HAW HAMBURG

BACHELOR THESIS
Jasper Laurens Wolny

Analyse und Evaluation der Einbindbarkeit von C++-Interfaces in Rust

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jasper Laurens Wolny

Analyse und Evaluation der Einbindbarkeit von C++-Interfaces in Rust

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Franz Korf
Zweitgutachter: Prof. Dr. Bettina Buth

Eingereicht am: 5. September 2024

Jasper Laurens Wolny

Topic of the work

Analysis and Evaluation of the Integrability of C++ Interfaces in Rust

Keywords

Rust, C++, interoperability, bindings, ffi, autocxx, bindgen

Summary

Rust and C++ are high-performance programming languages, and the trend of migrating C++ software components to Rust for its memory safety guarantees is growing. One of the biggest challenges in this process is the inter-language communication. This paper evaluates the integrability of C++ libraries into Rust using various tools through unit tests. It then demonstrates how the most promising tool, Autocxx, can be extended to better utilize C++ interfaces.

Jasper Laurens Wolny

Thema der Arbeit

Analyse und Evaluation der Einbindbarkeit von C++-Interfaces in Rust

Stichworte

Rust, C++, Interoperabilität, Bindings, Ffi, Autocxx, Bindgen

Kurzzusammenfassung

Rust und C++ sind leistungsstarke Programmiersprachen, und der Trend, C++-Softwarekomponenten aufgrund der Speichersicherheitsgarantien von Rust zu migrieren, wächst. Eine der größten Herausforderungen in diesem Prozess ist die Kommunikation zwischen den Programmiersprachen. In dieser Arbeit wird die Einbindbarkeit von C++-Bibliotheken in Rust mit verschiedenen Werkzeugen durch Unit-Tests bewertet. Anschließend wird gezeigt, wie das am vielversprechendsten eingeschätzte Werkzeug, Autocxx, erweitert werden kann, um die Nutzbarkeit von C++-Interfaces zu verbessern.

Inhaltsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

C++ ist laut der 2024 Einschätzung von IEEE Spectrum eine der beliebtesten Programmiersprachen des Jahres. [33] Auch im Open-Source Bereich ist C++ sehr beliebt, so gibt es beispielsweise auf Github.com 4,1 Millionen Repositories die Überwiegend C++ nutzen. [29] Trotz seiner weiten Verbreitung in vielen IT-Sektoren, z.B. der Computerspieleentwicklung [1, S. 105] ist C++ selbstvertändlich keine perfekte Sprache. Dies liegt vermutlich an ihrem Alter und ihrer Backwards-Compability mit C, die gewisse Veränderungen an der Sprache von vorneherein ausschließt. Rust ist eine vergleichsweise neue Sprache, die momtan schnell an Beliebtheit gewinnt und die in manchen Aspekten C++ voraus ist. Beispiele dafür sind gewisse Bug-Klassen, die von vorneherein nicht in kompilierbaren Rust Code ausdrückbar sind.

So gibt es beispielsweise standartmäßig keine dem Programmierer frei zugänglichen Pointer sondern nur sichere Referenzen.

In dem folgenden Beispielen wird versucht ein unsicherer C++-Code in Rust nachzuprogrammieren.

Listing 1.1: C++ - Undefiniertes Verhalten

```
int main() {
    int* test = 0; //Null-Pointer
    return *test; //undefiniertes Verhalten
}
```

Dies ist undefiniertes Verhalten, da der C++-Standart kein Verhalten für das Dereferenzieren eines Nullpointers definiert. Folglich ist das Ergebnis Hardwareabhängig.

Listing 1.2: Rust - Versuch undefiniertes Verhalten herzustellen

```
fn main() {
    //Rusts equivalent zu einem int* mit dem Wert 0
    let ptr : Option<&mut i32> = None;
```

```
        //Vermeindlich invalide Referenz
        let noneref : &mut i32 = ptr.unwrap();

        //Unerreichbarer Code
        let num : i32 = *noneref;
    }
```

Den C++-Code nachzustellen stellt sich als unmöglich dar. Dies liegt daran, dass das Gegenstück zu einem Null-Pointer in Rust eine Variante der generischen Option-Enum ist. In Rust sind Enums algebraische Summentypen, d.h. Rust betrachtet diese als eine von mehreren Varianten. *Option* hat hier die Varianten*None* und *Some(&mut i32)*. Im Gegensatz zu C++-Unions ist Rust allerdings konservativ und nur durch Fallunterscheidung (z.B. mit dem *match*-Schlüsselwort) wird Zugriff auf die *Some* -Variante und folglich auf die Referenz in dieser gegeben. Der Fall, das die Option die *Some(&mut i32)*-Variante ist, kann in dem Beispiel nicht eintreten und somit auch kein Dereferenzierung der Integer-Referenz.

Als nächstes wurde in dem Beispiel die *unwrap*-Funktion genutzt, um vermeindlich Zugriff auf die Referenz zu ermöglichen. Dies ist allerdings nicht der Fall, denn die *unwrap*-Funktion beendet das Programm, falls die *None*-Variante eintritt. Allerdings ist dies kein undefiniertes Verhalten sondern ein geplantes Terminieren der Anwendung, was Vorteile bringt. Einige dieser Vorteile sind bessere Debugbarkeit, da die Zeilennummer und optional ein Stack-Trace angezeigt wird. Außerdem ist dies sicherer, da in manchen Fällen undefiniertes Verhalten für IT-Angriffe ausgenutzt werden kann.

So wurden beispielsweise laut eigenen Angaben des Microsoft Security Response Centers in 2019, 70% der als Common Vulnerabilities and Exposures eingeschätzten Bugs durch Speichersicherheitsprobleme verursacht. [30]

Gleichzeitig bietet Rust vergleichbare Möglichkeiten wie z.B. Systemnahe-Programmierung, hohe Anzahl an unterstützten Kompilierungszielen und ähnlich hohe Performance. [4]

Diese Arbeit beschäftigt sich mit dem Nutzen von C++-Interfaces aus Rust und nicht andersherum, weil dies sowohl die Interoperabilität der Sprachen erhört und gleichzeitig die Nutzbarkeit der großen Menge von C++-Bibliotheken erhöht.

Da ein beträchtlicher Teil der industrieführenden Software, wie z.B. die Unreal Engine bei den Spieleengines oder der Windows Kernel, bereits in C oder C++ geschrieben ist, wäre es ineffektiv alle diese in Rust neu zu schreiben. Daher wäre es von Vorteil nur ausgewählte oder neue Komponenten in Rust umzusetzen zu können, bei denen sich so eine Übersetzung besonders lohnen würde. Solche sind Komponenten mit hohen Sicherheits- und Verlässlichkeitsanforderungen wie z.B. Autopiloten.

Softwarekomponenten sind über Interfaces mit uni- oder bidirektional Kommunikation verbunden. Allerdings ist die Interoperabilität von Rust und insbesondere dem C-Quasi-Superset C++ nach wie vor nur beschränkt und aufwendig.

Idealerweise wäre es möglich in der Cargo.yaml C++-Bibliotheken direkt als Dependency zu definieren und dann per Use-Statement am Anfang von Dateien importieren und sicher nutzen zu können.

Ziel der Thesis ist es, den Stand der Interoperabilität von C++ in Rust-Richtung zu bewerten und praktisch zu verbessern. Das heißt konkret die Nutzbarkeit der in den Headerdateien definierten Interfaces zu erhöhen und/oder zu verbessern. In Rust benötigt die Nutzung der externen Funktionen *unsafe*-Blöcke, weswegen sichere Wrapper die Ergonomie erhöhen.

Dies passiert in der Regel durch die Generierung von sog. Bindings, die wie eine Klebeschicht die externen, über die C-ABI umgesetzten Funktionen und Datentypen in der Zielsprache verfügbar machen. Außerdem werden die Structs und Globalen definiert sowie die Funktionnparameter den Structs und Standarddatentypen Gegenstücken in der Zielsprache zugeordnet.

Zwischen C++ und safe-Rust gibt es ein paar fundamentale Unterschiede, welche zu Problemen führen.

Die wichtigsten davon sind:

- Rust erlaubt keine implizite move- und copy-Semantik
- Safe Rust erlaubt keine selbstreferenzierenden Pointer
- Unterschiedliche implementationen von Standarddatentypen wie String-Literalen und ADTs[1]

[1] Abstract data type: Ein Verkapselung von Daten und Operationen, welche für diese Daten valide sind

- Rust erlaubt keine Vererbung von Produkttypen, nur Komposition

Dies liegt daran, dass Rust davon ausgeht, dass alle Datentypen im Speicher frei bewegt werden können ohne invalide zu werden. Die Annahme, dass alle Strukturen trivial bewegt werden können erlaubt Rusts Move-Sematik.

Als nächstes werden in Kapitel 2 die grundlegenden Konzepte die zum Verständniss der restlichen Kapitel notwendig sind ausgelegt. In Kapitel 3 wird als nächstes der Stand der Forschung zum Thema C++-Rust Interoperabilität bewertet. Dann wird in Kapitel 4 beschrieben, wie durch Test des C++-Interface Syntax mit vorhandenen Tools eine Verbessung dessen möglich wird. In Kapitel 5 werden die Tests ausgewertet, Verbesserungsmöglichkeiten eingeschätzt und ein Tool zur Erweiterung ausgewählt. In Kapitel 6 wird eine exemplarische Erweiterung des Tools Autocxx erläutert. Zuletzt wird in Kapitel 7 die Arbeit reflektiert, zusammengefasst und Anstöße zur zukünftigen Forschung gegeben.

2 Grundlagen

In diesem Kapitel wird die Interoperabilität zwischen den Programmiersprachen Rust und C++ untersucht, wobei der Schwerpunkt auf der Einbindung von C++-Code in Rust liegt.

2.1 C++

C++ ist eine weit verbreitete Programmiersprache, dessen Entwicklung 1979 durch Bjarne Stroustrup unter dem Namen „C with Classes" begann. [7, S. 22] Sie erweitert die prozeduralen Programmierparadigmen von C um Objektorientierung und Vererbung, inspiriert an der Programmiersprache Simula. Nach und nach wurden mehr Funktionalitäten hinzugefügt, beispielsweise Referenzen, virtuelle Funktionen, Operator-Überladung und Templates.

2.2 Rust

Rust ist eine moderne Programmiersprache, die 2006 von Graydon Hoare erfunden wurde. Die Entwicklung wurde später von Mozilla übernommen. Hauptziele waren Systemsicherheit und Nebenläufigkeit zu verbessern. Sie bietet strenge Speichersicherheitsgarantien sowie eine Syntax, die auf Sicherheit und Performance abzielt. Gleichezeitig wird eine vielzahl von Programmierparadigmen gleichzeitig unterstützt, u.A.:

- Pattern-Matching und algebraische Datentypen
- Funktionen höherer Ordnung
- Polymorphismus durch Traits
- Generics

Rust ist in 3 Kanälen erhältlich: Stable, Nightly und Beta.

Außerdem gibt es ein standartisiertes System zum Verwalten von Abhängigkeiten die Crates genannt werden. Rust organisiert Code in Modulbäumen. Es gibt pro Crate i.d.R. eine lib.rs und/oder main.rs welche die Modulwurzeln sind. Will man Programmcode in weiteren Dateien anlegen, müssen diese von einer Datei die Teil des Modulbaumes ist als Submodul eingebunden werden. Dies ist grundlegend unterschiedlich zu traditionellem C++. Modul-basiertes Programmieren wird momentan ebenfalls in C++ eingeführt.

2.3 Konzepte

2.3.1 Deklarationen

In C++ sind Deklarationen syntaktische Elemente die Namen einführen oder neu einführen. Definitionen sind Sonderformen von Deklarationen, die alleinstehend zur Nutzung des Elementes ausreichend sind.

Listing 2.1: C++ - Vergleich Definition und Deklaration

```
extern int bar; //Deklaration
int foo = 4; //Definition
```

2.3.2 Einfache-Deklaration

Eine Einfache-Deklaration bezeichnet in C++ eine Sequenz von Spezifikatoren gefolgt von einer Deklarator-Liste. Letzteres ist optional für benannte Klassen und Enums.

Spezifikatoren modifizieren den Datenypen und dessen Eigenschaften.

Listing 2.2: C++ - Einfache-Deklaration

```
    int            *a,          c=2,                          ;
//  ^Spezifikator ^Deklarator   ^ Deklarator mit Initialwert
```

Deklarator

Deklaratoren in C++ definieren einen zu initialisiertenden oder deklarierenden Namen und die Eigenschaften des Namens. Hier kann z.B. ein Pointer auf den spezifizierten Typ oder ein Array Deklariert werden. Der Hauptunterschied zwischen Deklaratoren und den Spezifikatoren ist, dass man keine verschiedenen Spezifikatorenlisten in einer einzelnen Deklarator-Liste haben kann.

Außerdem können im Fall von Definitionen jeweils ein Initialwert zugewiesen werden.

2.3.3 Spezifikator

Const- / Volatile-Qualifikatoren

Der *volatile*-Qualifikator verhindert Entfernung und Umordnung der Anweisung mit den *volatile*-Variaben. Dies wird beispielsweise genutzt, wenn ein Wert im Speicher von Hardware beschrieben wird. [7, S. 1206]

2.3.4 Header

C- und C++-Header sind konventionell Dateien, die nur Deklarationen von Funktionen, Datenstrukturen und globalen Variablen enthalten, die in mehreren anderen Quelldateien verwendet werden können. Da die C++ include-Anweisung nur durch den Inhalt der zu inkludierenden Datei ersetzt wird, ist dies notwendig. Andernfalls würde es im Falle von nicht-*inline*-Definitionen zu einem Mehrfachdefitions-Fehlern kommen. Header-Dateien werden als Interface zwischen Softwarekomponenten verwendet.

2.3.5 Linking

Das Linking ist der Prozess, bei dem verschiedene Teile eines Programms von verschiedenen Kompilationseinheiten zu einer ausfürbaren Datei oder Dynamischen-Bibliothek zusammengesetzt werden. Diese können sogar von verschiedenen Kompilern kommen.

2.3.6 Bindings

Bindings sind Interfaces zwischen Programmiersprachen, die es ermöglichen, Funktionen und Datenstrukturen aus einer anderen Programmiersprache zu nutzen. Dies geschieht indem die Funktionssignaturen, globalen Variablen und Datentypstrukturen in der Programmiersprache nachgestellt werden und dann durch Linkinganweisungen mittels sogenannten Symbolen durch den Linker mit den Funktions- und globalen Variablendeklarationen verknüpft weren.

Listing 2.3: C++ - Binding Ziel

```
extern int bar; //Deklaration

class Test {
    int a;
    int b;
};

void foo(Test&);
```

Listing 2.4: Rust - Binding Beispiel

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Test {
    pub a: ::std::os::raw::c_int,
    pub b: ::std::os::raw::c_int,
}
extern "C" {
    #[link_name = "\u{1}?foo@@YAHAEBH@Z"]
    pub fn foo(arg1: *const ::std::os::raw::c_int);
}
extern "C" {
    #[link_name = "\u{1}?bar@@3PEBDEB"]
    pub static mut bar: *const ::std::os::raw::c_char;
}
```

2.3.7 Name-Mangling

Da Linker unabhängig von dem Kompiler funktionieren ist es notwendig unterschiedlichen Elementen mit dem gleichen Namen unterschiedliche Linkingsymbole zu geben. Dieser Prozess nennt sich Name-Mangling und ist nicht im C++-Standart definiert und somit unterschiedlich je Kompiler. Rust kann das C++-Name-Mangling nicht reproduzieren, folglich muss ein Binding-Generator den C++-Kompiler erkennen und die gemangleten Symbole nachstellen. Alternative kann eine C++-Wrapperfunktion genutzt werden. (Siehe Kapitel 6.1, S. 48)

2.3.8 Templates

Templates ermöglichen generische Programmierung, indem sie Funktionen und Klassen für unterschiedliche Datentypen definieren können. So wird es beispielsweise ermöglicht abstrakte Datentypen wie beispielsweise eine Liste für jegliche Elemente zu entwickeln. Rust verfügt über ähnliche Fähigkeiten mit dem Unterschied, dass der Compiler den Syntax der Templates schon vor der Substitution der Templateparameter überprüft. Mit anderen Worten muss der Templatecode für jeden substituierbaren Datentyp funkionieren. In folgendem Beipiel ist dies nicht der Fall und es kompiliert nicht:

Listing 2.5: Rust - Negativbeispiel generische Addition

```
fn calculate<T>(number: T) -> T {
    number + number
}
```

Um die Methode kompilieren zu lassen ist es notwendig den generischen Parameter um die *Add*- und *Copy*-Traits zu beschrenken, denn der +-Operator ist nicht für alle Datentypen implementiert. Der *Copy*-Trait ist ein Marker-Trait, d.h. er implementiert keine Methoden sondern enthält die Information, dass eine triviale Kopie auf Bit-Ebene möglich ist.

Listing 2.6: Rust - Positivbeispiel generische Addition

```
fn calculate<T : Add + Copy>(number: T) -> T::Output
{
    number + number
}
```

Nichttyp-Template-Parameter

Nichttyp-Template-Parameter sind Template-Parameter, die keine Typen, sondern konstante Werte sind. Daher sind diese in Rust als „Const Generics" bekannt.

Listing 2.7: C++ - Nichttyp-Beispiel

```
float rad(float A) {
    return A * 3.1415;
}
template<float A>
float rad() {
    return A * 3.1415;
}
```

Dies führt zu Performanceoptimierungen auf Kosten der Executable-Größe. Vergleicht man in dem folgendem Außschnitt die erste Version der rad-Funktion mit der Monomorphisation für A = 2.000000 sieht man, dass die Multiplikation zur Kompilierungszeit durchgeführt wurde.

Listing 2.8: C++ - Nichttyp-Beispiel Assembly

```
; https://godbolt.org/
; Kompiler: x64 msvc v19.latest
; Args: -std:c++20 -O3
   float rad(float) PROC ; Funktionsbeginn
   movss DWORD PTR $A[rsp], xmm0 ; a = param1;
   cvtss2sd xmm0, DWORD PTR A$[rsp] ; xmm0 = (double)a;
   mulsd xmm0, QWORD PTR __real@400921cac083126f ; xmm0 *= 3.1415
   cvtsd2ss xmm0, xmm0 ; xmm0 = (float)xmm0;
   ret 0 ; return xmm0 (float/double Return-Register bei cdecl)
float rad(float) ENDP ; Funktionsende

float rad<2.000000>(void) PROC ; Funktionsbeginn
   movss xmm0, DWORD PTR __real@40490e56 ; xmm0 = 6.283
   ret 0 ; return xmm0 (float/double Return-Register bei cdecl)
float rad<2.000000>(void) ENDP ; Funktionsende
```

Teil-Template-Spezialisierung

Teil-Template-Spezialisierung erlaubt es, eine Teilmenge der möglichen Template-Elemente zu spezialisieren.

Explizite Vollständige Template-Spezialisierung hingegen ist die Spezialisierung einer Bestimmten Monomorphisation.

Listing 2.9: C++ - Vollständige und Teil-Template-Spezialisierung

```
// Beispiel: Unendlicher Graph
// Alle Knoten im Universum sind mit Knoten "4" Verbunden
// Knoten 2 ist mit Knoten "3" Verbunden

// Kompiler: x64 msvc v19.latest
// Args: -std:c++20 -O1
template<int A, int B>
inline const bool graph = false; //Primary template

template<int A> //Teil-Template-Spezialisierung
inline const bool graph<A, 4> = true;

template<> //Volle-Template-Spezialisierung
inline const bool graph<2,3> = true;

static_assert(graph<2,3> && graph<1,4>); //Valider Assert
static_assert(!graph<4,2> && !graph<9,1>); //Valider Assert
```

Rust hingegen hat lediglich sogenannte impl-Spezialisierung, ein instabiles Feature welches wohldefinierte Regeln nutzt die spezifischere Trait-Implementierungen bevorzugen, um Performance-Optimierungen zu erlauben. Dieses Feature ist deutlich weniger mächtig als die Teil-Template-Spezialisierung. [32]

2.3.9 Attributdeklarationen

Attributdeklarationen in C++ können verwendet werden, um spezielle Eigenschaften und Verhaltensweisen für Funktionen und Datenstrukturen zu definieren. In Rust gibt

es ebenfalls Attribute mit entsprechenden Effekten. In Rust ist es außderdem möglich benutzerdefinierte Attribut-Macros zu definieren.

2.4 Funktionen

2.4.1 C-ABI

Das C application binary interface (ABI) definiert, wie Funktionen und Daten zwischen verschiedenen Programmen auf der Binärebene ausgetauscht werden. Rust untersützt das C-Speicherlayout durch die #[repr(C)]-Annotation und die C-Aufrufkonvention. Aufrufkonventionen sind ein Vertrag wie ein Funktionsaufruf auf Maschinencode ebene durchgeführt wird.

C++ nutzt Standartmäßig die C-Aufrufkonvention und das C-Speicherlayout. Somit ist es möglich Rust-Structs zu definieren die die Felder exakt auf die C++-Felder projezieren und Funktionsaufrufe mit C++ durchzuführen.

2.4.2 Ausnahmen

Ausnahmen in C++ ermöglichen die Fehlerbehandlung über Throw- und Catch-Blöcke. Diese Exceptions die in einer Funktion aufteten können sind allerdings nicht zwangsläufig Teil der Funktionssignatur. [1] In Rust hingegen werden *Result*- und *Option*-Typen zur Fehlerbehandlung genutzt.

2.4.3 Umwandlungsfunktionen

Umwandlungsfunktionen ermöglichen die Konvertierung zwischen verschiedenen Datentypen in C++. Es gibt 2 Arten von Umwandlungsfunktionen, explizit und implicit. In Rust hingegen wird Typumwandlung nur explizit durch das *From* und *Into* trait ermöglicht.

[1] Die Dynamische-Ausnahme Spezifikation wurde in C++-17 entfernt [20]

2.4.4 Vararg-Funktionen

Vararg-Funktionen in C und C++ akzeptieren eine variable Anzahl von Argumenten. Sie lassen sich auf den backwardskompatiblen C-Stil oder per variadischem Funktionstemplate implementieren:

Listing 2.10: C++ - Vararg Funktionen im C-Stil und als Template

```
#include <iostream>
#include <cstdarg>
using namespace std;

int my_min(int cnt,...){
    va_list ap;
    int i, current, minimum;
    va_start(ap, cnt);
    minimum = 99999;
    for (i=0;i<cnt;i++){
        current = va_arg(ap, int);
        if (current < minimum)
            minimum = current;
    }
    va_end(ap);
    return minimum;
}

template<typename First, typename... Rest>
int min_t(First a, Rest... pack) {
    return a < min_t(pack...) ? a : min\_t(pack...);
}
template<typename First>
int min_t(First f) {
    return f;
}

int main()
{
```

```
    cout <<
    min_t(10,28,27,24,29)
    << "\n";
    cout << my_min(5,10,28,27, 24, 29);
    return 0;
}
```

In Rust werden variadische Funktionen durch Macros zur Kompilationszeit umgesetzt.

2.5 Klassenbezogen

2.5.1 Bitfelder

Bitfelder ermöglichen die Definition von Feldern mit einer bestimmten Bitlänge. Es ist nicht im C++ Standart definiert, aber normalerweise teilen sich Bitfelder Bytes, wenn sie in den vorherig angefangenen Byte passen. [15] In Rust gibt es von der Sprache keine Bitfeldunterstützung.

2.5.2 Vererbung

Mehrfachvererbung

C++ unterstützt Mehrfachvererbung, bei der eine Klasse von mehreren Basisklassen erben kann. Dies erlaubt fortgeschrittene Designmuster wie Mixins:

Listing 2.11: C++ - Mixin Beispiel

```
#include <iostream>

class Draggable {
    bool isDragging;
    float tempX,tempY;
public:
    void startDrag() {
        this->isDragging = true;
        this->tempX = 520.0;
```

```
        this->tempY = 420.0;
    }
};

class Control {
    bool isEnabled = true;
public:
    void disable() {
        this->isEnabled = false;
    }
};

struct Button : virtual Control, virtual Draggable {};
struct Window : virtual Draggable {};

int main()
{
    Button db;

    db.startDrag();
    db.disable();

    Window win;
    win.startDrag();

    return 0;
}
```

Rust verwendet Kompositions- und Trait-basierte Ansätze zur Vermeidung der Komplexität von Mehrfachvererbung.

Virtuelle-Methoden

Virtuelle Methoden sind der Mechanismus der in C++ dynamische Methodenbindung ermöglicht. In C++ werden virtuelle Methoden mit dem Schlüsselwort *virtual* in Basisklassen deklariert und können in abgeleiteten Klassen überschrieben werden. Eine virtuelle

Tabelle (vtable) ermöglicht einen Methodenaufruf der erst zur Laufzeit entschieden wird. Rust hingegen verwendet Traits und Trait-Objekte (*dyn*), um Laufzeit-Polymorphismus zu erreichen.

Pure-Spezifikator / Abstrakte-Klassen

Der Pure-Spezifikator in C++ definiert reine virtuelle Funktionen. Er wird genutzt, um abstrakte Klassen zu erstellen, die nicht instanziiert werden können. Eine reine virtuelle Funktion wird deklariert, indem ein virtuelle Funktion von *=0* gefolgt wird. Diese Funktion muss in den abgeleiteten Klassen implementiert werden. Reine virtuelle Funktionen dienen dazu, ein Interface zu definieren, das von allen abgeleiteten Klassen implementiert werden muss.

2.6 Schlüsselwörter

2.6.1 Inline und Inline-Namespace

Inline-Funktionen, *inline*-Variablen und *inline*-Namespaces sind Techniken zur Optimierung und Organisation von Code. Sie bewirken, dass der Inhalt der Inline-Blöcke an den Stellen der Nutzung kopiert wird, anstatt referenziert zu werden. Da der Aufruf von *inline*-Funktionen den Inhalt der Funktionsdefinition in den Ort des Aufrufs kopiert ist die Interoperabilität mit Rust hier erschwert und erfordert einen C++-Kompiler.

2.6.2 Static-Assert

Static-Asserts sind Prüfungen, die zur Kompilierzeit durchgeführt werden, um Invarianten zu überprüfen. In Rust können ähnliche Mechanismen verwendet werden, um die Integrität des Codes zu sichern. Dies ist allerdings im Kontext der Bindinggeneration nicht zwangsläufig notwendig, da die Static-Asserts bereits zur Kompilationszeit des C++-Codes überprüft wurden.

2.6.3 Using-Direktive, Using-Deklaration

Using-Direktiven und -Deklarationen vereinfachen den Zugriff auf Namen in Namespaces und Klassen. Using-Deklarationen fügen einzelne Elemente in den Namensraum hinzu während Using-Direktiven alle Elemente eines Namespaces einführen. In Rust ist dies mit dem *use*-Schlüsselwort möglich.

2.6.4 Friend

Das *friend*-Schlüsselwort in C++ ermöglicht den Zugriff auf *private*- und *protected*-Mitglieder von Klassen. Da C++-Klassen nur andere C++ Klassen kennen ist dies im Kontxt der Bindinggeneration keine Herausforderung.

2.6.5 Constexpr, Consteval und Constinit

constexpr, *consteval* und *constinit* sind Schlüsselwörter in C++, die konstante Ausdrücke und Initialisierungen definieren.

constexpr-Variablen werden immer zur Kompilationszeit ausgewertet. *constexpr*-Variablen und -Funktionen sind konstant. *constexpr*-Funktionen werden nicht zwangsläufig zur Kompilation ausgewertet während *consteval* dies erzwingt. *constinit* erforert nur eine konstante Auswertung zur Kompilationszeit, zur Laufzeit ist der Wert allerdings nicht zwangsläufig konstant. [17][16][18]

2.6.6 Extern / Static

Der *extern*-Spezifikator erlaubt es globale Variablen zu erstellen. Funktionen sind standartmäßig *extern*.

Das *static*-Schlüsselwort bewirkt in C++ bei Klassenmitgliedern und lokalen Variablen, dass als *static* deklarierten Variblen vom Verhalten der Lebenszeit global werden. Methoden hingegen werden vom Verhalten von Methoden zu Funktionen. Allerdings bleibt der Namensraum der statischen Elemente unverändert.

Bei bereits globalen Variablen und Funktionen bewirkt das *static* hingegen, dass diese intern gelinkt werden, d.h. nicht außerhalb von der eigenen Quelldatei.

3 Stand der Forschung

In diesem Kapitel werden zunächst die relevanten Arbeiten und deren Nutzen vorgestellt. Danach werden die Interoperabilitäts-Tools aus diesen eingeführt. Diese sind hauptsächlich Binding-Generatoren und Source-Code-Transpiler.

3.1 Ansätze von anderen Arbeiten

Benefits and Drawbacks of Using Rust in an Existing C/C++ Codebase

Die erste Kernquelle ist ein Paper, welches den Mehrwert und die Machbarkeit einer teilweisen oder vollständigen Reimplementierung einer großen C++-Codebasis des Cern in Rust bewertet. Hier wurde zu dem Schluss gekommen, dass die automatischen Code-Übersetzung mit den 2 getesteten Tools (crust und c2rust) nicht verlässlich genug war. Deswegen wurde eine manuelle Implementierung einer Softwarekomponente durchführt. Warum Bindgen nicht getestet wurde, ist unklar, da Bindgen zum Zeitpunkt der Veröffentlichen schon 3 Jahre existierte. Autocxx gab es zum Zeitpunkt des Papers noch nicht, da der erste Commit in dessen öffentlichen Repository erst 2020 statt fand. Durch die Erfahungen der Cern-Mitarbeiter kann man den Wert einer automatisierten Lösung besser einschätzen und womöglich manuelle Lösungen, die von automatisierten Binding-Generatoren noch nicht unterstützt werden, übertragen. [3]

Rust programming language in the high-performance computing environment

Die nächsten zwei Kernquellen sind Arbeiten, die die Nutzung von Bindgen untersuchen. Das erste ist ein Paper, welches Rust für den High Performance Computing Bereich bewertet. Es wird sich mit der praktischen Anwendung von Rust in einem C++-dominierten

Bereich auseinandergesetzt. Es wird aufgezeigt, dass Bindings mittels Bindgen und cbindgen (für die umgekehrte Richtung) generiert werden können, ein Test oder eine Bewertung finden allerdings nicht statt. Es wird auch erwähnt, dass Cxx bzw. Autocxx verglichen mit Bindgen zusätzliche Funktionalität bieten können. [8]

Konzeptionierung, Implementierung und Validierung einer Rust basierten Software-Bibliothek zum Management von Zertifikaten in Feldbusgeräten

Die zweite Literatur ist eine Case-Study der Entwicklung einer Rust-Bibliothek. Diese ist zum Verwalten von Zertifikaten. Es wird auf vorhandenen C++-Bibliotheken aufgebaut, wofür Bindgen und Cbindgen genutzt werden. Die Arbeit demonstriert, wie man somit bidirektionale Bindings zwischen Rust und C++ herzustellen kann. Es wird auch gezeigt, wie man eine automatisierte Build-Pipeline für so ein Projekt aufsetzen kann. [10]

Embedding Rust within Open MPI

Die dritte Literaturquelle bewertet Rust für den High-Performance-Computing-Bereich. MPI ist ein de-facto Standart für Nachrichtenaustausch zwischen mehreren Computern, die verteilte Berechnungen koordienieren. Sie schreibt eine Komponente der Open MPI-Middleware erfolgreich in Rust neu und erstellt eine automatische Mehrsprachen-Build-Pipeline. Die Vorteile von einer Reimplementierung in Rust, wie bessere Testbarkeit, Speichersicherheit und Korrektheit, werden versucht nachzuweisen. [9]

Rust for Visual Effects

Zuletzt gibt es noch ein Paper, welches einen eigenen Ansatz zum automatischen Erstellen der Bindings konzipiert hat. Es beschäftigt sich mit der Anwendung von Rust im Bereich der visuellen Effekte (VFX) und hat sich als Ziel gesetzt hoch-qualitative Bindings zu einem existierenden VFX-Software-Stack automatisch zu erstellen. Die Lösung nennen sie Cppmm. Es ist sinnvoll diese Lösung mit der von Bindgen und Autocxx zu vergleichen.

Es wird auf dem GitHub angegeben, dass Template-Methoden / -Funktionen und STL[1]-Kontainer funktionieren, was anderen Tools in manchen Aspekten vorraus sein könnte. [2]

3.1.1 Zusammenfassung

Die analysierten Arbeiten zeigen, dass die Interoperabilität zwischen Rust und C/C++ weiterhin Herausforderungen birgt. Automatisierte Tools wie Crust und C2Rust sind nicht vollständig ausgereift und erfordern oft manuelle Anpassungen, um qualitativ hochwertige Ergebnisse zu erzielen. Bindgen ist das am häufigsten genutzte Tool zur Generierung von Bindings, zeigt jedoch Schwächen bei komplexen C++-Features. Neue Ansätze wie Cppmm und Autocxx versuchen diese Lücken zu schließen, wobei ein formaler Vergleich und eine umfassende Bewertung der Tools bisher fehlen. Zukünftige Forschungen sollten sich auf die Verbesserung und Evaluierung dieser Tools konzentrieren, um eine effizientere Integration von Rust in C/C++-Projekte zu ermöglichen.

3.2 C++-Interoperabilität Tools

3.2.1 Bindinggeneratoren

Bindgen

Bindgen generiert automatisch Bindings zu C- und C++-Bibliotheken. Man kann das Tool entweder per Kommandozeile oder in dem Rust-Build-Script build.rs nutzten. Es nutzt libclang zum Parsen des Codes. Libclang ist ein C-Interface zum Compiler-Frontend Clang welches u.A. C++ unterstützt. Compiler-Frontends sind hauptsächlich für das generieren des Abstrakten Syntaxbaumes verantwortlich. Es hat keine Unterstützung für Features die C++-Kompilation erfordern, wie *inline*-Funktionen. Außerdem sind C++-Features wie Templates, Exceptions und manche C++-spezifische Calling-Conventions nach eigenen Angaben problematisch. [21]

[1] STL: Die Standartlibrary von C++ - von standart template library

Cppmm

Cppmm ist ein Tool der ASFW Rust Workgroup, welches den C++-Code zuerst in C-Header umwandelt und aus diesen dann Rust-Bindings generiert. Das Tool hat nicht den Anspruch mehr C++-Code verarbeiten zu können als für die Maintainer nötig, ist jedoch quelloffen. Es ist ziemlich ähnlich zu Bindgen, nutzt allerdings eine Whitelist mit zu bindenden C++-Interfaces, um präzisere Kontrolle darüber zu ermöglich was genau gebindet werden soll. So ist es sogar möglich nur gewisse Monomorphisationen einer Template-Klasse zu binden. [28]

Cxx

Cxx ist ein Tool, mit dem man in Rust Bindings für sowohl C++ als auch Rust manuell definieren kann. Dies geschieht mittels Makros die eine DSL[2] bilden. Diese Macros generiert dann sowohl für C++ als auch für Rust die gewünschten Bindings und geteilten Structs. [22]

Autocxx

Autocxx ist ein Tool, welches eine Variante von Bindgen nutzt, um automatisch die Cxx-Binding-Definitionen zu erstellen. Im Gegensatz zu Bindgen und Cxx ist es zu einem höherem Grade automatisiert und sicherer. So bietet es beispielsweise eingebaute String-Wrapper und automatische Konstruktoreninvokation. Die Bildpipeline ist ebenfalls deutlich komplexer und in der Lage C++-Code zu kompilieren. So werden beispielsweise Wrapper für *inline*-Funktionen generiert. [23]

3.2.2 Sourcecode-Transpiler

Crust

Crust ist ein Transpiler der C- und C++-Code zu Rust-Code umwandelt. Das Tool selber ist auch ein Rust Programm. Der Transpiler ist allerdings nicht vollständig, so werden z.B. includete Header-Dateien und Funktionspointer nicht Transpiliert. Außerdem ist das Projekt seit 3 Jahren inaktiv. [26]

[2] DSL: Domain specific language - Eine für eine bestimmte Software geschaffene Syntax

C2Rust

C2Rust ist ein Transpiler der C (genauer C-99-Standard) Code in unsafen Rust-Code umwandelt, der den C-Code möglichst nahe spiegeln soll. Außerdem wird ein Befehl angeboten, der probiert, möglichst viel von dem unsafen-Code durch safen-Code zu ersetzten. Es gibt kleine Unvollständigkeiten, aber es wird fast jeder C-99-Code unterstützt. Zudem gibt es ein Tool, was den transpilierten Code mit dem Original-C-Code abgleicht, indem es die Funktionsaufrufe vergleicht. C2Rust nutzt das Clang-Compiler-Frontend, um den C-Code zu parsen, exportiert den AST[3] und dieser wird in Rust umgewandelt. Dieses Tool könnte relevant sein, wenn man zuerst den C++-Code in C-99-Code umwandelt. Es ist Stand September 2024 aktiv. [24]

cc

Cc ist ein Tool, welches es erlaubt von der Rust Buildpipeline aus C- und C++-Code zu kompileren. Sollte man beispielsweise von Rust aus ein C++-Template nutzten wollen wäre es vielleicht möglich die Monomorphizationen mittels Cc zu kompilieren. Es ist außerdem nützlich, wenn man von dem Rust Buildscript aus C- und C++ Quelldateien kompilieren möchte. Cc nutzt den gleichen C++-Compiler der von Rust-Buildpipeline genutzt wird. [27]

3.2.3 Alternative Ansätze

cpp

Das Cpp-Crate erlaubt es inline C++-Snippets einzufügen. Es ist möglich es zusammen mit automatischen Binding-Generatoren zu nutzen. Die inline-Snippets sind Funktionsblöcke und es wird probiert dem Rückgabewert dessen ebenfalls durch das Snippet zurückzugeben. Außerdem wird es versucht zu ermöglichen Rust-Variablen in dem Snippet verfügbar zu machen. Somit soll bidirektionaler Datenaustausch mit dem Inline-C++ möglich werden. [25]

[3] Abstrakt syntax tree: Eine Datenstruktur die den Syntax der Programmniersprache als Datenstruktur repräsentiert

4 Ist-Analyse

Als Erstes werden alle existierenden Binding-Generatoren durch Kriterien gefiltert, welche sicherstellen, dass eine Verbesserung oder Erweiterung zu einem tatsächlichen Fortschritt in der Interoperabilität führt.

Diese Kriterien sind:

1. Das Tool muss in der Lage sein vollautomatisch Bindings herzustellen, um menschliche Fehler zu vermeiden und effiziente Entwicklung zu ermöglichen
2. Die Lizenz darf nicht einschränkend auf die Nutzbarkeit wirken, da proprietäre Software nicht frei genutzt werden kann
3. Das Projekt muss aktiv gewartet werden

Dann wird eine möglichst allumfassende Menge von C++-Interfacefeatures definiert. Hierfür wird die cppreference.com Seite zu Deklaration iterativ abgearbeitet. [19]

Alternativ könnte man auch das Kapitel „Declarations" von open-std.org nutzen, da es sich direkt um die ISO-Spezifikation handelt. [5] cppreference.com bringt die identischen Spezifikationen jedoch kompakter und übersichtlicher auf den Punkt, weswegen es gewählt wurde.

Da C++-Bibliotheken letztendlich eine Menge von Deklarationen in Namespaces sind, welche bei mehrfacher Einbindung nicht zu mehrfach definierten Symbolen führen, besteht diese Menge von Features aus den Syntaxregeln für Deklarationen und *inline*-Definitionen. Definitionen sind Sonderformen von Deklarationen.

Die Wichtigkeit der C++-Features wird anhand des C++ Standarts eingeschätzt (98, 11, 14 etc.) mit dem diese eingeführt wurden. Dies lieg daran, dass C++ versucht möglichst viel Backwards-Compability bereitzustellen. Wenn also eine Verbesserung eines C++98 Features erreicht wird, profitieren davon angenommenerweise auch Nutzer von C++14. Der Standart mit dem die Features eingeführt wurden, ist bei diesen annotiert.

Es ist zu beachten, dass nicht alle C++-Interfacesfeatures Auswirkungen auf die Bindings haben. Das *[[noexcept]]* Attribut beispielsweise ist lediglich ein Vertrag mit der C++-Implementierung. Manche Deklarationen sind nicht einmal von C++ aus interagierbar. *static_assert* ist hierfür ein gutes Beispiel. Zuletzt sind nicht alle Interfaceseigenschaften explizit durch eine Deklaration definiert, manche sind durch die C++-ABI gegeben (z.B. Exceptions) oder sind implizit, wie manche Konstruktoren oder Destruktoren. [6]

Für diese Features werden dann minimale Beispiele in Form von C++-Interfaces geschrieben. Die von allen zu testenden Tools generierten Bindings werden dann bewertet.

Die Bewertung projiziert jedes C++-Interfacefeature auf eine Einschätzung in der Menge:

Tabelle 4.1: Bewertungsstufen der C++-Interfacefeatures

Bewertung	Beschreibung
1	Fehlerbelastete Bindings generiert
2	Lässt Binding-Generation fehlschlagen
3	Keine Bindings für das Feature generiert
4	Teil der Interfacesfunktionalität wird bereitgestellt
5	Interfacesfunktionalität vollständig bereitgestellt

Mit fehlerbelastete Bindings ist gemeint, dass die Bindings erlauben Code zu generieren der die Invaranten des C++-Codes verletzt. Eine Invariante könnte beispielsweise sein, dass eine selbstreferenzierender Pointer immer auf das Element in dem es sich befindet zeigt. In C++ ist dies möglich durch das Überladen mehrer Operatoren.

Es ist es notwendig die Rust-Tests in unabhängige Modulbäume (Kapitel 2.2, S. 6) zu trennen, da alle Tests eines gesamten Modulbaumes gar nicht erst starten, wenn ein einziger Syntaxfehler enthalten ist. Inkorrekter Rust-Code ist notwendig, um die Nichtexistenz von Bindings nachzuweisen. Dies gelingt mit zwei Methoden:

1. Mehrere bin-Module im bin Ordner anlegen
2. Trennen der Tests in eigene Crates

Nach ersten Tests hat sich herausgestellt, dass der erste Ansatzt, auch wenn übersichtlicher, nicht praktikabel ist, da ein Binding-Generator (Autocxx) so designt ist, dass er nur ein mal pro Crate genutzt werden kann. Außerdem gibt es einen potentiellen

Bug in dem Buildscriptsystem von Rust: die Linking Anweisung cargo::rustc-link-arg-bin=BIN=FLAG wendet ein Linkingparameter konditional für ein bestimmtes bin-Modul an. Wenn zwei gleichnamige Objeke A in zwei verschiedenen bin-Modulen auf diese Art verlinkt werden sollen kommt es jedoch zu einem Mehrfach-Symboldefintions-Error.

Die Namenskonvention der Testfälle ist „feature_<Testnummer". Die Testnummern sind an den Syntaxregeln von cppref.com abgeleitet. Wenn ein Interfacesfeature teilweise funktioniert ist eine weitere Crate notwendig, damit der syntaktisch korrekte Code weiterhin getestet werden kann. Die zusätzlichen Datein werden mit einem durch einen Bindestrich eingeleiteten Suffix markiert. Beispiel hierfür wäre „testcase_7-infallible".

Hierfür wird ein build-Script in dem Wurzel-Crate genutzt. Dieses arbeitet eine Datei „testcases.txt" ab, in der jede Zeile einen Testfall enthält. Der Buildscript erstellt die individuellen Test-Crate vollständig.

Dann wird eine Diskussion über die Binding-Generatoren geführt, und ein Tool so wie Features für eine Verbesserung gewählt.

4.1 Ist-Aufnahme

In Kapitel 3 wurden folgende Ansätze zum Generieren von Rust zu C++-Interoperabilität identifiziert:

- Bindgen
- Cppmm
- Cxx
- Autocxx
- Cpp-Crate

Die Cpp-Crate erstellt jedoch keine Bindings, sondern erlaubt lediglich inline C++ per prozeduralem Macro zu schreiben. Cxx ist nicht vollautomatisch und fällt somit auch weg. Betrachtet man die Aktivität der übrigen Projekte erkennt man, dass sowohl an Autocxx und Bindgen aktiv gearbeitet wird. Cppmm allerdings hat seit über einem Jahr kein Commit mehr. Autocxx ist unter Apache 2.0 und MIT lizenziert. Bindgen unter der

BSD-3-Clause Lizenz. Diese Lizenzen sind geeignet zur Kontribution. Folglich werden als nächstes Bindgen und Autocxx untersucht.

Das Ergebniss der iterativen Abarbeitung sind folgende Gruppen von zu untersuchenden C++-Interfacefeatures:

Tabelle 4.2: Getestete C++-Interfacefeatures

- Template-Deklaration (einschließlich teilweiser Spezialisierung)
- Explizite Template-Instantiierung
- Explizite (Vollständige) Template-Spezialisierung
- Namespace-Definition
- Linkung-Spezifikation
- Attributdeklarationen (Seit C++11)
- Funktionsdeklaration ohne Spezifikatoren
- Blockdeklarationen (Deklarationen, die sich in Blöcken befinden Können)
- Spezifikatoren
- Deklaratoren
- Exceptions
- Standartbibliothek

5 Evaluation der Bindinggeneratoren

Als Toolchain für die Tests wird nightly-x86_64-pc-windows-msvc genutzt. Die Rustversion ist rustc 1.80.0-nightly genutzt. Die von Bindgen und Autocxx genutzte Clang-Version ist 16.0.5. Die C++-Quelldateien wurden durch das Cc-Crate mit C++-20 kompiliert. Autocxx wird im Buildscript angewiesen Clang und die Kompilation der C++-Wrapper C++-20 nutzen zu lassen. Diese Rahmenbedingugen gelten für alle folgenden Code-Listings in dieser Arbeit.

Die Test-Crates wurden automatisch auf Basis von /testcases.txt generiert.[1] Der C++-Quellcode befindet sich in /test_headers. Die Nummern der Ergebnisse sind aus Kapitel 4.1 (S. 24) zu entnehmen.

Es wurde Autocxx 0.26.0 und Bindgen 0.70.1 verwendet.

5.1 Template Deklarationen

In der modernen Softwareentwicklung ist die Unterstützung für Templates ein wichtiges Feature, insbesondere bei C++. Templates sind ein zentrales Feature in C++, das die Erstellung generischen Codes ermöglicht. Sie erlauben es, Funktionen und Klassen zu schreiben, die mit beliebigen Datentypen arbeiten können, ohne den Code für jeden speziellen Datentyp erneut schreiben zu müssen.

Insgesamt ist die Unterstützung für Templates in den Binding-Generatoren jedoch noch unausgereift. Obwohl Template-Klassen keine Fehler verursachen, werden deren Methoden in Bindgen und Autocxx nicht eingebunden. Um diese Methoden auszuführen, ist es daher notwendig, Hilfs-C++-Code zu schreiben. Methoden als Klassenmitglieder könnten eingebunden werden, indem man die Methoden der Monomorphisationen in Rust an die von C++ linkt. Autocxx nutzt bereits Monomorphisationen für Template-Typen,

[1] Die Tests befinden sich in /autocxxtest/test/testcase_x_y_z & /bindgentest/test/testcase_x_y_z.

Tabelle 5.1: Testergebnisse Template Deklarationen

Bewertung	Test	Autocxx	Bindgen
2.2.1.1	Nichttyp-Parameter Template Class / Struct	3	4
2.2.1.2	Type-Parameter Template Class / Struct	4	4
2.2.1.3	Template-Parameter Class / Struct	3	4
2.2.1.4	Parameter-Pack Template Class / Struct	4	4
2.2.2.1	Nichttyp-Parameter Union	3	1
2.2.2.2	Type-Parameter Union	4	1
2.2.2.3	Template-Parameter Union	3	1
2.2.2.4	Parameter-Pack Union	4	1
2.2.3	Template Member-Class	4	4
2.2.3.1	Nicht-Typ Template Member-Class	4	1
2.2.4	Template (Member-)Funktion	3	3
2.2.6	Template statische Variable	3	3
2.2.7	Template-Alias (C++11)	5	5
2.2.8	Teil-Template-Spezialisierung	3	2
2.3	Requires / Concept (C++20)	4	1

daher wäre dies in Autocxx vermutlich leichter zu implementieren. Bindgen hingegen definiert eine tatsächlich generische Struktur für Template-Klassen. Ohne Kompilermodifikationen gibt es keinen Weg, gewisse Rust-Monomorphisationen an C++ zu linken. Eine Möglichkeit, ohne einen Extratyp pro Monomorphisation Mitglieder-Methoden zu erstellen, wäre es, Extension-Methods zu nutzen:

Listing 5.1: C++ - Templateklassen-Funktion

```
struct T<A> {
    //Generische methode die an Rust bebindet werden soll
    inline void method(A num) {
        //...
    }
}
```

Listing 5.2: Rust - Binding an generische C++-Methoden

```
#[test]
fn test() { //Nutzung der C++-Methoden
```

```
    let a : T<u8> = T{ a: 2 };
    a.method(2);
    let a : T<u16> = T{ a: 2 };
    a.method(2); //Error, nicht in C++ Monomorphisiert
}
// Implementierung einer spezifischen Monomorphisationen
impl T_methods for T<u8> {
    fn method(&self, u: u8) {
        //call method linked to rust monoporphized method
        T_method_char(u);
    }
}
pub trait T_methods { //Erweiterungs-Trait
    fn method(&self, u: u8);
}
extern "C" {
    //gemangletes Symbol (versimpelt)
    #[link_name = "\u{1}?T_methodZ"]
    pub fn T_method_char(u8);
}
```

Somit könnte man ein Linking pro Monomorphisation erreichen. Das finale Ziel wäre es, in Rust neue Monomorphisationen generieren zu können. Autocxx ermöglicht es bereits mit dem *concrete!*-Makro, neue Monomorphisationen zu erstellen. Idealerweise würde die Nutzung neuer Monomorphisationen automatisch erkannt werden. Dies würde allerdings Rust-Kompilermodifikationen oder einen Präprozessor erfordern, der alle Monomorphisationen erkennt. Dann könnte hierfür neuer C++-Code generieren werden. Rusts Kompiler bietet ein Monomorphisation-Sammler der eine Iteration über diese ermöglicht. [31]

Template-Unions

Bei Template-Unions und Nichttyp-Parameter Memberklassen kommt es bei Bindgen zu Syntaxfehlern in den Bindings. Außerdem ist es in Bindgen möglich, die Invarianten von Requires und Teil-Template-Spezialisierung zu brechen.

Nichttyp-Templateparameter

Nichttyp-Templateparameter-Klassen wurden von Autocxx vollständig ignoriert und von Bindgen als opake behandelt. Im Nightly-Rust gibt es Unterstützung für Const Generics (siehe Kapitel 2.3.8, S. 9), welche man hinzuzufügen könnte.

Template globale Variablen sind von beiden Generatoren nicht unterstützt, ließen sich aber durch eine globale Variable pro Monomorphisation lösen.

Template-Alias

Templatealiase (Test 2.2.7) sind Typalias-Deklarationen (Kapitel 5.8, S. 34), welche Template-Parameter auf der Rechten-Seite des Alias nutzen. Sie wurden überwiegend unterstützt, Methoden sind allerdings ebenfalls nicht gebunden.

Teil-Template-Spezialisierung

Teil-Template-Spezialisierung ist eine generische Template-Spzialisierung, d.h. eine solche die für eine Teilmenge der möglichen Templateparameter gilt. Dies gelingt beispielsweise durch die Ersetzung von einem Template-Parameter durch einen konkreten Typ oder die mehrfache Verwendung eines Template-Parameters. Dieses Feature wurde garnicht unterstützt und hat bei Bindgen sogar zum scheitern der Bindinggeneration geführt. Eine Lösung wäre es die Templates in C++ bzw. LLVM aufzulösen und dann die anderen Lösungsansätze mit der spezialisiertung Linkung durchzuführen.

5.2 Explizite Template-Instantiierung

Tabelle 5.2: Testergebnisse explizite Template-Instantiierung

Bewertung	Test	Autocxx	Bindgen
3	Explizite Template-Instantiierung	5	5

Explizite template-Instantiierung stellt sich als kein Hinderniss für die Binding-Generatoren dar, da es keine Änderung der Interfaces bewirkt.

5.3 Explizite Template-Spezialisierung

Tabelle 5.3: Testergebnisse explizite Template-Spezialisierung

Bewertung	Test	Autocxx	Bindgen
4.1	Funktionstemplate	2	1
4.2	Class-Template	3	3
4.3	Variable (C++14)	3	3
4.4	Memberfunktion von Class-Template	3	3
4.5	Statischer Datenmember von Class-Template	3	3
4.6	Member-Class von Class-Template	2	3
4.7	Member-Enum von Class-Template	2	3
4.8	Member-Class-Template	3	3
4.9	Memberfunktions-Template	3	3
4.10	Membervariablen-Template	3	3

Explizite Template-Spezialisierung ist die Spezialisierung einer bestimmten Template-Monomorphisation. Die Binding-Generatoren bieten hierfür keine Unterstützung oder scheitern in einigen Fällen vollständig. In Rust gibt es fast keine Unterstütztung für explizite Template-Spezialisierung mit der Ausnahme von Trait-impl-Spezialisierung. [11] Dies ist allerdings ohnehin irrelevant, weil von vorneherein nur die in C++ monomorphisierten Varianten genutzt werden können und somit eine Implementierung für jegliche Typen unmöglich ist.

Eine Unterstützung ließe sich durch die gleichen Methoden die in für Templates genannt wurden erreichen, denn nachdem man die Monomorphisationen in C++ oder durch LLVM aufgelöst wurde ist es belanglos, ob es sich um ein teil- oder vollständig spezialisiertes Template handelt.

5.4 Namespace

Namespaces werden überwiegend unterstützt. Auch inline-Namespaces funktionieren mit dem kleinen Manko, dass die explizite Addressierung des *inline*-Namensraums nicht funktioniert. Anonyme Namensräume wurden von beiden Tools ignoriert. Mit :: Verschachtelte Namenräume sind mehrere verschachtelte Namensräume die mit einem *namespace*-Keyword definiert werden. Ein Punkt in dem hier Autocxx Bindgen überlegen ist, dass

Tabelle 5.4: Testergebnisse Namespace

Bewertung	Test	Autocxx	Bindgen
5	Namespace	5	5
5.1	Inline-Namespace (C++11)	4	4
5.2	Anonymer Namespace	3	3
5.3	Verschachtelter Namensraum (Mit ::) (C++17)	5	5
5.3.1	Inline-verschachtelter Namensraum (Mit ::)	2	2

Namespaces in Rust Modulen abgebildet werden, während Bindgen diese nur mit Unterstrichen kennzeichnet. Die Inline-Verschachtelten Namensraumdefinitionen sollten theoretisch auch funktionieren können.

5.5 Linking

Tabelle 5.5: Testergebnisse Linking

Bewertung	Test	Autocxx	Bindgen
6.1	Extern "C" Blöcke	3	5
6.2	Extern "C" Deklarationsmodifikator	3	5

Extern "C" Blöcke und Linking-Modifikatoren wurden von Bindgen vollständig unterstützt, Autocxx hat die als extern "C" deklarierten Funktionen und Variablen vollständig ignoriert. Dies ist vermutlich nachbesserbar.

5.6 Attributdeklarationen (Seit C++11)

Tabelle 5.6: Testergebnisse Attributdeklarationen

Bewertung	Test	Autocxx	Bindgen
7	Attributdeklarationen (Seit C++11)	5	5

Die in dem Standart definierten C++-Attribute [14] verursachen alle keine Fehler, haben auf die Interfaces jedoch keinen Einfluss. Das *alignas*-Attribut sowie Übertragung

der C++-Dokumentation als Rust-doc-Annotationen funktioniert ebenfalls. Es gibt kleine Ungenauigkeiten, wie z.B. das *deprecated*-Attribute oder *nodiscard*-Attribute nicht in Rust umgesetzt werden, obwohl es hierfür Gegenstücke gibt.

5.7 Funktionsdeklaration ohne Spezifikatoren

Tabelle 5.7: Testergebnisse Funktionsdeklaration ohne Spezifikatoren

Bewertung	Test	Autocxx	Bindgen
9.1	Destruktor	5	1
9.2	Konstruktor	5	4
9.3	Umwandlungsfunktion	3	3

In C++ gibt es 3 verschiedene Arten von Funktionsdeklarationen ohne Spezifikatoren.

5.7.1 Destruktor

Autocxx garantiert im Gegensatz zu Bindgen den Aufruf der Destruktoren. Dies geschiet durch eine Implementierung des Drop-Traits, die an den C++-Destruktor gelinkt ist.

5.7.2 Konstruktoren

Explizite Konstruktoren werden von beiden Generatoren gebunden. Implizite Konstruktoren werden allerdings nur von Autocxx verfügbar gemacht. Da dies in Bindgen nicht passiert, hat man in solchen Fällen auch keinen Zugriff auf die Standartwerte der Memberfelder und als einzige Option die Rust Initialisierungsliste. Die Copy- und Move-Konstruktoren wurden außerdem von beiden Generatoren gebunden. Man sollte überlegen, ob man den Copy-Konstruktor nicht in Rust *clone* nennt, da dies konventionell der Methodenname für nicht-triviale Kopien ist.

5.7.3 Umwandlungsfunktion

Umwandlungsfunktionen werden nicht gebunden. Obwohl Rust keine implizite Konvertierung ermöglicht, wäre es möglich, die Implementierung des Rust-Traits *From* an die Umwandlungsfunktionen zu binden.

5.8 Blockdeklarationen

Tabelle 5.8: Testergebnisse Blockdeklarationen

Bewertung	Test	Autocxx	Bindgen
10.2	Typalias-Deklaration (C++11)	5	4
10.3	Namensraumalias-Definition	3	3
10.4	Using-Deklaration	5	5
10.5	Using-Direktive	3	3
10.6	Using-Enum-Deklaration	3	3
10.7	Static_assert (C++11)	5	5
10.8	Opake-Enum-Deklaration (C++11)	5	5

Blockdeklarationen sind eine Gruppe von Deklarationen die sich an in jeglichen Blöcken befinden können. Dies geht davon aus, dass die gesamte Kompilierungseinheit einen globalen Block bildet. Asm-Deklarationen wurden nicht untersucht, da sie nur genutzt werden könnten um Funktionsdefinitionen zu generieren.

5.8.1 Typalias-Deklaration

Typaliasdeklarationen sind ähnlich zu dem C-typedef, unterstützen jedoch auch Templateparameter. Es ist zu beachten, dass diese ebenfalls für den Template-Alias-Test genutzt wurden (Test 2.2.7, Kapitel 5.1, S. 28). In Bindgen hat sowohl die Nutzung mit und ohne Templateparameter funktioniert, während bei Autocxx nur Typaliase ohne Templateparameter funktioniert haben.

5.8.2 Namensraumalias-Definition

Namensraumaliasdefinitionen wurden von den Tools ignoriert. Man könnte allerdings testen, ob ein Re-export in dem Alias-Modul nicht den gewünschten Effekt hätte:

Listing 5.3: Rust - Exemplarische Umsetzung von Namensraumaliasen

```
mod fbz { pub use foo::bar::baz; }
```

5.8.3 Using

Für *using* siehe Kapitel 2.6.2 (S. 16).

Using-Deklarationen haben bei beiden Generatoren einwandfrei funktioniert.
Using-Direktiven haben bei beiden Tools nicht funktioniert und ließen sich vermutlich mit der gleichen Methode wie Namensraumaliase ermöglichen.

Listing 5.4: Rust - Exemplarische Umsetzung von Using-Direktive

```
pub use foo::bar::baz::*;
```

5.8.4 Using-Enum-Deklarationen

Using-Enum-Deklarationen sind ein C++20-Feature, dass es ermöglicht Enum-Felder zu Klassen hinzuzufügen. Diese könnte man vermutlich mit Associate-Konstanten ermöglichen:

Listing 5.5: Rust - Exemplarische Umsetzung von Using-Enum-Deklaration

```
enum Fruit { Orange, Apple }

struct Test {}

impl Test {
    const ORANGE : Fruit = Fruit::Orange;
    const APPLE : Fruit = Fruit::Apple;
}
```

5.8.5 Static_assert

static_assert (Kapitel 2.6.1, S. 16) hat bei beiden Generatoren zu keinen Problemen geführt.

5.8.6 Opake-Enum-Deklaration

Opake-Enum-Deklarationen sind ein in C++11 eingeführter Syntax für eine Enum ohne Angabe der Elemente. Opaque enums wurden vollständig von beiden Generatoren unterstützt. Somit könnte man aus dem Blickwinkel von C++-Interoperabilität womöglich für Blockdeklarationen volle Unterstützung erreichen.

5.9 Exceptions

Tabelle 5.9: Testergebnisse Exceptions

Bewertung	Test	Autocxx	Bindgen
12.1	Exception Abfangen	4	3
12.2	Exception Abfangender Block	3	3
12.3	Ausnahmen durch Rust übergeben	3	5

Autocxx fängt C++-Ausnahmen ab, zeigt sie in der Standartausgabe an und terminiert das Programm. Bindgen hingegen lässt die Exception hingegen freien Lauf. Es wäre möglich eine Art *try*-*catch*-Block bereitzustellen der, das Behandeln der Ausnahme erlaubt:

Listing 5.6: Rust - Protoyp eines try-catch-Makros

```
try! [
    {
        ffi::ThrowsMaybe(); //Ffi-Code
    } => (...) { //catch
        //Exception handler
        println!("Exception!");
    }
]
```

Dies wäre in Autocxx leichter, da es bereits die Funktionsaurufe in einen *try*-*catch*-Block einschließt. Dann könnte man die *try*-*catch*-Blöcke als Standard für jeden Methodenaurruf entfernen und somit das Übergeben von Ausnahmen durch Rust erlauben.

Der Test des Übergebens einer Ausnahme durch Rust war in Autocxx nich erfolgreich, da die Funktion die einen Funktionspointer als Parameter nimmt nicht generieret wurde. Mit

Bindgen hat die Durchgabe einwandfrei funktioniert, es müsste jedoch genauer untersucht werden, ob dies verlässlich funktioniert.

5.10 Standartbibliothek

Tabelle 5.10: Testergebnisse Standartbibliothek

Bewertung	Test	Autocxx	Bindgen
13.1	std::String	5	3
13.2	std::Vector	4	3
13.3	std::List	3	3
13.4	std::Set	3	3
13.5	std::Map	3	3

Bindgen bietet keine besondere unterstützung für die ADTs der C++-Standartbibliothek an. Autocxx bietet u.A. für String und Vector Wrapper an, jedoch nur für einen kleinen Teil der STL.

5.11 Spezifikatoren

5.11.1 Inline

Unter den Spezifikatoren (Kapitel 2.6, S. 16) sind besonders *inline*-Funktionen bemerkenswert, da nur Autocxx diese unterstützt. *inline*-Variablen hingegen werden von beiden Generatoren vollständig unterstützt, da sie als Konstanten umgesetzt werden können.

5.11.2 Friend

Das *friend*-Schlüsselwort (Kapitel 2.6.3, S. 17) hat in Rust kein Equivalent und es wurde lediglich getestet, ob die Generatoren keine Probleme verursachen. Dies war bei beiden Generatoren der Fall.

Tabelle 5.11: Testergebnisse Spezifikatoren

Bewertung	Test	Autocxx	Bindgen
11.2.1.1	Typedef	5	5
11.2.1.2	Inline-Funktion	5	3
11.2.1.3	Inline-Variable (C++17)	5	5
11.2.1.4	Friend	5	5
11.2.1.5	Constexpr (C++11)	5	4
11.2.1.6	Consteval / Constinit (C++20)	3	2
11.2.1.7.2	Static	4	5
11.2.1.7.3	Thread_Local (C++11) (mit extern / static)	3	2
11.2.1.7.4	Extern	4	5
11.2.1.7.5	Mutable	5	5
11.2.1.8.2	Enum-Spezifikator	5	5
11.2.1.8.3	Standartdatentypen	2	5
11.2.1.8.4	Auto / Decltype (Beides C++11)	5	5
11.2.1.8.6	Zuvor Deklarierter Datentyp oder Typalias (C++11)	5	5
11.2.1.8.7	Template-Name mit Template-Argumenten	5	5
11.2.1.9	Elaborierte Typspezifikatoren	3	5
11.2.1.11	Const- und Volatile-Qualifikatoren	5	5

5.11.3 Constexpr / Consteval / Constinit

Siehe Kapitel 2.6.4 (S. 17).

Autocxx bietet im Gegensatz zu Bindgen volle Unterstützung für *constexpr*-Variablen sowie Funktionen an. *consteval* und *constinit* wurden nicht unterstützt. Die Nutzung von *consteval*-Funktionen ist an eine Auswertung des C++-Kompilers gebunden und folglich aus Rust schwer. Eine Möglichkeit wäre es die *consteval*-Funktion als funktionsartiges prozedurales Makro in Rust zur Verfügung zu stellen. Dieses wertet dann zur Rust-Kompilationszeit die C++ *consteval*-Funktion aus. Dies wäre dann auch mit *constexpr*-Funktionen möglich, was zu Performancegewinn führen würde. *constinit* globale Variablen sind im Gegensatz zu *constexpr*-Variablen nicht zwangsläufig konstant, sondern nur die Initialisierung muss zur Kompilationszeit auswertbar sein. Somit müssten diese auch vollständig unterstützbar sein.

5.11.4 Static

Binding unterstützt *static* (Kapitel 2.6.6, S. 17) vollständig, Autocxx nur die statischen Methoden während die statischen Felder vollständig ignoriert wurden, sogar konstante. Insbesondere das Binden von konstanten statische Feldern sollten relativ trvial umsetzbar sein.

5.11.5 Thread_local

Thread_local globale Variablen sind von Autocxx noch nicht unterstützbar, da es keine nicht-konstanten Globalen unterstützt. Bindgen hingegen hat die thread_local Globale gebindet, die Nutzung dieser in zwei verschiedenen Threads hat jedoch zu einem Programmabsturz mit Zugriffsverletzung geführt.

5.11.6 Extern-Spezifikator

Der *extern*-Spezifikator (Kapitel 2.6.5, S. 17) wird vollständig von Bindgen unterstützt, doch Autocxx unterstützt nur die externen Funktionen, selbst wenn diese konstant sind. Nicht konstante globale Variablen werden von Autocxx nicht unterstützt.

5.11.7 Mutable

Das C++-*mutable*-Keyword erlaubt bestimmte Felder von konstanten Objektinstanzen schreibbar zu machen. Dies ist in Rust nicht trivial umzusetzen, man könnte jedoch einen Getter und Setter als Methoden bereitstellen, welche Unsafe-Code nutzen, um die Modifizierbarkeit zu erreichen. Hierzu ist es notwendig zusätzlichen C++ zu kompilieren.

5.11.8 Enum & Standartdatentypen

Enums wurden von beiden Generatoren vollständig untersützt.

Außerdem hat der seit C++20 existierende Standartdatentyp *char8_t* sowie *char32_t* und *wchar_t* die die Bindinggeneration von Autocxx vollständig scheitern lassen.

5.11.9 Auto / Decltype (Beides C++11)

Das *auto*-Schlüsselwort inferiert Datentypen und das *decltype*-Schlüsselwort wertet den Datentyp einer bereits existierenden Deklaration aus. Diese Typespezifikatoren wurden beide von den Bindinggeneratoren erfolgreich genutzt.

5.11.10 Zuvor deklarierter Datentyp

Zuvor deklarierte Datentypen so wie Typealiase wurden von beiden Bindinggeneratoren unterstützt. Dies gilt auch für Template-Datentypen und Typaliase.

Außerdem erlauben es elaborierte Typspezifikatoren Datentypen zu nutzen die von einem Variablennamen versteckt wurden. Dies hat nur bei Bindgen zu Bindinggeneration der Deklaration geführt.

5.11.11 Const- und Volatile-Qualifikatoren

Siehe Kapitel 2.3.3 (S. 7). Die *const*- und *volatile*-Qualifikatoren wurden unterstützt. Der *volatile*-Qualifikator hatte jedoch keinen Effekt auf die Bindinggeneration weswege hier Vorsicht wichtig ist.

5.12 Klassen-Spezifikatoren

Tabelle 5.12: Testergebnisse Klassen-Spezifikatoren

Bewertung	Test	Autocxx	Bindgen
11.2.1.8.1.1	Öffentlich / Privat	5	1
11.2.1.8.1.2.1	Bitfelddeklaration	3	5
11.2.1.8.1.2.2	Pure-Methoden	5	5
11.2.1.8.1.2.3	Override / Final (C++11)	5	4
11.2.1.8.1.2.4	Triviale Methoden	5	5
11.2.1.8.1.5	Static_assert-Deklarationen	5	5
11.2.1.8.1.10	*Nicht Pod-Sicherheit*	5	1
11.2.1.8.1.11	*Vererbung (inkl. Polymorphismus)*	4	4
11.2.1.8.1.12	*Multi-Vererbung*	4	4
11.2.1.8.1.13	*Standardfelderwerte*	5	4

Klassen-Spezifikatoren sind Spezifikatoren, die nur im Kontext von Klassen- bzw. Unionfeldern anwendbar sind. Die Klassenspezifikatoren werden größtenteils unterstützt.

5.12.1 Public / Private

Obwohl Rust genau wie C++ private und public Datenfelder hat, werden private Felder von Bindgen in Rust als public deklariert. Dies ist womöglich eine bewusste Entscheidung, da Bindgen implizite Konstruktoren nicht bindet und es sonst womöglich keinen Weg geben könnte die Objekte zu initialisieren. Würde man diese Binden, dann wären private Felder wahrscheinlich relativ trivial zu aktivieren.

5.12.2 Bitfelder

Bitfelder (Kapitel 2.5, S. 14) hingegen werden nur durch Bindgen unterstützt, durch Getter- und Setter-Methoden. Dies könnte man auch in Autocxx einführen.

5.12.3 Abstrakte Klassen

Abstrakte-Klassen (Kapitel 2.5.2, S. 16) lassen sich mit beiden Generatoren erfolgreich binden.

5.12.4 Virtuelle Methoden

Virtuelle-Methoden (Kapitel 2.5.2, S. 15) wurden von beiden Generatoren unterstützt. Für das Erben von Klassen und Überschreiben von virtuellen Methoden hat nur Autocxx Mechanismen bereitgestellt, welche funktionieren. Dies ließe sich in Bindgen hinzufügen. Hier könnte man ergonomischere Ansätze probieren, als die Annotationsmakro-basierte Lösung von Autocxx.

5.12.5 Triviale Methoden

Triviale Methoden werden von beiden Generatoren vollständig unterstützt.

5.12.6 Nicht-Pod-Sicherheit

Pod in C++ (eng. plain old data) bedeutet, dass Klassen sich trivial kopieren, bewegen und zerstören lassen. Ist dies nicht der Fall, dann sind diese keine Pod-Klassen und erfordern erweiterte Invarianten.

Nicht-Pod-Sicherheit ist die Aufrechterhaltung dieser Invarianten.

Nicht-Pod-Sicherheit wird nur von Autocxx geboten, bei Bindgen hingegen ist man für die Speichersicherheit und das Aufrufen von Move- und Kopierkonstruktoren allerdings vollständig selbst verantwortlich. Im Falle von Mehrfachvererbung funktioniert ebenfalls genau das, was bei einfacher Vererbung funktioniert.

5.12.7 Vererbung

Vererbung sowie Multivererbung funktioniert grundsätzlich, jedoch ist es nicht möglich Pointer und Referenzen von Kindklassen als Paramter der Baseklassen zu nutzen. Dise ist auch als Subtyp-Polymorphismus bekannt. Außerdem werden nicht-virtuelle Methoden nicht auf Kinderklassen vererbt und virtuelle Methoden, welche nicht überschrieben wurden, wurden ebenfalls nicht vererbt.

In Bindgen ist es möglich diese Limitierungen leicht zu überwinden, da die Baseklasse ein öffentliches Feld ist. Es wäre allerdings möglich alle diese Hindernisse zu überwinden, indem man Traits als Parameter nimmt und die Methoden für diese Traits implementiert. Dann baut man die Klassenhierarchie nach, indem man die Baseklassentraits generisch für alle Kinderklassen implementiert.

5.12.8 Standardwerte

Standardfelderwerte funktionieren bei beiden Generatoren, wenn der Konstruktor explizit ist. Im Falle von einem impliziten Konstruktor ist es in Bindgen nötig den Rust-Feldinitializator zu nutzen und somit ist kein Zugriff auf die Standardwerte möglich.

Tabelle 5.13: Testergebnisse Deklaratoren

Bewertung	Test	Autocxx	Bindgen
11.2.2.4	Pointerdeklarator	5	5
11.2.2.5	Funktionspointerdeklarator	3	1
11.2.2.6	Pointer auf Memberdeklaration	1	5
11.2.2.7	L- / R-Wert Deklaration	2	5
11.2.2.8	Array-Deklarator	5	5
11.2.2.9.1	Trailing-Return Typ (C++11)	5	5
11.2.2.9.2	Ref-Qualifikator (C++11)	2	5
11.2.2.9.4	Noexcept-Spezifikation (C++17)	5	5
11.2.2.9.5	Requires-Klausel (C++20)	2	3
11.2.2.9.6	Default Parameter	3	3
11.2.2.9.7	Variadische Funktionen	3	5

5.13 Deklaratoren

Zum Testen der Deklaratoren (Kapitel 2.3.2, S. 6) wurden Funktionsrückgabewerte genutzt, da die Unterstützung von globalen Variablen in Autocxx unvollständig ist und diese ebenfalls den Syntax für Deklaratoren nutzen, jedoch ohne Namen. Alternativ könnte man auch Klassenfelder zum Testen nutzen.

5.13.1 Pointerdeklarator

Pointer funktionieren einwandfrei und wurden zu Rust-Pointern übersetzt. Alternativ könnte man auch einen Rust-Borrow der in einer *Option* ist nutzen.

5.13.2 Funktionspointerdeklarator

Funktionspointer werden von Bindgen gebunden und es wird die C-ABI angenommen. Dies kann zu Fehlern führen. Da Rust die C++-ABI nicht unterstützt dürften keine Bindings generiert werden. Funktionspointer auf extern "C" Funktionen funktionieren allerdings ebenfalls.

5.13.3 Pointer auf Memberdeklaration

Pointer auf Memberdeklarationen werden bei Bindgen und Autocxx gebunden, bei Autocxx werden sie allerdings fehlerhafterweise einem 8-Byte Pointer und nicht einem 4-Byte Wert zugewiesen, was zu fehlerhaftem Verhalten führt, da die vier signifikantesten Bytes Undefined Behavior entsprechen.

5.13.4 L- / R-Wert Deklaration

L-Werte (eng. lvalue) sind Werte, die auf beiden Seiten einer Zuweisungsoperation stehen können, während R-Werte (eng. rvalue) nur auf der rechten Seite stehen können. L- bzw. R-Wert-Referenz-Deklaratoren lassen sich jeweils mit & und && ausdrücken.

Bindgen untersützt beide als Pointer. Autocxx allerdings generiert R-Wert-Referenzen bei Rückgabewerten nur dann, wenn es die Lifetime anhand der Funktionsparameter deduzieren kann. Außerdem scheitert die Bindinggeneration vollständig, wenn man Referenzen als Klassenfelder nutzt.

5.13.5 Array-Deklarator

Arrays wurden vollständig unterstützt. Die Trailing-return-type Schreibweise wurde eingeführt, um die Limitation, dass Template-Funktionsrückgabewerte nicht anhand der Parametertypen deduziert werden konnten, zu beseitigen. Diese Schreibweise wurde ebenfalls unterstützt.

5.13.6 Ref-Qualifikator

Die *ref*-Qualifikatoren sind nur mit Klassenmethoden nutzbar und erlauben eine Überladung abhängig davon, ob *this* ein lvalue oder rvalue ist. Da Rust keine verschiedenen Typen für r- und lvalue-Referenzen hat, ist dies schwer in Rust auszudrücken. Bindgen hat die Methoden gebunden und die Qualifikatoren ignoriert, während die Bindinggeneration in Autocxx vollständig gescheitert ist.

5.13.7 Noexcept-Spezifikation (C++17)

Die *noexcept*-Spezifikation wurde von beiden Bindinggeneratoren unterstützt und hat zu keine Veränderung in den Bindings geführt.

5.13.8 Requires-Klausel

Mit *requires*-Klauseln kann man in C++ auf Parameter von Templatemethoden Einschränkungen erzwingen.

5.13.9 Default-Paremeter

Default-Paremeter werden von beiden Generatoren ignoriert. Eine mögliche Lösung wäre es, diese Werte in eine Rust-Enum mit den Varianten Some(T) und Default zu verkapseln. Dann müssten die Default-Werte nach Notwendigkeit durch einen Wrapper eingefügt werden.

5.13.10 Variadische Funkionen

Variadische Funkionen (Kapitel 2.4.4, S. 13) werden nur vor Bindgen gebunden. Diese müssten auch in Autocxx unterstützbar sein.

5.14 Zusammenfassung und Bewertung

Die Summe der Ergebnisse wie in Tablle 4.1 (S. 24) beschrieben entspricht:

Tabelle 5.14: Summe der Testergebnisse

Bewertung	Autocxx	Bindgen
1	1	11
2	8	4
3	33	26
4	14	13
5	34	36

Zusammenfassend haben beide Tools zwar vergleichbare Nutzbarkeit, jedoch Große Unterschiede:

Bindgen ist ein minimalistischeres Werkzeug, welches dem Entwickler die Verantwortung überlässt, die Speichersicherheitsprobleme zu lösen und weniger Extras wie z.B. String-Wrapper mitliefert. Es ist weniger einschränkend in der Nutzung, kann aber zu schwer zu findenden Bugs führen, da es leicht macht C++-Invarianten zu verletzen.

Autocxx hingegen versucht die Nutzung der Bindings so verlässlich wie möglich zu machen, erhöht dabei aber die Komplexität. So ist man beispielsweise gezwungen mit verschiedenen Wrappern für C++-Objekte und Gepinnten Referenzen zu arbeiten. Außerdem ist die Kompilierzeit der Bindings mit Autocxx merkbar langsamer.

Darüber hinaus werden andere Funktionalität geboten, die Bindgen nicht bereitstellt, wie z.B. das Überschreiben von virtuellen C++-Methoden durch eine Rust-Struct.

Ein vielgehörter Grund warum Rust von Entwicklern bevorzugt wird, sind dessen hohe Verlässlichkeits- und Sicherheitsgarantien. Da Autocxx diese deutlich weniger verletzt, was besonders an den 11 Invariantenverletzungen von Bindgen gegen einer von Autocxx deutlich wird, wird nun exemplarisch eine Erweiterung von Autocxx durchgeführt.

6 Implementation

In diesem Kapitel wird die Vorgehensweise beschrieben, die zur praktischen Verbesserung der C++-Interoperabilität in Rust mittels des Autocxx-Tools angewandt wurde. Zunächst wird die Struktur und Funktionsweise von Autocxx im Detail erläutert, um ein grundlegendes Verständnis für das Tool und seinen Ablauf zu schaffen.

Anschließend wird exemplarisch ein ausgewähltes Feature in Autocxx implementiert, nähmlich Subtyp-Polymorphismus von Methoden und Parametern. Das fertige Feature soll als Kontribution zu dem Projekt taugen.

Zuletzt werden die restlichen fehlenden Interface-Features sowie gefundenen Bugs von Autocxx identifiziert und in ihrer Wichtigkeit und Schwierigkeit der Implementierung eingeschätzt.

6.1 Struktur von Autocxx

In Autocxx wird eine im Buildscript angegebene Rust Datei zum konfigurieren der Bindings genutzt. Diese muss ebenfalls das *include_cpp!*-Makro enthalten, indem die zu bindenden C++-Dateien angegebene werden, sowie Whitelist-Einstellungen.

Listing 6.1: C++ - Autocxx Beispiel

```
// example.hpp
inline int meaning() { return 42; }
```

Listing 6.2: Rust - Autocxx Beispiel

```
use autocxx::prelude::*;

include_cpp! {
    #include "example.hpp"
```

```
    safety!(unsafe)
    generate!("meaning")
}

pub fn main() {
    meaning();
}
```

Autocxx verarbeitet die C++-Header zu einer Cxx-Bridge-Definition.

Listing 6.3: C++ - Autocxx Beispiel - Generierter Cxx-Bridge

```
//versimpelt und formatiert
mod ffi {
    #[cxx::bridge]
    mod cxxbridge {
        unsafe extern "C++" {
            pub fn meaning() -> c_int;
            type c_int = autocxx :: c_int;
            include ! ("header.hpp");
        }
        extern "Rust" {
        }
    }
    pub use cxxbridge::meaning;
}
```

Cxx wiederum nutzt dann Wrapper-Methoden um die Funktionsaufrufe in C++ durch zu führen. Dies erlaubt sogar das Binden von *inline*-Methoden:

Listing 6.4: C++ - Autocxx Beispiel - Cxx-Bridge Handler

```
//gen0.cxx
void cxxbridge1$meaning(::c_int *return$) noexcept {
    ::c_int (*meaning$)() = ::meaning;
    new (return$) ::c_int(meaning$());
}
```

6.2 Kontribution zu Autocxx

Zum Beitragen von Verbesserungen am Projekt muss eine CLA[1] genutzt werden, dann können mit dieser Pull-Requests eingereicht werden. [13]

Das Autocxx-Projekt hat folgendene Verzeichnisse die für Modifizierung besonders relevant sind:

1. parser - Verarbeitet das *include_ cpp!*-Macro
2. engine - Code Generation
3. macro - Enthält die Definition des *include_ cpp!*-Macros
4. src - Enthält den Autocxx-Prelude

Buildprozess

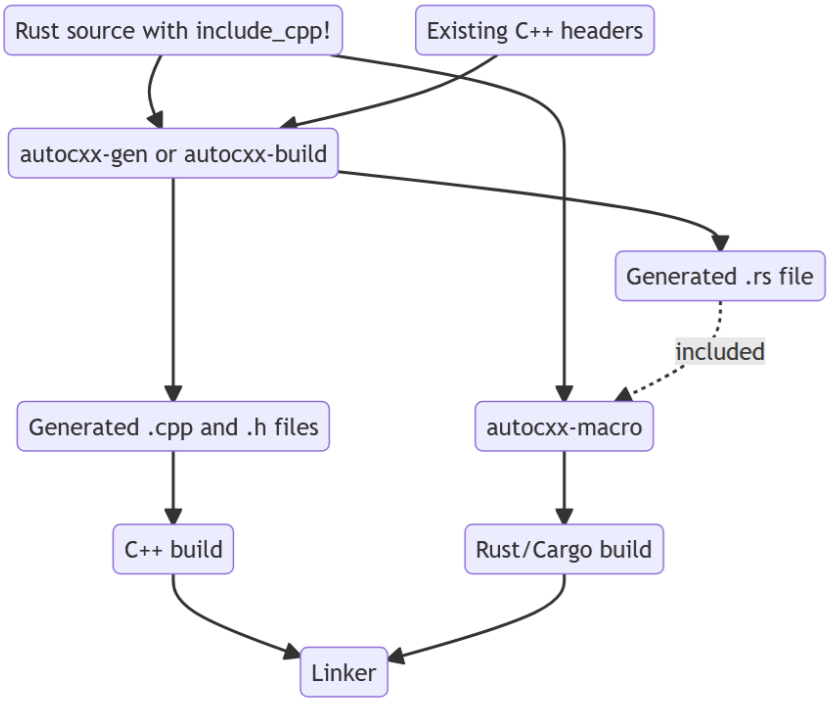

Abbildung 6.1: Autocxx Build-Prozess [12]

[1] Eng.: Contributors license agreement

Das Verzeichniss gen/build wird durch das Buildscript genutzt um die Codegeneration durchzuführen. Der Buildprozess (Abbildung 6.1) läuft folgendermaßen ab:

1. Buildscript ruft *autocxx_build::build()* auf
2. autocxx_build parsed das *include_cpp!-Macro*
3. autocxx_build verarbeitet C++-Headerdateien
4. autocxx_build nutzt die engine-Crate um C++-Hilfscode und Rust-Bindings zu generieren
5. Das *include_cpp!*-Macro includiert die Bindings
6. Der Linker verknüpft die Bindings mit dem C++-Hilfscode und dem C++-Code

Somit ist insbesondere die engine-Crate interessant für Modifikation.

Sie enthält folgende Module:

1. conversion
2. conversion/parse
3. conversion/analysis
4. conversion/codegen_cpp
5. conversion/codegen_rs

Die engine-Crate nutzt zuerst eine modifizierte Version von Bindgen um die C++-Headerdateien zu analysieren. Dann wird der AST von dem parse-Modul zu einem *ApiVec<NullPhase>* verarbeitet. Der ApiVec wird dann durch das analysis-Modul in folgender Reihenfolge durch mehrere Phasen verarbeitet:

1. *ApiVec<NullPhase>*
2. *ApiVec<TypedefPhase>*
3. *ApiVec<PodPhase>*
4. *ApiVec<FnPrePhase2>*
5. *ApiVec<FnPhase>*

Dann wird der *ApiVec* zuletzt durch den *RsCodeGenerator* zu einem Rust AST und den *CppCodeGenerator* zu einem C++-Header- und Sourcepaar verarbeitet. Dies geschiet in dem codegen_rs- und codegen_cpp-Modulen.

6.3 Beispielimplementation Subtyp-Polymorphismus

Obwohl Autocxx volle Unterstützung für Polymorphismus von virtuellen Methoden hat, gibt es tatsächlich 3 weitere Formen von Polymorphismus:

1. Ad-hoc-Polymorphismus (Funktionsüberladungen)
2. Parametrischer-Polymorphismus (Templates in C++)
3. Subtyp-Polymorphismus, welches Subklassen syntaktisch ihren Baseklassen entsprechen lassen

Autocxx bietet keinerlei Unterstützung für Subtyp-Polymorphismus: Referenzparameter von Baseklassen werden nicht akzeptiert. Nicht-virtuelle Methoden von Baseklassen werden in Autoxcxx ebenfalls nicht vererbt. Dies ist im Grunde auch ein Fall von Subtyp-Polymorphismus auf dem *this*-Pointer.

Die naive-Lösung die Klassen *DerefMut<Target = Baseclass>* implementieren zu lassen ist nicht möglich, da die nicht Konstanten-Methoden und Referenzparameter *Pin<&mut T>* als *self* konsumieren.

Pin<T> ist eine generische Wrapperklasse, die eine Referenz auf ein Objekt darstellt, welches sich nicht im Speicher bewegen darf. Der generische Parameter *T* muss dabei dereferenzierbar sein, d.h. *Deref* implementieren.

Das *DerefMut*-Trait erlaubt es hingegen nur Referenzen zu produzieren. Zudem ist das Deref trait nur einmal pro Typ implementierbar und ist somit ungeeignet um Mehrfachvererbung zu unterstützten.

Außerdem ist es aufgrund von Multiinheritance der Cast zur Baseklasse nicht trivial, da der Basepointer des Subtyps nicht zwangsläufig equivalent ist.

Autocxx leistet jedoch bereits Vorarbeit für die Casts und implementiert *AsRef<Base>* für alle Kinderklassen von *Base*. Dies lässt sich ausnutzen und man kann einen Trait definieren, der diesen Cast für nicht-konstante Referenzen durchführt.

Listing 6.5: Rust - Subclass Polymorphismus - ToBaseClass

```
impl<Base: AsRef<Base>> ToBaseClass<Base> for Child {
    fn as_base_mut(&mut self)-> Pin<&mut Base> {
        #[allow(mutable_transmutes)]
        unsafe {
         Pin::new_unchecked(transmute(self.as_ref()) )
      }
    }
}
```

transmute konvertiert Typen der gleichen Größe und wird hier genutzt, um aus der *&Base* ein *&mut* Base zu machen. Transmutes mit unterschiedlicher Mutabilität sind standartmäßig nicht erlaubt, doch in diesem Fall ist es anzunehmen, dass es nicht zu ungewünschtem Verhalten kommt, da *self* ursprünglich ebenfalls eine nicht-konstante Referenz war. Eine bessere Lösung wäre es Autocxx zu erweitern und eine nicht-konstante Version von *AsRef* für die Klassen zu implementieren. Diese würde aus einem *Pin<&mut Child>* ein *Pin<&mut Base>* machen können.

Wenn man dann in dem codegen_rs-Modul *AsRef* ebenfalls für die generierten Typen selbst implementiert ist es möglich alle validen konstanten Parameter als *&impl AsRef<Base>* und alle nicht-konstanten als *&mut impl ToBaseClass<Base>* in Funktionsparamern auszudrücken.

Da die durch Autocxx instanziierten Objekte (z.B. *Pin<MoveRef<T»*) nicht *DerefMut<Target = T>* implementieren, sondern nur *Deref<Target = T>*, ist die Lösung allerdings noch nicht für die nicht-konstanten Referenzen ausreichend.

Da alle vier Arten Objekte zu instanziieren jedoch einen *Pin<T>* erzeugen, wo *T* zu *ToBaseClass* dereferenziert, ist es allerdings möglich alle zu nicht-konstanten Referenzen konvertierbaren Typen durch folgenden Trait zu beschreiben:

Listing 6.6: Rust - Subclass Polymorphismus - CppPinMutRef

```
impl<Base, Child, C: ToBaseClass<Base>> CppPinMutRef<Base>
for Pin<Child> where Child: DerefMut<Target = C> {
    fn as_cpp_mut<'c>(&'c mut self) -> Pin<&'c mut Base> {
        let ptr = self.as_mut();
        let raw_ptr = unsafe { ptr.get_unchecked_mut() };
        unsafe { transmute(raw_ptr.as_base_mut()) }
```

```
    }
}
```

Zur Vereinheitlichung ist ein Trait für konstant referenzierbare Typen sinnvoll:

Listing 6.7: Rust - Subclass Polymorphismus - CppCoinstRef

```
impl<Base, Child, C: AsRef<Base>> CppConstRef<Base>
for Child where Child: Deref<Target = C>{
    fn as_cpp_ref<'c>(&'c self) -> &'c Base {
        unsafe { transmute(self.deref().as_ref()) }
    }
}
```

Nun ist es möglich in der codegen_rs-Phase jegliche Funktionen und Methoden wo dies notwendig ist durch einen Wrapper zu ersetzen. Dieser Wrapper ersetzt die *&T* mit *&impl CppConstRef<T>* sowie *Pin<&mut T>* mit *&mut impl CppPinMutRef<T>*. Dann kann man die ursprüngliche Funktion / Methode verstecken und durch einen weiteren Wrapper aufrufen. Hierfür ist es notendig jegliche der ersetzen Parameter mit *.as_cpp_ref()* bzw. *.as_cpp_mut()* zu versehen.

Dies hat außerdem den Vorteil, dass die Ergonomie von Referenzparametern und Methoden erhöht ist, da man nicht die unterschiedlichen Methoden nutzen muss um einen *Pin<&mut T>* zu erhalten. Stattdessen kann man mit dieser Modifikation die Methoden alle unabhängig von Speichermedium direkt nutzten und ebenfalls die Objekte nach Erstellung direkt als Borrow bzw. nicht-konstanten Borrow als Parameter nutzten.

Listing 6.8: Rust - Subclass Polymorphismus - Wrapperfunktion

```
//Methodenname vereinfacht
pub fn call_inherit_me_autocxx_wrapper_poly
   (a: &impl CppConstRef<A>) -> c_int {
   cxxbridge::call_inherit_me_autocxx_wrapper(a.as_cpp_ref())
}
```

Bei dem *self*-Parameter von Methoden ist dies allerdings nicht so einfach, da impl-Trait-Typen nicht als self agieren können. Folglich ist es notwendig für Klassen, die Methoden vererben, die Methoden einzelnd zu implementieren.

Eine andere mögliche Lösung, die hier angewandt wurde, ist es für konstante und nicht-konstante Methoden jeweils ein Extension-Trait zu nutzen. Diese sind dann mit den gleichen Traitbounds wie *CppConstRef* bzw. *CppPinMutRef* auf *self* implementierbar.

Listing 6.9: Rust - Subclass Polymorphismus - Methoden-Extension-Trait

```
impl<Child, C: ToBaseClass<A>> MethodsAMut<C> for Pin<Child>
where Child: DerefMut<Target = C> {
    fn inherit_mut(&mut self, poly: &mut impl CppPinMutRef<A>)
        -> autocxx :: c_int {
        let ptr = self . as_mut ();
        let raw_ptr = unsafe { ptr . get_unchecked_mut () };
        inherit_mut_autocxx_wrapper(
            raw_ptr.as_base_mut(), poly.as_cpp_mut()
        )
    }
}
```

Der unsafe-Code entfernt den Pin mit *get_unchecked_mut*. Dann wird mit *as_base_mut* genutzt, um die Subtyp-Refrenz zu erhalten. In dem Fall des konstanten Parameters wird *as_cpp_mut* genutzt. Diese Parameter werden dann an die ursprüngliche Methode übergeben.

6.3.1 Implementierung

Zur Implementierung dieser Strategie wurde die codegen_rs-Phase gewählt. Diese konvertiert den *VecApi<FnPhase>* zuerst in einen Vector aus *RsCodegenResult* und dessen qualifizierten Namen. Dies ist der letzte Schritt bevor es zum Rust-AST konvertiert wird. Hier wurde deswegen eine Methode *subtype_polymorphism* eingefügt, die den *RsCodegenResult* Vector wie spezifiziert modifiziert. [2]

Listing 6.10: Rust - Subclass Polymorphismus - subtype_polymorphism-Funktion

```
//Leicht vereinfacht

type QN = QualifiedNamespace;
```

[2] Die folgende Implementierung befindet sich unter /ch7 und https://github.com/jasper310899/autocxx/tree/autoxcxx-polymorphism

```
pub(crate) fn subtype_polymorphism
    (rs_codegen_results: &mut Vec<(QN, RsCodegenResult)>) {
     let mut impls_to_trait = RequiredPolyMethods::default();

     for results in &mut *rs_codegen_results {
       polymorphise_impls(&mut impls_to_trait, results);
       polymorphise_cxx(&mut impls_to_trait, results);
     }

     make_method_traits(
       rs_codegen_results,
       &mut impls_to_trait
    );
}
```

Die Funktion iteriert durch die *RsCodegenResult*-Objekte und modifiziert zuerst jegliche in impl-Blöcken definierten Methoden und dann jegliche Funktionen. In *impls_to_trait* werden dabei Methoden gesammelt, die in der letzten Phase *make_method_traits* als Extension-Methods implementiert werden. Diese müssen dafür in konstante und nichtkonstante Methoden gruppiert werden. Außerdem wird in der letzten Phase nicht *AsRef* für alle Typen generiert, damit sie selbst und nicht nur Subtypen als Parameter genutzt werden können.

Ein bemerkenswerter Sonderfall sind virtuelle Methoden; Autocxx implementiert diese bereits für Subklassen und folglich kommt es zu Fehlern aufgrund von ambivalenten Methodendefinitionen. Deswegen wurde in die Struktur für die Impl-Blöcke in den RsCodegenResults ein Feld aufgenommen, dass angibt, ob die Methode virtuell ist. Dann wird diese Methode für Subtyp-Polymorphismus übersprungen. Dies hat den Nachteil, dass Subtyp-Polymorphismus auch nicht für die Parameter von virtuellen Methoden funktioniert.

Um die Qualität der Implementierung zu sichern wurden die Integrationstests von Autocxx genutzt, um Regressionsbugs zu identifizieren. Dieser Prozess wurde zwei mal wiederholt und es sind inzwischen ca. 75% der Tests erfolgreich. Vor einer Kontribution müssten diese alle beheben werden.

Außerdem wurde der Code mit Rustfmt standartisiert formatiert und alle Funktionen wurden mit Kommentaren dokumentiert.

7 Konklusion

In dieser Arbeit wurde die Einbindbarkeit von C++-Interfaces in Rust untersucht. Das zentrale Problem bestand in der Herausforderung, die Interoperabilität zwischen diesen beiden Programmiersprachen zu gewährleisten, um C++-Bibliotheken effizient und sicher in Rust verwenden zu können. Angesichts der zunehmenden Migration von Softwarekomponenten zu Rust aufgrund seiner Speichersicherheitsgarantien ist es sinnvoll, die vorhandenen Tools zur Generierung von Bindings zu verbessern. Da es bisher keine Wissenschaftliche Evaluation von den vorhandenen Tools gab, wurde dies mittels Unit-Tests pro C++-Syntaxelement durchgeführt.

Die Analyse hat gezeigt, dass bestehende Binding-Generatoren wie Autocxx und Bindgen zwar vielversprechend sind, aber in Bezug auf die vollständige Unterstützung komplexer C++-Features wie z.B. Templates, Vererbung und Exceptions noch Defizite aufweisen.

Insbesondere wurde deutlich, dass Autocxx, obwohl es die Nutzung der Bindings einschränkt deutlich höhere Sicherheit bietet. Bindgen hingegen überlässt dem Programmierer mehr Möglichkeiten die Invarianten der C++-Interfaces zu brechen. Deswegen wurde examplarisch eine Erweiterung von Autocxx durchgeführt, um Subtyp-Polymorphismus zu unterstützten.

Für die Zukunft sollten mehr Verbesserungen an Autocxx durchgeführt werden, da es die Verlässlichkeit von Rust weniger komprimitiert. Insbesondere sollten grundlegende Features von C++, die noch fehlen verfügbar gemacht werden, wie u.A.:

1. (Mutable) globals (static / extern)
2. Generische Funkionen
3. Deprecated / Nodiscard
4. Standartlibrary-Support ausweiten
5. Bitfelder

6. Default-Funktionsparamerter
7. Vararg-Funktionen
8. Umwandlungsfunktionen
9. Const-Generics
10. Namespace- / Using-Verbesserungen
11. Ausnahmenbehandlung

Schließlich könnte eine vertiefte Evaluierung alternativer Ansätze, wie z.B. die Verwendung von Sourcecode-Transpilern oder Inline-C++-Makros, zusätzlich sinnvoll sein.

Literaturverzeichnis

[1] Gregory, Jason: *Game Engine Architecture*. Taylor & Francis, 2018. – ISBN 1138035459

[2] Langlands, Anders ; Titley, Luke ; Nelson, Owen: Rust for Visual Effects. In: *Proceedings of the 2021 Digital Production Symposium*. New York, NY, USA : Association for Computing Machinery, 2021 (DigiPro '21). – URL https://doi.org/10.1145/3469095.3469275. – ISBN 9781450385954

[3] Martins, Bruno: Benefits and Drawbacks of Using Rust in an Existing C/C++ Codebase. In: *17th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2020, S. WECPR02

[4] Ng, Vincent: *Rust vs C++, a Battle of Speed and Efficiency*. 05 2023

[5] *C++ - Standards*. – URL https://www.open-std.org/jtc1/sc22/wg21/docs/standards. – Zugegriffen: 31.08.2024

[6] *Itanium C++ ABI: Exception Handling (Revision : 1.22)*. – URL https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html. – Zugegriffen: 31.08.2024

[7] Stroustrup, Bjarne: *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 2013. – ISBN 0275967301

[8] Sudwoj, Michal: *Rust programming language in the high-performance computing environment*. ETH Zurich, 2020. – URL https://www.research-collection.ethz.ch/handle/20.500.11850/474922

[9] Tronge, Jake ; Pritchard, Howard: Embedding Rust within Open MPI. In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. New York, NY, USA : Association for Computing Machinery, 2023 (SC-W '23), S. 438–447. – URL https://doi.org/10.1145/3624062.3624112. – ISBN 9798400707858

[10] Urban, Marius: *Konzeptionierung, Implementierung und Validierung einer Rust basierten Software-Bibliothek zum Management von Zertifikaten in Feldbusgeräten*, Hochschule Offenburg, Bachelor's Thesis, 2023. – URL https://opus.hs-offenburg.de/6897

[11] *1210-impl-specialization - The Rust RFC Book*. – URL https://rust-lang.github.io/rfcs/1210-impl-specialization.html. – Zugegriffen: 31.08.2024

[12] *Building - Autocxx Manual*. – URL https://google.github.io/autocxx/building.html. – Zugegriffen: 31.08.2024

[13] *Contributing - Autocxx Manual*. – URL https://google.github.io/autocxx/contributing.html. – Zugegriffen: 31.08.2024

[14] *cppreference.com - Attribute specifier sequence(since C++11)*. – URL https://en.cppreference.com/w/cpp/language/attributes. – Zugegriffen: 31.08.2024

[15] *cppreference.com - Bit-field*. – URL https://en.cppreference.com/w/cpp/language/bit_field. – Zugegriffen: 31.08.2024

[16] *cppreference.com - consteval specifier*. – URL https://en.cppreference.com/w/cpp/language/consteval. – Zugegriffen: 31.08.2024

[17] *cppreference.com - constexpr specifier*. – URL https://en.cppreference.com/w/cpp/language/constexpr. – Zugegriffen: 31.08.2024

[18] *cppreference.com - constinit specifier*. – URL https://en.cppreference.com/w/cpp/language/constinit. – Zugegriffen: 31.08.2024

[19] *cppreference.com - Declarations*. – URL https://en.cppreference.com/w/cpp/language/declarations. – Zugegriffen: 31.08.2024

[20] *cppreference.com - Dynamic exception specification*. – URL https://en.cppreference.com/w/cpp/language/except_spec. – Zugegriffen: 31.08.2024

[21] *Generating Bindings to C++ - The bindgen User Guide*. – URL https://rust-lang.github.io/rust-bindgen/cpp.html. – Zugegriffen: 31.08.2024

[22] *GitHub - dtolnay/cxx*. – URL https://github.com/dtolnay/cxx. – Zugegriffen: 31.08.2024

[23] *GitHub - google/autocxx*. – URL https://github.com/google/autocxx. – Zugegriffen: 31.08.2024

[24] *GitHub - immunant/c2rust*. – URL https://github.com/immunant/c2rust. – Zugegriffen: 31.08.2024

[25] *GitHub - mystor/rust-cpp*. – URL https://github.com/mystor/rust-cpp. – Zugegriffen: 31.08.2024

[26] *GitHub - NishanthSpShetty/crust*. – URL https://github.com/NishanthSpShetty/crust. – Zugegriffen: 31.08.2024

[27] *GitHub - rust-lang/cc-rs*. – URL https://github.com/rust-lang/cc-rs. – Zugegriffen: 31.08.2024

[28] *GitHub - vfx-rs/cppmm*. – URL https://github.com/vfx-rs/cppmm. – Zugegriffen: 31.08.2024

[29] *GitHub.com - Repository search results*. – URL https://github.com/search?q=language%3AC%2B%2B&type=repositories&ref=advsearch. – Zugegriffen: 31.08.2024

[30] *A proactive approach to more secure code | MSRC Blog*. – URL https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. – Zugegriffen: 31.08.2024

[31] *Rust Compiler Development Guide - Monomorphization*. – URL https://rustc-dev-guide.rust-lang.org/backend/monomorph.html#collection. – Zugegriffen: 31.08.2024

[32] *Standard library developers Guide - Specialization*. – URL https://std-dev-guide.rust-lang.org/policy/specialization.html. – Zugegriffen: 31.08.2024

[33] *The Top Programming Languages 2024*. – URL https://spectrum.ieee.org/top-programming-languages-2024. – Zugegriffen: 31.08.2024

A Anhang

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original