Sauberes Programmieren

Carsten Deeg

22.01.2017

Inhaltsverzeichnis

Die folgenden Regeln helfen, typische Programmierfehler zu vermeiden. Es wird insbesondere auf C/C++ eingegangen, es beginnt allerdings mit allgemeingültigen Regeln, die generell gelten.

1. Allgemeingültiges

Regel 1.1. *Es gibt* KEINEN *Grund, daß ein Programm abstürzt. Für jeden Absturz ist das Programm selbst verantwortlich. Jede potentielle Fehlerquelle ist zu berücksichtigen.*

Einzige Ausnahme von diesem Kriterium sind Hardware-Fehler (kaputter Prozessor usw.). In C/C++ sind ein `assert` und eine unbehandelte Exception aus Anwendersicht gleichwertig mit Abstürzen!

Regel 1.2. Jede *Aktion ist auf Erfolg zu prüfen (Speicher reservieren, Datei öffnen ...). Auf jede Fehlerquelle ist immer geeignet zu reagieren.*

Das gilt für Rückgabewerte und das Behandeln von Exceptions. Sie sind immer auszuwerten und es muß geeignet darauf reagiert werden. Meistens kann nah am Entstehungsort des Fehlers noch am meisten gerettet werden.

Häufig ist eine geeignete Reaktion nur schwierig zu definieren. In kritischen Steuergeräten muß eine Lösung gefunden werden. Bei gewöhnlichen Anwendungen ist wenigstens eine sprechende Fehlermeldung Pflicht!

Regel 1.3. ALLES*, was man vom System bekommt (Speicher, Dateien ...), ist anschließend wieder freizugeben.*

Für Entwickler, die unter früheren Multitasking-Betriebssystemen das Programmieren gelernt haben, als es noch keinen Speicherschutz usw. gab (z. B. auf einem Amiga), ist dieses Kriterium selbstverständlich. Für viele andere leider nicht...

Regel 1.4. *Redundanzen vermeiden*

Wiederholungen im Programmtext sind offensichtlich und werden durch die gängigen Programmiertechniken (Funktionen, Vererbung ...) vermindert. Redundanzen sind aber nicht nur einfache Kopien im Programmtext, sondern alle Dinge, die mehrfachen Aufwand beim Erstellen und Pflegen erfordern. Und sie sind auch fehleranfällig, weil man bei Änderungen leicht einzelne Stellen vergißt anzupassen.

Noch erträglich sind Redundanzen, die der Compiler prüfen kann und ggf. einen Fehler meldet. Nicht automatisch prüfbare Redundanzen sind zu unterlassen. Wenn sie unvermeidbar sind, dann unbedingt an allen Stellen im Kommentar auf alle anderen hinweisen.

Regel 1.5. *Compiler-Fehlermeldungen nutzen*

Nach Möglichkeit sollen Programmierfehler bereits beim Kompilieren zu Fehlern führen, die der Compiler melden kann. Es gibt drei Stufen, die alle nutzbar sind: Präprozessor, Compiler und Linker

Im **Präprozessor** können Abfragen und Fehlermeldungen programmiert werden:

```
#if !a
#  error a ist nicht erfuellt
#endif
```

Im **Compiler** gibt es seit C11 und C++11 das neue Schlüsselwort `static_assert`:

```
static_assert(a, "a␣ist␣nicht␣erfuellt");
```

In älteren Versionen kann man sich behelfen mit Konstrukten, die bei falscher Parametrierung nicht mehr kompilierbar sind (z. B. Typfehler):

```
/// Kompilierzeit-assert innerhalb von Funktionen
#define CASSERT(a, text) do{ \
  extern char c_assert_ ## text[(a) ? 1 : -1]; \
  (void)(c_assert_ ## text); \
} while(false)

/// Kompilierzeit-assert ausserhalb von Funktionen
#define CASSERT_F(a, text) \
  extern char c_assert_ ## text[(a) ? 1 : -1]
```

Wenn `a` nicht erfüllt ist, wird versucht, ein Array mit negativer Größe anzulegen. Mit `text` kann etwas Text mitgegeben werden, allerdings nicht als String, sondern nur als Bestandteil des Bezeichners. Mehr Text kann als Kommentar bei dem Aufruf des `CASSERT` geschrieben werden.

Praktisch ist auch eine in Rechnungen eingebettete Prüfung. Allerdings muß man prüfen, daß der Compiler bei Division durch eine konstante 0 warnt und sonst alles wegoptimiert:

```
#define CASSERT_R(a) (0/(!!(a)))
#define BEISPIEL_CAST_U8(x) (CASSERT_R((a)<=255) + (uint8_t)(a))
```

Eine weitere Möglichkeit ist das Erzeugen von Fehlern, die erst dem **Linker** auffallen. Damit kann z. B. getestet werden, ob bestimmte Programmteile erfolgreich vom Compiler wegoptimiert werden:

```
#define LASSERT(a)      LASSERT_1(a, __LINE__)
#define LASSERT_1(a, z) LASSERT_2(a, z)
#define LASSERT_2(a, z) do {  \
  extern int assert_fehlgeschlagen_in_zeile_ ## z;      \
  if(!(a))   assert_fehlgeschlagen_in_zeile_ ## z = 1;  \
} while(false)
```

Wenn `a` nicht erfüllt ist, wird auf eine externe Variable zugegriffen, die es nicht gibt. Der Linker wird üblicherweise eine Fehlermeldung ausgeben, die automatisch den Dateinamen enthält und durch die Konstruktion mit `__LINE__` im Variablennamen auch die Zeilennummer. Der stufenweise Aufruf ist notwendig, damit `__LINE__` tatsächlich durch die Zahl ersetzt wird und nicht als Text an den Variablennamen angehängt wird.

Regel 1.6. *`assert` sparsam einsetzen*

`assert` sollte nur in ausgesuchten Fällen Anwendung finden, da es nicht zur Laufzeit behandelbar ist. Nach Möglichkeit sollten potentielle Fehler schon zur Kompilierzeit durch den Compiler erkannt werden und zu Compiler-Fehlermeldungen führen. `assert` ist nur eine Notlösung für Fälle, in denen das nicht funktioniert. Dann muß auch sichergestellt sein, daß alle möglichen Vorkommnisse getestet werden. Ein `assert` ist ein bedingter Haltepunkt beim Testen. Fehler, deren Auftreten im Betrieb nicht ausgeschlossen werden können, dürfen nicht nur mit `assert` abgefangen werden, sondern müssen gezielt behandelt werden.

In einem `switch`-`case`-Block kann ein eigentlich unmöglicher `default`-Zweig ein `assert` enthalten, so daß es beim Testen sofort bemerkt wird. Für den produktiven Betrieb muß hier aber dennoch eine sinnvolle Operation implementiert werden.

Regel 1.7. *Sprechende Fehlermeldungen programmieren*

Zwar kostet das Programmieren vieler Fehlermeldungen Zeit. Allerdings spart man ein Vielfaches davon, wenn man beim Auftreten eines Fehlers gleich eine genaue Meldung erhält, was schief gelaufen ist.

Regel 1.8. *Standardkonform programmieren*

Wenn es irgend geht, sollte man sich an den Sprachstandard halten. Proprietäre Sonderwege einzelner Compiler sind nur sehr selten erforderlich. Es ist auch zu beachten, daß nichtspezifiziertes Verhalten schon mit der nächsten Compiler-Version ganz anders aussehen kann.

Nicht standardkonforme Programmtexte lassen sich nicht leicht portieren, schon ein anderer Compiler funktioniert wahrscheinlich nicht mehr. Und andere Programmierer verstehen standardkonformen Programmtext meistens schneller. Vorteile bringen Sonderwege fast nie (außer vielleicht die Bequemlichkeit, nicht im Standard nachschlagen zu müssen).

Die Verwendung von `#pragma` ist auf das absolut Notwendige zu minimieren.

Regel 1.9. *Programmtext für den Leser schreiben*

Der Programmtext sollte in erster Linie für einen Leser verständlich sein. Auch der Programmierer selbst wird zum Leser, wenn seit dem Schreiben genug Zeit vergangen ist. Deshalb zu erst für den Leser optimieren und erst nachgelagert für den Compiler.

Regel 1.10. *Unsicherheiten klären*

Wenn etwas nicht klar ist, nicht einfach „zur Sicherheit“ irgend etwas tun, sondern offene Fragen und Unsicherheiten klären, um eine fundierte Entscheidung zu treffen. Keine Unsicherheiten unter den Teppich kehren.

Regel 1.11. *Nicht Konzepte von Programmiersprachen nutzen, nur weil sie möglich sind*

Programmierer scheinen gern alle Konzepte einer Sprache ausnutzen zu wollen. Insbesondere in C++ kann man durch sinnlos überladene Operatoren und `templates` unübersichtliche Programme schreiben. Also: nur Konzepte anwenden, die für die Problemstellung wirklich sinnvoll sind. In C++ nur Operatoren überladen, wenn es sinnvoll und verständlich ist (z. B. „+“ nur überladen, wenn auch eine mathematische Addition gemeint ist).

Regel 1.12. *Im Sprachstandard vorhandene Konzepte und Namen benutzen und nicht parallel neu definieren*

Eine Neudefinition führt zur Verwirrung und erfordert unnötigen Mehraufwand.

2. Undefiniertes Verhalten

Regel 2.1. *Die tatsächliche Größe der Standarddatentypen ist undefiniert. Es darf von keinen „üblichen“ Werten ausgegangen werden.*

Auf 32-Bit-Rechnern gilt nur „zufällig“ die Gleichheit dieser Datentypgrößen:
`sizeof(int) == sizeof(long) == sizeof(void*) == 4`.

Wenn bestimmte Größen notwendig sind, sind diese explizit zu programmieren. Seit C99 gibt es die passenden Typbezeichner in `stdint.h`.

Es ist nicht einmal festgelegt, aus wievielen Bits ein `char` besteht (abfragbar mit `CHAR_BIT`). Weiterhin sind außer bei `char` auch sogenannte Padding-Bits möglich, so daß nur diese Beziehung gilt (beachte das $\leq$): $\texttt{UTYP\_MAX} \leq 2^{\texttt{sizeof(typ)}\cdot\texttt{CHAR\_BIT}} - 1$

Laut Standard sind lediglich folgende Eigenschaften zugesichert:

`CHAR_BIT`	≥ 8
`UCHAR_MAX`	$= 2^{\texttt{CHAR\_BIT}} - 1$
`USHRT_MAX`	$\geq \max(2^{16} - 1, \texttt{UCHAR\_MAX})$
`UINT_MAX`	$\geq \texttt{USHRT\_MAX}$
`ULONG_MAX`	$\geq \max(2^{32} - 1, \texttt{UINT\_MAX})$
`ULLONG_MAX`	$\geq \max(2^{64} - 1, \texttt{ULONG\_MAX})$
`UINTMAX_MAX`	$\geq \texttt{ULLONG\_MAX}$

Regel 2.2. *Den Datentyp `char` nur für Zeichen verwenden*

Ob ein `char` standardmäßig vorzeichenbehaftet ist, ist nicht festgelegt. Deshalb ist das immer explizit anzugeben.

Ausnahme ist die Verwendung für Zeichen bzw. Zeichenketten, wofür explizit `char` benutzt wird.

Regel 2.3. *Wenn Operatoren mit Seiteneffekten auf Variablen angewendet werden, dürfen diese Variablen nicht mehrfach in demselben Ausdruck vorkommen.*

Die Auswertereihenfolge von „`i`“ und „`i++`“ ist bei „`i + i++`“ oder „`f(i, i++)`“ nicht definiert.

Regel 2.4. *Das Resultat bei Überläufen in Rechnungen mit vorzeichenbehafteten Ganzzahltypen ist nicht definiert.*

Nur bei vorzeichenlosen Typen ist garantiert, daß die Zahlen einfach umlaufen. Zwar ist dieses Verhalten auch bei vorzeichenbehafteten üblich, aber eigentlich nur „Zufall“. Das gilt auch für die einfache Typkonvertierung von vorzeichenlosen in vorzeichenbehaftete Typen (z. B. beim Zuweisen), wenn das Ergebnis nicht paßt.

Regel 2.5. *Die Operatoren „`>>`“ und „`<<`“ nicht auf vorzeichenbehaftete Zahlen anwenden*

Es ist dem Compiler freigestellt, ob bei „`>>`“ das Vorzeichen beibehalten wird, also ggf. eine 1 nachgeschoben wird, oder eine 0. Bei „`<<`“ ist das Verhalten für negative Zahlen völlig undefiniert.

Regel 2.6. *Zweierkomplement ist nicht garantiert*

Es kann Systeme und C-Compiler geben, die Zahlen nicht im Zweierkomplement verarbeiten. Zulässig sind das Zweierkomplement (üblich), Einerkomplement und Betrag mit Vorzeichen.

Regel 2.7. *Der Präprozessor rechnet immer in `intmax_t`.*

Wenn in Präprozessorabfragen gerechnet wird, erfolgt das immer im maximalen Ganzzahltyp. Er kennt keine Typumwandlung und läßt solche ggf. einfach weg. Dadurch können identische Rechnungen im Präprozessor andere Ergebnisse haben als im Compiler.

Regel 2.8. *Das Resultat bei Überläufen bei Typkonvertierungen ist selten definiert.*

`static_cast<unsigned short>([un]signed long)`	immer definiert, auch bei Überlauf
`static_cast<signed short>([un]signed long)`	undefiniert bei Überlauf
`static_cast<[un]signed int>(float)`	undefiniert bei Überlauf
`static_cast<float>([un]signed int)`	undefiniert bei Überlauf

Regel 2.9. *Berücksichtigen/beeinflussen von Füll-Bytes*

Zur Beschleunigung von Zugriffen fügen Compiler Füll-Bytes zwischen Einträge von Strukturen ein. Da jeder Compiler das unterschiedlich machen kann, ist ggf. darauf Einfluß zu nehmen. Leider geht das nur durch proprietäre Anweisungen (z. B. „`#pragma pack`“ oder „`__attribute__((packed))`“) und seit C11/C++11 in Grenzen auch mit `alignas` und `alignof`.

Erforderlich ist das immer, wenn Programme von verschiedenen Compilern oder auf verschiedenen Plattformen auf dieselbe Struktur zugreifen. In solchen Strukturen sind keine Datentypen erlaubt, deren Länge nicht spezifiziert ist (z. B. `short`, `int`, `long`, untypisierte `enum`, `bool`, `float`, `double`). Stattdessen sind die speziellen fest definierten Typen aus `stdint.h` zu verwenden.

Vom Standard ist festgelegt, daß keine Füll-Bytes vor dem ersten Element eingefügt werden, es also die Adresse `0` bekommt. Weiterhin ist die Reihenfolge der Elemente garantiert. Alles weitere ist freigestellt, Füll-Bytes können beliebig zwischen und nach den Elementen eingefügt werden. Casts zwischen Strukturen und Arrays sind damit undefiniert!

„`#pragma pack`“ ist aber gefährlich. Es führt zu nicht portablen Programmen. Auf manchen Architekturen sind Zugriffe auf z. B. `int` an ungeraden Adressen verboten und führen zum Absturz. Bei Zugriff über die gepackte Struktur (`struct_zeiger->int_variable`) weiß der Compiler das und erzeugt automatisch die richtigen Maschinenbefehle, um das zu umgehen (ggf. langsam). Nach „`int* z = &(struct_zeiger->int_variable)`“ ist eine Dereferenzierung von `*z` aber undefiniert und führt zum Absturz.

Für die Effizienz ist es sinnvoll, Strukturelemente so zu sortieren, daß möglichst wenig Füll-Bytes hinzugefügt werden (zu erst große, dann kleine Elemente). Bei Klassen ist auch die V-Tabelle zu berücksichtigen (die je nach Compiler unterschiedlich angelegt werden kann). Auf x86-Plattform sollten die am häufigsten benutzten Elemente in den ersten 128 Byte sein. Große Arrays sollten immerhin innerhalb der 128 Byte beginnen.

Regel 2.10. *Keine spezielle Speicheranordnung bei Vererbung erwarten*

Viele Details bezüglich der Speicheranordnung von Klassenelementen sind dem Compiler freigestellt. Außer der Reihenfolge der Elemente innerhalb einer einzelnen Klasse und kein Füllbyte vor dem ersten Element (siehe Regel 2.9) sollte man nichts als gegeben erwarten. Die Reihenfolge der verschiedenen Klassen bei Verarbeitung ist nicht klar, schon gar nicht bei Mehrfachvererbung. Die V-Tabelle liegt auch noch irgendwo.

Regel 2.11. *`scanf` nur mit definierten Feldlängen verwenden*

Es ist nur definiert, was passiert, wenn der Wert in der Zeichenkette auch im Zielformat darstellbar ist. Bei einer Zeichenkette ist es offensichtlich, daß die Größe des Zielspeichers angegeben werden muß (`%4s`). Interessanterweise ist auch undefiniert, was beim Einlesen von z. B. 100 000 in eine Ganzzahlvariabel von 16 Bit passiert, da diese Zahl nicht mit 16 Bit darstellbar ist. Es existieren tatsächlich Implementierungen, die bei zu großen Zahlen abstürzen.

Die einzige Möglichkeit ist die Festlegung einer maximalen Zeichenanzahl wie bei den Zeichenketten (`%4i`). Für 16 Bit können so nur 4 Ziffern sicher genutzt werden, da es Zahlen mit 5 Ziffern gibt, die zu groß sind.

Bei wörtlicher Interpretation des Standards sind Fließkommazahlen generell nicht nutzbar, weil normale Werte wie z. B. 0, 1 nicht genau darstellbar sind. Das ist sicherlich nicht gemeint.

Eine sinnvolle sichere Verwendbarkeit von `scanf` und verwandter Funktionen ist angesichts der Einschränkungen kaum möglich. Man sollte also eine Implementierung mit anderen Funktionen erwägen.

3. Speicherverwaltung und Zeiger

Regel 3.1. *Ungültige Zeiger niemals dereferenzieren*

Jedes Dereferenzieren von ungültigen Zeigern ist ein Fehler, der theoretisch sofort zum Programmabsturz führen kann. Man darf also auch nicht ungeprüft „`*p`“ schreiben zur Übergabe an eine Funktion, die eine Referenz erwartet.

Abfragen wie „`if(this)`“ innerhalb von Methoden sind demnach sinnlos, weil bereits der Methodenaufruf mittels des Operators `->` eine Dereferenzierung des `this`-Zeigers darstellt.

Regel 3.2. *Zeiger mit ungültigen Inhalten auf `NULL`/`nullptr` setzen*

Das gilt insbesondere, nachdem der Speicher mit `free`/`delete` freigegeben wurde. Ausnahme sind lokale Variablen am Ende ihres Gültigkeitsbereichs und Klassenvariablen im Destruktor, da sie danach ohnehin nicht mehr zugreifbar sind.

In C ist `NULL` und in C++ `nullptr` zu verwenden.

Regel 3.3. *Vor `free`/`delete` keine Abfrage auf `NULL`/`nullptr`*

Diese Abfrage ist redundant, da sie laut Standard intern durchgeführt wird.

Das gilt auch für `#undef`, vor dem nicht mit `#ifdef` auf die Existenz des Bezeichners geprüft werden muß.

Regel 3.4. *`new` liefert niemals `nullptr` zurück, sondern wirft ggf. eine Exception, die behandelt werden muß.*

Also entweder `try-catch` verwenden oder `new(nothrow)`. Aber Exceptions aus dem Konstruktor können auch dann noch auftreten!

Manche Compiler halten sich allerdings nicht unbedingt an diese im Standard definierte Vorschrift.

Regel 3.5. *Bei Verwendung von `realloc` ist eine temporäre Zwischenvariable für den Zeiger erforderlich.*

Wenn `realloc` fehlschlägt, bekommt man `NULL`/`nullptr` zurück, der alte Speicher bleibt aber bestehen. Wenn man schreibt „`p = realloc(p, s)`“, hat man keinen Zeiger mehr auf den alten Speicher, also ein Speicherleck. Es ist also eine Zwischenvariable nötig, um im Fehlerfall an den alten Speicher heranzukommen.

Übrigens ist es oft effizienter, Speicher erst freizugeben und dann neuen zu reservieren, sofern man den alten Inhalt nicht mehr braucht. `realloc` sollte also nur eingesetzt werden, wenn man den alten Speicherinhalt weiterhin braucht.

Regel 3.6. *Zeigerbesitz eindeutig und einfach gestalten*

Grundsätzlich ist es am übersichtlichsten, wenn Speicher dort wieder freigegeben wird, wo er belegt wurde. Das gilt grundsätzlich auch für andere Dinge wie geöffnete Dateien usw.

Eine verteilte Verwaltung (nicht eindeutiger „Besitz“ eines Zeigers) sollte nur mit gutem Grund gewählt werden.

Bei Übergabe der Verwaltung eines Zeigers oder verteilter Verwaltung von Zeigern sind die im Standard vorhandenen Klassen `std::unique_ptr` und `std::shared_ptr` zu benutzen und keine proprietären Varianten. Damit lassen sich nicht nur Zeiger damit verwalten, sondern auch andere Dinge, die nach Verwendung wieder freigegeben werden müssen.

Regel 3.7. *Bibliotheksfunktionen und (überladene) Operatoren können fehlschlagen und müssen überwacht werden.*

Viele Operationen in C++ (besonders in der STL) nutzen intern dynamischen Speicher. Sie können also fehlschlagen und werfen dann meistens eine Exception: Abfangen!

Regel 3.8. *Zeiger auf ein Array und Zeiger auf das erste Element sind nicht dasselbe.*

Bei einem Array „`char s[10]`“ haben die Zeiger „`s`“, „`&s`“ und „`&s[0]`“ zwar denselben Zahlenwert, sie sind aber von unterschiedlichem Typ. „`s`“ und „`&s[0]`“ sind Zeiger auf einzelne `char`s, „`&s`“ auf Blöcke von 10 `char`s. Bei „`s + 1`“ wird der Zeiger also um 1 `char` weiterbewegt, bei „`&s + 1`“ aber um 10 `char`s. Die Nutzung von „`&s`“ ist unüblich.

Regel 3.9. *`sizeof` auf Variablen und nicht Typen anwenden*

Richtig: `typ var; p = malloc(sizeof(var));`

Falsch: `p = malloc(sizeof(typ))`

Redundanzen sind immer schlecht (siehe Regel 1.4), besonders wenn man sie von Hand pflegen muß und der Compiler sie nicht prüfen kann. Wird der Typ von `var` geändert, stimmt die Größe im `malloc` nicht mehr, wenn dort nicht auch der Typ angepaßt wird. Durch `sizeof(var)` erledigt sich das von selbst.

Regel 3.10. *Array-Längen mittels `sizeof` abfragen*

Richtig: `char s[100]; strncpy(s, "text", sizeof(s))`

Falsch: `strncpy(s, "text", 100)`

Es gilt das gleiche wie in Regel 3.9. Notfalls kann eine Konstante definiert werden, die *überall* eingesetzt wird. Da dieses "überall" aber nicht durch den Compiler prüfbar ist, sollte dem `sizeof` der Vorzug gegeben werden. Wenn das Array nicht aus `char` besteht, kann die Anzahl der Elemente ebenfalls mittels `sizeof` bestimmt werden:

```
#define ELEMENTE(a) (sizeof(a)/sizeof(*a))
```

Oder mit C++:

```
template<typename T, size_t N>
static constexpr size_t ELEMENTE(const T(&)[N]) noexcept {return(N);}
```

Der Compiler löst das schon zur Kompilierzeit auf, es kostet also keine Laufzeit.

Regel 3.11. *Bei Arrays als Parameter eine maximal erlaubte Länge in Array-Schreibweise angeben, sofern eine Einschränkung besteht*

„`void f(char s[100])`“ ist zwar äquivalent zu „`void f(char* s)`“, stellt aber eine übersichtliche Art dar, die Anforderung zu dokumentieren. Auf dennoch zu lange Arrays muß aber selbst geprüft werden, da das der Compiler nicht kann!

Seit C99 kann man auch eine variable Länge angegeben, die zuvor als Parameter übergeben wird: „`void f(size_t n, char s[n])`“

Hinweis: `sizeof(s)` liefert weiterhin die Größe eines Zeigers und nicht des ganzen Arrays.

Regel 3.12. *Bei Zeigerrechnungen auf sinnvolle Typen achten*

Zeiger haben viele individuelle Typen. Auf Maschinenebene sind es vorzeichenlose Ganzzahlen. Differenzen von Zeigern liefern den vorzeichenbehafteten Ganzzahltypen `ptrdiff_t`. Seine Einheit ist nicht Byte, sondern die Elementgröße, auf die die Zeiger zeigen. Größen werden im vorzeichenlosen Ganzzahltypen `size_t` in Byte abgelegt.

Regel 3.13. *Nur konstante Referenzen als Funktionsparameter verwenden*

In C++ sind Referenzen bei der Benutzung nicht von Variablen unterscheidbar. Das gilt für den Funktionsaufruf und bei der Nutzung der Parameter innerhalb der Funktion. Deshalb sind Referenzen nur für konstante Parameter zu benutzen:

```
void f(const typ& eingang, typ* const ausgang);
```

Beim Funktionsaufruf wird so durch den Adreßoperator „`&`“ deutlich, wenn die Variable durch die Funktion verändert wird. Innerhalb der Funktion zeigt der Dereferenzierungsoperator „`*`“ an, daß nicht eine lokale Variable beschrieben wird.

4. Deklarationen und Datentypen

Regel 4.1. *Alle Variablen/Parameter, die nicht geändert werden, mit `constexpr` bzw. `const` kennzeichnen*

Der Compiler kann so überprüfen, ob eine eigentlich als konstant geltende Variable versehentlich doch beschrieben wird. Das gilt auch für Funktionsparameter und erst recht für Zeiger.

Bei der Programmierarbeit genügt so ein einfacher Blick auf die Variablendeklaration bzw. zum Funktionskopf, um zu wissen, daß der Wert tatsächlich unverändert ist. Natürlich soll diese Regel die Verwendung von veränderlichen Variablen keinesfalls einschränken, nur sollen konstante kenntlich gemacht werden.

Weiterhin können so markierte Ausdrücke auch an Stellen verwendet werden, die Konstanten erfordern. Und der Compiler kann teilweise auch besser optimieren.

Regel 4.2. *Ein `const` darf nur in Ausnahmefällen mittels Cast entfernt werden. Es ist zu kommentieren, warum.*

Weg-casten von `const` stellt immer ein unsauberes Programmieren dar. Wenn aber eine Bibliothek die Verwendung von `const` nicht konsequent verfolgt, hat man manchmal keine andere Wahl (siehe Regel 5.1).

Regel 4.3. *Definitionen, die nicht exportiert werden, mit `static` kennzeichnen*

Das gilt für Variablen und Funktionen. So werden zufällige Namenskollisionen beim Linken verhindert, und es ermöglicht dem Compiler weitergehende Optimierungen.

Regel 4.4. *Statische und globale Variablen vermeiden*

Meistens sind globale Variablen ein Indiz für eine schlechte Software-Architektur.

Statische Variablen in Funktionen und Klassen sind problematisch, wenn Nebenläufigkeit vorliegt bzw. ein Programm (nachträglich) parallelisiert wird oder auch nur mehrere Instanzen eines Objekts angelegt werden sollen. Lediglich für konstante Variablen oder spezielle Aufgaben sind sie sinnvoll. Außerdem schadet eine starke Verteilung der genutzten Speicherbereiche der Effizienz des Caches (lokale Variablen im Stack liegen an anderen Speicheradressen als statische Variablen).

Eine Ausnahme bilden Konstanten, die als `static const` deklariert werden sollten. Sie können auch lokal definiert werden, ohne bei Nebenläufigkeiten Schwierigkeiten zu erzeugen. `static` ist hier logisch gesehen nicht unbedingt erforderlich, ist aber eine Hilfe für den Compiler. Für Listeninitialisierungen von lokalen Konstanten ist `static` wichtig, da sie so nur einmal zu Programmbeginn initialisiert werden.

Regel 4.5. *Nichts im `namespace std` definieren*

Laut Standard ist es nicht erlaubt, selbst irgend etwas im `namespace std` zu definieren.

Regel 4.6. *Implizite Typumwandlungen vermeiden*

Die Regeln für implizite Typumwandlungen bei Rechnungen mit Variablen unterschiedlicher Datentypen sind zwar definiert (siehe Abschnitt D.1), aber explizite Umwandlungen sind offensichtlicher und besser lesbar. Insbesondere die Bevorzugung von vorzeichenlosen Typen bei der impliziten Konvertierung sind oft unerwünscht. Auch wird nie mit kleineren Datentypen als `int` gerechnet. Auch aus Effizienzgründen kann es sinnvoll sein, gezielt die Typen auszuwählen.

Compiler geben teilweise Warnungen aus, wenn die Konvertierung nicht explizit durchgeführt wird.

Ab C++11 können implizite Parameterumwandlungen gezielt unterbunden werden, indem die entsprechende Funktion explizit verboten werden:

```
void f(double d);
void f(int) = delete;  // kein Aufruf von f() mit int
template<class T> void f(T) = delete;  // alles ausser double verboten
```

Bei Zuweisungen möglichst nicht explizit die `double`-Zahl `0.0` verwenden, sondern `0`. Ggf. wandelt der Compiler das automatisch verlustlos in eine Fließkommazahl um. Falls sich der Variablentyp mal ändert, bleibt so keine versteckte Fließkommazahl stehen. Das gilt insbesondere in `templates`.

Regel 4.7. *Fließkommatypen nur über Funktionsaufrufe in Ganzzahltypen konvertieren*

Bei Verwendung von z. B. `floor` oder `ceil` wird deutlich, wie gerundet wird. Das ist besser lesbar.

Regel 4.8. *In C++ nicht die alten Typumwandlungen von C verwenden, sondern die neuen `dynamic_cast`, `static_cast`, `reinterpret_cast`, `const_cast`*

Die alten C-Casts sind unübersichtlich und können unbeabsichtigt auch `const` vom Typ entfernen.

Normalerweise sind `dynamic_cast` (für Polymorphie-Umwandlungen) und `static_cast` zu verwenden. Für Standardtypen oder wenn es aus Polymorphie-Sicht garantiert funktioniert, ist `static_cast` die richtige Wahl (also fast immer). `dynamic_cast` braucht man nur selten, und dann sollte man einen guten Grund dafür haben.

`reinterpret_cast` und `const_cast` sind nur in Ausnahmefällen zu verwenden. Inkompatible Typen (z. B. `*struct` ↔ `*array` oder `int` ↔ Zeiger) sollten nicht ineinander konvertiert werden. Wenn es ausnahmsweise unumgänglich ist, einen Zeiger in eine Ganzzahlvariable zu konvertieren, so ist hierfür der vordefinierte Typ `intptr_t` zu verwenden, in den Zeiger garantiert passen.

Eigentlich inkompatible Zeiger können notfalls mit `reinterpret_cast` konvertiert werden. `nullptr` wird aber mit `static_cast` in einen typisierten Zeiger konvertiert.

Grundsätzlich sollten möglichst wenig Typumwandlungen verwendet werden, wenn dann aber explizit.

Die funktionale Schreibweise sollte zum Casten nicht verwendet werden, sondern nur als Konstruktoraufruf, auch wenn es technisch dasselbe ist.

Regel 4.9. *Verschiedene Zeiger nicht ineinander umwandeln*

Verschiedene Ganzzahldatentypen können problemlos mit `static_cast` ineinander umgewandelt werden, sofern der Zahlenwert in den Zieltyp paßt.

Zeiger auf verschiedene Ganzzahldatentypen sind nicht kompatibel, da die Datentypen unterschiedliche Größen aufweisen können (deshalb wäre ein `reinterpret_cast` erforderlich). Es muß eine Zwischenvariable verwendet werden:

```
void f(long* const l);
void g(short* const s) {
  // Richtig:
  long l;
  f(&l);
  *s = static_cast<short>(l);
  // Falsch:
  f(reinterpret_cast<long*>(s));
}
```

Das gleiche gilt auch für `float` und `double`.

Regel 4.10. *Keine eigenen Typbezeichner einführen für Typen, die im Standard bereits existieren*

Es ist sinnlos, z. B. für `uint32_t` einen eigenen Bezeichner einzuführen. Dieser und weitere sind seit C99 bereits vorhanden. Es ist zu empfehlen, bei Compilern, denen diese Typen fehlen, eine passende `stdint.h` mit genau diesen Bezeichnern selbst zu erstellen.

Bei Verwendung dieser Datentypen sollten auch Zahlen wie 1 000 000 mit Hilfe der passenden Makros angegeben werden. Bei Schreibweisen wie `1000000UL` ist nicht sichergestellt, ob die Zahl tatsächlich als `unsigned long` darstellbar ist, da die Größe von `long` nicht definiert ist. Soll es sich um eine 32-Bit-Zahl handeln, ist die richtige Schreibweise `UINT32_C(1000000)`. Andernfalls wird der Datentyp anhand einiger unsichtbarer impliziter Regeln ermittelt (siehe Abschnitt D.2).

Wenn man für spätere Änderungen einen eigenen Datentyp vorhalten will, so hat er einen passenden Namen zu tragen und nicht die Bit-Breite im Namen, sondern die Bedeutung bzw. den Einsatzzweck.

Häufig wird auch `bool` selbst neu definiert. In C++ und seit C99 auch in C gibt es den vordefinierten Typ `bool`. Ein eigenes Definieren kann dennoch sinnvoll sein, wenn z. B. Objekte verschiedener Compiler gelinkt werden. Selbst der C- und C++-Compiler desselben Herstellers können unterschiedliche Bit-Breiten für `bool` verwenden. Um das zu umgehen, kann eine eigene Definition helfen. Völlig sinnlos ist eine eigene Definition von `true` und `false`! Dafür sollten niemals andere Werte als `1` und `0` vergeben werden.

Regel 4.11. *Oktalzahlen nicht verwenden*

Alle Zahlen mit einer führenden `0` sind Oktalzahlen. Die Verwendung ist irreführend und unüblich. Offensichtliche Ausnahme ist nur die `0` selbst.

```
int a = 001;  // schlecht: oktal 1  = dezimal 1
int b = 010;  // schlecht: oktal 10 = dezimal 8
int c = 100;  // dezimal 100
int d = 0;    // oktal 0 = dezimal 0
```

Regel 4.12. *In Deklarationen das spezifizieren, was tatsächlich erforderlich ist*

Meistens ist es nicht erforderlich, eine genaue Größe von Variablen zu definieren, sondern es geht um eine Mindestgröße. Wenn der Speicherverbrauch keine große Rolle spielt (es sich also nicht um ein riesiges Array handelt), ist eigentlich der schnellste Datentyp gefragt, der die notwendige Mindestgröße bietet.

All das ist seit C99 möglich (wenn auch leider nur für vorgegebene Bit-Breiten). Für einen Ganzzahldatentyp, der z. B. mindestens $\pm 1\,000\,000$ abbilden können soll, sollten Deklarationen also die folgende Rangfolge berücksichtigen:

`int_fast32_t`	Der schnellste Datentyp, der mindestens 32 Bit aufweist.
`int_least32_t`	Der kleinste Datentyp, der mindestens 32 Bit aufweist.
`int32_t`	Genau 32 Bit
`int`	Ein Datentyp, der *vielleicht* 32 Bit aufweist.

Für Zahlen, die immer nur positiv sind, sollten vorzeichenlose Datentypen verwendet werden (auch wegen der Regeln 2.4 und 2.5). Aber Vorsicht, wenn `for`-Schleifen abwärts in Richtung zur 0 laufen.

Regel 4.13. *Ganzzahldatentypen können überlaufen.*

Auch wenn zum Programmierzeitpunkt vielleicht nur ein Bruchteil des Zahlenbereichs einer Variable genutzt wird, kann das in ein paar Jahren ganz anders aussehen. Dann muß die Software mindestens passende Fehlermeldungen ausgeben oder besser gleich beim Kompilieren scheitern. Oder sie ist darauf vorbereitet und kann korrekt mit der Situation umgehen.

Regel 4.14. *Fließkommazahlen können überlaufen.*

Ein richtiger Überlauf ist wohl eher selten. Auf jeden Fall wird die Auflösung bei großen Zahlen aber immer schlechter. Bei sehr großen Werten bewirkt eine Addition von kleinen Zahlen überhaupt keine Änderung mehr.

Ein typischer Fehler ist die Verwendung von Fließkommatypen für Zeitstempel. Deren Auflösung wird allmählich immer schlechter. Auf Microcontrollern gibt es oft nur 32-Bit-`float`. Deren Auflösung ist schon nach $2\,\mathrm{h} + 20\,\mathrm{min}$ schlechter als 1 ms. Es sollten immer vorzeichenlose Ganzzahltypen benutzt werden, die eine konstante Auflösung bieten und mit Überläufen korrekt umgehen können.

Zeitdifferenzen benötigen hingegen einen vorzeichenbehafteten Ganzzahltyp. Es bietet sich an, dafür 2 Klassen zu implementieren, die nur die sinnvollen Operatoren definieren. Zeitdifferenzen und Zeitstempel sollten nicht vermischt werden und verschiedene Datentypen benutzen (vergl. Zeiger und deren Differenzen in Regel 3.12).

Regel 4.15. *Fließkommazahlen nicht auf Gleichheit überprüfen*

Wegen der unumgänglichen Rundungsfehler ist es sehr unwahrscheinlich, daß zwei Variablen identischen Inhalt haben. Vergleiche sind deshalb immer auf ein geeignetes Intervall auszuführen. Ausnahme bildet nur eine explizit zugewiesene 0, da diese als einzige Zahl garantiert exakt darstellbar ist.

Regel 4.16. *Fließkommagenauigkeit bei Rechnungen ist undefiniert.*

Es gibt keinerlei im Standard zugesicherte Rechengenauigkeit bei Fließkommazahlen. Dadurch sind Korrektheitsbeweise bei Fließkommazahlen prinzipiell nicht möglich.

Regel 4.17. *`float` nur in begründeten Fällen verwenden*

Bei aktuellen Implementierungen ist die Auflösung der Mantisse mit 24 Bit (inklusive Vorzeichen) relativ klein. `float` ist auf PCs fast nie schneller als `double` (53 Bit Mantisse). Der Speicherverbrauch spielt nur eine relevante Rolle, wenn sehr viele Zahlen gespeichert werden müssen.

Laut Standard sind nur folgende Mindestauflösungen definiert, alles andere ist plattformspezifisch:

`FLT_DIG` ≥ 6	`[L]DBL_DIG` ≥ 10
`FLT_EPSILON` $\leq 10^{-5}$	`[L]DBL_EPSILON` $\leq 10^{-9}$
`{FLT|[L]DBL}_MIN` $\leq 10^{-37}$	
`{FLT|[L]DBL}_MAX` $\geq 10^{+37}$	

Rechnungen werden oft auch für `float`-Rechnungen in `[long] double` ausgeführt. Um plattformunabhängig den effizientesten Typ mit einer vorgegebenen Mindestgröße zu wählen, gibt es die Typen `float_t` und `double_t` aus `math.h`. Deren Verwendung erhöht auch die Wahrscheinlichkeit von Berechnungen beim Kompilieren.

Wenn der Compiler automatisch vektorisieren soll, sind kleine Datentypen zu bevorzugen (also doch auch `float` statt `double`), damit mehr in einen Vektor passen. Bei Ganzzahltypen ist das ebenfalls zu beachten, so daß je nach Kontext unterschiedliche Typen gewählt werden sollten. Das ist aber sehr stark Plattform- und Compiler-abhängig und muß im Einzelfall getestet werden. Eine automatische Vektorisierung erfordert in den meisten Fällen eine Compiler-Option wie „`-ffast-math`“ beim `gcc`.

Regel 4.18. *Auf die Typkorrektheit bei Vergleichen zwischen vorzeichenbehafteten und -losen Zahlen achten*

Die notwendige Typkonvertierung ist explizit auszuführen. Negative Zahlen sind zwar in einem vorzeichenlosen Datentyp nicht darstellbar, werden aber immerhin definiert durch Umlauf behandelt. Umgekehrt ist nicht definiert, was passiert, wenn eine zu große Zahl in einen vorzeichenbehafteten Typ konvertiert wird.

Compiler geben meistens Warnungen aus, wenn die Konvertierung nicht explizit durchgeführt wird.

Regel 4.19. *Überläufe in vorzeichenlosen Ganzzahltypen sind unproblematisch, sofern immer nur Differenzen ausgewertet werden.*

Differenzen „`uint2 - uint1`“ funktionieren auch korrekt, wenn `uint2` übergelaufen ist. Das funktioniert, solange die wahre Differenz kleiner als der Wertebereich der Zahlen ist.

Bei vorzeichenbehafteten Zahlen ist das nicht garantiert (auch wenn es meistens funktioniert)! Solange die Differenz garantiert positiv ist, erreicht man das Ziel durch vorherige

Konvertierung in den korrespondierenden vorzeichenlosen Typ. Andernfalls hilft nur eine Fallunterscheidung. Bei dem folgenden Ausdruck muß zusätzlich sichergestellt sein, daß die Ergebnisse im Zieldatentyp abbildbar sind:

```
signed d = (uint2>uint1) ? +static_cast<signed>(uint2-uint1)
                         : -static_cast<signed>(uint1-uint2);
```

Regel 4.20. *Standarddatentypen durch Zuweisungen initialisieren*

Auch in C++ ist eine Zuweisung leichter lesbar als die Konstruktorschreibweise: „`int i = 5`“ statt „`int i(5)`“.

Regel 4.21. *Initialisierungslisten von Strukturen mit benamten Elementen schreiben*

Seit C99 gibt es die Möglichkeit, die Werte direkt den Elementnamen zuzuweisen. Dadurch ist die Initialisierung von der Elementreihenfolge unabhängig und damit sicherer; das geht auch in einer `union`: `struct {int x,y;} s = {.x=1, .y=2};`

Leider hat C++ das nicht übernommen.

Regel 4.22. *Als physikalische Einheiten bei Fließkommazahlen immer die Basiseinheiten des si-Systems verwenden*

Dadurch besitzt das Ergebnis automatisch ebenfalls die zugehörige Basiseineit. Fehleranfällige Einheitenumrechnungen sind damit überflüssig. Nur bei Festkommazahlen muß man sich gezwungenermaßen eine passende Skalierung überlegen.

Regel 4.23. *`enum` verwenden, und kein `#define` oder Ganzzahltypen.*

Mit `enum` wird ein eigener Typ definiert, so daß der Compiler auch Typkorrektheit prüfen kann.

Oft muß für alle Einträge eines `enum`s dieselbe Aktion erfolgen. `for`-Schleifen können eine Variable vom Typ eines `enum` nicht hochzählen. Dafür ist eine Ganzzahlvariable erforderlich, die alle Einträge aufnehmen kann. Mit einem zusätzlichen abschließenden Element in `enum`s kann man leicht die Gesamtanzahl abfragen und z. B. für `for`-Schleifen verwenden. Achtung: `sizeof` von `enum`s ist je nach Compiler und Anzahl der Elemente verschieden. Seit C++11 kann der Datentyp festgelegt werden, sofern das für eine Schnittstelle erforderlich ist (siehe Regel 2.9):

```
enum class Farben : int8_t {ROT, BLAU};
```

Weitere Definitionen können erfolgen, ohne die Liste redundant pflegen zu müssen:

```
// Die Liste aller Elemente: manuell zu pflegen
#define ENUM_LISTE \
  ENUM(EINS)  /* Nummer 1 */ \
  ENUM(ZWEI)  /* Nummer 2 */ \
  ENUM(DREI)  /* Nummer 3 */

// Die Nummern als enum: automatisch generiert
enum class Liste {
# define ENUM(a) a,
  ENUM_LISTE  // Die Liste aller Elemente im enum
  ENUM_MAX    // Elementeanzahl (z.B. als Schleifenbegrenzung)
# undef ENUM
};

// Elemente als Strings: automatisch generiert
constexpr const char* /*const*/ zustand_str[] = {
# define ENUM(a) #a,
  ENUM_LISTE
  "ENUM_MAX"
# undef ENUM
};
```

Eine weitere praktische Möglichkeit in C ist die Nutzung der `enum`-Werte beim Initialisieren von Arrays. Innerhalb der eckigen Klammern kann auch gerechnet werden. Leider ist das in C++ nicht möglich:

```
enum {ROT, BLAU};
constexpr const char* /*const*/ farbe[] = {
  [ROT]  = "rot",
  [BLAU] = "blau"
};
```

Regel 4.24. *Den Datentyp `bool` ggf. durch `enum` ersetzen*

Bei Funktionsaufrufen mit `bool`-Parametern ist deren Bedeutung nicht mehr erkennbar, wenn dort nur `true`/`false` steht. Ein `enum` mit den beiden Bedeutungen als sprechende Bezeichner ist typsicher und damit besser als einzelne konstante Variablen mit den Werten `true`/`false`.

Alternativ kann der Parametername beim Aufruf eingesetzt werden: `f(/*parname=*/true);`

Regel 4.25. *Typen für Zähler in `for`-Schleifen automatisch bestimmen*

Beim Deklarieren von Zählervariablen in `for`-Schleifen sollten Redundanzen nach Möglichkeit vermieden werden (siehe Regel 1.4). Mit den seit C++11 vorhandenen Techniken (`auto` und `decltype`) ist das oft leicht möglich.

Leider können `enum`-Typen nicht als Zähler verwendet werden, so daß ein anderer Typ verwendet werden muß, der aber unbedingt groß genug sein muß.

Varianten in absteigender Präferenz:

```
for(const auto& i : v);
for(auto i=v.begin(); i!=v.end(); ++i);
for(decltype(N) i=0; i<N; ++i);
for(auto i=N; i>=0; --i);  // Achtung bei vorzeichenlosem N!
for(std::underlying_type<T>::type i=0; i<ENUM_MAX; ++i);
for(unsigned i=0; i<ENUM_MAX; ++i) {
  static_assert(sizeof(i)>=sizeof(ENUM_MAX), "C");
  static_assert(numeric_limits<decltype(i)>::max()>=ENUM_MAX, "C++");
}
```

Regel 4.26. *Typunabhängig programmieren*

Seit C++11 kann vieles typunabhängig formuliert werden. Insbesondere in `templates` ist das hilfreich. Aber Typangaben sind oft Redundanzen, da sie zu den übrigen Variabeln passen müssen. Wenn der Typ dieser Variablen geändert wird, müssen in Rechnungen oft Zwischenvariablen ebenfalls angepaßt werden. Im besten Fall beschwert sich der Compiler. Oder er konvertiert automatisch mit unbekannten Folgen.

So hilfreich typunabhängige Programmierung mit `auto` oder `decltype` auch sein kann, so versteckt es doch die Typen und macht den Programmtext oft schwieriger lesbar. Die Abwägung ist individuell zu treffen.

Allgemein sinnvoll ist die Anwendung bei `for`-Schleifen (siehe Regel 4.25). Bei Ergebnissen von Rechnungen oder Funktionsaufrufen ist „`const auto&`“ oft die richtige Wahl, es sei denn, `auto` soll explizit lesbar mit dem jeweiligen Typ ersetzt werden.

Um mittels `decltype` sicher den Basistypen eines Ausdrucks zu extrahieren, sind weitere Aufrufe notwendig:

```
#define DECLTYPE_NOREF(a) std::remove_reference<decltype(a)>::type
#define DECLTYPE_BASE( a) std::remove_cv<typename DECLTYPE_NOREF(a)>::type
```

Regel 4.27. *Die sogenannte „Ungarische Notation“ nicht benutzen*

Bei aktuellen Editoren reicht das Plazieren des Mauszeigers auf einen Variablennamen, um den Typ angezeigt zu bekommen. Oder man kann mit einer Tastenkombination zur Deklaration springen.

Das Integrieren des Datentyps in den Variablennamen stellt eine Redundanz dar, die von Hand gepflegt werden muß und entsprechend fehleranfällig ist (siehe Regel 1.4). Bei Änderungen am Datentyp muß der Variablenname an allen Verwendungsstellen angepaßt werden. Wird das vergessen, so stimmt der Datentyp nicht mehr mit dem Namen überein.

Dieselben Aussagen gelten auch für die Kennzeichnung von Klassenvariablen durch vorangestelltes `m_`.

Regel 4.28. *Eigene Bezeichner nicht mit Unterstrich beginnen*

Bezeichner, die mit einem Unterstrich beginnen, können durch den Compiler für beliebige Dinge definiert sein. Deshalb dürfen eigene Bezeichner zur Vermeidung von Kollisionen nicht mit einem Unterstrich beginnen. In C++ sind sogar alle Bezeichner mit zwei aufeinander folgenden Unterstrichen an beliebiger Stelle im Bezeichner betroffen. Genaugenommen ist der Sachverhalt etwas komplizierter, manche Varianten mit einem Unterstrich am Anfang wären erlaubt, das sollte man aber unterlassen.

5. Funktionen und Methoden

Regel 5.1. *Methoden, die ein Objekt nicht verändern, als* `const` *bzw.* `constexpr` *kennzeichnen*

Ohne `const` kann man diese Methoden nicht für konstante Objekte aufrufen. Das seit C++11 vorhandene `constexpr` ermöglicht oder erzwingt sogar noch weitergehende Optimierungen durch den Compiler.

Regel 5.2. *Linkage klein halten*

Lokale Funktionen sollten immer `static` deklariert werden, um nicht übergreifend über mehrere Übersetzungseinheiten sichtbar zu sein (siehe Regel 4.3). Bei `template`s und `inline` Funktionen ist das nicht zwingend erforderlich, da hier mehrere identische Kopien beim Linken erlaubt sind. Es ist dennoch auch in diesen Fällen empfehlenswert (siehe auch Regel 5.3).

In C++ kann alternativ auch alles, was lokal in einer Übersetzungseinheit bleiben soll, in einem anonymen `namespace` definiert werden. Es gibt aber einen Fall, in dem das nicht weiterhilft: Wenn eine deklarierte Klasse als Typ zwischen mehreren Übersetzungseinheiten ausgetauscht wird, muß sie außerhalb des anonymen `namespace` definiert sein.

`template`-Methoden werden üblicherweise auch in der `h`-Datei definiert und sind so in mehreren Übersetzungseinheiten vorhanden. Wenn in einzelnen expliziten Spezialisierungen von `template`s keine undefinierten `template`-Parameter übrig sind, sind aber keine redundanten Definitionen mehr erlaubt. Weil es Methoden sind, kann das Schlüsselwort `static` nicht wie bei Funktionen eingesetzt werden. Wenn der anonyme `namespace` auch nicht eingesetzt werden kann, müssen die Spezialisierungen entweder in einer `cpp`-Datei definiert werden. Oder, wenn sie in der `h`-Datei bleiben sollen, deklariert man sie explizit als `inline`.

Das Schlüsselwort `inline` wird so zwar zweckentfremdet und die Linkage bleibt unverändert, aber es sind wieder redundante Definitionen in mehreren Übersetzungseinheiten erlaubt. Das `inline` sollte bei der betreffenden Definition stehen, da es nicht zur rein logischen Deklaration gehört und den Benutzer der Klasse eigentlich nicht zu interessieren hat:

```
struct Klasse {
  template<class T>  Klasse f(T t);
};
template<class T> Klasse Klasse::f(T t) {...}
template<> inline Klasse Klasse::f<int>(int t) {...}
```

Regel 5.3. *`inline` normalerweise nur in Verbindung mit `static` verwenden*

Methoden, die direkt in der Klassendeklaration definiert werden, sind automatisch implizit `inline`. Der Compiler kann sich an den Programmiererwunsch `inline` halten oder auch nicht. Oder er kann etwas `inline` machen, ohne daß es so definiert war. Die explizite Verwendung von `inline` ist also oft wirkungslos und kann daher häufig besser weggelassen werden.

Verwendet man „`inline void f() {...}`“ ohne `static`, so muß man eine entsprechende Implementierung hinzulinken. Oder man läßt automatisch eine externe Funktion erzeugen, indem man eine zusätzliche Zeile ohne neue Definition hinzufügt: „`extern void f();`“

Regel 5.4. *Überladung von Funktionen sparsam einsetzen, das gilt auch für Vorgabeargumente*

Verschiedene Funktionen mit demselben Namen können leicht zu Verwechslungen führen. Deshalb dürfen Funktionen nur gleich heißen, wenn sie tatsächlich die gleiche Aufgabe haben.

Zu beachten ist, daß Methoden in abgeleiteten Klassen alle gleichnamigen der Basisklasse verdecken. Diese müssen ggf. mit `using` (alle) wieder sichtbar gemacht werden.

Regel 5.5. *`templates` sparsam einsetzen*

`templates` können sinnvoll sein, machen aber auch vieles sehr unübersichtlich. Insbesondere die Template-Meta-Programmierung ist daher nur in begründeten Fällen anzuwenden.

Regel 5.6. *Abgeleitete virtuelle Methoden als `virtual` kennzeichnen und `override`/`final` benutzen*

Das Wiederholen des Schlüsselwortes `virtual` für virtuelle Methoden in abgeleiteten Klassen ist im Standard nicht vorgeschrieben. Um diese Eigenschaft deutlich zu machen, ohne daß man bis zur Basisklasse recherchieren muß, ist das Schlüsselwort `virtual` in allen abgeleiteten Klassen zu wiederholen.

Seit C++11 sollten alle Methoden, die eine Basisklassenmethode überschreiben, immer mit `override` markiert werden. Das Überschreiben von Methoden kann in der Basisklasse mit `final` verboten werden. Soll von einer Klasse gar nicht abgeleitet werden, so wird das in der Klassendeklaration mit `final` sichergestellt (siehe Abschnitt A.1 für ein Beispiel).

Virtuelle Methoden kosten etwas mehr Laufzeit. Polymorphismus also bei laufzeitkritischen Stellen möglichst vermeiden.

Regel 5.7. *Vererbung nur bei Vererbung*

Eine Vererbung nur verwenden, wenn es sich tatsächlich um eine solche Beziehung handelt, die abgeleitete Klasse also auch als Basisklasse aufgefaßt und als solche verwendet werden kann.

Daraus folgt, daß Vererbungen eigentlich immer `public` erfolgen sollten.

Mehrfachvererbung ist meistens unübersichtlich und spricht für ein Architekturproblem. Die Laufzeit wird ebenfalls etwas beeinträchtigt, also nur benutzen, wenn es sein muß.

Regel 5.8. *Basisklassen müssen einen virtuellen Destruktor besitzen.*

Andernfalls wird der Destruktor der abgeleiteten Klassen möglicherweise nicht aufgerufen. Nur in `final`-Klassen muß der Destruktor nicht virtuell sein (siehe Abschnitt A.1 für ein Beispiel).

Regel 5.9. *Vom Objekt unabhängige Methoden als `static` kennzeichnen*

Das hilft dem Verständnis des Lesers und ermöglicht dem Compiler weitergehende Optimierungen (erspart auch den impliziten Parameter, den `this`-Zeiger).

Regel 5.10. *Konstruktoren als `explicit` kennzeichnen*

Andernfalls verwendet der Compiler Konstruktoren, die mit nur einem Parameter aufgerufen werden können, möglicherweise automatisch zur Typkonvertierung. Nur wenn das ausdrücklich gewollt ist, darf das `explicit` entfallen (siehe Abschnitt A.1 für ein Beispiel). Das ist entsprechend zu kommentieren.

Auch Konstruktoren mit einer anderen Anzahl an Parametern sollten als `explicit` gekennzeichnet werden. Dann kann es nicht vergessen werden, wenn die Parameteranzahl nachträglich geändert wird.

Seit C++11 gilt das gleiche auch für Typkonvertierungsoperatoren.

Regel 5.11. *Konstruktoren dürfen keine virtuellen Methoden aufrufen*

Die V-Tabelle ist noch nicht vollständig initialisiert. Es ist daher nicht klar, welche Methode tatsächlich aufgerufen wird.

Regel 5.12. *Implizite Klassenmethoden nicht zulassen*

Standardkonstruktor, Kopierkonstruktor und Zuweisungsoperator sind als `private` zu deklarieren und *nicht* zu implementieren, wenn sie nicht genutzt werden sollen/dürfen. Dadurch werden die automatisch generierten Methoden unterdrückt und sind nicht mehr verfügbar, so daß der Compiler eine versuchte Nutzung mit einer Fehlermeldung verweigert.

Seit C++11 gibt es das explizite Löschen von Methoden, was der `private`-Deklaration vorzuziehen ist.

Siehe Abschnitt A.1 für Klassenvorlagen, die die zuvor beschriebenen Dinge berücksichtigen. Ggf. sind einzelne Zeilen an die jeweilige Situation anzupassen, aber die Vorlage zwingt zum bewußten Ändern und vermeidet, etwas zu vergessen.

Regel 5.13. *Zuweisungsoperatoren einschränken*

Eine Zuweisung an ein r-value (also ein temporäres Objekt) ist normalerweise nicht gewünscht. Seit C++11 kann das vom Compiler überwacht werden, indem der `this`-Zeiger (mittels `&`-Zeichen) nur noch l-values akzeptiert.

Außerdem sollten Zuweisungsoperatoren nur `const`-Referenzen zurückgeben. Eine Veränderung eines Zuweisungsergebnisses in demselben Ausdruck ist nicht erforderlich und eher ein Programmierfehler als beabsichtigt:

```
class K {
  const K& operator= (const K&) &;
  const K& operator+=(const K&) &;
};
```

Regel 5.14. *Beseitigung von Warnungen zu unbenutzten Parametern*

Parameter können z. B. bei vorgegebenen Signaturen für Callbacks unbenutzt bleiben. Soll die zugehörige Compiler-Warnung nicht global deaktiviert werden, muß jeder unbenutzte Parameter einzeln behandelt werden.

Eine Zuweisung an sich selbst wird teilweise vom Compiler wegoptimiert und verbietet sich, wenn der Parameter als `const` definiert ist.

Ein Weglassen des Parameternamens ist unschön und unpraktisch, wenn die Verwendung z. B. von `#ifdef`s abhängt.

Ein einfaches Casten des Parameters nach `void` mögen manche Compiler nicht für unvollständige Typen (Vordeklarationen `class K;`).

Folgendes Makro löst das Problem:

```
#define UNBENUTZT(a) ((void)(&reinterpret_cast<const int&>(a))) //C++
#define UNBENUTZT(a) ((void)(a)) //C
void f(x) {UNBENUTZT(x);}
```

Regel 5.15. *Captures bei Lambda-Funktionen nur explizit*

Lambda-Funktionen sind zunächst nur eine kompakte Schreibweise für anonyme Funktionen. Mit der Capture-Technik bieten sie aber auch eine neue Art der Parameterübergabe. Diese sollte für die Übersichtlichkeit immer explizit erfolgen (also nicht einfach alles per `[&]` oder `[=]`).

Regel 5.16. *Rückgabetypen von Funktionen in alter Schreibweise*

Für eine einheitliche und damit übersichtliche Schreibweise soll immer die alte Schreibweise genutzt werden. Die neue ist nur erlaubt, wenn es sein muß.

```
int  f(int);          // alt
auto f(int) -> int;   // neu
```

Regel 5.17. *Exceptions vermeiden, notfalls bewußt nutzen und behandeln*

Versteckt auftretende Exceptions führen häufig zum überraschenden Programmende. Sie müssen *immer* behandelt werden.

An einer Funktionsdeklaration ist vor C++11 nicht direkt ersichtlich, ob sie Exceptions erzeugen kann. Exceptions bilden einen versteckten zusätzlichen Kontrollfluß. Es gibt die Möglichkeit, mit `throw(...)` am Ende des Funktionskopfes zu spezifizieren, welche Exceptions auftreten können. Das kann dann auch der Compiler überwachen. Das ist guter Programmierstiel aber leider nicht üblich und seit C++11 leider sogar abgeschafft. Compiler erzeugen bei Verwendung dieser Deklaration auch gern zusätzliche Abfragen, so daß die Gesamteffizienz leidet.

Seit C++11 kann immerhin mit `noexcept` deklariert werden, daß eine Funktion keine Exception wirft. Das sollte unbedingt genutzt werden. Besonders für Move-Konstruktoren und -Zuweisungsoperatoren ist das wichtig.

Bei Berücksichtigung von Regel 1.2 werden bei Exceptions die traditionellen `if`-Blöcke lediglich durch `try-catch`-Blöcke ersetzt. Der Programmtext vereinfacht sich nicht.

Die Behandlung von Exceptions (auch wenn sie nicht auftreten) kostet ganz allgemein Laufzeit. Das kann man sparen, indem man die Exception-Behandlung im Compiler komplett deaktiviert, wenn man sie nicht braucht.

Exceptions bieten nur selten echte Vorteile.

Nur bei fehlschlagenden Konstruktoren sind Exceptions eine Alternative. Oder man untergräbt die schöne Eigenschaft, nach dem Konstruktor immer ein fertig initialisiertes Objekt zu erhalten, und verlagert die Initialisierung in eine `Init`-Methode. Beides hat Nachteile.

6. Sicherheitsregeln

Regel 6.1. *Unsichere String-Operationen (wie `sprintf`, `strcat`, `strcpy`) vermeiden*

Diese Operationen dürfen nur verwendet werden, wenn sicher ist, daß das Ergebnis in den Zielspeicher paßt (unbedingt kommentieren). Stattdessen `snprintf`, `strncat`, `strncpy` verwenden. Achtung: Das abschließende `'\0'` wird unterschiedlich behandelt:

```
snprintf(s, sizeof(s), q);  // strlen(s) <= sizeof(s)-1
strncat (s, q, sizeof(s)-strlen(s)-1);
strncpy (s, q, sizeof(s));  // '\0' fehlt moeglicherweise
s[sizeof(s)-1] = '\0';      // '\0' nach strncpy manuell
```

Von der Verwendung der modifizierten nicht standardkonformen Versionen von Microsoft sollte man absehen. Die sind entbehrbar und man bindet sich unnötig an eine proprietäre Insellösung.

Regel 6.2. *In `printf/scanf` müssen die Formate (die Buchstaben nach dem „`%`“) mit den Datentypen der Parameter übereinstimmen.*

`gcc` kann das beim Kompilieren prüfen, es ist also eine prüfbare Redundanz. Auch für Zeiger und spezielle Typen wie `size_t` gibt es eigene Formatbuchstaben. Für die Typen aus `stdint.h` gibt es ebenfalls vordefinierte Formate `PRIu32`, `SCNu32` usw. Niemals nur zufällig übereinstimmende Formate verwenden.

Regel 6.3. *Formatierte Textausgabe lieber mit `printf` statt mit Streams*

Streams haben zwar den Vorteil, daß man mittels `<<` beliebige Typen als Text ausgeben kann und keine Übereinstimmung zwischen Formatbuchstabe und Variablentyp beachtet werden muß.

Aber sie sind unübersichtlich, insbesondere wenn der Text ordentlich formatiert werden soll. Ggf. vorgenommene Formatierungseinstellungen bleiben für den Stream erhalten, so daß an ganz anderen Stellen im Programm diese veränderten Einstellungen aktiv sind, ohne es zu wissen. Eigentlich muß also vor jeder Ausgabe die Formatierung gesetzt werden:

```
printf("%-␣10x\n", zahl);
printf("%+-␣10.3f\n", fzahl);
// Identische Ausgaben mit Streams:
std::cout << std::left << std::setfill('␣') << std::setw(10)
          << std::setbase(16) << zahl << std::endl;
std::cout << std::showpos << std::left << std::setfill('␣')
          << std::setw(10) << std::setiosflags(std::ios::fixed)
          << std::setprecision(3) << fzahl << std::endl;
```

Außerdem arbeiten sie viel mit dynamischem Speicher, was sich mit `snprintf` umgehen läßt (siehe Regel 7.4).

Regel 6.4. *Zeichenketten mit `strtol` konvertieren statt mit `atol`*

Die Funktionen `atol`, `atof` usw. bieten keinen Mechanismus, um Fehler zu erkennen.

Regel 6.5. *Wird in einem `switch`-`case`-Block ein `break` bewußt weggelassen, so ist das durch einen entsprechenden Kommentar an dieser Stelle zu dokumentieren.*

Das gilt nur, wenn zwischen den `case`-Marken auch Anweisungen stehen.

Regel 6.6. *Bei Verwendung von mehreren Operatoren in einem Ausdruck sind sie zu klammern.*

Die Ausführungsreihenfolge innerhalb von Ausdrücken ist zwar eindeutig definiert (siehe Abschnitt D.3). Durch Klammerung wird die Ausführungsreihenfolge offensichtlich, auch ohne die genauen Regeln auswendig zu kennen.

Regel 6.7. *Vor mathematischen Operationen Parameter auf Gültigkeit prüfen*

Häufigster Fall ist die Division durch Null, weitere typische Problemfälle sind `/`, `%`, `sqrt`, `log`, `asin` usw. Undefinierte Resultate müssen verhindert werden, bevor die Rechnung ausgeführt wird.

Bei der Division von Fließkommazahlen genügt nicht die Prüfung auf Null, da sie auch überlaufen können, wenn der Zähler groß und der Nenner klein ist. Außerdem ist auf die speziellen denormalisierten Werte zu achten. Eine vollständige Abfrage läßt sich für `double` so machen:

```
double nenner, zaehler;
if(nenner!=0 && fabs(zaehler)/DBL_MAX<=fabs(nenner) &&
   (fabs(nenner)>DBL_MIN || fabs(zaehler)<=fabs(nenner)*DBL_MAX))
{
  double bruch = zaehler / nenner;
} else {
  fehler();
}
```

Regel 6.8. *`atan2` den anderen Winkelfunktionen `asin`, `acos` und `atan` vorziehen*

`atan2` liefert eine bessere Winkelauflösung mit weniger Rundungsfehlern und ist für den Vollkreis definiert. Nur wenn sich das Argument immer sicher im „guten“ Bereich von `asin`, `acos` oder `atan` befinden wird, kann auf `atan2` verzichtet werden.

Regel 6.9. *Explizite Zahlen dürfen nicht direkt im Programmtext vorkommen.*

Konstante werden als konstante Variablen definiert. Ausnahmen bilden die neutralen Elemente der Grundrechenarten sowie mathematische Gleichungen, die bestimmte Zahlen enthalten (z. B. $s = \frac{1}{2}at^2$).

7. Effizienz

Regel 7.1. *Große Objekte nicht über den Stack übergeben*

Sollen große Objekte (also Strukturen bzw. Klassen) als Parameter an eine Funktion übergeben werden, so wird das aus Effizienzgründen mittels Referenz oder Zeiger erledigt, damit die Daten nicht in den Stack kopiert werden müssen. Weiterhin hat der Stack eine beschränkte Größe, bei Windows standardmäßig 1 MB.

Genauso soll ein großes Objekt nicht direkt per `return` zurückgegeben werden, sondern besser über einen Zeiger, der als Parameter übergeben wird. Wenn eine Funktion doch ein Objekt per `return` zurückgibt, so sollte damit eine Referenz initialisiert werden. Dadurch wird der Aufruf des Kopierkonstruktors vermieden. Die Definition dieser Referenz verlängert die Lebensdauer des Rückgabewerts über das Semikolon hinaus für die Lebensdauer der Referenz: `const objekttyp& objekt = Funktion();`

Vorsicht mit dieser Technik, wenn die Funktion ihrerseits eine Referenz und keinen Wert zurückgibt!

Viele Compiler erledigen je nach Kontext einige dieser Optimierungen auch automatisch.

Regel 7.2. *Zum Kopieren von Bereichen vorhandene Funktionen nutzen*

Für das kopieren von Speicher gibt es `memcpy`. Eigene `for`-Schleifen werden nicht effizienter aber unübersichtlicher sein. Das gleiche gilt für `memset`.

Strukturen können auch direkt einander zugewiesen werden, das bewirkt falls möglich automatisch ein `memcpy`, ist aber übersichtlicher.

Regel 7.3. *Bei der Benutzung von Funktionen/Operatoren den internen Aufwand bedenken*

Beim Programmieren geht leicht die Übersicht verloren, wie aufgerufene Funktionen intern arbeiten. Ein unscheinbarer Aufruf kann viel Laufzeit kosten. C++ ist in der Laufzeitbibliothek häufig auf dynamischen Speicher angewiesen, dessen Verwaltung vergleichsweise viel Zeit kosten kann. Dieser interne Aufwand ist dem Programmtext bei der Verwendung oft nicht mehr direkt anzusehen. C++ ist gut geeignet, die wahre Komplexität von Operationen gut zu verbergen.

Insbesondere auf die interne Behandlung von Parametern und möglicherweise temporäre Zwischenergebnisse sollte geachtet werden.

Regel 7.4. *Speicherverwaltung kostet Zeit.*

Dynamische Speicherverwaltung ist immer aufwendiger als Variablen auf dem Stack oder statisch anzulegen. Wenn es die Größe von Objekten/Arrays zuläßt (oder sie statisch angelegt werden können), dann sollte auf `new`/`malloc` verzichtet werden. Dann weiß der Compiler auch mehr über den Speicherbereich (z. B. Ausrichtung) und kann besser optimieren.

In zeitkritischen Anwendungen (z. B. Echtzeit) kann dynamische Speicherverwaltung ggf. ungeeignet sein. Das gilt dann auch für alle Bibliotheksfunktionen, die darauf beruhen (STL usw.).

Regel 7.5. *Teure Rechenoperationen sparsam einsetzen, nach Möglichkeit aus Schleifen herausziehen oder reduzieren*

Typische Rechnungen dieser Kategorie sind `sqrt`, `exp`, `log`, `pow`, „`/`“, „`%`“. Oft kann man die Rechnungen umstellen oder in Schleifen durch Additionen o. ä. ersetzen.

Letzteres können einige Compiler automatisch für Ganzzahlvariablen oder Array-Zugriffe (normalerweise ist eine Multiplikation von Index und Elementgröße erforderlich). Für Fließkommarechnungen muß das meistens von Hand erfolgen. Das Verfahren kann für beliebige polynomiale Berechnungen aus der Schleifenvariablen immer alle Multiplikationen eliminieren und in Additionen verwandeln. Es ist immer dann sinnvoll, wenn der

neue Wert leichter aus dem vorhergehenden berechnet werden kann als aus der Schleifenvariablen. Wie weit man dabei gehen sollte, bis es wieder negative Laufzeiteffekte hat, ist vom Compiler und Prozessor abhängig. Auch Rückgriffe auf noch ältere Werte können sinnvoll sein.

Auf üblichen PCs ist eine `if`-Abfrage für Umlauferkennungen übrigens schneller als „`%`“.

Regel 7.6. *Algebraische Optimierungen nutzen*

Algebraische Optimierungen machen Compiler teilweise automatisch. Bei `bool` funktioniert das relativ gut, bei Ganzzahlen nur bei einfachen Ausdrücken, und bei Fließkomma kaum, weil durch eine andere Ausführungsreihenfolge der Berechnungen andere Rundungsfehler entstehen würden, was nicht erlaubt ist (Compiler bieten dafür Optionen an, beim `gcc` heißt sie „`-ffast-math`“). Seit C++11 ermöglicht `constexpr` weitere Optimierungen zur Kompilierzeit.

Fließkommadivisionen mit einer Konstante können von Hand in Multiplikationen mit dem Kehrwert konvertiert werden, nicht aber automatisch vom Compiler (`a/5.0 != a*0.2`).

Bei mehreren Divisionen in einer Formel können sie häufig zu einer zusammengefaßt werden, z. B. durch Bildung des Hauptnenners. Das funktioniert auch über Formelgrenzen hinweg, indem alle mit dem Kehrwert des „Hauptnenners“ und den Nennern der anderen multipliziert werden. Bei Vergleichen kann der Nenner durch Multiplikation auf die andere Seite gebracht werden.

Ausklammern von Ausdrücken oder Ablegen von Zwischenergebnissen in Variablen zur mehrfachen Verwendung kann ebenfalls helfen.

Bei allen Umformungen ist aber ggf. schlechteres numerisches Verhalten zu beachten (Rundungsfehler und Fehlerfortpflanzung).

Regel 7.7. *Rechnungen geeignet klammern*

Laut C-Standard werden mathematische Ausdrücke von links nach rechts ausgewertet. Im Ausdruck „`x + 5 + 6`“ kann der Compiler daher nichts optimieren, wenn `x` eine Fließkommavariable ist. Besser sind daher die beiden Schreibweisen „`x + (5 + 6)`“ und „`5 + 6 + x`“, die zur Kompilierzeit optimiert werden können: „`x + 11`“ bzw. „`11 + x`“

Dieser Sachverhalt tritt bei Verwendung von `#define` oft versteckt auf.

Neben der Zusammenfassung von Konstanten zur Kompilierzeit sind Formeln auch so zu klammern, daß möglichst viele Unterausdrücke unabhängig parallel berechnet werden können.

Regel 7.8. *Mehrfach benötigte Zwischenergebnisse speichern*

Werden die Ergebnisse von teuren Berechnungen mehrfach benötigt, sollten die in Zwischenvariablen aufbewahrt werden und nicht mehrfach berechnet werden. Das können einfache Winkelfunktionen oder auch komplexe Algorithmen sein. Teilweise machen Compiler das auch automatisch, wenn sie erkennen, daß die Ergebnisse dieselben sind und die Funktion keine anderen Seiteneffekte haben.

Die Datentypen von Zwischenvariablen sind insbesondere in Makros teilweise nicht vorhersehbar. Seit C++11 bieten sich dafür die neuen Konzepte `auto` und `decltype` an. Seit C11 kann auch `_Generic` eine ähnliche Aufgabe übernehmen.

Regel 7.9. *Anzahl von Vergleichen durch Umstellungen minimieren*

Ein Beispiel: „`(i>=min) || (i<=max)`“ umformen zu „`(i-min) <= (max-min)`“ (gilt für vorzeichenlose Variablen durch Ausnutzung von Überlauf)

Regel 7.10. *`bool`-Ausdrücke sinnvoll sortieren*

Operanden von „`&&`“ und „`||`“ sollten immer sinnvoll sortiert sein, je nach Häufigkeit, daß sie wahr sind, und/oder wie lange ihre Berechnung dauert, so daß der zweite nicht mehr ausgewertet werden muß.

Regel 7.11. *Moduloberechnung für Winkel einfach halten*

Oft ist es unwichtig, ob ein Winkel mehrmals umläuft. Winkelfunktionen wie `sin` funktionieren auch mit z. B. 7π. Wenn Auflösungsprobleme zu erwarten sind durch extrem häufiges Umlaufen, muß der Winkel auf z. B. $\pm 2\pi$ abgebildet werden:

```
a = fmod(a, 2 * M_PI)
```

Wenn dieser Wert unbedingt auf $\pm\pi$ eingeschränkt werden muß, anschließend:

```
a = fmod(a + 3 * M_PI, 2 * M_PI) - M_PI
```

Bei Rechnungen mit Festkommazahlen, sind 360° auf Zweierpotenzen abzubilden (und zwar mit vorzeichenlosem Ganzzahltyp). Dann genügen einfache Bit-Operationen bzw. Überläufe.

Regel 7.12. *Bei Fließkommarechnungen Ergebnisse wie `nan` vermeiden*

Auch wenn `inf` und `nan` als Ergebnisse definiert sind und teilweise sogar sinnvoll einsetzbar sind, sollte das nicht genutzt werden. Die Berechnung wird durch Sonderbehandlungen oft sehr viel langsamer. Und Optimierungen wie mit der `gcc`-Option `-ffast-math` sind nicht mehr möglich.

Regel 7.13. *Prä-/Postin- und -dekrementoperatoren gezielt auswählen*

Bei aufwendigen Klassen, die die Inkrement- oder Dekrementoperatoren überladen, sollen nur die Varianten „`++x`“ und „`--x`“ verwendet werden. Andernfalls muß der Compiler erst eine Kopie des Objekts erstellen, die nach Ausführung des Operators verwendet wird.

Bei Standarddatentypen wie *int* ist der Effekt geringer. Hier kann je nach Situation auch die Post-Variante besser sein, je nach Optimierungen des Compilers auf den Prozessor.

Regel 7.14. *Synchronisation bei Nebenläufigkeit schlank halten*

Es gibt viele Methoden: Semaphoren, Mutex, Critical Sections. Hierbei sind vor C++11 immer plattformspezifische Bibliotheken erforderlich. Die verschiedenen Methoden sind

sehr unterschiedlich in den Laufzeitkosten. Eine genaue Prüfung lohnt sich. Seit C++11 sollten die Standardmethoden verwendet werden.

Einzelne Threads können auch rein mit C ganz ohne Bibliotheken und garantiert ohne Blockieren synchronisiert werden (z. B. Datenaustausch über Ringpuffer). Hierfür braucht man nur eine Variable, die mit atomaren Operationen geschrieben und gelesen werden kann. Einen solchen Typ gibt es in C: `sig_atomic_t`

Regel 7.15. *Je mehr der Compiler über ein Objekt weiß, desto besser kann er optimieren.*

Das gilt für alles, was dem Compiler helfen kann (z. B. `const`, `static` ...). Vordeklarationen wie "`class Klasse;`" zum reinen Bekanntmachen des Namens helfen da nicht (siehe aber auch Regel 8.7).

Moderne Compiler bieten auch programmübergreifende Optimierungen, die über die Übersetzungseinheiten hinaus optimieren können. Dadurch werden viele Einschränkungen beseitigt, allerdings dauert das Linken damit deutlich länger.

Regel 7.16. *Zeiger ggf. als `restrict` deklarieren*

Die Verwendung von `restrict` kann dem Compiler helfen. Es ist auch ein wichtiger Hinweis an den Aufrufer einer Funktion, wenn die Funktion entsprechend implementiert ist. C++ schließt `restrict` allerdings aus. Die verschiedenen Compiler bieten auch proprietäre Varianten an.

Regel 7.17. *Bei Typkonvertierungen die Laufzeiten der Zielplattform berücksichtigen*

Konvertierungen zwischen Fließkomma und Ganzzahl sind auf üblichen PCs relativ langsam, hängen aber vom Compiler und Prozessor ab. Auch ein Vorzeichen spielt eine Rolle: `(float)int` ist schneller als `(float)uint`.

Die Zeiten für `(float)double` und `(double)float` hängen von Compiler und Prozessor ab, können aber relativ langsam sein. Beide Typen sollten also nicht in Rechnungen gemischt werden. Das gilt auch für `float*3.0`, stattdessen lieber `float*3.0f` schreiben, sonst muß der Compiler in `double` rechnen.

Regel 7.18. *Bei Rechenoperationen die Laufzeiten der Zielplattform berücksichtigen*

Auf üblichen PCs ist die Rechnung `uint/Konstante` schneller als `int/Konstante`. Das gilt auch für „`%`“. Es kann sich also eine explizite Typumwandlung lohnen, wenn die Variable sicher positiv ist: `(unsigned)int/Konstante`. Einige Compiler ersetzen eine Division durch eine konstante Zweierpotenz automatisch durch eine schnelle Bit-Verschiebung.

Regel 7.19. *`for`-Schleifen mit Ganzzahllaufvariablen nutzen*

Wenn eine Schleife eigentlich mit einer Fließkommalaufvariablen arbeitet, sollte zusätzlich eine Ganzzahllaufvariable eingeführt und in der Abbruchbedingung genutzt werden. Ist die Richtung der Laufvariablen egal, sollte sie nach 0 laufen. Sequentielle Array-Zugriffe sollten wegen des Caches aber vorwärts erfolgen.

Regel 7.20. *Organisation von Arrays*

Bei mehrdimensionalen Arrays sollte der letzte Index immer am schnellsten laufen, damit der Cache effektiv arbeiten kann.

Für nicht sequentiellen sondern zufälligen Zugriff sollten die inneren Größen Zweierpotenzen aufweisen (gilt für einzelne Elemente und bei mehrdimensionalen Arrays für ganze Zeilen). Bei sequentiellem Zugriff ist das nicht erforderlich. Allerdings kann je nach Zugriffsmuster eine Zweierpotenz für den Cache auch gerade besonders schädlich sein.

Regel 7.21. *RTTI kostet Zeit*

Die Laufzeitauswertung von Datentypen (RTTI) in C++ kostet Zeit. Normalerweise braucht man das auch nicht, sondern bedeutet ein Architekturproblem. In Projekten, in denen das nicht erforderlich ist (die weder `dynamic_cast` noch `typeid` verwenden), kann RTTI im Compiler ganz abgeschaltet werden. Beim `gcc` mit „`-fno-rtti`“

8. Präprozessor

Regel 8.1. *h-Dateien müssen beliebig oft eingebunden werden können.*

Jede `h`-Datei muß sich vor mehrfacher Auswertung selbst schützen durch eine Klammerung in `#ifdef`/`#endif`. Sonderformen wie „`#pragma once`“ sind proprietär und daher zu unterlassen.

Regel 8.2. *Jede h-Datei muß für sich nutzbar sein, ohne daß zuvor andere geladen werden müssen.*

Wenn zusätzliche Abhängigkeiten bestehen, muß die `h`-Datei alle Abhängigkeiten selbst einbinden.

Regel 8.3. *Jede Datei darf nur die Abhängigkeiten einbinden, die sie selbst braucht.*

Weitere Abhängigkeiten einer `c`-Datei werden innerhalb der `c`-Datei eingebunden.

Regel 8.4. *In C++ die h-Dateien von C über die neuen Namen einbinden*

Statt „`#include <math.h>`“ heißt es in C++ „`#include <cmath>`“.

Regel 8.5. *System-h-Dateien mit „<>“ einbinden, projekteigene mit Anführungszeichen „""“*

Damit werden unterschiedliche Pfade festgelegt und die Zuordnung der Dateien wird unmittelbar sichtbar.

Regel 8.6. *h-Dateien in fester Reihenfolge einbinden*

Grundsätzlich sind alle h-Dateien am Anfang der c-Dateien einzubinden.

Als erstes in einer c-Datei die eigene h-Datei einbinden, um sie automatisch auf Vollständigkeit zu prüfen. Dann die Systemdateien. So werden mögliche ungewollte Namenskonflikte minimiert. Dann fremde Bibliotheken und am Schluß der Rest aus dem eigenen Projekt:

```
// Am Anfang von meine.cpp
#include "meine.h"             // Eigene h-Datei
#include <cmath>               // C
#include <algorithm>           // C++
#include "fremde_lib.h"        // Bibiotheken
#include "eigenes_projekt.h"   // Sonstiges aus dem Projekt
```

Dateien, die ausnahmsweise erst später in der c-Datei und ggf. sogar mehrfach eingebunden werden, bekommen eine andere Dateiendung, z. B. „`.inc`“.

Regel 8.7. *h-Dateien klein halten*

In h-Dateien nur das definieren, was wirklich exportiert werden soll. Dinge, die nur in der zugehörigen c-Datei erforderlich sind, gehören in die c-Datei. Braucht man temporär `define`s in der h-Datei, so sollten diese Definitionen hinterher wieder mit `#undef` entfernt werden.

Eine h-Datei darf kein globales `using namespace` enthalten. Das sollte auch sonst überall vermieden werden.

Alle exportierten Definitionen können externe Dateien beeinflussen, in denen die h-Dateien eingebunden werden. Außerdem ist es guter Programmierstil, die exportierten Schnittstellen kompakt und übersichtlich zu halten.

Auch wenn sich reine Vordeklarationen von Klassen (`class Klasse;`) ohne Implementierung hinsichtlich Compiler-Optimierungen negativ auswirken können (siehe Regel 7.15), gibt es gute Gründe dafür. Sie reduzieren die Dateiabhängigkeiten beim Kompilieren, was neben schnellerem Kompilieren auch zyklische Abhängigkeiten von h-Dateien auflösen kann.

Aber vor jeder Verwendung solcher vordeklarierten Elemente muß es unbedingt richtig definiert werden. Schon ein Weiterreichen des bloßen Zeigers (oder Referenz) ist eine Verwendung. Andernfalls kommt es zu vielerlei Problemen, die Compiler nicht erkennen und auch nicht davor warnen (z. B. typabhängige Überladungen, unklare Vorgabeargumente).

Regel 8.8. *Präprozessorzeilen immer am Zeilenanfang beginnen*

Das Zeichen `#` soll immer ganz vorne stehen und Einrückungen nicht mitmachen, um auch mitten im Programmtext den Präprozessor sofort zu erkennen. Einrückungen werden dann hinter dem Zeichen gemacht (siehe Regel 4.23):

```
class Klasse1 {
  class Klasse2 {
#   define ENUM(a) void a(void);
    LISTE
#   undef ENUM
  };
};
```

Regel 8.9. *Für Präprozessordefinitionen nur Großbuchstaben verwenden*

Dadurch sind Präprozessoraktionen von anderen unterscheidbar.

Regel 8.10. *Makroparameter und -ausdrücke bei Verwendung klammern*

Richtig: `#define SQR(a) ((a)*(a))`

Falsch: `#define SQR(a) a*a`

Da der Präprozessor eine reine Textersetzung durchführt, können ohne Klammern unerwartete Effekte eintreten. Mit Klammern verhalten sich Makros, was die Rechenergebnisse angeht, wie Funktionen.

Regel 8.11. *Bei der Definition von Makros dürfen keine Leerzeichen vor den Parameterklammern eingefügt werden.*

Andernfalls erkennt der Präprozessor nicht, daß es sich um ein Makro handelt. Im folgenden Beispiel wird der Aufruf `SQR(x)` folgendermaßen ersetzt:

Richtig:	`#define␣SQR(a)␣((a)*(a))`	→	`((x)*(x))`
Falsch:	`#define␣SQR␣(a)␣((a)*(a))`	→	`(a)␣((a)*(a))(x)`

Regel 8.12. *Besteht ein Makro aus mehreren Anweisungen, in einen `do-while`-Block einbetten*

Damit kann dieser Block an beliebigen Stellen wie eine Funktion genutzt werden. Es folgt kein Semikolon am Ende, das wird üblicherweise bei Verwendung des Makros angefügt.

```
#define M(a) do {...} while(false)
```

Regel 8.13. *Konstanten statt mit `#define` als Variable mit `const` bzw. `constexpr` definieren*

Das erlaubt eine Typprüfung durch den Compiler.

Regel 8.14. *Makros nach Möglichkeit durch `static inline`-Funktionen definieren*

Das erlaubt eine Typprüfung durch den Compiler und teilweise sogar bessere Optimierungen. Typunabhängigkeit wird mit `templates` erreicht.

Regel 8.15. *Mathematische Konstanten wie `M_PI` nicht selbst definieren*

Diese Konstanten sind zwar nicht C-Standard, dennoch bietet jeder Compiler sie an.

Bei Microsoft muß das Symbol `_USE_MATH_DEFINES` dafür vor dem Einbinden von `math.h` bzw. `cmath` definiert werden.

Der `gcc` bietet verschiedene Schalter für das Freischalten von bestimmten Spracherweiterungen an. Diese Konstanten können mit der Definition des Symbols `_XOPEN_SOURCE` aktiviert werden.

Diese Symbole sollten nicht in den Dateien direkt vor dem betreffenden `#include` eingefügt werden, sondern auf `make`-Ebene. Diese Dinge sind einerseits proprietär und anderseits kann nur so sichergestellt werden, daß sie garantiert vor dem ersten `#include` der jeweiligen `h`-Datei definiert sind, weil `h`-Dateien oft implizit durch andere `h`-Dateien irgendwo in der Hierarchie schon vorher eingebunden sind.

Regel 8.16. *Funktionszeiger über Makros definieren*

Funktionszeiger sind in C schlecht lesbar. Praktisch ist eine Definition mittels Makro, wodurch sichergestellt ist, daß alle beteiligten Definitionen konsistent sind (z. B. für Callbacks):

```
#define NAME_M(name) void (name)(int parameter, ...)
```

Der Funktionstyp `name_t` kann dann mit dem Makro definiert werden:

```
typedef NAME_M(name_t);
```

Und passende Funktionen werden ebenfalls über das Makro definiert:

```
NAME_M(Funktionsname) {...}
```

Seit C++11 gibt es auch neue Möglichkeiten mit `std::function`.

Regel 8.17. *Sprachversionen können abgefragt werden*

Für die Auswertung in `#if`-Abfragen kann die Sprachversion abgefragt werden. Dies sollte nur wenig verwendet werden, um die Portabilität zu maximieren, aber manchmal ist es erforderlich:

`__STDC_VERSION__`	
C89	undefiniert
C95	199409L
C99	199901L
C11	201112L

`__cplusplus`	
C++98	199711L
C++11	201103L
C++14	201402L

9. Dokumentieren

Regel 9.1. *Kommentare als Zeilenkommentare schreiben („//“)*

Dadurch können beim Testen ganze Blöcke leicht mit „`/*...*/`“ auskommentiert werden.

Regel 9.2. *Kommentare kompatibel zu `Doxygen` schreiben („`///`“)*

Der Zusatzaufwand für `Doxygen` ist vernachlässigbar, und man hat gleich eine gute Dokumentation. Ausnahmen sind interne Beschreibungen innerhalb von Funktionen, die nicht in die `Doxygen`-Dokumentation gelangen.

Entsprechend werden Stellen mit offenen Aufgaben mit `\todo` innerhalb eines `Doxygen`-Kommentars (also nach „`///`“) eingeleitet. Für das eigentlich übersichtlichere `\TODO` muß die `Doxygen`-Konfiguration angepaßt werden.

Regel 9.3. *Den Zweck jeder Funktion beschreiben*

Eine kurze Beschreibung im Vorspann der Funktion erleichtert das Verständnis sehr. Beispiel:

```
/// Alle reservierten Dinge freigeben.
/// \post   Alles ist freigegeben (Speicher, Dateien, Thread).
/// \return Fehlerwert als Information über Erfolg/Misserfolg.
return_t Freigeben(void);
```

Regel 9.4. *Den Zweck jeder Quelltextdatei am Dateianfang beschreiben*

Eine kurze Beschreibung im Vorspann der Datei erlaubt eine erste Orientierung. Vorlage:

```
/// Der erste programmierbare Rechner.
/// \file
/// \author Konrad Zuse
/// \date   22.06.1941 08:15:42
```

Regel 9.5. *Kommentare mit zusätzlicher Information*

Die bloße Wiederholung des Inhalts eines Bezeichners ist nicht hilfreich.

Regel 9.6. *Deklarationen, die in der `h`- und `c`-Datei vorkommen, nur einmal in der `h`-Datei dokumentieren*

`Doxygen` muß sich bei mehreren Dokumentation für dasselbe Objekt für eine entscheiden. Außerdem ist es fehleranfällig, mehrere Dokumentationen synchron zu halten. Die `h`-Datei ist wichtiger, da hier die exportierten Dinge beschrieben werden.

Zusätzliche Implementierungsdetails sollen aber in der `c`-Datei beschrieben werden. Dann aber nicht für `Doxygen`.

Regel 9.7. Kommentare für jede *Variablen- und Parameterdeklaration. Es* muß *eine physikalische Einheit enthalten (in eckigen Klammern), sofern es sich um eine Größe mit Einheit handelt.*

Von dieser Regel sollte es keine Ausnahmen geben. Bei einfachen Zählervariablen wie `i` kann der Kommentar allerdings sehr kurz ausfallen. Beispiel:

```
constexpr double g = 9.80665;  ///< [m/s^2] Erdbeschleunigung
```

Normalerweise sollte der Kommentar hinter der Variable mit 2 Zeichen Abstand stehen, so sind Variablenlisten übersichtlich untereinander ausrichtbar. Bei zu viel Text muß entschieden werden, ob mehrere Zeilen dahinter oder davor übersichtlicher sind.

Regel 9.8. *Funktionsparameter immer* hinter *der Parameterdeklaration dokumentieren*

Eine Dokumentation vor der Funktion ist fehlerträchtig, da sie von Hand synchronisiert mit der eigentlichen Deklaration gehalten werden muß. Zu beachten ist, daß das abschließende Semikolon der Funktionsdeklaration in der Zeile nach dem letzten Parameterkommentar stehen muß (die Position der schließenden runden Klammer ist egal). Beispiel:

```
/// Newton-Verfahren zum Suchen von Nullstellen der Funktion f(x).
/// Es wird eine feste Anzahl von Iterationen durchgeführt.
/// \return Position der genaeherten Nullstelle
double Newton(Funk_t*       f,      ///< Die Funktion f(x)
              Funk_t*       df,     ///< Ableitung von f: df/dx
              const double  start,  ///< Startwert fuer Suche
              const uint8_t zahl    ///< Anzahl Iterationsschritte
             );
```

Eine Ausnahme bildet nur die Dokumentation von Makros, bei denen der gesamte Kommentar vor der Definition stehen muß, inklusive der Zeilen mit `\param`.

Regel 9.9. *Auch wenn die Schlüsselwörter von C/C++ englisch sind, dürfen Kommentare (und Bezeichner) in deutsch verfaßt sein.*

Es ist sinnlos, zwanghaft englisch schreiben zu wollen, wenn sowohl der Schreiber als auch der Leser dafür Wörterbücher brauchen und teilweise die falschen Wörter benutzen (z. B. weit verbreitet ist die Verwechslung von actual und current). Die eigene Muttersprache ist am besten geeignet. Keine Vermischung mehrerer Sprachen.

10. Quelltextformatierung

Regel 10.1. *Einheitlichkeit*

Für eine Lesbarkeit ist eine einheitliche Formatierung wichtig. Das gilt auch für Namenskonventionen, die verwendete Sprache usw.

Regel 10.2. *Gleichartige Programmzeilen systematisch ausrichten*

Die Lesbarkeit wird dadurch stark verbessert. Diese Anforderung beschränkt sich nicht nur auf das Ausrichten von Gleichheitszeichen, sondern gilt auch für Parameter, Operatoren, Klammern usw.

Regel 10.3. *Keine harte Grenze für Zeilenlägen*

Es gibt nur die Regel, daß es sinnvoll und übersichtlich sein muß.

Regel 10.4. *Unnötige vertikale Platzverschwendung vermeiden*

Wenn der Programmtext vertikal zu sehr gestreckt wird, verliert man den Überblick über den Algorithmus. Unnötige Zeilen sind also zu vermeiden.

Öffnende geschweifte Klammern kommen in dieselbe Zeile wie der Kopf des Blocks (`for`, `if`, `while` usw.). Ausnahme: Der Kopf geht über mehrere Zeilen. Weiterer Sonderfall: Funktionsdefinitionen.

Weiterhin sollen in einer Zeile stehen: „`} else {`“, „`} else if() {`“ und „`} while();`“.

Die schließende Klammer kommt in eine eigene Zeile, wodurch die Einrückung abgeschlossen wird.

Die Lesbarkeit wird durch die Einrückung hinreichend sichergestellt.

Regel 10.5. *Die Einrückung einheitlich mit 2 Leerzeichen. Tabulatoren nicht verwenden.*

Durch die Einrückung werden die Blöcke optisch kenntlich gemacht. Zu breite Einrückungen führen zu horizontaler Platzverschwendung und fördern die Lesbarkeit nicht (bis zu vier Leerzeichen sind akzeptabel). Tabulatoren führen in verschiedenen Editoren zu unterschiedlichen Ergebnissen, so daß die Einrückung teilweise falsch aussieht. Leerzeichen sind überall eindeutig. Proportionalschriftarten, wie in manchen Editoren angeboten, sind beim Programmieren generell zu unterlassen.

Regel 10.6. *Klammerausdrücke über mehrere Zeilen entsprechend der Klammerung einrücken*

Fortsetzungen von nicht abgeschlossenen Klammerausdrücken werden entsprechend eingerückt fortgesetzt:

```
while(a && (b ||
            c))
{...}
```

Regel 10.7. *Leerzeichen bei Operatoren*

Binäre Operatoren werden durch Leerzeichen von ihren Parametern getrennt.

Unäre Operatoren werden ohne Leerzeichen direkt an den Parameter geschrieben.

Nach dem Komma-Operator und dem Semikolon folgt ein Leerzeichen, falls nicht ohnehin eine neue Zeile beginnt.

Zusätzliche Leerzeichen können eingefügt werden, um mehrere Zeilen gleich auszurichten.

Regel 10.8. *Funktionsklammern ohne Leerzeichen direkt hinter den Funktionsnamen*

Das gilt auch für textuelle Operatoren (siehe Regel 10.9).

Regel 10.9. *Alle textuellen Operatoren wie* `if`, `for`, `while`, `return` *wie Funktionen schreiben.*

Es sollen also Klammern um die Parameter gesetzt werden, die ohne Leerzeichen an die Operatoren gesetzt werden. Diese Operatoren können als Funktionen interpretiert werden, die teilweise als zusätzlichen „Parameter“ einen Anweisungsblock (ggf. in geschweiften Klammern) für „lazy evaluation“ erhalten.

Regel 10.10. *Leere Schleifen kennzeichnen*

Wenn Schleifen keinen Rumpf haben, dann soll das explizit gemacht werden, indem anstelle des Rumpfes entweder ein leerer `{}` oder `continue` steht.

Regel 10.11. *Keine Sonderzeichen im Quelltext*

Sonderzeichen wie Umlaute sind im C-Quelltext nicht erlaubt. Einige Compiler kommen damit nicht klar.

Regel 10.12. *Formatierung von Klassen*

Die Zugriffsmodifizierer `private:` usw. werden eingerückt. Alles was danach kommt nochmals. Dadurch sind die Methoden gegenüber der Klasse doppelt eingerückt.

Die öffnende geschweifte Klammer steht mit dem Klassennamen in einer Zeile, es sei denn, es gibt Basisklassen:

```
class Klasse
: public Klasse1,
  public Klasse2
{
  public:
    void Klasse(void);
};
Klasse::Klasse(void)
: Klasse1(),
  Klasse2()
{...};
```

Diese Regeln gelten genauso für Strukturen. Diese sollten aber außer einfachen Konstruktoren keine Intelligenz (Methoden) besitzen. Keine virtuelle Methoden und Vererbung in Strukturen!

Regel 10.13. *Formatierung von* `templates`

Das Schlüsselwort `template` steht in einer eigenen Zeile vor der Klasse oder Funktion. So können auch einzelne Parameter wie Funktionsparameter einzeln Dokumentiert werden:

```
template<typename A,
         typename B>
A f(const B& b);
```

11. Compiler

Regel 11.1. *Der Compiler sollte möglichst viele Warnungen generieren.*

Grundsätzlich sollten alle verfügbaren Warnungen genutzt werden. Meistens haben sie einen Sinn. Nur wenige wirklich störende/sinnlose Warnungen dürfen deaktiviert werden (siehe Abschnitt B).
Weitere Dinge können mit zusätzlichen Analyseprogrammen überprüft werden, z. B. mit `cppcheck`.

Regel 11.2. *Warnungen müssen behoben werden.*

Beheben und nicht deaktivieren! Um das sicherzustellen, sollte mit `-Werror` übersetzt werden.

Regel 11.3. *Überwachung der Standardkonformität*

Die Optionen für den `gcc` lauten für C "`-std=c11 -pedantic`" und für C++ "`-std=c++11 -pedantic`". Für weitere Standardversionen existieren entsprechende Werte.

Regel 11.4. *Optimierungen nutzen*

Optimierungsfähigkeiten des Compilers sollten genutzt werden. Der Geschwindigkeitsgewinn kostet keine Entwicklungszeit (siehe Abschnitt B).

A. Vorlagen

A.1. Klassenvorlage

A.1.1. Vor C++11

```
class K98 {
  private:
    // Ggf. implementieren, sonst als private leer lassen:
    // explicit: Nur weglassen, wenn unbedingt erforderlich
    // throw:    Alle Exceptions dokumentieren.
    //           Leider ist throw() unbrauchbar, deshalb als Kommentar
    [explicit] K98(void)       /*throw(...)*/;
    [explicit] K98(const K98&) /*throw(...)*/;
    const K98& operator=(const K98&) /*throw(...)*/;
  public:
    // virtual:  Genau dann, wenn von K98 abgeleitet werden koennte
    // throw:    Nur in Ausnahmefaellen exceptions aus Destruktoren
    [virtual] ~K98() /*throw(...)*/ {}

    // virtual:  Eine virtuelle Basisklassenmethode wird ueberschrieben
    //           oder Kindklasse darf ueberschreiben.
    // const:    Keine von aussen sichtbare Seiteneffekte innerhalb
    //           und ausserhalb des Objekts
    [virtual] T Methode([const] int i,  // Normalerweise const
                        const   T&  r,  // Konstante grosse Parameter
                        T* const    z   // [out] Ausgabeparameter
                       ) [const] /*throw(...)*/;
};
```

A.1.2. Seit C++11

```
// final: Wenn von K11 nicht abgeleitet werden soll
class K11 [final] {
  // Implizite Konstruktoren/Operatoren verbieten:
  // explicit:  Nur weglassen, wenn unbedingt erforderlich
  // noexcept:  Es werden keine Exceptions geworfen, besonders wichtig
  //            bei Move-Konstruktor und -Zuweisungsoperator.
  //            Andernfalls in throw() alle Exceptions dokumentieren.
  //            Leider ist throw() deprecated, deshalb als Kommentar.
  // delete:    Ggf. durch =default oder eigene Implementierung ersetzen
  [explicit][constexpr] K11(void)       [noexcept|/*throw(...)*/] = delete;
  [explicit][constexpr] K11(const K11&) [noexcept|/*throw(...)*/] = delete;
  [explicit][constexpr] K11(K11&&     ) [noexcept|/*throw(...)*/] = delete;

  // const:     Nur bei =default weglassen (leider nicht erlaubt).
  [constexpr][const] K11& operator=(const K11&) & [noexcept|/*throw(...)*/]
    = delete;
  [constexpr][const] K11& operator=(K11&&     ) & [noexcept|/*throw(...)*/]
    = delete;

  // virtual:   Genau dann, wenn K11 nicht final.
  // override:  Eine virtuelle Basisklassenmethode wird ueberschrieben,
  //            dann auch virtual.
  // noexcept:  Nur in Ausnahmefaellen exceptions aus Destruktoren.
  [virtual] ~K11() [override] [noexcept|/*throw(...)*/] {}

  // virtual:   Eine virtuelle Basisklassenmethode wird ueberschrieben
  //            (dann auch override) oder Kindklasse darf ueberschreiben.
  // constexpr: Die Methode hat keine Seiteneffekte und das Ergebnis basiert
  //            nur auf den Parametern und this (automatisch auch const).
  // const:     Keine von aussen sichtbaren Seiteneffekte innerhalb und
  //            ausserhalb des Objekts
  // final:     Eine Kindklasse darf nicht ueberschreiben.
  [virtual][constexpr] T Methode([const] int i, // Normalerweise const
                                 const   T&  r, // Konstante grosse Parameter
                                 T* const    z  // [out] Ausgabeparameter
                                ) [const] [override] [final]
                                  [noexcept|/*throw(...)*/];

  [explicit][constexpr] operator Typ_cast()
    const [override] [final] [noexcept|/*throw(...)*/];
};
```

A.2. Funktionsvorlage

```
// static:    Funktion nur innerhalb dieser Uebersetzungseinheit sichtbar.
// constexpr: Keine Seiteneffekte und Ergebnis basiert nur auf Parametern.
[static][constexpr] T Funktion([const] int i, // Normalerweise const
                               const   T&  r, // Konstante grosse Parameter
                               T* const    z  // [out] Ausgabeparameter
                              ) [noexcept|/*throw(...)*/];
```

B. Compiler

B.1. gcc

B.1.1. Warnungen

Der gcc besitzt leider keine Option, um wirklich alle Warnungen zu aktivieren. Empfohlene Compiler-Optionen für Version 4.9 sind:

Allgemeine Warnungen: `-fdiagnostics-color=auto -Wall -Wextra -Werror -pedantic`
`-Wundef -Wcast-qual -Wcast-align -Wwrite-strings -Wmissing-field-initializers`
`-Wpacked -Wredundant-decls -Wcomment -Wendif-labels`
`-Wstrict-overflow -Wmissing-declarations -Wconversion -Wsign-conversion`
`-Wuninitialized -Wlogical-op -Wdouble-promotion -Wunused-local-typedefs`
`#-Wunreachable-code # Meldungen bei inline-Funktionen und nach Endlosschleifen`
`#-Wframe-larger-than # Überwachung der Stack-Groesse`

Nur für C: `-Wbad-function-cast -Wstrict-prototypes -Wold-style-definition`
`-Wmissing-prototypes -Wnested-externs -Wc++-compat`
`#-Wtraditional-conversion # Falschalarm fuer Parameter aus char oder bool`

Nur für C++: `-Wctor-dtor-privacy -Wold-style-cast`
`-Woverloaded-virtual -Wnon-virtual-dtor -Wnoexcept`
`-Wzero-as-null-pointer-constant`

B.1.2. Optimierungen

Hier ist eine Auswahl von je nach Projekt geeigneten Optionen vom gcc:

Allgemeine Optimierung: `-march=native -flto -finline-functions-called-once`
`-floop-interchange -floop-strip-mine -floop-block`

Optimierungen für Geschwindigkeit: `-O3`
`-fmodulo-sched -fmodulo-sched-allow-regmoves`
`-fgcse-sm -fgcse-las -fgcse-after-reload`
`-ftree-loop-linear -ftree-loop-im -ftree-loop-ivcanon -fivopts`
`-fvect-cost-model -ftracer`
`-funroll-loops -fvariable-expansion-in-unroller -funswitch-loops`
`-fbranch-target-load-optimize`

Projektspezifisch: `-static -fno-rtti -Ofast -ffast-math -ffunction-sections -fdata-sections`

Plattformspezifisch: `-march -mfpmath -mmmx -msse -msse2 -msse3 -mssse3 -msse4.2 -mavx -mveclibabi -m32 -m64 -maccumulate-outgoing-args -minline-all-stringops -mrecip`

C. Bit-Tricks

Gezieltes Arbeiten mit Bits sollte wenn möglich unterbleiben, da es unübersichtlich ist. Aber wie immer gibt es Ausnahmen. Dann kann die folgende Tabelle helfen. Alle diese Aktionen sind nur mit vorzeichenlosen Datentypen auszuführen.

Ausdruck	Bedeutung
`~-x`	`== x-1`
`-~x`	`== x+1`
`~x + 1`	`== -x`
`x & -x`	extrahiert niedrigstes gesetztes Bit
`x | -x`	setzt alle Bits >= niedrigstes gesetztes Bit, löscht den Rest
`x ^ -x`	setzt alle Bits > niedrigstes gesetztes Bit, löscht den Rest
`x ^ (x-1)` `~x ^ -x`	setzt alle Bits <= niedrigstes gesetztes Bit, löscht den Rest

`~x & (x-1)`
`~(x | -x)`
`(x & -x) - 1`
	setzt alle Bits < niedrigstes gesetztes Bit, löscht den Rest

`x | (x-1)`	setzt alle Bits < niedrigstes gesetztes Bit
`x | (x+1)`	setzt niedrigstes ungesetztes Bit
`x & (x-1)`	löscht niedrigstes gesetztes Bit
`x / (x & -x)`	schiebt nach rechts bis niedrigstes gesetztes Bit unten steht

D. C-Eigenheiten

D.1. Implizite Typkonvertierungen

Wenn zwei Variablen oder Zahlen miteinander verrechnet werden (z. B. mittels eines Operators der Grundrechenarten), wird automatisch ein gemeinsamer Typ bestimmt, in den die beiden Operanden konvertiert werden. Folgende Regeln gelten dabei, von oben nach unten gelesen, bis eine Zeile zutrifft.

Operand 1	Operand 2	Gemeinsamer Zieltyp
Usual arithmetic conversions:		
`long double`	`y`	`long double`
`double`	`y`	`double`
`float`	`y`	`float`
Integer promotions:		
`unsigned x`	`unsigned y`	`unsigned max(rank(x), rank(y))`
`signed x`	`signed y`	`signed max(rank(x), rank(y))`
`unsigned x`	`signed y`	`unsigned x` (wenn `rank(x) >= rank(y)`)
`unsigned x`	`signed y`	`signed y` (wenn `signed y` alle Werte aus `unsigned x` darstellen kann)
`unsigned x`	`signed y`	`unsigned y` (sonst; dieser Fall wird auf üblichen Architekturen nie erreicht)

D.2. Ganzzahlkonstanten

Direkt angegebene Ganzzahlkonstanten erhalten einen Datentyp entsprechend der folgenden Tabelle. Es wird der jeweils erste Typ der folgenden Tabelle gewählt, in dem die betreffende Zahl dargestellt werden kann (s=`signed`, u=`unsigned`, l=`long`).

Suffix	Dezimal	Oktal/Hexadezimal
ohne	s, ls, lls	s, u, ls, lu, lls, llu
`U`	u, lu, llu	u, lu, llu
`L`	ls, lls	ls, lu, lls, llu
`UL`	lu, llu	lu, llu
`LL`	lls	lls, llu
`ULL`	llu	llu

D.3. Operatorprioritäten

Bei einer Verkettung mehrerer Operatoren in einer Anweisung werden sie entsprechend folgender Prioritätsliste ausgeführt:

Operatoren	Auswerterichtung
`::`	→
`a++`, `a--`, `f()`, `[]`, `.`, `->`	→
`++a`, `--a`, `+a`, `-a`, `!`, `~`, `(cast)`, `*a`, `&a`, `sizeof`, `new`, `delete`	←
`.*`, `->*`	→
`*`, `/`, `%`	→
`+`, `-`	→
`<<`, `>>`	→
`<`, `<=`, `>`, `>=`	→
`==`, `!=`	→
`&`	→
`^`	→
`|`	→
`&&`	→
`||`	→
`?:`, `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	←
`throw`	←
`,`	→
`return`	←