom Hersteller im Rahmen der Qualitätssicherung die korrekte Funktion des erzeugten Codes sichergestellt wird.

Diese Aussage hat folgende Probleme:

- Die Optimierungsoptionen werden vermutlich besonders genannt, weil sie Einfluss auf den erzeugten Code haben. Das gilt aber auch für andere Optionen, beispielsweise solche zur Wahl des Speichermodells oder der Aufrufkonventionen für Funktionen. Warum werden die nicht erwähnt?

- Wer ist mit „Hersteller“ gemeint: der des Compilers oder der der Software? Falls letzteres: ist mehr als das gemeint, was jeder professionell arbeitende Hersteller von Software heutzutage im Rahmen seiner Qualitätssicherungsmaßnahmen für die eigene Software sowieso tut oder soll der Hersteller wirklich jedesmal wieder den Compiler testen? Falls ja, warum wird das dann nicht logisch konsequent bis zur Hardware durchgezogen und gefordert, dass der Software-Hersteller auch sicherstellen muss, dass die richtige Anzahl Elektronen durch die CPU wandert?
- Logisch gehört diese Forderung vermutlich in den Abschnitt (2).

3.2 Lücken

M100 (S): Es fehlt eine Regel, die Code mit undefiniertem Verhalten generell untersagt.

An einigen Stellen werden Konstruktionen verboten, die nach dem Standard zu undefiniertem Verhalten führen (beispielsweise das Ändern von String-Literalen in Abschnitt (10)). Solche Konstruktionen sind als Programmierfehler aufzufassen und man müsste sie eigentlich nicht in Programmierregeln anführen. Trotzdem kann dies aus Gründen der Klarheit wünschenswert sein.

Wie ein Blick in den C-Standard zeigt [20, Abschnitt J.2], sind die in der Vorgabe genannten Fälle aber nur ein verschwindend geringer Anteil der möglichen Fälle undefinierten Verhaltens. Man sollte sie dann alle prinzipiell untersagen.

M101 (P): Es fehlt eine Regel, die Code untersagt, der sich auf unspezifiziertes Verhalten verlässt.

Unspezifiziertes Verhalten [20, Abschnitt J.1] kann vom Compiler-Hersteller von Einflüssen abhängig gemacht worden sein, die dem Entwickler nicht bekannt sind. Eine Regel, die Code verbietet, der sich auf so etwas verlässt, erhöht nicht nur die Portabilität sondern vor allem die Zuverlässigkeit des Codes.

M102 (p): Es fehlt eine Regel, die Code schützt, der sich auf implementationsdefiniertes Verhalten verlässt.

Implementationsdefiniertes Verhalten [20, Abschnitt J.3] kann sich zwischen Compilern unterscheiden. Eine Regel, die Einschränkungen an Code verhängt (beispielsweise Kapselung in Compiler-abhängige `#if`-Bedingungen), der sich auf so etwas verlässt, erhöht die Portabilität des Codes.

M103 (L): Es fehlt eine Aussage, für welche Art von Programmen die Regeln gedacht sind.

Aus einigen der Aussagen gewinnt man den Eindruck, dass bei den Regeln nur an Anwendungs-Code oberhalb von Compiler und Betriebssystem gedacht wurde (beispielsweise bei der Wahl von Bezeichnern in (9)). Das sollte man dann sagen.

M104 (P): Es fehlen viele sinnvolle Regeln.

Neben den vorstehend genannten übergreifenden Lücken kann man beim Vergleich mit industrieüblichen Programmierregeln zahlreiche weitere sinnvolle Einschränkungen finden, die in der Vorgabe nicht genannt sind. Hier eine persönliche und unvollständige Auswahl:

- Regeln zum Inhalt von Kommentaren (Sprache, kein Duplizieren des Codes, Übereinstimmung mit Code)
- In Headern nicht mehr inkludieren als für den Header selber nötig (ggf. Vorwärtsdeklarationen statt `#include`-Direktiven)

- Initialisierte Arrays immer ohne Größenangabe
- Bei Bedingungen, die ihrem Wesen nach nicht boolesch sind, sollte man den Vergleich mit `0` explizit ausschreiben.
- Keine Verwendung der Ergebnisse von Vergleichsoperatoren in arithmetischen Ausdrücken
- Einsatz von `sizeof` statt Konstanten
- `#else` und `#endif` mit Kommentar
- Regeln zum Verhindern zu großer Shifts oder arithmetischer Überläufe
- Shifts nicht als Ersatz für arithmetische Operationen verwenden
- Verbot von Casts bei Zeigern auf Funktionen
- Verbot von Statements oder Initialisierungen zwischen `{` und dem ersten `case` im `switch`
- Makro-Parameter immer in Klammern, ebenso zusammengesetzte Makro-Ergebnisse
- Kennzeichnung absichtlich leerer Statements
- Keine Nebenwirkung von `assert()`-Argumenten
- Zeiger auf freigegebenen Speicher sofort neu belegen
- Immer auf erfolgreiche Allokation testen
- Abbruchbedingungen von Schleifen möglichst robust formulieren
- Bei der Wartung den Stil einer Datei bewahren

3.3 Bewertung

Die große Anzahl der hier beschriebenen Mängel ist sicherlich bedenklich[15], wichtiger ist aber die Art der dabei aufgedeckten Probleme. Ich fasse sie wie folgt zusammen:

- Das Dokument enthält zahlreiche Lücken und Widersprüche in Bezug auf die Sprachdefinition von C.
- Das Dokument enthält zahlreiche logisch missverständliche, unverständliche, überflüssige und widersprüchliche Aussagen.
- Bei etlichen Forderungen sind die praktischen Auswirkungen nicht hinreichend bedacht worden.
- Das Dokument zeigt im Vergleich mit anderen Regelwerken dieser Art umfangreiche praxisrelevante Lücken, sowohl bei der Darstellung (vor allem fehlende Begründungen) als auch im Inhalt.

Man kann zwar hinter den Forderungen der Vorgabe in vielen Fällen durchaus sinnvolle Ziele ahnen, sie gehen aber fast nie über das hinaus, was fast jede andere Programmierrichtlinie für C auch leistet.

Diese Vorgabe des Eisenbahn-Bundesamtes ist daher meines Erachtens im Vergleich zu industrieüblichen Regelwerken dieser Art nicht einmal von durchschnittlicher Qualität; für die direkte Anwendung durch Entwickler halte ich sie für gänzlich untauglich, da die Ableitung zutreffender und brauchbarer Aussagen zuviel Aufwand erfordert und unter Umständen sogar unterbleibt, so dass es zu Fehlern kommt.

[15] Nach meiner persönlichen Erfahrung entspricht die Mängeldichte etwa dem Doppeltem von dem, was man von einem Dokument durchschnittlicher Qualität erwarten muss, das noch nie einem Review unterzogen wurde.

4 Vorgaben des Eisenbahn-Bundesamtes zum Einsatz von C++

Dieser Abschnitt befasst sich mit den Vorgaben des Eisenbahn-Bundesamtes zum Einsatz der Programmiersprache C++ [12]. Diese Vorgaben sind nahezu identisch zum vorausgegangenen Entwurf [9].

4.1 Inhalt der Vorgaben

Die folgenden Unterabschnitte enthalten jeweils Aussagen zu den gleichnamigen Abschnitten in [12]. Angaben wie „[3.1.1]“ beziehen sich auf den mit dieser Nummer bezeichneten Unterabschnitt von [12].

4.1.1 Vorbemerkung

M105 (L/s): Der Bezug zu C wird unzureichend behandelt.

Die Vorgabe sagt:

> Im Sprachumfang der Programmiersprache C++ ist die Programmiersprache C mit enthalten. Es gelten deshalb zusätzlich zu diesem Teil die Anforderungen des Teils 42 720.

Der erste Satz ist zwar weitgehend korrekt aber nicht ganz: die Menge gültiger C-Programme ist keine Teilmenge der Menge gültiger C++-Programme. Der Anhang C des C++-Standards [22] führt insbesondere solche Unterschiede der beiden Sprachen an, die bei Verwendung in einem C-Programm erlaubt sind, in C++ jedoch nicht.[16]

Darüber hinaus verbietet der Teil 42 720 [11] aufgrund seiner Betonung der Angabe zugelassener Elemente (beispielsweise bei der expliziten Liste erlaubter Operatoren) viele Erweiterungen gegenüber C, obwohl sie nach [12] erlaubt sind (z. B. `new`-Aufrufe); daher widersprechen sich die beiden Dokumente.

M106 (S): Der zugrundeliegende Sprachstandard ist nicht spezifiziert.

Die Vorgabe sagt:

> Der zugelassene Sprachumfang von C++ ist eine Untermenge des ANSI-Standards.

Im Gegensatz zur Situation bei C hat es keinen eigenständigen amerikanischen Standard für C++ gegeben; stattdessen hat man von vornherein den internationalen Standard ISO/IEC 14882 erarbeitet.

Ich beziehe mich im Folgenden auf den C++-Standard von 2003 [22]. Für das vorliegende Dokument sind die Unterschiede zur Vorgängerversion von 1998 [19] allerdings irrelevant, so dass die hier gemachten Aussagen auch dafür gelten.

4.1.2 Voraussetzungen für den Einsatz

Dieser Abschnitt weist die gleichen Mängel auf wie der gleichlautende Abschnitt der C-Vorgaben (siehe Seite 10).

[16] Es ist für die hier angestellten Betrachtungen irrelevant, aber man sollte sich trotzdem darüber im Klaren sein, dass der C++-Standard von 2003 [22] inhaltlich weiterhin auf den C-Standard von 1990 Bezug nimmt und nicht auf den von 1999 [20].

4.1.3 Strukturvorgaben

M107 (v): [3.1.1] Es wird nicht erläutert, was mit „Basisliste“ gemeint ist.

Insbesondere ist unklar, ob man in einer „Basisliste“ einen *access-specifier* verwenden darf. Das `class`-Beispiel suggeriert, dass das untersagt wird; aus dem Beispiel in Abschnitt 3.1.4 folgt, dass man es darf.

M108 (p): [3.1.2] Die Sortierung der *member-declarations* nach den Zugriffsrechten ist nicht immer sinnvoll.

Als Beispiel betrachte man den Fall eines Attributes, das `private` ist und über `public` get- und set-Methoden zugänglich sein soll. Der resultierende Code wird wartbarer sein, wenn diese drei Elemente in der Klassendefinition nebeneinander stehen.

Ich bin zwar der Meinung, dass die genannte Regel im Allgemeinen sinnvoll ist, aber ich bin auch der Überzeugung, dass die hinter einer solchen Regel typischerweise stehende Begründung[17] auf der nicht immer haltbaren Annahme basiert, dass eine Klassendefinition in erster Linie vom Anwender lesbar sein muss und nicht den Zielen des Implementierers unterzuordnen ist. Dieser Interessenkonflikt ist selbstverständlich nicht immer lösbar, aber ich hätte mir zumindest eine informierte Diskussion dieses Punktes und vielleicht sogar eine entprechend flexibel definierte Regel gewünscht.

M109 (P): [3.1.3] Der übliche Grund für das Verbot, `inline`-Funktionen in der Klassendefinition zu definieren, ist überholt und das Ergebnis ist schlechter wartbarer Code.

Die EBA-Vorgaben geben keinen Grund für dieses Verbot an. Es machte Sinn, solange die `inline`-Unterstützung der C++-Compiler noch unzureichend war (bis etwa zur Mitte der neunziger Jahre) und man damit rechnen musste, `inline`-Funktionen auf einigen Plattformen als normale Funktionen definieren zu müssen. Das ist längst überholt.

Darüber hinaus macht es den Code wartbarer, wenn man nur eine einzige Deklaration einer Funktion zu pflegen hat.

M110 (s): [3.2.1] Die „implicit `int`“-Regel von C (vor 1999) gilt in C++ nicht.

Die Vorgabe behauptet:

> Ohne die Angabe des Rückgabetyps wird vom Compiler als Rückgabetyp `int` implizit verwendet, ...

Diese Aussage ist falsch [22, Abschnitt 7.1.5], daher ist die Regel überflüssig.

M111 (P): [3.2.2] Für das Weglassen der Bezeichner von Funktionsparametern gibt es sinnvolle Einsatzfälle.

Die Vorgabe sagt:

> Die Namen von Funktionsargumenten müssen spezifiziert werden. Sie sind identisch in Deklarationen und Definitionen.

Gemeint sind offensichtlich Funktionsparameter und nicht Funktionsargumente.

[17] Die EBA-Vorgaben sagen nur, dass dies der „Übersichtlichkeit“ dienen soll; in [17, Abschnitt 6.1] findet man eine ausführliche Begründung.

> In C++ ist ausdrücklich vorgesehen, dass man in einer Funktionsdefinition einem Parameter keinen Bezeichner geben muss, falls dieser Parameter im Funktionsrumpf nicht verwendet wird [22, Abschnitte 8.3.5 und 8.4]. Diese Situation ist für Parameter nützlich, die an einer Schnittstelle für zukünftige Erweiterungen vorgesehen sind oder die bei der Implementation einer Callback-Funktion nicht benötigt werden; das Weglassen des Bezeichners unterdrückt dann oft Warnungen des Compilers, dass dieser Parameter nicht verwendet wird.

Zudem finde ich die Regel, dass die Funktionsparameter in Deklarationen und Definitionen gleich benannt sein müssen, zwar nicht schädlich aber auch nicht besonders hilfreich. Die eigentlichen Probleme (Verständnis der Funktionsdeklaration durch den Aufrufer und semantische Konsistenz von Schnittstellenbeschreibung und Implementation) werden mit dieser Regel nicht gelöst.

4.1.4 Programmiervorgaben

M112 (V/s): [4.1.1] Die Einschränkung für die Verwendung von `struct` ist nicht verständlich.

Die Vorgabe sagt:

> Das Schlüsselwort `struct` (Strukturen) darf im Bereich von objektorientierten Programmen nur bei Schnittstellen zu C-Code und bei globalen Daten verwendet werden. Dabei dürfen die Strukturen nur Objekte enthalten die als `public` deklariert sind (Standardzuweisung).

Diese Aussage hat folgende Probleme:

- Wann ist ein C++-Programm ein „objektorientiertes Programm“?
- Was sind „globale Daten“?
- Was ist der Zweck dieser Einschränkung?

Da `class` und `struct` abgesehen von den anfänglichen Zugriffsrechten, die man in beiden Fällen überstimmen kann, in C++ identische Bedeutung haben und somit eines der beiden Schlüsselwörter eigentlich überflüssig wäre, liegt es nahe, diese beiden mit verschiedenen Bedeutungen zu versehen.

Eine auf syntaktischen Aspekten basierende Möglichkeit ist, `struct` für *POD-struct* [22, Abschnitt 9] zu reservieren. Solche Typen sind POD-Typen, daher mit C-Mitteln kopierbar [22, Abschnitt 3.9] und auch in C-Programmteilen interpretierbar. Die Einschränkung auf `public` ist dafür irrelevant; sie ist nur von Bedeutung, wenn dieselbe Definition in C- und C++-Programmen verwendet werden soll, aber dann ist die Regel unzureichend, um syntaktische Kompatibilität zu erreichen: abgesehen von dem offensichtlichen Fall eines eigentlich überflüssigen „`public:`“ beachte man, dass die EBA-Vorgabe bei `struct` weder Vererbung noch Methoden verbietet.

Eine andere (auf der Semantik basierende) Variante wäre, `struct` für Klassen zu reservieren, die keine Konsistenzbedingungen zwischen ihren Datenmitgliedern garantieren [26].

M113 (v/P): [4.1.2] Die Regel zur Verwendung von `union` ist unklar formuliert und verbietet nützliche Anwendungen.

Die Vorgabe sagt:

> Das Schlüsselwort `union` (Varianten) darf nur verwendet werden, wenn die Eindeutigkeit der Varianten gewährleistet ist.

Vermutlich ist statt „Eindeutigkeit der Varianten“ „eindeutige Erkennbarkeit der Varianten-Zugehörigkeit eines Objekts“ gemeint.

Ein häufiger Verwendungszweck von `union` ist aber die effiziente und typsichere Uminterpretation eines Objektes, beispielsweise die Aufspaltung eines `unsigned int` in einzelne benannte Bitfelder. In diesem Fall sind die beiden Varianten prinzipiell nicht unterscheidbar, da es sich nur um unterschiedliche Zugriffsarten auf dasselbe Objekt handelt.

Es wäre auch hilfreich gewesen, an dieser Stelle klarzumachen, inwieweit die angegebene Regel die Vorgabe aus [11], Abschnitt (6), modifiziert oder sogar aufhebt.

M114 (p): [4.1.3] Die Forderung, alle Datenmitglieder (data members) einer `class` als `private` zu definieren, ist unnötig restriktiv.

Die Begründung der Vorgabe lautet:

> Eine ungewollte Veränderung der Mitgliedsdaten wird somit vermieden. Außerdem würden als `public` deklarierte Datenelemente gegen grundsätzliche Regeln der objektorientierten Programmierung verstoßen.

Die Behauptung zu den „grundsätzlichen Regeln der objektorientierten Programmierung“ bezieht sich vermutlich auf das Prinzip der Datenkapselung.[18]

Offensichtlich sind `const`-Attribute gegen Änderungen geschützt und können daher immer ohne Änderungsrisiko offengelegt werden.

Das Problem der Änderungen betrifft aber sowieso nur Daten, bei denen die Klasse eine Konsistenz mit anderen Daten voraussetzt; in diesen Fällen sollte man in der Tat den ändernden Zugriff immer über Funktionen leiten, denen diese Konsistenzbedingungen bekannt sind. Wenn es aber keine solchen Randbedingungen gibt, spricht aus diesem Grund nichts dagegen, ein solches Attribut auch `public` zu machen. Ein möglicher Anwendungsfall ist beispielsweise ein Datenmitglied, dessen Wert den Umfang von Trace-Ausgaben steuert.

Datenkapselung ist natürlich nur sinnvoll auf solche Daten anwendbar, die nicht nach außen sichtbar sein sollen. Das hängt zum einen von der Semantik der Daten ab (wenn sie logisch Teil einer Schnittstelle sind, sollte man sie auch syntaktisch offenlegen) und zum anderen vom Ausmaß, in dem ihre Implementation logisch zwingend vorgegeben ist (Information Hiding). Auch da ist die richtige Umsetzung aber fallweise zu entscheiden und man kann sie nicht über eine formale syntaktische Regel wie hier erreichen.

Eine `private`-Regel für Datenmitglieder ist nach meiner Meinung zwar als Empfehlung durchaus tauglich, da sie in den meisten Fällen zur besseren Lösung führt, als Forderung ist sie aber zu streng.

M115 (P/s): [4.1.4] Die Regel für `const`-Methoden sollte erweitert werden und ihre Begründung ist unzutreffend.

Die Vorgabe sagt:

> Mitgliedsfunktionen einer Klasse, die keine Daten ändern sollen, sind mit dem Schlüsselwort `const` zu deklarieren.
>
> ...
>
> Mit Funktionen, die `const` deklariert wurden, können ungewollte Veränderungen von Mitgliedsdaten der Klasse vermieden werden.

Diese Aussagen haben folgende Probleme:

- Man sollte nicht nur Instanzmethoden, die keine Daten ändern sollen, als `const` deklarieren, sondern alle Instanzmethoden, die als `const` deklarierbar sind. Andernfalls sind diese Methoden für `const`-Objekte nicht verwendbar.
- Auch in einer `const`-Methode kann man unter Umständen ein Datenmitglied des betroffenen Objektes legal und ohne Cast modifizieren. Dies geht bei als `mutable` deklarierten Mitgliedern oder über Pointer, die nicht auf `const` zeigen. Eine `const`-Deklaration der Methode ist also nicht ausreichend, „ungewollte Veränderungen“ zu verhindern.

M116 (s): [4.1.5] Jede Klasse weist einen Kopier-Konstruktor auf.

Vermutlich ist ein Anwender-definierter statt eines implizit deklarierten Kopier-Konstruktors gemeint.

[18] „A public variable represents a violation of one of the basic principles of object-oriented programming, namely, encapsulation of data.“ [17, Abschnitt 7.1, Grund 1]

M117 (P): [4.1.5–4.1.6] Die Regeln für Klassen mit Verwaltung dynamischen Speichers sind unübersichtlich formuliert und nicht ausreichend.

Die Vorgabe fordert für Klassen, die `new` und `delete` einsetzen, in [4.1.5] einen Kopier-Konstruktor, in [4.1.6] einen Zuweisungsoperator und als Folge von [4.1.8] auch einen Destruktor. Diese Forderung hat folgende Mängel:

- Das hinter den ersten beiden Regeln stehende Problem (member-wise copy durch implizit deklarierte Funktionen) beschränkt sich nicht auf Klassen, die `new` und `delete` einsetzen, sondern betrifft alle Klassen, die Zeiger für Beziehungen verwenden, bei denen mit dem Erzeugen, Ändern oder Zerstören des Zeigers besondere Aktionen verbunden werden müssen. Das Verwalten objektspezifisch dynamisch allokierten Speichers ist nur ein Spezialfall davon; andere Varianten sind das Nutzen objektspezifischen Speichers aus einem statischen Pool (also ohne `new` oder `delete`) sowie das An- und Abmelden bei der anvisierten Objektinstanz, beispielsweise für Reference Counting.
- Es reicht nicht, zum Verhindern des implizit deklarierten Zuweisungsoperators irgendeinen Zuweisungsoperator zu definieren: es muss ein Kopier-Zuweisungsoperator sein.
- Die Definition der genannten Funktionen ist nicht ausreichend, die Probleme beim Umgang mit Zeigern zu beheben. Dafür sind Auflagen an das Verhalten dieser Funktionen sowie *sämtlicher* definierter Konstruktoren und Zuweisungsoperatoren notwendig.

Übersichtlicher und korrekter wäre eine Vorgabe folgender Art, die insbesondere alle implizit deklarierten Methoden (member functions) unterbindet:

> Klassen, die Zeiger enthalten, bei denen Änderungen mit Aktionen auf dem anvisierten Objekt verknüpft sein sollen, müssen explizite Deklarationen für den Kopier-Konstruktor, den Kopier-Zuweisungsoperator und den Destruktor enthalten. Alle definierten Konstruktoren, Zuweisungsoperatoren und der Destruktor müssen die geforderten Aktionen berücksichtigen.

Man kann auch noch einen nicht-Kopier-Konstruktor zur Liste zu definierender Funktionen hinzufügen, da man ihn dann ohnehin benötigt; allerdings ist er nicht aus Gründen der Konsistenz notwendig, da der Kopier-Konstruktor bereits die implizite Deklaration eines Default-Konstruktors verhindert.

Auf zwei Punkte der obigen Regel will ich besonders hinweisen:

- „Zeiger enthalten“: damit ist die Realisierung über normale Zeiger gemeint. Falls aber beispielsweise eine Speicherverwaltung stattdessen über „smart pointer“ geschieht, die „tief“ kopieren, zeigen die implizit deklarierten Funktionen korrektes Verhalten, so dass man keine expliziten Deklarationen braucht. Die Regel ist dann auf die Klasse für smart pointer anzuwenden. (Ein Beispiel für so etwas ist die Standard-Klasse `string`.)
- Die Regel fordert nur Deklarationen, nicht Definitionen.

M118 (s): [4.1.5] Man kann Klassen nicht einander zuweisen.

Die Vorgabe sagt:

> Ein Kopier-Konstruktor wird benötigt, um Fehler zu vermeiden, wenn einer Klasse eine andere Klasse gleichen Typs zugewiesen wird.

Gemeint sind hoffentlich Objekte und nicht Klassen, der Kopier-Konstruktor wird bei Initialisierungen und nicht bei Zuweisungen aufgerufen, und er wird auch gerufen, wenn es im Programm nicht um Objekte gleichen Typs sondern um Instanzen von Typen mit einer gemeinsamen Basisklasse (der Klasse des betrachteten Kopier-Konstruktors) geht.

M119 (S/P): [4.1.6] Der Beispiel-Code enthält gravierende Fehler.

- Die beispielhaft geschilderte Situation

  ```
  X x = y; /* x und y sind Objekte vom Typ X */
  ```

  enthält entgegen der Behauptung der Vorgabe keine Zuweisung sondern eine Initialisierung; daher wird der Kopier-Konstruktor und nicht ein Zuweisungsoperator aufgerufen.
- Bei der Klassendefinition fehlt an 3 Stellen ein Semikolon und der *access-specifier* muss `private` statt `privat` heißen.
- Beim Kopier-Konstruktor wird gegen die Forderung der Namensgleichheit in Regel [3.2.2] verstoßen.
- Die beiden Konstruktoren sind nicht gegen Nullzeiger für `cp` bzw. `s.sp` abgesichert.
- In beiden Konstruktoren wird im `new`-Aufruf ein Zeichen zuwenig angefordert.
- Im Destruktor wird `delete` statt `delete[]` aufgerufen.

Jeder einzelne der drei letzten Fehler kann den Absturz des Programms verursachen.

Außerdem ist es verwirrend, im Kopier-Zuweisungsoperator als Namen für den Parameter denselben Bezeichner wie für ein Datenmitglied zu verwenden.

Einige dieser Fehler finden sich auch im nahezu wortgleichen Beispiel 25 der Ellemtel-Regeln [17]. Zusammen mit anderen Parallelen ist daher fast sicher, dass den Autoren des EBAs dieses Dokument vorgelegen hat und als Ausgangsmaterial verwendet wurde. Allerdings hätten Anstand und intellektuelle Redlichkeit dann auch geboten, dies mit einem Literaturhinweis anzuerkennen. Vielleicht wäre dann ferner jemandem aufgefallen, dass ein paar Regeln durch nach der Veröffentlichung von [17] erfolgte Änderungen an C++ obsolet geworden sind (z. B. beim in M110 erwähnten „implicit `int`“). Eine überarbeitete Fassung [18] der Ellemtel-Regeln existierte bereits vor der ersten Erstellung der EBA-Vorgaben.

M120 (S): [4.1.7] Die Regel für virtuelle Destruktoren hat eine falsche Voraussetzung.

Die Regel und ihre Begründung lauten:

> Sind in einer Klasse virtuelle Funktionen deklariert, so ist auch immer ein virtueller Destruktor zu deklarieren.
>
> ...
>
> Eine Klasse, die als Basisklasse dient und virtuelle Funktionen enthält ohne einen virtuellen Destruktor zu besitzen, kann zu fehlerhaftem Code führen, wenn Zeiger auf Objekte vom Typ der abgeleiteten Klassen benutzt werden.
>
> Werden diese Zeiger zum Entfernen der Objekte mittels `delete` benutzt, so würde nur der Destruktor der Basisklasse aufgerufen werden, auch wenn die abgeleitete Klasse einen eigenen Destruktor besitzt.

Die erste der beiden Begründungen ist falsch und nur die zweite trifft zu.[19] Die Regel basiert aber auf der ersten Begründung.

Der Fall, um den es hier geht, ist der Aufruf eines `delete` für ein Objekt einer abgeleiteten Klasse aber über einen Zeiger vom Typ „Pointer auf eine Basisklasse“. In dieser Situation sollte die Basisklasse einen virtuellen Destruktor haben. Die An- oder Abwesenheit anderer virtueller Funktionen ist für dieses Problem dagegen irrelevant; insbesondere kann der Fehler auch bei Klassen auftreten, die keine einzige virtuelle Funktion haben.

Stattdessen hätte eine Forderung wie die Regel 10.4 in [18] Sinn gemacht, nach der jede öffentliche Basisklasse entweder einen öffentlichen virtuellen Destruktor oder einen geschützten Destruktor haben muss.

[19] Die erste ist vermutlich übernommen aus [17, Abschnitt 7.5], Regel 26. Die Erläuterung scheint auch von dort kopiert zu sein.

M121 (P): [4.1.8] Die Regel über zu definierende Funktionsmitglieder ist nicht erklärlich.

Es wird kein Grund für diese Regel angegeben.

Ich vermute, ihre Formulierung ist aus Betrachtungen von Klassen mit dynamischer Speicherverwaltung (Regeln [4.1.5–4.1.6]) entstanden. Dort ist diese Forderung (nach einigen Präzisierungen und Erweiterungen) auch berechtigt, in anderen Situationen ist sie es im Allgemeinen nicht. Als Beispiel betrachte man eine Klasse, die aus irgendeinem Grund (beispielsweise klassenweite Initialisierungen und Aufräumarbeiten) einen Zähler über die Anzahl ihrer Instanzen führen muss; in diesem Fall benötigt man Anwender-definierte Konstruktoren und den Destruktor aber keinen Zuweisungsoperator.

Dagegen lässt sich durchaus für die *Deklaration* dieser Funktionen argumentieren. Fügt man dann auch noch den Default-Konstruktor zu dieser Liste hinzu (und mindestens einen weiteren Konstruktor muss man zwingend haben, sonst kann man keine Objekte anlegen), erhält man die „orthodoxe kanonische Klassen-Form“ [5].

M122 (p): [4.1.9] Die Einschränkungen an Referenzen und Zeiger sind nicht immer gerechtfertigt.

Dies ist das gleiche Problem wie in M114.

M123 (v): [4.1.10] Diese Regel ist unverständlich formuliert.

Die Vorgabe sagt:

> Funktionen einer Klasse, die in einer abgeleiteten Klasse detailliert werden, sind grundsätzlich als virtuelle Funktionen zu vereinbaren.

In erster Linie ist unklar, was mit „detaillieren“ gemeint ist. Vermutlich soll aber der Fall betrachtet werden, dass eine Funktion, identifiziert über ihre Signatur, in einer abgeleiteten Klasse spezialisiert werden soll, ohne dass sich die Semantik gegenüber der Implementation in der Basisklasse ändert. Für diesen Fall ist die Regel korrekt, da nur so polymorphes Verhalten erreicht werden kann.

Die Formulierung der Regel ist aber unglücklich, da sie Anforderungen an die Basisklasse aufgrund von Ansprüchen abgeleiteter Klassen stellt. Da Basisklassen in der Regel vor abgeleiteten Klassen vorliegen, stellt dies die tatsächliche auftretende Situation auf den Kopf. Besser wäre eine Formulierung, die der abgeleiteten Klasse verbietet, eine Funktion mit identischer Signatur wie eine nicht-virtuelle Funktion in einer Basisklasse zu definieren.

M124 (P): Es fehlt eine Regel, die semantisch einheitliches Verhalten für alle Implementationen einer virtuellen Funktion fordert.

In der Regel bedingt dies, dass die oberste Basisklasse mit einer Deklaration der Funktion eine Schnittstellenbeschreibung vorzugeben hat, an die sich alle Implementationen halten müssen.

Formalisiert man dies mit Vor- und Nachbedingungen, so folgt, dass eine Implementation in einer abgeleiteten Klasse höchstens die gleiche Vorbedingung voraussetzen darf und mindestens die gleiche Nachbedingung garantieren muss („expect less and deliver more“ [26]).

M125 (P): [4.1.11] Das Verbot von Mehrfachvererbung ist nicht erklärlich.

Es gibt Anwendungsfälle für Mehrfachvererbung, für die in C++ kein gleichwertiger Ersatz existiert (beispielsweise Mixin-Klassen [1]), daher ist ein generelles Verbot nicht verständlich.

Möglicherweise ist auch nur das Verbot der Diamant-förmigen Vererbung gemeint; das vermeidet zwar Ärger mit Up-Casts (bei nicht-virtueller Vererbung) oder bei der Initialisierung (bei virtueller Vererbung), doch außer wenn man die Übersicht verliert verhindert es keine Fehler, da der Compiler eventuelle Probleme entdeckt (was dann den Ärger verursacht).

Manche Programmierregeln (z. B. [26]) raten, die Mehrfachvererbung auf Basisklassen zu beschränken, bei denen es sich um reine Schnittstellenklassen handelt (also keine nicht-statischen Datenmitglieder, alle nicht-statischen Funktionsmitglieder sind „pure virtual“). Das sollte aber nur für das öffentliche Ableiten und alle Basisklassen bis auf eine gelten.

M126 (s/p): [4.2.1] Leere Parameterlisten sind in C++ keine unspezifizierten Parameterlisten.

Die Vorgabe sagt:

> Nicht spezifizierte Parameterlisten sind verboten.

Das darauf folgende Beispiel macht deutlich, worum es geht:

```
int funktion_1();         /* verboten */
int funktion_2(void);     /* erlaubt */
```

Diese beiden Funktionen haben aber identische Signatur, denn eine leere Parameterliste bedeutet in C++ nicht wie in C eine unspezifizierte Liste. Die Regel bewirkt also nichts als das Einfügen überflüssiger `void`-Angaben und macht den Code damit unübersichtlicher.[20]

Es gibt mindestens ein Regelwerk zu C++ [16, Abschnitt 9.1], das `void`-Angaben an dieser Stelle außer in gemeinsamem C- und C++-Code ausdrücklich untersagt.

M127 (P): [4.2.2] Die Regel, nicht modifizierte Parameter als `const` zu deklarieren, ist weitgehend überflüssig und verfehlt ein wichtigeres Ziel.

Das für die Regel angegebene Beispiel,

```
int funktion(const int A);
```

zeigt deutlich den Unsinn der Regel: einen „by value“ übergebenen Parameter wie hier als `const` zu deklarieren, ist für die Aufrufer wirkungslos. Aus diesem Grund legt der Standard auch fest [22, Abschnitt 8.3.5], dass solche *cv-qualifier* nicht in den Typ der Funktion eingehen, d. h. Deklarationen mit und ohne dieses `const` bezeichnen die gleiche Funktion. Ein `const` für diesen Parameter wirkt nur, wenn man es in der *Definition* der Funktion angibt; das erscheint mir bei Funktionsparametern (die ja keine Compile-Zeit-Konstanten sein können) nicht sonderlich nützlich. Auf keinen Fall aber sollte man eine solche interne Implementationseigenschaft im Header offenlegen.

Statt dessen wäre eine Regel sinnvoll gewesen, nach der Parameter, die nicht immer modifiziert werden und die mehr Speicher als ein Pointer benötigen, nach Möglichkeit über Zeiger auf `const` oder über `const`-Referenzen an die Funktion übergeben werden sollten, um unnötiges Kopieren zu vermeiden.

M128 (P/ü): [4.2.3] Dieses Verbot ist je nach Standpunkt unzureichend oder überflüssig.

Die Vorgabe sagt:

> Zeiger oder Referenzen lokaler Variablen dürfen von öffentlichen Funktionen nicht zurückgegeben werden oder Mitgliedsdaten zugewiesen werden.

Diese Regel verbietet folgendes nicht:

- Rückgabe von Zeigern oder Referenzen auf `auto`-Variablen aus einer als `protected` oder `private` deklarierten Funktion
- Anlegen von Zeigern oder Referenzen auf `auto`-Variablen in Variablen statischer Speicherdauer oder in Variablen eines umschließenden Blockes

[20] Regeln die verlangen, dass man dort, wo nichts ist, „der Klarkeit halber“ auch hinschreiben muss, dass dort nichts ist („This page intentionally left blank.“), sind aber auch ziemlich lächerlich. Man sieht das am einfachsten, wenn man sie logisch konsequent durchzieht.

Richtig ist aber, dass all dies Programmierfehler sind, die beispielsweise durch die in M84 angegebene Regel verboten werden, wenn man sie von Zeigern auf Referenzen erweitert.

M129 (V): [4.3.1] Es ist nicht erkennbar, was mit dem Verbot von Zeigern auf Methoden erreicht werden soll.

Die Vorgabe sagt:

> Zeiger auf Mitgliedsfunktionen von Klassen sind verboten.

Das ist erklärungsbedürftig (siehe auch M85).

M130 (V): [4.3.2] Die Aussage zum Wechsel zwischen Zeigern und Referenzen ist unverständlich.

Die Vorgabe sagt:

> Wechsel zwischen Zeigern und Referenzen sind innerhalb von Funktionsaufrufen zu vermeiden.

Aus dem Begriff „Wechsel“ wird klar, dass es um eine Situation gehen muss, in der sich der Programmierer zwischen Zeigern und Referenzen entscheiden kann. Das kann nur in Variablendefinitionen geschehen und die gibt es in Funktionsaufrufen normalerweise nicht, also kann es nicht um Aufrufe sondern es muss um Funktionsdefinitionen gehen.

Folglich sieht es so aus, als wollten die Autoren eine Einschränkung an Mischungen von Zeigern und Referenzen in Funktionssignaturen verhängen. Aber warum? Eventuelle Fehler in den Aufrufen (fehlendes oder überzähliges `&` oder `*`) werden wegen der Typsicherheit von C++ sowieso vom Compiler aufgedeckt. Und in welchem Rahmen sind solche Mischungen zu vermeiden: innerhalb einer einzelnen Funktion, für alle Funktionen einer Klasse, für sämtliche Funktionen einer Quelldatei oder für das ganze Projekt?

Oder wollte man vielleicht einfach nur sagen, dass man innerhalb einer Funktion nicht parallel zueinander einen Zeiger und eine Referenz zum Zugriff auf das gleiche Objekt verwenden soll?

M131 (P): [4.4.1] Das Verbot von `dynamic_cast` ist nicht verständlich.

Die mit `dynamic_cast` erreichbare Funktionalität kann man selber mit virtuellen Funktionen nachimplementieren.[21] Dieser Weg ist nicht verboten, er erfordert aber zusätzlichen Aufwand und bietet unnötigerweise Gelegenheit, zusätzliche Fehler zu machen. All das ließe sich mit `dynamic_cast` vermeiden.

M132 (p): [4.4.1] Die Regel zu `reinterpret_cast` lässt sich verbessern.

Die Regel, `reinterpret_cast` nach Möglichkeit zu vermeiden, ist zwar sinnvoll, nach meiner Einschätzung ist dieser Cast aber so oft tatsächlich notwendig, dass man Abweichungen von der Regel detaillierter behandeln sollte.

Eine Möglichkeit wäre die Forderung, dass bei der Verwendung von `reinterpret_cast` in einem Kommentar begründet werden muss, warum diese Umwandlung nötig und harmlos ist. Beispielsweise sind alle Casts harmlos, bei denen man (vielleicht für eine Signaturberechnung oder zum Auslagern in eine Datei) nur an die Byte-Folge des Objektzustands gelangen möchte (Cast auf `unsigned char *`), da sie dem Speicherbereich keine dem Inhalt widersprechende Struktur aufzwingen.

Statt die Cast-Operatoren als „zu bevorzugen“ zu klassifizieren, hätte man die C-Casts besser ausdrücklich untersagen sollen. Nach dem jetzigen Stand lassen sich einige der für die Cast-Operatoren verhängten Regeln mit C-Casts umgehen, ohne gegen die Vorgaben zu verstoßen.

[21] Vor der Einführung von `dynamic_cast` war dies der übliche Weg.

M133 (p): [4.4.2] Die Regel zu `void *` ist unnötig restriktiv.

Dies ist zum Teil dasselbe Problem wie in [11, Regel (17)]; siehe auch M86. Immerhin ist hier eine ausdrückliche Ausnahme für den Fall vorgesehen, dass die Struktur des betroffenen Speicherbereichs nicht interpretiert wird, doch ist nicht erklärlich, warum das nur an Schnittstellen zu „Betriebssystem- und Libraryfunktionen“ gestattet sein soll. (Und ist mit „Libraryfunktion“ eine Funktion aus der Standard-Bibliothek gemeint oder etwas anderes?)

M134 (L/V): [4.5.1] Die Anforderungen an die dynamische Speicherverwaltung sind zu ungenau formuliert, um entscheidbar zu sein.

Die Vorgabe sagt:

> Die dynamische Speicherverwaltung muss ein logisch und zeitlich vorhersehbares Verhalten besitzen. Ferner muss die dynamische Speicherverwaltung die zeitlichen Anforderungen der Anwendung gewährleisten.

In dieser Formulierung sehe ich folgende Probleme:

- Was ist ein „logisch vorhersehbares Verhalten“?
  Die einzige Interpretation, die mir dazu einfällt, ist, dass die Funktionen der dynamischen Speicherverwaltung genau das tun sollen, was sie in ihren Schnittstellenbeschreibungen zugesichert haben, und dass diese Beschreibungen funktional vollständig sein müssen. Das sollte für jede andere Funktion aber genauso gelten.
- Was ist ein „zeitlich vorhersehbares Verhalten“?
  Ist gemeint, dass die Funktionen der dynamischen Speicherverwaltung in ihren Schnittstellenbeschreibungen Aussagen zum zeitlichen Verhalten machen müssen? Wenn ja, muss dies so detailliert sein, dass der Programmierer aus den Daten seines Codes eine garantierte obere zeitliche Schranke für die Aufrufe ableiten kann? Und warum wird das nur für die dynamische Speicherverwaltung und nicht auch für andere Funktionen gefordert? Für Echtzeitsysteme wäre dies generell sinnvoll.
- Was für besondere zeitliche Anforderungen der Anwendung sind gemeint, die sich nur auf die dynamische Speicherverwaltung beziehen? Und in welcher Weise unterscheidet sich dieser Punkt vom „zeitlich vorhersehbaren Verhalten“? Generell kann jede Anwendung doch nur solche Funktionalität einsetzen, die ihr gestattet, ihre Anforderungen zu erfüllen.

M135 (p): [4.5.2] In Schnittstellen zu C-Code kann man gezwungen sein, `malloc()` oder `free()` einzusetzen.

Sinnvoll wäre stattdessen ein Verbot gewesen, die beiden Funktionsgruppen für dasselbe Objekt zu mischen, da das Verhalten dann implementationsspezifisch ist.

M136 (v): [4.5.2] Es wird nicht erläutert, in welchem Umfang `new` und `delete` eingeschränkt werden.

Die Formulierung lautet:

> Statt dessen sind im [sic] eingeschränktem Umfang die Operatoren `new` und `delete` zu benutzen.

Vermutlich sind nur diejenigen Einschränkungen gemeint, die in späteren Abschnitten der Vorgabe verhängt werden. Da ein entsprechender Hinweis fehlt, erweckt die Formulierung aber den Eindruck, dass es davon unabhängige Einschränkungen gibt.

M137 (ü): [4.5.3] Die Regel für `delete[]` ist überflüssig.

Diese Einschränkung wird bereits von der Sprachdefinition verhängt [22, Abschnitt 5.3.5].

Man beachte ferner, dass es ebenfalls Probleme in der umgekehrten Situation (Anwendung von `delete[]` auf Zeiger zu einem nicht-Array-Objekt) gibt, die in der Vorgabe nicht betrachtet wird.

M138 (p): [4.5.4] Die Regel zur `NULL`-Zuweisung ist zu einschränkend formuliert.

Wichtig ist, dass ein Zeiger auf freigegebenen Speicher sogleich wieder einen gültigen Wert bekommt, es muss aber nicht unbedingt der Nullwert sein.

M139 (ü): [4.5.5] Diese Regel ist überflüssig.

Die Formulierung lautet:

> Speicherbereiche dürfen nur dann freigegeben werden, wenn keine Referenzen mehr auf ihn [sic] existieren.

Der Kontext legt nahe, dass „Referenz“ hier umgangssprachlich gemeint ist (Verweis, d. h. pointer oder reference).

Unabhängig davon ist ein Verstoß gegen diese Regel ein Programmierfehler; eine eigene Regel dafür ist nicht nötig. Die Gründe sind die gleichen wie in M84.

M140 (V): [4.5.6] Es ist nicht klar, was mit einer „durchgängigen Fehlerbehandlung“ gemeint ist.

Die Vorgabe sagt:

> Es muss eine durchgängige Fehlerbehandlung für die Operatoren `new` und `delete` implementiert sein.

Die Einschränkung auf `new` und `delete` zeigt, dass es nicht um Fehler in Konstruktoren oder Destruktoren geht, denn die werden auch in anderen Situationen aufgerufen. Folglich muss es sich im Fehler handeln, die beispielsweise auch bei der dynamischen Speicherverwaltung eines `int` auftreten können. Das einzige, womit man in der Situation auch bei fehlerfreier Programmierung rechnen muss, ist aber das Versagen von `new`, weil kein freier Speicher verfügbar ist; `delete` wird dagegen immer erfolgreich sein. Eine Regel, dass Fehler bei `new`-Aufrufen immer in Betracht gezogen werden müssen, wäre also sinnvoll (und ist auch üblich), eine entsprechende Regel für `delete` ist jedoch sinnlos, wie man auch an der Definition von `delete` sehen kann (kein Return Code, keine Exceptions [22, Abschnitt 18.4.1]).

Inwieweit eine solche Regel für `new` „durchgängig“ im Sinne der Vorgabe wäre, ist nicht erkennbar.

M141 (s): [4.6.1] Die geschweiften Klammern `{}` sind keine Operatoren und können daher auch nicht überladen werden.

M142 (V/P): [4.6.1] Das Verbot des Überladens der genannten Operatoren ist erklärungsbedürftig und unnötig einschränkend.

Verboten wird in dieser Regel das Überladen der Operatoren `&&`, `||`, des Komma-Operators `,`, des Aufruf-Operators `()` sowie des Subskript-Operators `[]`.

- Die Gründe für das Verbot, `&&` und `||` zu überladen, werden nicht genannt. Die Tatsache, dass dies bei `!`, `&` und `|` nicht verboten wird, legt aber nahe, dass dahinter die besonderen Regeln für die Reihenfolge der Auswertung bei `operator&&(bool, bool)` und `operator||(bool, bool)` stehen. Das scheint mir keine nennenswerte Fehlerquelle zu sein, allerdings sehe ich auch keine Situation, in der mir ein Überladen von `&&` oder `||` nützlich wäre.

- Die Notationen für `operator()()` und `operator[]()` sind zu nützlich, um das Überladen dieser Operatoren zu verbieten. Hielte man die Regel aufrecht, wären beispielsweise ganze Teile der Standard-Bibliothek verboten.

M143 (S): [4.6.2] `new` und `delete` sind schon im Standard überladen.

Der Standard definiert bereits je 3 Varianten für `new`, `new[]`, `delete` und `delete[]` [22, Abschnitt 18.4.1]. Damit verbietet die Vorgabe also den Einsatz der Standard-Funktionen.

Es ist möglich, dass mit dieser Regel eigentlich das Verbot von Typ-spezifischen `new`/`delete`-Implementationen gemeint war.

M144 (v/p): [4.7.1] Die Regel zur Prüfung von Templates ist missverständlich und unzureichend.

Die Vorgabe sagt:

> Werden Schablonendefinitionen eingesetzt so sind alle Instanzen zu prüfen.

Die Bezeichnung „Instanz“ verwendet man im Deutschen üblicherweise für Objekte („Instanz einer Klasse“), bei Templates spricht man dagegen von „Instanziierung“ (template instantiation).

Gemeint ist also vermutlich, dass Template-Definitionen mit allen genutzten Template-Argumenten zu testen sind, bevor man sie verwenden darf. Das ist sinnvoll, sollte dann aber auch für Funktionsschablonen gelten und nicht nur für Klassen (die Überschrift für den Abschnitt 4.7 lautet „Schablonendefinitionen (templates) von Klassen“).

M145 (P): [4.8.1] Die Beschränkung von `break` auf Ausnahmebehandlungen ist nicht sinnvoll.

C++ fehlt genau wie vielen anderen Programmiersprachen[22] ein vordefiniertes Konstrukt für eine Schleife mit einem Ausgang in der Mitte.[23] Manchmal behilft man sich so:

```
while (true) {
    ...
    if (...) break;
    ...
}
```

In diesem Fall wird `break` für einen regulären Schleifenausgang verwendet und nicht für eine Ausnahmebehandlung. Mir erscheint das legitim und nicht qualitätsmindernd.

Statt einer formalen Regel wie in der Vorgabe wäre eine inhaltliche Regel sinnvoll, nach der jede Schleife nur einen einzigen regulären Ausgang haben darf. Man könnte ferner fordern, dass Schleifen wie oben besonders gekennzeichnet werden müssen, damit der Leser sie nicht mit unendlichen Schleifen verwechselt.

M146 (v): [4.8.1] Die Beschränkung von `continue` auf Ausnahmebehandlungen ist unklar formuliert.

Während man bei `break` die Bezeichnung „Ausnahmebehandlung“ einigermaßen sicher als Synonym für „Schleifenausgang im Fehlerfall“ identifizieren kann, ist bei `continue` nicht klar, was

[22] Die einzige mir erinnerliche Ausnahme ist Modula-2 mit der `LOOP`-Anweisung.

[23] Die Alternative ist Code-Duplikation mit dem Risiko von Inkonsistenzen. Im Prinzip ist das wiederum reduzierbar durch Einführen einer Hilfsfunktion, allerdings hängt die Robustheit dieser Lösung vom Umfang und der Form der Parameter ab, die diese Funktion dann benötigt.

eine „Ausnahme“ sein soll. Ein typischer Anwendungsfall für `continue` ist beispielsweise das zeilenweise Einlesen und Verarbeiten einer Datei, die auch zu ignorierende Kommentarzeilen enthält. Ist das Auftreten einer solche Zeile eine „Ausnahme“ oder nicht?

Darüber hinaus ist nicht erkennbar, was durch diese Regel an Qualität gewonnen wird. Generell führt `continue` meiner Meinung nach zu übersichtlicherem Code, wenn es um zwei Entscheidungswege mit deutlich unterschiedlichem Umfang geht, insbesondere dann, wenn einer der Zweige leer ist. Nur bei etwa gleichem Umfang oder gleichem semantischem Gewicht ist ein `if/else` angemessener.

M147 (P/s): [4.8.2] Diese Regel ist nicht sinnvoll und sie verhindert das selbstgesteckte Ziel.

Die Vorgabe sagt:

> Ausnahmebehandlungsmechanismen in Unterprogrammen mit `catch`, `throw` und `try` sind zu vermeiden (mehrere Ausgänge eines Unterprogramms).

Das Ziel ist also, mehrere Ausgänge eines Unterprogramms zu verhindern. Wenn in einer Funktion aber eine andere Funktion aufgerufen wird, die eine Exception werfen kann (`new` ist das wichtigste Beispiel), dann ist ein `try`-Block in der ersten Funktion *zwingend* erforderlich, um einen zweiten „Ausgang“ dieser Funktion zu verhindern.

Es ist sicherlich sinnvoll (und auch üblich), den Einsatz von `throw` auf Fehlerfälle zu beschränken. Das ist ein Spezialfall der allgemeineren Regel, dass eine Funktion nur einen einzigen regulären Ausgang haben sollte (siehe M19). Ein weitergehendes Verbot führt dagegen manchmal zu komplexeren Schnittstellen (zusätzliche Fehlerwerte) oder schlechteren Design-Entscheidungen (weil beispielsweise ein Konstruktor Fehler nur über Exceptions melden oder sie sich merken kann).

Verbote von `try` oder `catch` sind dagegen gar nicht motivierbar, im Gegenteil: es macht sogar beispielsweise Sinn zu fordern, dass die Verarbeitung in `main()` immer innerhalb eines `try`-Blocks stattfinden muss, der einen Default-Handler, einen Handler für `exception` und ggf. auch für projektspezifische Exceptions enthält.

M148 (P/s): [4.8.3] Diese Regel verbietet auch die Definition von Konstanten.

Die Vorgabe sagt:

> Header-Dateien sollen keine Definitionen von Funktionen und Daten enthalten.

Damit ist so etwas wie

```
const int Antwort = 42;
```

in Header-Dateien verboten, denn es ist eine Definition von Daten.

Darüber hinaus gibt es in Header-Dateien sogar sinnvolle Anwendungsfälle für Definitionen, die keine Konstanten beschreiben.[24] Zu einem Problem kann so etwas nur dann werden, wenn es sich um Elemente mit external linkage handelt. Ein Verbot dafür verhindert Konflikte beim Binden, allerdings sollten Fehler dieser Art auch immer vom Linker erkannt werden.

M149 (V): [4.8.5] Inhalt und Zweck dieser Regel sind nicht erkennbar.

Die Vorgabe sagt:

> Beim Einbinden von fremden [sic] Code (nicht C++) in C++-Programme muss die Eindeutigkeit des einzubindenden Codes gewährleistet sein.
>
> Änderungen am vorhandenen fremden Code sind mittels eines geeigneten Compilers (Assemblers), bzw. eines C++-Compilers mit entsprechenden Optionen zu überprüfen und zu dokumentieren.

[24] Ein Beispiel dafür war der alte XPG-Header `<regexp.h>`.

Das auf diese Aussagen folgende Beispiel weist dann noch darauf hin, dass das Überladen eines Funktionsnamens von einem C-Compiler als Fehler erkannt und von einem C++-Compiler akzeptiert werden muss.

Diese Aussagen weisen eine Reihe von Problemen auf:

- Was ist mit „Einbinden von Code“ gemeint?
  Zunächst würde man an Einbinden in eine C++-*translation-unit* denken, also Einbinden durch den Compiler. Das ist aber nur mit einer `#include`-Direktive und nur dann möglich, wenn dieser Code gültiger C++-Code ist. Es soll aber ausdrücklich um nicht-C++-Code gehen, also kann das hier nicht gemeint sein.
  Die zweite Möglichkeit ist das Zusammenbinden mit Objektdateien, die aus in einer anderen Sprache geschriebenen Quellen erzeugt wurden. Dieser Fall ist praxisnäher als der erste, allerdings ist sein wichtigstes Werkzeug die Linkage Specification über `extern`, und die wird hier nicht einmal erwähnt.
- Was ist „Eindeutigkeit des einzubindenden Codes“? In welchen seiner Eigenschaften soll der einzubindende Code eindeutig sein und innerhalb welchen Rahmens?
- Zunächst wird über das Einbinden von fremdem Code gesprochen, dann über das Ändern von Code einer anderen Programmiersprache, und schließlich darüber, dass ein und dasselbe Code-Stück nicht in allen betrachteten Sprachen gültig sein muss. Wo ist der Zusammenhang zwischen diesen Aussagen?

Beginnt man das Lesen dieser Regel von hinten, gewinnt man den Eindruck, dass es eigentlich um Header-Dateien gehen soll, die Schnittstellen zwischen C und C++ beschreiben, d. h. um Code, der gleichzeitig C- und C++-Code ist. Auch dafür fehlt allerdings die Erwähnung des essenziellen `extern "C"`, und das Problem der „Eindeutigkeit“ wird durch diese Interpretation auch nicht erhellt.

4.2 Lücken

Die C++-Vorgaben weisen dieselben allgemeinen Lücken auf wie die C-Vorgaben des EBA, beschrieben im Abschnitt 3.2 auf Seite 35. Ich wiederhole sie hier nicht sondern beschränke mich auf zusätzliche Mängel.

M150 (V/S): Es ist unklar, warum einige in C++ gegenüber C neue Syntaxelemente nicht erwähnt sind.

Dies betrifft mindestens Scope Resolution, Namespaces, Conversion Functions, Pointer to Member und `mutable`.

Man beachte dabei, dass ohne Scope Resolution oder `using` der Einsatz Standard-konformen C++-Codes nicht allzuviele sinnvolle Anwendungsmöglichkeiten bietet.

M151 (P): Es fehlen viele sinnvolle Regeln.

Hier eine Auswahl bedenkenswerter Regeln:

- Regeln zur Verhinderung von „name space pollution“
- Initialisierung statt Zuweisung
- In Konstruktoren immer alle Datenmitglieder initialisieren
- Keine Aufrufe mit dynamic binding in Konstruktoren oder Destruktoren
- In Zuweisungsoperatoren abgeleiteter Klassen immer auch den Zuweisungsoperator der Basisklasse(n) berücksichtigen
- Keine Änderung von Default-Werten in virtuellen Funktionen abgeleiteter Klassen (oder sogar Verbot von Default-Werten für virtuelle Funktionen)
- Kein gleichzeitiges Überladen bezüglich integraler und Zeiger-Typen
- Überladen von Operatoren immer in zusammengehörigen Funktionsgruppen, konsistent und intuitiv verständlich
- get-Methoden für Datenmitglieder immer `inline`
- Für call-by-reference außer bei Arrays Referenzen statt Zeiger verwenden

- Niemals `delete[]` auf Zeiger auf Basisklasse anwenden (dies ist einer der wenigen Fälle, wo ich eine von der Sprachdefinition bereits verhängte Regel in Programmierregeln wiederholen würde)

4.3 Bewertung

Diese Vorgabe des Eisenbahn-Bundesamtes zum Einsatz von C++ hat Mängel, die noch ein Stück über die bei der Vorgabe für C beschriebenen hinausgehen. Sie zeigt im Vergleich zu letzterer verstärkt eine autoritäre Formulierung der Regeln (unerläuterte Verbote), gleichzeitig aber lange Beispiele zu oder Erklärungen von Sachverhalten, die jeder C++-Programmierer sowieso wissen sollte (beispielsweise in [3.1], [4.1.5] und [4.5.3]).

Meines Erachtens ist die Schlussfolgerung unausweichlich, dass bei der Erstellung und dem Review dieses Dokumentes nicht die für diese Aufgaben nötige Sorgfalt, Sachkenntnis und Erfahrung zur Verfügung standen. Die haarsträubenden Fehler im Programmierbeispiel des Abschnittes [4.1.6] (beschrieben in M119 auf Seite 42) sind dafür ein besonders deutlicher Beleg, allerdings auch nicht der einzige.

Das Ergebnis ist ein Regelwerk, das als Vorgabe für die Software-Entwicklung in C++ nicht einmal ernst zu nehmen ist, geschweige denn, dass es auch nur mittelmäßigen Qualitätsansprüchen genügen würde.

5 Schlussfolgerungen

Nach meiner Einschätzung sind die Vorgaben des Eisenbahn-Bundesamtes für den Einsatz von C und C++ [11, 12] von so bestürzend schlechter Qualität, dass sie nicht einmal als Ausgangspunkt für eigene Programmierregeln tauglich sind; für die direkte Anwendung durch Entwickler sind sie erst recht unbrauchbar.

Meines Erachtens kann man sie daher nur so behandeln, dass man zuerst, ohne auf die Vorgaben des EBAs Bezug zu nehmen, ein Regelwerk von mindestens industrieüblicher Qualität aufstellt. In einem zweiten Schritt sollte man dann in einem separaten Dokument nachweisen, dass diese Regeln mindestens so gut sind wie die des Eisenbahn-Bundesamtes. Dieses zweite Dokument kann sich auf den vorliegenden Bericht stützen.