

# The MYLL Programming Language: C++ Resyntaxed and Extended

Eine Weiterentwicklung der C Style Syntax

zur Erlangung des Grades

## Master of Science Informatik

an der

## Hochschule Niederrhein

Fachbereich Elektrotechnik und Informatik

vorgelegt von

## Jan Helge Reitz

Geboren 1978-10-06 in Moers, Deutschland

Matrikelnummer: 756470

Datum: 2020-07-03

Prüfer: Prof. Dr. \*\*\*\*

Zweitprüfer: Prof. Dr. \*\*\*\*

# Zusammenfassung

Ich schlage eine neue Syntax für C++ vor, welche den Umgang mit C++ unproblematischer und den Einstieg erleichtern soll.

Die von mir vorgeschlagene neue Syntax für C++ hat den Codenamen **Myll** (*frei von My Lang*). Der Fokus des Sprach-Designs balanciert zwischen zwei grundlegenden Zielen; zum einen soll er alteingesessene C++ Programmierer abholen und dessen Arbeit in Zukunft leichter gestalten und zum anderen soll es Neulingen den Einstieg, in eine immer komplizierter werdende Sprache, erleichtern.

Um dieses Ziel zu erreichen habe ich viele Sprachkonstrukte auf die Probe gestellt und mir überlegt wie man diese besser gestalten kann, teils von anderen Sprachen inspiriert, aber auch komplett selbst erdacht.

Des Weiteren soll Myll einfach ergänzbar sein: Beispielsweise nutzen QT, UE4 und viele andere Bibliotheken *Makros*, welche an den richtigen Stellen in den Code eingesetzt werden und teilweise durch einen zusätzlichen Präprozessor ausgeführt werden. Diese verstreuten Makros können in Myll etwa durch ein Klassen- oder Variablen-Attribut ersetzt werden.

Durch die Nutzung von Myll ließen sich auch eigene Warnings/Errors im Falle der Nutzung vom Projekt verpönten/untersagten Sprachkonstrukten einbauen.

Umgesetzt ist Myll als ein Language to Language Compiler, auch Transpiler genannt. Dieser erzeugt als Ausgabe keinen Maschinencode, sondern Quellcode einer anderen Programmiersprache, in diesem Fall C++.

Betrachtet wird die Umsetzbarkeit dieses Unterfangens; Lässt sich eine ähnlich komplexe Programmiersprache wie C++ überhaupt handhaben? Welche Aspekte von C++ sind problematisch und lösbar? Sind ähnlich positive Aspekte zu erzielen wie es Transpilation bei Webentwicklungen hervorgebracht hat?

??? erklären ???

Es soll auch gezeigt werden das durch diesen Zwischenschritt, viele positive Merkmale anderer Sprachen einfach verfügbar gemacht werden können.

??? Kürzen ???

# Abstract

I propose a new syntax for C++, which should make the general usage of C++ less problematic and also make it easier to get started.

The new syntax for C++ proposed by me has the code name **Myll** (*free from My Lang*). The focus of the language design balances between two basic goals; on the one hand, it should pick up old-established C++ programmers and make their work easier in the future, and on the other hand, it should make it easier for newcomers to get started with a language which has become more and more complicated.

In order to achieve this goal, I have put many language constructs to the test and thought about how they could be better designed, partly inspired by other languages, but also completely conceived by myself.

Furthermore Myll should be easy to extend: For example QT, UE4 and many other libraries use macros which are inserted at the right places in the code and are partially executed by an additional preprocessor. These scattered macros can be replaced by an attribute on a type or variable in Myll .

...???

# Inhaltsverzeichnis

<b>Zusammenfassung // Abstract.....</b>	<b>2</b>
<b>Abstract.....</b>	<b>2</b>
<b>Inhaltsverzeichnis.....</b>	<b>3</b>
<b>1 Einleitung.....</b>	<b>5</b>
1.1 Definitionen.....	5
1.1.1 Formatierung dieses Dokumentes.....	5
1.1.2 Begriffsdefinitionen.....	5
1.2 Idee / Motivation.....	7
1.2.1 Alteingesessene C++ Programmierer abholen.....	7
1.2.2 Neulingen den Einstieg erleichtern.....	7
1.3 Grundlagen.....	8
1.3.1 Was macht C++ besonders?.....	8
1.3.2 Wieso noch eine neue Programmiersprache?.....	8
<b>2 Analyse.....</b>	<b>10</b>
2.1 Vergleich von C++ mit anderen Sprachen.....	10
2.1.1 Vergleich mit C.....	10
2.1.2 Vergleich mit C# und Java.....	10
2.1.3 Vergleich mit D.....	11
2.1.4 Vergleich mit Rust.....	11
2.1.5 Vergleich mit Ruby.....	11
2.2 Problematische Aspekte von C++.....	13
2.2.1 Deklarationen, Definitionen und Initialisierung.....	13
2.2.2 Fehleranfällige Variablen & Typen Deklarationen.....	14
2.2.3 Schlechtes Standardverhalten.....	16
2.2.4 Inkonsistente Syntax.....	17
2.2.5 Problematische Reihenfolge der Syntax.....	19
2.2.6 Schwer durchschaubare automatische Verhaltensweisen.....	20
2.2.7 Schlechte Namensgebung.....	21
2.2.8 Verbose Schreibweise.....	22
2.2.9 Keyword reusage.....	22
2.2.10 East Const und West Const.....	22
2.2.11 WOANDERS HIN:???	22
<b>3 Konzept.....</b>	<b>23</b>
3.1 Design der Syntax.....	23
3.1.1 Grundsätze der Sprache Myll.....	23
3.1.2 Neue Keywords.....	24
3.2 Behandlung der Probleme.....	24
3.2.1 Deklarationen, Definitionen und Initialisierung.....	24
3.2.2 Fehleranfällige Variablen & Typen Deklarationen.....	25
3.2.3 Schlechtes Standardverhalten.....	27
3.2.4 Inkonsistente Syntax.....	28
3.2.5 Problematische Reihenfolge der Syntax.....	28
3.3 Neue Features welche keine Problemfälle waren.....	29

3.3.1 Benamte Parameter.....	29
<b>4 Implementierung.....</b>	<b>31</b>
4.1 Umsetzung.....	31
4.2 C++ predigen, aber C# und Java trinken.....	31
4.3 Alternative Umsetzungsmöglichkeiten.....	31
4.4 Details.....	32
4.4.1 Lexer / lexikalischer Scanner / Tokenizer.....	32
4.4.2 Parser / syntaktische Analyse / Zerteiler.....	33
4.4.3 Traversal / Durchlauf.....	33
4.4.4 Semantische Analyse / Resolve Symbols.....	33
4.4.5 Generator.....	34
<b>5 Auswertung &amp; Zusammenfassung.....</b>	<b>36</b>
5.1 Das sieht ja gar nicht mehr wie C++ aus!.....	36
5.2 Performance Benchmark.....	37
<b>6 Ausblick.....</b>	<b>39</b>
6.1 Intelligendere Parameter Übername.....	39
6.2 Ranges.....	40
6.3 Rest.....	40

# 1 Einleitung

## 1.1 Definitionen

### 1.1.1 Formatierung dieses Dokumentes

Normaler Text mit einem *"Eingebetteten Zitat"* und dann weiter mit etwas höchst **Wichtigem**.

*"Dies ist ein Blockzitat  
Von einer im Text genannten Person"*

Exemplarisch eine Codedatei mit dem Namen *myfile.cpp*:

```
// Dies hier ist Code
int a;
float b;
```

Hier wird auf die Variablen *a* und *b* Bezug genommen und die Typen *int* und *float* erwähnt, stets im Singular. Wenn Code im Fließtext geschrieben wird dann sieht es so aus `int a[] = {1,2,3};`.

### 1.1.2 Begriffsdefinitionen

In jedem Dokument, welches über mehrere Programmiersprachen schreibt, müssen zunächst Mehrdeutigkeiten und Kollisionen ausgeräumt werden. Besonders Erwähnenswert sind die Unterschiede zur Namensgebung in C++, auf welche sich Myll ja stützt.

In dieser Arbeit werden oft die englischen Varianten von Begriffen aus der Informatik / Programmierung genutzt. ??? erläutern

**??? Wird am Ende Sortiert**

**GC - Garbage Collection**

???

**C++**

Wenn ich über C++ schreibe, dann meine ich sehr oft beides: die *Sprache C++* an sich und die *C++ Standard Library*. Diese sind, insbesondere seit C++11, sehr eng miteinander verbunden; so wurde die Sprache erweitert um bestimmte Funktionalität in der Bibliothek zu ermöglichen. Ebenso sind Komponenten der Bibliothek Notwendig um die Sprache vollständig zu nutzen.

Beispiele: `std::initializer_list`, `std::move`, `std::dynamic_cast`

**??? überarbeiten**

**Keyword - Schlüsselwort**

Wort mit spezieller Bedeutung innerhalb der Sprache welches nicht für eigene Namen verwendet werden kann.

**FhO - Funktion höherer Ordnung**

Beschrieben im Kontext einer Prozeduralen Programmiersprache:

Eine FhO ist eine Funktion, welche eine weitere Funktion (oder auch Lambda) und eine Kollektion als Parameter übergeben bekommt. Die übergebene Funktion wird auf jedes Element der Kollektion angewendet. Diese Kollektion kann auch durch einen Iterator repräsentiert werden.

**TMP - Template Metaprogramming**

Alternative Art der Programmierung bei der die Logik zur Kompilationszeit ausgewertet wird.

**Dynamisches Array**

Der `std::vector` aus der C++ Standard Library ist ein dynamisch wachsendes Array und wird in diesem Dokument *Dynamisches Array* oder *dyn\_array* genannt.

**Map & Reduce**

*Map* ist eine FhO, bei der pro Ursprungselement ein Zielelement erzeugt wird, welche wiederum in einer Kollektion gespeichert werden. In C++ ist dies durch `std::transform` implementiert.

Sie wird oft zusammen mit der FhO namens *Reduce* genutzt, welche aus allen Ursprungselementen ein einzelnes Zielelement erzeugt.

???MOVE ins Kapitel

In C++ ist dies als `std::accumulate`, neuerdings auch als `std::reduce` implementiert.

Ein simples *Map & Reduce* Beispiel in Ruby:

```
fortytwo = [4, 6, 14].map{|e| e / 2}.reduce{|accu,e| accu * e}
```

Die drei Zahlen im Array (4, 6, 14) werden zunächst in *map* jeweils durch 2 dividiert, dann in *reduce* in einen Akkumulator aufmultipliziert und ergeben die Zahl 42.

**Sorted Dictionary & Search Tree**

Ein sortiertes assoziatives Array welches in C++ als `std::map` bekannt ist.

**Unsorted Dictionary & Hash Table**

Ein unsortiertes assoziatives Array welches in C++ als `std::unordered_map` bekannt ist.

**Funktion & Prozedur**

Manche Programmiersprachen unterscheiden hart zwischen diesen zwei Unterprogramm-Typen. (Als *Unterprogramm* versteht sich hier nicht die Definition welche COBOL verwendet)

Zum Beispiel liefern in Pascal *Prozeduren* keine Rückgabewerte. Außerdem wird in vielen Sprachen in *Funktionen* das Resultat nur aus den Eingaben erzeugt und hat keine Nebeneffekte, dieser Spezialfall wird hier "*pure Funktion*" genannt. In C++ werden all diese Unterprogramme *Funktionen* genannt, da es dort aber kein Keyword dafür gibt, konnten Benutzer im Sprachgebrauch natürlich ihre eigene Namensgebung verwenden.

In diesem Dokument wird sich der Gepflogenheit von C++ bedient: Alles sind Funktionen und können, soweit nicht anders annotiert, Nebeneffekte haben und auch *void* zurückgeben.

{{{Myll führt Keywords für Funktionen ein und bietet beide Namensgebungen um beide Lager abholen. Diese können im fast synonym genutzt werden, analog zur Einführung von *class* zu *struct* in C++.}}}

**Methode**

Ist eine Funktion die mit einem Objekt verknüpft ist.

**UB - Undefined Behaviour - undefiniertes Verhalten**

Tritt auf wenn der Programmierer die Regeln der Sprache verletzt. Nach deren Auftritt ist es dem Programm erlaubt jedwedes Verhalten zu zeigen. Bei dieser Art der Fehlfunktion muss ein C++ Compiler weder die Kompilation abbrechen, noch eine Warnung ausgeben.

**Proto-Myll**

Enthält nur Teile der Myll-Syntax, welche im aktuell betrachteten Kontext Sinn machen.

**Konstrukte**

Alles innerhalb einer Programmiersprache: Deklarationen, Definitionen, Statements, Expressions, Attribute.

## Deklaration & Definition

Deklaration ist ???

Definition ist ???

## 1.2 Idee / Motivation

Die Idee zur Entwicklung dieser Sprache kam mir ursprünglich durch ein schon etwas älteres Dokument namens 'A Modest Proposal: C++ Resyntaxed' von Werther und Conway. In Ihrem Dokument bemängeln sie zurecht einige Unschönheiten in der Grammatik von C++, welche zum Teil der Kompatibilität zu C geschuldet ist. Ihre syntaktische Reimaginierung von C++ nennen sie **SPECS**.

Auch der Erfinder von C++, Bjarne Stroustrup, glaubt

*"within C++ there is a much smaller and cleaner language struggling to get out"*

was er in 'The Design and Evolution of C++' schrieb, welches auch SPECS zitierte.

(:Neu formulieren:)

Diese Meinung teile ich, denn wenn ich beim schreiben von C++ Code nachdenke wie ich es umsetze, dann habe ich wesentlich kompaktere Gedanken, als nachher die Umsetzung aussieht.

Zwar hat modernes C++ (11/14/17) frischen Wind in die Sprache gebracht, aber trotzdem fast keine Zöpfe abgeschnitten. Auch wurden neue Sprachkonstrukte eingeführt welche teils syntaktisch unschön und verbos sind, dem geschuldet das alter Code durch diese Ergänzungen nicht aufhören darf zu Funktionieren.

Kleine Beispiele der neuen Features.!!!

Retrospektiv betrachtet hat die in SPECS vorgeschlagene Syntax nicht mehr viel mit C++ zu tun und dessen stark von Pascal angehauchte Syntax liegt fern von meinem Ästhetikempfinden. Mein Lebensweg als Programmierer ist von C und C++ ähnlichen Sprachen geprägt weshalb die hier vorgestellte Lösung weit von der von SPECS abweicht.

### 1.2.1 Alteingesessene C++ Programmierer abholen

Alteingesessene C++ Programmierer abholen und dessen Arbeit in Zukunft leichter zu gestalten.

- Don't fix what's not broken
  - Welche Aspekte sind bewahrenswert
- Provide an easy exit path, no time invested is ever wasted
  - Generierter Code soll einmalig übernehmbar und lesbar sein
- Close to modern C++
  - Semantisch wie C++
  - Syntax entschlackt
- Make features available earlier than the C++ Standard Committee does
  - Zuerst indem man sie emuliert (siehe Polyfills), in Zukunft indem man sie einfach durchreicht.

### 1.2.2 Neulingen den Einstieg erleichtern

Neulingen den Einstieg in eine immer komplizierter werdende Sprache zu erleichtern.

- Unify Code in one file
  - Hilft bei DRY
  - Was gehört in welche Datei?

- Keine .cpp, .h, .impl.h, .inl.h Dateien mehr nötig
- Fix broken defaults ???
  - Implicit constructor, private inheritance
- Fix cumbersome constructs
  - Includes
- Fix common pitfalls
- Provide convenience known from other contemporary languages

!!!Detaillierter beschreiben was jedes Einzelne bedeutet.

Diese Ziele stehen sich natürlich auch, teils extrem, entgegen. Ich versuche einen gangbaren Mittelweg zu finden und Entscheide im Zweifel meist für die zukünftige Wartbarkeit und gegen die Vergangenheit.

## 1.3 Grundlagen

### 1.3.1 Was macht C++ besonders?

Es gibt sehr viele neue und moderne Programmiersprachen die vielfältige Einsatzgebiete haben, einfacher und sicherer zu benutzen sind und in Teilbereichen performanter als C++ sind. Trotzdem weigert sich C++ (weiter???) an Popularität einzubüßen. **??? an Relevanz zu verlieren**

**??? was ich sagen will ist: Ja, es ging mal begab, aber dieser fallende Trend setzt sich nicht fort**

Einsatzgebiete in denen häufig C++ für die Umsetzung verwendet wird:

- Betriebssysteme
- Treiber
- Embedded Computing
- Computerspiele
- Simulationen
- Statistik

In diesen Bereichen wird C++ entweder aufgrund seiner guten Performance oder wegen der geringen Größe der Laufzeitumgebung und des Kompilats eingesetzt. C++ war früher auch als Entwicklungssprache für Desktopanwendungen verbreitet, wurde in diesen Bereichen aber in weiten Teilen von Java, C# aber auch von Webanwendungen abgelöst.

Wieso hält sich C++ in den genannten Bereichen so hartnäckig?

- Maximale Kontrolle über den Speicher
- Keine erzwungene Garbage Collection (GC)
- Keine / schlanke Laufzeitumgebung
- Native Binärdateien
- Nichts bezahlen, was nicht genutzt wird  
(abschaltbar: RTTI, Exceptions, Bounds/Overflow checks, etc)
- Keine erzwungene virtuelle Vererbung
- Mehrere Jahrzehnte alter C und C++ Code läuft fast immer ohne Anpassung

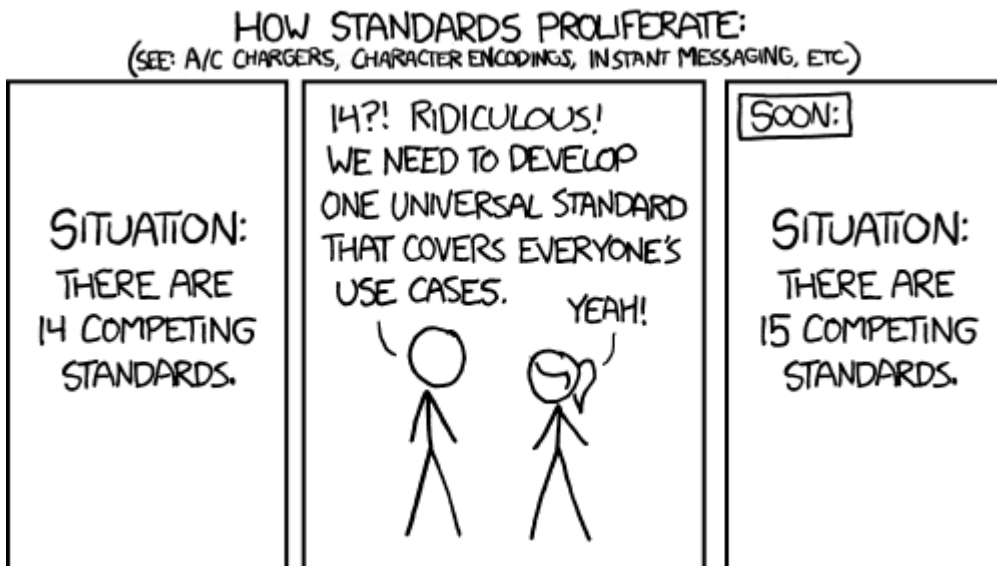


### 1.3.2 Wieso noch eine neue Programmiersprache?

*Warum eine neue Programmiersprache wenn C++ doch seine Bereiche so erfolgreich bedient?*

Weil es sich mit vielen der anderen hier betrachteten Sprachen (siehe 2.1) einfacher entwickeln lässt als mit C++.

Trotzdem wären viele Sprachkonstrukte, sobald man sie auf Sprachebene und nicht nur auf Bibliotheksebene angehen kann, eigentlich leicht übertragbar auf C++.



<https://xkcd.com/927/>



## 2 Analyse

### 2.1 Vergleich von C++ mit anderen Sprachen

Als Kompilationsziel von Myll soll zwar C++ dienen, jedoch mag sich Myll zum einen Syntaktisch und zum anderen von den Features anderer Programmiersprachen inspirieren lassen. Aus diesem Grund sei hier C++ mit einigen ausgewählten Programmiersprachen verglichen.

#### 2.1.1 Vergleich mit C

Der Vorgänger von C++.

- + Viel Kontrolle, man sieht genau was passiert
- + Viel simpler als C++
- + Sehr viel Freiheiten, aber auch Verantwortung bei der Speicherverwaltung
  
- Schlechte Encapsulation / Keine Native OOP Unterstützung
- Unübersichtlich bei großen Projekten, denn ohne Namespaces landet alles im globalen Raum
- Es ist der Ursprung der meisten hier betrachteten Syntaktischen Schwächen
- Mehr LOC für die Lösung des gleichen Problems

Hier ist ein Beispiel welches Aufzeigt was ich mit *viel Kontrolle* und *viel simpler* meine. Es ist Code welcher so in C, als auch in C++ auftreten kann. Die Kommentare beschreiben was in C++ zusätzlich zum offensichtlichen passieren kann.

```
{
    Foo f;           // kann in C++ einen potentiell teuren Konstruktor von Foo aufrufen
    A * a = f.get(); // kann in C++ eine Exception werfen und den folgenden Code überspringen
    a->update();      // kann in C++ eine virtuelle Methode aufrufen
}                   // kann in C++ einen potentiell teuren Destruktor von Foo aufrufen
```

Lerneffekt für Myll:

Keine Kompromisse in Bezug auf C's Syntax

#### 2.1.2 Vergleich mit C# und Java

C# und Java sind als designierte Nachfolger von C++ mit einer Transfusion dessen OOP Bluts entstanden. Sie haben sehr viel untereinander gemein und werden deshalb hier gemeinsam betrachtet. ??? Recherche

- + Es ist leichter fehlerfreien C# und Java Code zu schreiben
- + Fehlermeldungen in Generics/Templates beschreiben viel präziser potentielle Fehlerfälle
- + Einfach auf vielen Architekturen ausführbar (oft keine Neukompilierung nötig)
- + Weniger LOC für die Lösung des gleichen Problems
- + Viele äquivalente Features viel früher verfügbar als in C++ (concepts, auto, ranges, override, ...)
  
- Erzwungene GC, weniger Kontrolle über den Speicher
- Keine multiple Vererbung

- Nicht abschaltbare Bounds/Overflow checks, Reflection, Exceptions
- Ein Performance Defizit welches bei Desktopanwendungen verschmerzbar ist, jedoch bei den Performancekritischen Einsatzgebieten nicht. (Siehe: Was macht C++ besonders?)

Lerneffekt:

Weiter Featurevorsprung zu C++  
Entschlackte Syntax

### 2.1.3 Vergleich mit D

D hatte sich vorgenommen die Reihe der Sprachen BCPL  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  C++  $\rightarrow$  D fortzusetzen.

Leider konnte D nicht annähernd den Erfolg von C oder C++ erreichen, obwohl vieles von C++ gelernt und wohl möglich besser gemacht wurde.

- + Sehr nah an C++ angelehnt
- + Weniger LOC für die Lösung des gleichen Problems
- Erzwungene GC, weniger Kontrolle über den Speicher
- Keine multiple Vererbung

Lerneffekt für Myll:

Myll hat sich ein ähnliches Ziel wie D gesetzt; Nämlich ein besseres C++ zu sein.  
Myll verzichtet nicht (absichtlich) auf Kernfeatures von C++.

### 2.1.4 Vergleich mit Rust

Eine sehr neue Programmiersprache welche ihren Fokus auf die Sicherheit dessen was maschinell überprüfbar ist legt. Die statischen Überprüfungen welche der Rust Compiler vornimmt sind in der Ausprägung noch nicht verbreitet gewesen.

- + Stärkere statische Checks welche zu wesentlich sicherem Code führen sollen
- + In Zukunft ist Rust ein Contender um einige Bereiche welche C++ heute bedient
- (Anfänglich) schwerer zu schreiben durch Paradigmenwechsel des Lifetimemanagements
- Kompilationszeiten

Lerneffekt für Myll:

Eine leicht verbesserte Sicherheit wird dadurch erreicht dass manches Standardverhalten angepasst wird, welches einen vor Flüchtigkeitsfehlern beschützt.  
Eine Konkurrenz zu Rust in Sicherheitsfragen will Myll nicht werden.

### 2.1.5 Vergleich mit Ruby

Dieser Vergleich ist ein wenig weiter her geholt als die anderen betrachteten Sprachen. Ich wollte hier dennoch gern über den nahen Tellerrand hinausblicken um mich auch von weiter weg inspirieren zu lassen.

- + Unglaublich wenig Code ist nötig um simple Probleme zu lösen
- + Lambdas sind schon sehr lange ein gut integrierter und fast unsichtbarer Teil der Sprache

- Interpretierte Sprache (langsam in der Ausführung)
- Erzwungene GC, weniger Kontrolle über den Speicher
- Keine Typsicherheit

## 2.2 Problematische Aspekte von C++

Einige der problematischen Konstrukte in C++ sind der direkten Kompatibilität mit C geschuldet, andere sind von C++ so eingeführt worden.

Als Quelle dient zu Teilen das Buch 'The C Programming Language' von Kerninghan und Ritchie.

(Mehr Referenzieren!!!)

### 2.2.1 Deklarationen, Definitionen und Initialisierung

C++ verhält sich selbst in sehr simplen Beispielen wie eine Single-Pass parsende Programmiersprache und erfordert dadurch vom Programmierer ein hohes Maß an Aufmerksamkeit bei der Reihenfolge der Programmierung oder fordert die Wiederholung von bereits Niedergeschriebenem.

#### 2.2.1.1 Reihenfolge und Prototypen

Dieser Code kompiliert so nicht, er wirft den Fehler das a() nicht deklariert wurde.

```
int main() {
    a();
}
void a() {...}
```

Um diesen Fehler zu beseitigen müssen wir entweder den Code so umstellen das a() vor main() definiert wird oder wir fügen eine Prototypen-Deklaration für a() vor main() ein.

Im Falle eines alternierend rekursiven Funktionsaufrufs bleibt einem nur die Möglichkeit des Prototypen.

```
void partition(...);
void qsort(...) { ... partition(...); ... }
void partition(...) { ... qsort(...); ... }
```

Computer sind schnell genug diese Probleme für einen zu lösen. Lediglich der C als auch C++ Standard stehen der Unterstützung solch einfacher Inferenz??? im Wege.

#### 2.2.1.2 Aufteilung auf verschiedene Orte

Deklarationen, Definitionen und Initialisierung können entweder innerhalb einer Datei an verschiedenen Stellen und verschiedenen Schreibweisen vorkommen oder oft sogar über Dateigrenzen hinweg verteilt sein.

Dies macht den Code schwerer durchschaubar. Jegliche andere moderne Programmiersprachen erlaubt es einem (oder zwingen einen gar dazu) alles was z.B. zu einer Klasse gehört in einer einzelnen Datei abzulegen.

Dieses Beispiel soll einige Probleme aufzeigen:

Datei *myclass.h*:

```
class MyClass {
    int myVar;
    int myOtherVar;
    int myVar3 = 3;           // Funktioniert seit C++11
    static int myStatic;      // Funktioniert nicht wie myVar3
public:
    MyClass();
    int myProc();
    int myOtherProc() {...}   // Kann inlined werden, implizit weil inline implementiert
    inline int myFunc() const;
```

```

    virtual int myVirtual();
}
int MyClass::myFunc() const {...}    // Kann inlined werden, Keyword const wird wiederholt

```

Datei *myclass.cpp*:

```

#include <myclass.h>
int MyClass::myStatic = 1; // Keine Wiederholung des Keywords static, Typ int wiederholt
MyClass::MyClass() : myVar(1) {
    myOtherVar = 2;
}
int MyClass::myProc()    {...}    // Kann nicht inlined werden (nur mit whole program optimization)
int MyClass::myVirtual() {...}    // Keine Wiederholung des Keywords virtual

```

Bewertung:

- Manche Keywords müssen in Deklaration und Definition geschrieben werden, andere nicht
- Der Abschnitt "`MyClass::`" wird sehr oft wiederholt geschrieben
- Oft gibt es mehrere unterschiedliche Möglichkeiten das gleiche zu schreiben

### 2.2.1.3 Aufteilung im Kontext lebendiger Codebases

Die Verteilung auf mehrere Dateien ist besonders problematisch wenn ein C++ Projekt *lebt*, d.h. es immer wieder einschneidende Änderungen gibt. So muss im Beispiel das eine Methode *inline* fähig gemacht werden soll (selbst wenn die Logik unverändert bleibt) ihre komplette Implementierung aus der .cpp Datei ausgeschnitten und in die .h Datei eingefügt werden. Dadurch verliert man z.B. in seiner Versionsverwaltung den Bezug und kann nicht mehr einfach die Veränderungshistorie der Methode nachschlagen.

Das gleiche Problem tritt auf wenn eine Klasse templatisiert werden soll, nur muss in diesem Fall die komplette Implementierung in die Header Datei wandern.

## 2.2.2 Fehleranfällige Variablen & Typen Deklarationen

Wie findet die Deklaration von Variablen in C und C++ statt und welche Problematischen Situationen können dabei entstehen.

### 2.2.2.1 Ambiguität einer Deklaration

Sind die folgenden Statements Deklarationen oder nicht?

```

a * b;
c d;
c e();

```

Die Antwort darauf lautet: Es kommt drauf an.

Sollte *a* ein Typ sein, wird *b* als ein Pointer auf ein *a* Deklariert, ist *a* aber eine Variable wird `a.operator*(b)` aufgerufen. Der andere Fall ist zum einen *d*, welcher eine Variable vom Typ *c* deklariert und zum anderen *e* welcher grade nicht eine Variable vom Typ *d* deklariert die den Standardkonstruktor aufruft, sondern es ist eine Prototypen-Deklaration einer Funktion namens *e* welche ein *c* zurück gibt.

Diese Probleme bieten sich nicht nur dem Leser dieser Statements, sondern auch der Compiler musste aufwändiger erstellt werden um diese Konstruktionen richtig interpretieren zu können. Belege???

template typename dependant type problematik???

### 2.2.2.2 Pointer

Naiv hätte ich bei der Variablendeklaration von

```
int * ip, jp;
```

erwartet das *ip* und *jp* Pointer auf *int* sind. Damit lag ich (so wie viele meiner Kommilitonen und Kollegen) falsch und als ich zum ersten Mal über dieses Problem stolperte, stellte sich nach etwas Recherche raus das *ip* ein Pointer und *jp* nur ein *int* Objekt waren, richtig wäre an dieser Stelle

```
int *ip, *jp;
```

gewesen.

Der Ursprung dieser Syntax ist in [K&R] so beschrieben:

*"The declaration of the pointer ip,*  
`int *ip;`  
*is intended as a mnemonic; it says that the expression \*ip is an int."*

Das heißt dass man später durch Aufruf von *\*ip* an das bezielte *int* gelangt. Kommt noch ein zweiter Variablenname hinzu, muss auch er mit der Eselsbrücke versehen werden wie man an sein *int* gelangt.

Obwohl ich diese Gedächtnisstütze jetzt schon lange kenne, arbeitet mein Hirn mit einem anderen mentalen Modell. In diesem Modell besteht eine Variablendeklaration aus einem Typ (der Indirektionsebenen beinhaltet) und einem oder mehreren Namen. Nicht aus einem Basistyp (ohne Indirektionsebenen) und Namen vermischt mit Eselsbrücken wie ich an den Basistyp komme.

Als eines von vielen möglichen Beispielen von Template-Klassen verhalten sich die in C++11 eingeführten Smartpointer getreu meinem mentalen Modell, so das

```
unique_ptr<int> ip, jp;
```

jeweils *ip* und *jp* als (Unique) Pointer auf ein *int* definiert werden.

### 2.2.2.3 Array

Ein ähnliches Bild wie bei den Pointern findet sich bei der Syntax zur Definition von Arrays [K&R]:

*"The declaration*  
`int a[10];`  
*defines an array a of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ... , a[9]."*

Auch hier kommt man im Nachhinein durch Aufruf von *a[zahl]* an die jeweiligen *int*, interessanterweise ist aber genau der Aufruf von *a[10]* ein Fehler welcher UB hervorruft. Um zwei oder mehr Arrays zu erzeugen muss man dies hier schreiben: `int a[10], b[10];`

Das C++11 Äquivalent dazu ist `array<int,10> a, b;` bei dem wieder *a* und *b* jeweils Arrays der Größe 10 vom Typ *int* sind.

### 2.2.2.4 Die Kombination von Pointern und Arrays

Sofern man Variablen immer jeweils einzeln erzeugt oder man dem sich mit K&Rs mentalem Modell anfreunden konnte, aber man dennoch nicht auf etwas Problematik verzichten will, dann kann man die beiden zuvor genannten Fälle kombinieren:

```
int *c[4];
```

Man weiß hier zwar das `*c[4]` ein `int` ist, aber der Teil mit dem `int` ist auch das leicht Verständlichste.

Ob dies ein Pointer auf ein Array mit 4 `int` ist oder ob es ein Array aus 4 Pointern auf `int` ist ist nicht direkt ersichtlich. Dafür muss man wissen welche Operation zuerst ausgeführt wird, die Dereferenzierung oder das Subskript. Und es stellt sich noch eine weitere Frage: Wenn es das Eine ist, wie bekommt man es dazu das Andere zu sein?

(Auflösung: Es ist ein Array mit 4 Pointern auf `int`. Die Umkehr wird durch Klammersetzung im Typen erreicht: `int (*c)[4];`)

Alternativ dazu sind die C++11 Äquivalente zwar schreibintensiv, aber eindeutig und direkt verständlich:

```
unique_ptr<array<int,4>> uptr_to_ary;
array<unique_ptr<int>,4> ary_of_uptr;
```

### 2.2.2.5 Funktionspointer

Ein besonders schwer zu lesen und schreibender Teil der C & C++ Syntax ist die der Funktionspointer.

Hier 2 Beispiele:

```
int f(int a,int b) {...}           // Definition der Funktion f mit zwei int Parametern und int Rückgabe
int(*fp)(int,int) = f;           // Pointer auf eine Funktion fp vom gleichen Typ wie f, die auf f zeigt
void(*ffp)(int*)(int,int));      // Pointer auf eine Funktion ffp die einen Parameter vom Typ von fp hat
```

Das Problematische hier sind die Funktionspointer deren Namen inmitten der Deklaration *versteckt* ist, welches sich noch undurchsichtiger bei verschachtelten Funktionspointern auswirkt, da diese Namenlos nur durch ein "`(*)`" identifiziert werden.

Die zweite Zeile kann auf den ersten Blick für eine *Functional Cast Expression* oder eine Deklaration eines Pointers auf ein `int` gehalten werden; vergleiche mit der Auflösung des vorangegangenen Punktes: `int(*c)[4];`.

## 2.2.3 Schlechtes Standardverhalten

### 2.2.3.1 Ein Parameter Konstruktor

Ein Konstruktor, welcher einen einzelnen Parameter übernimmt, erfüllt nicht nur die Rolle eines Konstruktors mit einem Parameter, sondern er exponiert damit auch die Funktionalität der impliziten Konvertierung des Parametertypen zum Klassentypen. Dieses Default-Verhalten bringt den nicht-Profi in die Situation, Funktionalität bereitzustellen ohne davon zu wissen.

```
class C {
    C(int) {...}
    explicit C(float) {...}
};
C c1 = C(1);
C c2 = 2;
C c3 = C(3.0f);
C c4 = 4.0f;           // Compiletime Error
```

Der Entwickler wollte `c1` und bekommt `c2` ohne jede Warnung hinzu. Gleichfalls funktioniert `c3`, nur `c4` wird mit einem Compiletime Fehler verhindert.



### 2.2.3.2 Private Ableitung

```
class D : C {...};           // D leitet private von C ab, was keinen Sinn ergibt...
class D { C base; ... };    // ...sollte man dies gewollt haben, kann man es auch so schreiben
class D : public C {...};   // Dies ist der Fall den man üblicherweise will
```

### 2.2.3.3 Alles kann Exceptions werfen

Funktionen sind, wieder aufgrund von Kompatibilitätsgründen, auch ohne explizite Auszeichnung so definiert das sie Exceptions werfen können. Das führt dazu das man in Exception-Armen Projekten jede Funktion mit *noexcept* auszeichnen muss. Mitglieder des C++ Standardkomitees denken darüber nach in weniger Funktionalität der Standardbibliothek Exceptions zu nutzen und auch ein neues invertiertes Keyword Namens *throws* einzuführen.

### 2.2.3.4 Switch fällt von selbst

Als *Labels* und *goto* noch ein oft genutzter Teil von Programmiersprachen war, konnte man noch einfach verstehen was an dem folgenden Beispiel falsch läuft. Da aber heutzutage, bis auf das switch/case Statement, alle Kontrollstrukturen durch geschweifte Klammern begrenzt sind, tritt der hier gezeigte Fehler nicht selten auf.

```
switch( a ) {
    case 1:
        do_this();
    case 2:
        or_do_that();
        break;           // ???
    case 3:
    case 4:
        two_cases_want_the_same();
        break;
}
```

Hier wird im Falle das *a = 1* ist *do\_this* ausgeführt und nach dessen ordnungsgemäßer Beendigung auch *or\_do\_that*.

## 2.2.4 Inkonsistente Syntax

### 2.2.4.1 Funktions- & Methodendeklarationen

In C++ gibt es zahlreiche unterschiedliche Möglichkeiten Funktionen und Methoden zu definieren.

```
float pow(float b, int e)      {...}           // nicht optimal weil fest vorgegebener Typ
auto pow(auto b, int e) -> float {...}         // ab C++11, auto vorn überflüssig, aber syntaktisch
nötig
auto pow(auto b, int e)      {...}           // ab C++14
template <typename T>
T pow(T b, int e)            {...}           // Analog auch auto Rückgabetyt, auch anhängend möglich
[](auto b, int e) -> auto     {...}           // Lambda-Syntax, kein Rückgabetyt vorweg
```

Die Spezifikation der Typen von Parameter und Rückgabe ist in jedem aufgeführten Fall unterschiedlich, besonders die Positionierung des Rückgabetypen variiert stark. Die mit C++11 eingeführte Lambda-Syntax besitzt gar keine Möglichkeit eines vorangestellten Rückgabetypen mehr.

### 2.2.4.2 Notwendigkeit des Semikolons

Manchmal braucht man ein Semikolon nach Sprachkonstrukten und ein andermal braucht man es bei sehr ähnlichen Konstrukten nicht.

```
class E {...};
do {...} while(...);

namespace A {...}
while(...) {...}
if(...) {...}
```

### 2.2.4.3 Verteilung der Attribute

In C++ sind die Attribute einer Funktion / Methode chaotisch um den Kern der Deklaration verteilt.

Gegeben sei dieses Beispiel, zur besseren Lesbarkeit nach logischen Gruppen umgebrochen, formatiert und erklärt:

```
[[noreturn]]           // Attribut welches angibt das die Funktion nicht zurückkehren
wird
virtual               // Methode ist virtuell überschreibbar
const float const      // Rückgabotyp
foo()                 // Methodenname und leere Parameterliste
const override final noexcept // Methode verändert this nicht, ist überschrieben, nicht weiter...
{...}                // ...überschreibbar und wirft keine Exceptions
```

Man beachte das *const* dreimal vorkommt, wobei sich die ersten zwei Vorkommen (redundant) auf den Rückgabeparameter, das dritte auf die Methode selbst bezieht. Man betrachte das *virtual*, *override* und *final* verwandte Terme sind (die innerhalb einer Definition wie hier aufgeführt nicht gemeinsam verwendet werden) aber dennoch an unterschiedlichen Positionen geschrieben werden.

Diese wahllos wirkende Verteilung der Attribute ist der Evolution von C++ geschuldet, *virtual* ist ein Keyword aus der Entstehung von C++, *override* und *final* sind relativ neu und lediglich *identifiers with special meaning*, welche nur in einem speziellen Kontext als Keyword dienen. In anderen Kontexten können sie etwa als Typ oder Variablenname verwendet werden, aus diesem Grund können sie nicht an der gleichen Position wie *virtual* geschrieben werden.

??? Standard zitieren

### 2.2.4.4 Implizite Konvertierungen

In C++ sind viele unsichere implizite Konvertierungen möglich. Dies betrifft insbesondere numerische Datentypen.

```
void foo(int) {...}
void foo(void*) {...}
int a = 707, b = 1337;
char c = a + b;           // Konvertierungen zu kleineren Typen erzeugten keine Warnung
static_assert(-1 < 1u);    // Dies hier schlägt fehl, weil -1 implizit zu unsigned konvertiert wird
int * ip = 0;             // 0 ist implizit zu jeglichem Pointer konvertierbar...
int * jp = NULL;          // ...das ist so, weil das NULL Makro oft einfach als 0 definiert ist
foo(NULL);                // Ruft foo(int) auf
foo(nullptr);             // Ruft foo(void*) auf
void * vp = ip;           // Alle Pointer sind implizit zu void Pointern konvertierbar
if( a ) {...}             // Jegliche numerische Datentypen sind implizit zu bool konvertierbar,...
if( ip ) {...}            // ...was auch für Pointer gilt
```

Das die Addition zweier *int* sich sehr Wahrscheinlich nicht vollständig in einem *char* speichern lässt, ist kein Zustand über den sich C++ beschwert. Noch absurder wird es wenn -1 nicht mehr kleiner als 1 sein soll. Das liegt daran dass für diesen Vergleich von *signed* und *unsigned int* beide in *unsigned int* konvertiert werden. Diese Konvertierung findet höchst performant statt, was bedeutet dass das -1 *signed int* einfach als *unsigned int* reinterpretiert wird und dann den Wert 4294967295 hat, welcher nicht kleiner als 1 ist.

Um einem Pointer mit dem besonderen leeren Wert zu initialisieren wird ihm entweder 0, *NULL* oder *nullptr* zugewiesen, wobei aber nur der letztgenannte vom Typ Pointer ist. Dies wirkt sich insbesondere

negativ bei Funktionsüberladungen aus, welche im Falle der Parametrierung etwa mit *NULL* die Integer Überladungen bevorzugen.

## 2.2.5 Problematische Reihenfolge der Syntax

### 2.2.5.1 Unerwartete Operator Precedence

Operatoren sind im aktuellen C++ in 17 Ebenen einer Vorrangs-Tabelle eingeteilt.

[??? Vorrangstabelle Einfügen]

Manche dieser Ebenen sind in einer Reihenfolge, in welcher man sie nicht vermuten würde. Die Kommentare zeigen mittels Klammerung welcher Teil zuerst evaluiert wird.

```
cout << 1 & 2;           // (cout << 1) & 2;
if( a ^ b == 2 )      ... // if( a ^ (b == 2) ) ...
cout << a == b;       // (cout << a) == b;
```

Die Einordnung in der Vorrangs-Tabelle der Bitweisen *Und* und *Oder* Operation geht auf BCPL zurück. Dort erfüllte der Operator "&" kontextsensitiv entweder die Funktionalität der Konjunktion (logischem *Und*) oder des bitweisen *Und*, analog traf das selbe auf den Operator "|" und *Oder* zu. In der Entwicklung von C wurde die Funktionalität in & und && aufgespalten.

Quelle: <https://www.lysator.liu.se/c/clive-on-bcpl.html#operators>

[https://en.wikipedia.org/wiki/Operators\\_in\\_C\\_and\\_C%2B](https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B)

[%2B#Criticism\\_of\\_bitwise\\_and\\_equality\\_operators\\_precedence](https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B#Criticism_of_bitwise_and_equality_operators_precedence)

(Auszug aus der) Tabelle?

<https://www.bell-labs.com/usr/dmr/www/chist.html>

*Their tardy introduction explains an infelicity of C's precedence rules. In B one writes*

```
if (a==b & c) ...
```

*to check whether a equals b and c is non-zero; in such a conditional expression it is better that & have lower precedence than ==. In converting from B to C, one wants to replace & by && in such a statement; to make the conversion less painful, we decided to keep the precedence of the & operator the same relative to ==, and merely split the precedence of && slightly from &. Today, it seems that it would have been preferable to move the relative precedences of & and ==, and thereby simplify a common C idiom: to test a masked value against another value, one must write*

```
if ((a&mask) == b) ...
```

*where the inner parentheses are required but easily forgotten.*

Die Notwendigkeit B Programme nach C umzusetzen ist nicht mehr gegeben, dennoch schleppt C++ immer noch diesen Ballast mit sich herum.

### 2.2.5.2 Pre- und Suffix Operationen

Ähnlich zur Variablendeklaration von Pointern in Kombination mit Arrays bietet auch das Zusammenspiel von Pre- und Suffix Operationen oft eine Ambiguität mit sich, welche Kammersetzung erfordert.

Verwirrend ist auch die Nutzung der gleichen Symbole für unterschiedliche Arten von Operationen: &, \*, +, -

Diese werden zum einen für Unäre Prefix- sowie Binäre Operationen verwendet, als auch in doppelter Erwähnung als Prefix, Suffix und Binäre Operation.

```
*a[i];           // Ausgeführt wird *(a[i]) denn [] bindet stärker als *
(*a)[i];         // Sollte man a zuerst dereferenzieren wollen so muss man dies hier schreiben
```

```
(*a)->draw();      // Auch bei mehrfach Dereferenzierung muss geklammert werden

&i & & j && &k;    // Keine Verletzung der Sprache! Funktioniert z.B. nicht mit int...
++i++ + ++j++;     // ...aber kompiliert wenn i, j und k von einem präpariertem Typ sind
*i**j;             // In diesem Beispiel sind nicht mal Leerzeichen nötig, Ausführung: (*i)*(**j);
```

### 2.2.5.3 Zuweisungen und Throw Expressions

```
if( a = 1 )          // Fälschliche Zuweisung

??? throw bspl
```

### 2.2.5.4 Textkomposition mit Streams

(MOVE)

BSPL

Es ist zu erwähnen das die *stream* Schreibweise bei Textkomposition inhärent suboptimal für viele Einsatzgebiete ist; z. B. bei Übersetzungen, wenn die Reihenfolge der Argumente je nach Sprache anders ist.

Myll bietet deshalb die *fmt.dev* Bibliothek mit an, welche eine sehr gute Mischung aus Features, Compile-time, Binarysize und Runtimeperformance bietet.

??? änderung das es nur noch eine footnote bei den ops << >> ist

## 2.2.6 Schwer durchschaubare automatische Verhaltensweisen

### 2.2.6.1 Special Member Functions

Es gibt sechs spezielle an eine Klasse gebundene Funktionen in C++ und diese werden unter jeweils unterschiedlichen Bedingungen automatisch generiert.

Siehe: [https://en.wikipedia.org/wiki/Special\\_member\\_functions](https://en.wikipedia.org/wiki/Special_member_functions)

- Default Constructor
- Destructor
- Copy Constructor
- Copy Assignment Operator
- Move Constructor
- Move Assignment Operator

Problematisch ist dies in vielerlei Hinsicht. Zuallererst muss man sie überhaupt erkennen um sie mit der notwendigen Sorgfalt zu behandeln. Drei von ihnen sind Konstruktoren, zwei davon erwarten einen einzelnen Parameter mit vorgegebenem Typ. Diese gleichen Typen erwarten auch die speziellen Zuweisungsoperatoren.

Sobald man einen davon selbst implementiert, fallen automatisch einige der anderen weg. Dies ist schwer durchschaubar und wirft im Normalfall noch zusätzliche Probleme auf, so das die Verhaltensregeln Rule-of-Three, Rule-of-Five und Rule-of-Zero erdacht wurden. Diese geben vor, das wenn man eine der speziellen Funktionen implementiert (mit Ausnahme des Default Constructors), das man in dem Zuge auch einige der anderen Implementieren soll. Anders der Fall des Rule-of-Zero, welcher vorgibt das man komplett auf die Implementierung der speziellen Funktionen (wiederum mit Ausnahme des Default Constructors) verzichten sollte.

Diese Regeln existieren auf dem Papier, aber leider nicht in der Sprache, somit ist die Validation deren Befolgung auch komplett an den Benutzer übertragen.

## Standard from 10.2/2

Zusammenfassung:

- Man muss sie erkennen
- Sie müssen in besonderen Konstellationen auftreten
- Der Compiler validiert nichts davon

Lösung:

```
[rule_of=5] // Dies wirft einen Fehler beim Kompilieren: Es fehlt dtor und move=
class Test {
    move_ctor(other) {...} // Der Leser sieht direkt das es sich um einen Move Constructor handelt
    copy_ctor(other) {...} // Der Leser sieht direkt das es sich um einen Copy Constructor handelt
    copy=(other) {...} // Der Leser sieht direkt das es sich um ein Copy Assignment handelt
}
```

## 2.2.6.2 Overloading mit Vererbung, Shadowing

<https://www.geeksforgeeks.org/does-overloading-work-with-inheritance>

```
class Base {
    void f(int) {...};
}
class Derived : public Base {
    // using Base::f; // diese Zeile würde das Problem beheben
    void f(double) {...};
}
int main() {
    Derived d;
    d.f(1337); // Ruft Derived::f(double) auf und nicht das logischere Base:f(int)
}
```

Lösung:

Inversion des Verhaltens, erwarte vom Benutzer das er explizit erwähnt das er Shadowing will.

```
class Derived : public Base {
    void f(double) {...}; // using Base::f; wird automatisch hinzugefügt
}
class OtherDerived : public Base {
    [shadow]
    void f(double) {...}; // using Base::f; wird nicht hinzugefügt
}
int main() {
    Derived d;
    d.f(1337); // Ruft Base::f(double) auf
    OtherDerived d2;
    d2.f(1338); // Ruft wie explizit angefordert Derived::f(double) auf
}
```

## 2.2.7 Schlechte Namensgebung

vector  
map  
unsigned long long int  
constexpr

Lösung:

dyn\_array  
search\_tree oder sorted\_dict // vermittelt sofort das es sortiert ist  
uint64  
???

## 2.2.8 Verbose Schreibweise

```
unique_ptr
static_cast<>
(auto)
(move)
```

## 2.2.9 Keyword reusage

Inflationäre Nutzung von:

```
const
static
constexpr
constexpr
```

Lösung:

```
[pure]
[(no)global]
unnamed namespace { ... }
[RT] stmt;    // Force execution at Runtime
[CT] stmt;    // Force execution at Compiletime
```

## 2.2.10 East Const und West Const

Gangkriminalität zwischen verfeindeten Clans ist hier nicht gemeint, es geht lediglich um die mögliche Positionierung des Keywords *const*. (Auch wenn es auch hier analoge Auseinandersetzungen wie zwischen der *geschweifte Klammer in neuer Zeile* mit der *geschweifte Klammer auf der gleichen Zeile* Fraktionen gibt)

```
const int * a;
int const * b;
int * const c;
```

Die Variablendeklaration von *a* und *b* sind identisch, das *const* bezieht sich auf das *int*. Bei der Deklaration von *c* hingegen bezieht sich das *const* auf den Pointer. Das heißt dass es an manchen Stellen möglich ist das *const* links von der zu attributierenden Stelle zu schreiben (west const) und an anderen rechts (east const).

Auch wenn es wesentlich häufiger der Fall ist *a* anstelle von *b* anzutreffen, gibt es in C++ nur eine Schreibweise die immer Funktioniert, welches natürlich der unnatürlichere Fall des *east const* ist.

## 2.2.11 WOANDERS HIN:???

FhO

LINQ

(Schöner machen, ergänzen und strukturieren)

## 3 Konzept

### 3.1 Design der Syntax

Um bei der Konzeption der Sprache das Ziel nicht aus den Augen zu verlieren, wurden einige simple Grundsätze definiert. Diese Grundsätze sind auch auf Englisch verfügbar.

Der Autor vertritt die Auffassung das viel Inspiration von anderen Sprachen eine gute Sache ist und mehr Wiedererkennungswert den Einstieg einfacher macht.

#### 3.1.1 Grundsätze der Sprache Myll

1. Erwarte vom Benutzer nicht sich zu wiederholen, wenn es nicht nötig ist
2. Außergewöhnliches Verhalten muss Explizit sein
3. Das was man Naïv  $\geq 75\%$  der Zeit will, kann Implizit oder Standard sein
4. Breche nicht mit der grundsätzlichen Semantik von C++
5. Breche mit C sofern es einen Nutzen bringt
6. Entwickle die Syntax so weiter das es zu keiner Most Vexing Parse, o.ä. kommen kann
7. Sei auch dann nützlich, wenn die Entwicklung am einmalig erzeugten C++ Code weitergeht
8. Spare nicht mit neuen Keywords, die Lesbarkeit dankt

##### 3.1.1.1 Fundamental Principles of Myll

1. Don't ask the user to repeat themselves, if it's not necessary
2. Exceptional behavior needs to be explicit
3. What you naively expect to happen in  $\geq 75\%$  cases can be implicit or default
4. Don't break with C++'s general semantic
5. Do break with C if there is a benefit
6. Evolve the syntax that it can't have a Most Vexing Parse or alike
7. Be useful even if a one-time translation to C++ is all that's wanted
8. Don't be frugal with new keywords, the readability benefits

##### 3.1.1.2 Bewertung der Grundsatz Einhaltung

Im späteren Verlauf der Umsetzung soll die entstandene Programmiersprache sich an der Einhaltung dieser Grundsätze messen.

??? Verschieben

1. Dedublizierung durch Reduktion auf eine Datei  
Keine Notwendigkeit von Prototypen
2. Implicit Konstruktoren  
Array Decay  
Switch Fall
3. Bitweise Operationen vor Gleichheit  
Kein Shadowing

Switch Break

Public Ableitung

Noexcept

4. Größte Sorgfalt wurde darauf gelegt diese Semantik beizubehalten
5. Jeder Bruch mit der Syntax von C ist dokumentiert und dient einem Nutzen
6. Viele syntaktische Makel welche in C & C++ auftraten wurden beachtet und vermieden
7. ??? Sei auch dann nützlich, wenn die Entwicklung am erzeugten C++ Code weitergeht  
Dies kann etwa durch die Erzeugung von vom Menschen lesbarem C++ Code passieren
8. Es wurden einige Keywords eingeführt um Ambiguitäten zu unterbinden, den Code Lesbarer zu gestalten und neue Konstrukte mit angenehmer Syntax einzuführen

### 3.1.2 Neue Keywords

Myll nutzt einige neue Keywords, denn Syntax ohne Keywords hat sehr schnell alle *schönen* Konstellationen verbraucht und Monster wie diese hier entstehen:

```
[](){}();           // valides C++11, leeres Lambda welches direkt aufgerufen wird
void(*ffp)(int*)(int,int)); // valide seit dem Anbeginn von C++
```

Die Liste der neuen Keywords ist:

```
ctor, dtor, func, proc, method, requires, alias, var, field, get, set, refget, loop, ???
```

Es wurden Abkürzungen mittlerer Länge gewählt welches sich an die ursprünglichen Keywords aus C++ anlehnt: struct(ure), enum(eration), float(ing point number), double (precision float), int(eger). Für Funktionen wäre *fn* zu wenig Information und kollidiert zu leicht. Für Methoden wäre *meth* eine ungewollte Kollision mit der Droge Methamphetamin gewesen.

## 3.2 Behandlung der Probleme

Hier werden die Probleme behandelt, welche in Kapitel 2.2 aufgezeigt wurden.

### 3.2.1 Deklarationen, Definitionen und Initialisierung

#### 3.2.1.1 Reihenfolge und Prototypen

In Myll gibt es keine separate Deklaration und Definition, beides erfolgt in einem Schritt. Sollte im erzeugten C++ Code dadurch ein Reihenfolge Problem entstehen, wird automatisch eine Prototyp-Deklaration vor der Nutzungsstelle erzeugt.

#### 3.2.1.2 Aufteilung auf verschiedene Orte

Durch diese Zusammenführung ist auch die Aufteilung auf separate Dateien unnötig geworden. Es werden natürlich für den erzeugten C++ Code weiterhin .h und .cpp Dateien erzeugt.

Das folgende Beispiel enthält die gleiche Menge an Information wie die Dateien *myclass.h* und *myclass.cpp* aus der Analyse. Es lassen sich aus diesem Beispiel die originalen Dateien (fast identisch) wiederherstellen.

Datei *myclass.myl*:

```
class MyClass {
    field int myVar;
```



```

    field int myOtherVar;
    field int myVar3 = 3;
    [static] field int myStatic = 1;
public:
    ctor() {
        myVar = 1;
        myOtherVar = 2;
    }
    proc myProc() -> int {...}
    [inline] proc myOtherProc() -> int {...}
    [virtual] proc myVirtual() -> int {...}
    [inline] func myFunc3() -> int {...} // func ist implizit const, alternativ:
    [inline, const] proc myFunc3() -> int {...} // proc mit const attribut auch möglich
}

```

Analyse des Ursprungscodes und der bereinigten Syntax, ohne Whitespace und Kommentare:

Ursprung, C++: 18 Zeilen, 34 Wörter, 351 Zeichen

Bereinigung: 13 Zeilen, 26 Wörter, 264 Zeichen

Diese Transformation brachte in diesem sehr simplen, von Implementierung befreiten, Beispiel eine Reduktion der Zeilen um 28%, der Wörter um 24% und der Zeichen um 21%. Es geht in diesem betrachteten Punkt des Konzepts auch explizit nicht um die Implementierung. Größere und realistischere Beispiele werden späteren Verlauf der Thesis Betrachtet und Ausgewertet.

### 3.2.1.3 Aufteilung im Kontext lebendiger Codebases

#### 2.2.1.3 Erklärt lebendige Codebases???

Lebendiger Code profitiert von der irrelevant gewordenen Reihenfolge von Deklarationen und der Vereinigung auf eine Datei. Ein inline hinzu, ein inline entfernt, erzeugt eine geänderte Zeile im *diff* der Versionen.

## 3.2.2 Fehleranfällige Variablen & Typen Deklarationen

### 3.2.2.1 Ambiguität einer Deklaration

Um der Ambiguität ob etwas Beispielsweise eine Variable, eine Multiplikation oder einen Funktions-Prototyp ist entgegen zu wirken führt Myll Keywords zur einfachen Identifikation deren ein.

```

var a * b;           // Variablen Deklaration
const a * b;         // Konstanten Deklaration
a * b;              // Multiplikation
var c d;             // Variablen Deklaration
var c e();           // Variablen Deklaration mit Aufruf des Standard Konstruktors
func e() -> c;        // Funktionsdeklaration (unnötig in Myll, mehr dazu in dediziertem Kapitel)

```

Das Keyword *var* für Variablen, *const* für Konstanten (was in C++ bereits so aussieht), *func* für Funktionen.

Außerdem ist es gleichförmig zu anderen Konstrukten in C++, welche mit einem vorangestellten Keyword den Leser direkt wissen lassen was folgen wird: *class*, *struct*, *union*, *enum*, *namespace*, *using*, *static*, selbst *template*, *typename*, *typedef*, *#define* erfüllen dieses Kriterium.

Inspiriert war diese Syntax von einigen anderen Programmiersprachen, aber die Entscheidung zu genau diesen Keywords kam durch das Projekt Bitwise von Per Vognsen welcher in seiner Programmiersprache **Ion** die gleichen Keywords verwendet???

### 3.2.2.2 Pointer und Array

In Myll ändert sich die Pointer und Array Syntax so das der Stern und auch das Kaufmanns-Und sowie die eckigen Klammern zum Typ gehören und der Typ für alle Variablennamen gleich ist:

```
var int*    a, b; // Variablen a und b sind beide Pointer auf int
var int[10] c, d; // Variablen c und d sind beides int Arrays mit 10 Elementen
```

Verloren geht dadurch die Möglichkeit total unterschiedliche Initialisierungen wie diese hier in einer Zeile schreiben:

```
int i = 1, *p = NULL, f(), (*pf)(double), a[10];
```

Diese Änderungen beseitigt nicht nur die potentiell Fehleranfällige Syntax von K&R C, sondern harmonisiert außerdem mit der Template Schreibweise in C++ und der allgemeinen Schreibweise in anderen Sprachen wie C#, Java, D.

Dies ist ein Trade-off welcher die häufiger genutzte Art der Deklaration vereinfacht und die seltener genutzte erschwert.

### 3.2.2.3 Pointer auf Arrays

Des weiteren wird eine neue Schreibweise für Pointer auf Arrays eingeführt:

```
int[*] a;
int[*] a; // besser???
```

Damit lassen sich möglicherweise schon in Myll Fehler diagnostizieren wenn ein Pointer auf ein Skalar so verwendet wird wie ein Pointer auf ein Array z.B.:

```
int    skalar;
int[10] ary;
int*    ptr_to_skalar = &skalar;
int[*]  ptr_to_ary    = &ary;      // Expliziter Decay des Arrays zu einem Array-Pointer
int[*]  ptr_to_ary    = &ary;      // Expliziter Decay des Arrays zu einem Array-Pointer, besser???
ptr_auf_skalar++;
ptr_auf_ary++;                    // Compiletime Error: Pointer auf Skalare verbieten Arithmetik
ptr_auf_ary++;                    // Funktioniert wie erwartet
```

Auch wird der implizite Decay von Arrays zu Pointern verhindert, will man, dass ein Array zum Pointer wird, nutzt man die identische Syntax um ein Pointer auf ein Skalar zu bekommen.

### 3.2.2.4 Die Kombination von Pointern und Arrays

Die Kombination aus Pointern und Arrays ist durch die bereits genannten Änderungen schon abgedeckt und die Unklarheit der Reihenfolge, welche in C vorhanden war, ist nicht mehr vorhanden. Es gibt:

```
int[4]* ptr_to_ary_of_4; // Ein Pointer auf ein Array aus 4 int
int*[4] ary_of_4_ptrs;  // Ein Array aus 4 Pointern auf int
```

Dadurch besteht keine Notwendigkeit Klammern zusetzen und es ist auch nicht so schreibintensiv wie die Template-Syntax.

### 3.2.2.5 Funktionspointer

In Myll erkennt am linken Rand der Deklaration sehr schnell ob etwas eine Funktion, ein Funktionspointer oder etwas anderes ist, die Namen sind außerdem leicht auffindbar an der rechten Seite der Deklaration. Auch verschachtelte Funktionspointer sind nicht schwerer lesbar als Funktionspointer im Allgemeinen.

Das identische Beispiel von 2.2.2.5 in Myll:

```
func f(int,int)->int {...}
var func(int,int)->int fp = f;
var func(func(int,int)->int)->void ffp;
```

## 3.2.3 Schlechtes Standardverhalten

### 3.2.3.1 Ein Parameter Konstruktor

Die Lösung zum Konstruktor, welcher einen einzelnen Parameter übernimmt, ist die Umkehrung der Bereitstellung der impliziten Konvertierung. Sollte diese Konvertierung gewünscht sein, muss sie explizit erwähnt werden.

Auch Jonathan Müller, C++ Library Developer und Konferenzsprecher betrachtete dieses Problem in einem Blogpost und Urteilte:

*"As with most defaults, this default is wrong. Constructors should be explicit by default and have an implicit keyword for the opposite."*

Quelle: <https://foonathan.net/blog/2017/10/11/explicit-assignment.html>

Dieses Beispiel in Proto-Myll weist die gleiche Bereitstellung wie im C++ Beispiel auf:

```
class C {
    implicit C(int) {...}
    C(float) {...}
};
C c1 = C(1);
C c2 = 2;           // ok, da explizit erwähnt wurde das von int implizite Konvertierungen erlauben soll
C c3 = C(3.0f);
C c4 = 4.0f;        // Error
```

Der Entwickler bekommt ein Funktionierendes *c1* und wie angefordert auch *c2*. Gleichfalls funktioniert *c3*, jedoch wird *c4* mit einem Compiletime Fehler verhindert.

### 3.2.3.2 Private Ableitung

```
class D : C {...};           // Dies ist der Fall den man üblicherweise will
class D : private C {...};  // Wenn man will kann D private von C ableiten
```

### 3.2.3.3 Alles kann Exceptions werfen

Das für zukünftige C++ Standards angedachte Keyword *throws* ist in Myll nötig um in einer Funktion Exceptions nutzen zu können, dieses Keyword kann auch einmalig einer gesamten Klasse zugewiesen werden.

### 3.2.3.4 Switch fällt von selbst

Umkehr der Verhaltensweise, *fall* wenn Fallthrough erwünscht, *break* ist der default.

```
switch( a ) {
    case 1:
        do_this();
                                // implizites break
    case 2:
        or_do_that();
        fall;                    // fällt in den darunterliegenden case
    case 3, 4:                  // wenn mehrere cases genau das gleiche wollen geht es auch so
        two_cases_want_the_same();
}
```

## 3.2.4 Inkonsistente Syntax

### 3.2.4.1 Funktions- & Methodendeklarationen

So sehen die identischen Deklarationen in Myll aus.

```
func pow(float b, int e) -> float {...} // nicht optimal weil fest vorgegebener Typ
func pow(auto b, int e) -> auto {...} // Rückgabotyp
func pow(auto b, int e) {...} // das gleiche wie die Zeile davor
func pow<T>(T b, int e) -> T {...} // Auch auto Rückgabotyp möglich
func(auto b, int e) -> auto {...} // Lambda-Syntax, fast identisch zur Funktions-Syntax

proc do_something() {...} // Momentan Synonym zu func
method draw() {...} // Innerhalb von Klassen Synonym zu func
```

Funktionen und Methoden aller Art werden mit dem neuen Keyword *func* eingeleitet, die Rückgabotypen werden nur Folgend angegeben. In der Evolution von C++ wurde der folgende Rückgabotyp mit C++11 eingeführt und gewann seit dem immer mehr an Bedeutung und Funktionalität. Die Template-Syntax wurde für simple Fälle wie diesen hier entschlackt, mehr dazu in separatem Kapitel. Lambdas sind nun leichter Identifizierbar und bis auf ihre Namenlosigkeit identisch in der Schreibweise zu Funktionen.

Die Nutzung der Keywords *proc* und *method* ist auch möglich, *proc* ist momentan komplett Synonym zu *func*, *method* hingegen nur im Klassenkontext nutzbar.

Analog zu *const* Methoden, welche eine Veränderung des assoziierten Objektes verbieten, gibt es die Möglichkeit jede Funktion als *pure* zu markieren, welche die Veränderung von jeglichem globalen Zustand verbietet.

### 3.2.4.2 Notwendigkeit des Semikolons

Ohne die Notwendigkeit, wie in C, für alles via *typedef* einen nutzbaren Alias zu erzeugen, fällt auch die Notwendigkeit des Semikolons nach Klassen, Strukturen, Unions und Enumerationen weg. Verloren ist die Möglichkeit der direkten Erzeugung von Variablen des neuen Typs.

```
class E {...}
do {...} while(...)
```

### 3.2.4.3 Verteilung der Attribute

Das gleiche Beispiel auch hier nach logischen Gruppen Formatiert:

```
[noreturn, pure, virtual, override, final, noexcept] // Alles was sich auf die Methode bezieht
func foo() // Rückgabotyp mit west-const
-> const float
{...}
```

Myll gruppiert Attribute in Eckigen Klammern vor dem jeweiligen Konstrukt, dadurch sind zukünftige Erweiterungen einfach möglich, ohne das es Kollisionen mit bestehendem Code & Syntax gibt. *West const* ist hier die einzig Korrekte Schreibweise um etwas *const* zu machen.

## 3.2.5 Problematische Reihenfolge der Syntax

### 3.2.5.1 Unerwartete Operator Precedence

In Java und C# sind die Operatoren *&*, *|* und *^* in ähnlichem Vorrang wie in C++ und seinen Vorgängern einsortiert und diese verhalten sich je nach booleschen oder ganzzahligem Kontext analog zu C++.

Die Sprachen Rust und Ruby hingegen sortieren die Operatoren `&`, `|` und `^` direkt über die Vergleichsoperatoren ein und eliminieren die in Kapitel 2.2.5.1 angesprochene Problematik.

Myll ordnet die booleschen Operatoren in die Ebenen der Punkt- und Strichrechnung ein: Operator `&` in die Ebene der Punktrechnung, `|` und `^` in die Ebene der Strichrechnung.

Die Idee hierzu kam durch das Projekt Bitwise von Per Vognsen welcher in seiner Programmiersprache **Ion** die gleiche Änderung der Operator Precedence umgesetzt hat.

Der letzte aufgeführte Problemfall wurde nicht Behandelt damit dies hier noch so funktioniert:

```
if( a << 1 == 23 ) // if( (a << 1) == 23 )
```

Siehe auch 4.1.4.3. what??? leider Pointer auf umbenanntes Kapitel

### 3.2.5.2 Pre- und Suffix Operationen

Pech:

```
(*a)->draw(); // Auch bei mehrfach Dereferenzierung muss geklammert werden
```

### 3.2.5.3 Zuweisungen und Throw Expressions

Zuweisungen und *throw* sind keine Expressions mehr. Dadurch sind die genannten Beispiele nicht mehr valider Code und müssen in einfacher durchschaubaren Code umgeschrieben werden. In den seltenen Fällen wo eine Zuweisung eine Expression war muss sie nun einzeln als Statement geschrieben werden. C# und ??? löst es genau so.

## 3.3 Neue Features welche keine Problemfälle waren

### 3.3.1 Benamte Parameter

NICE TO HAVE, DONT WASTE TIME, DUDE!

```
func spawn_entity(bool is_visible) {...}
func spawn_entity(bool put_a_hat_on) {...}

spawn_entity(is_visible: true);
```

Diese beiden Funktionen würden in C++ durch ihre identische Signatur nicht gemeinsam existieren können. In Myll ist ein Aufruf mit `is_visible: true` jedoch eindeutig der ersten Variante zuzuordnen. Ein Aufruf dieser mit benannten(benannten) Parametern überladenen Funktion ohne die Nennung der Argumentnamen ist allerdings nicht mehr möglich, außer der ein Attribut markiert die primär/namenlos zu verwendende Funktion oder ein Invertiertes markiert die namenlos Untauglichen.

```
[primary/nameless] func spawn_entity(bool initially_visible) {...}
// ODER
[explicit/onlynamed] func spawn_entity(bool put_a_hat_on) {...}

spawn_entity(true); // ruft die primäre Funktion auf
spawn_entity(is_visible: true); // trotzdem schöner zu lesen, weil man sieht was das true bedeutet
```

Benannt == Benamt?

```
BSPL von R6  
bool isVisible = true;  
spawn_entity(isVisible);
```

## 4 Implementierung

### 4.1 Umsetzung

Meine Arbeit mit TypeScript (welches das Arbeiten mit JavaScript erträglich macht???) und ähnlichen Projekten, welche eine Source to Source Übersetzung (Transpilation) durchführen, brachte mich dazu dieses Projekt mit dem gleichen Verfahren zu beginnen.

Durch Transpilation entgeht man vielen problematischen und arbeitsintensiven Aspekten wie:

- Erzeugung von Assembly / Maschinencode
- Optimierung von Assembly / Maschinencode
- Erzeugung eines ABI (Application Binary Interface, Binäre Repräsentation ??? move to Begriffsbla)
- Linking von Object Files
- Bereitstellung eines Debuggers

Es bietet einem die Möglichkeit eine Sprache zu bedienen, ohne sie schreiben zu müssen, sowie die gesamte Infrastruktur der Zielsprache nutzen zu können.

Auch andere aktuelle Entwicklungen nutzen Transpilation um schneller einsatzbereit zu werden, wie zum Beispiel Jonathan Blow's JAI, welches anfänglich zu C++ Code transpilierte und Per Vognsen's ION welches immer noch zu C Code transpiliert. C++ selbst war in seinen Anfängen (C with Classes) als Transpiler zu C-Code umgesetzt.

Myll soll direkt mit C++ Code zusammenspielen, also einer Mixtur aus .cpp/.h und .myll Dateien, es soll die C++ Standardbibliothek, sowie C und C++ Libraries nutzen können.

Die Umsetzung erfolgt in C# und nutzt Antlr4 als Lexer und Parser Generator.

#### 4.1.1 C++ predigen, aber C# und Java trinken

Man sollte immer das beste Werkzeug für seine Arbeit nutzen. Aufgrund der Komplexität eine Sprache wie C++ zu Lexen und zu Parsen, bediene ich mich eines weitverbreiteten Parser-Generators und einer einfacher zu nutzenden Sprache. Die *Performance* welche bei dieser Arbeit im Vordergrund steht, ist die Umsetzung von möglichst viel Funktionalität im gegebenen Zeitfenster, dies lässt sich mit C# einfacher realisieren. ANTLR ist in Java geschrieben und erfordert lediglich für die Übersetzung der Grammatik ein installiertes JRE (Java Runtime Environment).

#### 4.1.2 Alternative Umsetzungsmöglichkeiten

Im Kontext der Erstellung dieser Masterarbeit habe ich auch Alternativen zur Umsetzung betrachtet.

- Selbstgeschriebener Lexer/Parser
  - Nach einigen kleinen Prototypen war klar das eine Programmiersprache, welche annähernd die Komplexität von C++ aufweist, sich nicht in absehbarer Zeit manuell Lexen & Parsen lässt
  - Dieses Unterfangen wäre sicher Interessant gewesen, ist hier aber nicht der gewünschte Fokus
- C++ mit Boost Spirit X3
  - Sehr schwierig eine komplexe und rekursive Grammatik zu definieren
  - Lange Kompilationszeiten
  - Spirit X3 hat angeblich eine gute Runtime Performance

- C++ mit PEGTL
  - Sehr ähnliche Probleme wie mit Spirit X3
- C++ mit ANTLR
  - Grammatik relativ einfach zu definieren, nah an der Backus-Naur-Form
  - Unterstützung von ANTLR in C++ ist nicht so ausgereift wie die von C#
  - Siehe die aufgeführten Nachteile von C++ in diesem Dokument
  - Implementations-Geschwindigkeit wichtiger als Runtime-Performance
- C# mit ANTLR
  - Grammatik relativ einfach zu definieren, nah an der Backus-Naur-Form
  - ANTLR hat die beste Interoperabilität mit C# von den unterstützten Sprachen welche ich beherrsche
  - Die Arbeit mit C# erlaubt einfache Nutzung einer Graphischen Oberfläche und hat auch sonst eine schnelle Implementationsgeschwindigkeit

## 4.2 Details

Die Pipeline eines Compilers ist meist in diese Schritte unterteilt:

- Lexikalische Analyse mittels Lexer
- Syntaktisch Analyse mittels Parser
- Semantische Analyse mittels Sema
- Ausgabe des Kompilats – hier Quellcodeerzeugung
- ???
- Profit :-P

Auch Myll hat diese Schritte.

## 4.3 Lexer / lexikalischer Scanner

Lexer werden verwendet um einen Eingabetext in logisch zusammenhängende Stücke zu zerteilen, diese Stücke werden Tokens genannt. Dieser Schritt, welcher *lexikalische Analyse* genannt wird, dient der Vorverarbeitung der Eingabe und wird von nahezu jedem Compiler durchgeführt. Leerraum wird im Normalfall in diesem Schritt verworfen (außer bei Sprachen in denen Leerraum Bedeutung hat, wie etwa Python). Oft werden Lexer durch ein Tool erzeugt, diese nehmen ein Vokabular und erzeugen daraus Programmcode. Alternativ zur Generierung können sie aber durchaus auch von Hand erstellt werden.

Ein Beispiel:

```
var int test = 99;
```

Diese Eingabe wird in folgende mit `|` getrennte Tokens aufgeteilt, Klammern zeigen optionalen Inhalt:

```
VAR | INT | ID( test ) | ASSIGN | INTEGER_LITERAL( 99 ) | SEMI
```

Der für Myll implementierte Lexer hat nur wenige Besonderheiten, zu seiner Erstellung wurde der Lexer-Generator von ANTLR 4 zu Hilfe genommen wurde, hierzu musste ein Vokabular definiert werden.

Die üblichen Spezialfälle wie *String* und *Character-Literale* sind so definiert das sie an ihrem einleitenden Zeichen anfangen (`»"«` und `»'«`). Von da an werden alle folgenden Zeichen und die meisten Escape-



Sequenzen zum Literal gezählt, bis die Verarbeitung auf ein schließendes Zeichen stößt, welches identisch mit dem respektiven einleitenden Zeichen ist. Das Character-Literal muss es genau ein Zeichen enthalten.

Die Escape-Sequenzen für Oktale-, Hexadezimale- und Unicode-Zeichen werden bislang nicht unterstützt. Alternativ dazu ist aber die direkte Eingabe von UTF-8 Zeichen im Texteditor möglich.

Auch *Kommentare* sind in Myll möglich und erlauben die aus C++ bekannten Varianten, *bis zum ende der Zeile* via `// ...` und *über mehrere Zeilen hinweg* via `/* ... */`. Sie schreiben alle Eingaben in ihrem Geltungsbereich in einen speziellen Kanal, welcher abseits der zu verarbeitenden Code-Tokens liegt. Aktuell wird dieser Kanal noch nicht weiter genutzt, d.h. die Kommentare werden nicht in den erzeugten übernommen, es ist aber durchaus möglich dies zu einem späteren Zeitpunkt nachzurüsten.

Die Sprachen C sowie C++ leiden unter einem Entscheidungsproblem welches ich in 2.2.2.1 schon aufgeführt habe. Die Problematik ist, dass hier Identifier (Bezeichner) schon während der lexikalischen Analyse in *Typen Identifier* und *Variablen Identifier* klassifiziert werden müssen. Zur Lösung dieser Problematik wird ein sogenannter *Lexer-Hack* verwendet um die Tokens richtig zu klassifizieren. Darin wird dem Lexer vom Parser ein Rückwärtskanal in Form einer Symboltabelle bereitgestellt, anhand derer eingesehen werden kann ob ein Identifier nun einen Typen oder eine Variable darstellt.

(Siehe [https://en.wikipedia.org/wiki/Lexer\\_hack](https://en.wikipedia.org/wiki/Lexer_hack))

Clang verzichtet auf diesen Hack und klassifiziert zunächst alle Identifier in eine gemeinsame Kategorie und entscheidet weit später, während der semantischen Analyse, die richtige Klassifikation.

Myll braucht diesen Hack ebenfalls nicht und unterscheidet Identifier auch nicht. Es ist durch die eindeutigere Syntax aber schon im Parser bekannt ob es sich um einen Typ oder eine Variable handelt.

Der für Myll implementierte Lexer betrachtet aber auch weiterhin die Tokens `>>` nicht gemeinsam, sondern als jeweils zwei separate `>` Tokens, so wie C++. Dies ist notwendig da sie je nach Auftrittsort entweder der logische bitweise Verschiebung nach rechts oder das schließen zweier Templates gemeint sind. Dies ist außerdem nun auch für den neuen Potenzierungs-Operator `**` nötig, da es in anderem Kontext auch zwei Dereferenzierungs-Aufrufe sein können.

```
vector< vector< int >>    // Zwei separate Templates werden geschlossen
                        84 >> 1 // Logischer rechts-shift
8 ** 2                    // 8 hoch 2
**a                       // a wird zwei mal dereferenziert
```

Nach dem Schritt der lexikalischen Analyse ist bekannt ob der eingegebene Text Anomalien aufweist. Keine Anomalien heißt aber noch lange nicht das ein Text Sinn macht, also ein syntaktisch korrektes Programm zeigt.

```
if if int ());           // OK vom Lexer: Keine Anomalie
€                        // Lexer meldet Fehler: € ist weder Keyword, Operator oder Identifier
"                        // Lexer meldet Fehler: Das Anführungszeichen wird nicht geschlossen
```

Die komplette Auflistung des Vokabulars des Lexers ist im Anhang ??? einzusehen.

## 4.4 Parser / syntaktische Analyse

Parser nehmen den vom Lexer erzeugten Token-Strom und erzeugen daraus entweder einen AST (Abstract Syntax Tree, einen Syntax Baum) oder einen Parse Tree (auch CST, Concrete Syntax Tree).

Mit dem linearen Strom an Tokens welcher der Lexer erzeugt hat, durchläuft der Parser in der syntaktischen Analyse Phase meist einen Entscheidungsbaum, in jenem er versucht alle Tokens zu passenden Regeln zu zuordnen. Die definierten Regeln sind mittels *Und* und *Oder* Gliedern verkettet und der gesamte Regelsatz wird Grammatik genannt. *Und* Verknüpfungen deuten an das mehrere Tokens in Reihe überein-

stimmen müssen, *Oder* Verknüpfungen bieten alternative Pfade an, trifft eine Regel nicht zu wird die nächste versucht. Jede Regel, die mit dem aktuell betrachteten Token übereinstimmt, wird in die Tiefe des Baums gefolgt. Sollte eine Regel in keinem Pfad zu einem Token passen, wird im Baum wieder nach oben gesprungen und von dort aus Alternativen durchlaufen. Dabei werden alle erfolgreich übereinstimmenden Regeln aufgezeichnet. Sollten bis zum Ende der syntaktischen Analyse alle Tokens verbraucht worden und der Regel-Baum wieder am Wurzelknoten sein, so bilden diese aufgezeichneten Knoten den AST oder CST des eingegebenen Textes und bestätigen das der eingegebene Text gültig ist. Wie Lexer, werden auch Parser oft von einem Tool erzeugt. Deren händische Erstellung ist ebenso möglich aber, je nach Komplexität der gewünschten Grammatik, ein sehr ambitioniertes Unterfangen.

Der für Myll implementierte Parser wurde mittels des ANTLR 4 Parser Generators erstellt. In jenem kann eine kontextfreie Grammatik in Erweiterter Backus-Naur-Form (EBNF) formuliert werden.

Anhand der folgenden stark reduzierten Grammatik werden die beispielhaft erzeugten Tokens aus dem Lexer Beispiel syntaktisch analysiert. Tokens werden hier komplett groß geschrieben, Regeln starten mit einem Kleinbuchstaben.

```
stmt      : ( RETURN expr SEMI
            | VAR typespec idVars SEMI );
typespec  : ( BOOL
            | FLOAT
            | INT );
idVars    : idVar (COMMA idVar)*;
idVar     : ID (ASSIGN expr)?;
expr      : ( expr PLUS expr
            | ID
            | INTEGER_LIT );
```

Der Token Strom des Lexer Beispiels

```
VAR | INT | ID( test ) | ASSIGN | INTEGER_LIT( 99 ) | SEMI
```

Der Einstiegspunkt in der Grammatik sei für dieses Beispiel die Regel `stmt` und das aktuelle Token ist `VAR`. Die Regel `stmt` wird betrachtet und enthält als erste Alternative `RETURN` (es ist eine Alternative weil die beiden Regeln mit einem *Oder* getrennt sind), dies stimmt nicht mit dem aktuellen Token überein also wird jetzt die Alternative betrachtet. Diese enthält `VAR` als erstes Element, was mit dem aktuellen Token übereinstimmt, das heißt der Token Strom springt zum nächsten Token. Zwischen `VAR` und `typespec` steht kein Operator, deswegen ist es eine implizite *Und* Verknüpfung. `typespec` ist eine weitere Regel und kein Token, deswegen wird sie zur weiteren Überprüfung betreten. In `typespec` wird jetzt geschaut ob ein Token mit `INT` übereinstimmt (natürlich der Reihe nach), und ja ist es: Strom einen weiter. Da `typespec` keine weiteren Regeln mehr enthält gilt es als erfolgreich erfüllt. Der Parse Tree / CST (welcher von ANTLR seit Version 4 erzeugt wird) sieht aktuell so aus:

```
stmt    → VAR
        → typespec → INT
```

Zurück im Ursprung in `stmt` steht `idVars` als nächste Regel an, diese wird betreten und es findet sich eine weitere Regel `idVar`, auch dort wird hineingesprungen. `ID` stimmt mit der `ID` aus dem Strom überein, der textuelle Inhalt des Token ist hier egal, die Regel ist erfüllt, Strom++. Dann kommt ein optionaler Anteil (zu erkennen am Fragezeichen als Suffix) welcher erfüllt sein kann, aber nicht muss. Es wird überprüft ob der nächste Token mit `ASSIGN` übereinstimmt, was er macht, Strom++. Der optionale Bereich ist aber noch nicht vorbei, d.h. die Regel `expr` wird betreten, welche wiederum Alternativen enthält. Die erste Regel enthält wieder `expr` als erste Regel. Aktuell sieht es anscheinend so aus als würde dies in einem ewigen Abstieg durch immer wieder der ersten (linken) Regel machen. Glücklicherweise hat ANTLR 4 genau für diese Problematik eine Sonderbehandlung, welche die direkte Rekursion der Regel ganz links erlaubt, welche bei vielen anderen Parsergeneratoren jetzt dazu geführt hätte das man seine Grammatik auf kom-

plett andere weise neu implementieren müsste. Siehe: Precedence climbing ???

([https://en.wikipedia.org/wiki/Operator-precedence\\_parser#Precedence\\_climbing\\_method](https://en.wikipedia.org/wiki/Operator-precedence_parser#Precedence_climbing_method))

Die Sonderbehandlung führte dazu das die erste Regel temporär übersprungen wurde und nun INTEGER\_LIT mit ID verglichen wird, was nicht übereinstimmt. Die nächste Alternative ist INTEGER\_LIT, ein Treffer also Strom++ und damit ist die optionale (ASSIGN expr)? Regel erfüllt, als auch idVar beendet. Jetzt steht eine optionale als auch wiederholbare Regel an (Erkennbar am Asterisk als Suffix), welche mit einem COMMA beginnen soll; hier könnten weitere Variablen vom gleichen Typ definiert werden. Diese Regel ist nicht erfüllt und idVars ist damit abgeschlossen. In stmt wird noch verlangt das noch ein SEMI folgt. Wäre dies nicht der Fall, wäre alle Arbeit verworfen worden und der Text als ungültig bewertet worden. Der komplette Parse Tree sieht am Ende so aus:

```

stmt    → VAR
        → typespec    → INT
        → idVars      → idVar    → ID( test )
                                   → ASSIGN
                                   → expr    → INTEGER_LIT
        → SEMI

```

??? Baumstruktur graphisch umsetzen, 90° im Uhrzeigersinn gedreht

??? PS: C ist nicht LL(1) Parsebar.

Nach dem Schritt der syntaktischen Analyse ist bekannt ob der eingegebene Text grammatisch valide ist, was noch nicht heißt das dessen Code so auch ausgeführt werden kann.

Die vollständige Auflistung der Grammatik des Parsers ist im Anhang ??? einzusehen.

## 4.5 Semantische Analyse

In der semantischen Analyse betrachtet die Bedeutung des Textes ... Allgemein???

Clang nennt die für die semantische Analyse genutzte Softwarekomponente Sema, dieser Namensgebung möchte ich mich hier anschließen.

Für die semantische Analyse gibt es keine Generatoren, zumindest nicht von ANTLR.

Da ANTLR anstelle eines AST einen CST erzeugt hat, wird dieser Schritt hier nachgeholt. Der Unterschied eines AST zu einem CST besteht darin das ein AST meist nur Blätter und Verzweigungen enthält, während der Baum eines CST viele unnötige Zwischenschritte beinhaltet. So sieht der AST des zuvor erzeugten Beispiels aus.

```

stmt    → VarDecl    → INT
                                   → ID( test )
                                   → INTEGER_LIT

```

??? Baumstruktur graphisch umsetzen, 90° im Uhrzeigersinn gedreht

VarDecl hat zwei verpflichtende und einen optionalen Kindsknoten: Typ, Name und opt. Initialisierung. Der Informationsgehalt des AST entspricht dem des CST, die Informationsdichte des AST ist höher.

Bei der Frage wie der CST durchlaufen werden soll, bot ANTLR zwei Möglichkeiten: *Visitor* (Besucher) und *Listener* (Lauscher). Die Entscheidung fiel auf den *Visitor*, da dort der Durchlauf des Syntax Baums selbst gesteuert werden kann; man kann Visit() auf die Kindersknoten in einer selbst gewählten Reihen-

folge aufrufen oder auch manche Teile gar nicht besuchen. Alternativ dazu hätte der *Listener* alle Knoten von selbst in vorgegebener Reihenfolge durchlaufen und einem lediglich die Möglichkeit gegeben dabei zuzuhören.

Die Semantische Analyse ist in Myll in drei Phasen unterteilt, die erste dieser Phasen findet schon beim Traversal des CST statt. Mehr???

Neu:

1. Durchlauf des CST, Erzeugung eines vorläufigen AST des kompletten Programms.  
Die Namen von global sichtbaren Deklarationen werden in einer Datenstruktur erfasst:
  - Namespaces
  - Typen Deklarationen (struct, class, enum, union, alias)
  - globale-, statische- und instanz-Variablen
  - Funktionen und Methoden

Alle importierten Module müssen diesen Schritt durchlaufen haben, bevor der nächste Schritt beginnen darf, weil sonst Typen womöglich falsch aufgelöst werden.

2. Alle globalen Namen sind bekannt, alle Referenzen auf Globale können jetzt zugeordnet werden.  
Diese Referenzen sind:
  - Using Namespace
  - Basisklassen
  - Typen der Variablen
  - Parameter und Rückgabetypen von Funktionen und Methoden

Die Zuordnung geschieht durch die Ersetzung der textuellen Representation durch einen Zeiger auf das aufgelöste Element.

Aus `MyUsingDecl { namespace = "std" }` wird `MyUsingDecl { namespace = &MyStdDecl }`

!!! Dieser Schritt ist nicht vollständig implementiert im aktuellen Prototypen

3. Funktionen, Methoden und Expressions können nun aufgelöst werden

??? HIER WEITER 16. Mai 2020

Was sind Module??? So etwas wie *includes*, aber Module eskalieren keine Imports.

- Mod b importiert Mod a
- Mod c importiert Mod b
- Mod a ist nicht in Mod c verfügbar

Dies ist gegensätzlich zu *includes* in C++, welche sämtliche *sub-includes* mitbringen.

Selbst wenn ich Schritt 2 noch nicht mache, weis ich dennoch was Pointer sind und was nicht und ich könnte einen Zugriff auf `Circle* c`; via `c.area()` erlauben und in `c->area()` übersetzen. (Siehe Go Lang)

In C++ gibt es nichts was die Syntax *c.irgendwas* auf einen Pointer sinnvoll nutzt.

1. Globale Typendeklarationen und Identifier erfassen
2. Globale Typen werden vervollständigt

### 3. Funktionskörper werden Analysiert

Was bedeutet 'fertig'??? erklären???

Beispiel

```
var int g_i;                                // (1) g_i
func myfunc() -> BC::Sub {                  // (2) myfunc
    var BC a;
    var BC::Sub b;
    a.c( b, g_i );
    return b;
}
class BC : SomeBase {                       // (3) BC
    class Sub {}                             // (4) BC::Sub
}
class SomeBase {                             // (5) SomeBase
    func c<T>( BC::Sub& b, T i ) {...}       // (6) SomeBase::c<T>
}
```

In diesem Beispiel sind sechs globale Deklarationen enthalten

Phase 1:

- Alle globalen Deklarationen werden hierarchisch durchlaufen
- Die globale Variable namens *g\_i* wird mit dem eingebauten Typen *int* erfasst, diese Deklaration ist damit schon vollständig aufgelöst und wird als fertig markiert
- *myfunc* wird global als Funktion erfasst mit *BC::Sub* als Rückgabety, da dieser bisher unbekannt ist, wird er aktuell nur textuell erfasst und die Implementierung von *myfunc* wird vorerst ignoriert
- *BC* wird global als Klasse erfasst, die Basisklasse *SomeBase* ist noch unbekannt und wird deswegen auch nur textuell erfasst
- *Sub* wird unter *BC* als Klasse erfasst, ist vollständig aufgelöst und wird als fertig markiert
- *SomeBase* wird global als Klasse erfasst
- Die Funktion *c* wird unter *SomeBase* mit Parametertyp *BC::Sub* textuell erfasst, *T* bleibt offen
- Implementierung von *SomeBase::c* wird vorerst ignoriert

Phase 2:

- Alle nicht fertigen globalen Deklarationen werden erneut hierarchisch durchlaufen
- Der Rückgabety von *myfunc* wird jetzt durch einen Verweis auf die in Phase 1 erfasste *BC::Sub* Klasse ersetzt und da ihre Signatur vollständig aufgelöst ist wird sie als fertig markiert
- Die Basisklasse von *BC* wird durch einen Verweis auf die Klasse *SomeBase* ersetzt, sie wird als fertig markiert, da hier nichts direkt auf die (bisher nicht fertige) Basisklasse *SomeBase* verweist
- Die Parametertyp *BC::Sub* von *SomeBase::c* wird aufgelöst, *T* bleibt offen, wird als fertig markiert
- Rückkehr zur Klasse *SomeBase*, diese wird als fertig markiert da alle Kinder fertig sind
- Phase 2 wird ein erneut durchlaufen sollte etwas nicht als fertig markiert sein
- Sollte die Anzahl der noch nicht fertigen Deklarationen nicht zwischen den erneuten Aufrufen schrumpfen, liegt ein fehlerhaftes Programm vor

Phase 3:

- Alle nicht-Template Funktionskörper werden aufgelöst
- Lokaler Scope von *myfunc* wird erzeugt
- Variable *a* vom Typ *BC* wird in lokalem Scope erzeugt
- Variable *b* vom Typ *BC::Sub* wird in lokalem Scope erzeugt

- Variable *a* ruft die Methode *c* auf, in dieser wird der Templateparameter *T* mit dem Typen *int* der globalen Variable *g\_i* instanziiert und der Typ der Variable *b* ist kompatibel mit dem ersten Parameter
- Variable *b* wird zurückgegeben, dessen Typ ist kompatibel mit dem Rückgabetyt der Funktion
- Der lokale Scope wird geschlossen
- Templatefunktionen wie *SomeBase::c* werden (wie in C++) nur im Falle der Nutzung aufgelöst. Im Falle ihrer Nutzung sind sie natürlich vollständig typisiert und werden wie normale Funktionen erzeugt.

### 4.5.1 Generator

Problem: Ich brauche eine spezialisierte String Klasse / Datenstruktur welche die effiziente Erzeugung von Output unterstützt.

Eckpunkte:

- Erzeugung von Iteratoren auf Positionen im Text, welche sich bei Einfügeoperationen mit verschieben und Reallokationen mitmachen
- Einfügen von Text an jeglicher Stelle der genannten Iteratoren, ohne das für jeden Aufruf alle Folgezeichen verschoben werden müssen
- Unterstützung von sehr großen Strings und mehrere dessen parallel

Was ich nicht brauche:

- Thread-Sicherheit

**DONE: Das ganze Programm überprüfen ob ich noch irgendwo NEW class variables benutzt hab!!!**

## 5 Auswertung & Zusammenfassung

### 5.1 Das sieht ja gar nicht mehr wie C++ aus!

Ja, das stimmt schon... Aber sieht denn C++11 noch so aus wie C++98?

Nein, siehe folgendes Beispiel:

```
VALIDIEREN!!!

// C++98
int * pi = new int(42);
delete pi;
typedef a b;

// Hier der Standard, verpönt in der Zukunft
// Freigeben nicht vergessen!
// Typalias

// C++11
unique_ptr<int> pi = make_unique<int>(42); // So soll es in C++11 aussehen, delete unnötig
using b = a;                               // Moderner Typalias

Mehr BSPL???
perfect forwarding, move semantics
oder einfach einen guten Artikel verlinken?
```

Wird C++20 noch so aussehen wie C++11?

Auch Nein: Der Standard ist schon abgeseegnet und einiges von diesem neuen Standard hat sich im Vergleich zum letzten Großen neun Jahre vorher verändert. Es gibt schon Compiler welche viele Funktionen des neuen Standards unterstützen.

Das Feature `constexpr` hat schon viel TMP unnötig gemacht und wird dies Dank der Einführung von `constexpr` fortsetzen. Man kann damit beispielsweise Funktionen schreiben welche zur Compiletime ausgeführt werden, die selben aber auch zur Runtime verwenden. Kein

Concepts werden die am schwersten zu durchschauenden Template-Parametrierungs-Fehler durch Sinnvolle Fehlermeldungen ersetzen und einige schwer zu schreibende SFINAE TMP Konstruktionen unnötig machen.

Der Spaceship Operator `<=>` wird einem die Arbeit abnehmen die meisten Vergleichsoperatoren zu implementieren. Er verhält sich relativ analog zu `strcmp`.

Es ist auch gut das man vom einen C++ Standard zum anderen eine Veränderung sehen kann, sonst hätte sich ja auch nicht viel verbessern können.

Genau das versuche ich auch zu bezwecken, Myll soll *Vertraut* aber dennoch *Anders* sein. Vielleicht in eine andere Himmelsrichtung Anders als die kommenden C++ Standards, aber sonst wäre Myll ja auch sinnlos.

#### Known issues

(Partial) Template Specialization might not work, thus TMP might not work as well

Keine Lambdas ATM

Es wird Code geben der von Myll nach C++ kompiliert, dann aber beim kompilieren des C++ Codes scheitert. Aufgrund von unvorhergesehener Namenskollision oder ähnlichem. Wenn man Myll in dieser Form scheitern lassen will, dann schafft man dies auch.

Letter to:

Bjarne Stroustrup,

Tell what I do and why I ask.

I know that you do not like C# because of its proprietary nature. Thus I am only interested in your opinion on just C#'s Syntax. Especially in the light of your sentence: Cleaner Language...

I personally think that C#'s syntax for defining template classes and functions is almost always as expressive as C++'s and way easier to use. Can you relate to my gut feeling?

A. Heijlsberg,

Tell what I do and why I ask.

Do I even have Questions?

A. Alexandrescu,

I recently read an Answer on Quora concerning the DLanguage

Herb Sutter

Parameter passing, sink, passing shared, unique and raw pointers

Per Vognsen,

Jonathan Blow:

NO NAGGING stuff...

You don't like the name 'Vector' for auto growing arrays, what would be your choice if you would not want to use purely 'Array'. C# calls these 'List' and Java 'ArrayList', do you think these names are good or do you have a better idea how to call them?

Read the surroundings of the quotes.

## 5.2 Performance Benchmark

Ruby Script um die Benchmark-Datei zu erzeugen:

```
f=File.read("stack_proto.myll");16.times{|n|fo=File.open "stack_big_"+n.to_s+".myll","w";fo.write "module
stack_big_"+n.modulo(4).to_s+"\n";500.times{|i|fo.write "namespace Rethmann::Uebung_"+i.to_s+"\n";fo.write f}; fo.close}
```

Parallelisierbarkeit

==== [#IN](#) =====

Genereller Scope von Myll:

Einführung geplanter C++17/20 features ahead of time.

Einführung nicht von ISO C++ geplanter Features.

Änderung alter C Syntax, Modernisierung auf aktuelles C++ Verhalten. (Typen, Deklarationen)

Einführung von Funktionalität welche nur im Transpiler existiert, für warnings oder errors verwendet wird, im Endeffekt aber keinen C++ Code erzeugt.

Vielfache Nutzung von static für unterschiedlichste Zwecke

Beschränkung der Nutzung und ersetzen durch Alternativen, Bestenfalls als Attribut, kollidiert nicht mit der Sprache.

Different operator precedence table

Boolsche Operatoren sind schlecht einsortiert:  $a \& 1 == 0$  ist  $a \& (1 == 0)$  sollte  $(a \& 1) == 0$  sein.

Idee von /PV.

Neue Operatoren, von C# und selbst Ausgedachte.



All variables initialized by default, uninitialized must be specified.

Idee von /JB

Default relational ops and compound assignment

Overrideable

Header and Implementation in a single file

Assignment is a Statement where left hand side Type communicates with the right hand side

Wirkt effektiv gegen `if(a=1)`, macht anderen code auch wesentlich schwerer obfuszierbar.

Enums can be linear or quadratic auto-indexed (flags enum)

Enums have reflection back to their string representation, oder andersherum

Leichter zu debuggen oder configs einzulesen.

Easier safe cast Expressions

Reuse the easy cast syntax from C, but this is only `static_cast`, others equally simple

Easier use of Smartpointers

`int @! a; für unique_ptr<int> a;`

Easier Template syntax

C# zeigt das es auch anders geht.

Concepts Lite Lite

Make Concepts available for all the easy cases

Spaceship Operator

UFCS

Nur in einer Richtung: aus `foo(a)` wird das hier nutzbar `a.foo()`

Accessors

Getter/Setter incl. dem syntaktischen zucker welcher sie ohne `()` aufrufbar macht.

Implicit `ret var`

Functions/Procedures liefern automatisch eine Variable die `ret` heißt und dem Rückgabetyt entspricht.

Enum methods

## 6 Ausblick

Everything is constexpr, when it can be, you don't need to write it.

### 6.1 Intelligenter Parameter Übername

Der Programmierer muss im allgemeinen mehr Gehirnschmalz in die Gestaltung der Übernahmeparameter stecken als man Naïv annimmt.

IDEA: Jeder Typ spezifiziert zu Compiletime welche Kosten ein Copy, ein Move oder ein Referenzieren hat. Auch für Literals des Typen

Als Beispiel dient eine Funktion welche nur wenige Parameter lesen will:

```
func(  int          a,
      vector<int>    b,
      shared_ptr<image> c,
      unique_ptr<database> d
) -> void { ... }
```

An welchen Stellen treten hier Probleme auf und welche Möglichkeiten bieten sich diese zu unterbinden?

- Dieser Parameter ist unproblematisch. Hier ließe sich lediglich noch ein *const* hinzufügen, was an der Korrektheit nach Außen nichts ändert, es wirkt sich lediglich als nur-lesen Einschränkung nach innen aus.
- Dieser Parameter hat in dieser Form ein potentiell performance Problem, da der Vektor mit seinem kompletten Inhalt kopiert wird. Mir fällt nur ein guter Grund ein dies so zu schreiben und zwar wenn man den Vektor innerhalb der Funktion ändern will, sich diese Änderung aber nicht nach aussen hin auswirken soll.
- Höchstwahrscheinlich passiert hier das Richtige, es ist im Falle des nur-lesens lediglich ein überflüssiges inkrementieren und dekrementieren des usage-counters möglich.
- Dies ist sehr wahrscheinlich ein Fehler, es sei denn man wollte wirklich das die Funktion die Datenbank konsumiert, sprich das übergebene Argument invalidiert.

Bauen wir die Parameter so um das sie wahrscheinlich das Richtige tun.

```
func(  const int ODER int ODER int& ODER int*          a,
      const vector<int>& ODER vector<int>& ODER const vector<int>* ODER vector<int>*  b,
      const shared_ptr<image> ODER const shared_ptr<image>& ODER const shared_ptr<image>*
      shared_ptr<image> ODER shared_ptr<image>& ODER shared_ptr<image>*          c,
      unique_ptr<database>& ODER unique_ptr<database>* ODER database*          d
) -> void { ... }
```

Dieses Feature lässt sich möglicherweise mit TMP lösen, wird dann aber wahrscheinlich eine verbose Syntax haben.

Nein, denn es ließe sich mit viel SFINAE Magic maximal die Signatur der Funktion ändern, die Position des Aufrufs bleibt unangetastet... außer es soll zusätzlich auch um jedes Argument aller Call Sites gewrapt werden, weil wenn z.B. in der decl aus *T -> T\** wird muss an der call site aus *arg -> &arg* werden.

```
template <typename T>
void foo(typename look_t<T>::type t) {...}
```

```
...
foo(look(argument));
```

## 6.2 Ranges

Läuft das so?

```
int sum = container | filter | sort | reduce;
```

## 6.3 Rest

==== [#MAYBE](#) ====

- ‘maybe\_const’, is an Aspect of a Function which will make it implement a const and a non-const version of itself. It is a
- Named function arguments, makes refactoring easier and calls like this easier to read: DoSomething(false) -> DoSomething(checkBefore:false).  
For bools even this syntactic sugar might be possible: DoSomething(!checkBefore), DoSomething(NOTcheckBefore), DoSomething(checkBefore), DoSomething(DOcheckBefore)
- Glue Classes (with UFCS?)
  - Classes that tie two or more classes together and only exist when both(all) classes are loaded
- Multithreaded compiler
- Be configurable for the user e.g. “@!” can be std::unique\_ptr or a custom pointer class
- Enable aspect oriented programming through Attributes
- Warnings for many holes in Structs due to alignment/bad ordering
- Reimagined Parameter definitions (lend, sink)
  - Copy:
  - Look:
  - Edi
- UTF8 in strings, 32bit Codepoint for single chars

==== [#OUT](#) ====

Use ifs everywhere as alternative to #ifdefs

Dynamic size for integer types: i23 for a 23 bit integer

Do not piggyback on C++ anymore