Hochschule Augsburg
University of Applied Sciences

HSA_innos
Institut für innovative Sicherheit

# Secure Coding

## C vs. Rust

**Autoren:**
Simon Flauger, Stefan Pschenitza, Andreas Wundlechner, Matthias Pabel, Mücahit Uzun, Micha Sengotta, Dominik Rößle, Eugen Matery, Christoph Wengenmayr, Raphael Mayr, Rudi Loderer, Georgia Oustria, Oliver Lassonczyk, Tobias Hainzinger, Katrin Ruttmann, Christan Kreutmeier, Hanne Beuter, Liam Zinth, Florian Klein, Timo Gruber, Thomas Hüttenhofer, Fabio Aubele, Martin Lautenbacher, Marco Wölfel, Djene Cherif, David Yesil, Alexander Ruhl, Benedikt Handschuh, Simon Hellbrück, Adam Lozinski, Michael Kovalenko, Ömer Aydin, Maximilian Klimm, Fasika Kebede, Lukas Edlböck, Maximilian Schöberl, Markus Ziefle, Andreas Hafner, Franziska Bartenschlager, Sebastian Grußler, Daniel Hein, Adrian Friese ...

**Betreuer:**
Prof. Dr.-Ing. Dominik Merli

24. Juni 2019

# Inhaltsverzeichnis

# 1 Einleitung

Egal ob im Automobil, in der Industrie 4.0 oder in der IT-Infrastruktur von Unternehmen - sichere Software ist ein entscheidender Beitrag zur IT-Sicherheit. Um jedoch sichere Software entwickeln zu können bedarf es vieler Richtlinien und Werkzeuge.

In diesem studentischen Projekt werden die Sprachen C und Rust anhand diverser Secure Coding Regeln miteinander verglichen und bewertet, um ein Gefühl dafür zu bekommen, welchen Beitrag Rust zur Entwicklung sicherer Software leisten kann.

# 2 Ausgewählte Secure Coding Regeln

In den folgenden Abschnitten wird jeweils eine spezifische Secure Coding Regel dahingehend untersucht inwiefern sie in C bzw. in Rust beachtete werden muss bzw. wie deren sichere Umsetzung in der jeweiligen Sprache aussieht.

Die Regeln beziehen sich auf diese Website:
`https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard`

## 2.1 EXP30-C

In diesem Kapitel geht es um Probleme, wenn der Compiler die Reihenfolge der Abarbeitung des Codes selbst bestimmt. Der Code kann so geschrieben werden, dass er unterschiedlich interpretiert werden kann. Dazu gibt es die Regel *„Do not depend on the order of evaluation for side effects“*. Im folgenden wird erläutert, was damit gemeint ist.

### 2.1.1 Definition der Regel

Wenn ein Objekt mehrmals zwischen Sequence Points geändert wird, kann nur schwer gesagt werden, was das Endresultat sein wird. Sequence Points sind, zum Beispiel Semikolon, Komma und das logische UND. Wenn der Code ausgeführt wird und es zu einem Sequence Point gelangt, wird versichert, dass alle vorherigen Seiteneffekte bis zu diesem Punkt fertig ausgeführt wurden.

### 2.1.2 Inkonsistenz bei C-Compilern

Ein kurzes Beispiel wird hierbei helfen, um dies zu verstehen. Hierzu ist die Zeile 4 im Listing 1 zu betrachten. Wie vorher beschrieben, wird hier eine Variable `i` mehrfach zwischen zwei Sequence Points verändert. Der vordere Sequence Point ist das Semikolon am Ende der Zeile 3 und der hintere ist das Semikolon am Ende der Zeile 4.

In so einer Codezeile ist das Verhalten undefiniert, da keine genaue Reihenfolge erkannt werden kann. Alles zwischen den 2 Sequence Points wird mit einem Mal ausgeführt. Sichtbar wird es, wenn dieser Codeblock mit verschiedenen Compilern kompiliert wird. In diesem Artikel wird GCC 8.3.0 und CLANG 8.0.0 verwendet. Nachdem dieser Code mit beiden Compilern kompiliert und ausgeführt wurde, kommt es zu zwei unterschiedlichen Ergebnissen. Der mit dem GCC kompilierte Code hat in der Variable `result` den Wert 3, wohingegen sies mit CLANG den Wert 2 erhält.

```
1 int main(void)
2 {
3     int i = 2;
4     int result = --i + i++;
5     return 0;
6 }
```

Listing 1: Variable i wird mehrfach in einem Befehl verändert

### 2.1.3 Hinter den Kulissen

Als nächstes wird darauf eingegangen, was von den Compilern erzeugt wird, um die unterschiedlichen Ergebnisse zu verstehen. Wie erwartet gehen beide Compiler mit der Initialisierung der Variable `i` ähnlich um, unterscheiden sich jedoch deutlich mit dem Umgang mit der Zeile 3 des C-Codes.

#### 2.1.3.1 GCC

Der GCC dekrementiert im Anschluss die Variable i und speichert die Zahl 1 final ins `%eax` Register. Danach wird das umschließende Register mit dem dekrementierten Wert in der Zeile 7 kopiert, inkrementiert und auf `%edx` geschrieben. In `%edx` steht nun die 2. Zeile 8-9 sind unnötige Kopierbefehle. Schlussendlich werden in der Zeile 10 die Werte 1 und 2 addiert und es kommt eine 3 raus.

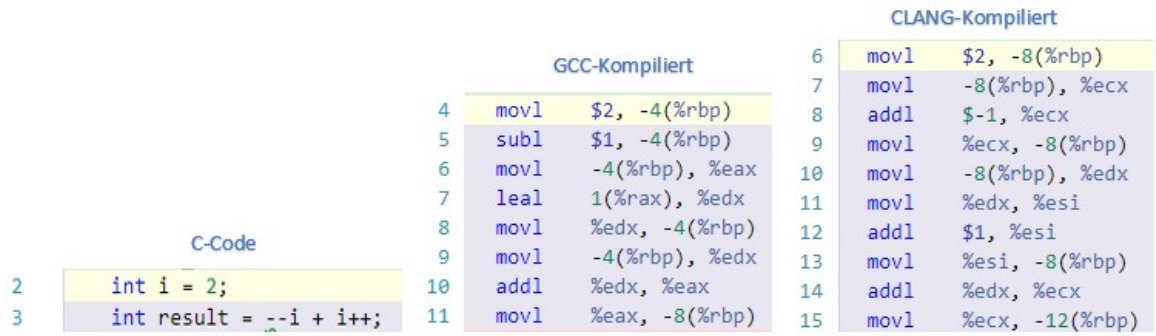


Abbildung 1: Kompilierter Code

#### 2.1.3.2 CLANG

Ähnlich wie vorher wird die Zahl 2 dekrementiert und in `%ecx` gespeichert. Danach wird die dekrementierte Variable `i` nach `%edx` kopiert und eine weitere Kopie nach `%esi` verschoben. Die Kopie in `%esi` wird inkrementiert, aber in der darauf folgenden Zeile 14 werden nur `%edx` und `%ecx` addiert. Also werden 1 und 1 addiert und es kommt die zwei raus. Als Ergebnis wird hier der inkrementierte Wert im `%esi` Register ignoriert.

### 2.1.4 Lösung

Eine mögliche Lösung wäre diese Modifikationen in eigene Zeilen zu bewegen. So sind diese Operationen wieder klar geordnet und klar durch das Sequence Point Semikolon getrennt. Werden die Anweisungen umgestellt, liefern wieder beide Compiler dieselben Werte. Ob der Code an sich sinnvoll ist, ist nicht wichtig.

### 2.1.5 Wie geht Rust damit um?

Rust macht es sich einfach. Pre- und Post-Inkrementierung wird in Rust nicht unterstützt. Sie halten es für komplex und wissen, dass sie zu undefiniertem Verhalten führen können[1].

### 2.1.6 Fazit

Rust hat dieses Problem von der Wurzel eliminiert und mit C ist eine Neustrukturierung des Codes kein allzu großer Akt. Es sollte einfach darauf geachtet werden, dass solche undefinierbaren Konstrukte nicht gebildet werden. Ein paar Zeilen mehr Code würden den Code lesbarer machen und diese Fälle eliminieren.

[1] https://doc.rust-lang.org/1.6.0/complement-design-faq.html#why-no—x-or-x

## 2.2 EXP32-C

Die Regel EXP 32-C besagt, dass ein als volatil deklariertes Objekt nicht durch eine nicht volatile Referenz referenziert werden darf.

### Volatile in C.

Mit dem Typ-Qualifizierer *volatile* können in C Objekte, insbesondere Variablen, modifiziert werden. Volatile bedeutet so viel wie flüchtig und impliziert, dass sich der Inhalt der Variablen auf unvorhersehbare Weise ändern kann. So kann bspw. auf eine als *volatile*deklarierte Variable durch externe Hardware, Prozesse oder Threads zugegriffen werden. Für die konkret Implementierung und den Compiler sind also der Inhalt der Variablen, und das daraus resultierende Verhalten unbekannt. Für den Compiler bedeutet dass, das der Zugriffe auf volatile Variablen nicht optimieren werden darf und der Inhalt immer aus dem Hauptspeicher gelesen werden muss.

```
1 static volatile int led = 0; //volatile Variable led
2 static int *led_p; // nicht volatiler Pointer led_p
3 led_p = &led;  //Referzierung einer nicht volatilen Referenz auf ein volatiles Objekt
    --> Fehler!!
4 if (*led_p == 1) //Kann vom Compiler optimiert werden da led nie den Wert 1 annimmt
5 {printf("Die LED ist an");} // Wird nie ausgeführt
```

Listing 2: Falsche Referenzierung

Referenziert nun, wie im oberen Beispiel, eine nicht volatile Referenz auf ein volatiles Objekt, führt dies zu undefiniertem Verhalten. Bei einem Zugriff auf den Pointer wird nur der Inhalt des Speicherbereichs ausgelesen, nicht aber dessen Volatilität beachtet. Dem Compiler ist also nicht klar, dass der Inhalt *volatile* ist. Entsprechend könnte er die if-abfrage wegoptimieren. Richtig wäre den Zeiger auch als volatile zu qualifizieren.

```
1 static volatile int led = 0;
2 static volatile int *vol_led_p;
3 vol_led_p = &led;  //korrekte Referenzierung
4 if (*led_p == 1) //Wird nicht optimiert, da led_p volatile ist
5 {printf("Die LED ist an");} //Kann ausgeführt werden.
```

Listing 3: Richtige Referenzierung

### Volatile in Rust.

In Rust ist das Konzept der volatilen Objekte nicht zu finden. Es können also keine Objekte als *volatile* qualifiziert werden. Allerdings gibt es Möglichkeiten Code vor Optimierung zu schützen und volatile Operationen durchzuführen.

### Volatile Operationen mittels intrinsischer Funktion

In Rust gibt es die Möglichkeit, volatile Schreib und Lese-operationen mittels intrinsischer Funktion durchzuführen. Im Gegensatz zu Objekten, können diese Funktionen als volatile gekennzeichnet werden. Hierzu gibt es die Funktionen *std::ptr::read_ volatile* und *std::ptr::write_ volatile*. Volatile Funktionen werden vom Compiler nicht optimiert. Diese Funktionen erwarten einen Zeiger als Parameter und liefern den Wert der referenzierten Variablen als Rückgabe.

```
1 let mut led = 1 ;       //"volatile" Variable led
2 let led_p = &mut led;  //Zeiger auf led
3     if unsafe {std::ptr::read_volatile(led_p)} == 1 {  // Aufruf wird nicht optimiert
```

```
4     println!("Die LED ist an");}
```

Listing 4: intrinsische volatile Funktion

Die Funktionen sind als *unsafe* deklariert, da Rust die Integrität der übergebenen Referenz nicht garantieren kann. Dies ist Aufgabe des Programmierers.

**Atomare Datentypen und Funktionen.**

Der Zugriff über volatile Funktionen ist dann sinnvoll, wenn die Objekte zur Interaktion mit externer Hardware eingesetzt werden. Kommen Threads ins Spiel, kann es sinnvoll sein, atomare Datentypen und Zugriffe zu verwenden. Diese sichern, je nach Implementierung, den Code gegen ungewollte Optimierung und ermöglichen die Synchronisation der Zugriffe. Hierbei ist allerdings zu beachten, dass “atomar“ nicht mit *volatile* gleichzusetzen ist. Der Schutz vor ungewollter Optimierung wird primär durch das*Ordering* bestimmt und kann nicht garantiert werden.

```
1 use std::sync::atomic::{AtomicPtr, Ordering}; //Einbindung der Atomics
2 let mut led = 0; // Variable mit "volatilem" Inhalt
3 let atomic_led_p = AtomicPtr::new(&mut led); // Erstellung des atomaren Pointers
4
5 if unsafe {*atomic_led_p.load(Ordering::SeqCst)} == 1  //Dereferenzierung Und Zugriff
      auf led via .load. Wieder Unsafe
6 //(Ordering::SeqCst) gibt die Zugriffs und Optimierungsrestriktionen an
7 {println!("Die LED ist an");}
```

Listing 5: intrinsische volatile Funktion

**Fazit**

Die Problematik der EXP 32-C spielt in Rust keine Rolle mehr, da hier Objekte nicht volatile sein können. Allerdings können volatile Funktionen in Rust nur unsicher ausgeführt werden. Dadurch, dass Rust die Integrität der übergebenen Referenz nicht prüft, ergibt sich hier eine Sicherheitslücke und zusätzlicher Aufwand für die Entwickler. Ähnlich verhält es sich bei atomaren Typen und Funktionen. Hier ist zu beachten, dass sie nicht immer und sicher die ungewollte Optimierung unterbunden wird. Es muss also im Einzelfall geprüft werden welche Lösung sinnvoll ist.

## 2.3 EXP36-C

Nach der Regel EXP36-C des SEI CERT C Coding Standards sollten Pointer nicht in Pointertypen mit strengerem Alignment umgewandelt werden.

Das Alignment eines Typs gibt an wo im Speicher Objekte dieses Typs gespeichert werden dürfen. Objekte mit Alignment $n$ dürfen nur an Adressen gespeichert werden, die ein Mehrfaches von $n$ sind[2]. Die Möglichkeit für unterschiedlicher Alignments bei verschiedenen Speicherobjekttypen führt dazu, dass sich beim Casten von Pointern das Alignment des Pointers ändern kann. Falsch aligned sind Pointer dann, wenn ihre Adresse kein Mehrfaches des Alignments ihres Typen ist. Das Dereferenzieren eines solchen Pointers hat undefiniertes Verhalten und führt - abhängig von der Architektur - zum Absturz des Programms oder beeinträchtigter Performance.

### 2.3.1 Pointer-Misalignment in C

Der C Standard erlaubt es grundsätzlich Pointer von einem Typ zu einem anderen Typ umzuwandeln.

```
1    char *c = malloc(64);
2    int *ip = (int *) (c + 1);
3    i = *ip     // Versucht i von ungerader Adresse zu laden
```

Listing 6: Misalignment-Fehler beim Type Casten von Pointern

Wie in Listing 6 beispielhaft dargestellt entstehen Probleme unter anderem beim Versuch ein Integer von einer Adresse zu laden, die kein Mehrfaches von 4 ist. Aktuelle Versionen von GCC erkennen mit dem Flag $-$Wcast$-$align manche solcher Fehler.

```
1    char *c = malloc(64);
2    memcpy(&i, c + 1, sizeof(a));
```

Listing 7: Vermeiden von Misalignment-Fehlern durch memcpy()

Listing 7 zeigt wie sich Probleme im obigen Beispiel durch die Verwendung von memcpy() vermeiden lassen. Weiter erlaubt der C Standard jeden Objektpointer nach und von **void** $*$ zu casten.

```
1    int *loop_function(void *vp) {
2      /* ... */
3      return vp;
4    }
5
6    void func(char *cp) {
7      int *ip = loop_function(cp);
8      /* ... */
9    }
```

Listing 8: Misalignment-Fehler durch ungenaue Funktionsdeklaration

Zum Vermeiden des in Listing 8 dargestellten Fehlers gilt es relevante Funktionen - wenn möglich - so zu deklarieren, dass sie nur Pointer eines bestimmten Typs annehmen. Das obige Beispiel kompiliert mit GCC 4.8 auf Ubuntu Linux 14.04 ohne Warnung.

### 2.3.2 Pointer in Rust

In Rust werden zwei Arten von Pointern unterschieden:

- References

[2]Beispielsweise können Objekttypen mit 4-byte Alignment nur an den Adressen 0, 4, 8 etc. gespeichert werden, was unter anderem der Grund für Structure-Padding ist.

- Raw Pointer

References erlauben es auf einen Wert zu verweisen ohne dessen Ownership[3] zu übernehmen (Borrowing). Rust gestattet References grundsätzlich weder das Casten zu oder von Integern noch das Casten in einen anderen Typ. Rust's Compiler stellt sicher, dass References immer gültig sind.

Im Gegensatz zu References besitzen Raw Pointer keine Lebensdauer, können null sein oder zwischen Typen gecastet werden. Rust garantiert außerdem nicht, dass Raw Pointer auf gültigen Speicher zeigen.

```
1    let address = 0x012345usize;
2    let r = address as *const i32;
3
4    unsafe {
5        println!("r is: {}", *r);
6    }
```

Listing 9: Erstellen und Dereferenzieren eines Raw Pointers

Wie in Listing 9 gezeigt ist das Erstellen von Raw Pointern in Safe Rust[4] möglich, außerhalb von Unsafe-Blöcken können diese aber nicht dereferenziert werden.

### 2.3.3 Fazit

Zusammenfassend lässt sich sagen, dass Rust die Probleme von falschem Pointer-Alignment vollständig behebt, solange man sich in Safe Rust bewegt. In Unsafe Rust müssen dagegen in dieser Hinsicht die gleichen Regeln beachtet werden wie beim Schreiben in C. Da sich im Allgemeinen Pointer in C und Rust's Raw Pointer nahezu identisch verhalten sind die im Abschnitt 2.3.1 aufgeführten Hinweise gut auf das Programmieren in Unsafe Rust übertragbar.

[3] Rust managt Speicher durch ein Ownership-System, in dem jeder Wert immer genau einen Owner besitzt, der wiederum die Lebensdauer des Werts bestimmt.

[4] Rust beinhaltet eine sichere und eine unsichere Programmiersprache.

## 2.4 EXP37-C

Ob selbst erstellt oder aus fertigen Bibliotheken, Funktionen werden in allen Programmiersprachen gebraucht und verwendet. Dabei können Fehler gemacht werden, welche der Secure Coding Paragraph EXP37-C behandelt. Dieser Paragraph soll kurz, anhand von C Beispielen, erklärt werden und anschließend mit der Programmiersprache Rust verglichen werden.

### 2.4.1 Regel

Die Regel besagt wörtlich „Call functions with the correct number and type of arguments“. Das bedeutet genauer betrachtet für das Verwenden von Funktionen in C:

- Die Anzahl der Argumente muss mit der Funktionsdefinition übereinstimmen
- Die Datentypen der Argumente müssen mit der Funktionsdefinition übereinstimmen

### 2.4.2 Analyse in C

Das folgende Listing 10 zeigt ein kurzes Beispiel für den Aufruf einer Funktion mit einem falschen Datentyp.

```
1  int teile(int divident, int divisor)
2  {
3      return (divident / divisor);
4  }
5
6  int main()
7  {
8      int a = 10;
9      float b = 0.99;
10     a = teile(a,b); //2. Argument stimmt nicht mit Definition ueberein
11     return 0;
12 }
```

Listing 10: Übergabe eines falschen Datentypen in C

Die Funktion *teile()* teilt zwei Zahlen. Definiert ist diese mit Übergabeparametern des Types Integer. Jedoch wird der Funktion als zweites Argument eine Gleitkommazahl übergeben. Der Compiler gibt in diesem Fall keine Warnung aus, wodurch unerwünschte Effekte, wie eine falsche Berechnung oder ein Programmcrash, auftreten können. In weiteren Testversuchen stellte sich heraus, dass der Compiler falsche Datentypen erkennen kann. Jedoch wird bestenfalls eine Warnung ausgegeben. Vorallem bei der Verwendung von Pointern mahnt der Compiler einen falschen Datentyp oft an.

Der zweite Aspekt dieser Regel bezieht sich auf die Anzahl an Parametern, welche Funktionen übergeben werden können. Werden zu viele oder zu wenige Argumente mitgegeben erkennt der Compiler das und gibt entsprechende Fehlermeldungen aus, falls die Funktionsdeklaration korrekt und eindeutig ist! Ein kurzes Beispiel ist im Listing 11 zu erkennen. Die Funktion *addiere()* bildet die Summe aus drei Parametern. Wird nicht die genaue Anzahl an Parametern angegeben, wie in Zeile sieben und acht, kann das Programm nicht gebaut werden und Fehlermeldungen werden angezeigt.

```
1  int addiere(int a, int b, int c)
2  {
3      return a+b+c;
4  }
5  int main()
6  {
7      addiere(1,2); //zu wenig Parameter
8      addiere(1,2,3,4); //zu viele Parameter
9      addiere(1,2,3);
```

```
10      return 0;
11  }
```

Listing 11: Falsche Anzahl an Funktionsparametern

Ein Spezialfall in C sind die Funktionen mit Variablen Argumenten (eng. variadic functions wie zum Beispiel *printf()*). Diese sind zwar sehr nützlich, aber auch sehr gefährlich, aufgrund der Komplexität und Unberechenbarkeit während der Laufzeit. Somit sollten diese weitgehend vermieden werden.

### 2.4.3 Analyse in Rust

Beim Betrachten des falschen Datentypes bei Funktionsaufrufen, ist der Rustcompiler sehr strikt und meldet Umstände direkt als Fehler. Somit ist wird der Entwickler direkt gezwungen sich Gedanken über Typecast oder ähnliches zu machen.

Zu viele oder zu wenige Argumente beim Funktionsaufruf werden korrekt als Fehler erkannt. Rust, wie auch C, sind in diesem Gebiet sehr strikt. Bezüglich der Übergabe optionaler Argumente soll hier auf Kapitel EXP47-C verwiesen werden, welches diesen Aspekt besser betrachtet.

### 2.4.4 Fazit

Zusammengefasst muss sich bei C immer vergewissert werden, dass der Funktionsaufruf korrekt ist. Aufgrund der Inkonsistenz in der Fehlererkennung durch den Compiler kann sich nicht auf diesen Verlassen werden. Zudem ist eine korrekte Definition oder Deklaration der Funktion vorausgesetzt (siehe SEI CERT C Coding Standard DCL40-C).

Rust ist in diesen Aspekten einheitlicher. Ein falscher Datentyp wird genauso als Fehler gemeldet wie eine falsche Anzahl an Argumenten. Dies kann Schwachstellen verhindern aber auch die Geduld des Entwicklers auf die Probe stellen.

## 2.5 EXP39-C

EXP39-C behandelt den Zugriff auf Variablen durch Pointer eines inkompatiblen Typs. Dieser kann sowohl beabsichtigt erfolgen, um beispielsweise die dadurch ausgelösten, jedoch nicht unter allen Umständen garantierten, Nebenwirkungen auszunutzen, als auch unbeabsicht als Folge einer Speicherreallokation entstehen.

### 2.5.1 Problematischer Code

Beim nachfolgenden Beispiel besteht das Problem darin, dass auf `b->a` zugegriffen wird, bevor dieses neu initialisiert worden ist. Wäre die Anordnung der Werte innerhalb des structs `A` eine andere könnte der Code vielleicht je nach Architektur bedingt durch Padding noch einen sinnvollen Wert hervorbringen, so jedoch umfasst der Speicher an dieser Stelle mitunter die kombinierten Werte der chars `a` und `b`.

```
1 struct A { char a; char b; int c;};
2 struct B { int a; char b; char c;};
3
4 struct A *a = (struct A *)malloc(sizeof(struct A));
5
6 /* initialization of a */
7 struct B *b = (struct B *)realloc(a, sizeof(struct B));
8
9 printf("%d", b->a);
```

Listing 12: C safe

### 2.5.2 Konforme Ansätze

Um die oben genannten Problematiken zu umgehen, gibt es - situationsbedingt - unterschiedliche Lösungsansätze.
Sollen beispielsweise Nebenwirkungen des Zugriffs durch einen inkompatiblen Typ ausgenutzt werden, ist es sinnvoller stattdessen eventuell vorhandene Bibliotheken zur Herbeiführung des selben Effektes einzusetzen. Sollten entsprechende Bibliotheken nicht vorhanden sein, muss die gewünschte Funktionalität durch eigens definierte Funktionen erzeugt werden, da die Nebenwirkungen aufgrund ihrer Natur als undefiniertes Verhalten keine verlässliche Grundlage bilden können.
Die Umgehung der möglichen Fehler bei der Wiederverwendung von bereits genutztem Speicher gestaltet sich dahingehend einfacher, dass zur Vermeidung der Entstehung fehlerhafter Werte lediglich der reallokierte Speicher vor dessen Nutzung komplett überschrieben und anschließend neu initialisiert werden muss. Dies kann beispielsweise durch die Nutzung von `memset` erreicht werden. Code-Beispiel:

```
1 struct A { char a; char b; int c;};
2 struct B { int a; char b; char c;};
3
4 struct A *a = (struct A *)malloc(sizeof(struct A));
5
6 /* logic */
7
8 struct B *b = (struct B *)realloc(a, sizeof(struct B));
9 memset(b, 0, sizeof(struct B));
```

Listing 13: C safe

### 2.5.3 Unzulässige Zugriffe in Rust

Wenngleich auch schwerer zu reproduzieren und damit wahrscheinlich auch weniger häufig anzutreffen, besteht auch in Rust die Möglichkeit auf Speicher durch einen Pointer eines inkompatiblen Typs zuzugreifen. Normalerweise werden typfremde Zuweisungen bereits durch den Compiler verhindert, jedoch ist es möglich diesen eingebauten Mechanismus durch die Verwendung von Raw-Pointern zu umgehen: Hierbei wird die Funktion `transmute` aus `std::mem` dazu verwendet um den ursprünglichen Wert eines Typen als den eines anderen zu reinterpretieren. Die notwendigen Vorraussetzungen hierfür sind das Ausführen von `transmute` innerhalb eines `unsafe`-Blocks, sowie eine einheitliche Speichergröße beider Typen. Die Funktion wird innerhalb der Dokumentation der Sprache wörtlich als 'incredibly unsafe' bezeichnet und eine allgemeine Verwendung wird nicht empfohlen.[5]
Code-Beispiel:

```
1  fn main() {
2      let int = 15;
3
4      let float: *mut f32 = unsafe {
5          std::mem::transmute(&int)
6      };
7
8      unsafe {
9        println!("{}", *int);
10       println!("{}", *float);
11     }
12 }
```

Listing 14: Rust example

```
1 15
2 0.000000000000000000000000000000000000000000021
```

Listing 15: Result

### 2.5.4 Fazit

Die wesentlich strikter ausfallenden Compilervorschriften seitens Rusts in Verbindung mit der Nutzung von Referenzen an Stelle von Pointern sind grundlegend durchaus dazu geeignet Fehler auf Basis inkompatibler Typenzugriffe zu verhindern. Die Folge hiervon ist ein in der Regel stabilerer und sichererer Code. Dieser Effekt erstreckt sich allerdings lediglich auf Code-Bereiche, die sich außerhalb eines 'unsafe'-Blocks befinden. Befindet sich der Code in einem Solchen können jedoch ähnliche Probleme wie bei der Programmierung in C auftreten, wodurch dort mit entsprechender Sorgfalt vorgegangen werden muss.

[5] https://doc.rust-lang.org/std/mem/fn.transmute.html

## 2.6 EXP40-C

EXP40-C beschreibt die Regel, dass als konstant definierte Objekte nicht modifiziert werden sollen. Der C Standard, 6.7.3, Paragraph 6 beschreibt, dass die Veränderung eines als konstant definierten Wertes durch die Veränderung eines als nicht konstant definierten Wertes undefiniertes Verhalten ist. In anderen Worten, wenn eine Konstante direkt von einer Variable abhängt und diese Variable verändert wird, ist die undefiniertes Verhalten. Das Risiko wird mit geringer Schwere und Wahrscheinlichkeit bewertet. Der Aufwand dies zu beheben ist als Moderat gelistet.

### C

Dieses Verhalten kann in C beispielsweise dadurch verursacht werden, wenn ein konstanter Pointer von einer Variable abhängig ist oder ein variabler Pointer auf eine Konstante verweist. Wenn in Fall 1 die Variable verändert wird auf die der konstante Pointer verweist, wird dadurch implizit der konstante Pointer verändert. Dies geschieht weil der Pointer sich eigentlich nicht verändert hat, jedoch der Wert auf welchen er verweist. Wenn in Fall 2 der variable Pointer welche auf die Konstante verweist dereferenziert wird und dieser Wert verändert wird, verändert sich dadurch implizit die Konstante. Dies geschieht weil der Pointer dadurch den Wert der Konstante im Speicher ändert. Dieses Verhalten kann in Listing 2.31.3.1 beobachtet werden.

```
1 //Fall 1 - Konstanter Pointer verweist auf Variable
2 int i = 42; //Variable
3 const int *ip = &i; //Konstanter Pointer
4 i = 15; //Durch Veränderung der Variable wird implizit der Pointer verändert
5 //Fall 2 - Variabler Pointer verweist auf Konstante
6 const int j = 42; //Konstante
7 int *jp = &j; //Variabler Pointer
8 *jp = 15; //Durch Veränderung des Pointers wird implizit die Konstante verändert
```

Listing 16: C Constraint Violation

Die Regel EXP40-C empfiehlt, dass Werte welche nicht verändert werden können sollen nicht als Konstante definiert werden. Dieses Problem kann ebenfalls dadurch behoben werden, dass Konstanten nicht von Variablen abhängig sind und Pointer welche auf Konstanten verweisen ebenfalls als Konstante definiert werden. Diese Lösung ist in Listing 17 ersichtlich.

```
1 //Fall 1 - Konstanter Pointer verweist auf Konstante
2 const int i = 42; //Konstante
3 const int *ip = &i; //Konstanter Pointer
4 i = 15; //Kompiler Fehler
5 //Fall 2 - Konstanter Pointer verweist auf Konstante
6 const int j = 42; //Konstante
7 const int *jp = &j; //Konstanter Pointer
8 *jp = 15; //Kompiler Fehler
```

Listing 17: C Valid

### Rust

In Rust werden sogenannte References verwendet um auf andere Objekte zu verweise. In Rust bedeutet dies, dass sich die Referenzen das Ownership des Wertes ausleihen. Ownership bedeutet, dass Speicher freigegeben wird sobald der Besitzer out-of-scope geht. Ausgeliehene Objekte können nicht verändert werden.[6] Dies wirkt sich auf die im C Teil beschriebenen Probleme aus. Wenn wie in Fall 1 ein konstante Referenz auf eine Variable verweist, ist dies möglich. Jedoch lässt der Kompliler

[6] https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

es nicht zu nach der Referenzierung die Variable zu verändern, weil ausgeliehene Werte nicht verändert werden können. Wenn wie in Fall 2 eine variable Referenz versucht eine Variable auszuleihen, wird dies vom Kompiler nicht zugelassen. Diese Beispiele können in Listing 18 eingesehen werden.

```
1 //Fall 1 - Konstante Referenz verweist auf Variable
2 let mut i = 42; //Variable
3 let ip = &i; // Konstante Referenz
4 i = 15; //Kompiler Fehler, sobald konstante Referenz aufgerufen wird
5 //Fall 2 - Variable Referenz verweist auf Konstante
6 let j = 42; // Konstante
7 let jp = &mut x; //Kompiler Fehler
```

Listing 18: Rust Safe

Der Unsafe Modus von Rust erlaubt es die Speicher Sicherheit weitergehen selbst zu verwalten als in regulärem Rust. Dadurch werden jedoch nicht alle Sicherheitsüberprüfungen von Rust deaktiviert. Wenn wie in Fall 1 versucht wird eine konstante Referenz auf eine Variable verweisen zu lassen, ist dies erlaubt. Eine Veränderung der Variable verändert ebenfalls implizit die Konstante Referenz, weil diese selbst nicht verändert wurde. Wenn jedoch wie in Fall 2 versucht wird eine variable Referenz auf eine Konstante verweisen zu lassen, lässt dies der Kompiler nicht zu. Dies ist bedingt dadurch, dass eine konstante Referenz auch im unsafe Modus nicht auf eine Variable verweisen darf. Beide Beispiele können in Listing 19 eingesehen werden.

```
1 unsafe{
2 //Fall 1 - Konstante Referenz verweist auf Variable
3 let mut i = 42; //Variable
4 let ip = &i as *const i32; //Konstante Referenz
5 i = 15; //Veränderung der Variable verändert implizit konstante Referenz
6 //Fall 2 - Variable Referenz verweist auf Konstante
7 let j = 42; //Konstante
8 let jp = &mut j; //Kompiler Fehler
9 }
```

Listing 19: Rust Unsafe

Zusammenfassen lässt sich also sagen, das Rust die Probleme der Regel EXP40-C im regulären Modus komplett behebt. Es ist dem Entwickler jedoch freigestellt diese teilweise durch den unsafe Modus zu deaktivieren wenn es für nötig erachtet wird.

## 2.7 EXP43-C

Bei EXP43-C geht es darum, dass eine fehlerhafte Verwendung des restrict-qualifiers[7] zu undefiniertem Verhalten führen kann. Das undefinierte Verhalten äußert sich als Inkonsistenzen bei der Datenintegrität.

### 2.7.1 Restrict-qualified Pointer in C

Die restrict Anweisung ist eine Information vom Entwickler an den Compiler.[8][9] Damit wird dem Compiler mit geteilt das Pointer die den gleichen Typ haben **keine Aliase** von einander sind und das alle Lese und Schreib Zugriffe, auf einen Speicherbereich, nur mit diesem Pointer stattfinden, solange dieser existiert. In vielen Fällen kann zur Compilezeit nicht bestimmt werden ob weitere Aliase zur Laufzeit auf den gleichen Speicherbereich existieren werden. Diese Information wird allerdings benötigt um signifikate Optimierungen zu ermöglichen.

- Eliminiert Lese Zugriffe, für Werte die nicht verändert wurden.
- Eliminiert Schreib Zugriffen, da niemals vor dem nächsten schreiben gelesen wird.
- Bessere Befehlsfolge, da mehr Befehle unabhängig von einander sind.

Das Code Snippet aus Listing 20 wurde einmal mit und einmal ohne restrict-qualified Pointer kompiliert. Ohne restrict, in Listing 21, müssen die Befehle genau in der gleichen Reihenfolge wie in Listing 20 abgearbeitet werden. Da `velocity_x` und `position_x` Pointer vom selben Typ sind, kann nicht garantiert werden das es sich dabei um keine Aliase hält. Dies könnte dazu führen das ein neuer Wert in `velocity_x` auch zu einem neuen Wert in `position_x` führen kann. In Listing 22 hingegen konnte der Compiler die Reihenfolge der Lese und Schreib Zugriffe optimieren, da Ihm bekannt war, das beide Speicherbereiche unabhängig von einander sind.

```
1  void  move( vector3* /*restrict*/ velocity,
2              vector3* /*restrict*/ position,
3              vector3* /*restrict*/ acceleration,
4              float time_step)
5  {
6    for (size_t i=0;i<count;i++) {
7      velocity[i].x += acceleration[i].x * time_step;
8      velocity[i].y += acceleration[i].y * time_step;
9      velocity[i].z += acceleration[i].z * time_step;
10     position[i].x += velocity[i].x     * time_step;
11     position[i].y += velocity[i].y     * time_step;
12     position[i].z += velocity[i].z     * time_step;
13   }
14 }
```

Listing 20: Restrict C Snippet

```
1  stfsx   4,8,3      # Store velocity_x
2  stfs    3,4(9)     # Store velocity_y
3  stfs    2,8(9)     # Store velocity_z
4  lfsx    11,8,4     # Load position_x
5  lfs     10,4(11)   # Load position_y
6  lfs     9,8(11)    # Load position_z
```

Listing 21: Ohne restrict (PowerPC)

```
1  lfsx    11,8,4     # Load position_x
2  lfs     10,4(11)   # Load position_y
3  lfs     9,8(11)    # Load position_z
4  stfsx   4,8,3      # Store velocity_x
5  stfs    3,4(9)     # Store velocity_y
6  stfs    2,8(9)     # Store velocity_z
```

Listing 22: Mit restrict (PowerPC)

[7] https://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html
[8] Der Compiler kann diese Information für Optimierungen nutzen, muss aber nicht
[9] Zum kompilieren sollten die Flags `-fstrict-aliasing -std=c99` verwendet werden

### 2.7.2 Risikobewertung

Wird ein restrict-qualified Pointer verwendet, aber dennoch über ein Alias auf den Speicher zugegriffen, kann eine der in Kapitel 2.7.1 beschriebenen Optimierungen zu falschen Daten führen. Da es sich hierbei um Information vom Entwickler an den Compiler handelte, kann auch meist nicht mit Warnungen vom Compiler gerechnet werden. Restrict wird selten, aber häufig falsch verwendet. Die produzierten Fehler sind meist schwer zu finden. Der Fehler kann behoben werden indem der restrict-qualifier entfernt wird.

### 2.7.3 Aliase in Rust

In Rust sieht das Referenzierungsmodel[10] vor das es geteilte und veränderbare Referenzen gibt. Für veränderbare Referenzen dürfen keine Aliase existiren. In Rust wird über `&mut` ausgeschlossen das es weitere Aliase[11] gibt, oder sich die Speicher überlappen. Dadurch können die Optimierungen ohne weitere Information angewendet werden. In Listing 23 muss deswegen der Wert in `input` nicht erneut geladen werden nachdem in `output` geschrieben wurde. Dies ist aber bei raw pointers und UnsafeCell nicht möglich.

```
1 fn compute(input: &u32, output: &mut u32) {
2     if *input > 10 {     // Load input value
3         *output = 1;     // Store value that has same type as input
4     }
5     if *input > 5 {      // Use the value from cache, no need to reload
6         *output *= 2;
7     }
8 }
```

Listing 23: Rust Mutable Reference Snippet

### 2.7.4 Fazit

Durch das strikte Ownership Modell in Rust kann von vornherein ausgeschlossen werden das weitere Aliase vorhanden sind und deswegen der Code sicher optimiert werden. Änderungen in diesem Verhalten müssen vom Entwickler explizit verlangt werden.

[10] https://doc.rust-lang.org/nomicon/references.html
[11] https://doc.rust-lang.org/nomicon/aliasing.html

## 2.8 EXP44-C

Dieses Kapitel thematisiert die Coding Rule EXP44-C. Diese soll anhand einer Gegenüberstellung in den Programmiersprachen C und Rust näher erläutert werden. Die Regel besagt, dass einige Operatoren ihren Fokus ausschließlich auf die Typinformation legen, die der übergebene Operand zur Verfügung stellt.

### 2.8.1 Analyse der Regel in der Programmiersprache C

In der Programmiersprache C werden Nebeneffekte, die gewissen Operatoren mit übergeben werden, ignoriert und nicht ausgeführt. In C sind „Sizeof“, „_Alignof“ und „_Generic“ von dieser Coding Rule betroffen.

- Sizeof: Hier werden Speichergrößen in Byte ermittelt.
- Alignof: Statt der Speichergröße wird hier die Ausrichtung in Bytes angegeben.
- Generic: Wenn ein Datentyp nicht feststeht, wird die Auswahl von mehreren Ausdrücken zur Kompilierzeit ermöglicht. Beispielsweise kann in der Präprozessoranweisung ein Makro definiert werden, das abhängig vom übergebenen Typen, spezifische Werte zuweist.

Eine Ausnahme in dieser Coding Rule kann Auftreten wenn es sich bei dem Operanden um ein Array mit variabler Länge handelt. Hier kann es unter gewissen Voraussetzungen zur Auswertung eines Nebeneffektes kommen. [12]

### 2.8.2 Problem und Lösung in C

```c
#include <stdio.h>
#include <math.h>

void not_compliant(){
  double a = 4.0;
  sizeof(a = pow(a, 2.0));
  printf("Not Compliant Wert von a: %lf\n", a);
}
```

Listing 24: Noncompliant in C

In Listing 24 wird zunächst eine Variable mit dem Wert 4.0 deklariert. Daraufhin soll mithilfe der pow-Funktion der Wert von „a“ mit dem zweiten Argument 2.0 potenziert werden. Im Kontext des Programmablaufs wird erwartet, dass „a“ als Ergebnis den Wert 16.0 zurückliefert. Mithilfe des sizeof-Operators soll dann die Speichergröße des ermittelten Wertes ausgegeben werden. Die Problematik: Der Operand wird nicht ausgewertet, da der sizeof-Operator die Potenzierung ignoriert. Die pow-Funktion wird als Nebeneffekt interpretiert und es erfolgt die Aufhebung der Variablenänderung.

Das Problem lässt sich folgendermaßen umgehen:

```c
#include <stdio.h>
#include <math.h>

void compliant(){
  double a = 4.0;
  a = pow(a, 2.0);
  sizeof(a);
  printf("Compliant Wert von a: %lf\n\n", a);
}
```

Listing 25: Compliant in C

[12] https://de.wikibooks.org/wiki/C-Programmierung:_Ausdrücke_und_Operatoren

In Listing 25 wird ersichtlich, dass die Variable separat in der Funktion aufgerufen und potenziert wird. Die Verwendung des Operators erfolgt erst hinterher. Dadurch wird eine erfolgreiche Auswertung des Operanden gewährleistet. Man erhält für „a“ schlussendlich den Wert 16.0.

### 2.8.3 Lösung durch Rust

Anders als in C werden in Rust Inkrement- und Dekrement-Operatoren nicht unterstützt. Diese komplexen Operatoren erfordern Kenntnisse der Bewertungsreihenfolge und verursachen häufig schwer erkennbare und nicht eindeutige Fehlerquellen. Somit fallen potentielle Gefahren von Beginn an weg. [13]

```
1 use std::mem::size_of_val;
2 fn main() {
3   let mut a: f64 = 4.0;
4   size_of_val(&(a = a.powf(2.0)));
5   println!("Wert von a: {}", a);
6   }
```

Listing 26: Codebeispiel in Rust

In Rust muss die Deklaration des sizeof-Operators in Kombination mit einem Pointer erfolgen. Aus Listing 26 erschließt sich, dass die Evaluierung des Ausdrucks erfolgreich abgeschlossen wird. Hier wird der Nebeneffekt der Funktion während der size_of_val-Operation berücksichtigt. Der Wert 16.0 wird ausgegeben. In allen Fällen beträgt die Größe des Datentyps 8 Byte. [14]

### 2.8.4 Fazit

Bei der Verwendung dieser Operatoren sollte schlussendlich auf Operanden mit Nebeneffekten verzichtet werden. Personen, die am Programm arbeiten, könnten fälschlicherweise davon ausgehen, dass die Nebeneffekte ausgewertet werden. Ein Qualitätsverlust der Software könnte daraus resultieren. Bei der Programmierung mit Rust ist dies nicht der Fall. Die Operationen berücksichtigen die Nebeneffekte der Operanden. Hinzukommt das irreführende Inkrement- und Dekrement-Operatoren nicht unterstützt werden.

[13] https://doc.rust-lang.org/1.2.0/complement-design-faq.html#why-no—x-or-x++?
[14] https://doc.rust-lang.org/book/

## 2.9 EXP46-C

Die Coding Rule EXP46-C besagt, dass man keinen bitweisen Operator mit einem boolean-ähnlichem Operanden benutzen soll.

### 2.9.1 Erklärung der Grundbegriffe

Ein bitweiser Operator ist ein Operator, der beispielsweise auf eine oder mehrere Zahlen angewandt wird. Mit diesem bitweisen Operator kann direkt auf die Binärdarstellung einer Zahl zugegriffen werden. Die wichtigsten bitweisen Operatoren in C und Rust sind AND, OR, XOR, Left Shift, Right Shift und NOT. In Listing 27 wird das AND benutzt, mit dem nur die Bits gleich 1 werden, an denen beide Bit-Reihenfolgen der Variablen eine 1 gesetzt haben. [15]

Logische Operatoren werden meist in Bedingungen für beispielsweise eine if-Anweisung oder while-Schleife angewandt. Sie überprüfen, ob der Wert von einem oder mehreren Operanden die Werte true oder false liefern. Logische Operatoren in C und Rust sind „&&“, „||“ und „!“. Das „&&“ in Listing 28 sorgt dafür, dass eine Bedingung dann true wird, wenn beide Werte nicht null sind. Außerdem trifft das Gegenteil einer Bedingung ein, wenn vor dieser ein „!“, wie in Listing 28 zu sehen, ist. [16]

### 2.9.2 Beschreibung des Problems und Lösungswegs in C

```c
#include <stdio.h>
int main()
{
  int zahl1 = 321;
  int zahl2 = 3;
  if(!(zahl1 & (zahl2 == 0))){
    printf("unsicher");
  }
    return 0;
}
```

Listing 27: unsicherer Ansatz in C

Die if-Anweisung in Listing 27 erwartet den Wert 0 (false) oder nicht 0 (true), damit die print-Anweisung ausgeführt werden kann. In der if-Anweisung sieht man, dass „zahl2 == 0“, als Erstes ausgeführt wird, da diese in Klammern steht. Da die Zahl 3 ungleich 0 ist, ist das Ergebnis von „zahl2 == 0“ gleich 0. Danach wird die Variable zahl1 mit dem Wert 0 konjugiert. Und genau hier liegt das Problem, da in C Integer für true und false stehen, werden die beiden Werte 321 und 0 konjugiert. Die Negation von 0 ergibt dadurch 1 (true). Die Lösung für das Problem lautet folgendermaßen:

```c
#include <stdio.h>
int main()
{
  int zahl1 = 321;
  int zahl2 = 3;
  if(!(zahl1 && (zahl2 == 0))){
    printf("sicher");
  }
    return 0;
}
```

Listing 28: sicherer Ansatz in C

[15] https://www.tutorialspoint.com/cprogramming/c_bitwise_operators.htm
[16] https://www.tutorialspoint.com/cprogramming/c_logical_operators.htm

Wie man in Listing 28 sehen kann, wird das logische „&&“ benutzt, damit man nur noch logische Operationen in der if-Abfrage durchführt. Dadurch vergleicht man richtigerweise, ob die jeweiligen Werte true oder false sind.

### 2.9.3 Risikobewertung

Die Schwere des Riskograds ist gering, da das Programm trotzdem durchlaufen kann, jedoch logisch nicht ganz richtig ist. Dieser Fehler kann sehr einfach beseitigt werden, indem man entweder gewollte bitweise Operationen in Klammern aufführt oder mit logischen Operatoren ersetzt. Es kommt außerdem zu erhöhten Laufzeiten, da bitweise Operationen meist aufwändiger sein können. Beim logischen „||“ beispielsweise würde die if-Anweisung bereits true werden, nachdem der erste Teil der if-Anweisung true wäre.

### 2.9.4 Wie sieht das Ganze in der Programmiersprache Rust aus?

In Rust gibt es, anders wie in C, den Typen Boolean. Deshalb gibt es auch keine boolean-ähnlichen Operanden. Wenn man den sicheren Code aus Listing 28 in Rust umschreibt, erhält man folgenden Code:

```
1 fn main() {
2   let zahl1 = 321;
3   let zahl2 = 3;
4   if !(zahl1 && (zahl2 == 0)) {
5     println!("test");
6   }
7 }
```

Listing 29: Codebeispiel in Rust

In Listing 29 kommt beim Ausführen des Codes die Fehlermeldung, dass ein Wert des Typs Boolean nicht mit einem Integer verglichen werden kann. Wenn hier ein „&“ anstelle von „&&“ stehen würde, käme es ebenfalls zu einer Fehlermeldung, da man in Rust keine Integer-Werte mit Boolean-Werten konjugieren kann.

### 2.9.5 Fazit

C lässt es zu, dass ein Integer mit einem Boolean konjugiert werden kann, da es keinen expliziten Datentypen für Booleans in C gibt. In Rust unterscheidet man genau zwischen Boolean und Integer, weshalb Variablen der beiden Datentypen nicht verglichen werden können.

## 2.10 EXP47-C

Die Regel EXP47-C weist darauf hin, dass beim Verwenden von variablen Argumentenlisten (VarArgs) in der Programmiersprache C die ”default argument promotions” zu berücksichtigen sind. Im Folgenden wird dargelegt, wie die Regel auf C-Code anwendet wird. Außerdem wird untersucht, ob die Regel auch bei der Programmierung mit Rust relevant ist.

### C

VarArgs erlauben das Übergeben beliebig vieler Argumente an eine Funktion. Eine Besonderheit in C ist die Tatsache, dass der Typ der Argumente im VarArg nicht spezifiziert ist. Das heißt: Es können Werte beliebigen Typs übergeben werden. Beim Lesen der Werte aus dem VarArg sollte man aufgrund der ”default argument promotions” Vorsicht walten lassen. Sie zwingen den Entwickler jedes Argument, dessen Typ <=32 Bit ist, als `int` und jedes andere Argument als `double` auszulesen [17]. Erst danach kann er das Argument zum ursprünglichen Typ casten. Missachtet man diese Spezifikation, so ist das Verhalten undefiniert und davon abhängig, ob der geschriebene Wert als `int` oder `double` interpretiert werden kann. Falls ja wird der Wert falsch interpretiert, andernfalls sind Laufzeitfehler möglich. (Ziel der ”default argument promotions” ist vermutlich ein Performancegewinn bei der Adressierung der Parameter.[18])

```c
void func(size_t num_vargs, ...) {
  va_list ap;
  va_start(ap, num_vargs);
  for (int i = 0; i < num_vargs; i++) {
    // Noncompliant access to VarArg
    // char c = va_arg(ap, char); // Illegal instruction error (gcc 5.4.1 c99)
    // Compliant access to VarArg
    char c = (char) va_arg(ap, int);
    printf("%c\r\n", c);
  }
  va_end(ap);
}

int main() {
  char c0 = 'a'; // Char (8 bit) will be stored as int
  char c1 = 'b';
  func(2, c0, c1);
  return 0;
}
```

Listing 30: C Code

Die, im Listing 30 definierte, `func`-Funktion erhält als zweiten Parameter so viele Zeichen, wie im ersten Parameter angegeben. Beim Ausführen der Funktion werden die übergebenen Zeichen zeilenweise auf die Konsole geschrieben. Die mit ”`Noncompliant access to VarArg`” betitelte Zeile verfolgt dabei einen naiven Ansatz und berücksichtigt die Regel nicht. Ihre Ausführung endet mit einem Laufzeitfehler (verwendeter Compiler: gcc 5.4.1 c99). Die mit ”`Compliant access to VarArg`” betitelte Zeile hingegen implementiert die Regel richtig.

[17] C language standard (ISO/IEC 9899:1999), Abschnitt 6.5.2.2, Punkt 6
[18] https://stackoverflow.com/questions/4666598/self-promoting-data-types

### Rust

Im Standard von Rust sind keine variablen Argumentenlisten definiert. Um eine beliebige Anzahl an Parametern zu verarbeiten, greifen Funktionen der Rust Standard-Bibliotheken (wie zum Beispiel `std::print`) auf Makros zurück. ”Default argument promotions”, oder ähnliche Mechanismen, sind kein Teil der Makro-Implementierung von Rust. Der Typ der Argumente, die einem Makro übergeben werden, wird also nicht verändert.

Trotz dieses Vorteils eines Rust-Makros gegenüber einer C-Funktion mit VarArg darf man nicht vergessen, dass die Verwendung von Makros die Größe des compilierten Programms erhöht. Kollidiert dieser Nachteil mit den Anforderungen an das Programm, so muss man sich mit dem Überladen von Funktionen behelfen.

### Fazit

In Listing 31 wird dieselbe Funktion implementiert wie in Listing 30. Vergleicht man beide Implementierungen so fällt auf, dass alle folgenden Aktionen in C vom Rust-`repetition`-Operator implementiert werden:

- VarArg initialisieren
- Iterieren über alle Elemente des VarArgs
- Elemente des VarArgs type-casten
- VarArg deinitialisieren

**In Bezug auf die Regel EXP47-C gilt es festzuhalten, dass die Verwendung von Rust den Overhead beim Einsatz von VarArgs minimiert und die Codequalität erhöht.**

```
1  macro_rules! func(
2      // This rule matches a list of comma-separated expressions
3      // (where: expression == argument)
4      ($($argument:expr),*) => {{
5          $( // The following two lines will be repeated for each expression
6              let c: char = $argument;
7              println!("{}", c);
8          )*
9      }};
10 );
11
12 fn main() {
13     let c0: char = 'a';
14     let c1: char = 'b';
15     func!(c0, c1);
16 }
```

Listing 31: Rust Code

## 2.11 INT30-C

Mit diesem Kapitel wird auf die Secure Coding-Regel INT30-C eingegangen, welche besagt, dass sichergestellt werden muss, dass Operationen mit Variablen des Datentyps *unsigned integer* nicht überlaufen.

Zunächst wird hier der Ursprung der Problematik und der Umgang mit dem resultierenden Verhalten in den Programmiersprachen C und Rust dargestellt. Zum Abschluss steht ein Fazit, welches einen Vergleich der beiden Sprachen mit der genannten Thematik zieht.

### 2.11.1 Problemstellung

Bei der Darstellung einer *unsigned integer* Variablen sind nur positive Zahlen und Null möglich. Hier können am Beispiel einer 8-Bit Zahl die Werte 2^8 - 1 (255) bis 0 angezeigt werden. Wird während arithmetischen Operationen nur innerhalb dieses Wertebereichs gearbeitet, so kommt es zu keinen Fehlern. Beim Versuch der Darstellung einer Zahl größer 255 oder kleiner 0, kommt es jedoch zum genannten Fehlerfall, dem Wrappen der Variablen.

### 2.11.2 Programmiersprache C

Im folgenden Programmcode kommt es zu zwei Überläufen der Variablen. Bei der Subtraktion wird der Wertebereich unterschritten, bei der Multiplikation überschritten. In beiden Fällen kommt es zum Wrappen des Wertes.

```
1 #include <stdint.h>
2 int main() {
3     uint8_t ui8Var = 0;
4
5     ui8Var--; //Dekrementieren der Zahl auf -1 entspricht 255
6     ui8Var *= 3; //Multiplikation mit 3 -> ui8Var enthaelt 253
7 }
```

Listing 32: Wrappen einer 8-Bit unsigned integer Variablen

Um die beschriebene Problematik zu lösen, muss entweder vor oder nach den Operationen überprüft werden, ob die Variable den gültigen Wertebereich verlassen hat [19], da es keine andere Möglichkeit gibt, dies aus Compiler-Sicht zu bemerken.

### 2.11.3 Programmiersprache Rust

Beim Kompilieren des Rust Programmcodes muss zwischen zwei Modi, dem Debug und Release Modus, unterschieden werden. Diese Profile enthalten unterschiedliche Compiler-Einstellungen, welche für den entsprechenden Verwendungszweck optimiert sind. Dies wird bei der Ausführung des untenstehenden Codes deutlich, welcher mit dem Entwicklungs-Modus kompiliert wurde.

```
1 fn main() {
2     let mut ui8Var: u8 = 1;
3
4     ui8Var -= 3;
5     //thread 'main' panicked at 'attempt to subtract with overflow'
6 }
```

Listing 33: Abbruch durch Wrappen einer 8-Bit unsigned int Variablen

[19] https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap, Addition

Hier kommt es, wie bereits im Listing vermerkt, zur Beendigung des Programms durch eine *panic*. Grund hierfür ist das Überlaufen der Variable *ui8Var*. Der fehlerbedingte Abbruch des Programms resultiert aus dem Mechanismus zur Überprüfung des Wrappens des Datentyps, welcher während der Kompilierung durch das Debug Profil hinzugefügt wurde.

Da bei einigen Implementierungen das Überlaufen einer Variablen beabsichtigt ist, gibt es während des Entwicklungsprozesses eine Möglichkeit, ohne Änderung der Compiler-Einstellungen, ein Beenden des Programms zu verhindern.

```
1 fn main() {
2     use std::num::Wrapping;
3
4     let mut ui32_zero = Wrapping(0u32);
5     let ui32_one: u32 = 1;
6
7     ui32_zero -= Wrapping(ui32_one);
8 }
```

Listing 34: Verwendung des Wrapping Datentyps in Rust

Durch die Verwendung des *Wrapping* Datentyps kommt es im obenstehenden Programm zu keinem Absturz, da ein potenzielles Überlaufen der Variable hier explizit deklariert wurde (vgl. [20]). Wichtig zu vermerken ist, dass arithmetische Operationen mit Variablen unterschiedlichen Datentyps nicht möglich ist. Darum müssen diese, wie im obenstehenden Listing, auf den Wrapping Datentyp gecastet werden.

### 2.11.4 Fazit

Im Vergleich der beiden Programmiersprachen bieten diese neben der Überprüfung eines potenziellen Überlaufs durch vorherige oder anschließende Abfrage keine Möglichkeiten, diesen zu beseitigen. Hier ermöglicht ausschließlich der Compiler für Rust einen Check für das Wrappen der Variablen im Debug Mode. Durch das Beenden des Programms mit einer *panic* können somit Fehler während der Entwicklung erkannt werden. Jedoch entfällt beim Wechsel in den Release Mode diese Überprüfung, da ein Abstürzen des Programms im normalen Betriebsmodus für gewöhnlich nicht beabsichtigt ist, was ein ausgiebiges Testen der Funktionalität voraussetzt.

[20] https://doc.rust-lang.org/beta/std/num/struct.Wrapping.html

## 2.12 INT31-C

**Regel:** Bei Umwandlungen von Ganzzahlen ist sicherzustellen, dass Daten weder verlorengehen noch fehlerhaft interpretiert werden.

**Umwandlung von Integer-Werten in C:** Die Umwandlung von (Integer-)Werten in andere Datentypen geschieht implizit (vom Compiler) oder explizit (vom Entwickler). Beide Arten der Umwandlung sind nicht immer sinnvoll und können zu fehlerhaften Daten führen. Die einzigen Umwandlungen, die als sicher angesehen werden, sind Umwandlungen, die – vorausgesetzt der Status ist ebenfalls `signed` beziehungsweise `unsigned` – von einem kleineren Wertebereich zu einem größeren durchgeführt werden [1]. Umgekehrt angewendet können Umwandlungen zu verfälschten Werten führen, wenn der Wert vom neuen Datentyp nicht repräsentiert werden kann. Außerdem können bei Variablen, die vom Typ `signed` zum Typ `unsigned` umgewandelt werden, negative Werte zu fehlerhaften Daten führen, da der Maximalwert des neuen Typs solange addiert wird, bis sich der ursprüngliche Wert im Wertebereich des neuen Typs befindet. Nachfolgend finden zwei Umwandlungen statt: `signed char` zu `signed int` und `signed int` zu `unsigned int`.

```c
void charToIntUnsafeC(void) {
    signed char a = 'ä';
    unsigned int b = a;
    printf("%u", b);
    // a + UINT_MAX + 1
    // = 4294967204
}
```

Listing 35: Fehlerhaftes Ergebnis bei der Umwandlung von `signed char` zu `unsigned int`

```c
void charToIntSafeC(void) {
    signed char a = 'ä';
    unsigned int b = 0;
    if (a < 0) {
        // Error handling
    } else {
        b = a;
    }
}
```

Listing 36: Fehlerbehandlung bei der Umwandlung von `signed char` zu `unsigned int`

Bei der Übergabe von Integer-Werten an bestimmte C-Bibliotheken, wird der übergebene Integer-Wert als `unsigned char` interpretiert und kann außerhalb des Wertebereichs liegen.

```c
void intToCharUnsafeC(int character, FILE *stream) {
    // Aufruf von fputc(character, *stream); in Schleife
}
```

Listing 37: Unsicherer Aufruf der C-Bibliothek *fputc()*

Diese Problematik tritt auch bei weiteren C-Bibliotheken auf (unter anderem bei *memchr()*, *ungetc()* und *memchr()*). Außerdem werden alle Integer-Werte, die beim Aufruf der Funktionen *strchr()*, *strrchr()* und allen in *<ctype.h>* enthaltenen Funktionen zu `signed char` umgewandelt und sind damit ebenfalls für fehlerhaft interpretierte Daten anfällig. Eine Fehlerbehandlung für erwartete `unsigned char` Werte sähe folgendermaßen aus.

```c
void intToCharSafeC(int character, File *stream) {
    if(character >= 0 && character <= UCHAR_MAX) {
        // Aufruf von fputc(character, *stream); in Schleife
    } else {
        // Error handling
    }
}
```

Listing 38: Sicherer Aufruf der C-Bibliothek *fputc()*

Zusammenfassend lässt sich sagen, dass potentiell fehlerhaft konvertierte Integer-Werte in C häufig vorkommen und deren Fehlerbehandlung als kostspielig angesehen wird.

**Umwandlung von Integer-Werten in Rust**: Rust unterstützt keine implizite Umwandlung, sondern erwartet, dass bei einer Zuweisung die Datentypen dieselben sind.

```
fn char_to_int_unsafe_rust() {
    let a: c_schar = 'ä' as i8;
    let b: u32 = a as u32;
    println!("{}", b);
    //  a + u32::max_value()
    /* + 1 = 4294967268 */ }
```

Listing 39: Compiler verhindert Umwandlung

```
fn char_to_int_safe_rust() {
let a: c_schar  = 'ä' as i8;
    if a < 0 {
        // Error handling
    } else {
        let b: u32 = a as u32; }}
```

Listing 40: Fehlerbehandlung durch bedingte Verzweigung

Mithilfe einer Bibliothek lässt sich der zu `signed char` in C äquivalente Datentyp in Rust importieren [2]. Beim Versuch einen 32-Bit-Integer-Wert in einen `char` umzuwandeln, weist der Compiler daraufhin, dass lediglich 8-Bit-Integer-Werte in `chars` umgewandelt werden dürfen.

```
fn int_to_char_unsafe_rust(a:i32)
{
    let c = a as char; // Only
}   // 'u8' can be cast as 'char'
```

Listing 41: Compiler verhindert (fehlerhafte) Umwandlung

```
fn int_to_char_safe_rust(a:i32){
   if a >= u8::min_value() as i32
   && a <= u8::max_value() as i32
   {
       let b: u8 = a as u8;
       let c = b as char;
   } else {/* Error handling */}}
```

Listing 42: Gültige Umwandlung

**Fazit:** Der Rust-Compiler informiert den Entwickler – im Normalfall – über mögliche Lösungen (*rustc –explain*) bei Kompilierungsproblemen, während C-Programmierer häufiger Gefahr laufen, fehlerhafte Umwandlungen erst dann zu bemerken, wenn Daten nicht mehr vorhanden oder falsch interpretiert worden sind. Außerdem wird durch die erzwungene explizite Umwandlung das Bewusstsein für Datentypen und deren Verwendungszweck geschärft.

[1] https://www.geeksforgeeks.org/type-conversion-c/

[2] https://doc.rust-lang.org/rust-by-example/types/cast.html

## 2.13 INT32-C

Die Regel INT32-C besagt, dass Operationen mit vorzeichenbehafteten, ganzen Zahlen nicht zu einem Überlauf führen dürfen, denn diese können wiederum zu undefiniertem Verhalten oder Zuständen führen. Infolgedessen haben Implementierungen einen großen Spielraum, wie sie mit Integer-Überlauf umgehen. Eine Implementierung, die vorzeichenbehaftete Ganzzahlentypen beispielsweise als Modul definiert, muss keinen Integer-Überlauf erkennen. Implementierungen können auch auf arithmetische Überläufe zurückgreifen oder einfach davon ausgehen, dass Überläufe nie passieren werden. Es ist auch möglich, dass die konforme Implementierung Code emittiert, der in verschiedenen Kontexten ein unterschiedliches Verhalten zeigt. So kann beispielsweise eine Implementierung festlegen, dass eine in einem lokalen Bereich deklarierte Variable nicht überlaufen kann.

In manchen Fällen ist es wichtig sicherzustellen, dass Operationen auf vorzeichenbehafteten ganzen Zahlen nicht zu einem Überlauf führen. Von besonderer Bedeutung sind Operationen mit Werten, die aus einer gefährlichen/unbekannten Quelle stammen und als beliebige Zeigearithmetik, Zuweisungsausdruck für die Deklaration eines Arrays oder als Funktionsargumente vom Typ $size\_t$ oder $rsize\_t$ verwendet werden.

Ganzzahlige Operationen werden überlaufen, wenn der Wert nicht durch die zugrunde liegende Darstellung der Zahl dargestellt werden kann. Bei folgenden Operationen ist ein Überlauf möglich: Addition, Subtraktion, Multiplikation, Division und Modulo. In diesem Abschnitt wird anhand der Addition das Problem des Überlaufes genauer erklärt. [21]

### Betrachtung von C

Das Problem kann in C auftreten, wenn zum Beispiel bei einer Addition von mehreren Zahlen der Wertebereich des Zahlenraums überschritten wird. In dem nachfolgenden Beispiel wird dies mithilfe der Addition von eins zur maximal möglichen Ganzzahl gemacht. Zum Compilieren wurden die C Compiler GCC Version 7.3.0 und die Clang Version 6.0.0 eingesetzt. Weder während des Compilierens noch bei der Ausführung wurde eine Fehlermeldung ausgegeben, die auf den Überlauf hingewiesen hätte. Das Ergebnis wurde bei beiden Compilern mit Überlauf ausgegeben.

```
1  int main() {
2  int a = INT_MAX; // setzt die Variable auf Maximale was in integer möglich ist (+
       2147483647)
3  int b = 1; //Variable wird auf 1 gesetzt
4  signed int sum = a + b; //Addiert a und b Speichert Ergebniss in sum
5  printf("Ergebnis Addition = %5d \n", sum); //Ausgabe der Summe
6  return 0;
7  }
```

Listing 43: C Addition

Durch eine einfache Abfrage kann sichergestellt werden, dass der Additionsvorgang nicht überlaufen kann.

```
1  int main() {
2  int a = INT_MAX; // setzt die Variable auf Maximale (+ 2147483647)
3  int b = 1; //Variable wird auf 1 gesetzt
4  signed int sum = 0;
5  if (((b > 0) && (a > (INT_MAX - b))) ||
6  ((b < 0) && (a < (INT_MIN - b)))) { // Abfrage ob Fehlerfall vorliegt
7  printf("Fehler Integerueberlauf \n");  //Fehlerausgabe
8  } else {
9  sum = a + b;   //Addiert a und b Speichert Ergebniss in sum
10 }
11 printf("Ergebnis Addition = %5d \n", sum);
12 return 0;
```

[21] wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow

```
13 }
```

Listing 44: C Addition mit Überlaufschutz

Auch bei den anderen Berechnungsarten lässt sich der Integer-Überlauf, durch mehr oder weniger komplexe Abfragen verhindern. Dies muss immer explizit vom Programmierer gemacht werden.

### Betrachtung von Rust

Vergleicht man dieselben Operationen in Rust, so muss in Rust nicht wie in C eine zusätzliche Abfrage eingebaut werden, um diesen Fehler ausschließen zu können. Es wurden alle Integer-Überlaufe beim Ausführen erkannt und mit einer klar verständlichen Fehlermeldung zurückgegeben.

```
1 fn main() {
2 let a = i32::max_value(); // setzt die Variable auf Maximale was in Integer 32 möglich
      ist (+ 2147483647)
3 let b = 1; //Variable wird auf 1 gesetzt
4 let sum = a + b;  //Addiert a und b Speichert Ergebniss in sum
5 println!("Ergebnis Addition (a+b) =  {}", sum);
6 }
```

Listing 45: Rust Addition mit Überlaufschutz

Es gibt in Rust aber die Möglichkeit in den Modus *unsafe* zu wechseln. Dies ermöglicht dem Programmierer mehr Freiheit, Sicherheitslücken können somit vom Compiler aber nicht mehr erkannt werden können. Aber auch in diesem Modus werden die verschieden Überläufe zur Laufzeit abgefangen. [22].

Zusammenfassend lässt sich also sagen, Rust bietet dem Programmierer viel Unterstützung bei der Vermeidung von Integer-Überläufen. Das Problem der Regel INT32-C ist in Rust nicht mehr gegeben. Es bietet allerdings dem Programmierer auch nicht die Möglichkeit einen solchen Überlauf künstlich herbeizuführen.

[22] https://doc.rust-lang.org

## 2.14 INT34-C

In der INT34-C wird das bitweise Verschieben von Ganzzahlen betrachtet. Dafür bezieht sich die Regel auf drei Problemstellungen, die C nicht automatisch erkennt. Das erste Problem ist, dass C keine selbstständige Überwachung der Bitbreite bietet. Die zweite Problematik bezieht sich auf das Verschieben einer Ganzzahl mit Vorzeichen um einen negativen Wert. Beim dritten und somit letzten Problem dieser Regel wird das bitweise Verschieben einer Ganzzahl nach rechts betrachtet.

Das Risiko, das die genannten Fehler eintreten, wurde von der *SEI CERT C Coding Standard* auf gering eingestuft, sowie das Ausmaß der Probleme. Der Aufwand zum Beheben der Probleme wurde mit durchschnittlich angegeben. Nachfolgend werden die genannten Probleme in C betrachtet und anschließend mit der Programmiersprache „Rust“ verglichen.

### 2.14.1 Betrachtung in C

Die genannten Probleme treten in C auf, wenn eine Ganzzahl *a* um einen Wert *b* verschoben werden soll. Nachfolgend werden alle drei problematischen Anwendungsfälle betrachtet. Es wird mit Szenario 1 begonnen, bei welcher eine vorzeichenlose Ganzzahl nach links verschoben wird (siehe Codeabschnitt 46).

```
1  //Szenario 1 – Bitweises Linksverschieben einer vorzeichenlosen Variable
2  void shift_left(unsigned int ui_a, unsigned int ui_b) {
3    unsigned int uresult  = ui_a << ui_b;
4  }
5  //Szenario 2 – Bitweises Linksverschieben einer vorzeichenbehafteten Variable
6  void shift_signed(signed long si_a, signed long si_b) {
7    signed long result = si_a << si_b;
8  }
9  //Szenario 3 – Bitweises Rechtsverschieben einer vorzeichenbehafteten Variable
10 void shift_right(signed long si_a, signed long si_b) {
11   signed long result = si_a >> si_b;
12 }
```

Listing 46: Codebeispiel in C

Hierbei wird durch den Aufruf der Funktion *shift_left()* die vorzeichenlose Ganzzahl um den Wert ui_b verschoben. C prüft in diesem Szenario nicht, ob der Wert von ui_b größer ist, als die Bitbreite des definierten Datentyps (im ersten Beispiel wären dies 32 Bit). Beim zweiten Szenario wird eine vorzeichenbehaftete Ganzzahl si_a um den Wert si_b nach links verschoben. In diesem Beispiel müsste eine Überprüfung des Überlaufs, wie im ersten Szenario erfolgen. Zusätzlich muss bei einer vorzeichenbehafteten Ganzzahl auf negative Werte geprüft werden. Denn falls si_b negativ ist, könnte ein undefiniertes Verhalten eintreten. Aus diesem Grund empfiehlt der Coding Standard von C, dass bitweises Verschieben von Zahlen nur mit vorzeichenlosen Ganzzahlen erfolgen sollte. Das dritte Problem *shift_right()* betrachtet das bitweise Verschieben nach rechts und ergänzt die Szenarien eins und zwei. Falls die zu verschiebende Zahl „a“ negativ ist, kann das Ergebnis des Verschiebevorgangs nicht mehr klar vorhergesagt werden. Dies liegt daran, dass die jeweilige Implementation des Verschiebevorgangs eine Rolle spielt, denn sie kann sowohl arithmetisch (1), als auch logisch (0) erfolgen. Damit wird entweder das erste Bit durch eine 0 oder durch eine 1 ersetzt. Da dies von der Implementation abhängt, empfiehlt der Coding Standard von C auch hierbei, dass nur mit vorzeichenlosen Ganzzahlen geschoben werden sollte. Nachdem die Probleme von INT34-C betrachtet wurden, erfolgt nun der Vergleich in Rust.

### 2.14.2 Rust

In diesem Abschnitt wird betrachtet, ob Rust die Probleme von C beheben kann. Um dies zu überprüfen, wurden die drei Szenarien aus dem Codebeispiel 46 in Rust geschrieben und im nachfolgenden

Codebeispiel 47 dargestellt.

```rust
//Bitweises Linksverschieben einer vorzeichenbehafteten Variable
fn shift_left(si_a:i32, si_b:i32) {
  let uresult:i32 = si_a << si_b;
}
//Bitweises Rechtsverschieben einer vorzeichenbehafteten Variable
fn shift_right(si_a:i32, si_b:i32) {
  let uresult:i32 = si_a >> si_b;
}
```

Listing 47: Codebeispiel in Rust

In Rust wurden zwei einfache Funktionen geschrieben, bei denen die Probleme betrachtet werden. Für eine bitweise Linksverschiebung wird die Funktion *shift_left()* verwendet und für die Rechtsverschiebung dann die Funktion *shift_right()*, wie im Codebeispiel 47 dargestellt.

Anschließend wurden die Szenarien eins, zwei und drei nachgestellt und die Fehler künstlich erzeugt. Während der Kompilierung konnten die Fehler vom Compiler nicht erkannt werden. Erst durch das Ausführen wurden die eingebauten Fehler bemerkt und das Programm angehalten. So bringt das Programm zum Beispiel beim Ausführen der Funktion *shift_left()* bei einer Bereichsüberschreitung der Bitbreite, wie in Szenario eins beschrieben, die Fehlermeldung: „attempt to shift left with overflow“. Das Abbrechen zur Laufzeit erfolgte sowohl bei einer Bereichsüberschreitung, als auch bei einem negativen Verschiebevorgang. Die Implementation bei einem Verschiebevorgang erfolgt bei negativen Zahlenwerten arithmetisch (1) und bei positiven logisch (0).[23]

Selbst wenn die Funktion als *unsafe* deklariert ist und in einem unsicheren Abschnitt ausgeführt wird, so fängt Rust dennoch die genannten Probleme ab. Auch hier kommt es zu einem Abbruch während der Laufzeit, mitsamt Fehlermeldung.

Zusammenfassend lässt sich sagen, dass die genannten Probleme aus INT34-C von Rust komplett behoben werden. Damit muss sich der Programmierer bei einer bitweisen Links- oder Rechtsverschiebung nicht um die Fehlerbehandlung kümmern.

[23] https://doc.rust-lang.org/reference/expressions/operator-expr.html#arithmetic-and-logical-binary-operators

## 2.15 INT36-C

Im Unterschied zu Variablen verweisen Pointer nicht auf einen Wert, sondern auf eine Adresse. Anders als bei primitiven Datentypen ist die Größe von Pointern nicht immer glei-ch. In erster Linie beschränkt die vorhandene CPU Architektur die maximale Länge einer Adresse, aber Betriebssystem und Compiler können ebenfalls Einfluss darauf haben. Ausser den Problemen mit der Pointerlänge kann die Manipulation von Pointern auch dazu führen, dass eine Referenz nicht mehr auf den vorher deklarierten Datentyp verweist und das Programm nach dessen Aufruf abbricht. Um zu verhindern, dass die Umwandlung zwi-schen Pointern und Integern Fehler verursacht, wird folgende Regel formuliert:

### Regel

Konvertiere keinen Integer-Typ zu einem Pointer wenn der erzeugte Pointer die falsche Stelle im Speicher referenziert, nicht auf eine Entität des spezifizierten Typs verweist oder es sich um eine *trap representation* [24] handelt. Konvertiere keinen Pointer zu einem Integer, wenn das Ergebniss nicht mit dem Integer-Typ repräsentiert werden kann.

### C

Um herauszufinden, wie groß ein C Pointer zur Laufzeit ist, kann mit Hilfe von `sizeof()` die Speichergröße einer Variablen ausgegeben werden. Auf einem System mit 64-Bit CPU, Betriebssystem und Compiler liefert `sizeof(void *)` ein Ergebnis von `8` (in Bytes), also ins-gesamt 8*8=64 Bits. Wird das Programm auf einem 32-Bit System kompiliert, so erzeugt der Aufruf einen Wert von `4`, eine Adresse ist in diesem Fall also 32-Bit lang.

```
1 int main(int argc, char * argv[]){
2     int *ptr;
3     unsigned int i = (unsigned int)ptr;
4 }
```

Listing 48: Fehleranfällige Implementierung

In der Programmiersprache C können Pointer voneinander abgezogen werden, um zu be-stimmen, wie viele Elemente sich zwischen den Adressen befinden. Andere arithmetische Operationen wie Addition, Multiplikation oder Division sind nicht möglich. Ein naiver cast von einem Pointer auf einen Integer wie in Listing 48 erzeugt zum Zeitpunkt der Kompilierung mit `gcc` folgende Warnung:

```
cast from pointer to integer of different size [-Wpointer-to-int-cast]
```

Um dieses Problem zu umgehen gibt es seit dem C99 Standard in `stdint.h` den Datentyp `uintptr_t`[25]. Dieser Typ garantiert, dass die erzeugte Variable ausreichend Platz für einen Pointer besitzt. Eine korrekte Implementierung mit `uintptr_t` ist in Listing 49 zu sehen.

```
1 #include <stdint.h>
2
3 int main(int argc, char * argv[]){
4     int *ptr;
5     uintptr_t i = (uintptr_t)ptr;
6 }
```

Listing 49: Sichere Implementierung mit `uintptr_t`

[24] https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-traprepresentation
[25] https://en.cppreference.com/w/c/types/integer

Erfolgt die Umwandlung in die entgegengesetzte Richtung, können verschiedene Fehler durch Compileroptimierungen oder OS-Interrupts entstehen. Um dies zu verhindern gibt es das Keyword `volatile`. Die Verwendung von `volatile` signalisiert dem Compiler, dass sich der Wert einer Variablen jederzeit ändern kann, auch wenn dies nicht im direkten Umfeld der Zuweisung ausgelöst wird.

### Rust

Durch das Ownership-Prinzip in Rust ist der Zugriff auf Referenzen fundamental anders als in C. Raw Pointer in Rust entsprechen eher den Pointern aus C, sind aber *unsafe*. Das Anlegen eines Raw Pointers benötigt selbst keinen `unsafe`-Block, wird dieser jedoch ohne einen solchen Block dereferenziert bricht der Compiler mit folgendem Fehler ab:

```
dereference of raw pointer requires unsafe function or block
```

Im Gegensatz zu Referenzen und Smart Pointern können Raw Pointer null sein, implementieren keinen automatischen Cleanup und es gibt keine Garantie, dass sie auf eine gültige Stelle im Speicher zeigen.
In Rust gibt es den primitiven Datentypen `usize`. Die Größe dieses Typen ist von der Zielarchitektur abhängig und erfüllt somit die selbe Funktion wie der `uintptr_t` Typ in C. Im Unterschied zu C erlaubt Rust keine Subtraktion zweier Raw Pointer, dafür ist eine Umwandlung in `usize` sinnvoll. Ein weiteres Werkzeug für den Umgang mit Raw Pointern gibt es die `ptr::offset`-Methode, welche ebenfalls `unsafe` ist.

## Fazit

Raw-Pointer in Rust erlauben es zwar, manche Vorgehensweisen aus C zu übernehmen, aber es können dadurch auch potentiell viele Fehlerquellen und Sicherheitsrisiken wieder auftreten - vor allem in Hinsicht auf die Speicher- und Threadsicherheit - die eigentlich durch das Ownership-Prinzip und Typisierung in Rust gelöst wurden. Daher sollte ihr Einsatz nur im Notfall erfolgen.

## 2.16 FLP30-C

### 2.16.1 Regel

Laut der Secure Coding Regel FLP30-C soll man keine Gleitkommazahl als Schleifenzähler benutzen[26].

### 2.16.2 Einführung

Eine Gleitkommazahl stellt eine reelle Zahl in Wissenschaftlicher Notation vor, die wird oft wegen Größere Genauigkeit als die ganzen Zahlen benutzt. Falls die Aufgabe ein Ergebnis mit nicht unbedingt idealer Genauigkeit zulässt, die Gleitkommazahlen sollten kein Problem darstellen aber, wenn es erfordert ist, dass das Programm deterministisch sein muss, kann es anderes sein. Aufgrund der Ungenauigkeit der Gleitkommazahlen, können die unvorhersehbaren Sachen im Programm generieren, die problematisch und schädlich für ein Ergebnis sein können. In diesem Fall wird die Sicherheit sehr negativ beeinflusst. Ein Beispiel zu diesem Problem ist die Benutzung von Gleitkommazahlen als eine Variable in den Schleifen.

### 2.16.3 Gleitkommazahlen in den Schleifen in C

In der C-Sprache, es ist zugelassen eine Gleitkommazahl als eine Variable in einer Schleife zu benutzen, jedoch das Ergebnis ist nicht deterministisch. In Abhängigkeit von einer Implementierung, kann die Anzahl der Iterationen, nachdem die Ausführung von der Schleife endet, unterschiedlich sein. Der Grund dafür ist die oben genannte Ungenauigkeit der Gleitkommazahlen. Ein Beispiel dafür, wo die Schleife 99 oder 100 mal durchlaufen kann, wird in Listing 50 angezeigt.

```
1 #include  <stdio.h>
2 void func(void){
3   for (float x = 0.1; x <= 10.0; x += 0.1) {
4   }
5 }
```

Listing 50: Beispiel kompiliert mit gcc hat 99 Durchläufe.

Falls man eine Gleitkommazahl für weitere Berechnungen braucht, kann man dieses Problem vermeiden, dadurch, dass man Integer als eine Variable in einer Schleife benutzt und dann erlangt man eine Gleitkommazahl von dem Integer. Das untenstehende Beispiel (Listing 51) zeigt die Lösung.

```
1 #include  <stdio.h>
2 int func(void){
3   for (int count = 1; count <= 100; ++count){
4     float x = count / 10.0;
5   }
6 }
```

Listing 51: Beispiel kompiliert mit gcc hat genau 100 Durchläufe.

Die Zählvariable, die Gleitkommazahl ist, ist wenig genauer als Integerzahl. Man kann das in der Ausgabe der Programme merken (siehe Tabelle 1).

### 2.16.4 Gleitkommazahlen in den Schleifen in Rust

Rust ist eine sicherheitsorientierte Programmiersprache, deshalb besitzt sie einen Sicherheitsmechanismus, der das besprochene Problem schon auf Compiler-Niveu löst[27]. Während Kompilation

[26] https://wiki.sei.cmu.edu/confluence/display/c/FLP30-C.+Do+not+use+floating-point+variables+as+loop+counters
[27] https://www.rust-lang.org/learn

Tabelle 1: Wert der x-Variable

| Iterationsnummer | Gleitkommazahl als Zaehlvariable | Integerzahl als Zaehlvariable |
|---|---|---|
| 27 | 2.700000 | 2.700000 |
| 28 | 2.799999 | 2.800000 |
| 29 | 2.899999 | 2.900000 |
| 30 | 2.999999 | 3.000000 |
| 31 | 3.099999 | 3.100000 |

des Programms, in dem man Gleitkommazahl benutzt, unterbricht der Compiler die Operation und gleichzeitig alarmiert, dass die Zählvariable eine Gleitkommazahl ist (siehe Listing 52).

```
fn main(){
  // throws Exception: expected usize, found floating-point number
  for x in (1..10).step_by(0.1){
  }
}
```

Listing 52: Gleitkommazahl als Zaehlvariable in Rust.

Solche Lösung zwingt den Programmierer, die Nutzung der Gleitkommazahl als Zählvariable zu vermeiden. Selbstverständlich gibt es eine Möglichkeit, eine Gleitkommazahl von Integer-Variable zu erlangen, wie im Fall von C (siehe Listing 53).

```
fn main(){
  for x in (1..10).step_by(1){
    let x = x as f32 * 0.1;
  }
}
```

Listing 53: Alternative zur Gleitkommazahl als Zaehlvariable in Rust.

### 2.16.5 Zusammenfassung

Zusammenfassend lässt sich sagen, dass mit Rücksicht auf der Ungenauigkeit der Gleitkommazahlen, die Benutzung von Gleitkommazahlen als Zählvariable in einer Schleife nicht empfohlen ist. Im Fall der C-Sprache ist solche Operation zugelassen aber fehlerhaft. Dank der Voraussetzungen der Rust-Sprache ist solche Operation unmöglich durchzuführen, wodurch schützt das Programm vor den Fehler schon auf Compiler-Niveu.

## 2.17 FLP32-C

### 2.17.1 Regel

Die Secure Coding Regel FLP32-C behandelt das Verhindern und Erkennen von Domänen- und Bereichsfehlern in mathematischen Funktionen der Programmiersprache C.

### 2.17.2 Analyse Programmiersprache C

Nach Definition des C Standards (7.12.1) gibt es drei Arten von Fehlern, die sich speziell auf mathematische Funktionen beziehen[28].

- *Domain Error*: Das Eingabeargument ist nicht im Bereich der mathematischen Funktion definiert.
- *Pole Error*: Das Resultat der Funktion ist genau unendlich.
- *Range Error*: Das Ergebnis kann aufgrund der Größe nicht vom aktuellen Typ repräsentiert werden.

```c
#include <math.h>

double unsafe_log10(double value)
{
  return log10(value);
}
```

Listing 54: C Unsichere Log10 Funktion

Um eine mathematische Funktion sicher nutzen zu können, darf der Entwickler bei jeder Funktion nur Bereiche und Resultate zulassen, welche definiert sind und sicher keine Fehler werfen.

Im Falle der Log10 Funktion würde das Bedeuten, dass auf Domain- und Pole Fehler geachtet werden muss.

```c
#include <math.h>

double safe_log10(double value)
{
  if (value <= 0)
  {
    fprintf(stderr, "log requires a nonnegative argument");
    /* Handle domain / pole error */
  }
  return log10(value);
}
```

Listing 55: C Sichere Log10 Funktion

### 2.17.3 Risikobewertung

Werden die Fehler nicht verhindert oder rechtzeitig behandelt, so kann dies zu unerwarteten Verhalten führen, wodurch die Software potentiellen Angriffen eine Schwachstelle bietet. Da es ein ordentlicher Mehraufwand ist, jede dieser Funktionen sicher zu verwenden, wird dies oftmals nicht beachtet und ist somit keine Seltenheit. Die Schwere der Fehler wird als moderat eingestuft, sollte aber auf Grund der Häufigkeit nicht unterschätzt werden.

[28] https://wiki.sei.cmu.edu/confluence/display/c/FLP32-C.+Prevent+or+detect+domain+and+range+errors+in+math+functions

### 2.17.4 Analyse Programmiersprache Rust

In Rust ist das gleiche Verhalten festzustellen, wie in der Programmiersprache C. Wird ein Wert übergeben, der nicht von der mathematischen Funktion definiert ist, so wird auch in Rust *NaN* zurückgegeben. Ebenso entspricht das Verhalten von *Pole* und *Range* Fehlern dem von C. Der einzige Unterschied ist in der Verwendung der Methoden festzustellen, da diese in Rust als *Extension Methods* gehandhabt werden[29].

Möchte der Entwickler dieses Verhalten nun abfangen, so ist der gängige Lösungsansatz der Programmiersprache Rust der gleiche wie in C. Es werden lediglich Bereiche und Resultate zugelassen, die von der mathematischen Funktion vorgesehen sind.

```
1 fn safe_log10(value: f64) -> f64
2 {
3     if value <= 0_f64
4     {
5         println!("log requires a nonnegative argument");
6     }
7     return value.log10();
8 }
```

Listing 56: Rust Sichere Log10 Funktion

### 2.17.5 Fazit

Zusammenfassend lässt sich feststellen, dass sich Rust und C beim behandeln von Domänen- und Bereichsfehlern gleich verhalten. Es existieren bei beiden Sprachen keine direkten Fehlermeldungen, die bei fehlerhaften Eingaben in mathematische Funktionen ausgelöst werden. Die Fehler treten frühstens bei der Verwendung der Resultate auf und können bei beiden Sprachen zu unerwarteten Verhalten führen.

[29] https://doc.rust-lang.org/1.12.1/std/primitive.f64.html

## 2.18 FLP34-C

Die Regel FLP34-C erläutert, dass bei dem Konvertieren von floating point Werten ein undefiniertes Verhalten auftreten kann, wenn der Zieltyp ein floating point mit kleinere Reichweite oder Genauigkeit, sowie ein integer Typ ist. Im Folgenden werden die Regeln aus dem C Standard genauer untersucht.

### 2.18.1 Die Regel

Der C Standard 6.3.1.4 beschreibt im ersten Paragraph das Verhalten von endlichen Werten bei floating point Konvertierung zu integer, außer boolean. Um eine Gleitkommazahl in eine Ganzzahl wandeln zu können, werden Nachkommastellen gekürzt. Die daraus entstehende Zahl soll bei einem integer Typen repräsentiert werden können. Falls dies nicht der Fall ist, tritt ein undefiniertes Verhalten auf. Der zweite Paragraph betrachtet das Konvertieren von integer zu floating point. Falls der Wert im Zieltyp nicht exakt dargestellt werden kann, aber im Bereich des Repräsentierbaren des Zieltyps liegt, wird entweder der naheliegende kleinere Wert oder größere Wert dargestellt. Dies hängt von der Implementierung ab. Falls der Wert nicht repräsentierbar ist, gilt ebenfalls ein undefiniertes Verhalten.

Die Subklausel 6.3.1.5 beschäftigt sich ausschließlich mit Umwandlungen innerhalb von floating point Typen. Dies ist, laut C Coding Standard, für Systeme mit IEEE 754 floating points nicht mehr relevant.

Der IEEE 754 Standard[30] legt Regeln zu Gleitkommazahlen für verschiedene Verfahren fest und definiert die Darstellung dieser Werte. In der heutigen Zeit verwenden die meisten Kompiler diesen Standard, welcher auch in *Annex F - IEC 60559 Floating Point Arithmetic* von *ISO/IEC 9899:201x*[31] aufgeführt ist.

Somit wird in den folgenden Beispielen von C und Rust nur erläutert ob IEEE 754 verwendet wird, sowie die Aussage der beiden Paragraphen von 6.3.1.4 näher betrachtet.

### 2.18.2 C

In der Programmiersprache C gibt es den Flag `__STDC_IEC_559__` der aussagt, das eine vollständige Unterstützung von IEEE 754 floating points vorhanden ist. Dieser ist bei der C Standard Revision C17 und C11 nur optional. Auch wenn dieser Flag nicht gesetzt ist können IEEE 754 floating points verwendet werden.

```
1 // [1]
2 float a = FLT_MAX; // Float maximaler Wert. ca. 3 * 10^38.
3 int b = a;         // Float zu integer
4 // [2]
5 int c = INT_MAX; // Integer maximaler Wert. 2147483647
6 float d = c;     // Integer zu float
```

Listing 57: FLP34-C (C)

Bei der Konvertierung von `float` zu `int` (siehe Listing 57, [1]) wird als Ausgangslage der maximale float Wert verwendet, da dieser nicht in integer repräsentierbar ist. Das Ergebnis in den Kompilern gcc (8.1.0), clang (7.0.0) und Visual Studio C Compiler (19.20.27508.1) ist `-2147483648` (minimum `int`). Mit dem Optimierungsbefehl für gcc `-O` oder mit der Version gcc (Raspbian 6.3.0) kann aber auch `2147483647` (maximum `int`) als Ergebnis resultieren. Das undefinierte Verhalten ist hier der integer Overflow, welcher laut C Coding Standard von den Implementierung gehandhabt wird. In Visual Studio gibt es beim Kompilieren in der Debug Konfiguration eine Warnung des möglichen Datenverlustes. Bei dem Szenario zu Paragraph 2 (siehe Listing 57, [2]) ist die Ganzzahl der maximale integer Wert. Da dieser bei IEEE 754 floating points nicht dargestellt werden kann, ist das Ergebnis

[30] https://standards.ieee.org/standard/754-2008.html
[31] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf

`2147483648` der naheliegende größere Wert. Bei Visual Studio gibt es im Debug Modus ebenfalls eine Warnung des möglichen Datenverlust.

### 2.18.3 Rust

Rust verwendet laut der Dokumentation ebenfalls IEEE 754 floating points[32].

```
1 // [1]
2 let a = std::f32::MAX as f32; // ca. 3 * 10^38.
3 let _b = a as i32;            // Float zu integer
4 // [2]
5 let _d = std::i32::MAX as i32; // 2147483647
6 let c = _d as f32;             // Integer zu float
```

Listing 58: FLP34-C (Rust)

Um ein ähnliches Szenario wie bei C zu schaffen wird in dem Beispiel mit 32 Bit Datentypen gearbeitet und ebenfalls der maximale float und integer Wert bei den Umwandlungen verwendet. Bei der ersten Konvertierung (siehe Listing 58, [1]) gibt es bei Rust verschiedene Ergebnisse. Im Debug Modus (1.33.0) ist der resultierende Wert identisch wie bei C, also `-2147483648`. Im Release Modus ist das Ergebnis unterschiedlich. Die Stable Version (1.33.0) gibt `0`, die Beta (1.34.0-beta.8) und Nightly (1.35.0-nightly) `32766` als Wert an. Hier lässt sich auf ein undefiniertes Verhalten schließen. Bei der Konvertierung von `i32` zu `f32` (siehe Listing 58, [2]) gibt es bei den Kompiler Versionen keine Unterschiede. Das Ergebnis ist `2147483600.0`. Der Ausgangswert wird hierbei, wie bei C, nicht korrekt konvertiert.

### 2.18.4 Fazit

Es stellt sich heraus das bei einigen Konvertierungen das Problem des möglichen Datenverlustes besteht oder ein undefiniertes Verhalten auftreten kann. Dies betrifft beide Programmiersprachen. Um das zu verhindern bietet sich an, eine Funktion zu erstellen, die überprüft ob der Ausgangswert eine niedrigere Genauigkeit des Zieltyps besitzt und ob dieser im Bereich des Repräsentierbaren liegt. Ist das der Fall, kann der Wert konvertiert werden.

[32] https://doc.rust-lang.org/reference/types/numeric.html

## 2.19 FLP36-C

Die Regel FLP36-C besagt, dass beim Umwandeln von Ganzzahlwerten in Gleitkommazahlen die Genauigkeit berücksichtigt werden muss. Das bedeutet, dass durch das Wechseln zwischen Typen Fehler auftreten können, wenn diese nicht kompatibel zueinander sind. Für die Kompatibilität spielt dabei nicht nur die definierte Bit-Größe eine Rolle, sondern auch deren Aufbau. Dass eine größere Zahl nicht ohne Weiteres in eine kleine konvertierbar ist, sollte weitreichend bekannt sein. Aus diesem Grund wird folgend die Umwandlung zwischen Ganz- und Gleitkommazahlen gleicher Bit-Größe betrachtet.

### 2.19.1 Aufbau von Ganz- und Gleitkommazahlen

Zahlen, die dem Typen Integer zugeordnet werden, unterscheiden sich nur in zwei Merkmalen. Dies sind die Bitlänge, sowie die Eigenschaft, ob der Typ nur positive Zahlen („unsigned“) oder auch negative Zahlen („signed“) abbilden kann. So werden bei unsigned Integer Typen alle Bits für den Zahlenwert verwendet und bei signed Typen alle, außer dem ersten Bit, dem sogenannten Signbit, welches bestimmt ob der Zahlenwert positiv oder negativ ist.

Gleitkommazahlen sind in ihrem Aufbau deutlich komplexer. Sie sind in der IEEE 754[33] standardisiert und unterteilen sich in `single`, auch `float` genannt, und `double`. Nach IEEE 754 ist die Größe des `float`-Typen auf 32 Bit, unterteilt in ein Signbit, acht Exponentenbits und 23 Mantissenbits (siehe Abbildung 2), und die des `double`-Typen auf 64 Bit, mit ebenfalls einem Signbit, 11 Exponentenbits und 56 Mantissenbits, definiert.

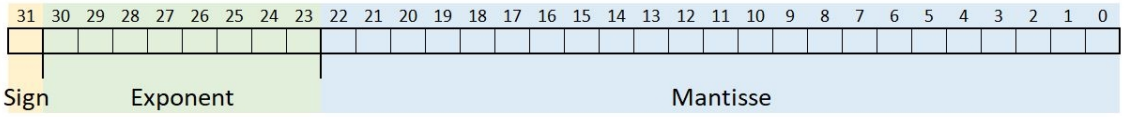


Abbildung 2: Aufbau einer 32 Bit Gleitkommazahl nach IEEE 754

Beim Aufbau ist zu beachten, dass die Mantisse den eigentlichen Zahlenwert enthält, dessen Komma so verschoben wurde, dass in dualer Schreibweise vor dem Komma nur noch eine 1 steht. In der Mantisse sind jedoch nur die Nachkommastellen abgebildet und da die Vorkommastelle stets eine 1 darstellt, wird diese weggelassen. Diese Stelle, bzw. Bit, wird deshalb auch als „Hiddenbit“ bezeichnet. Der Exponent hält die Information, um wieviele Stellen sich das Komma verschoben hat. So ergibt sich der genaue Zahlenwert der Gleitkommazahl aus der folgenden Formel.

$$Gleitkommazahl = Vorzeichen \cdot Mantisse \cdot 2^{Exponent} \tag{1}$$

### 2.19.2 Umwandlung von Ganz- zu Gleitkommazahlen allgemein

Für die Umwandlung einer Ganz- in eine Gleitkommazahl gleicher Speichergröße ergibt sich somit eine Einschränkung. So kann ein Gleitkommazahlentyp nur Ganzzahlen korrekt darstellen, deren führende 1 in Dualform maximal so viele Stellen von der letzten 1 entfernt ist, wie die Länge der Mantisse lang ist. Somit ergibt sich z.B. für 32 Bit, dass die Zahl `00000010000000000000000000000010 (Dual) = 16777218 (Dezimal)` konvertierbar ist. Die Zahl `00000010000000000000000000000001 (Dual) = 16777217 (Dezimal)` hingegen nicht, da bei ihr mehr als 23 Stellen zwischen der führenden und der letzten 1 stehen.

[33] IEEE 754 - Standard for Floating-Point Arithmetic (https://standards.ieee.org/standard/754-2008.html)

### 2.19.3 Umwandlung unter C und unter Rust

Weder unter C (siehe Listing 59), noch unter Rust (siehe Listing 60) werden Zahlen der oben beschriebenen Struktur korrekt konvertiert. Ebenso wird in keinem der getesteten Compiler[34] eine Warnung ausgegeben, sodass der Fehler unter Umständen unerkannt bleibt.

```c
int main()
{
    int convertibleInt    = 0b00000010000000000000000000000010;
    int notConvertibleInt = 0b00000010000000000000000000000001;
    printf("Integer: %d\n", convertibleInt);           // >> 16777218
    printf("Float:   %f\n", (float)convertibleInt);    // >> 16777218.000000
    printf("Integer: %d\n", notConvertibleInt);        // >> 16777217
    printf("Float:   %f\n", (float)notConvertibleInt); // >> 16777216.000000
    return 0;
}
```

Listing 59: Umwandlung von 32 Bit Integerzahlen zu 32 Bit Floatzahlen unter C.

```rust
fn main() {
    let i = 16777217 as i32; // 32 Bit Integer
    let f = i as f32;        // Converting to 32 Bit Float
    println!("Integer: {:?}", i); // >> 16777217
    println!("Float:   {:?}", f); // >> 16777216.0
}
```

Listing 60: Umwandlung von 32 Bit Integerzahlen zu 32 Bit Floatzahlen unter Rust.

### 2.19.4 Fazit

Das Umwandeln von Ganzzahlen zu Gleitkommazahlen ist sowohl unter C, als auch unter Rust, kritisch zu betrachten. Vor allem unter Rust sollte zumindest eine Warnung ausgegeben werden, da diese Sprache oft als „Secure by default“ bezeichnet wird. Die hier auftretenden Konvertierungsfehler könnten weitreichende Folgen nach sich ziehen.

Unter C kommt belastend hinzu, dass im C-Standard[35] Ganzzahltypen zwar definiert, aber deren Größe nicht festgelegt wird. So könnte theoretisch jeder Compiler eine andere Größe für jeden Typen festlegen.

[34] Verwendete C-Compiler: gcc 9.0.1 | clang 9.0.0, Rust-Compiler: rust 1.33.0-dev | rust 1.18.0
[35] ISO 9899 C-Standard Kapitel 6.2.5

## 2.20 FLP37-C

Die Regel FLP37-C bezieht sich auf den Vergleich zweier Gleitkommazahlen und wie dieser durchgeführt werden sollte. Definiert wird FLP37-C durch folgende Aussage:

> ”Do not use object representations to compare floating-point values”

Variablen in C können auf zwei verschiedene Arten abgerufen werden. Entweder es wird der Eintrag im Speicher ausgelesen, welcher durch ein Bit-Pattern kodiert und dort abgelegt wurde oder es wird direkt auf die Variablenwerte zugegriffen.[1]

### 2.20.1 Problembeschreibung

Ein gutes Beispiel ist der Vergleich zwischen 0.0 und -0.0. Während der Wert von beiden Zahlen exakt der Gleiche ist, nämlich 0, wird der Eintrag im Speicher unterschiedlich sein, da das Vorzeichen ebenfalls kodiert wird. Somit würde ein Abgleich der Bitsequenzen im Speicher zu einem falschen Ergebnis führen, obwohl der direkte Vergleich der Variablen mittels Operatoren das korrekte Ergebnis zurückliefert. [1]

Entsprechend sollten Vergleiche mit Operatoren genutzt werden. Das fehlerhafte Verhalten tritt insbesondere beim Abgleich zweier **Not a Number (NaN)** Einträgen auf. Solche entstehen vor allem bei Divisionen, wie 0.0 / 0.0. Es kann dadurch zu unvorhersehbaren Ergebnissen kommen, welche vermieden werden sollten, um den Code sicher zu halten.

### 2.20.2 Vergleiche in C

Hierzu ein Codebeispiel: Zunächst werden zwei float Variablen erstellt, **a** und **b**. Diese enthalten den Wert 0.0f und -0.0f. Anschließend werden sie als konstante void Variablen gecasted, damit der **memory compare (memcmp)** funktioniert. Diese Prozedur ist notwendig, da memcmp keine veränderbaren Daten auswertet. Darauf folgend wird ein Abgleich auf die ersten 10 Bit gestartet und in der Variable result gespeichert.

Der Code hierzu sieht folgendermaßen aus:

```
1 //Initialisierung von Variablen result und Floats a,b mit 0.0 und -0.0
2 int result;
3 float a = 0.0f;
4 float b = -0.0f;
5 //Const pointer erzeugen
6 const void *n1 = (void*)&a;
7 const void *n2 = (void*)&b;
8 //Ergebnis erzeugen und speichern
9 result = memcmp(n1, n2, 10) //memory compare von n1 und n2 auf 10 Bitstellen
```

Listing 61: Memory Compare Gleitkommazahlen

Wird der Code ausgeführt und das result überprüft, gibt das Programm als Rückgabe zurück, dass a und b nicht übereinstimmen. Da es allerdings keinen Unterschied zwischen 0.0 und -0.0 gibt, müsste eigentlich eine Übereinstimmung vorhanden sein.

Hier ist ein Beispiel zu sehen, bei dem nur Operatoren genutzt werden:

```
1 float a = 0.0f;
2 float b = -0.0f;
3 //Vergleich mit if-Statement und == Operator
4 if(a==b){
5   printf("a und b stimmen überein\n");
6 }
```

Listing 62: Vergleich durch Operanden

Wird dieser Code ausgeführt, liefert das Programm die richtigen Ergebnisse zurück.

### 2.20.3 Vergleiche in Rust

In Rust wird diese Problematik in der Regel nicht auftreten. Dies liegt daran, dass die Standardbibliotheken von Rust gar keine Methode bieten, um einen memory compare zwischen zwei Floats zu erzeugen. Stattdessen verwendet Rust ein Verfahren, welches die Werte der Gleitkommazahlen zunächst rundet, bis sie einen darstellbaren Wert bilden. Diese Werte werden dann verglichen. Dies geschieht mit Hilfe von Operanden, wie bei C, wobei dieser Vergleich fehlschlägt, sobald nur ein minimaler Unterschied zwischen den Werten besteht. Da Rust aber meistens Werte rundet, tritt dies häufiger auf, als gewünscht. Deswegen, gibt es noch andere Verfahren, welche die gerundeten Werte vergleichen können.[2]

Die Funktion **ApproxEqUlps** gibt aus, dass zwei Werte sich gleichen, wenn die Anzahl an repräsentierbaren Floats zwischen den beiden Werten unterhalb einer bestimmten Anzahl liegt.[2]

Im Gegensatz dazu erklärt **ApproxEqRatio** zwei Werte als gleich, wenn das Verhältnis des Unterschiedes zwischen zwei Floats geringer ist, als eine festgelegte Grenze.[2]

### 2.20.4 Fazit

Rusts Beschränkung, nicht den Speicherbereich vergleichen zu können, löst zwar die Problematik von C, kann allerdings zu Ungenauigkeiten bei Vergleichen von Gleitkommazahlen führen. Die Problematik in Rust ist allerdings anderer Natur und lässt sich durch ausgiebiges Testen womöglich beheben.

[1]https://wiki.sei.cmu.edu/confluence/display/c/FLP37-C.+Do+not+use+object+representations+to+compare+floating-point+values
[2]https://mikedilger.github.io/float-cmp/float_cmp/index.html

## 2.21 ARR30-C

Die Definition der Regel ARR30-C ist „Do not form or use out-of-bounds pointer or array subscripts".[36]

### 2.27.1 Problembeschreibung

Pointer und Indizes zeigen auf Speicherinhalte. Verschiebt man diese jedoch zu weit, können Daten erreicht werden, welche nicht für den Zugriff bestimmt waren. ARR30 beschreibt dieses Problem der sogenannten Out-of-bounds Pointer und Indizes und zeigt Möglichkeiten, diese zu verhindern.

### 2.27.2 Analyse in der Programmiersprache C

Eines der simpelsten Beispiele ist das Erstellen eines Out-of-bound Pointers, welcher auf ein Array zeigen soll. Im unterliegenden Beispiel wurde darauf geachtet, dass dem Pointer maximal die Anzahl der Stellen im Array übergeben wird. Jedoch schließt das Beispiel negative Zahlen nicht aus. Wird beispielsweise -1 übergeben, zeigt der Pointer auf das Datum, welches vor dem Array abgespeichert wurde. Dies führt zu einem undefinierten Verhalten des C-Compilers. Ein Angreifer könnte den Pointer ausnutzen um die dort liegenden Speicherdaten auszulesen oder dorthin zu schreiben.

```
1 int array[30];
2 int *tabellenwert(int offset) {
3   if(offset < 30) {
4     return table + offset;
5   }//[...]
6 }
```

Listing 63: Unsicherer Code in C

C bietet zwei verschiedene herangehensweisen, dieses Regel einzuhalten. Einerseits könnte man der If-Bedinung eine zweite Abfrage hinzufügen, welche auch negative Werte außerhalb des Arrays abgreift, andererseits könnte offset auch als *unsigned type* abgespeichert werden, und wäre damit nicht negativ.[37]

```
1 // int *tabellenwert(size_t offset){
2   if (0 <= offset < 30){
3     return table + offset;
4   }
5 //}
```

Listing 64: Sicherer Code in C

Die Regel bezieht sich jedoch nicht nur auf Pointer. Ein Zugriff auf ein Array mit einem zu großen oder kleinen Index führt ebenso zu undefiniertem Verhalten in C. Ein Beispiel wäre ein Array mit Größe 3 mit dem Index -1 aufzurufen. Dies kann wie im obigen Beispiel mit einer If-Bedinung eingegrenzt werden.

### 2.27.3 Analyse in der Programmiersprache Rust

Rust besitzt zwei verschiedene Pointerarten. Einerseits gibt es Raw-Pointer. Diese besitzen bereits eine Offset-Methode. Jedoch kann diese nur als *unsafe* ausgeführt werden und birgt die gleichen Probleme wie im C-Beispiel. Wird versucht den Code ohne *unsafe* auszuführen, gibt der Compiler einen Fehler aus, welcher besagt die Methode nicht in der sicheren Umgebung ausgeführt werden kann.[38]

```
1 fn main() {
2   let array= [3, 2, 1] ;
3   let pointer = array.as_ptr();
4   println!("{}", unsafe { *pointer.offset(-1) });
```

[36] https://wiki.sei.cmu.edu/confluence/display/c/ARR30-C.+Do+not+form+or+use+out-of-bounds+pointers+or+array+subscripts

[37] Quelle:https://wiki.sei.cmu.edu/confluence/display/c/ARR30-C.+Do+not+form+or+use+out-of-bounds+pointers+or+array+subscripts#ARR30-C.Donotformoruseout-of-boundspointersorarraysubscripts-FormingOut-of-BoundsPointer

[38] Quelle: https://doc.rust-lang.org/std/primitive.pointer.html

```
5   return;
6 }
```

Listing 65: Unsicherer Code in Rust

Desweiteren gibt es Smart Pointer. Diese erlauben es, sicher auf ein Array zuzugreifen. Hier zu erwähnen ist der Smart-Pointer Box, welcher Speicher auf dem Heap allokiert und ausschließlich diesen zur Verwendung beiträgt. Versucht man nun, mit negativen Werten auf das Array (bzw. den Speicher davor) zuzugreifen, wirft das Programm einen Fehler während des Kompilierens. Gibt man einen zu großen Wert an, *paniced* das Programm während der Laufzeit und beendet sich.[39]

```
1 fn main(){
2   let array: Box<[u32]> = Box::new([3, 2, 1]);
3   println!("{}",array[-1]) //anstelle von -1 führt bspw. 5 zu panic während der
      Laufzeit
4   return;
5 }
```

Listing 66: Sicherer Code in Rust

Möchte man ein Array erstellen, welches zur Laufzeit seine Größe noch ändern kann, sollte ein Vektor (Vec<T>) verwendet werden.[40] Zugriffe auf Arrays ohne Pointer, werden bei Indizes, welche außerhalb des Arrays liegen werden stets abgefangen.

### 2.27.4 Fazit

Falls ARR30-C nicht angewandt wird, kann es zu Komplikationen und Sicherheitslücken kommen, mit welchen unkontrollierter lesender und schreibender Zugriff möglich ist. Jedoch ist dies meist durch einfache Abfragen im Code zu verhindern. Rust bietet gute Alternativen, um die Verwendung von rare-Pointern zu reduzieren.

[39] Quelle:https://doc.rust-lang.org/book/ch15-01-box.html?highlight=box#enabling-recursive-types-with-boxes
[40] Quelle: https://doc.rust-lang.org/1.29.0/reference/types.html?highlight=dynamic,vec,array#array-and-slice-types

## 2.22 ARR32-C

Die Regel **ARR32-C** ist wie folgt definiert:

> “Ensure size arguments for variable length arrays are in a valid range.“

### 2.22.1 C

Variable length arrays (VLA) haben eine zur Laufzeit bestimmte Länge, die also nicht durch einene konstanten Wert definiert ist. Nicht jeder C Compiler unterstützt VLAs. Ist es nicht unterstützt, dient das Macro ”__STDC_NO_VLA__” zur Überprüfung.

```c
#include <stddef.h>
extern void do_work(int *array, size_t size);
void func(size_t size) {
    int vla[size];
    do_work(vla, size);
}
```

Listing 67: Naive C-Version

”size_t” soll laut INT01-C verwendet werden, um negative Werte auszuschließen. Der Wert 0 und zu hohe Werte werden dadurch nicht verhindert. Ein zu hoher Wert kann dazu führen, dass kritische Programmdaten überschrieben werden können.

Als Lösung soll die Größe innerhalb valider Grenzen liegen. Eine Größe über 0 und der Overflow von dem Produkt sizeof(int) * size können durch eine einfache Abfrage verhindert werden. Oberhalb einer benutzerdefinierten Grenze wird der Speicher dynamisch alloziert, für den Fall, dass der Speicher nicht ausreicht.

```c
#include <stdint.h>
#include <stdlib.h>
enum { MAX_ARRAY = 1024 };
extern void do_work(int *array, size_t size);
void func(size_t size) {
    if (0 == size || SIZE_MAX / sizeof(int) < size) {
        /* Handle error */
        return;
    }
    if (size < MAX_ARRAY) {
        int vla[size];
        do_work(vla, size);
    } else {
        int *array = (int *)malloc(size * sizeof(int));
        if (array == NULL) {
            /* Handle error */
        }
        do_work(array, size);
        free(array);
    }
}
```

Listing 68: Sichere C-Version

Ein weiteres Beispiel für dieses Problem ist, wenn ein Array zur Laufzeit alloziert wird und der sizeof()-Ausdruck den Speicherbedarf feststellen soll. Am vereinfachten Beispiel:

```c
typedef int A[rows][cols];
A *array = malloc(sizeof(A));
```

Listing 69: C - sizeof()

Der Rückgabewert von sizeof() könnte am Beispiel eines zweidimensionalen Arrays A[rows] [cols] überlaufen und unter dem mathematischen Produkt rows * cols * sizeof(int) liegen.

### 2.22.2 Rust

In Rust haben alle Arrays eine konstate Länge. Es gibt jedoch eine ähnliche Speicherform, die effektiv VLAs repräsentieren, nämlich Vektoren. Als vergleichbarer Typ zu ”size_t” wird hierzu ”usize” verwendet. Im Gegensatz zu C wird der Nutzer zur Nutzung von ”usize” vom Compiler gezwungen.

Vektoren zeigen laut Dokumentation nicht immer auf einen allozierten Speicherplatz, besonders bei einer Länge von 0. Bei Zugriff auf einen invaliden Speicherplatz wird eine Panic ausgelöst.

```
let v = vec![0, 2, 4, 6];
println!("{}", v[6]); // it will panic!
```

Listing 70: Rust - Index Out of Bounds

Da die Länge von Vektoren (im Gegensatz zu C-Arrays) nach der Initiierung verändert werden kann, wird auch bei der Reservation über den Kapazitäten von usize eine Panic ausgelöst.

Das sizeof()-Problem konnte in Rust nicht mit Vektoren repliziert werden, da die Länge eines Vektors kein Teil dessen Typs ist (z.B. Typ Vec<Vec<usize>>). Bei einem Array mit konstanter Größe wird ein Compiler-Fehler geworfen: ”error: the type ‘[[usize; 10000000]; 10000000]‘ is too big for the current architecture”.

```
use std::mem;
struct VecWrapper {
    vector: [[usize; 10000000]; 10000000],
}
fn main() {
    let size = mem::size_of::<VecWrapper>();
    println!("{}", size);
}
```

Listing 71: Rust - size_of()

### 2.22.3 Fazit

Rust fängt die in C auftretenden Risikofälle ab und wirft gegebenenfalls Fehlermeldungen zur Laufzeit. Es kann natürlich trotzdem noch zu Array-Zugriffen-”Out of Bounds” etc. kommen, jedoch stellen diese keine Sicherheitslücken dar. Diese sollten trotzdem abgefangen werden, um die Stabilität der Software zu steigern.

## 2.23 ARR36-C

Die Regel **ARR36-C** ist wie folgt definiert:

> "Do not subtract or compare two pointers that do not refer to the same array"

Diese Regel besagt, dass bei einer Differenzbildung zweier Zeiger ausschließlich entsprechende Elementreferenzen verwendet werden müssen, welche demselben Array-Objekt zugehören bzw. nur maximal ein Element über dem letzten Eintrag des Arrays hinausragen. Anderweitige Implementierungen bzw. fehlerhafte Programmkonstrukte resultieren in einem undefinierten Verhalten, welches bei der Subtraktion von nicht im Array-Objekt inkludierten bzw. unmittelbar darüber hinausragende Zeiger voneinander abzieht und ohne Validierung fortfährt [41]. Dadurch können unter anderem durch Neuanordnungen von Variablen im Speicher, ausgehend und abhängig vom Compiler, die Speicheradressen nicht als konstant betrachtet werden und somit für die erwähnte Operation, der Subtraktion zweier Zeiger, nicht herangezogen werden. Ein weiteres undefiniertes Verhalten entsteht auf ähnliche Weise durch den Vergleich von Zeigern durch die Vergleichsoperatoren <, <=, >= und >, welche sich nicht im gleichen Array befinden. Der Grund dafür ist, dass sich Zeiger, welche nicht auf dasselbe Aggregat, Vereinigung oder Array-Objekt zeigen, ohne weitere Bedingungen durch die Vergleichsoperatoren vergleichen lassen [41].

### 2.23.1 C

Für die beschriebene Problematik der Vergleichsoperatoren wird in Listing 72 ein nicht konform gemäßes Beispiel herangezogen, welches durch den Vergleich zweier Zeiger bestimmen soll, ob die maximale Anzahl an Array-Einträgen bereits erreicht ist.

```
1 enum { SIZE = 32 };
2 void func_noncompliant(void) {
3   int end, array[SIZE], is_free_space;
4   int *element_ptr = array;
5   /* Increment element_ptr as array fills */
6
7   is_free_space = &end > element_ptr;
8 }
```

Listing 72: Unsichere C-Version

Das Programm nimmt an, dass sich die Variable **end** zu jedem Zeitpunkt der Programmausführung oberhalb und somit benachbart zur Array-Variable **array** befindet. Aufgrund der Berechtigung eines Compilers für das Einfügen von Padding-Bits bzw. durch eine Neuanordnung der Variablen wird der Code-Abschnitt als nicht konform eingestuft. Die konforme Lösung, abgebildet durch das Listing 73, für die Bestimmung einer bereits erreichten maximalen Anzahl an Array-Einträgen wird anhand der Speicheradresse des Objekts, welches sich direkt benachbart zum letzten Element sowie außerhalb des Array-Objekts befindet, ermittelt. Dabei wird die Speicheradresse mit der Adresse des zuletzt eingefügten Elementes verglichen.

```
1 enum { SIZE = 32 };
2 void func_compliant(void) {
3   int array[SIZE], is_free_space;
4   int *element_ptr = array;
5   /* Increment element_ptr as array fills */
6
7   is_free_space = &(array[SIZE]) > element_ptr;
8 }
```

Listing 73: Sichere C-Version

[41] https://wiki.sei.cmu.edu/confluence/display/c/CC.+Undefined+Behavior

### 2.23.2 Rust

Die beschriebene Problematik und Umsetzung der Codestruktur wird im Folgenden mithilfe der Programmiersprache Rust verglichen. Auch mit Rust können die Zeiger nur innerhalb eines allokierten Objekts sinnvoll miteinander verglichen bzw. voneinander abgezogen werden [42]. Somit sind undefinierte Verhalten durch den Einsatz von Rust in Bezug auf die Subtraktion als auch den Vergleich von Zeigern nicht ausgeschlossen. Rust bietet dennoch Methoden für Zeigeroperationen, welche sowohl innerhalb der Objektgrenzen als auch außerhalb der Grenzen von Objekten eingesetzt werden können. Der Zugriff auf ein Array wird bei einem konstanten Index zur Compile-Zeit überprüft [43]. Bei dynamischen Zugriffen wird die Überprüfung der Grenzen zur Laufzeit ausgeführt und bei ungültigen Indizes durch einen *panic*-Zustand repräsentiert. Des Weiteren verfügt Rust über die Möglichkeit ein Array zuzuschneiden, was generell als Slicing verstanden wird. Das Listing 74 zeigt ein in Rust implementiertes und konformes Programm zur Feststellung noch freier Array-Elemente.

```
1 enum Global { SIZE = 32 }
2 fn compliant_solution() {
3   let mut array: [u32; Global::SIZE as usize] = [0; Global::SIZE as usize];
4   let array_ptr: *const u32 = array[Global::SIZE as usize..].as_ptr();
5   let mut element_ptr: *const u32 = array.as_ptr();
6   let is_free_space;
7   /* Increment element_ptr as array fills by slicing the array with the current
      position as index. Example: element_ptr = array[CUR_POS..].as_ptr(); */
8   is_free_space = array_ptr > element_ptr;
9 }
```

Listing 74: Rust-Version

### 2.23.3 Fazit

Durch die automatische Grenzüberprüfung auf Array-Operationen stellt Rust eine sichere und unproblematische Variante gegenüber der Implementierungsweise von C dar. Dennoch können mit Rust Zeiger miteinander verglichen werden, welche nicht demselben Objekt zugehören und schließen somit ein undefiniertes Verhalten nicht aus.

[42] https://doc.rust-lang.org/std/primitive.pointer.html
[43] https://doc.rust-lang.org/reference/expressions/array-expr.html

## 2.24 ARR37-C

Die Regel besagt, dass die Pointer Arithmetik nur für Pointer ausgeführt werden darf, die auf Elemente von Array-Objekten verweisen. Der C Standard (6.5.6) garantiert nur für ein Array, dass alle seine Elemente im Speicher direkt hintereinander liegen. Die Verteilung anderer Typen im Speicher ist im C-Standard hingegen nicht definiert. Dies bedeutet, dass nicht vorhergesagt werden kann, ob sie im Speicher aufeinander folgen.
In den folgenden Unterkapiteln wird zuerst in C ein nicht-kompatibles Beispiel beschrieben und danach wie dies kompatibel gemacht werden kann. Zu guter Letzt noch ein Beispiel in Rust.

### 2.24.1 C

Im folgenden Beispiel wird gezeigt, wie mit Hilfe der Zeigerarithmetik auf struct-Elemente zugegriffen wird. Hierbei wird nicht garantiert, dass die struct-Mitglieder zusammenhängend sind.

```c
struct numbers {
  short num_a, num_b, num_c;
};

int sum_numbers(const struct numbers *numb){
  int total = 0;
  const short *numb_ptr;

  for (numb_ptr = &numb->num_a; numb_ptr <= &numb->num_c; numb_ptr++) {
    total += *(numb_ptr);      // Aufsummieren durch Pointerzugriff
  }

  return total;
}

int main(void) {
  struct numbers my_numbers = { 1, 2, 3 };
  sum_numbers(&my_numbers);
  return 0;
}
```

Listing 75: Nicht kompatibles Beispiel in C

Die Methode liefert nicht das richtige Ergebnis (in diesem Fall 6), sondern einen unvorhersagbaren Wert.

Um den Code kompatibel zu machen, muss die struct so definiert werden, dass sie ein Array-Mitglied enthält. So werden die Nummern in einem Array und nicht in einer struct gespeichert. Hierbei wird garantiert, dass die Array-Elemente im Speicher zusammenhängend sind.

```c
struct numbers {
  short a[3];              // struct besteht nun aus einem Array
};

int sum_numbers(const short *numb, size_t dim) {
  int total = 0;
  for (size_t i = 0; i < dim; ++i) {
    total += numb[i];
  }

  return total;
}

int main(void) {
  struct numbers my_numbers = { .a[0]= 1, .a[1]= 2, .a[2]= 3};
  sum_numbers(my_numbers.a, sizeof(my_numbers.a)/sizeof(my_numbers.a[0]));
  return 0;
```

```
18 }
```

Listing 76: Kompatibles Beispiel in C

### 2.24.2 Rust

In Rust wird Arithmethik auf Pointern generell als unsicher betrachtet. Demnach lässt der Compiler weder Addition noch Subtraktion auf Pointern zu, unabhängig davon, ob es ein Array-Pointer oder ein Pointer zu einem anderen Typen ist. Dieses Verhalten spiegelt die „secure by default“-Mentalität von Rust wieder. Möchte man dennoch Pointer-Arithmetik unter Rust betreiben, kann auf die „offset()“-Methode eines Pointers zurückgegriffen werden. Diese gilt jedoch als unsicher und muss dementsprechend durch das „unsafe“-Keyword gekennzeichnet werden. Allerdings übernimmt der Compiler keine Garantie mehr, dass der Code sicher ausgeführt werden kann.[44]

```
1 let s: &str = "123";
2 let ptr: *const u8 = s.as_ptr();
3
4 unsafe {
5   println!("{}", *ptr.offset(1) as char);
6   println!("{}", *ptr.offset(2) as char);
7 }
```

Listing 77: Beispiel in Rust

### 2.24.3 Fazit

Im Vergleich zwischen C und Rust zeigt sich deutlich, dass Rust primär auf sicheren Code abzielt. Durch die „secure by default“-Metanlität wird der Code geschützt, indem das „unsafe“-Keyword eingebunden wird. Im Gegensatz zu C muss die Pointer-Arithmetik bei Verwendung explizit gekennzeichnet werden.

[44] Quelle: https://doc.rust-lang.org/std/primitve.pointer.html

## 2.25 ARR38-C

Die im SEI CERT C Coding Standard definierte Regel ARR38-C besagt, das sicherzustellen ist, das Bibliotheksfunktionen keine ungültigen Pointer formen. Dies bezieht sich aber nicht nur auf Bibliotheksfunktionen, sondern auf den Umgang mit allen Funktionen und Pointern bzw. Arrays im allgemeinen.

### 2.25.1 Erklärung

Vereinfacht darstellen lässt sich diese Regel anhand des folgenden Programmes, bei dem die gleiche Funktion auf zwei Arrays mit unterschiedlichen Typen angewendet wird.

```c
#include <wchar.h>
#include <stdio.h>

int main(void){
    const char    s1[] =  "Hello World";
    const wchar_t s2[] = L"Hello World";
    printf("sizes char:%zu; wchar:%zu\n",sizeof(s1), sizeof(s2));// 12 48
}
```

Listing 78: Charactergrößen

Dieses Programm gibt die Menge an Bytes aus, die durch die beiden Strings verwendet werden. Auf einem 64-bit Linux System benötigt der erste String 12 Byte und der zweite 48 Bytes. Der Unterschied basiert auf der Eigenschaft, dass der Typ `wchar_t` auf dem genannten System mit 4 Bytes definiert ist. Dies bedeutet, dass wenn ein Programm dazu in der Lage sein sollte die verschiedensten Sprachen zu verarbeiten, darauf geachtet werden sollte, die richtigen Bibliothektsfunktionen aufzurufen.

```c
#include <wchar.h>
#include <string.h>
#include <stdio.h>

int main(void){
    const char    s1[] =  "Hello World";
    const wchar_t s2[] = L"Hello World";
    printf("s1: \"%s\"\ns2: \"%ls\"\n", s1, s2);
    puts("Lengths of the strings with different functions:");
    printf(" strlen s1=%lu; s2=%2lu\n", strlen(s1), strlen(s2));// 11  1
    printf("wstrlen s1=%lu; s2=%lu\n",  wcslen(s1), wcslen(s2));// 10 11
    printf("correct s1=%lu; s2=%lu\n",  strlen(s1), wcslen(s2));// 11 11
}
```

Listing 79: Richtige und falsche Funktionsaufrufe

Der aufgeführte Code demonstiert das zuvor vorgestellte Problem, indem des die Auswirkungen von falschen Aufrufen wirkungsvoll darstellt. Es ist hierbei deutlich zu erkennen, das bei der Wahl der falschen Funktion unerwartete Ergebnisse zurückgegeben werden, die zu weiteren Problemen wie Datenkorruption, Programmabstürzen oder möglicherweise sogar Sicherheitslücken führen können.

### 2.25.2 Rust

In Rust existiert das Problem in dieser Form nicht. Der Grund hierfür ist, das an Funktionen übergebene Arrays nicht zu Pointern “zerfallen“. Dies hat zur Folge, das in Rust auch stets die Länge des Arrays zur Verfügung steht und nicht als zusätzlicher Paramter übergeben werden muss. Dies ist zwar nicht absolut notwendig, da so in einer sehr kleinen Anzahl von Fällen die Länge des Arrays unnötigerweise implizit mit übergeben wird, da die Länge des übergebenen Arrays möglicherweise

auf einer Konstanten basiert. Ein Beipiel hierfür ist ein Array, das einen Punkt in einem zwei dimensionalen Raum repräsentiert. Aber selbst in diesen Fällen ist die Angabe der Länge zu akzeptieren, wenn dies hilft, die sehr häufig auftrenten Probleme im Umgang mit extra Paramtern für die Länge des Arrays zu verhindern. Listing 80 zeigt hierzu am Ende auch die Funktion zur Ausgabe der Länge eines Arrays.

```
fn main() {
    let s1 = "Hello World";
    let s2 = "Hello Worldß";
    println!("s1: {}\ns2: {}", s1, s2);
    println!("Lengths of the strings with different functions:");
    println!("len    s1={}; s2={}", s1.len(), s2.len());// 11 13
    println!("chars  s1={}; s2={}",                     // 11 12
             s1.chars().count(), s2.chars().count());
    println!("len array={}", [1, 2, 3].len());          //  3
}
```

Listing 80: Zeichenlänge in Rust

Ein in Rust möglicherweise auftretendes Problem ist der für manche Programmierer unerwartete Unterschied zwischen der Länge eines Strings under der Menge an Bytes, die den jeweiligen String repräsentieren. Dies wird in Listing 80 deutlich. Die beiden Strings unterscheiden sich nur durch das einzelne Zeichen "ß". Trotzdem benötigt der zweite String zwei Bytes mehr. Außerdem zeigt sich, das in Rust die Funktion `len()` für einen String die Menge der benötigten Bytes ausgibt und nicht die Anzahl der Zeichen. Um die Zahl der im String enthaltenen Zeichen abzurufen, werden die beiden Funktionen `chars()` und `count()` benötigt. Die Erklärung für dieses Verhaltens ist die Verwendung von UTF-8 zur Kodierung von Zeichen, weshalb Zeichen mit einem mehrfachen von einem Byte kodiert sein können.

### 2.25.3 Fazit

Ein Unterschied zwischen den beiden Sprachen ist, das wegfallen von Bibliothektsfunktionen zur Bestimmung der Länge von Strings und Arrays, da die Funktionen direkt über das Objekts aufgerufen werden können. Die Verwendung von freistehenden Funktionen wie in C ist daher nicht notwendig, wodurch auch die Gefahren des verwechselns der Funktionen entfallen. Die in Rust zur Verfügung gestellten Funktionalitäten stellen einen großen Sicherheitsgewinn dar, der viele in C vorgekommenen Probleme damit verhindern dürfte. Dadurch stellt sich die Sprache in eine gute Position um fehlerunanfälligere Progeramme zu entwickeln. Gleichzeitig bietet Rust falls notwendig über die Verwendung von `unsafe` auch die Möglichkeit teils notwendige Optimierungen im Programmcode zu implementieren.

## 2.26 ARR39-C

Die Regel **ARR39-C** definiert sich wie folgt: In der Programmiersprache C, sollen keine skalierten Integerwerte zu einem Zeiger addiert bzw. davon subtrahiert werden. Im Folgenden wird erläutert wie sich die Regel in der Programmiersprache C definiert und es wird dargelegt, ob die Regel auch in Rust Anwendung finden kann.

### 2.26.1 C

Werden Zeiger um feste Werte erhöht bzw. verringert, spricht man von sogenannter Zeigerarithmetik. Wird durch Zeigerarithmetik beispielsweise der Anfangszeiger (Zeiger auf erstes Element) eines Arrays um den Integer-Wert »2« erhöht, führt der Compiler eine automatische Skalierung des verwendeten Wertes aus und ermöglicht den Zugriff auf das dritte Element des Arrays. Auf einem System mit 32-Bit int-Werten, werden somit vier Byte je Array-Wert, für die Speicherung benötigt. Der Zugriff auf das dritte Element eines int-Arrays erfolgt somit durch einen 8-Byte großen Offset, ausgehend von der Startadresse des ersten Elements. Die Abbildung des Integer-Wertes »2« auf den resultierenden 8-Byte-Offset nennt man Skalierung (engl. scaling).

Da durch Zeigerarithmetik bereits eine Skalierung des entsprechenden Wertes durchgeführt wird, ist eine zusätzliche Skalierung durch den »sizeof« Operator, wie in Listing 81 nicht notwendig und davon abzuraten.

```c
enum { BUFSIZE = 5 };
int main(void) {
    int buf[BUFSIZE] = {1, 2, 3, 4, 5};
    int *ptr = buf;
    printf("Start-Adresse Buffer: %p\n", buf);
    while (ptr < (buf + sizeof(buf))) { // Doppelte Skalierung eines Wertes
        printf("Adresse %p | Wert %d\n", ptr, *ptr);
        ptr++;
    }
    return 0;
}
```

Listing 81: Unsichere C-Version

Eine entsprechende Implementierung kann sogenannte »Out-of-Bounds Lese- und Schreiboperationen« ermöglichen. Dadurch kann unter Umständen zusätzlicher Speicherbereich ausgelesen bzw. überschrieben werden[45].

In Listing 82 ist in der Bedingung der While-Schleife eine einfache Skalierung des zu verwendeten Integer-Wertes zu sehen. Neben dem »sizeof« Operator führt auch das Makro »offsetof« eine Skalierung, entsprechend des Datentyps der erhaltenen Variablen aus. Bezieht sich der sizeof-Operator auf ein short-Array, resultiert hierbei eine Skalierung von 2-Byte, ein int-Array hingegen liefert eine Skalierung von 4-Byte.

```c
    /* ... */
    while (ptr < (buf + BUFSIZE) { // Einfache Skalierung eines Wertes
        printf("Adresse %p | Wert %d\n", ptr, *ptr);
        ptr++;
    }
    /* ... */
}
```

Listing 82: Sichere C-Version

[45] https://cwe.mitre.org/data/definitions/468.html

### 2.26.2 Rust

In Rust ist die Funktion »offset()« für die Durchführung von Zeigerarithmetik verantwortlich (siehe Listing 83). Ebenso wie in C wird eine Skalierung an dem erhaltenen Wert vorgenommen. Eine zusätzliche Skalierung ist daher nicht notwendig. Die Funktion setzt hierbei voraus, dass der Quellcode in einem »unsafe«-Abschnitt implementiert wird.

```
fn main() {
    let buf: [u32; 5] = [1, 2, 3, 4, 5];
    let mut ptr1: *const u32 = buf.as_ptr();
    unsafe {
        let ptr2: *const u32 = buf.as_ptr().offset(buf.len() as isize);
        //let ptr2: *const u32 =
        //buf.as_ptr().offset(mem:size_of_val(&buf) as isize);
        while ptr1 < ptr2 {
            println!("Adresse {:?} | Wert {}", ptr1, *ptr1);
            ptr1 = ptr1.offset(1);
        }
    }
}
```

Listing 83: Rust-Implementierung

Der Einsatz eines »unsafe« Codeblocks gestattet unter anderem, dass Funktionen von »Unsafe-Rust« ausgeführt werden können. Überprüfungen in Bezug auf Speichersicherheit (engl. Memory Safety) [46] werden bei diesen Operation, durch den Compiler, nicht durchgeführt.

### 2.26.3 Fazit

Da eine zusätzlich durchgeführte Skalierung eines Integerwertes zu der der Zeigerarithmetik nicht zwingend zu einem Fehler führt (nur bei Speicherbereichsverletzungen) liegt es einzig beim Programmierer diese Regel zu beachten. Der Einsatz von Zeigerarithmetik und somit die doppelte Skalierung von Integerwerten kann in der Programmiersprache Rust durch die Indexierung von entsprechenden Array-Elementen vermieden werden.

[46] https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

## 2.27 STR30-C

Die Regel STR30-C bezieht sich auf die Bearbeitung von String-Literalen in C. Im Folgenden wird genauer auf die Problematik eingegangen und gezeigt, welche Mechanismen zur Problemverhinderung in der Programmiersprache Rust existieren.

### 2.27.1 Problemstellung

Die Arbeit mit Strings erfolgt in C mit Hilfe von Zeigern. Diese verweisen auf den Speicherort der Zeichenkette. Dadurch kann von verschiedenen Variablen auf den gleichen Text referenziert werden. Wird über eine Referenz der Speicher des Textes bearbeitet, so gilt diese Bearbeitung ebenfalls für die anderen Zeiger.

String-Literale werden im C Standard nur bezüglich ihrer Definition beschrieben. Wie die Zeichenkette schließlich im Speicher abgelegt wird, bietet Interpretationsmöglichkeiten für die verschiedenen Compiler. So können für sämtliche Zuweisungen neue Speicherbereiche angelegt werden. Öfter werden identische Literale jedoch nur einmal angelegt (z.B. bei GCC und Clang). Um unbeabsichtigte Veränderung zu verhindern, werden in diesem Fall String-Literale in schreibgeschütztem Speicher abgelegt. Schreibzugriffe auf diesem Speicher führen zu einem Laufzeitfehler. Da die Implementierung nicht im C Standard definiert ist, folgt aus der Bearbeitung von String-Literalen undefiniertes Verhalten (vgl. Listing 84). [47]

```
1 char *textChar = "Hallo\n";
2 textChar[1] = 'e'; // <-- undefiniertes Verhalten
3 printf(textChar);
```

Listing 84: Die Kompilierung über GCC führt zu einem Speicherzugriffsfehler zur Laufzeit. Mit anderen Compilern könnte jedoch der Code fehlerfrei ausführen.

Dieses Problem kann verhindert werden, indem der String in einen eigenen Speicher kopiert wird. Dies kann explizit über `memcpy()` oder implizit über die Initialisierung als Array erfolgen (vgl. Listing 85). Wird der String nicht verändert, so kann auch der Zeiger verwendet werden. Die Variable sollte jedoch als `const` deklariert werden [48]. Diese Regeln gelten ebenfalls für Referenzen auf Teile der Literale, da diese auf den gleichen Speicher verweisen. Solche Referenzen können z.B. über die Suche nach Substrings mit `strstr()` erhalten werden.

```
1 char textStr[] = "Hallo\n";
2 textStr[1] = 'e';
3 printf(textStr); // Ausgabe: Hello
```

Listing 85: Korrektur von Listing 84 über Arrayinitialisierung in Zeile 1.

### 2.27.2 Risikobewertung

Das Problem, das bei der Bearbeitung von String-Literalen auftritt, ist lediglich ein Laufzeitfehler, der das Programm vorzeitig beendet. Allerdings ist die Wahrscheinlichkeit für das Auftreten hoch, da die Unterschiede zwischen sicherem und unsicherem C-Code vor allem für Anfänger nicht offensichtlich sind. Der Fehler kann jedoch mit geringem Aufwand behoben werden. Sollte fehlerhafter Code in einem Abschnitt liegen, der im Regelbetrieb nicht ausgeführt wird, jedoch erreichbar ist, so könnte darüber ein Denial of Service (DoS) Angriff ausgeführt werden. Sollte nur dieser Fehler im Programm vorhanden sein, so sind sensible Daten weiterhin geschützt und Schadcode vom Angreifer wird nicht ausgeführt.

[47] https://en.cppreference.com/w/cpp/language/string_literal
[48] https://www.geeksforgeeks.org/char-vs-stdstring-vs-char-c/

### 2.27.3 Rust

Rust löst dieses Problem indem es grundsätzlich zwei verschiedene Arten von Zeichenketten gibt. Die Klasse `String` und die Klasse `str`. Literale werden als `str` abgespeichert. Objekte dieser Klasse sind nicht modifizierbar (vgl. Listing 86), wodurch mehrfach Referenzen auf die gleiche Zeichenkette ohne Probleme verwendet werden können. Ungültige Schreibzugriffe werden bereits vom Compiler erkannt. Soll die Zeichenkette jedoch bearbeitet werden können, so kann ein `str` über die Funktion `String::from()` zu einem `String` umgewandelt werden. Hierbei wird der Text in einen eigenen Speicherbereich für die Variable kopiert (vgl. Listing 87). [49]

```
1 let text_str = "Hallo";
2 text_str[1] = "e"; // `str` cannot be mutably indexed by `{integer}`
3 println!("{}", text_str);
```

Listing 86: Fehler beim Kompilieren in Zeile 2 wegen dem schreibendem Zugriff.

```
1 let mut text_string = String::from(text_str);
2 String::replace_range(&mut text_string, 1..2, "e");
3 println!("{}", text_string);
```

Listing 87: Korrigierte Funktion aus Listing 86 über die Verwendung des `String`-Typs.

Funktionen, die ähnlich zur Suche von Substrings sind, geben eine schreibgeschützte Referenz vom Typ `str` zurück. Der Schreibschutz wird hierbei über das „Ownership“ und „Borrowing“-System von Rust gewährleistet. Dieses System garantiert beim Kompilieren, dass entweder von nur einer Variable aus auf den Speicher geschrieben werden kann oder von mehreren Referenzen gelesen wird. Die zweite Variante ist hier der Fall. [50]

### 2.27.4 Fazit

Das einmalige Abspeichern von gleichen Strings aus verschiedenen Variablen spart Speicherplatz. Jedoch sollen Strings auch in eigenem Speicher bearbeitet werden können. Somit ist es wichtig, dass beide Speichermöglichkeiten vorhanden sind. Die Bearbeitung von String-Literalen in C zu vermeiden ist nicht schwierig, jedoch muss der Fehlertyp dem Entwickler bekannt sein. In Rust existieren ebenfalls beide Speichermöglichkeiten, jedoch existiert das Problem durch die verschiedenen Datentypen nicht.

[49] https://doc.rust-lang.org/book/ch08-02-strings.html
[50] https://doc.rust-lang.org/book/ch04-03-slices.html

## 2.28 STR31-C

**Regel:** „Garantiere, dass der Speicherort eines Strings genügend groß für Zeichen und das Null-Byte ist"

Bei der Arbeit mit Strings kann es oft zu Pufferüberläufen kommen. Eine Ursache dafür ist, dass das speichern von Daten in einem dafür zu kleinen Puffer. Dies kann zu unvorhergesehenen Effekten führen. Dazu gehört als eine der harmlosen Fehler, der Programmabsturz durch einen Speicherzugriffsfehler. Viel tiefgreifender sind die Auswirkungen, wenn durch den Überlauf andere Daten überschrieben werden. Ein beliebter Angriff ist dabei das überschreiben der Rücksprungadresse zu einem im Speicher befindlichen Schadcode.

### 2.28.1 C

Ein häufig auftretender Fehler ist der sogenannte „Off-by-One Fehler". In einer Schleife wird von `src` nach `dest` kopiert. Wird nun die Größe von `dest` genau so groß wie die Textlänge in `src` ausgelegt, kann es bei dem in Listing 88 gezeigten Beispiel dazu führen, dass das Null-Byte ein Byte nach `dest` geschrieben wird. C bietet hier kein vorhersehbares Verhalten. In manchen Fällen führt das schreiben nach dem eigentlichen Array zu einem Speicherzugriffsfehler, in anderen werden die im Speicher nachfolgenden Daten einfach überschrieben.

```
1 void copy(size_t n, char src[n], char dest[n]) {
2     size_t i;
3
4     for (i = 0; src[i] && (i < n); ++i) {
5         dest[i] = src[i];
6     }
7     dest[i] = '\0';
8 }
```

Listing 88: Fehlerhafte Kopierfunktion

Eine simple Lösung für den Fehler ist, die Schleifenbedingung auf `i < n - 1` (siehe Listing 89) zu ändern. Dadurch wird sichergestellt, auch wenn `src` größer als `dest` ist, dass das Null-Byte immer innerhalb des Arrays geschrieben wird.

```
1 void copy(size_t n, char src[n], char dest[n]) {
2     size_t i;
3
4     for (i = 0; src[i] && (i < n - 1); ++i) {
5         dest[i] = src[i];
6     }
7     dest[i] = '\0';
8 }
```

Listing 89: Korrigierte Kopierfunktion

Viele fehlerhafte Bibliotheksfunktionen wurden mit der Einführung von C11 entfernt. Korrigierte Varianten stehen seit C99 zur Verfügung. Dazu gehört zum Beispiel die Funktion `gets()`, die durch `fgets()` ersetzt wurde.

### 2.28.2 Rust

Rust bietet in der Standardbibliothek eine `String` Implementierung. Mit dieser Implementierung können einfach `Strings` angelegt und bearbeitet werden, ohne dass eine manuelle Speicherverwaltung stattfinden muss. Das in Listing 88 gezeigte Beispiel einer Kopierfunktion ist in der Bibliothek von Rust bereits vorhanden und in Listing 90 aufgeführt.

```rust
fn copy(src: &String) -> String {
    return String::clone(&src);
}
```

Listing 90: String kopieren in Rust

Mit der Verwendung der vorhandenen Bibliotheksfunktionen kann der „Off-by-One Fehler" einfach und sicher vermieden werden.

Auch bei dem Kopieren von einzelnen Charakteren muss nicht auf die Position des Null-Bytes geachtet werden. So funktioniert der in Listing 91 gezeigte Code auch zuverlässig, wenn `src` gleich groß wie `dest` ist.

```rust
fn copy(src: &String,  dest: &mut [char]) {
    let mut counter: usize = 0;
    for c in src.chars() {
        dest[counter] = c;
        counter+=1;
    }
}
```

Listing 91: Einzelne Charakter kopieren in Rust

Sollte der Ausgangsstring größer als der vorhandene Speicher sein, wird bei einem Zugriff außerhalb des Zielarrays eine „Out-Of-Bound" Fehlermeldung erzeugt. Dieses Sicherheitsmerkmal funktioniert auch in einem mit `unsafe` markiertem Code-Block. Prinzipiell sollte in einem unsicheren Abschnitt stets mit besonderer Sorgfalt gearbeitet werden.

### 2.28.3 Fazit

Im Allgemeinen lässt sich feststellen, dass die gesamten `String` Operationen in Rust robuster und einfacher zu handhaben sind. Während der Fehler für C als schwerwiegend und wahrscheinlich eingestuft ist, ist für Rust die Wahrscheinlichkeit, dass der Fehler auftritt, sehr gering.

## 2.29 STR32-C

**Regel:** „Übergebe keine Zeichenfolge, die nicht mit Null abgeschlossen ist, an eine Bibliotheksfunktion, welche eine Zeichenfolge erwartet."

Bibliotheksfunktionen arbeiten oft mit Zeichenketten und nehmen diese als Argument in Form eines String- oder Wide-String-Argumentes entgegen. Dabei ist es wichtig, dass der übergebene String mit Null abgeschlossen ist. Wenn einer Funktion eine Zeichenfolge übergeben wird, welche der Vorgabe der Nullterminierung nicht entspricht, besteht das Risiko, dass ein Fehlverhalten auftritt. Um diese Problematik zu eliminieren, wurde in dieser Regel festgelegt, dass kein `String`, bzw. `Wide-String` übergeben werden soll, der nicht mit Null terminiert, wenn eine Bibliotheksfunktion eine Zeichenfolge dieses Typs erwartet.

### 2.29.1 C

Konkretes Fehlverhalten, dass bei der Missachtung dieser Regel auftreten kann, wird im folgenden Beispiel (siehe Listing 92) dargestellt. Hierbei wird zwei Arrays vom Typ `char` jeweils ein String fester Größe zugewiesen. Die aufgezeigte Lösung ist hierbei jedoch nicht konform. Die Konvention greift den Grundsatz der Empfehlung „STR11-C" auf, in welcher erläutert wird, dass keine Begrenzung der Zeichenfolge angegeben werden soll. Da dies im Codebeispiel (siehe Listing 92) aber der Fall ist, treten hier bei der Ausführung Fehler auf. Genau genommen hat die Missachtung hierbei zur Folge, dass zwar der String abgespeichert wird, jedoch kein verbleibender Platz mehr für die terminierende Null vorhanden und dadurch das Vorgehen auch nicht konform ist.

```
1 int main(void) {
2   char string_one[2] = "Um";        // char-Arr fester Grösse
3   char string_two[4] = "laut";      // char-Arr fester Grösse
4   strcat(string_one, string_two);
5   printf("Wort: %s", string_one);
6 }
```

Listing 92: Nicht konforme Zeichenfolge - C-Code

Wird die Größe des Arrays nicht explizit, sondern ohne Begrenzung angegeben, verschwindet der Error. Der Compiler weiß so, dass er automatisch genügend Speicherplatz zu reservieren hat, um die gesamte Zeichenfolge einschließlich des abschließenden „Nullzeichens" zu sichern. Das folgende Beispiel (siehe Listing 93) zeigt den Codeausschnitt, in dem keine explizite Größe für das Array angegeben wird, damit das Beispiel aus (Listing 92) funktioniert.

```
1   char string_one[] = "Um";          // char-Arr ohne feste Grösse
2   char string_two[] = "laut";    // char-Arr ohne feste Grösse
```

Listing 93: Konforme Zeichenfolge - C-Code

### 2.29.2 Risikobewertung

Bei Weitergabe einer nicht korrekt mit Null terminierten Zeichensequenz an eine Bibliotheksfunktion, kann es zu Speicherüberläufen kommen. Außerdem besteht die Gefahr, dass veralteter Code mit den Berechtigungen des anfälligen Prozesses ausgeführt wird. Weiterhin können auch `Null-Termination-Error` auftreten, bei denen oft unbeabsichtigt Informationen offengelegt werden.

### 2.29.3 Rust

Gegenüber der Programmiersprache C, bei der Strings Arrays vom Datentyp `char` darstellen, [51] gibt es bei Rust eigene „Structs“ (Strukturen). [52]

```
1 fn string_print(string_one: &str, string_two: &str) {
2   print!("{}{}", string_one, string_two);}   // Ausgabe d. Strings "Umlaut"
3 fn main() {
4   string_print("Um", "laut");}   // Aufruf string_print mit 2 Strings
```

Listing 94: Konforme Zeichenfolge - Rust-Code

Das Programm (siehe Listing 94) gibt zwei Zeichenketten des Typs `str`, welche nach Aufruf durch die Funktion `string_print()` und anschließend `print!()`, miteinander verbunden werden, aus. Hierbei handelt es sich um sicheren Code, da sowohl die Datentypen `String`, als auch `str` in Rust gute mit „UTF-8“ codierte Datentypen darstellen und einen ordnungsgemäßen Umgang mit Unicode sowie starke Sicherheitsmaßnahmen gewährleisten. [53] Weiterhin gibt es seitens Rust auch ein „Struct“ namens `CString`. Dabei handelt es sich um eine eigene, „c-kompatible“ Zeichenfolge, wobei eine Instanz dieses Typs eine statische Garantie darstellt, dass darunter liegenden Bytes keine internen „0-Bytes“ enthalten und das letzte Byte kein „0-Byte“ darstellt. [54]

### 2.29.4 Fazit

Allgemein lässt sich festhalten, dass Fehler die durch inkorrekte Nullterminierung entstehen, in erster Linie bei C eine Problematik darstellen. Speicherbereiche können hier überschrieben werden und zu schwerwiegenderen Fehlern führen. Bei Rust ist diese Problematik eher weniger gewichtig, da es hier spezielle „Structs“ gibt, mit denen Zeichenketten sicher verwendet werden können. Solange die von Rust bereitgestellten `WideString` und `CString` Typen, die ein „c-ähnliches“ Verhalten aufweisen, nicht dazu verwendet werden, auf die `String` und `str` Typen unter evtl. Verlust und ohne Kenntnis der Art der Konvertierung zurück zu konvertieren, sind diese sicher. [27]

[51] Quelle: https://doc.rust-lang.org/std/primitve.pointer.html
[52] Quelle: https://doc.rust-lang.org/rust-by-example/custom_types/structs.html
[53] Quelle: https://docs.rs/widestring/0.2.2/widestring/
[54] Quelle: https://doc.rust-lang.org/std/ffi/struct.CString.html

## 2.30 STR34-C

Diese Regel besagt, dass ein `char` zu einem `unsigned char` gecasted werden soll, bevor es zu einem größeren `integer` Datentyp konvertiert wird. Da ein `char` standardmäßig einem `signed char` in dessen Wertebereich, Verhalten und Repräsentation gleicht, gilt diese Regel auch für `signed chars`.

Für die Speicherung eines `chars` werden 1 Byte bzw. 8 Bits reserviert. Dadurch sind prinzipiell $2^8$ = 256 Werte möglich. Bei `signed chars` wird dagegen das führende Bit als Vorzeichen interpretiert, wodurch man einen Wertebereich von -128 bis +127 erhält. Diese zwei Interpretierungsmöglichkeiten können zu ungewolltem Verhalten führen. Im Folgenden wird diese Situation in den Programmiersprachen C und Rust untersucht.

### C

Um das Problem in C zu veranschaulichen, wurde in dem Code-Snippet aus Listing 95 die Situation künstlich erzeugt, bei dem jeweils ein `signed char` und ein `unsigned char` den Dezimalwert 255 aus einer Integer-Variablen erhält. Bei dem `printf` Statement wird nun wieder der Integer-Wert der Characeter ausgegeben. Handelt es sich nun um Char ohne Vorzeichen, so bleibt der Wert bei 255. Bei einem Char mit Vorzeichen allerdings ist der Wert nun -1. Das kann dann vor allem beim Lesen von Dateien zu unvorhersehbarem Verhalten führen. Dabei liest man eine Datei solange, bis man `EOF` erreicht. Dieses Schlüsselwort signalisiert das Ende einer Datei und besitzt per Definition den Wert -1.

```c
#include<stdio.h>

int main() {
  int c_val = 255;  // value of char

  signed char c_s = (signed char) c_val;
  unsigned char c_u = (unsigned char) c_val;

  printf("%u", c_u);  // Output: 255
  printf("%d", c_s);  // Output: -1

  bool b = false;
  if (EOF == c_u) {
    b = true; // Output: EOF true
  }

  printf("EOF reached: %d", b); // Output: EOF reached: true

  return 0;
}
```

Listing 95: Casten von chars in C

### Rust

Die Programmiersprache Rust bietet zwei Möglichkeiten zum Casten von verschiedenen Datentypen an. Ein safe cast lässt sich durch das Schlüsselwort `as` erreichen. Da in Rust der Datentyp - ob signed oder unsigned - explizit angegeben werden muss, kann der Compiler anschließend beim Casten schon unvorhersehbares Verhalten verhindern. Eine Variable als `u8` (unsigned 8 Bit) deklariert, kann ohne Probleme gecastet werden. Bei als `i8` (unsigned 8 Bit) deklarierten Variablen wird bereits zu Kompilierzeiten eine Fehlermeldung geworfen.

```rust
fn main() {

    let i_u = 255 as u8;
```

```
4       print!("{}", i_u as char); // Output: 255
5
6       let i_s = 255 as i8;
7       print!("{}", i_s as char); // Output: error
8
9   }
```

Listing 96: Casten von chars in Rust

Die andere Option bietet das Schlüsselwort `transmute` und erlaubt ein arbiträres und somit aber auch unsicheres Casting. Die Dokumentation von Rust weist allerdings explizit darauf hin, dass dies eines der gefährlichsten Features von Rust ist und in den meisten Fällen ausdrücklich vermieden werden sollte.

### Risikobewertung

Der SSEI CERT C Coding Standard"bewertet das Risiko durch das in diesem Kapitel beschriebene Problem als mittel ein. Die Wahrscheinlichkeit für das Auftreten dieses Problems ist auch weder hoch noch gering. Doch da der Aufwand für sicheren Code in diesem Fall ebenfalls moderat bleibt, sollten entsprechende Vorkehrungen definitiv berücksichtigt und umgesetzt werden. Außerdem findet der Programmierer bei Bedarf Unterstützung durch Analyse-Programme, welche den Code statisch analysieren und dieses Problem erkennen können.

### Fazit

In C muss der Programmierer selbst darauf achten, die richtigen Datentypen zu wählen bzw. zu den richtigen Datentypen zu casten. Da es nur in Grenzfällen zu Problemen führen kann, ist die Gefahr groß, den Fehler erst spät und zur Laufzeit zu entdecken. In Rust dagegen weist der Compiler den Programmierer sofort mit einer Fehlermeldung darauf hin, dass das Casten von `unsigned chars` zu Integer-Werten nicht möglich ist und verhindert somit Fehlverhalten bereits zur Kompilierzeit. Definitiv ist die Handhabung in Rust eine Erleichterung für den Programmierer und schließt automatisch diese potenzielle Sicherheitslücke.

## 2.31 STR37-C

Arguments to character-handling functions must be representable as an unsigned char.

### 2.31.1 Einleitung

Es gibt eine große Anzahl von Programmierungssprache, angepasst an verschiedene Anwendungen. Jede Programmiersprache hat seine eigene Syntax und spezifische Anweisungen. Nach der Regel STR37-C müssen die Argumente von Zeichenverarbeitungsfunktionen als unsigned char darstellbar sein. Aber ist es auch in Rust die gleiche Regel? Durch diese Regel werde ich einen kleinen Vergleich zwischen den beiden Programmiersprachen C und Rust anstellen.

### 2.31.2 Programmiersprache c

In C deklariert der Header <ctype.h> Funktionen ( tolower, isupper, isprint und viele andere), die als Domäne einen Typ int, dessen Wert als unsigned char (0-255) dargestellt werden kann. Aber wenn das Argument einen anderen Wert hat, ist das Verhalten undefiniert.

#### 2.31.2.1 C-Beispielcode

Im folgenden Beispiel ist der Parameter der Umwandlungsfunktion tolower als const char definiert und dieser Wert möglicherweise nicht als unsigned char dargestellt werden kann.

***Code mit Fehler***

```c
void tolower_char (const char *a, char *b) {
    size_t i; int len=strlen(a); //variable Deklaration
    for(i = 0; i < len ; i++) { // for Schleife
        b[i] = tolower(a[i]); //Fehler
    }
    b[len] = '\0';
}
```

Um die Fehlermeldung zu vermeiden, müssen Sie das Zeichen in unsigned char umwandeln, bevor es als Argument an die Funktion tolower übergeben wird.

***Lösung***

```c
void tolower_char (const char *a, char *b) {
    size_t i; int len=strlen(a); //variable Deklaration
    for(i = 0; i < len ; i++) { // for Schleife
        b[i] = tolower((unsigned char)a[i]); //Umwandlung von a in kleinb
    }
    b[len] = '\0';
    }
```

### 2.31.3 Programmiersprache Rust

Die Regel STR37-C ist bei Rust nicht nötwendig, da Rust problematischen code verhindert. Es bietet eine explizite Sprache, deren code klar aus einem kleinen Kontext erfassbar ist. Dazu ist es ein Ziel des Compilers, möglichst viele Fehler früh zu erkennen und zu beseitigen.
In Rust gibt es eine Reihe von Methoden und Implementierungen, wie (to-lowercase, to-uppercase, is-whitespace und viele andere), die von Unicode-Basiseigenschaften abgeleitet werden und eine Struktur für char erstellen.

#### 2.31.3.1 Rust-Beispielcode

Das folgende Beispiel zeigt die Verwendung von Funktionen to-lowercase und to-uppercase.
To-lowercase gibt einen Iterator zurück, der das Kleinbuchstaben eines Zeichens als ein oder mehrere Zeichen ergibt. Wenn ein Zeichen keine Kleinbuchstaben enthält, wird das gleiche Zeichen vom Iterator. Dies ist sicher, einfach und führt komplexe, bedingungslose Zuordnungen ohne Anpassung durch. Es ordnet ein Unicode Zeichen entsprechend der Unicode Datenbank.

***Funktion to-lowercase***

```
fn main()
    let lowercase_convert = "D".chars() // Initialisierung typ char
        .flat_map(char::to_lowercase) //map und Umwandlung
        .collect::<String>(); //
    println!("{}", lowercase_convert);
}
```

***Funktion to-uppercase***

```
fn uppercase (value: &str) -> String {
    let mut result: String = String::with_capacity(value.len());
    for c in value.chars() { // for schleife
        for low in c.to_uppercase() { // Umwandlung
            result.push(low); } // Daten einlesen
    }
    return result;
}
```

### 2.31.4 Fazit

Durch diese Regel können wir zusammenfassen, dass Rusts sich von C deutlich unterscheidet. Rust verbessert die Low-Level Programmierung durch eine Reihe von Sicherheitsprüfungen während der Kompilierung.

## 2.32 STR38-C

Die versehentliche Übergabe von Zeichenketten mit Zeichen einer bestimmten Breite in Funktionen, welche Zeichenketten mit einer anderen Zeichengröße erwarten, kann zu unerwarteten, undefinierten Laufzeitverhalten der kompilierten Anwendung führen.

### 2.32.1 Problembeschreibung in der Programmiersprache C

In der C-Standard-Bibliothek existieren Zeichenketten vom Typ `char[]` mit einer Zeichenbreite von einem Byte - die mit einem Null-Zeichen terminiert werden - und der Breitzeichenkettentyp `wchar_t[]` mit der Zeichengröße von zwei Bytes, der mit zwei Null-Zeichen endet [55] [56]. Aufgrund der verschiedenen Zeichenbreiten können durch die Übergabe von Zeichenketten des Typs `char[]` in Funktionen, die Breitzeichenketten des Typs `wchar_t[]` erwarten - oder die Übergabe von Zeichenketten mit breiten Zeichen in Funktionen, welche Zeichenketten mit normaler Zeichenlänge erwarten - mit einer hohen Wahrscheinlichkeit während der Laufzeit der kompilierten Anwendung Skalierungsprobleme auftreten. Listings 97 bis 99 zeigen die in diesem Sinne unkorrekte - sowie die korrekte - Verwendung von String-Funktionen der C-Standard-Bibliothek und die daraus folgenden Ergebnisse.

```
1 wchar_t wide_str[] = L"1234";
2 // gibt mit dem Compiler gcc 7.3.0 unkorrekterweise 1, statt 4, aus
3 printf("%zu \n", strlen(wide_str));
```

Listing 97: Inkorrekte Verwendung der Funktion `strlen`, welche die Länge einer Zeichenkette zurückgibt, auf eine breite Zeichenkette

```
1 char narrow_str[] = "1236";
2 // gibt mit dem Compiler gcc 7.3.0 unkorrekterweise 11, statt 4, aus
3 printf("%zu \n", wcslen(narrow_str));
```

Listing 98: Inkorrekte Verwendung der Funktion `wcslen`, welche die Länge einer Breitzeichenkette zurückgibt, auf eine Zeichenkette

```
1 char narrow_str[] = "1236";
2 printf("%zu \n", strlen(narrow_str)); // Gibt 4 aus
3 wchar_t wide_str[] = L"1236";
4 printf("%zu \n", wcslen(wide_str)); // Gibt 4 aus
```

Listing 99: Korrekte Verwendung der beiden Längen-Funktionen

Die Nichteinhaltung dieser Regel kann zu Pufferüberläufen, Referenzen zu abgeschnittenen Zeichenketten und anderen Defekten führen. Die Risikobewertung des SEI CERT Coding Standards bewertet das mögliche Schadensausmaß beim Eintreten von Problemen als hoch und die Eintrittswahrscheinlichkeit von Fehlern bei Nichteinhaltung der Regel als wahrscheinlich. Die Beseitigung des Mangels wird als einfach bewertet.

Moderne Compiler und statische Code Analyse Tools (Astrée, PRQA QA-C, RuleChecker, Polyspace Bug Finder, Coverity und Axivion Bauhaus Suite) erkennen den Unterschied zwischen den Typen `char*` und `wchar_t*`, wodurch sie Warnungen ausgeben können. Aktuelle Versionen der Compiler GCC, Clang und MSVC C geben diese Warnungen ohne explizite Konfiguration aus.

### 2.32.2 Lösung des Problems durch die Programmiersprache Rust

Rusts Standardbibliothek hat lediglich einen primitiven nicht veränderlichen UTF-8 kodierten Typ für Zeichenketten namens `str` [57]. Um neue Zeichenketten aus `str`-Instanzen erzeugen zu können, exis-

[55] http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/005_c_basisdatentypen_011.htm
[56] http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/005_c_basisdatentypen_013.htm
[57] https://doc.rust-lang.org/book/ch08-02-strings.html

tiert in Rust das struct `String`, welches aus einem Zeiger zum Puffer, welcher die Bytes der Zeichenkette enthält, einer Kapazität, welche die Puffergröße beschreibt, und der Anzahl der Zeichen einer `String`-Instanz besteht [58].

Der Rust-Compiler Rustc beendet die Kompilierung bei der Übergabe eines nicht erwarteten Zeichenkettentyps in eine Funktion mit einem Fehler. Daher ist die versehentliche Übergabe einer Instanz eines nicht erwarteten String-Typs in Funktionen durch Rusts starke statische Typisierung ausgeschlossen. Da das struct `String` bereits die Methoden zur Manipulation einer `String`-Instanz bereitstellt, wird automatisch die Methode des verwendeten Ziel-String-Typs aufgerufen, wodurch die Übergabe eines String-Parameters - der bei C-Funktionen der Standard-Bibliothek nötig ist [59] - bei Methoden, die nur einen Zielstring benötigen, nicht unterstützt [60] (auch zu sehen im Beispielaufruf von `String.len` in Listing 100). Auf diese Weise kann die Standard-Bibliothek von Rust Funktionsparameter - und somit Möglichkeiten falsche Parameter zu übergeben - verringern.

```
1 let mut s = String::from("Len"); // "Len" hat den primitiven Typ str
2 s.push_str("gth:"); // erweitert den String s um den str "gth"
3 // benötigt keinen Ziel-String-Parameter, da s die Zielinstanz ist
4 println!("{} {}", s, s.len()); // gibt Length: 7 aus
```

Listing 100: String-Methoden in Rust

### 2.32.3 Fazit

Unter Verwendung der C-Compiler GCC, Clang und MSVC C kann man Zeichenketten mit einer unerwarteten Zeichenbreite in Funktionen der C-Standard-Bibliothek übergeben und dadurch undefiniertes, unerwartetes Laufzeitverhalten auslösen. C-Compiler und statische Code-Analyse-Werkzeuge können diese Typunterschiede erkennen und Warnungen ausgeben. Rusts Compiler bricht die Kompilierung, wenn Funktionsparameter mit nicht erwarteten Typen im Quellcode erkannt wurden, aufgrund der starken Typisierung der Programmiersprache ab und umgeht somit diese Problematik.

[58] https://doc.rust-lang.org/std/string/struct.String.html
[59] http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/030_c_anhang_b_020.htm
[60] https://doc.rust-lang.org/std/string/struct.String.html

## 2.33 MEM30-C

Im Zusammenhang mit der Benutzung von Speicher ist darauf zu achten, diesen nach der Verwendung wieder freizugeben. Wird der Speicher zu einem falschen Zeitpunkt oder häufiger als dieser alloziert wurde, freigegeben, entsteht ein undefinierter Zustand, der von Angreifern ausgenutzt werden kann. Das undefinierte Verhalten wird vom C Standard festgelegt. Hier heißt es in Frage 11.33[61] “Anything at all can happen; the Standard imposes no requirements.“. Somit kann alles mögliche passieren, wenn dieses Verhalten erzeugt wird.

### 2.33.1 Risikobewertung

Die erste Schwachstelle des Coding Standards, welche sich mit dem Speicherzugriff auseinandersetzt, MEM-30-C, beschäftigt sich mit dem Zugriff auf bereits freigegebenen Speicher. Die Eintrittswahrscheinlichkeit von Schwachstellen dieser Art wird als “Likely“ bzw. als sehr wahrscheinlich eingestuft, die Schadenshöhe wird mit “High“ betitelt; die Risikoklasse, hier durch eine hohe Priorität (P18) und Level L1 dargestellt, soll dem Software Entwickler verdeutlichen, dass es sich hierbei um ein sehr hohes, nicht akzeptables Risiko handelt und Maßnahmen zur Minderung ergriffen werden müssen, um den sicheren Ablauf der Software zu garantieren. Nachfolgend wird das Problem evaluiert und ein möglicher Lösungsansatz erörtert, um Zugriff auf freigegebenen Speicher zu verhindern.

### 2.33.2 Probleme beim Zugriff auf freigegebenen Speicher in C

Im SEI CERT C Coding Standard wird anhand von Code Snippets verdeutlicht, wie Probleme beim Zugriff auf freigegebenen Speicher in C entstehen können. Grundsätzlich handelt es sich um Probleme bei der Verwendung der Funktion `void free(void* ptr)`. In Tabelle 2 sind diese Beispiele schematisch dargestellt.

Tabelle 2: MEM30-C Beispiele in C

| **Problem** | **Beschreibung** | **Lösungsansatz** |
|---|---|---|
| Verkettete Listen | Freigabe eines Pointers, ohne den Verweis auf das nächste Listenelement zu speichern | Zwischenspeichern des Pointers auf das nächste Element, bevor `free(*ptr)` aufgerufen wird |
| Freigabe des Speichers zum falschen Zeitpunkt | Der Speicher wird nach der Freigabe noch verwendet | Reihenfolge der Aufrufe ändern, Speicher nutzen → Freigeben |
| Mehrfache Freigabe von Speicher (double-free vulnerability) | Speicher wird innerhalb von Schleifen öfter freigegeben als er alloziert wurde | Fallunterscheidung vornehmen und testen, ob der Speicher bereits zu einem vorherigen Zeitpunkt freigegeben wurde |

Im Vergleich zu Sprachen wie Java oder Python, die zur Programmlaufzeit einen Garbage Collector[62] haben, muss das Freigeben von Speicher in C oder Rust explizit durch den Entwickler geschehen. Bei den in Tabelle 2 aufgeführten Beispielen entsteht ein Dangling Pointer, ein Zeiger, der auf einen Speicherbereich verweist, der bereits freigegeben wurde. Dies führt zu nicht definiertem Verhalten des Programms.

[61] http://c-faq.com/ansi/undef.html
[62] https://pcwalton.github.io/2013/03/18/an-overview-of-memory-management-in-rust.html

### 2.33.3 Lösungsansätze der Sprache Rust für das C Speicherproblem

In Rust werden zur Vermeidung solcher Probleme unter anderem die Lifetime[63] und Smart Pointers eingesetzt. Lifetime beschreibt den Zustand, unter dem Variablen ihre Gültigkeit verlieren, wenn sie den Scope verlassen. Hierdurch kann verhindert werden, dass beispielsweise nach der Freigabe des Speichers auf die Referenz zugegriffen werden kann. Es wird dadurch realisiert, dass beim verlassen des Scopes implizit der Trait `drop` ausgeführt wird. Beim Versuch, den Trait manuell aufzurufen, kommt es zu einem Fehler zur Compilezeit. Rust verhindert hiermit das double-free Problem. Zur manuellen Speicherverwaltung gibt es die Möglichkeit, die Funktion `std::mem::drop` aus der Rust-Standardbibliothek zu verwenden, um die Speicherfreigabe zu erzwingen. Der Compiler erkennt die Verwendung der Funktion und mögliche Fehler. In Listing 101 wurde versucht, den Speicher freizugeben und ihn auszulesen.

```
1 fn main() {
2   let ptr = Box::new(42i32); // allocates memory and places the value 42(uint_32) in
      it
3   drop(ptr)
4   println!("value: {}", ptr); // won't compile, ptr was dropped before
5 }
```

Listing 101: Rust Drop Funktion

Dies führt zur Ausgabe des Compilers in Listing 102. Das Programm wird nicht kompiliert und kann nicht ausgeführt werden, bis der Fehler behoben wird.

```
1 error[E0382]: use of moved value: `ptr
2 --> MEM30-C.rs:4:27
3 |      drop(ptr);
4 |           --- value moved here
5 |      println!("value: {}", ptr);
6 |                            ^^^ value used here after move
7 = note: move occurs because `ptr` has type
8 `std::boxed::Box<i32>`, which does not implement the `Copy` trait
9 error: aborting due to previous error
```

Listing 102: Rust Drop Funktion Compiler Output

### 2.33.4 Fazit

Im Vergleich zur Sprache C, die dem Programmierer alle Möglichkeiten und die daraus folgenden Verpflichtungen offen lässt, erkennt Rust mithilfe seiner Konstrukte bereits zur Compilezeit mögliche Probleme, die beim Zugriff auf Speicher entstehen können. Die automatisierte Erkennung ist essentiell hilfreich zur Erstellung sicheren Codes.

[63] https://doc.rust-lang.org/1.30.0/book/second-edition/ch10-03-lifetime-syntax.html

## 2.34 MEM31-C

Sobald auf dynamisch allozierte Speicherbereiche nicht mehr zugegriffen werden muss, sollen diese wieder freigegeben werden [64].

### 2.34.1 Auswertung in C

In der Programmiersprache C können mithilfe der Standardbibliotheksmethoden **malloc** (Memory Allocation) und **calloc** (Cleared Memory Allocation) Speicherbereiche dynamisch auf dem **Heap** alloziert werden. Der dabei zurückgegebene Pointer (abgelegt auf dem Stack) verweist auf die Startadresse des Speicherbereichs, der vom Betriebssystem zugewiesen wird [65].

```
1  #include<stdlib.h>  /* printf, NULL */
2  #include<stdio.h> /* malloc, calloc, free */
3
4  int test_alloc_mem(int size){
5    int *pointer_array;
6    pointer_array = (int *) malloc(size * sizeof(int));
7    if (pointer_array == NULL) {
8      printf("Speicherzuweisungs Fehler");
9      return -1; }
10   return 0;
11 }
```

Listing 103: C unsichere Speicherzuweisung

Zum Schließen des Speicherbereichs muss vor dem Return der Methode die Funktion **free** auf den Speicherpointer gesetzt werden (siehe Listing 128). Falls der Pointer als „static“ gekennzeichnet und somit bis zum Ende des Programms gültig ist, kann auf ein free innerhalb der Methode verzichtet werden. Trotzdem muss der Pointer innerhalb des Programms geschlossen werden, um ein sicheres Deallozieren des Speichers zu gewährleisten.

```
1  #include<stdlib.h>  /* printf, NULL */
2  #include<stdio.h> /* malloc, calloc, free */
3
4  int test_alloc_mem(int size){
5    int *pointer_array;
6    pointer_array = (int *) malloc(size * sizeof(int));
7    if (pointer_array == NULL) {
8      printf("Speicherzuweisungs Fehler");
9      return -1; }
10   free(pointer_array)
11   return 0;
12 }
```

Listing 104: C sichere Speicherzuweisung

### 2.34.2 Risikobewertung

Sollten offene Speicherbereiche nicht geschlossen werden, entstehen Speicherkapazitätsengpässe, die einen „Denial of Service“ auslösen können. Die Kosten zur Fehlerbehebung werden als „Medium“ eingestuft, da offene mallocs in komplexeren Applikationen leicht zu übersehen sind, jedoch durch eine mit den entsprechenden Tools automatisch durchgeführte Codeanalyse weitestgehend erkannt werden können. Auch der Schweregrad dieses Fehlers ist „Medium“, da die entstehenden Speicherleaks das System zwar dazu zwingen können, Daten auf die Festplatte auszulagern und

[64] https://wiki.sei.cmu.edu/confluence/display/c/MEM31-C.+Free+dynamically+allocated+memory+when+no+longer+needed
[65] http://www.cplusplus.com/reference/cstdlib/malloc/

somit zu verlangsamen, ein einzelner offener Speicherbereich aber nicht automatisch zum Ausfall des Systems führt.

### 2.34.3 Auswertung in Rust

Um in Rust Daten auf dem Heap zu allozieren, sollte der **Smart Pointer** „Box<T>“ verwendet werden. Dieser alloziert automatisch die richtige Größte für einen generischen Wert T auf dem Heap und gibt einen Pointer zurück. Das Konzept der Smart Pointer ermöglicht hierbei nicht nur das Referenzieren auf den Speicherbereich, sondern stellt zusätzliche Kapazitäten und Metadaten bereit. Der Smart Pointer Box<T> unterliegt auch dem Konzept des Ownerships, wodurch die Daten des Stack und Heaps beim Verlassen der Methode automatisch dealloziert werden [66]. Zur Veranschaulichung wurde das C Beispiel in Rust überführt (siehe Listing 105). Im Gegensatz zu malloc, bei dem ein Pointer auch NULL als Rückgabewert enthalten kann, bricht das Allozieren von Box<T> in der aktuellen Version im „default_out_of_memory_handler“ bei einem „Heap Out of Memory Error“ ohne das Aufkommen einer dazugehörigen Fehlermeldung in Form eines „Panics“ ab. [67]. Um dasselbe Verhalten auch in Rust nachzustellen, existiert der Trait „std:alloc:Alloc“, welcher jedoch als „unsafe“ sowie „nightly-only-experimental API“ gekennzeichnet ist und somit nicht standardmäßig verwendet werden sollte [68].

```
1 fn test_alloc_mem(val: i32) {
2   let pointer = Box::new(val)
3 }
```

Listing 105: Rust Speicherzuweisung

### 2.34.4 Fazit

Innerhalb der vorgegebenen sicheren Methoden existiert das C Problem MEM31-C in Rust nicht. So kümmert sich der Compiler bei Smart Pointern um die Freigabe der Ressourcen. Wird allerdings der Trait Alloc benutzt, greifen diese Mechanismen nicht mehr, wodurch die Schwachstelle wieder zum Tragen kommen kann. Durch den Status „experimental unsafe“ sollten die dazugehörigen Methoden für sicherheitskritische Applikationen nur in speziellen Fällen zur Anwendung kommen.

[66] https://doc.rust-lang.org/book/ch15-00-smart-pointers.html?highlight=smart#smart-pointers
[67] https://github.com/rust-lang/rust/blob/1.18.0/src/liballoc/oom.rs#L15-L19
[68] https://doc.rust-lang.org/std/alloc/trait.Alloc.html

## 2.35 MEM33-C

**Allocate and copy structures containing a flexible array member dynamically**
Bei der MEM33-C Regel wird die Verwendung von Strukturen betrachtet, die flexible Array-Attribute besitzen.

### 2.35.1 Was sind flexible Array-Attribute?

Laut dem C Standard 6.7.2.1[69] handelt es sich um einen Wert mit einem unvollständigen Array Typen. Dieser Wert muss das letzte Element einer Struktur sein und die Struktur muss mindestens einen weiteren Wert beinhalten. Auch wenn die flexiblen Array-Attribute bei der Definition der Struktur angegeben werden, werden diese ignoriert, wenn die Größe der Struktur berechnet wird[70].

### 2.35.2 Problem

Durch das Ignorieren der flexiblen Array-Attribute wird die Größe der Struktur falsch berechnet. Wenn dann daraufhin auf die flexiblen Array-Attribute zugegriffen wird, liefert der Aufruf einen undefinierten Wert zurück. Dies kann dann dazu führen, dass das Programm in einen undefinierten Zustand wechselt. Des weiteren können Strukturen mit flexiblen Array-Attributen nicht über eine Zuweisung kopiert werden, da auch hier die flexiblen Array-Attribute nicht mit kopiert werden. Das selbe Problem tritt auch bei der Übergabe von Strukturen, mit einem flexiblen Array-Attribut, als ein Funktionsargument auf.

### 2.35.3 Implementierung in C

```c
struct flex_array_struct {
  int num;
  int data[];
};
int flex_array_wrong() {
  struct flex_array_struct test_struct;
  test_struct.num = 2;
  printf("data: %i", test_struct.data[1]); // Undefinierter Wert vom Stack
  test_struct.data[1] = 123;               // Speicher wird korrumpiert
  printf("data: %i", test_struct.data[1]);
}
```

Listing 106: C-Code mit flexiblen Array-Attribute

In dem Listing 106 wird eine Struktur mit einem flexiblen Array-Attribut und eine Funktion, die auf der Struktur arbeitet, dargestellt. Dabei wird auf das erste Problem eingegangen, welches in 2.35.2 beschrieben ist. Falls ein Wert des data-Arrays ausgelesen wird, entsteht noch kein Fehler. Der ausgelesene Wert beinhaltet jedoch einen Wert, der im Stack nach der Struktur gelegen ist. Bei einer Zuweisung eines Wertes wird jedoch auf einen Speicherbereich geschrieben, der nicht für die Struktur vorgesehen ist, wodurch ein Segmentation Fault Fehler auftritt und das Programm abstürzt.

```c
struct flex_array_struct *test_struct;
test_struct = (struct flex_array_struct *)malloc(
sizeof(struct flex_array_struct) + sizeof(int) * array_size);
```

Listing 107: C-Code mit der korrekten Verwendung von flexiblen Array-Attributen

[69] http://c0x.coding-guidelines.com/6.7.2.1.html
[70] https://wiki.sei.cmu.edu/confluence/display/c/MEM33-C.++Allocate+and+copy+structures+containing+a+flexible+array+member+dynamically

Es ist jedoch auch möglich das Problem, wie es im Listing 2.35.2 dargestellt ist, in C zu beheben. Dafür muss bei der Initialisierung der Struktur, wie in Listing 107 zu sehen, Speicher reserviert werden, wobei dort schon die Größe des Arrays festgelegt wird.

### 2.35.4 Implementierung in Rust

Um den benötigten Speicher zu reservieren, muss Rust in der Lage sein die Größe, des zu reservierenden Speichers, zu ermitteln. Damit das Möglich ist, muss die Größe eines Arrays angegeben werden. Da es jedoch auch Fälle gibt, in der die benötigte Größe zur Kompilierzeit nicht bekannt ist, besitzt Rust sogenannte Dynamically Sized Types (DST). Dabei ist z.B. ein Vector in Rust ein DST.[71] Mit dem DST kann eine Struktur erstellt werden, wie in Listing 108 zu sehen, die ein flexibles Array-Attribut besitzt. Jedoch muss in Rust, im Gegensatz zu C, die Größe des Arrays bei dem Initialisieren der Struktur angegeben werden. Dafür wird ein generischer Typ verwendet.

```
struct FlexibleArrayStructNoSize<T: ?Sized> {
  num: i64,
  array: T,
}
fn flex_struct_without_size() {
  let test_struct = FlexibleArrayStruct {num: 1, array: [0; 4]};
  let dynamic_test_struct: &FlexibleArrayStruct<[i64; 4]> = &test_struct;
  println!("num: {}", test_struct.num);           // 1
  println!("array: {:?}", &test_struct.array);    // [0, 0, 0, 0]
}
```

Listing 108: Flexible Array-Attribute mithilfe von DST's in Rust implementiert

### 2.35.5 Fazit

Strukturen mit flexiblen Array-Attribute können sehr nützlich sein, falls die Größe bei dem Kompilieren noch nicht bekannt ist, auch wenn durch falsche Handhabung, Fehler im Code entstehen können. In Rust können diese Fehler umgangen werden, jedoch mit dem Nachteil, dass die Größe des Arrays bei der Initialisierung der Struktur bekannt sein muss.

[71] https://doc.rust-lang.org/nomicon/exotic-sizes.html#dynamically-sized-types-dsts

## 2.36 MEM34-C

Die Secure Coding Regel MEM34-C besagt, dass nur Speicher zur Laufzeit freigegeben werden darf, welcher zuvor dynamisch allokiert wurde. Hierzu soll nun das problematische Verhalten in der Programmiersprache C, sowie die Anwendbarkeit der Regel auf die Programmiersprache Rust analysiert werden.

### 2.36.1 Analyse Programmiersprache C

Die Programmiersprache C bietet zum Einen die Möglichkeit, Variablen und damit Speicher zur Kompilierzeit zu definieren und damit die Speicherverwaltung Kompiler und Laufzeitumgebung zu überlassen, zum Anderen die Möglichkeit der dynamischen Speicherverwaltung zur Laufzeit.

Dynamische Speicherverwaltung in der Programmiersprache C basiert auf drei Funktionen **malloc()**, **free()** und **realloc()**. Ein beliebig großer Speicherbereich kann über die Funktion **malloc()** allokiert werden. Wird der Speicherbereich nicht mehr benötigt, so kann er mit der Funktion **free()** freigegeben werden. Zur Identifizierung des Speicherbereichs wird ein einfacher Zeiger verwendet. Das Problem entsteht nun, da in der Programmiersprache C mit einem Zeiger auf eine prinzipiell beliebige Speicheradresse gezeigt werden kann. Im folgenden Listing 109 werden zwei Fälle anschaulich dargestellt

- Fall 1: Dynamischer Speicher wird korrekt allokiert und über den Zeiger zeiger_dynamisch referenziert.
- Fall 2: Der Zeiger zeiger_statisch referenziert die statische Variable globale_variable.

Der Aufruf der Funktion **free()** mit dem Parameter zeiger_dynamisch gibt korrekt den vorher allokierten Speicher frei, wobei hingegen der selbe Aufruf mit dem Parameter zeiger_statisch einen Programmabsturz verursacht. Letzeres ruft das im C-Standard beschriebene undefinierte Verhalten hervor, da die globale Variable statisch zur Kompilierzeit reserviert wurde und nicht dynamisch zur Laufzeit. Der gleiche Effekt entsteht, wenn ein Zeiger auf eine lokale Variable der Funktion **free()** übergeben wird.

```c
#include <stdlib.h>
int globale_variable;
int main()
{
    int* zeiger_dynamisch = malloc(sizeof(int)); // Dynamische Allokation
    int* zeiger_statisch = &globale_variable;  // Zeiger auf globale_variable
    // ...
    free(zeiger_dynamisch); // Ok.
    free(zeiger_statisch); // Fehler.
    return 0;
}
```

Listing 109: C Beispiel für richtigen und falschen Aufruf der Funktion free()

### 2.36.2 Analyse Programmiersprache Rust

Ein zentrales Thema der Programmiersprache Rust ist die sichere Verwaltung von Speicher, was in der Programmiersprache C eine der fehleranfälligsten Aufgaben ist, die der Entwickler lösen muss. Die Programmiersprache Rust verzichtet deshalb komplett auf eine explizit durch den Entwickler gesteuerte Speicherverwaltung durch Funktionsaufrufe. Rust bedient sich der Konzepte der Ownership, Borrowing und Lifetimes, sowie der Smart Pointer zur Speicherverwaltung.

Zusammenfassen lässt sich das Konzept der Speicherverwaltung in Rust folgendermaßen: Eine Variable hat grundsätzlich einen Gültigkeitsbereich (Scope). Wird dieser Gültigkeitsbereich verlassen,

wird diese Variable freigegeben bzw invalidiert. Auf Variablen kann per Referenz verwiesen werden. Rust stellt über das Konzept der Lifetimes sicher, dass keine Referenz die Lebenszeit ihre referenzierte Variable überdauert, also das eine Referenz auf eine nicht mehr gültige Variable existiert. Dynamische Speicherallokation erfolgt in Rust über die so genannten Smart Pointer. Diese verwenden das Konzept des Reference Countings, um die Anzahl der existierenden Referenzen auf die dynamisch allokierte Speicherbereich zu verfolgen. Erreicht diese Anzahl den Wert 0, wird der zum Smart Pointer gehörende Speicherbereich freigegeben. Listing 110 zeigt eine dynamische Allokation eines Integers, welcher ausgegeben und am Ende des Gültigkeitsbereichs (Scope) freigegeben wird.

```
1 fn main() {
2     let dynamischer_speicher = Box::new(1);
3     println!("Wert: {}", dynamischer_speicher);
4 } // Speicher wird freigegeben, wenn Variable Scope verlaesst
```

Listing 110: Rust Beispiel zur Verwendung von dynamischen Speicher mittels Smart Pointer

### 2.36.3 Fazit

Der Mechanismus der Speicherfreigabe mittels Gültigkeitsbereich erlaubt es nicht, lokale oder globale Variablen manuell per Funktionsaufruf freizugeben. Bei lokalen Variablen endet der Gültigkeitsbereich stets am Ende des Funktionsblocks. Globalen Variablen mit der expliziten Lebenszeit static bleiben stets bis zum Ende des Programmes bestehen und haben somit einen globalen Gültigkeitsbereich (Scope). Das in der Secure Coding Regel MEM34-C beschriebene Problem besteht somit in der Programmiersprache Rust nicht.

Der Vollständigkeit halber sei erwähnt, dass Rust mittels so genannten unsicheren Code (unsafe code) die Verwendung von Zeiger und Funktionen der Systembibliotheken erlaubt. Hierbei bestehen natürlich die gleichen Probleme wie bei der Programmiersprache C: es kann ein Zeiger auf eine statische Variable erzeugt und der **free()**-Funktion der Systembibliothek libc übergeben werden, was ebenfalls zu einem Programmabsturz führt.

## 2.37 FIO30-C

Folgend wird die Secure Coding Regel FIO30-C erörtert und dargelegt, mit welcher Problematik sie sich befasst. Weiter werden Lösungsvorschläge gegeben und ein Vergleich gezogen, wie die Sprachen C und Rust diese implementieren.

### 2.37.1 Problembeschreibung und Risikobewertung

Regel FIO30-C besagt, dass I/O-Funktionen keinen Format-String entgegennehmen dürfen, der Elemente aus unsicheren Quellen enthält (z.B. Benutzereingaben). Werden dem Format-String Parameter übergeben, die eigentlich zur Formatierung gedacht sind (`%x`, `%p` etc.)[72], werden auch überschüssige Parameter interpretiert und mit Werten aus dem Stack aufgefüllt. Wird die vorliegende Secure Coding Regel nicht eingehalten, ist es einem Angreifer möglich, den Inhalt eines Format-Strings zumindest in Teilen zu modifizieren. Dadurch ist er in der Lage, einen Prozess zum Absturz zu bringen sowie Inhalte vom Stack anzeigen zu lassen. Durch Verwendung des Format-Parameters `%n` ist es ebenso möglich, einen Integer-Wert in eine bestimmte Adresse zu schreiben. Weiter kann eigener Schadcode mit den Berechtigungen des angegriffenen Prozesses ausgeführt werden. Aus den eben genannten Gründen ist das Versäumnis, diese Regel zu berücksichtigen, und das daraus resultierende Schadenspotential als sehr schwerwiegend einzuordnen.

### 2.37.2 Fallbeispiel in C

Ein Beispiel für eine solche anfällige Implementierung zeigt sich in Listing 111. Bei der Ausgabe mit `printf()` wird die Benutzereingabe `%x` als Teil des Format-Strings und somit als Aufforderung, einen Wert als hexadezimale Zahl zu formatieren, interpretiert. Was nicht geprüft wird, ist, ob die Methode außerdem einen zu formatierenden Wert erhalten hat. Ist dies nicht der Fall, wird somit ein auf dem Stack befindlicher Wert ausgegeben. Für die folgenden Beispiele sei die Benutzereingabe `AAAA%x.%x.%x.[...].%x.%x.`

```
1 void greet_user_vulnerable(char *input) {
2     const char *msg = ", moin!\n";
3     strcat(input, msg); //Verkettet input und msg
4     printf(input); //Interpretiert den Format-String
5 }
6 Output: AAAAXXXXXXXX.XXXXXXXX.XXXXXXXX.[...].414141.78383025.2e78252e, moin!
```

Listing 111: C - Unsichere Implementierung

Die Zeichenfolge XXXXXXXX ist eine beliebige hexadezimale Zahl. Ein Teil der übergebenen Zeichenkette `AAAA%x.%x` findet sich als `414141.78383025.2e78252e` wieder. Diese Werte konnten also im Stack identifiziert werden. In Listing 112 wird stattdessen `fputs()` verwendet. Hier wird der Format-String nicht interpretiert. Als Konsequenz werden überschüssige Format-Parameter nicht mit Werten aus dem Stack aufgefüllt.

```
1 void greet_user_valid(char *input) {
2     const char *msg = ", moin!\n";
3     strcat(input, msg);
4     fputs(input, stdout); //Interpretiert den Format-String nicht
5 }
6 Output: AAAA%x.%x.%x.[...].%x.%x, moin!
```

Listing 112: C - Sichere Implementierung mittels `fputs()`

[72] https://www.owasp.org/index.php/Format_string_attack/

Ebenso ist es möglich, den Format-String und die Benutzereingabe getrennt zu behandeln, wie in Listing 113 zu sehen ist.

```
1 void main() {
2     printf("Enter your name\n");
3     fprintf(stdout, "%s, moin!", gets(input));
4 }
5 Output: AAAA%x.%x.%x.[...].%x.%x,moin!
```

Listing 113: C - Sichere Implementierung mittels `fprintf()`

Das Problem der Format-String-Vulnerabilities wurde in C durch mehrere Compiler bereits adressiert. So erkennt beispielsweise `gcc` seit Version 4.3.5 mittels `-Wformat-security`, ob eine Gefährdung vorliegt.[73]

### 2.37.3 Fallbeispiel in Rust

In Rust ist der Format-String-Parameter einer jeden formatierenden Funktion ausschließlich als String-Literal zu übergeben.[74] Variablen werden an dieser Stelle durch den Compiler nicht zugelassen. Auch wenn der gesamte String in einer Variable enthalten ist, muss für den Format-String ein String-Literal eingesetzt werden. Dies zeigt Listing 114.

```
1 fn greet_user(input: &str){
2     println!("{}, moin!", input); //Format-String als String-Literal
3 }
4 Output: AAAA%x.%x.%x.[...].%x.%x,moin!
```

Listing 114: Rust - Sichere Implementierung

### 2.37.4 Fazit

Auch wenn C-Compiler mittlerweile auf eine Verletzung der vorliegenden Regel prüfen und darauf hinweisen, ist es im Hinblick auf den Schweregrad der möglichen Folgen einer unsicheren Implementierung wichtig, sich der Gefahr von String-Vulnerabilities bewusst zu sein. Eine solche Anfälligkeit in C-Code zu vermeiden, ist grundsätzlich einfach. Gerade deshalb scheint die von Rust gebotene Lösung, an der kritischen Stelle ausschließlich String-Literale zuzulassen, sehr simpel, aber dennoch effektiv.

[73] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
[74] https://doc.rust-lang.org/std/fmt/

## 2.38 FIO39-C

Beim Öffnen von Dateien mit `fopen` wird durch den Parameter `mode` bestimmt in welchem Modus der Zugriff geschieht. Die mit einem „+“ gekennzeichneten Modi erlauben dabei sowohl das Lesen als auch das Schreiben auf dem Datenstrom. Für diese schreibt der Standard vor, dass auf eine Lese-Operation, die nicht auf `EOF` trifft, eine Positionierungsfunktion (`fseek`, `fsetpos`, `rewind`) folgen muss, bevor auf dem Datenstrom wieder geschrieben werden kann. Umgekehrt muss auf eine Schreib-Operation entweder der Puffer des Datenstroms geleert werden (`fflush`) oder eine Positionierungsfunktion aufgerufen werden, bevor wieder gelesen werden darf. In der Rationale des C-Standards wird diese Regel damit begründet, dass eben `fsetpos`, `fseek`, `rewind` und `fflush` sicherstellen, dass der IO Puffer geleert wurde. [75] Demnach sollten ungepufferte Datenströme also nicht von der Regel betroffen sein, dokumentiert ist dies aber weder im Standard noch der Rationale. Ansonsten ist das Verhalten bei Missachten der Regel im Standard **nicht definiert**. [76]

### 2.38.1 Codebeispiel und konforme Lösung (C)

```
1 void fio39_c_falsch(FILE *f, int c) {
2     if (fputc(c, f) != c) { /* Behandle Fehler */ }
3     if (fgetc(f) == EOF)  { /* Behandle Fehler */ }
4     if (fputc(c, f) != c) { /* Behandle Fehler */ }
5 }
```

Listing 115: Aufeinanderfolgendes Lesen/Schreiben ohne Flush/Positionsänderung

```
1 void fio39_c_richtig(FILE *f, char c) {
2     if (fputc(c, f) != c)                { /* Behandle Fehler */ }
3     if (fseek(f, 0L, SEEK_SET) != 0)     { /* Behandle Fehler */ }
4     if (fgetc(f) == EOF)                 { /* Behandle Fehler */ }
5     if (!feof(f))                          /* fseek nicht nötig nach EOF */
6         if (fseek(f, 0L, SEEK_END) != 0) { /* Behandle Fehler */ }
7     if (fputc(c, f) != c)                { /* Behandle Fehler */ }
8 }
```

Listing 116: Aufeinanderfolgendes Lesen/Schreiben mit Flush/Positionsänderung

### 2.38.2 Risikobewertung

Bisher gibt es keine bekannten Sicherheitslücken durch das Missachten dieser Regel. Da aber Dateioperationen besonders auf UNIX-ähnlichen Systemen von großer Bedeutung sind („*Everything is a file*“), resultiert das Nichtbeachten wahrscheinlich in einer ausnützbaren Schwachstelle. Deshalb erhält die Regel im *SEI CERT C Coding Standard* auch die Bewertung *L2 P6*, gilt also als mittelschwer. Erkennen lässt sich eine Missachtung mit statischen Code-Analyse-Tools wie *Parasoft C/C++test* und der *LDRA tool suite*. [77]

[75] http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf
[76] ISO/IEC. Programming Languages—C, 3rd ed (ISO/IEC 9899:2011). Geneva, Switzerland: ISO, 2011.
[77] https://wiki.sei.cmu.edu/confluence/x/L9YxBQ

### 2.38.3 FIO39-C in Rust

Auch in Rust können Dateien mit Lese- und Schreibrechten geöffnet werden. Im Gegensatz zu C sind die File IO-Operationen in Rust aber standardmäßig nicht gepuffert. Zwischen Lese- und Schreiboperationen sollte somit kein Aufruf von flush oder seek notwendig sein. Es ist aber auch möglich gepufferte Lese-/Schreiboperationen mit BufReader/BufWriter durchzuführen. Zu beachten ist hier, dass der interne Puffer eines BufWriters nur geleert wird, wenn der Writer durch verlassen des Geltungsbereichs oder drop zerstört wird, wenn der interne Puffer voll ist oder wenn flush oder seek aufgerufen wird. Der Positionsmarker für die Datei wird ebenfalls nur in diesen Fällen gesetzt. Gleiches gilt für BufReader. [78] Durch abwechselndes Lesen und Schreiben kann man deshalb leicht den Überblick verlieren an welcher Stelle gerade gelesen bzw. geschrieben wird. Aus diesem Grund wurde auch die Struktur BufStream, die gemischtes Lesen und Schreiben erlaubt, vorerst als *unstable* markiert. [79] Verhindern lässt sich diese Fehlerquelle wie auch in C durch seek bzw. flush.

### 2.38.4 Codebeispiel und konforme Lösung (Rust)

```
use std::fs::File;
use std::io::{BufWriter, BufReader};
fn schreiben_lesen(f: &File) {
    let mut writer = BufWriter::new(f);
    let mut reader = BufReader::new(f);
    let mut buffer = &mut String::new();
    // writer.seek(SeekFrom::Start(0)).unwrap(); // Kommentarzeichen entfernen
                                                 // um Fehler zu vermeiden
    writer.write_all("Hello, World!".as_bytes()).unwrap();
    reader.read_line(&mut buffer).unwrap(); // "Hello, World!" wurde noch
                                            // nicht in die Datei geschrieben
}
```

Listing 117: Gepuffertes Lesen/Schreiben

### 2.38.5 Fazit

Die Regel wurde in den C-Standard aufgenommen, da sonst nicht garantiert ist, dass der Puffer zwischen den Lese- und Schreiboperationen geleert wurde. In Rust dagegen ist genau definiert, wann die Puffer geleert werden. Das gemischte Lesen und Schreiben muss aber durch zwei eigenständige Objekte geschehen und deren Interaktion ist nicht intuitiv verständlich. Wenn das definierte Verhalten nicht explizit erwünscht ist, sollten auch hier die Puffer dazwischen geleert werden, um Fehler zu vermeiden.

Man kann in beiden Sprachen auch auf gepufferte IO-Operationen verzichten, zumindest für C ist aber nich klar, ob das Problem dadurch auch gelöst wird. Zudem würde das Programm dadurch bei vielen kleinen Lese- und Schreibvorgängen deutlich langsamer. In vielen Fällen ist dies also keine akzeptable Lösung.

[78] https://doc.rust-lang.org/src/std/io/buffered.rs.html
[79] https://github.com/rust-lang/rust/issues/17136

## 2.39 FIO40-C

FIO40-C bezieht sich auf Problemstellungen im C-Standard die Methoden *fgets()* und *fgetws()* betreffend. Im Folgenden wird diese Problemstellung in der Programmiersprache C genauer erläutert und untersucht inwieweit dieses Problem in der Programmiersprache Rust bekannt ist.

### 2.39.1 C

Die Methoden *fgets()* und *fgetws()* dienen dazu Zeichen über einen Stream einzulesen und diese abzuspeichern. Die Methoden bekommen als Übergabewerte den Speicher für die eingelesenen Zeichen, die Anzahl an Zeichen die eingelesen werden sollen und einen Pointer zum Ursprung der Zeichen. Tritt beim Lesen der Datei ein Fehler auf oder die Datei erreicht ihr Ende ohne das es möglich war ein Zeichen zu lesen wird von den Methoden *fgets()* und *fgetsws()* ein Null-Pointer zurückgeliefert. Problematisch in diesem Fall ist, dass der Inhalt des Speichers, in welchen die eingelesenen Zeichen abgespeichert werden sollten unbekannt ist [80]. Wird ein Null-Pointer von den Methoden zurückgeliefert können keine verlässlichen Aussagen über den Inhalt des Speichers getroffen werden. Wird der Speicher trotz fehlgeschlagener Lese- und Speicheroperation dennoch weiterverwendet, kann es zu unvorhersehbaren Verhalten des Programmes oder gar zum Abbruch führen. Generell wird diese Problematik als gering eingestuft und kann mittels verschiedener Techniken vermieden werden. Das Listing 118 zeigt die Verwendung von *fgets()* die das Abfangen der beschriebenen Problematik nicht behandelt und kann somit als Negativ Beispiel angesehen werden.

```
1 #define MAX 15
2 void violation(FILE *file) {
3   char store[MAX]; // used for multiple operations
4   fgets(store, MAX, file);
5   // In case of an error the contents of the store array are unknown.
6 }
```

Listing 118: Problematische Verwendung von fgets()

Die einfachste Möglichkeit einen Null-Pointer abzufangen und die Stabilität des Speichers zu garantieren ist, den Speicher bei Rücklieferung eines Null-Pointers nicht weiterzuverwenden. Dass heißt konkret den Speicher in diesem Fall nicht weiter anzusprechen. Eine weitere richtige Handhabung wird im Listing 119 beispielhaft aufgezeigt.

```
1 void compliant(FILE *file) {
2   char store[MAX];
3   if(fgets(store, MAX, file) == NULL) {
4     printf("fgets() returned a null pointer\n");
5     printf("reset store to an empty string");
6     *store='\0';
7   }}
```

Listing 119: Abfangen der Problematik mit C

Dabei wird im Falle einer Null-Pointer Rückgabe der Inhalt des Speichers auf einen leeren String zurückgesetzt. Somit kann der Speicher weiterverwendet werden und ein unvorhersehbares Verhalten kann damit ausgeschlossen werden.

### 2.39.2 Rust

Die Programmiersprache Rust bietet verschiedenste Möglichkeiten einen Dateiinhalt einzulesen und zu speichern. Dabei wird jedoch in den meisten Fällen die gesamte Datei oder einzelne Zeilen der

[80] https://wiki.sei.cmu.edu/confluence/display/c/FIO40-C.+Reset+strings+on+fgets%28%29++or+fgetws%28%29+failure

Datei eingelesen. Dadurch können mehrere Methoden aus Rust zum Vergleich herangezogen werden. Eine Methode die dem Verhalten der beiden Methoden *fgets()* und *fgetws()* sehr ähnlich ist, ist die Methode *read_exact* des Paketes std::io::Read [81]. Diese Methode liest die gegebene Anzahl an Bytes ein und speichert sie in einen übergebenen Puffer. Das Listing 120 zeigt eine beispielhafte Verwendung dieser Rust Methode. Auch hier tritt die Problematik auf, dass der Inhalt des Speichers im Falle eines Fehlers undefiniert/unbekannt ist.

```
1 fn main() {
2   let mut file = File::open("foo.txt").expect("unable to open file");
3   let mut store = vec![0;10];
4   file.read_exact(&mut content); //catch erros with .expect("");
5   let store_content = String::from_utf8(content).unwrap();
6   println!("{}", store_content);
7 }
```

Listing 120: Problematische Verwendung der Rust Methode

Auch hier muss der Fehlerfall abgefangen werden. Es gibt jedoch noch weitere Möglichkeiten Dateiinhalte einzulesen. So ist es beispielsweise möglich die Zeichen bis zu einem gewissen Byte einzulesen mit der Methode *read_until* des Paketes std::io::BufRead [82]. Auch in diesem Fall müssten Fehler abgefangen werden.

### 2.39.3 Fazit

Werden die Methoden *fgets()* und *fgetws()* explizit mit der Methode *read_exact* verglichen, sind die folgenden Aspekte zu beachten. Die C-Methoden haben zwar die vorgegebene maximal Zahl an Bytes, welche gelesen werden sollen, brechen jedoch das Lesen bei einem Zeilenumbruch ab. Das Lesen einer leeren Datei löst die Rückgabe eines Null-Pointers aus. Bei der vorgestellten Rust-Methode ist der Abbruch bei einem Zeilenumbruch nicht gegeben zudem werden die Inhalte des Speichers laut Dokumentation als undefiniert angegeben. Tests dieser Methode ergaben, dass die bis zum Ende der Datei gelesenen Bytes in den Speicher übernommen wurden und auch bei einer leeren Datei und mehreren Testdurchläufen kein Inhalt im Speicher enthalten war. Beide Programmiersprachen erfordern eine Handhabung der Problematik.

[81] https://doc.rust-lang.org/std/io/trait.Read.html/#method.read_exact
[82] https://doc.rust-lang.org/std/io/trait.BufRead.html#method.read_until

## 2.40 FIO42-C

**Regel**
Dateien sind zu schließen, sobald diese nicht mehr benötigt werden.

In der Programmiersprache C ist darauf zu achten, dass nach dem Öffnen einer Datei diese wieder zu schließen ist, sobald diese nicht mehr verwendet wird beziehungsweise wenn das ausgeführte Programm beendet wird, je nachdem welches Ereignis zuerst eintritt. Abhängig von den verwendeten Bibliotheken ist folgendes einzuhalten:
Bei der Programmierung mit den Standardbibliothek muss jede Datei welche mit `fopen()` oder `freopen()` geöffnet wurde nach Ende des Gebrauchs zwingend mit `fclose()` geschlossen werden, andernfalls ist die Coding-guideline verletzt.
Zu beachten ist hierbei, dass die Standardfunktion `exit()` zwar alle offenen Filedeskriptoren vom Typ `FILE*` schließt, hierbei jedoch nicht auf Fehler geprüft werden kann, was ebenso nicht als regelkonform gilt. Das Schließen einer Datei kann fehlschlagen, wenn beim abschließenden schreiben der Puffer ("Flushing") ein Fehler, beispielsweise durch unzureichenden Speicherplatz auf dem Datenträger, auftritt. Analog zur Standardbibliothek gilt bei der Verwendung der POSIX-Funktion `open()` der Aufruf von `close()`.
Wird mit der Windows API programmiert, müssen Dateihandles die mit `CreateFileA()` respektive `CreateFileW()` geöffnet wurden mit `CloseHandle()` geschlossen werden. Wird die Regel nicht eingehalten, kann dies dazu führen, dass das System oder der ausführende Prozess die maximale Anzahl an Dateidiskriptoren erreicht und keine weiteren Dateien geöffnet werden können. Wahrscheinlicher ist jedoch, dass Dateien unnötigerweise für andere Prozesse oder Routinen gesperrt bleiben.

```
1  int main()
2  {
3    FILE* f = fopen("file.txt", "r");
4    if(f)
5    {
6      if (somefileoperations(f)) exit(EXIT_SUCCESS); // non-compliant exit
7      else return EXIT_FAILURE; // non-compliant return
8    }
9    return 0xbadc0de;
10 }
```

Listing 121: Nicht konformes Beenden oder Zurückkehren aus main()

```
1  int main()
2  {
3    FILE* f = fopen("file.txt", "r");
4    if(f)
5    {
6      if(foo(f))
7      {
8        if(somefileoperations(f) != EOF) exit(EXIT_SUCCESS); // compliant exit
9        else exit(EXIT_FAILURE);
10     }
11     else
12     {
13       if(fclose(f) = EOF) return EXIT_FAILURE; //compliant return
14       else return EXIT_SUCCESS;
15     }
16   }
17   return EXIT_SUCCESS;
18 }
```

Listing 122: Konformes Beenden oder Zurückkehren aus main()

### 2.40.1 Lösungsansatz Rust

In der Programmiersprache Rust ist die Struktur `File` aus dem Modul `std::fs` so implementiert, dass Dateien automatisch geschlossen werden, sobald die Instanz den Gültigkeitsbereich verlässt. Um das Schließen der Datei zu forcieren, kann die Funktion `std::mem::drop` aufgerufen werden, es ist jedoch zu beachten, dass `drop` als leere Funktion implementiert ist, welche Ownership von einer Variable ergreift und sonst nichts weiter macht. Die Datei wird also geschlossen, da das Scope von `drop` wieder verlassen wird. Da die genannte Funktion kein `Result` zurückliefert kann nicht sichergestellt werden, dass es keinen Fehler beim Flushing gab. Hierfür gibt es die Funktionen `sync_all` und `sync_data` welche versuchen alle Puffer zu schreiben und ein `Result` retournieren mit dem überprüft werden kann, ob die Operation erfolgreich war.

```
fn main() {
    let mut f = match File::open("file.txt") {
        Err(e) => panic!("Couldn't open file"),
        Ok(f) => f,
    };
    /* some file operations */
    match f.sync_all() { // ensure flushing
        Err(e)=> println!("Syncing Error"),
        Ok(o) => println!("Syncing Ok"),
    }
    drop(f); // not necessary
} // automatically dropped
```

Listing 123: Regelkonformes Schließen in Rust

### 2.40.2 Fazit

Abschließend lässt sich feststellen, dass der Ansatz Dateien automatisch beim verlassen des Gültigkeitsbereiches zu schließen dem Programmierer in Rust eine Fehlerquelle erspart. Jedoch sollte beim Schreibzugriff auf Dateien immer geprüft werden, ob Puffer mit dem Datenträger synchronisiert wurden. Dies lässt sich durch `sync_all` und `sync_data` prüfen.

## 2.41 FIO45-C

Das in FIO45 beschriebene Problem adressiert eine generische Race-Condition, die theoretisch in jeder Programmiersprache auftreten kann. Dabei handelt es sich um die TOCTOU (time-of-check, time-of-use) Konstellation. Diese entsteht, wenn eine Ressource zu einem anderen Zeitpunkt verfügbar ist, als Operationen auf dieser ausgeführt werden. Dies geschieht erfahrungsgemäß im Kontext mehrerer Prozesse, die auf gleiche Dateien oder Speicherinhalte zugreifen. Darüber hinaus kann dieses Verhalten nicht nur zu unbeabsichtigtem Datenverlust führen, sondern auch von Angreifern ausgenutzt werden, um beispielsweise Dateien durch einen Symlink o.Ä. zu ersetzen. Aus diesem Grund sind die Auswirkungen einer solchen Verletzung der Regel als relativ schwerwiegend einzuordnen.

### C

In der Programmiersprache C kann eine Datei mit verschiedenen Optionen geöffnet bzw. geschrieben werden. Wenn eine Datei im Schreibmodus erzeugt wird, überschreibt diese die bereits bestehende Datei. Deshalb muss vorher eine Überprüfung der Zieldatei erfolgen. Dies kann über den Lesemodus realisiert werden. Der Rückgabewert des Lesemodus ist der Stream der Datei. Wenn dieser nicht leer ist, existiert die Datei mit dem gegebenen Namen bereits. Das in Listing 124 dargestellte Beispiel, erfüllt die beschriebenen Anforderungen.

```
1 FILE *datei = fopen(dateiName, "r"); // Oeffne Datei im Lese-Modus
2 if (datei) {
3   // Die Datei exisitiert
4 } else { // Schreibe nur auf Datei, wenn diese noch nicht existiert
5   fclose(datei);   // Schliesse den Stream der Datei vor erneutem Oeffnen
6   datei = fopen(dateiName, "w");    // Oeffne Datei im Schreib-Modus
7   fprintf(datei, "Schreibe in C");    // Schreib-Operation
8   fclose(datei);
9 }
```

Listing 124: C: Lese- und Schreibmodus einer Textdatei

Um in dieser Funktion eine TOCTOU Race-Condition zu erzeugen, müssen zwei Prozesse auf dieselbe Datei zugreifen. Dies scheint auf den ersten Blick durch die if-Anweisung ausgeschlossen, da die Schreiboperation nicht ausgeführt wird, falls die Datei bereits existiert. In diesem Szenario kann es trotzdem zu unerwartetem Verhalten kommen. Wenn Prozess 1 die if-Anweisung durchgeführt hat und anschließend vom Betriebssystem pausiert wird, kann potentiell ein zweiter Prozess gestartet werden. Dieser führt die Funktion ohne Unterbrechungen aus. Anschließend wird Prozess 1 fortgeführt und überschreibt die von Prozess 2 erzeugte Datei. Ein Angreifer würde an dieser Stelle versuchen, gezielt vor Prozess 2 zu starten und anschließend seinen schadhaften Programmcode auf die Datei zu schreiben. Dies könnte beispielsweise über eine Funktion bewerkstelligt werden, die direkt vor der Schreiboperation wartet.

```c
FILE *datei = fopen(dateiName, "wx"); // Oeffne Datei im geschuetzen Modus
if (datei == NULL) { /*Fehler beim Oeffnen*/  }
// Schreibe auf die Datei, falls diese bereits exisitert, wird das Programm mit
    folgender Ausgabe beendet:
//  './a.out' terminated by signal SIGSEGV (Address boundary error)
```

Listing 125: C: Korrekter Schreibmodus

Wie in Listing 125 exemplarisch dargestellt, kann dieses Problem leicht behoben werden. Im C11-Standard wurde ein neuer Schreibmodus für Dateien eingeführt. In diesem Modus wird das Programm beendet, falls die Datei bereits existiert.

### Rust

In Rust gibt es ebenfalls eine Vielzahl an Möglichkeiten, um mit Dateien bzw. deren Streams umzugehen. Dabei kann man natürlich genauso wie in Listing 124 verfahren. Das bedeutet, dass die TOCTOU Race-Condition ebenfalls durch nachlässige Entwicklung entstehen kann. Allerdings gibt einem Rust über die Stream-Syntax eine elegante Möglichkeit, die Operation genauer zu spezifizieren und die gewünschten Parameter anzupassen.

```rust
let mut datei = try!(OpenOptions::new().write(false).open(datei_name));
match datei.write_all(b"Ein einfacher Text") {
  Ok(_) => (),
  Err(e) => return Err(e),
}
Ok(())
```

Listing 126: Rust Dateistream

Das in Listing 126 gezeigte Beispiel legt über die *OpenOptions* fest, wie die Datei behandelt werden soll. In dem write-Funktionsaufruf wird angegeben, ob eine bestehende Datei überschrieben werden soll. In diesem Fall wird *false* übermittelt, weshalb die Operation an dieser Stelle abbricht. In der Rust Dokumentation [83] finden sich alle möglichen Optionen, welche verwendet werden können um bestimmte Abläufe, beispielsweise das Anfügen von Text am Ende der Datei, zu ermöglichen. Schlussendlich bietet Rust noch die Möglichkeit, dabei entstehende Fehler abzufangen und diese genauer zu identifizieren.

### Fazit

In beiden Programmiersprachen kann das hier untersuchte Problem vermieden werden. Der Ansatz von Rust geht dabei etwas direkter vor. Der Entwickler wird beim Aufrufen der write-Funktion im Stream direkt mit der Entscheidung des auszuführenden Schreib-Modus konfrontiert. Grundsätzlich sollte aber im Kontext mehrere Prozesse genau analysiert werden, welche Seiteneffekte und Angriffe auftreten können. Dies lässt sich in keiner Programmiersprache vermeiden.

[83] https://doc.rust-lang.org/std/fs/struct.OpenOptions.html

## 2.42 FIO46-C

In den folgenden Unterkapiteln wird die Regel FIO46-C und deren Problematik beschrieben. Des Weiteren wird ein Vergleich mit der Programmiersprache Rust durchgeführt.

### 2.42.1 Problembeschreibung

Bei FIO46-C geht es um den Zugriff auf programmatisch bereits geschlossene Dateien in der Programmiersprache C. Hierbei ist das Wort geschlossen so zu verstehen, dass der Entwickler nach einem bestimmten Stand im Code, z.B. nach einer Schreiboperation, den Zugriff auf die verwendete Datei kappt, um den Zugang, z.B. für andere Programme, freizugeben. In C besteht trotz der Trennung einer solchen Verbindung die Möglichkeit, dass die genutzte Zeigervariable der Datei weiterhin verwendet wird.

### 2.42.2 Risikobewertung in C

Dieses Verhalten hat das Potential zur Sicherheitslücke und kann womöglich für die Erstellung von Schadsoftware missbraucht werden. Die in C definierten Standard Streams für Eingabe (stdin) und Ausgabe (stdout) können hierbei eine Rolle spielen[84]. Das Problem führt zu einem undefinierten Verhalten. Dieses tritt auf, wenn eine bestimmte Operation im bereitgestellten Basiscode nicht behandelt wird[85]. Das Problem wird mit Priorität P4 und mit Level L3 eingestuft. Im Vergleich zu anderen Regeln, wird es demnach mit geringem Schweregrad, unwahrscheinlich für Ausnutzungen und teurer in der Lösung beschrieben[86].

### 2.42.3 Problembeispiel in C und konforme Lösung

Anhand des folgenden Beispiels kann das Problem nachvollzogen werden.

```
1 #include <stdio.h>
2 int main() {
3     FILE *fp;
4     fp = fopen("\\Pfad\\zur\\Datei\\text.txt", "w+");
5     fprintf(fp, "Dies steht in der Datei.");
6     fclose(fp);
7     fprintf(fp,"Weder in der Datei noch auf der Konsole");
8 }
```

Listing 127: FIO46-C Beispiel in C

In Programmzeile 7 wird versucht in eine bereits geschlossene Datei zu schreiben. Bei der Ausführung des Programms kommt es zum undefinierten Verhalten. Je nach verwendetem Compiler können unterschiedliche Aktionen stattfinden. Beim GCC-Compiler ist ein Nicht-Wiederfinden des zu schreibenden Textes der Fall. Eine konforme Lösung wäre entweder das Freigeben des Dateizeigers oder eine Unterbindung der Verwendung der Zeigervariable im nachfolgenden Code.

### 2.42.4 Problemvergleich in Rust

Der Ausgangspunkt beider Programmiersprachen scheint vorerst der Selbe zu sein. In Rust gibt es die gleichen Standard Streams für Ein- und Ausgabe wie in C. Der Unterschied von Rust und C

[84] https://docs.microsoft.com/en-us/cpp/c-runtime-library/stdin-stdout-stderr?view=vs-2017
[85] https://www.techopedia.com/definition/20923/undefined-behavior
[86] https://wiki.sei.cmu.edu/confluence/display/perl/Risk+Assessment

im Fall der FIO46-C Regel besteht im Schließen einer Datei. In C wird eine Datei mit dem Funktionsaufruf `fclose(DateiZeiger)` geschlossen. In Rust gibt es hierfür keinen Befehl. Dateien werden automatisch geschlossen sobald der Geltungsbereich der Datei verlassen wird. Dieses Verhalten kann mit der Speicherfreigabe einer Variable beim Wechsel des Geltungsbereichs verglichen werden[87]. Das Schließen der Datei kann auch mit der Funktion `drop(Variable)` durchgeführt werden, da es einer Freigabe der Variable und somit einem Wechsel des Geltungsbereichs gleichkommt[88]. Um einen genauen Vergleich mit der Programmiersprache Rust durchführen zu können, wird der eben beschriebene C-Code in Rust überführt und nachfolgend dargestellt.

```
1  fn main() -> io::Result<()> {
2
3      {
4          let mut file = File::open("\\Pfad\\zur\\Datei\\text.txt")?;
5          file.write_all(b"Dies steht in der Datei.")?;
6          drop(file);
7      }
8
9       file.write_all(b"Produziert einen Error")?;
10     Ok(())
```

Listing 128: Vergleichsbeispiel in Rust

Die Zeilen 3 und 7 werden künstlich für einen Geltungsbereichswechsel hinzugefügt. Die Zeile 6 mit der erwähnten Drop-Funktion ist demnach redundant. Anders wie bei C, endet Zeile 9 in einem Kompilierfehler. Je nach eingesetzter Art des Schließens der Datei erscheint ein ”nicht gefundene Variable”- oder ein ”Variable verschoben”-Fehler.

### 2.42.5 Vergleich und Fazit

Das Problem der für C aufgestellten FIO46-C Regel besteht in Rust nicht. Anstatt eines undefinierten Verhaltens beim Zugriff nach der Freigabe einer Datei, wird bereits während der Kompilierung ein Fehler generiert. In Rust ist es somit nicht möglich, dass nach dem Schließen der Zugang auf eine Datei zur Sicherheitslücke gemacht und für Schadsoftware ausgenutzt wird. Beim Programmieren mit Rust ist in diesem Szenario jedoch weiterhin Vorsicht geboten. Da es in dieser Programmiersprache keine wie die für C bereitgestellte Funktion `fclose(DateiZeiger)` gibt, wird unter Umständen die Zugriffsmöglichkeit nicht bedacht. Verbleibt der Geltungsbereich der Gleiche, ist der Zugriff auf die evtl. nicht mehr benötigte Datei weiterhin möglich. Bei einer sicherheitsgerichteten Software muss dies beachtet werden. Diese Bedrohung kann im Vergleich zur FIO46-C Problematik jedoch vernachlässigt werden, da dies ein geringeres Risiko für die Sicherheit darstellt.

[87] https://doc.rust-lang.org/std/fs/struct.File.html
[88] https://doc.rust-lang.org/std/ops/trait.Drop.html

## 2.43 FIO47-C

Innerhalb der Regel FIO47-C wird die korrekte Benutzung von Format-Strings definiert. Innerhalb dieses Kapitels wird diese Regel sowohl in C als auch in Rust analysiert und die resultierenden Ergebnisse werden verglichen.

### 2.43.1 Problemstellung in C

Für formatierte Ausgaben wird in C hauptsächlich die Funktion `printf()` verwendet. C bietet dabei die Möglichkeit, Platzhalter innerhalb des auszugebenden Strings zu definieren. Jeder dieser Platzhalter wird mit einem %-Zeichen eingeführt. Darauf folgt einer von mehreren fest definierten `specifier`. Die am häufigsten verwendeten sind dabei: `d` für int, `ld` für long, `f` für float/double, `c` für char, `s` für char* (String), `p` für eine Pointer-Adresse und `x` für einen hexadezimalen Wert. Zusätzlich kann ein Platzhalter weitere Argumente aufweisen, welche ihn entsprechend abändern (`flag`, `length`, `width` und `precision`). Bei anderen Ausgabe-Funktionen von C wie `fscanf()` existieren ähnliche Format-Strings.
Die zwei häufigsten Fehler bei der Benutzung von Format-Strings sind:

1. Benutzung eines mit dem `specifier` inkompatiblen Argumentes
2. Verwenden des falschen `specifier` für den übergebenen Datentypen

Generell führen beide Fehler innerhalb eines C-Programms zu 'undefined behaviour'. Der C-Compiler kompiliert das Programm also erfolgreich, behandelt den entsprechenden Code allerdings willkürlich. In Listing 129 ist ein nicht konformes Beispiel für Format-Strings zu sehen.

```c
#include <stdio.h>
int main(void)
{
  // Fehler 1 - Falsche Kombination - 'h' erwartet int
  printf("Falsche Kombination mit 'length': %hs \n", "Falsch");
  // Fehler 2 - Falsche Platzhalter fuer uebergebene Datentypen
  printf("Falsch fuer String: %d und fuer Int: %s \n", "String", 12);
  return 0;
}
```

Listing 129: Nicht konformes Beispiel in C.

Für die Kompilierung dieses Programmes wurde der gcc-Compiler verwendet. Dieser gibt für beide Fehler in Listing 129 Warnungen aus. Die erste für Fehler 1 besagt, dass `h` (Wert für `length` Parameter) nicht mit einem String kombiniert werden kann, sondern einen int erwartet, welcher in einen short umgewandelt werden soll. Hier muss also sowohl der `specifier` als auch der übergebene Wert geändert werden. Für die korrekte Verwendung der Argumente `flag` bzw. `length` wurde eine Tabelle erstellt, welche alle korrekten Kombinationen mit den `specifiern` aufzeigt[89]. Bei Fehlerfall 2 wurde der jeweils falsche `specifier` für einen int- und String-Wert verwendet, welches auch die ausgegebene Compiler-Warnung besagt. Statt dem Testwert 'String' wird für den ersten Platzhalter die Adresse des Wertes ausgegeben, da dieser als int interpretiert wird. Anders bei dem int-Wert, hier wird sogar eine Speicherzugriffsverletzung ausgegeben. Grund dafür ist, dass der Compiler `testI` als char-Pointer interpretiert und somit von der Adresse '12' lesen möchte, auf welche er keinen Zugriff hat. Um dies zu verhindern, müssen die `specifier` getauscht werden.
Allgemein zeigt sich, dass Regel FIO47 hauptsächlich für Programmiereinsteiger wichtig ist. Einem erfahrenen C-Programmierer sind die beschriebenen Zusammenhänge und Zuordnungen bereits klar, wodurch die erwähnten Fehler nicht auftreten.

[89] siehe: https://wiki.sei.cmu.edu/confluence/display/c/FIO47-C.+Use+valid+format+strings

### 2.43.2 Problemstellung in Rust

Innerhalb von Rust werden Format-Strings und vor allem Platzhalter anders definiert. Innerhalb eines auszugebenden Strings wird der Platzhalter mit '{' eingeleitet und mit '}' geschlossen. Innerhalb der geschweiften Klammern ist es möglich, Argumente zu übergeben. Diese werden mit einem Doppelpunkt eingeleitet. Möglich ist hier beispielsweise eine binäre (`:b`), octale (`:o`) oder hexadezimale (`:x`) Darstellung von Zahlen, auch das Angeben von Pointer-Adressen (`:p`) ist realisierbar. In Listing 130 ist zu sehen, wie Format-Strings in Rust verwendet werden können.

```
1 fn main() {
2   // unterschiedliche Datentypen
3   println!("Ausgabe mit String: {} \nAusgabe mit Int: {}", "String", 12);
4   // Benutzung von Argumenten - mit String nicht moeglich
5   println!("Int in Binary: {:b}", 12);
6 }
```

Listing 130: Format-Strings in Rust.

In der ersten Zeile wird deutlich, dass Rust innerhalb der Platzhalter keine Unterscheidungen zwischen verschiedenen Datentypen macht. Die Reihenfolge des Int-Wertes bzw. des String-Wertes macht hier keinen Unterschied. Für die Benutzung der Argumente gilt jedoch trotzdem, dass bestimmte Argumente nur mit bestimmten Werten kombiniert werden dürfen. Würde man für den Platzhalter mit dem Argument `:b` nun einen String verwenden, gibt der Compiler einen Fehler aus und kompiliert das Programm nicht.

### 2.43.3 Fazit

Zusammenfassend lässt sich sagen, dass das Erstellen von Format-Strings in Rust deutlich angenehmer ist als in C. Die in Regel FIO47-C erwähnten Schwachstellen werden größtenteils umgangen. Hilfreich ist hier vor allem, dass keine Datentypen innerhalb eines Platzhalters definiert werden müssen. Somit ist es nicht nötig `specifier` anzugeben, wodurch die Fehleranfälligkeit gemindert ist. Bei der Verwendung von Argumenten muss auch in Rust auf den übergebenen Datentyp geachtet werden. Allerdings kann es dabei in Rust nie zu 'undefined behaviour' kommen, da das Programm im Zweifelsfall nicht kompiliert wird.

# 3 Fazit

Im Großen und Ganzen ist festzustellen, dass die Sprache Rust eine Vielzahl der Secure Coding Regeln der Sprache C unnötig macht. Dies ermöglicht nicht nur die Entwicklung sichererer Software, sondern spart aufgrund der direkten Integration auch Zeit und Ressourcen. Dennoch kann auch bei der Verwendung von Rust nicht auf sämtliche Secure Coding Regeln verzichtet werden.