Objektorientierung
und
die Programmiersprache C++
Grundlagen und fortgeschrittene Konzepte

Gerd Hirsch
Revision: 1020

8. Oktober 2018

Inhaltsverzeichnis

Teil I

Einführung

1 Script

1.1 Ziele

Ziel dieses Scripts ist die Zusammenführung der wichtigsten Aussagen, Techniken, Fallstricke, usw. bzgl. C++ und der dazugehörigen Literaturverweise zur weiteren Vertiefung der Themen.

Es ist als Workshop begleitendes Script und nicht zum Selbststudium konzipiert.

Beim Lesen sollte eine Entwicklungsumgebung für C++ (z.B. Eclipse/CDT) zur Verfügung stehen, so dass die Beispiele sofort nachvollzogen werden können. Die Übungen stehen als Eclipse Workspace zum Experimentieren zur Verfügung.

1.2 Inhalt

Mit jeder weiteren Erfahrung aus den Projekten und Workshops wird dieses Script erweitert und angepasst. Für C++ liegt seit 2011/2014/2017 ein neuer Standard vor. Weitere sind geplant[1] Die für die C++ Seminare notwendigen Teile werden ebenfalls nach und nach in dieses Sript eingearbeitet.

Im zweiten Teil wird die Programmiersprache C++ vorgestellt. Die Teile III bis VI gehen auf Templates und die damit möglichen Programmiertechniken ein.

Bei Bedarf wird ein Teil Modellierung eingeschoben, in dem auf die notwendigen objektorientierten Grundlagen, die UML und auf objektorientiertes Design eingegangen wird.

Der Teil VII enthält kleinere Übungen zu C++ und eine durchgängige Übung zur Modellierung und Programmierung in Deutsch und bei Bedarf in englischer Sprache aus den Seminaren „Objektorientierte Analyse und Design mit der UML (OOAD)“ und „Objektorientiertes Design und Design Pattern (OODP)“.

Abgerundet wird das Script mit einem Anhang mit den üblichen Verzeichnissen und Literaturhinweisen und einem nicht perfekten Index.

Im Titel des Scripts wird die Revision und das Datum der letzten Änderung angegeben. Die Revisionsnummer wird nur bei größeren Änderungen, z.B. ein neues Kapitel, geändert, sie bleibt bei kleineren Änderungen und Korrekturen gleich.

Das Script lebt vom Feedback der Teilnehmer! Wenn Sie Fehler irgendwelcher Art entdecken oder Vorschläge zur Veränderung, Erweiterung des Scripts haben, senden Sie mir bitte eine E-Mail mit einem Hinweis.

[1] section 2.3 auf Seite 16

1.3 Konventionen

Folgende Textformate werden in diesem Script verwendet:

- wichtige Begriffe **Open Closed Principle**
- Sourcecode `#include <header.h> int i = 0;`
- Optionale Parameter in eckigen Klammern `[inline] [const]` ausser wenn eckige Klammern Bestandteil der Syntax sind

1.4 Links and downloads

Dieses Script ist unter http://www.gerdhirsch.de/downloads/OO_CPP_Schulung.pdf verfügbar.
Die Beispiele sind teilweise unter https://github.com/GerdHirsch/Cpp-Basics und in verschiedenen anderen Repositories verfügbar. Sie werden nach und nach aktualisiert.

Wenn Sie Fehler entdecken oder Verbesserungs- oder Erweiterungsvorschläge haben, senden Sie diese bitte an meine E-Mail Adresse.

2 Der C++ Standard

2.1 Der C++ Standard und TR1[2]

Der C++ Standard wurde 1998 verabschiedet. In 2003 wurde er durch einige geringe „Bug-Fixes“ erneuert.

Eine neue Version des Standards wurde 2011 C++11 verabschiedet der 2014 einer Revision (C++14) unterzogen wurde. Dieses Script bezieht sich auf den Standard C++03, die neuen Features sind mit C++14 gekennzeichnet. Auf die Unterschiede zwischen C++11 und C++14 wird in diesem Script nur in geringem Maße eingegangen.
Die Kernsprache und die Standard Bibliothek wurden gleichzeitig entwickelt. Die Bibliothek motiviert und profitiert von neuen Sprachfeatures und ist gleichzeitig der Prüfstein für diese. Die Erfahrungen mit der Sprache aus den Bibliotheken fließen wieder zurück in den Kern der Sprache.
Alle Namen der C++ Standardbibliothek sind im **namespace std** definiert. Die wichtigsten Bereiche des Standards sind:

- Die Standard Template Library (**STL**), mit den Containern, Iteratoren, Algorithmen und Funktionsobjekten
- Iostreams, mit Unterstützung für Benutzer definiertem buffering, internationalisiertem IO und den vordefinierten Objekten cin, cout, cerr und clog

[2] [Mey06a] Item 54

- Unterstützung für Internationalisierung, Arbeiten mit Unicode Zeichen
- Unterstützung für numerische Verarbeitung, Templates für komplexe Zahlen (complex) und Arrays von Werten(valarray)
- Eine Exception Hierarchie mit der Basisklasse exception und den davon abgeleiteten Klassen logic_error und runtime_error u.v.m.
- Die C89 Standard Bibliothek. Alles was im C89 Standard verabschiedet wurde ist Bestandteil des C++ Standards

Technical Report 1 (**TR1**) der C++ Library Working Group. Der zukünftige C++ (C++11) Standard hat einige neue sprachliche Eigenschaften definiert. Einige der Neuerungen sind durch den TR1 bekannt. Diese sind im **namespace std::tr1** zu finden. Es war zu erwarten, dass sich hier keine großen Änderungen mehr ergeben, auch wenn sich das Standardisierungskomitee das Recht vorbehielt, noch Änderungen vorzunehmen. Die wichtigsten Änderungen werden in einem eigenen Kapitel behandelt.

2.2 Der C++11/14 Standard[3]

2011 wurde der neue Standard verabschiedet. Die meisten Bibliotheken aus TR1 wurden in den Standard aufgenommen. Die weitere Entwicklung von C++ wird möglicherweise in TR2 beschrieben und zukünftig im `namespace std::tr2` zu finden sein.

Ein wichtiges Ziel des Designs des neuen Standards war die Source Code Kompatibilität zum alten Standard (keine binary Kompatibilität). Allerdings wurden neue Schlüsselworte eingeführt und Identifier mit diesen Zeichenketten sind dadurch ungültig.

2.2.1 Neue Features in C++11/14

Mit C++11 wurden **`throw`** Specifications `deprecated` erklärt und durch das Schlüsselwort `noexcept` ersetzt für Callable Entities die garantiert keine Exception werfen.

- auto (type deduction)
- Lambda-Expressions
- move Semantik und R-Value Referenzen
- perfect `forward<..>()` und `move(..)`
- Range-basierte For-Schleife
- Vereinheitlichte Initialisierung und `std::initializer_list<>`
- Multithreading und Memorymodell

[3] [Jos12]

- Smart pointers
- `nullptr` und `std::nullptr_t`
- constexpr
- bessere Untersützung für Metaprogrammierung, compile-time assertion
- Variadic Templates und Alias Templates
- Scoped Enums
- override, final
- decltype
- Benutzer definierte Literal Suffixe

2.2.2 Neue Bibliotheken in C++11/14

- thread
- Reguläre Ausdrücke
- Type-Traits
- Zufallszahlen
- Zeitbibliothek
- Reference Wrapper
- Smart Pointer: `unique_ptr, shared_ptr, weak_ptr`
- Container: Tupel, Array, einfach verkettete Listen, Hashtable
- bind und function
-

2.3 Der C++17 Standard

Die Entwicklung von C++ wird seit 2012 asynchron in sogenannten Technical Specifications[4] (TS) betrieben und veröffentlicht. Auf der ISO Website[5] kann der aktuelle Status eingesehen werden.

2.3.1 Neue Features in C++17

Eine gute Aufzählung der bisher bekannten verfügbaren Features ist von Bartlomiej Filipek [6]. Er unterteilt die Änderungen grob in folgende Kategorien[7]:

[4] https://www.iso.org/deliverables-all.html
[5] https://isocpp.org/std/status
[6] http://www.bfilipek.com/2017/01/cpp17features.html
[7] https://isocpp.org/blog/2017/12/summary-of-cpp17-features

- Fixes and deprecation
- Language clarification
- Templates
- Attributes
- Simplification
- Library changes - Filesystem
- Library changes - Parallel STL
- Library changes - Utils

Teil II

Die Programmiersprache C++

Eine objektorientierte Programmiersprache ist ein Werkzeug zur Umsetzung objektorientierter Konzepte und stellt dafür eine umfangreiche Menge sprachlicher Mittel zur Verfügung. Die Auswahl des „richtigen“ sprachlichen Mittels ist auch für fortgeschrittene Programmierer häufig eine Herausforderung. Die Grundlage für eine gute Auswahl sind zwei Dinge. Zum Einen, was soll zum Ausdruck gebracht werden, welche Idee soll objektorientiert umgesetzt werden. Zum Anderen, die genaue Kenntnis der jeweiligen Eigenheiten des ausgewählten sprachlichen Mittels, der möglichen Alternativen und der Konsequenzen, die sich aus der Auswahl ergeben. Java ist eine Programmiersprache, die mehr als 10 Jahre Entwicklungsgeschichte hinter sich hat. Ursprünglich konzipiert als plattformübergreifende Sprache zur Entwicklung von Applets, ist sie in der Zwischenzeit herangereift zu einer eigenen Plattform für Anwendungen verschiedenster Art. C++ ist eine Programmiersprache, die mehr als 25 Jahre Entwicklungsgeschichte hinter sich hat (2008). Bjarne Stroustrup[8] und Scott Meyers[9] empfehlen, die Sprache als Multiparadigmen-Sprache zu betrachten, um mit der Vielzahl der Sprachelemente besser umgehen zu können.

C++ unterstützt folgende Programmierstile und die Kombination dieser Stile:

Prozedurales Programmieren (C-Style): „Entscheiden Sie, welche Prozeduren Sie benötigen. Verwenden Sie den besten Algorithmus, den sie finden können.“ C++ ist ein besseres C, die Effizienzvorteile von C werden durch größere Typsicherheit erweitert

Modulares Programmieren: „Entscheiden Sie, welche Module Sie haben wollen. Unterteilen Sie das Programm so, dass die Daten in Modulen gekapselt sind.“

Programmieren mit benutzerdefinierten Typen: „Entscheiden Sie, welche Typen Sie benötigen. Erstellen Sie für jeden Typ einen vollen Satz an Operationen.“

Objektorientiertes Programmieren: „Entscheiden Sie, welche Klassen Sie brauchen. Erstellen sie für jede Klasse einen vollständigen Satz an Operationen. Verdeutlichen Sie Gemeinsamkeiten durch den Einsatz von Vererbung.“

Generisches Programmieren: Entscheiden Sie, welche Algorithmen Sie benötigen. Parametrisieren Sie sie so, dass sie für eine Vielzahl von geeigneten Typen und Datenstrukturen arbeiten.

Funktionale Programmierung: Auswertung von Templates zur Compile Time zur Berechnung von Konstanten und Typen

Einige Sprachelemente werden wir kennen und anwenden lernen, einige werden wir nur kurz skizzieren und einige davon nicht betrachten können.

[8] [Str00a] 2.3 bis 2.6
[9] [Mey06a]

3 Terminologie und Konventionen[10]

Zum gemeinsamen Verständnis wird hier das Vokabular, das jeder C++ Programmierer verstehen sollte, zusammengefasst. Bei der ersten Verwendung habe ich versucht jeweils den deutschen Begriff mit Schrägstrich dazu zu schreiben. Bsp.: Statement / Anweisung. Die Features des neuen Standards, soweit sie bereits eingearbeitet sind, sind mit C++11 gekennzeichnet.

3.1 Allgemeines

In diesem Script verwende ich Objekt und Variable sowie Klasse und Typ synonym. Viele Namen für Parameter und Argumente habe ich aus der Literatur und beim Lesen von Code von guten Programmierern übernommen. Widget ist ein häufig benutzter Klassenname ohne Bedeutung. Client und Server verwende ich wie im Script Einführung in die Objektorientierung beschrieben ist. Sourcecode wird wie folgt formatiert:

```
int i = 0;
```

lhs und **rhs** stehen für left-hand-side und right-hand-side bei Parametern für binäre Operatoren. Werden Operatoren als Elemente der Klasse, also als Member implementiert, wird der linke Operand durch den `this` Pointer repräsentiert[11].

Listing 1: Namen von Operanden

```
1 bool operator==(const Widget& lhs, const Widget& rhs);
2
3 Widget w1, w2;
4
5 if(w1 == w2) ... // lhs wird mit w1 initialisiert, rhs mit w2
```

In Kommentaren verwende ich **ctor** und **dtor** für Konstruktor und Destruktor als Abkürzung.

3.2 Keywords C++[12] und lexikalische Konventionen

Die Programmiersprache C++ ist formlos bis auf drei Ausnahmen[13]

- Zeilen-Kommentare[14]
- Präprozessor Direktive
- Zeichenketten-Konstanten

[10] [Mey06a] Introduction
[11] section 6.25.3 auf Seite 85
[12] Siehe http://en.cppreference.com/w/cpp/keyword
[13] http://www.wilkening-online.de/tutorial-c++.html
[14] siehe section 3.3 auf Seite 21

Die Schlüsselworte von C++11 sind in der Tabelle 1 gelistet.

Die Bedeutung des Schlüsselworts `auto` hat sich seit C++11 geändert. Siehe section 3.5.2 auf Seite 22. Zusätzlich zu den Schlüsselworten wurden die zwei identifier `override` und **`final`** mit spezieller Bedeutung hinzugefügt. Siehe section 6.32.9 auf Seite 133.

3.2.1 Vorschriften und Konventionen zur Bildung von Bezeichnern

Identifier / Bezeichner müssen mit einem Buchstaben oder einem Unterstrich beginnen. Dannach dürfen Zahlen und der Unterstrich verwendet werden: `[a-z,A-Z,_][a-z,A-Z,0-9,_]*`. Identifier dürfen nicht mit den Zeichenketten von Schlüsselworten kollidieren. Namen mit doppeltem vorangestellten Unterstrich, gefolgt von einem Großbuchstaben, sowie alle globalen Bezeichner mit vorangestelltem Unterstrich, sind für die Implementierung der Sprache reserviert.
Verwenden Sie sprechende Namen, keine Abkürzungen. Bilden Sie das Vokabular der Problem Domain in Ihren Systemen ab! Verwenden Sie keine Literale in Ihrem Code, außer zur Initialisierung von Konstanten: **No Magic Numbers!**

Namen von eigenen Klassen und Templates sollten mit einem Großbuchstaben beginnen. Operationen und Variablen mit einem Kleinbuchstaben. CamelCase[15] erhöht die Lesbarkeit.

Tabelle 1: Schlüsselworte in C++

`alignas` (C++11)	`alignof` (C++11)	`asm`
`auto`[16]	`bool`	**`break`**
`case`	**`catch`**	**`char`**
`char16_t` (C++11)	`char32_t` (C++11)	**`class`**
`const`	`constexpr` (C++11)	`const_cast`
`continue`	`decltype` (C++11)	**`default`**[17]
`delete`[17]	**`do`**	**`double`**
`dynamic_cast`	**`else`**	`enum`
`explicit`	`export`[18]	`extern`
`false`	**`float`**	**`for`**
`friend`	**`goto`**	**`if`**
`inline`	**`int`**	**`long`**
`mutable`	`namespace`	**`new`**
`noexcept` (C++11)	`nullptr` (C++11)	`operator`
`private`	**`protected`**	**`public`**
`register`	`reinterpret_cast`	**`return`**
`short`	`signed`	`sizeof`
`static`	`static_assert` (C++11)	`static_cast`
`struct`	**`switch`**	`template`

[15] http://en.wikipedia.org/wiki/CamelCase
[16] veränderte Bedeutung siehe section 3.5.2 auf Seite 22
[17] erweiterte Bedeutung siehe section 6.6 auf Seite 40
[18] für non-inline Templates, siehe section 8 auf Seite 167

this	thread_local (C++11)	**throw**
true	**try**	typedef
typeid	typename	union
unsigned	using[19]	virtual
void	**volatile**	wchar_t
while		

Tabelle 2: Alternative Namen für Operatoren

and	&&	and_eq	&=	bitand	&	bitor	|
compl	~	not	!	not_eq	!=	or	||
or_eq	|=	xor	^	xor_eq	^=		

3.3 Kommentare

Es gibt mehrzeilige /* */ und einzeilige // Kommentare.

Listing 2: Kommentare

```
1 /* Ein mehrzeiliger
2       Kommentar kann nicht geschachtelt werden
3 */
4 // Ein einzeiliger Kommentar endet am zeilen ende
```

3.4 Objekt, Klasse, Interface und Vererbung

Objekte siehe: section **??** auf Seite ?? und Vererbung siehe: section **??** auf Seite ??. Im Zusammenhang mit Vererbung bzw. multipler Vererbung sind die Begriffe **upcast, downcast** und **crosscast** von Bedeutung; siehe section 6.26 auf Seite 103.

3.5 Deklaration und Definition

3.5.1 Deklaration

Eine **Deklaration** / Declaration teilt dem Compiler Namen und Typ eines Objekts mit, aber sie lässt bestimmte Details weg.

- `extern int x;` Objektdeklaration, keine Speicherbelegung
- `std::size_t numDigits(int number);` Funktionsdeklaration, es gibt etwas, das numDigits heißt und eine Funktion ist, die einen int als Parameter erwartet. Der Aufruf der Funktion ist ein Ausdruck vom Typ `std:size_t`

[19] erweiterte Bedeutung siehe section 6.10 auf Seite 45

- `class Widget;` Klassendeklaration, es gibt etwas das Widget heißt und eine Klasse ist
- `template<typename T> class GraphNode;` template Deklaration, es gibt etwas das GraphNode heißt und ein Klassen Template ist
- `typedef unsigned int Word;` Word als Synonym für unsigned int

Eine Funktionsdeklaration zeigt die Signatur / Prototyp der Funktion. In der offiziellen C++ Spezifikation gehört der Rückgabetyp nicht dazu.

3.5.2 automatische Typerkennung mit `auto` (C++11)

Mit dem Schlüsselwort `auto` können Variablen ohne Angabe eines Typs deklariert werden, wenn der Typ durch die Initialisierung vom Compiler erkannt werden kann[20].

Listing 3: automatic type deduction with auto

```
1 auto i = 42; // i hat den Typ int
2 double f();
3 auto d = f(); // d hat den Typ double
4 auto a; // Error, Typ kann nicht erkannt werden
```

Weitere Qualifier sind möglich:
`static auto MwSt = 0.19;`
Der größte Nutzen von `auto` ergibt sich im Zusammenhang mit langen und komplizierten Typen wie z.B. in Listing 4 ist der Typ der Variablen `lambda`, ein lambda[21], das einen `int` als Parameter definiert und einen `bool` zurückliefert.

Listing 4: auto und komplexe Typen

```
1 vector<string> v;
2 ...
3 auto pos = v.begin(); // Typ: vector<string>::iterator
4
5 auto lambda = [](int x) -> bool{ return x < 0; }
```

3.5.3 Definition

Eine **Definition** stellt dem Compiler die Details zur Verfügung die die Deklaration weglässt. Eine Definition ist immer auch eine Deklaration. Es dürfen mehrere gleichlautende Deklarationen in einem Scope vorhanden sein, aber nur eine Definition (**One Definition Rule ODR**). Für Objektdefinitionen stellt der Compiler Speicherplatz zur Verfügung und verknüpft diesen mit dem Namen.

Grundsatz: Definieren Sie Objekte genau dann, wenn Sie dafür eine sinnvolle Initialisierung durchführen können.

[20] section 6.30 auf Seite 113
[21] siehe section 6.32.11 auf Seite 136

Für eine Funktion oder ein Funktionstemplate stellt die Definition den Funktionskörper, also den zu übersetzenden Code zur Verfügung. Für eine Klasse oder ein Klassentemplate listet die Definition die Elemente / Member der Klasse auf.

- `int x;` Mit dem Namen x wird Speicher verbunden, **nicht initialisiert**
- `int x = 3;` wie oben aber mit Initialisierung
- `std::size_t numDigits(int number){/*Funktionskörper*/}`
- `class Widget {/*Klassendefinition*/};`
- `template<typename T> class GraphNode{/*template Definition*/};`

3.6 Initialisierung und Zuweisung

Initialisierung ist die erste Belegung des Speicherplatzes, der mit dem Objekt verbunden ist, mit einem Wert. Die **Initialisierung** wird durch einen Konstruktor durchgeführt. Bei der **Zuweisung** wird einem bereits existierenden Objekt durch den copy assignment operator (=) ein neuer Wert zugewiesen. Siehe section 6.16 auf Seite 50

3.7 Bereich / Scope, Sichtbarkeit und Lebensdauer

Dieses Kapitel soll einen allgemeinen Überblick über die Sichtbarkeit und die Lebensdauer von Namen und Objekten geben. Die Details werden bei den spezifischen Programmiertechniken besprochen.

Der Bereich / scope, in dem ein Name/Identifier deklariert wird, bestimmt seine **Sichtbarkeit**. Namen können global oder modul global (innerhalb einer Übersetzungseinheit `static`), innerhalb einer Funktion (eines Blocks)[22], innerhalb eines Namespace[23], innerhalb einer Klasse[24] oder innerhalb einer scoped Enumeration[25] deklariert werden. Die Bereiche sind ineinander eingebettet, der äußerste Bereich ist der global scope. Namen in äußeren Bereichen sind in den inneren Bereichen sichtbar. Werden Namen **global** deklariert, sind sie überall sichtbar (global scope) oder nur innerhalb der Übersetzungseinheit (Speicherklasse `static`).

Gleichlautende **Namen** in inneren Bereichen überdecken **alle Namen** aus äußeren Bereichen. Soll auf Namen aus dem global scope, die überdeckt sind wie in Listing 5 auf der nächsten Seite, zugegriffen werden, muss der Scoperesolution Operator angewendet werden: `::name`

[22] siehe section 6.21 auf Seite 62
[23] siehe section 6.18 auf Seite 55
[24] siehe section 6.20.1 auf Seite 60
[25] siehe section 6.19 auf Seite 59

Listing 5: Namensüberdeckung

```
1  int x = 0; // global
2
3  namespace ENS{ // ExampleNameSpace
4      int x = 1;
5  }
6
7  void f(){
8     int x = 1; //lokales x, Sichtbar innerhalb von f() überdeckt globales x
9     {
10       // überdeckt erstes lokales x auf das nicht referenziert werden kann
11       int x = 2;
12       ::x = 3; // Zuweisung an das globale x
13       ENS::x = 4; // Zuweisung an x im namespace ENS
14    }
15 }
```

Sollen Namen aus einem Namespace außerhalb verwendet werden, müssen sie entweder mit dem Namen des `namespace` voll qualifiziert werden (`ENS::X`, `std::cout`) oder sie müssen über die using Klausel zugänglich gemacht werden.

`using namespace std;` macht alle Namen des namespace std im aktuellen scope sichtbar.

`using std::cout;` macht `cout` aus dem namespace std im aktuellen scope sichtbar.

Werden Namen **innerhalb einer Klasse** deklariert, sind sie in der Klasse sichtbar (class scope). Der Zugriff von außen hängt von den access specifiern (**`public`**, **`protected`**, **`private`**) ab[26]. In Vererbungshierarchien ist der scope der abgeleiteten Klasse in den scope der Basisklasse eingebettet.

Werden Namen **innerhalb einer Funktion** deklariert, sind sie nur dort sichtbar.

Der Ort, wo ein Objekt / Variable definiert wird und die Art, wie es definiert wird, bestimmen die **Lebensdauer** des Objekts.

Die Schlüsselworte **`static`**, **`class`**, `namespace` und das Compound Statement `{}` nehmen Einfluss auf Sichtbarkeit und die Lebensdauer des Namens/Objekts bei der Definition.

Die Operatoren: Scope Resolution `::`, ElementAuswahl `.` und `->` sowie die using Klausel nehmen Einfluss auf den Zugriff auf den Namen.

3.8 Anweisungen und Ausdrücke

Die konventionellen und **grundlegenden syntaktischen Elemente eines C++ Programmes sind Anweisungen und Ausdrücke**. Alles ist entweder eine Anweisung oder ein Ausdruck.

[26] siehe section **??** auf Seite ??

Anweisungen / **Statements** werden wegen ihrer Effekte bei der Ausführung durch den Rechner verwendet, sie haben keine Werte[27]. Ein objektorientiertes Programm besteht aus Objekten die sich Nachrichten senden, der Code der die Nachrichten versendet und verarbeitet, sind Ausdrucks-Anweisungen.

Ausdrücke / **Expressions** werden berechnet! [28] Das **Ergebnis** der Berechnung, der Wert, liegt in der Form eines Typs, dem **Ergebnistyp** der Operation oder des Operators vor. Alles was der Compiler zu sehen bekommt und nicht eine Anweisung ist, ist ein Ausdruck.

Ausdrücke können **R-Value** oder **L-Value** Ausdrücke sein. L-Value Ausdrücke haben einen Platz im Haupspeicher, eine **L**ocation. R-Value Ausdrücke befinden sich im **R**egister, sind also temporäre Objekte.

3.9 Funktion, Operation und Methode

Operationen sind in der Schnittstelle einer Klasse sichtbar. Eine Operation ist durch ihre Signatur eindeutig bestimmt. Die Methode ist der Funktionskörper der dieser Operation, aufgrund der übereinstimmenden Signatur, zugeordnet ist. Für eine polymorphe Operation (C++: `virtual`), können verschiedene Methoden in abgeleiteten Klassen durch Überschreiben zur Verfügung stehen. Welche Methode ausgewählt wird, ist vom Typ des Objekts, an das die Nachricht gesendet wird, abhängig[29].

Funktionen sind nicht an eine Klasse gebunden, sie sind meistens global sichtbar. Die default Speicherklasse ist `extern`, wird sie **`static`** deklariert, ist sie nur innerhalb der Übersetzungseinheit[30] sichtbar.

3.10 Parameter und Argument

Ein Parameter einer Funktion ist die Definition einer lokalen Variablen, die innerhalb der Funktion sichtbar ist und am Ende der Funktion ihre Gültigkeit verliert.

- `std::size_t numDigits(int number){/*Funktionskörper*/}`

number ist der **Parameter** vom Typ int. Lokalen Objekten wird automatisch Speicher auf dem Stack beim Funktionsaufruf zugeordnet.

Ein Argument ist das Objekt das beim Funktionsaufruf übergeben wird.

```
numDigits(5);
```

5 ist das **Argument** vom Typ int. Die Parameter werden beim Funktionsaufruf mit den korrespondierenden Argumenten initialisiert[31]. Dabei kommen für eingebaute

[27] siehe section 6.24 auf Seite 75
[28] siehe section 6.25.5 auf Seite 89
[29] siehe section **??** auf Seite ?? und section 6.32 auf Seite 125
[30] section 4 auf Seite 29
[31] siehe section 6.32 auf Seite 125

Datentypen die impliziten Typkonvertierungsregeln der Zuweisung zum tragen [32].

Funktionstemplates haben zwei Arten von Parametern:[33]

```
template<typename T> max(T const& a, T const& b){ return a < b ? b : a; }
```

wobei *T* der **template parameter** ist, *a* und *b* die **call parameter** sind, vom Typ `T const&`, wie bei nicht Template Funktionen. So wird auch zwischen **template argumenten** und **call argumenten** unterschieden. Template Argumente sind Typen, call Argumente sind Werte.

```
double rd = max<double>(3, 7.2F);
```

`double` ist das Template Argument, 3 und 7.2F sind die call Argumente.

Template Parameter können *Typen* oder Templates sein. Für Parameter die Templates sind wird der Begriff **template template parameter** verwendet, analog könnte man für Typen **template class parameter** verwenden. Außerdem können Templates sogenannte **nontype template parameter** haben[34].

3.11 Überladen und Überschreiben

Von „**überladen** / **overloading**“ oder einer überladenen Funktion sprechen wir, wenn für einen Funktionsnamen oder einen Operator mehrere Funktionskörper mit unterschiedlichen Parameterlisten zur Verfügung stehen. Der Compiler hat dann beim Aufruf der Funktion die Aufgabe, die Funktion auszuwählen, deren Parameter am besten mit den Aufrufargumenten passt (best match). Dieser Prozess wird Overload Resolution genannt. Ein überladen ausschließlich bezüglich des Rückgabetyps einer Funktion ist nicht möglich[35] , da ein Funktionsaufruf nach dem C++-Standard kontextunabhängig sein soll, d.h. der Aufruf einer Funktion mit den identischen Parametern in jedem Kontext zu dem gleichen Ergebnistyp führen muss.

Von „**überschreiben** / **overriding**“ sprechen wir im Zusammenhang mit Vererbung, wenn in einer abgeleiteten Klasse für eine `virtual` Operation der Basisklasse eine Methode mit identischer Signatur zur Verfügung gestellt wird.

Im Zusammenhang mit dynamischer Polymorphie, sollte das Überschreiben von *non-virtual* Operationen aus der Basisklasse durch geeignete QS Maßnahmen verhindert werden, da es technisch möglich, aber aus Design Sicht sinnlos ist. siehe section **??** auf Seite ?? .

Wird allerdings generisch mit Templates Code erzeugt, werden keine virtuellen Funktionen benötigt, da die jeweiligen konkreten Typen zur Verfügung stehen, die Namen in den abgeleiteten spezielleren Klassen, verbergen die Namen der Basisklasse.

[32] siehe section 6.25.5 auf Seite 90
[33] siehe section 10 auf Seite 197
[34] Siehe section 8.4 auf Seite 172
[35] [Här07] Kapitel 2.2

3.12 Function Overload Resolution und Template Argument Deduction [36]

Funktionen und Funktionstemplates können überladen werden. Es kann also mehrere Funktionen und gleichzeitig mehrere Funktionstemplates geben, die denselben Namen tragen, aber unterschiedliche Parameterlisten haben. Die Funktion mit den richtigen Parametern zu finden wird als **Overload Resolution** bezeichnet.

Argument Deduction: Wird nur bei Funktions**templates** durchgeführt. Wenn ein Funktionstemplate mit bestimmten **call Argumenten** aufgerufen wird, werden die **template Argumente** durch die Typen der call Argumente vom Compiler bestimmt. Implizite (automatische) Typkonvertierungen werden dabei nicht durchgeführt! Jeder template Parameter muss exakt übereinstimmen (exact match).

`double dr = max(3, 5);` // kein explizites Template Argument notwendig, das template-Argument ist `int`.

3.13 Templates und Instanzierung

Die Erzeugung einer Funktion oder einer Klasse aus einem template und die Ersetzung der template Parameter mit den konkreten Typen wird als **instantiation** bezeichnet.[37] Die konkrete Funktion oder Klasse als Instance des templates. Bei Funktionstemplates kommt der Prozess der **Argument Deduction** ins Spiel.

3.14 Design Pattern und Idiome

Ein Design Pattern (z.B. State Pattern) beschreibt eine bewährte Lösung für ein bestimmtes Entwurfsproblem abstrakt in einer Modellierungssprache, z.B. UML. Ein Idiom beschreibt eine bewährte Lösung eines Problems für eine bestimmte Programmiersprache, z.B. RAII[38] in C++.

3.15 Sonstiges

Die Standard Template Library (**STL**). Sie definiert einen mächtigen Satz an generischen Klassen (**template**) zur Verwaltung von Mengen (**container**). **Functor**s oder **function object**s sind Objekte von Klassen die den Funktionsaufrufoperator überladen. In der STL häufig als **Prädikate** verwendet. Für einige Konstrukte in C++ ist das Verhalten nicht definiert (**undefined behavior**), wie in Listing 6 auf der nächsten Seite

[36] [VJ03] Appendix B und 2.2 Argument Deduction
[37] [VJ03] 2.1.1 Using the Template
[38] siehe section 6.17 auf Seite 54

Listing 6: undefined behavior

```
1 int *p = 0;
2 std::cout << *p; // Einen null pointer zu dereferenzieren resultiert in undefined
      behavior
```

4 Der Entwicklungsprozess

Hier soll nur der „mikro“ Prozess des Programmierens vereinfacht vorgestellt werden.

4.1 Von der Analyse zum getesteten Modul

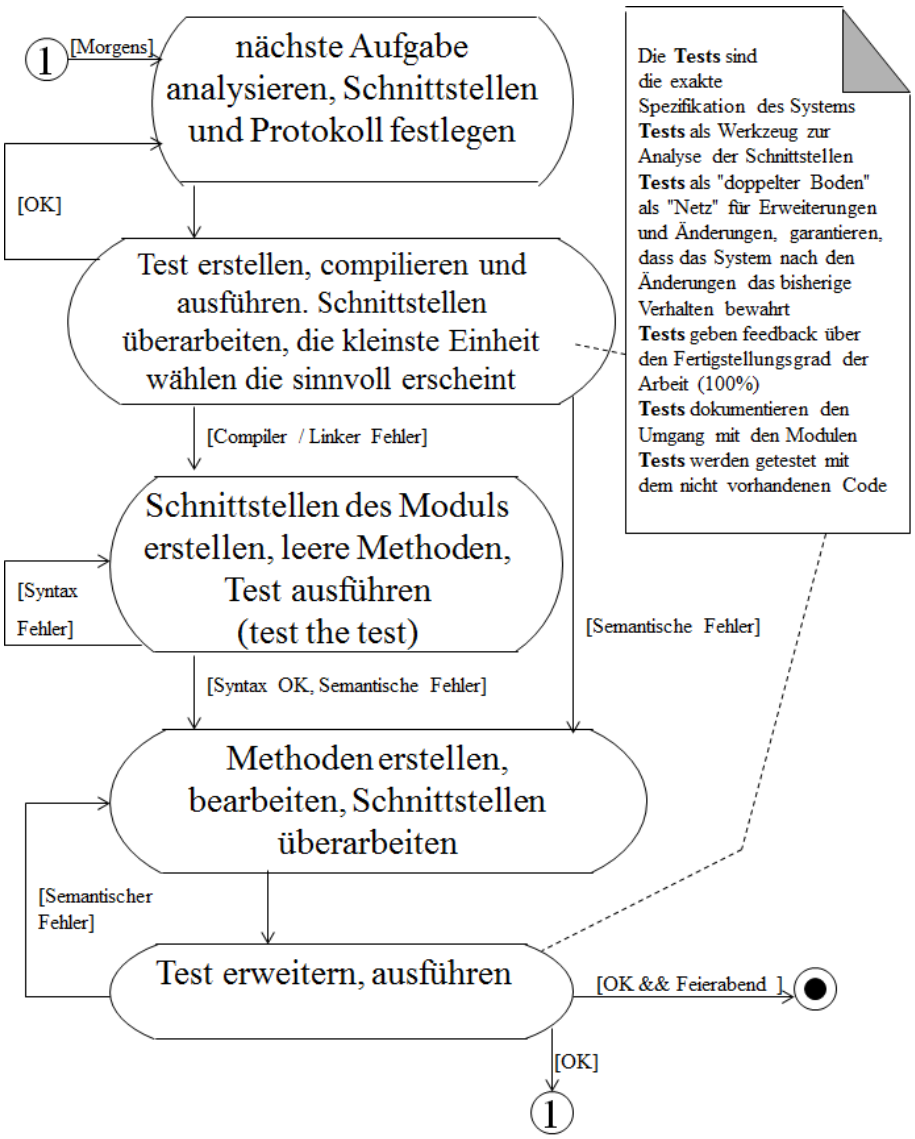

Diagramm 1: Von der Analyse zum getesteten Modul, UML Aktivitätsdiagramm

Das Diagramm 1 skizziert die testgetriebene Vorgehensweise zur Erstellung eines Moduls (Test driven development TDD[39]). Die Idee ist, zuerst Tests für ein Modul zu schreiben, **bevor** das Modul erstellt wird. Dadurch entstehen zwangsläufig isoliert testbare Module.

[39] [Ast03] und [Bec03]

4.2 Vom Editor zum ausgeführten Programm

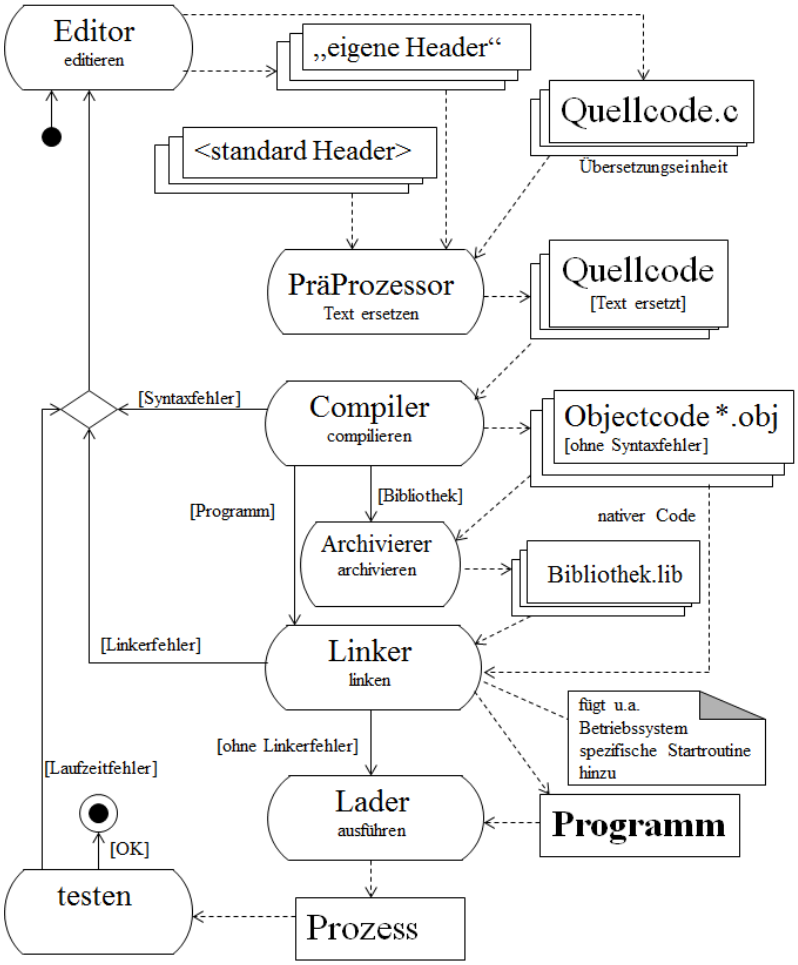

Diagramm 2: Die Toolkette: Vom Editor zum ausgeführten Programm. UML Aktivitätsdiagramm

Das Diagramm 2 zeigt die Begriffe, Aktivitäten und die dabei zum Einsatz kommenden Werkzeuge zur Erzeugung der verschiedenen Artefakte der Programmierung. Die Quellcode Dateien in C++ haben die Endung .cpp, in C ist es die Endung .c. Diese werden auch als **Translation Unit** / **Übersetzungseinheit** bezeichnet, da sie nach der Ergänzung durch den PräProzessor, dem Compiler zur Übersetzung übergeben werden. Korrekter wäre es, das Ergebnis des PräProzessors anstatt die Quellcode Dateien als Translation Unit zu bezeichnen. Das bekommt man aber nur mit der entsprechenden Compiler Option zu Gesicht.

5 Von C nach C++ migrieren

In diesem Kapitel werden die gemeinsamen Sprachkonstrukte und ihre Unterschiede kurz dargestellt. siehe http://www.caravan.net/ec2plus/guide.html und https://docs.oracle.com/cd/E19059-01/fortec6u2/806-7990/806-7990.pdf

5.1 Allgemeines

C++ wurde mit dem Ziel entworfen, die Effizienz von C beizubehalten und die Programmierung auf einem höheren Abstraktionsniveau zu ermöglichen.

Als Grundregel gilt, nur die Features der Sprache, die auch benutzt werden, finden Eingang in das Compilat, haben Einfluss auf die Größe des Programms.

C++ ist source-and-link kompatibel mit C, aber C ist keine Untermenge von C++. Es gibt C Code der „illformed“ C++ ist und C Code der von einem C++ Compiler übersetzt wird, aber in C++ eine andere Bedeutung hat. Eine umfangreiche Beschreibung solcher Abweichungen ist in „Compatibility of C and C++“[40] zu finden. Der aktuelle C Standard ist **ISO/IEC 9899:2011**[41]

5.2 Datentypen

Die elementaren Datentypen von C und C++ stimmen überein, in C++ kommt noch bool hinzu. Mit C++11 sind weitere Datentypen hinzugekommen[42].

5.2.1 char literal

`'A'` ist in C ein **`const int`**, in C++ ein **`const char`**! D.h. in C ist `sizeof('A')== sizeof(int)` in C++ ist `sizeof('A')== sizeof(char)== 1`. **`char`** hat in C++ immer die Größe 1 Byte !

5.2.2 Enumeration type

In C sind die Elemente einer Enumeration vom Typ int. Alles was sich mit einem Objekt vom Typ int machen lässt, ist auch mit einem enum Objekt möglich. Die C++ enums wurden mit C++11 um Scoped Enumerations erweitert, siehe section 6.19 auf Seite 59

[40] https://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B
[41] https://de.wikipedia.org/wiki/Varianten_der_Programmiersprache_C
[42] siehe section 6.22 auf Seite 69

5.3 Definitionen

5.3.1 Type Definitionen in Ausdrücken

In C können Typen in cast Ausdrücken, in Parameter Deklarationen oder in sizeof Ausdrücken definiert werden, das geht in C++ nicht.

```
void func(struct Name { char* name; } name){...}
```

5.3.2 Funktionsparameter

Nur in C möglich:

```
void f(a, b, c)int, int, int {...} //old style C
```

5.3.3 For

Listing 7: Unterschiede von for in C und C++

```
1 for(int i = 0; i < 3; ++i)
2    ;
3
4 If (i >= 4) // in C++ ist i hier nicht mehr gültig
5    ...
```

5.3.4 Definition überspringen

In C kann eine Definition mit einem goto oder switch Statement übersprungen werden, in C++ ist das nicht erlaubt.

```
goto LABEL:

  int v = 0;

LABEL:

...
```

5.3.5 char Array Initialisierung

```
char s[3] = "abc"; //in C erlaubt, in C++ nicht
```

`"abc"` ist mit der abschließenden 0, 4 char lang, s hat aber nur Platz für 3.

Um Zeichenketten zu verwalten sollte in C++ std::string oder std::vector verwendet werden, anstatt char Arrays.

5.4 Speicherklassen

Wie in C, gibt es auch in C++ die Speicherklassen, register, auto, static und extern. Die default Speicherklassen bei Deklarationen und Definitionen sind aber verschieden. Mit C++11 hat sich die Semantik des Schlüsselworts `auto` vollständig geändert[43]

5.4.1 Default Speicherklasse globaler Objekte

Gemeinsamkeiten: Eine Definition darf es nur einmal geben, die sogenannte „One Definition Rule (ODR“).

Unterschied:

Eine globale Variable in C++ hat extern linkage, die mit 0 initialisiert wird, wenn die Initialisierung fehlt.

In C ist die Deklaration ohne Initialisierung eine provisiorische Definition, folgt eine weitere Definition mit Initialisierung, wird die erste als Deklaration behandelt.

```
int i;

...

int i = 5; //in C++ doppelte Definition
```

5.4.2 const type qualifier

Speicherklasse von Objekten im file scope.

In C ist die default Speicherklasse eines globalen const Objekts extern (external linkage). In C++ ist die default Speicherklasse static (internal linkage), soll das gleiche Verhalten erzielt werden, muss der `extern` specifier verwendet werden.

5.5 Casts, Typkonvertierungen

In C wird ein cast durch die Angabe des gewünschten Typs in runden Klammern angegeben, das ist in C++ auch möglich, sollte aber vermieden werden. Statt des „C Style Casts“ sollten die C++ cast Operatoren `const_cast, static_cast, dynamic_cast und reinterpret_cast` verwendet werden, wie sie in section 6.26 auf Seite 103 beschrieben sind.

5.5.1 Konvertierung von void*

```
void* pV = malloc(n);
```

[43] section 3.5.2 auf Seite 22

```
T* pT = pV; //ohne cast nur in C möglich.

T* pT = reinterpret_cast<T*>(pV);
```

5.6 Dynamische Speicherverwaltung

In C existieren die Funktionen malloc/free und ihre Verwandten zur Anforderung von Heap Speicher.

In C++ wird Speicher für Objekte mit new angefordert und mit delete wieder freigegeben. Mischen der Funktionen führt zu undefiniertem Verhalten! Also Speicher mit malloc anfordern und mit delete freigeben oder umgekehrt.

5.7 struct

Structs werden in C++ wie Klassen behandelt. Der Unterschied zwischen struct und class ist der **default** access-spezifier. Bei Klassen ist der default access-specifier private, bei structs ist er public, ansonsten sind die beiden Sprachkonstrukte äquivalent.

5.8 Funktionsprototypen

In C++ müssen alle Funktionen vor ihrer ersten Verwendung durch eine Deklaration bekannt gemacht werden, der C Compiler trifft bestimmte Annahmen über eine nicht deklarierte Funktion, z.B. Rückgabetype int.

5.9 Linking to C Funktionen

C++ betreibt sogenanntes „name mangling“ zur Auflösung von überladenen Funktionen. Damit eine Funktion C Linkage bekommt, muss diese wie in Listing 8 deklariert werden.

Listing 8: C Linkage

```
1 extern "C" {
2    double sqrt(double);
3 }
```

5.10 Eingebettete Typen

In C sind eingebettete Typen im selben Scope wie der umgebende Typ, in C++ eröffnen Typen ihren eigenen Namensraum.

Listing 9: Eingebettete Typen

```
struct s {
   int a;
   struct T {...} t;
   ...
};

struct S s;

struct T t; // in C++ S::T t;
```

5.11 Namensräume

Sind neu in C++, sie dienen dazu Namenskonflikte zu vermeiden und können beliebig geschachtelt werden. Namespaces haben keine oder nur geringe Auswirkungen (durch die längeren Symbolnamen) auf den vom Compiler generierten Code.

Listing 10: Namensräume

```
namespace de {
   namespace gerdhirsch {
      void doIt();
   }
}

de::gerdhirsch::doIt(); // voll qualifizierter Name der Funktion doIt()
```

6 Basic Language Features

6.1 Der Program entry point

Ein Programm hat einen Anfang (Program-entry-point) und ein Ende. In C++ ist das die Funktion[44] `main()`. Der Standard definiert zwei zulässige Prototypen für diese Funktion:
`int` `main()` ohne Parameter und **`int`** `main(`**`int`** `argc, `**`char`**`* argv[])` mit zwei Parametern. Der erste Parameter `argc` ist die Anzahl der Argumente die dem Programm beim Aufruf übergeben werden, der zweite Parameter `argv` ist ein Array mit Zeigern auf die Zeichenketten, die dem Programm übergeben werden, alternativ kann er auch **`char`**`**` deklariert werden.

Die einzelnen Anweisungen in main werden nacheinander in einem eigenen Thread ausgeführt.

Listing 11: Programm Argumente

```
1 //Das Programm "'Argumentausgabe"'
2 int main(int argc, char** argv){
3    for(int i=0; i < argc; ++i)
4       cout << argv[i] << endl;
5    return 0; // wird vom Compiler ersetzt wenn nicht vorhanden
6 }
```

Programmausführung:
`Argumentausgabe Hello Argumente`

Erzeugt die Ausgabe:
`Hello`
`Argumente`

Die **`return`** Anweisung in `main` kann gemäß C++ Standard weggelassen werden, sie wird vom Compiler automatisch erzeugt und liefert 0 zurück.

Die einzelnen Bestandteile werden in den nachfolgenden Kapiteln erläutert.

6.2 Das Schlüsselwort `class`, `struct` und `template`

Das wichtigste Konzept für die objektorientierte Programmierung in C++ ist die Klasse. Eine Klasse ist ein vom Benutzer zu definierender Typ[45]. Der einzige Unterschied zwischen einer Klasse und einer Struktur sind die **`default`** `access specifier`. Bei **`class`** ist es **`private`** und bei `struct` **`public`**[46].
Das wichtigste Konzept für die generische Programmierung ist das Template. Ein Template ist eine Vorlage zur Erstellung von Funktionen oder Klassen. Die Templates stellen eine eigene funktionale Sprache in C++ dar. Die Programme die in

[44] section 6.32 auf Seite 125
[45] [Str00a] Vorwort zur ersten Auflage
[46] Siehe **??**

dieser Sprache erstellt sind, werden zur Compilezeit vom Compiler ausgeführt. Die wichtigste Anwendung dieser Sprache ist die Erzeugung von „normalem“ objektorientiertem Code.

6.3 Das Schlüsselwort `public`, `protected` und `private`

`public`, **`protected`** und **`private`** sind *access specifier*. Das Listing 12 zeigt die Syntax der access specifier. Die Auswirkung und die UML Notation sind in section **??** auf Seite ?? beschrieben.

Listing 12: Access Specifier public

```
1  class Base{
2  // default private:
3  public:
4     void op1();
5     // weitere public elemente
6  protected:
7     void op3();
8     // weitere protected elemente
9  private:
10    int attribut;
11    // weitere private elemente
12 }
```

6.4 Das Schlüsselwort `friend`

Die Member einer Klasse, die im **`private`** Bereich der Klasse deklariert werden, können nur von Methoden der Klasse selbst verwendet und verändert werden. In manchen Fällen ist es aber notwendig, anderen Klassen oder Funktionen, den Zugriff auf diese internen Member zu gewähren. Dafür kann in der Klasse die andere Klasse oder Funktion als `friend` deklariert werden.

Die `friend` Deklaration wird nicht vererbt. Wird eine Klasse `Base` in einer Klasse `A` als `friend` deklariert, ist die **`class`** `Derived :` **`public`** `Base{;}` nicht ebenfalls `friend` der Klasse `A`.

Wenn die Deklaration der `friend` Funktion die erste Deklaration der Funktion ist, wird diese als im nächsten umgebenden Scope der Klasse angenommen. Das wird als „Friend Name Injection"[47] bezeichnet. Der Name ist im umgebenden Scope aber nur im Zusammenhang der Klasse sichtbar, wenn argument dependent lookup (ADL)[48] zum tragen kommt.
Ein typisches Beispiel für `friend` Funktionen/Operatoren ist der Operator `std::ostream& operator<<(std::ostream&, Widget` **`const`**`& w)` wie er in section 6.25.10 auf Seite 94 beschrieben ist.

[47] [VJ03] S. 125
[48] section 6.18.2 auf Seite 56

Eine anderer Anwendungsfall ist die `friend` Deklaration in Templates, z.B. in einer Klasse `template<class T> class SmartPointer`, um den Konvertierungsoperationen[49], den Zugriff auf die Member zu gewährleisten: `template<class Z> friend class SmartPointer`. *Repository:Cpp-Basics/SmartPointer*

Ein typisches Beispiel für die Notwendigkeit der **gegenseitigen** `friend` Deklaration ist das StatePattern[50], bei dem der Kontext alle Nachrichten an seinen aktuellen Zustand delegiert. Der Zustand benötigt Zugriff auf den Kontext um diesen zu manipulieren.

6.5 Operationen die der Compiler zur Verfügung stellt[51]

Der Compiler stellt für Klassen die Operationen **default constructor, copy constructor, copy assignment operator** und einen **destructor** zur Verfügung.
Ab C++11 wird ausserdem noch ein **move constructor** und ein **move assignment operator** zur Verfügung gestellt. Sie werden `public` und `inline` zur Verfügung gestellt, damit einfache Klassen nicht durch Code überfrachtet werden, der immer gleich ist und deshalb synthetisiert werden kann.

Diese Operationen werden nur unter bestimmten Bedingungen zur Verfügung gestellt, wenn sie nicht in der Klasse definiert sind, aber auf Grund der Verwendung der Objekte benötigt werden (z.B: `Widget w2(w1)`).
Hier hat der Compiler die Möglichkeit, den notwendigen Code einzufügen um die Konstruktoren / Destruktoren der Basisklassen und der Datenelemente aufzurufen.

In einer Vererbungshierarchie, werden zuerst die Konstruktoren der Basisklassen, dann die Konstruktoren der eigenen Member und zum Schluss der eigene Konstruktor aufgerufen. Die Destruktoren werden in umgekehrter Reihenfolge wie die Konstruktoren aufgerufen.

Das Verhalten von Konstruktoren im Zusammenhang mit Exceptions wird in section 6.34.2 auf Seite 145 beschrieben.

6.5.1 Destruktor

Der Destruktor, den der Compiler erzeugt, ist `noexcept(true)` und nicht `virtual` außer die Klasse erbt von einer Basisklasse deren Destruktor `virtual` ist.

6.5.2 Default Konstruktor

Der default constructor wird nur zur Verfügung gestellt, wenn es gar keinen anderen Konstruktor gibt[52].

[49] section 10.7 auf Seite 201
[50] siehe section 16.3 auf Seite 288 und section 16.5.7 auf Seite 298
[51] [Mey06a]Item 5-12
[52] section 6.7 auf Seite 41

6.5.3 Kopie Operationen

Der copy constructor und der copy assignment operator, die der Compiler erzeugt, machen eine Kopie der Datenelemente der Objekte entweder mit den Kopieopertionen der Datenelemente oder bitwise[53].
Sie werden nur erzeugt, wenn es keine Move Operationen gibt.
Wenn es benutzerdefinierte copy- oder move- Operationen oder einen Destruktur gibt, ist das ein Hinweis, dass Objekte dieser Klasse in irgendeiner Weise Resourcen verwalten. Deshalb ist die Erzeugung der Kopieoperationen deprecated wenn es einen benutzerdefinierten Destruktor gibt.

6.5.4 Move Operationen

Der move constructor und der move assignment operator, die der Compiler erzeugt, verschieben die Datenelemente der Objekte entweder mit den Moveoperationen der Datenelemente oder bitwise[54].

Wenn es benutzerdefinierte copy- oder move- Operationen oder einen Destruktur gibt, ist das ein Hinweis, dass Objekte dieser Klasse in irgendeiner Weise Resourcen verwalten. Deshalb werden die move Operationen nur erzeugt, wenn es keine dieser Operationen gibt.

Wenn es keine Move Operationen gibt, werden die Kopieoperationen verwendet.

6.5.5 Template Versionen

Die templatisierten Versionen der Copy- oder Move- Operationen wie in Listing 13 werden bei den vorgestellten Regeln nicht berücksichtigt[55].

Listing 13: Die templatisierten Copy- und Move- Operationen

```
1  class Widget{
2     ...
3     // mit forwarding References T&&
4     template<class T>
5     Widget(T && rhs);
6     template<class T>
7     Widget& operator=(T && rhs);
8     ...
9     // oder T const &
10    template<class T>
11    Widget(T const& rhs);
12    template<class T>
13    Widget& operator=(T const& rhs);
14 };
```

[53] section 6.8 auf Seite 41
[54] section 6.9 auf Seite 42
[55] section 10.7 auf Seite 201

Die Versionen mit forwarding[56] References sollten vermieden werden[57].

6.6 Operationen `delete` und `default` C++11

Bestimmte Operationen, die vom Compiler bei Bedarf erzeugt werden, die aber für den Typ nicht möglich sein sollten, z.B. kopieren, konnten vor C++11 nur durch eine **`private`** Deklaration von der Verwendung durch die Benutzer ausgeschlossen werden.

Der Compiler wird diese Operationen dann nicht erzeugen, weil sie vorhanden sind und Clients können sie nicht verwenden, weil sie keinen Zugriff darauf haben.
Ab C++11 stehen dafür die Schlüsselworte **`default`** und `delete` zur Verfügung. Das Schlüsselwort `delete` hinter der Signatur einer Operation, weist den Compiler an, keine Methode für diese zu erzeugen, **`default`** weist den Compiler an, eine Default Methode zu erzeugen obwohl diese auf Grund der Regeln, nicht erzeugt würde. Siehe section 6.5 auf Seite 38.
In der Klasse `NonCopyable`[58] in Listing 14 wurde der Copy -Ctor und Assignment Operator mit `delete` gekennzeichnet, also definiert! Daher wird der Default Konstruktor nicht mehr vom Compiler zur Verfügung gestellt. Mit dem Schlüsselwort **`default`** wird er angewiesen, das aber trotzdem zu tun.

Listing 14: Die Schlüsselworte default und delete

```
1 class NonCopyable{
2 public:
3    NonCopyable() = default;
4    NonCopyable(NonCopyable const&) = delete;
5    NonCopyable& operator=()(NonCopyable const&) =delete;
6 };
```

`delete` kann auch auf globale Funktionen angewendet werden, z.B. wie in Listing 15 um mit einer Überladung, die mit `delete` gekennzeichnet ist, eine implizite Typkonvertierung zu verhindern[59].

Listing 15: keyword delete für globale Funktionen

```
1 bool isLucky(int number);
2
3 if(isLucky('a')) ... // is 'a' a lucky number?
4 if(isLucky(true)) ... // ?
5 if(isLucky(3.5)) ...  // truncate to 3 ?
6
7 bool isLucky(char) = delete; // weist chars zurück
8 bool isLucky(bool) = delete; // weist bool zurück
9 bool isLucky(double) = delete; // weist double und float zurück
```

[56] ehemals universal reference https://isocpp.org/files/papers/N4164.pdf
[57] [Mey15] Item 26 Avoid overloading on universal references
[58] [Gri12] 3.2.2 Explizite Klassendefinitionen
[59] [Mey15] Item 11

6.7 Default Konstruktoren und Default Argumente

Ein **default constructor** kann ohne Argumente aufgerufen werden, entweder weil er keine Parameter hat oder weil für jeden Parameter ein default Argument definiert ist.

Listing 16: Default Constructor

```
1 class A{
2    A(); //default
3    explicit A(int x); // kein default
4 };
5
6 class B {
7    explicit B(int x=0, bool b=true); //default
8 };
```

6.8 Copy Operationen

Nur zwei Operationen sollten Objekte kopieren. Der copy constructor und der copy assignment operator. Sie müssen gewährleisten, dass alle Attribute kopiert werden. In Vererbungshierarchien muss also die jeweilige Kopieoperation der Basisklasse aufgerufen werden.

Der Copy Konstruktor und Copy Assignment Operator wird vom Compiler synthetisiert, wenn er benötigt wird und kein Move Konstruktor und Move Assignment Operator definiert ist!

Der **copy constructor** in Listing 17 initialisiert ein Objekt mit den Werten eines anderen Objekts des gleichen Typs. Der copy constructor definiert wie ein Objekt **call by value** übergeben wird (Listing 18 auf der nächsten Seite Zeile 5).

Listing 17: Copy Konstruktor und Assignment

```
1 class Widget {
2    Widget(Widget const& rhs); // copy ctor
3    Widget& operator = (Widget rhs); // copy assignment
4 };
```

Der Parameter des copy constructors muss eine Referenz sein, da es sonst zu einer endlosen Rekursion kommt. Der **copy assignment operator**[60] (`T& operator =(T rhs)`) überschreibt ein bereits existierendes Objekt mit den Werten eines anderen Objekts desselben oder eines zuweisungskompatiblen Typs. Dabei kann es zu Type Slicing[61] kommen.

[60] section 6.25.11 auf Seite 96
[61] section 6.16.4 auf Seite 53

Listing 18: Copy Assignment Operator

```
void f(Widget w);
Widget w1; // Default Konstrutor
Widget w2(w1); // Initialisierung copy ctor explizit
Widget w3 = w1; // Initialisierung copy ctor implizit
f(w1); // Funktionsaufruf, w wird mit w1 durch den copy
construktor von Widget mit w1 initialisiert
w3 = w2; // Zuweisung
```

Für eingebaute Datentypen (Listing 19) gibt es einen default und einen copy Konstruktor und die Konstruktion mit Initialisierungsliste[62].

Listing 19: Konstruktoren für eingebaute Datentypen

```
int i = int(); // initialisierung mit 0!
int j(5); // initialisierung mit 5
int k = 5; // initialsierung mit 5
// C++11
int g{}; // initialisierung mit 0
int h{5}; // initialisierung mit 5
```

Der grundsätzliche Unterschied der Operationen copy Konstruktor und assignment Operator wird deutlich, wenn Objekte Resourcen besitzen und diese wieder freigeben müssen. Der copy Konstruktor muss lediglich eine Kopie der Resource anfordern und diese verwalten, während der copy assignment Operator die vorhandene Resource zusätzlich noch freigeben muss!

6.9 Move Operationen C++11

Kopieren ist meist eine teure Operation. Wenn einfach die Eigentümerschaft (Ownership) von Speicherbereichen und anderen Resourcen übernommen werden kann, ohne diese kopieren zu müssen, resultiert daraus ein erheblicher Performance Gewinn. Genau das ist der Sinn der Move Operationen.

Der Move Konstruktor und der Move Assignment Operator in Listing 20 müssen mit einer R-Value Referenz[63] als einzigen Parameter deklariert werden, damit der Compiler diese Operationen als Move Operationen erkennen und verwenden kann.

Listing 20: Move Konstruktor und Assignment

```
class Widget {
   Widget(Widget && rhs); // move ctor
   Widget& operator = (Widget && rhs); // move assignment
   ...
};
```

[62] siehe section 6.14 auf Seite 47

[63] siehe section 6.29 auf Seite 111 und section 6.31 auf Seite 121

Move Operationen werden vom Compiler immer ausgewählt, wenn ein Objekt kopiert werden müsste und das kopierte Objekt danach nicht mehr benötigt wird, wie `wTemp` in Listing 21. Gibt es keine passende Moveoperation, wird die korrespondierende Copyoperation verwendet.

Listing 21: Explizites und implizites Move

```
1  Widget foo(Data somedata){
2     Widget wTemp;
3     ....
4     return wTemp; // Compiler wählt implizit move ctor aus oder optimiert
5  }
6  int main(){
7     Vector<Widget> coll;
8     ...
9     Widget w = foo(data); // implizites move
10    ...
11    coll.push_back(std::move(w)); // explizit move operationen anfordern
12    // w ist hier in einem unbrauchbaren Zustand
13 }
```

Mit der Funktion `std::move(..)`, wird eine L-Value Referenz (hier:`w`) in eine R-Value Referenz gecastet `push_back(std::move(w))` und damit die Move Operation anstatt der Copy Operation vom Compiler verwendet. `std::move()` ist ungefähr wie in Listing 22 definiert[64].

Listing 22: Die Implementierung von std::move()

```
1  template<typename T>
2  typename remove_reference<T>::type&& //return Value
3  move(T&& obj){
4     return static_cast<typename remove_reference<T>::type&&>(obj);
5  }
```

Der Standard garantiert für die Funktion[65] `foo(...)` aus Listing 21:

- Wenn das lokale Widget einen copy oder move Konstruktor hat, kann der Compiler das Kopieren wegoptimieren, das ist die sogenannte *(named) return value optimization ((N)RVO)*, die es schon vor C++11 gab und von den meisten Compilern angewendet wird.
  Die Optimierung erfolgt nur, wenn der return Typ und der Typ des lokalen Ausdrucks übereinstimmen. `std::move(local)` sollte nicht auf lokale Objekte angewendet werden, weil der Rückgabetyp von `std::move(widget)` eine R-Value Reference auf das `widget` ist und damit die RVO nicht zum Tragen kommt. Der Compiler muss ein lokales Objekt, das zurückgegeben wird, als R-Value behandeln wenn er die (N)RVO nicht anwendet[66].
- Ansonsten, wenn `Widget` einen move Konstruktor hat, wird `wTemp` moved.
- Ansonsten, wenn `Widget` einen copy Konstruktor hat, wird `wTemp` kopiert.

[64] [Gri12]
[65] [Jos12]
[66] [Mey15] Item 25, S. 176

- Ansonsten wird der Versuch, `wTemp` zu kopieren, mit einem Compiler Error quitiert.

Mit C++17 wird die sogenannte „copy elision“ zwingend vorgeschrieben[67].

Die Move Operationen müssen das `rhs` Objekt in einem Zustand hinterlassen, in dem es ohne weiteres zerstört werden kann. Ansonsten kann keine Aussage über den Zustand eines solchen Objekts getroffen werden, das heist, es darf nicht mehr verwendet werden.

Die folgenden drei Schritte[68] müssen bei der Implementierung für die Move Operationen durchgeführt werden:

- Zielwert löschen oder swap (bei Move Assignment Operator)
- Quelle nach Ziel transferieren (z.B. Kopie eines Zeigers auf den Adressbereich der Daten)
- Quelle in einen definierten Zustand versetzen (z.B. Zeiger `nullptr`) zuweisen

Die Implementierung der Move Operationen ist in Listing 23 skizziert. Im Move Konstruktor werden die eigenen Member mit den Membern von `rhs` initialisiert und `rhs` die Ownership genommen. Im Move Assignment Operator kann ein Teil der Member (`data`) getauscht werden, der Destruktor von `rhs` wird zu gegebener Zeit den Speicher aufräumen. Wenn die eigene Resource möglichst früh freigegeben werden sollte, kann das hier auch direkt erfolgen (`delete data; data = rhs .data;`) und `rhs.data = nullptr` zugewiesen werden. Mit `std::move(..)` wird der Move Assignment Operator von `Vector` verwendet, der sicher das Richtige tut.

Listing 23: Die Implementierung der Move Operationen

```
1  class DemoMoveOperation{
2     Widget* data = nullptr;
3     size_t size = 0;
4     Vector<Widget> vec;
5  public:
6     ...
7     DemoMoveOperation(DemoMoveOperation && rhs)
8     : data(rhs.data), size(rhs.size), vec(std::move(rhs.vec))
9     {
10       rhs.data = nullptr; rhs.size = 0;
11    }
12    DemoMoveOperation& operator=(DemoMoveOperation && rhs)
13    {
14       swap(data, rhs.data); // oder gleich aufräumen
15       swap(size, rhs.size);
16       vec = std::move(rhs.vec);
17    }
18    ~DemoMoveOperation(){
19       delete data; // delete nullptr has no effect
20    }
```

[67] C++17: http://en.cppreference.com/w/cpp/language/copy_elision
[68] [Gri12]

```
21 };
```

6.10 Vererbung von Konstruktoren C++11[69]

Eine Klasse `class Widget : public BaseWidget` kann die Konstruktoren von `BaseWidget` durch eine `using` Klausel zugänglich machen.

Listing 24: Vererbung von Konstruktoren

```
1 class BaseWidget { ... };
2 class Widget : public BaseWidget {
3 public:
4    using BaseWidget::BaseWidget;
5 };
```

Bis auf den copy- und den default constructor werden in Listing 24 alle Konstruktoren mit ihren Eigenschaften zugänglich gemacht. Eigenschaften sind die access-specifier (`public, ...`), `inline, explicit, default, delete`. Durch die `using` Deklaration in `Widget` wird kein default construktor zur Verfügung gestellt, weil es andere Konstruktoren gibt. Siehe section 6.5 auf Seite 38.
Konstruktoren von BaseWidget werden von Konstruktoren in Widget, die die gleiche Signatur haben, überdeckt.
Zweideutigkeit durch Mehrfach Vererbung und gleicher Signatur der Konstruktoren, muss durch einen Konstruktor mit dieser Signatur aufgelöst werden. Dieser überdeckt die geerbten Konstruktoren.

Besitzt `Widget` eigene Attribute, besteht die Gefahr, dass diese nicht initialisiert werden, wenn die Konstruktoren der Basisklasse verwendet werden.

6.11 Delegation von Konstruktoren C++11

Konstruktoren können andere Konstruktoren, entweder derselben Klasse oder der Basisklasse aufrufen. In Listing 25 ruft der Default Konstruktor den Konstruktor `A(int)` auf. Das erspart die Wiederholung der Member in der Initializerlist[70] in allen Konstruktoren.

Listing 25: Delegation von Konstruktoren

```
1 class A{
2    A():A(42){}
3    A(int i):i(i){}
4
5    int i;
6 };
```

[69] [Gri12] 7.1.3 Vererbung von Konstruktoren
[70] siehe section 6.16.3 auf Seite 52

Es darf dabei nicht zu einem rekursiven Aufruf der Konstruktoren kommen. Ein Objekt gilt nach dem ersten Konstruktor als vollständig konstruiert. Das ist im Zusammenhang mit Exceptions von Bedeutung[71], da der Destruktor nach dem ersten Konstruktor gerufen wird, wenn in einem weiteren eine Exception ausgelöst wird.

6.12 constexpr Konstruktoren C++11[72]

Eine Klasse mit einem constexpr Konstruktor wird als *literal type* bezeichnet. Der Funktionskörper des Konstruktors muss leer sein und alle Member müssen mit potenziell constexpr Ausdrücken initialisiert werden[73]. Objekte solcher Klassen können constexpr deklariert werden und damit in anderen constexpr Ausdrücken verwendet werden. Operationen die constexpr deklariert sind, sind in C++11 implizit **const** das Schlüsselwort **const** kann daher weggelassen werden, in C++14 ist das nicht mehr so[74]. Das Listing 26 zeigt ein Beispiel für eine Klasse und Operationen.

Listing 26: Ein Literal Type mit constexpr Konstruktor

```
1 struct Point{
2    int x, y;
3    constexpr Point(int x, int y):x(x), y(y){/*empty body*/}
4    constexpr Point move(int dx, int dy){ return Point(x+dx, y+dy);}
5 };
6
7 constexpr Point origin(0,0); // Konstantes Objekt
8 constexpr int x = origion.x; // verwendung zur Initialisierung einer Konstanten
```

6.13 Einschränkungen von Konstruktoren

1. Die Verwendung von **this** ist nur eingeschränkt erlaubt (z.B. Weitergabe an einen anderen Thread), da das Objekt eventuell noch nicht vollständig konstruiert ist.

2. Aus demselben Grund dürfen in Konstruktoren keine polymorphen Operationen verwendet werden!

3. Konstruktoren können nicht polymorph sein.

[71] siehe section 6.34.2 auf Seite 145
[72] [Str13] S.265 10.4.3 Literal Types
[73] siehe section 6.33 auf Seite 140
[74] http://www.informatik-aktuell.de/entwicklung/programmiersprachen/neuerungen-in-cpp-14.html

6.14 Einheitliche Initialisierung und Initialisierungslisten C++11

Mit dem Standard C++11 sollte die Syntax der Initialisierung von Objekten vereinheitlicht werden, so dass überall, insbesondere in Templates, die gleiche Syntax für dieselbe Semantik bei der Initialisierung verwendet werden kann[75]. Leider wurde dieses Ziel nicht wirklich erreicht[76]

Listing 27: Einheitliche Initialisierungssyntax

```
1 int values[] {1, 2, 3, 4};
2 vector<int> v {1, 2, 3, 4}; // nur copyable elements möglich!
3 vector<string> cities {"Berlin", "New York"};
4 int i; // uninitialisiert, undefinierter Wert!
5 int j{}; // initialisiert mit 0
6 int* p{}; // initialisiert mit nullptr
```

Eine „initializer list“ erzwingt eine „so called“ *value initialization*, das bedeutet, dass auch lokale Variablen von fundamentalen Datentypen[77] wie in Listing 27, die normalerweise einen undefinierten Wert haben, wenn sie nicht explizit initialisiert werden, mit null bzw. mit nullptr initialisiert werden.

Bei der Initialisierung mit einer Initialsierungsliste werden keine implizite Typkonvertierungen[78] vorgenommen, wenn dabei Informationen verloren gehen.

Listing 28: Initialisierungslisten ohne Typkonvertierungen

```
1 int x1(5.3);   // Ok, aber x1 hat den Wert 5!
2 int x2 = 5.3;  // Ok, aber ebenfalls 5
3 int x3{5.3};   // Error: narrowing
4 char c1{7};    // Ok, obwohl 7 ein int ist, der wert passt in einen char
5 char c2{99999};   // Error: 99999 passt nicht in einen char
6 std::vector<int> v1{1, 2, 3}; // Ok
7 std::vector<int> v2{1, 2, 3.4}; // Error, narrowing doubles to ints
```

Das Listing 28 zeigt, innerhalb der Initialisierungsliste werden nicht nur die Typen, sondern auch die Werte bei der Entscheidung berücksichtigt, ob es sich bei der Konvertierung um „narrowing“, und daraus resultierende Informationsverluste handelt.

6.14.1 Konstuktoren mit Initialisierungsliste

Um das Konzept von Initialisierungslisten für Benutzer definierte Typen zu unterstützen, gibt es das Klassentemplate `std::initializer_list<>`. Die Klasse muss dafür einen entsprechenden Konstruktor zur Verfügung stellen.

[75] http://www.stroustrup.com/C++11FAQ.html#uniform-init
[76] [Mey15] Item 7 Distinguish between () and {} ...
[77] siehe section 6.22 auf Seite 69
[78] siehe section 6.25.5 auf Seite 90

Listing 29: Initialisierungslisten in Konstruktoren

```
struct Widget{
   Widget(int, int);
   Widget(std::initializer_list<int>);
};
Widget w(77, 5); // Widget::Widget(int, int) wird benutzt
Widget q{77, 5}; // Widget::Widget(initializer_list) wird benutzt
Widget r{77, 5, 42}; // Widget::Widget(initializer_list) wird benutzt
Widget  s = {77, 5}; // Widget::Widget(initializer_list) wird benutzt
```

Ohne den Konstruktor mit Initialisierungsliste würde in Listing 29 der Konstruktor `Widget(int, int)` zur Initialisierung von p, q und s verwendet, die Initialisierung von r wäre illegal.

Eine `std::initializer_list<type>` ist const und kann daher nur für Typen verwendet werden, die Kopieoperationen zur Verfügung stellen. Das heist, wenn `class Widget` nur Move Operationen bereitstellt, ist `vector v{A(1), A(2)};` nicht möglich, `A aA[]{A(1), A(2)};` aber schon;

Das Schlüsselwort `explicit` wird durch die Initialisierungsliste auch für Konstruktoren mit mehreren Parametern relevant.

Listing 30: Initialisierungslisten und explizite Konstruktoren

```
struct P{
   Widget(int, int);
   explicit Widget(int, int, int);
};
Widget p(77, 5); // Ok
Widget q{77, 5}; // Ok
Widget r{77, 5, 42}; // Ok
Widget  s = {77, 5}; // Ok
Widget  t = {77, 5, 42}; // Error, keine implizite Typkonvertierung erlaubt

void fp(const Widget&);

fp({47, 11}); // Ok, temporäres Widget wird mit Widget::Widget(int, int) erzeugt
fp({47, 11, 42}); // Error, keine implizite Typkonvertierung erlaubt
fp(Widget{47, 11}); // Ok, explizite Erzeugung eines temporären Objekts
fp(Widget{47, 11, 42}); // Ok, explizite Erzeugung eines temporären Objekts
```

Auf die selbe Art wie in Listing 30 wird die implizite Konvertierung durch einen Konstruktor mit Initialisierungsliste durch das Schlüsselwort `explicit` verhindert.

Wie in Listing 31 gezeigt, kann das Template auch verwendet werden, wenn eine Liste von Werten verarbeitet werden soll.

Listing 31: Anwendung `std::initializer_list`

```
template<class Iterable>
void print(Iterable& it){
   for(auto element : it)
      cout << element << " ";
```

```
5 }
6
7 print({1, 3, 5}); // Eine Liste von Werten ausgeben
```

6.15 Destruktoren

Destruktoren werden am Lebensende eines Objekts automatisch aufgerufen[79]. Sie werden vom Compiler erzeugt, wenn die Klasse keine Deklaration dafür hat[80].

Die Deklaration eines Destruktors der Klasse `Widget` hat die Form `~Widget()noexcept` .

Wird während der Konstruktion eines Objekts eine Exception geworfen, werden die Destruktoren für alle vollständig konstruierten Member gerufen[81].

6.15.1 Einschränkungen von Destruktoren

1. In polymorphen Klassen (von denen geerbt werden soll) müssen Destruktoren `virtual` deklariert werden.
2. Destruktoren dürfen nicht **`private`** und sollten nicht **`protected`** sein! Ausnahmen zu dieser Regeln ergeben sich z.B: aus dem DesignPattern Singleton, wenn die Klasse selbst für die Erzeugung und die Zerstörung der Objekte verantwortlich ist.
3. Destruktoren dürfen keine Exceptions werfen! (`~Widget()noexcept`[82]) Weil z.B: bei der Verarbeitung einer Exception alle Objekte auf dem Stack zerstört werden (stack unwinding) also die Destruktoren der Objekte gerufen werden. Wird während der Verarbeitung einer Exception, eine Exception geworfen, wird `std::terminate()` aufgerufen und das Programm terminiert[83].
4. in Destruktoren dürfen keine polymorphen Operationen verwendet werden, weil das Objekt eventuell nicht mehr vollständig ist!

Der Destruktor der polymorphen Klasse `Base` muss also wie folgt deklariert werden:

```
virtual ~Base()noexcept
```

6.15.2 Der virtual Destruktor

Wenn eine Klasse als Basisklasse verwendet werden können soll, muss im Zusammenhang mit dynamischer Polymorphie, der Destruktor virtual deklariert werden! Beispiel:

[79] siehe section 6.21 auf Seite 62, section 6.34 auf Seite 143
[80] siehe section 6.5 auf Seite 38
[81] siehe section 6.11 auf Seite 45
[82] siehe section 6.34.5 auf Seite 151, section 6.34.4 auf Seite 149
[83] [Str00a] 14.4.7 Ausnahmen in Destruktoren

Listing 32: Virtual Destructor

```
1  class Member{
2  public:
3     ~Member() noexcept { std::cout << "Member::~Member()" << std::endl; }
4  };
5
6  class A { // Baseclass
7  public:
8     virtual ~A() noexcept {...} // destruktor
9  };
10
11 class B : public A {
12 public:
13    ...
14    B():A() { /*hier resourcen beschaffen*/ }
15    virtual ~B() noexcept { /*hier resourcen freigeben*/ }
16 private:
17    Member member;
18 };
19
20 void f(){
21    A pA = new B; // die ctors werden gerufen
22    // etwas mit pA machen
23    delete pA; // Objekt zerstören, Speicher freigeben
24    // sizeof(A) == sizeof(void*);
25 }
```

Die Variable pA ist ein Zeiger auf ein A. Da B ein A gemäß der Vererbung ist, kann der Zeiger auf die Basisklasse mit der Adresse eines B Objekts durch einen impliziten upcast initialisiert werden[84]. Das wird auch als Zuweisungskompatibel bezeichnet. Wenn der Destruktor von A nicht virtual deklariert ist, wird bei der delete Anweisung nur der Destruktor von A, nicht aber der von B und auch nicht der Destruktor des Member aufgerufen! Das Objekt hat keine Chance, seine im Konstruktor beschafften Resourcen frei zu geben. Resourceleaks sind die Folge. Hier nützt auch die Befolgung des Ratschlags: „Resource acquisition is initialization (RAII)“ nichts.

Die Größe der Objekte ist, sobald eine virtual Operation in der Klasse vorhanden ist, mindestens die Größe einer Adresse! Der Compiler fügt den Objekten am Speicheranfang die Adresse der sogenannten virtual function table (vtbl) hinzu, über die die polymorphen Operationsaufrufe verwaltet werden.

6.16 Lebenszyklus und Speicheraufbau von Objekten

Dieses Kapitel zeigt den Lebenszyklus von Objekten und die in diesem Zusammenhang automatisch oder explizit aufgerufenen Methoden. Diese werden unter

[84] siehe section 6.26 auf Seite 103

bestimmten Umständen durch den Compiler synthetisiert, wenn Sie nicht vorhanden sind. Die Methoden und die dazu gehörenden Regeln sind in section 6.5 auf Seite 38 beschrieben.

Repository:Cpp-Basics/LifeCycle

Grundsätzlich gilt[85]: Objekte, die vom Compiler verwaltet werden, werden in umgekehrter Reihenfolge zerstört in der sie erzeugt wurden. Das gilt sowohl für lokale Objekte auf dem Stack innerhalb einer Funktion als auch für statische oder globale Objekte.
Das gilt nicht für Objekte, die durch den Benutzer verwaltet werden, die z.B. mit **new** erzeugt werden. Deren Lebensdauer wird durch den Benutzer beendet, z.B. durch den Aufruf von `delete` oder durch den expliziten Aufruf des Destruktors `~Widget()`[86].

Im Folgenden werden einige Beispiele beschrieben, ausgehend von einer Funktion `f()` die z.B. in `main()` aufgerufen wird. Die Funktion sowie die Klassen werden dabei in jedem Beispiel immer wieder neu definiert.

Listing 33: main()

```
1 void f();
2 int main() { f(); }
```

Die einzige Anweisung in main ist der Funktionsaufruf von f.

6.16.1 Leere Klasse

Listing 34: Leere Klasse

```
1 class A{}; // eine leere Klasse
2 void f() {
3    A a; // der default ctor wird gerufen
4    // sizeof(a) == sizeof(A) == 1 min 1 Byte groß
5 } // der dtor von A wird gerufen
```

Der this Pointer muss mit einer gültigen Adresse initialisiert werden können, daher haben auch leere Klassen mindestens eine Größe von einem Byte. Der Destruktor wird automatisch gerufen, wenn das Objekt aus dem Scope kommt (Ende des Blocks{...}).

6.16.2 Klasse mit einem Datenmember / Attribut

[85] see: https://akrzemi1.wordpress.com/2013/07/18/cs-best-feature/
[86] Bsp. im Destruktor in Listing 96 auf Seite 106

Listing 35: Klasse mit einem Datenmember / Attribute

```
class A {
public:
   explicit A(int i) : myInt(i) {} //ctor mit initialisierungsliste
   A():myInt(0) {}
   ~A() {...} //destruktor
   void setI(int i);
private:
   int myInt; //Attribut/Datenmember
};

void f() {
   A object; // der default ctor wird gerufen
   // sizeof(object) == sizeof(A) == sizeof(int) == true;
}
```

Der Speicheraufbau des Objekts wird durch die Attribute und deren Reihenfolge, wie sie in der Klasse definiert sind, bestimmt.

6.16.3 Konstruktoren und Member Initialisierungsliste

Listing 36: Konstruktoren und Member Initialisierungsliste

```
class B {
public:
   B(){}
   B(int i1, int i2) {
      a1.setI(i1);
      a2.setI(i2);
   }
   ~B(){...} // destruktor
private:
   A a1;
   A a2;
};

void f() {
   B object(1,2); // B(int, int) und A() wird verwendet
   // sizeof(object) == sizeof(B) == 2 * sizeof(A)
} // dtor B, dann dtor für a2, dann dtor für a1
```

Die Klasse `B` in Listing 36 hat zwei Member `a1`, `a2` der Klasse `A`. Ihre Member werden vor dem Objekt der Klasse erzeugt und nach dem Objekt zerstört.

In Listing 36 werden die Member zuerst mit dem default Konstruktor in der Reihenfolte a1, a2 erzeugt und anschließend der Konstruktor von B aufgerufen, der die Defaultwerte gleich wieder durch die Übergabewerte überschreibt. Besser ist es, die Member gleich in der Initialisierungsliste zu initialisieren. Der Konstruktor von B sieht dann wie folgt aus:

```
B(int i1, int i2): a1(i1), a2(i2){ }
```

In der Initialisierungsliste werden die passenden Konstruktoren der Member ausgewählt. Der überflüssige Aufruf der default Konstruktoren und die anschließenden Zuweisungen (`setI(..)`) entfallen.

Die Reihenfolge der Deklaration der Attribute in der Klassendefinition bestimmt die Reihenfolge im Hauptspeicher und die Reihenfolge ihrer Initialisierung. An der kleineren Adresse befindet sich a1, danach a2.
Die Reihenfolge in der Initialisierungsliste hat keinen Einfluss auf die Reihenfolge der Initialisierung. Deshalb sollten die Member in der Initialisierungsliste in der Reihenfolge, wie sie in der Klasse deklariert sind, gelistet werden. Dasselbe gilt bei Vererbung für die Basisklassen. Ansonsten quitiert der Compiler es mit einer Meldung z.B.: `warning: 'A::a2'will be initialized after A::a1 [-Wreorder]`

Listing 37: Vererbung

```
1  class A2{...}; // ähnlich wie class A
2  class B : public A, public A2 {
3  public:
4     B(){}
5     B(int i1, int i2): A(i1), A2(i2){}
6  };
7
8  void f(){
9     B object(1,2); // B(int, int), A(int) und A2(int) werden verwendet
10    // sizeof(object) == sizeof(B) == sizeof(A) + sizeof(A2)
11 }
```

Die Klasse B erbt sowohl von A als auch von A2. Das wird Mehrfachvererbung genannt. Werden keine Konstruktoren in der Initialisierungsliste ausgewählt, werden die Defaultkonstruktoren verwendet.

6.16.4 Type Slicing

Bei der Initialisierung eines Objekts einer Basisklasse mit einem Objekt einer abgeleiteten Klasse werden nur die Anteile der Basisklasse übernommen, die Anteile der abgeleiteten Klasse gehen verloren. Entsprechendes gilt bei der Zuweisung.

Listing 38: type slicing

```
1  class A {
2  public:
3     A();
4     ...
5  };
6  class B : public A {
7     ...
8     // Datenmember
9  };
10
```

```
11 void f1(A a); // call by value
12 void f2(A* a); // Parameter ist ein Zeiger
13 void f3(A& a); // call by reference
14 void f(){
15    B b;
16    f1(b); // nur der A Anteil wird kopiert
17    A a;
18    a = b; // der B Anteil wird abgeschnitten
19    // Zeiger auf ein A wird mit der Adresse eines B initialisiert
20    f2(&b); // der Adressoperator & ermittelt die Speicheradresse von b
21    f3(b);
22 }
```

Bei den Aufrufen von f2 und f3 gibt es kein type slicing, da f2 eine Adresse erwartet und f3 als Parameter eine Referenz deklariert. Es wird kein Objekt kopiert, nur der Zeiger, bzw. die Referenz wird beim Funktionsaufruf initialisiert.

6.17 Resource Acquisition is Initialization RAII

In Listing 39 wird ein Objekt auf dem Heap mit **new** erzeugt, also die Resource Memory angefordert. Fügt bei Wartungsarbeiten jemand ein **return** Statement ein oder wirft die Funktion `bar(p)` eine Exception, wird das `delete` Statement nicht erreicht und die Resource bis zum Programmende nicht mehr freigegeben. Mit einem **catch** Block[87] wird das Problem nur verlagert.

Listing 39: Resource Leak

```
1 void foo(){
2    Widget *p = new Widget;
3    ...
4    bar(p);
5    ...
6    delete p;
7 }
```

Mit Hilfe eines lokalen Objekts, wie in Listing 40 auf der nächsten Seite, das für die Freigabe der Resource verantwortlich ist und diese im Destruktor wieder freigibt, kann in diesem Fall keine Resource verloren gehen, da für lokale Objekte in jedem Fall der Destruktor gerufen wird. Das Objekt wird bei der Anforderung / Acquisition der Resource initialisert. Die STL stellt dafür verschiedene Smart-Pointer[88] wie z.B. `std::unique_ptr` zur Verfügung. Diese Technik sollte für alle Resourcen, die wieder freigegeben werden müssen, wie z.B. File- und Datenbankhandles, oder Netzwerk Connections und Mutexes, angewendet werden.
Mit dieser Technik kann auf die Definition eines Destruktors verzichtet werden, bzw. die Freigabe der Member muss nicht explizit mit `delete` erfolgen, wenn dynamisch erzeugte Member mit einem entsprechenden Smartpointer verwaltet werden.

[87] siehe section 6.34 auf Seite 143
[88] section 17.7 auf Seite 320

Listing 40: Resource Acquisition is Initialization

```
1 void foo(){
2    std::unique_ptr<Widget> p(new Widget);
3    ...
4    //keine delete notwendig, dtor von p räumt auf
5 }
```

Repository:Cpp-Basics/SmartPointer

Achtung: Werden `std::unique_ptr` für dynamisch erzeugte Member in einer Klasse verwendet, müssen die Typdefinitionen der Member im Header der Klasse includiert werden, wenn der Destruktor der Klasse im Header `inline` definiert oder vom Compiler generiert wird[89]. Sollen die Typen der Member nur als forward Deklarationen propagiert werden, muss der Destruktor der Klasse in einer eigenen Übersetzungseinheit definiert werden und dort die Header der Typen der Member includiert werden.

6.18 Namespaces[90]

Große Projekte nutzen häufig viele Libraries von verschiedenen Herstellern. Klassen reichen hier zur Gliederung des Projekts nicht aus. Die Wahrscheinlichkeit, dass einige Klassen denselben Namen tragen, nimmt mit der Anzahl der verwendeten Libraries zu. Namensräume / Namespaces sind hier das Mittel der Wahl zur Strukturierung, sie entsprechen den **package**s der UML bezüglich der Qualifizierung eines Namens aber leider gibt es in C++ keine `access-spezifier` für Namespaces. Siehe section ?? auf Seite ??.

6.18.1 Namespace Definition und Verwendung

Werden Namen **innerhalb eines Namespace** deklariert wie **`class`** `File` in Listing 41, sind sie innerhalb des Namespace sichtbar (namespace scope).
Namespaces können in verschiedenen Modulen erweitert werden. Sie sind nicht wie Klassendefinitionen an eine Datei gebunden.
Namespaces können geschachtelt werden, d.h. es können Namespaces in Namespaces definiert werden.

Listing 41: Namespace Definition

```
1 namespace hirsch{
2    namespace utilities{
3       class File;
4       void globalFunction(const File&);
5    }
6 }
7 // qualifizieren
```

[89] siehe http://en.cppreference.com/w/cpp/memory/unique_ptr Notes: „T must be complete at the point in code where the deleter is invoked“

[90] [Str00a]

```
8  hirsch::utilities::File myFile; // voll qualifizierter Name
9
10 globalFunction(myfile);
11
12 //C++17
13 namespace hirsch::utilities {
14  ... // verkürzte Schreibweise verschachtelter Namespaces ohne mehrfach Klammern
15 }
```

Sollen die Namen außerhalb des Namespace verwendet werden, müssen sie entweder mit dem Namen des Namespace voll qualifiziert werden (z.B. `std::cout`) oder sie müssen über die using Klausel zugänglich gemacht werden.

6.18.2 Using Directive und Deklaration

Um nicht immer den voll qualifizierten Namen schreiben zu müssen, kann die `using` Directive verwendet werden.
`using namespace std;` macht alle Namen des namespace std im aktuellen scope sichtbar.

Listing 42: using directive

```
1 using hirsch::utilities;
2 File myfile;
3 ...
```

Mit der **using directive** werden in Listing 42 alle Identifier aus dem `namespace hirsch::utilities` im aktuellen Scope erreichbar. Dabei kann es zu Namenskonflikten kommen. Diese können nur aufgelöst werden, in dem der voll qualifizierte Name für die Namen, die kollidieren[91], verwendet wird.
Die **using declaration** `using std::cout;` macht `cout` aus dem namespace std im aktuellen scope sichtbar.

Listing 43: using declaration

```
1 using hirsch::utilities::File;
2 File myfile;
3 ...
```

Mit der using declaration wird `File` in Listing 43 ein lokales Synonym im aktuellen Scope für `hirsch::utilities::File`.

Funktionen, die außerhalb des Namespace verwendet werden, müssen nicht voll qualifiziert werden, wenn ein oder mehrere Argumente aus demselben Namespace kommen. Diese Regel wird **argument dependent lookup (ADL)** oder **Koenig lookup**[92] genannt.

In Headern sollte niemals eine globale `using` Directive oder Deklaration verwendet werden, weil dadurch dem Nutzer des Headers ohne seinen Willen ein Name

[91] en: name clash
[92] Andrei Koenig

oder Namensraum eröffnet wird, was zu drastischen Änderungen der Bedeutung seiner verwendeten Namen führen kann. Hier bleibt nur der voll qualifizierte Namen zur Nutzung.

Die using directive oder using declaration kann nicht im class scope verwendet werden, der Compiler quitiert das mit den Fehlermeldungen wie z.B. in Listing 44. Mit einer using directive oder using declaration im Scope einer Funktion werden die Namen nur lokal in den Scope der Funktion eingeführt.

Listing 44: Using declaration in class scope

```
// Widget.h
#include <iostream>
class Widget{
   //error: expected nested-name-specifier before 'namespace'
   using namespace std;
   //error: using-declaration for non-member at class scope
   using std::cout;
   using std::endl;
public:
   void print(){
      using namespace std; // ok
      cout << "Widget::print() cout << ..." << endl;
   }
};
// main.cpp
#include "Widget.h"

#include<iostream>
//using namespace std;

int main(){
   std::cout << "NamespaceUsingScope" << std::endl;

   Widget w;
   w.print();
}
```

6.18.3 Der Namespace std des Standards

Die C++ Standard Library definiert alle Namen im `namespace std`. Der `namespace std` enthält seit C++11 die folgenden namespaces:

- `namespace rel_ops`
- `namespace chrono`
- `namespace placeholders`
- `namespace regex_constants`
- `namespace this_thread`

Innerhalb des `namespace std` waren vor C++11 die zukünftigen Erweiterungen im `namespace tr1` definiert und ergänzende relationale Operatoren im `namespace rel_ops`.

6.18.4 Namespace Alias

Für die Namen von `namespace`s können kurze Aliasnamen definiert werden. Das Listing 45 zeigt die Syntax.

Listing 45: Namespace Alias

```
1  namespace ALongNamespaceName {
2     void f(){...}
3  }
4  namespace AnotherLongNamespaceName {
5     void f(){...}
6  }
7
8  // Verwendung der Symbole in Übersetzungseinheit
9  namespace n1 = ALongNamespaceName;
10 namespace n2 = AnotherLongNamespaceName;
11 n1::f(); // calls ALongNamespaceName::f
12 n2::f(); // calls AnotherLongNamespaceName::f
```

6.18.5 Anonyme Namespaces

Um Deklarationen die nur lokal in einer Übersetzungseinheit genutzt werden sollen, vor Kollisionen zu schützen, ohne einen weiteren Namen für den Namespace einführen zu müssen, können Namespaces ohne Namen definiert werden. Die Verwendung des Namespace in Listing 46 entspricht einer impliziten using Deklaration wie in Listing 47.

Listing 46: Anonymer Namespace

```
1  namespace {
2     int a;
3     void f(){...}
4  }
5  // Verwendung der Symbole nur in dieser Übersetzungseinheit
6  a = 42;
7  f();
```

Listing 47: Anonymer Namespace explizit

```
1  namespace XXX {
2     int a;
3     void f(){...}
4  }
5  using namespace XXX;
6  a = 42;
```

```
7 f();
```

Der Name XXX ist in jeder Übersetzungseinheit verschieden.

6.19 Das Schlüsselwort enum

6.19.1 Non Scoped Enumerations

In C++ ist jede non scoped Enumeration ein eigenständiger Typ. Es gibt implizite Konvertierungen von einem enum zu einem Integertyp, aber nicht umgekehrt. Der Typ der enum Konstanten ist bedarfsabhängig, kann also auch long sein.

Listing 48: Enumeration type

```
1 enum RGB { red, green, blue } rgb = red;
2 ++rgb; // ERROR in C++ wenn kein Operator ++ für den enum wie folgt definiert ist:
3 RGB& operator++(RGB &rgb) {
4    return rgb = RGB(rgb + 1);
5 }
```

Die Namen der Konstanten sind im umgebenden Scope sichtbar und können einen Namenskonflikt / name clash verursachen.

6.19.2 Scoped Enumerations C++11[93]

Mit C++11 sind sogenannte **scoped enumeration**s hinzugekommen, bei denen der zugrundeliegende Typ explizit bestimmt werden kann. Scoped `enum`s werden mit dem Schlüsselwort **`class`** oder `struct` hinter dem Schlüsselwort `enum` definiert.
`enum` **`class`** `AmpelState :` **`char`** `{ AUS, BLINKEND, ...};`
Der Typ der Konstanten kann weggelassen werden.

Scoped `enum`s haben folgende Vorteile gegenüber non scoped enumerations:

- Die Namen der Konstanten sind nicht im Scope, in dem der `enum` definiert ist. Daher können sie keinen Namenskonflikt verursachen. Der Zugriff erfolgt über den Namen: `AmpelState::AUS`
- Der Typ (hier **`char`**) der Konstanten der Enumeration kann explizit angegeben werden. Der default Typ ist **`int`**.
- Es gibt keine impliziten Konvertierungen der Konstanten in den Typ der Werte
- Der Typ der Enumeration (`AmpelState`) kann durch eine forward declaration bekannt gemacht werden. Wenn nur der Typ der Enumeration benutzt wird, muss ein Client nicht mehr neu compiliert werden, wenn die Enumeration durch neue Werte erweitert wird: (`State* getState(AmpelState s);`)

[93] [Jos12] Scoped Enumerations

Der Typ der Konstanten kann mit dem type trait `std::underlying_type<AmpelState >::type` bestimmt werden.

6.20 Namensauflösung / Scope Resolution

Der Bereich / scope, in dem ein Name/Identifier deklariert wird, bestimmt seine **Sichtbarkeit**. Die Bereiche sind ineinander eingebettet, der äußerste Bereich ist der global scope. Namen in äußeren Bereichen sind in den inneren Bereichen sichtbar.

Die Auflösung von Namen in Funktionen, Klassen und abgeleiteten Klassen führt der Compiler immer durch, wenn er auf einen Namen stößt. Die Suche nach einer Deklaration des Namens erfolgt vom innersten Scope nach außen bis zum globalen Scope (`::`). Sobald eine Deklaration des Namens gefunden wird, bricht die Suche ab. Daher werden **alle Namen**, die in einem weiter außen liegenden Scope deklariert sind von **gleichen Namen**, die weiter innen deklariert sind versteckt (`shadow`). In Klassenhierarchien mit überladenen Operationen führt das zu einem völlig unerwarteten Verhalten!

Soll auf Namen aus dem global scope, die überdeckt sind, zugegriffen werden, muss der Scoperesolution Operator angewendet werden: `::name`[94].

6.20.1 Sichtbarkeit von Namen in Vererbungshierarchien[95]

Werden Namen **innerhalb einer Klasse** deklariert, sind sie in der Klasse sichtbar (class scope).

Der Name der Klasse selbst kann innerhalb der Klasse unqualifiziert (ohne namespace) verwendet werden[96], z.B. um einen Zeiger auf ein Objekt der Klasse zu definieren: `class C{ C* pC; ...}`.

Der Zugriff von außen hängt von den access specifiern (`public`, `protected`, `private`) ab. In Vererbungshierarchien ist der scope der abgeleiteten Klasse in den scope der Basisklasse eingebettet. Werden Operationen in der abgeleiteten Klasse mit einem Namen definiert, der in der Basisklasse bereits verwendet wird, sind diese nicht mehr in der abgeleiteten Klasse sichtbar.

In Listing 49 auf der nächsten Seite werden die Namen der Operationen `mf1` und `mf3` in der abgeleiteten Klasse überdeckt, sie werden daher nicht gefunden. Leider sollen die geerbten Operationen aber in der abgeleiteten Klasse zur Verfügung stehen.

[94] siehe section 3.7 auf Seite 23 Listing 5 auf Seite 24
[95] [Mey06a] Item 33 Avoid hiding inherited names
[96] [VJ03] 9.2.3 Injected Class Names

Listing 49: Sichtbarkeit von Namen in Vererbungshierarchien

```
class Base{
private:
   int x;
public:
   virtual void mf1()=0;
   virtual void mf1(int){};

   virtual void mf2(){};
   void mf3(){};
   void mf3(double){};
};
class Derived : public Base{
public:
   virtual void mf1(){};
   void mf3(){}
   void mf4(){};
};
void demoVisibility(){
   Derived d;
   int x;
   d.mf1();    // calls Derived::mf1
   d.mf1(x);   // error: no matching function for call to Derived::mf1(int&)
   d.mf2(); // calls Base::mf2
   d.mf3(); // calls Derived::mf3
   d.mf3(x);   // error: no matching function for call to Derived::mf3(int&)
}
```

Durch eine Deklaration `using Base::mf1` und `using Base::mf3` im **public** Bereich der Klasse `Derived` können sie sichtbar gemacht werden. Sollen nicht alle Operationen der Basisklasse in der Schnittstelle verfügbar sein, was für public inheritance fragwürdig ist, muss in der abgeleiteten Klasse für die gewünschten Operationen jeweils eine forward Methode wie in Listing 50 definiert werden.

Listing 50: forward Methode

```
class Derived{
   ...
   virtual void mf1(int i){
      Base::mf1(i);
   }

};
```

Namen in abgeleiteten Klassen überdecken die Namen aus der Basisklasse. Im Zusammenhang mit *public inheritance* ist das nicht erwünscht und kann durch eine `using` Deklaration oder durch entsprechende forward Methoden aufgehoben werden.

6.21 Sichtbarkeit und Lebensdauer von Objekten

6.21.1 Der Prozess im Haupspeicher

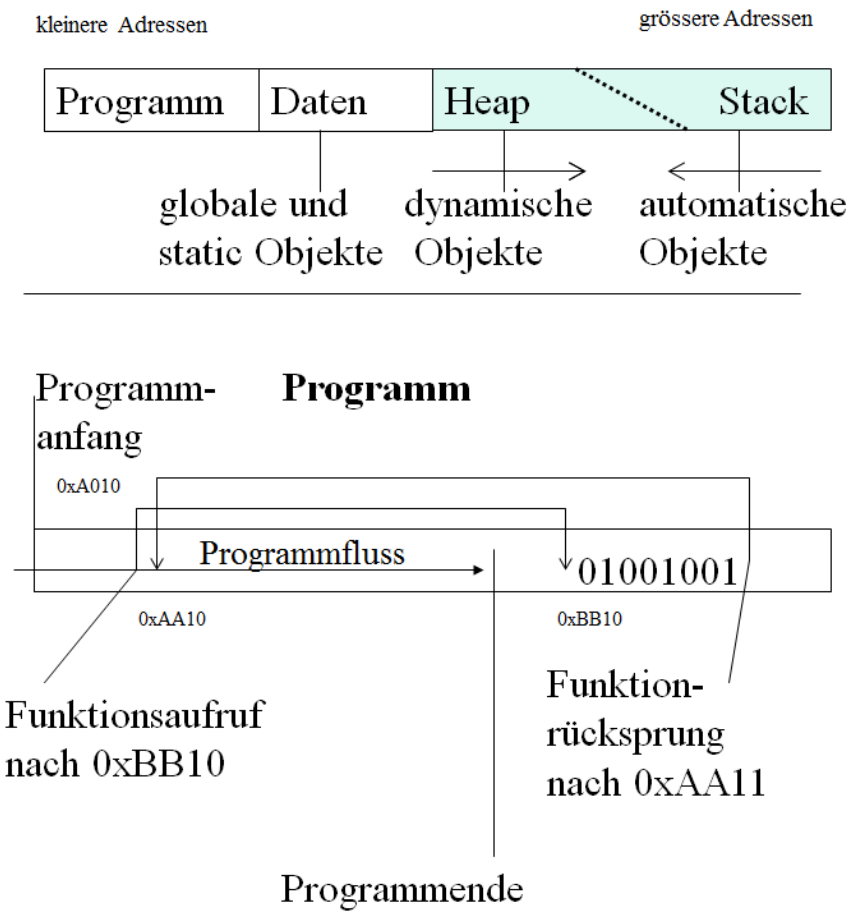

Abbildung 1: Der Prozess im Haupspeicher

Die Abbildung 1 zeigt ein vereinfachtes Modell eines Prozesses im Hauptspeicher. Die Adressen werden von links nach rechts größer. Globale Objekte und **`static`** deklarierte Objekte (Speicherklasse static) werden im Datenbereich angelegt.

Nicht statische lokale Objekte werden automatisch auf dem Stack erzeugt und wenn der Stack abgebaut wird, wieder zerstört.

Objekte die mit **`new`** erzeugt werden, landen auf dem Heap und müssen, wenn sie nicht mehr benötigt werden, mit `delete` wieder zerstört werden, dabei wird ihr Speicher wieder der Freispeicherverwaltung zurückgegeben.

6.21.2 Lokale, globale, externe und statische Objekte

Globale Objekte sind aus Sicht des Designs und aus wartungstechnischen Gründen verboten. Diesbezügliche Techniken sind hier nur wegen der Vollständigkeit dargestellt.

Zur Demonstration der Sichtbarkeit, der Lebensdauer und des Lebenszyklus von lokalen, globalen und statischen Objekten und einiger C++ Implementierungstechniken (Idiome) wird in der Anwendung aus Listing 51 auf der nächsten Seite die Klasse `C` und die Klasse `B` verwendet. Die Klasse `C` ist in Listing 52 auf Seite 65 und Listing 53 auf Seite 66, die KLasse `B` in Listing 54 auf Seite 66 definiert.

Den Konstruktoren der Klassen wird der Ort und die Art der Definition des Objekts als Text übergeben (z.B. in Listing 51 auf der nächsten Seite Zeile 7: `static C MgC("'Main.cpp static C mgC"')`). Die Objekte erzeugen darauf basierend in ihren Konstruktoren und Destruktoren jeweils eine Ausgabe, mit der der Zeitpunkt ihrer Erzeugung und Zerstörung nachvollzogen werden kann.

Soweit möglich werden vor der Definition von Objekten und vor und nach wesentlichen Codeabschnitten entsprechende Ausgaben erzeugt, mit deren Hilfe die Zuordnung der gesamten Ausgabe zu den jeweiligen Codeabschnitten möglich sein sollte.

Die Ausgabe der Anwendung ist in Listing 56 auf Seite 67 abgebildet.

Ausserdem werden in der Anwendung die Module 1 + 2 aus Listing 55 auf Seite 66 verwendet, die bis auf die Namen (Modul1, Modul2) und die Definition/-Deklaration des globalen Objekts `gC`, einmal `extern` einmal global, identisch sind. Sowohl in `main.cpp` als auch in den Modulen 1 und 2 wird jeweils ein modul globales Objekt **`static`** `C mgC(..)` definiert.

Im Header `Declarations.h` sind alle verwendeten Funktionsnamen deklariert.

In die Namen der Objekte ist ihre Speicherklasse und ihr Typ codiert:

- C für den Typ `C`
- l für local
- g für global
- mg für modul global
- s für **`static`**

`slC` steht also für ein lokales C das **`static`** deklariert ist.

In den Funktionen `showVisibility...()` wird jeweils der Name der Objekte `mgC` und `gC` ausgegeben.

Ein Objekt das in einer Übersetzungseinheit global, also ausserhalb einer Funktion oder einer Klasse definiert ist, ist im gesamten Programm sichtbar (`C gC(...)`). Wenn aus einer Übersetzungseinheit auf globale Objekte aus anderen Übersetzungseinheiten zugegriffen werden soll, muss der Name vorher durch eine `extern` Deklaration: `extern C gc;` bekannt gemacht werden.

Wird das Schlüsselwort **`static`** `C mgC(...);` bei der Definition verwendet, ist der Name nur innerhalb der Übersetzungseinheit[97] sichtbar (Modul global).

Globale Objekte von eingebauten Datentypen werden mit Null initialsiert. Es sollte aber immer eine explizite Initialisierung vorhanden sein!

[97] siehe section 4.2 auf Seite 30

Die Reihenfolge, in der globale Objekte initialisiert werden, kann nicht beeinflußt werden. Sie werden in umgekehrter Reihenfolge, in der sie initialisiert wurden, am Ende des Prozesses zerstört.

Listing 51: main() der Anwendung Sichtbarkeit und Lebensdauer

```
#include <iostream>
#include "C.h"
#include "Declarations.h"
using namespace std;

// irgendwo muss es ein Objekt gC geben
extern C gC;

// mgC ist nur in Main.cpp sichtbar
static C mgC("Main.cpp static C mgC");

int main()
{
   cout << "===== main() SichtbarkeitUndLebensdauer begin =====" << endl << endl;
   showVisibilityMain();
   showVisibilityModul1();
   showVisibilityModul2();
   cout << "===== main() C lC(...);" << endl;
   C lC("main() local C");
   cout << "===== main() createStaticInstanceModul1();" << endl;
   createStaticInstanceModul1();
   cout << "===== main() open block ===" << endl;
   {
      cout << "===== main() block opened ==" << endl;
      cout << "===== main() createStaticInstanceModul1();" << endl;
      createStaticInstanceModul1();
      cout << "===== main() createStaticInstanceModul2();" << endl;
      createStaticInstanceModul2();
      cout << "===== main() C blockC(...);" << endl;
      C blockC("main() blockC");
      cout << "===== main() static C staticC(...);" << endl;
      static C staticC("main() staticC");

      cout << "staticC: " << staticC.getName() << endl;
      cout << "== close block ==" << endl;
   }
   //cout << "staticC: " << staticC.getName() << endl;
   cout << "=== block closed ===" << endl;
   cout << "==== end main() ====" << endl;
}
void showVisibilityMain(){
   cout << "showVisibility() main.cpp" << endl;
   cout << "gC: " << gC.getName() << endl;
   cout << "mgC: " << mgC.getName() << endl;
}
```

Listing 52: C.h der Anwendung Sichtbarkeit und Lebensdauer

```
1  #ifndef C_H
2  #define C_H
3
4  #include <iostream>
5  #include <string>
6  #include "B.h"
7
8  class C
9  {
10 private:
11    std::string name;
12     static int classAttribute;
13     static B staticB;
14 public:
15    C(std::string const& name) : name(name)
16    {
17       classAttribute++;
18       std::cout << "C(" << name << ") No:" << classAttribute << std::endl;
19       createB();
20       //std::cout << "C::createB():" << createB().getName() << std::endl;
21       //std::cout << "C::staticB:" << staticB.getName() << std::endl;
22    }
23    ~C()
24    {
25       std::cout << "~C(" << name << ") No:" << classAttribute << std::endl;
26       classAttribute--;
27    }
28    std::string const& getName(){ return name; }
29
30    static B& createB() {
31       static B b("C::createB() static B");
32       return b;
33    }
34 };
35 #endif //C_H
```

Die Klasse `C` hat die privaten Klassenattribute **`static int`** `classAttribute` und **`static`** `B staticB` und ein Attribute `string name`. Für Klassenattribute muss Speicher durch eine Definition wie in Listing 53 auf der nächsten Seite für `B C::staticB (...)` oder **`int`** `C::classAttribute = 0;` zur Verfügung gestellt werden.

Die Reihenfolge, in der Klassenattribute initialisiert werden, kann nicht beeinflußt werden. Sie werden in umgekehrter Reihenfolge, in der sie initialisiert wurden, am Ende des Prozesses zerstört.
Im Gegensatz dazu kann eine Klassenoperation **`static`** `B& createB()` wie in Listing 52 `inline` deklariert werden. Sie kann eine Referenz auf ein lokales statisches Objekt zurückliefern. Dadurch kann der Zeitpunkt, wann das Objekt erzeugt wird, direkt durch den Aufruf der Methode beeinflußt werden. Ein solches Objekt wird am Prozessende zerstört.

Listing 53: C.cpp der Anwendung Sichtbarkeit und Lebensdauer

```
#include "C.h"

int C::classAttribute = 0;
B C::staticB("static C::B(...)");
```

Die Klasse B aus Listing 54 gibt lediglich ihren Namen im Konstruktor und im Destruktor aus. Die verschiedenen Möglichkeiten von einer Klasse zu erben oder einen Member der Klasse zu definieren und die daraus resultierenden Konsequenzen, sind in section 6.16 auf Seite 50 beschrieben. Hier soll nur gezeigt werden, wann Klassenattribute und statische Objekte in Funktionen erzeugt und zerstört werden.

Listing 54: B.h der Anwendung Sichtbarkeit und Lebensdauer

```
#ifndef B_H
#define B_H
#include <iostream>
#include <string>

class B
{
private:
   std::string name;
public:
   B(std::string const& name) : name(name)
   {
      std::cout << "B(" << name << ")" << std::endl;
   }
   ~B()
   {
      std::cout << "~B(" << name << ")" << std::endl;
   }
   std::string const& getName(){ return name; }
};
#endif //B_H
```

Listing 55: cpp Modul 1 und 2 der Anwendung Sichtbarkeit und Lebensdauer

```
#include <iostream>
#include "C.h"

// mgC ist nur in Modul1.cpp sichtbar
static C mgC("Modul1.cpp static C mgC");
//extern C gC;
C gC("Modul2.cpp global C gC");

void createStaticInstanceModul1()
{  // slC ist nur in dieser Funktion sichtbar
   static C slC("createStaticInstanceModul1() static C slC");
   C lC("createInstanceModul1() C lC");
}
```

```
14
15 using namespace std;
16 void showVisibilityModul1(){
17    cout << "showVisibility() Modul1.cpp" << endl;
18    cout << "gC: " << gC.getName() << endl;
19    cout << "mgC: " << mgC.getName() << endl;
20    //cout << "slC: " slC.getName() << endl;
21 }
```

In `main()` wird `createStaticInstanceModul1()` zweimal aufgerufen. Das lokale statische Objekt `slC` wird aber nur einmal initialisiert, das nicht statische Objekt `lC` wird jedesmal, wenn die Funktion gerufen wird, erzeugt und wenn die Funktion zurückkehrt und das Objekt aus dem Scope geht, wieder zerstört. Statische Objekte in Funktionen werden initialisiert, wenn der Kontrollfluss das erste mal ihre Definition erreicht und erst am Ende des Programms zerstört. Existieren mehrere Objekte im Datenbereich, werden diese in der Umgekehrten Reihenfolge ihrer Erzeugung zerstört.

Ausgabe in Listing 56 Zeile 1 bis 6:
Die Reihenfolge in der globale Objekte und Klassenattribute (**static** `B staticB;` in `C`) erzeugt werden, kann nicht beeinflusst werden, bzw. hängt davon ab, in welcher Reihenfolge der Linker die Objectfiles zusammenfügt.

In der Anwendung wird zuerst `mgC` aus Modul2 erzeugt, das die Klassenoperation `createB()` im Konstruktor aufruft, die das statische Objekt `B` erzeugt. Danach wird `mgC` aus Modul1 erzeugt, das ebenfalls die Klassenoperation `createB()` im Konstruktor aufruft, da das Objekt `B` **static** declariert ist, wird es nur beim ersten Aufruf initialisiert.

(letzte Zeile in Listing 56).

Listing 56: Ausgabe der Anwendung Sichtbarkeit und Lebensdauer

```
1  C(Modul2.cpp static C mgC) No:1
2  B(C::createB() static B)
3  C(Modul1.cpp static C mgC) No:2
4  C(Modul2.cpp global C gC) No:3
5  C(Main.cpp static C mgC) No:4
6  B(static C::B(...))
7  ===== main() SichtbarkeitUndLebensdauer begin =====
8
9  showVisibility() main.cpp
10 gC: Modul2.cpp global C gC
11 mgC: Main.cpp static C mgC
12 showVisibility() Modul1.cpp
13 gC: Modul2.cpp global C gC
14 mgC: Modul1.cpp static C mgC
15 showVisibility() modul2.cpp
16 gC: Modul2.cpp global C gC
17 mgC: Modul2.cpp static C mgC
18 ===== main() C lC(...);
19 C(main() local C) No:5
20 ===== main() createStaticInstanceModul1();
```

```
C(createStaticInstanceModul1() static C slC) No:6
C(createInstanceModul1() C lC) No:7
~C(createInstanceModul1() C lC) No:7
===== main() open block ===
===== main() block opened ==
===== main() createStaticInstanceModul1();
C(createInstanceModul1() C lC) No:7
~C(createInstanceModul1() C lC) No:7
===== main() createStaticInstanceModul2();
C(createStaticInstanceModul2() static C slC) No:7
C(createInstanceModul2() C lc) No:8
~C(createInstanceModul2() C lc) No:8
===== main() C blockC(...);
C(main() blockC) No:8
===== main() static C staticC(...);
C(main() staticC) No:9
staticC: main() staticC
== close block ==
~C(main() blockC) No:9
=== block closed ===
==== end main() ====
~C(main() local C) No:8
~C(main() staticC) No:7
~C(createStaticInstanceModul2() static C slC) No:6
~C(createStaticInstanceModul1() static C slC) No:5
~B(static C::B(...))
~C(Main.cpp static C mgC) No:4
~C(Modul2.cpp global C gC) No:3
~C(Modul1.cpp static C mgC) No:2
~C(Modul2.cpp static C mgC) No:1
~B(C::createB() static B)
```

Ein lokales Objekt wird initialisiert, wenn der Kontrollfluss seine Definition erreicht[98]. Sein Name ist danach sichtbar und überdeckt gegebenenfalls die Namen von Objekten in einem umgebenden Scope[99]. Alle Objekte bis auf `gc`, `mgc` und den Attributen der Klassen sind lokal.

Eingebaute Datentypen werden nicht mit einem Defaultwert initialisiert, müssen also explizit initialisiert werden: `int i = 0; int j{}`[100]. Für Objekte von Klassen wird ein Konstruktor und beim Verlassen des Scopes, der Destruktor gerufen.

Objekte, die innerhalb einer Funktion (z.B. `createStaticInstanceModulx()`) mit der Speicherklasse `static` definiert sind, werden nur einmal, wenn der Kontrollfluss das erste mal ihre Definition erreicht, initialisiert und bei Programmende, in umgekehrter Reihenfolge ihrer Erzeugung, zerstört.

Attribute (UML) oder Member (C++) von Klassen sind im Scope der Klasse sicht-

[98] [Str00a] 7.1.2 Statische Variable
[99] siehe section 3.7 auf Seite 23
[100] siehe section 6.14 auf Seite 47

bar, der Zugriff von außen hängt von den access specifiern[101] ab, ihre Lebensdauer von der Lebensdauer des Objekts, in dem sie enthalten sind.

Attribute von Klassen die mit dem Schlüsselwort **`static`** deklariert sind[102]; existieren nur einmal im Process und werden beim Programmstart initialisiert und am Prozessende zerstört.

Der Wert von `si` wird beim ersten Aufruf mit 0 initialisiert und bei jedem Aufruf (1 bis n) der `function` erhöht. Der Wert von `ai` wird bei jedem Aufruf wieder neu initialisiert.

Die Variable `pi` nimmt die Adresse des int Objekts, das mit **`new`** erzeugt wird auf. Der Speicher wird am Ende der Funktion wieder freigegeben, ohne das delete wäre der Speicher für die Laufzeit des Programms verloren, da niemand mehr die Adresse kennt.

Die Sichtbarkeit der Namen `si` und `ai` ist auf den Scope des Blockes ab der Stelle ihrer Deklaration und nachfolgender innerer Blöcke beschränkt. Werden in inneren Blöcken die gleichen Namen verwendet, werden die Namen aus dem umgebenden Scope überdeckt. Vermeiden Sie Namensüberdeckung solange sie den gewünschten Effekt anders erreichen können. In verschiedenen Sprachen (Java) ist die Deklaration eines gleichen Namens in inneren Bereichen explizit verboten.

6.22 Eingebaute Datentypen und Literale

C++ stellt einige in die Sprache eingebettete arithmetische Datentypen, sogenannte fundamentale Datentypen, zur Verfügung.
Es gibt in C++11 insgesamt 18 elementare Datentypen (in C++03 gab es nur 14 elementare Datentypen). Eine gute online Übersicht ist in `http://en.cppreference.com/w/cpp/language/types` zu finden.
Im Einzelnen werden diese bei Bedarf besprochen. Hier sollen sie nur der Vollständigkeit halber mit ihren möglichen `modifiern` und mit ihrer vom Standard garantierten Mindestgröße aufgelistet werden.

Literale sind konstante Werte die im Code direkt angegeben werden. Sie haben einen default Typ der durch anhängen von Buchstaben verändert werden kann. Literale sollten nur zur Initialisierung einer Konstanten verwendet werden. Durch den Namen der Konstanten sollte die Bedeutung des Wertes zum Ausdruck gebracht werden. Damit wird die Lesbarkeit durch den Namen der Konstanten gewährleitet und die Wartbarkeit des Codes durch die eine Stelle, an der der Wert festgelegt wird, gewährleistet. Literale die im Code verwendet werden, werden als „magic numbers“ bezeichnet, weil sich ihre Beutung nicht durch ihren Wert erschließt.

Die Größe eines Objekts in Byte kann mit dem `sizeof`[103] Operator ermittelt werden. in Listing 57 auf der nächsten Seite sind die Typen und ihre Ordnung darge-

[101] siehe section ?? auf Seite ??
[102] siehe section **??** auf Seite ??
[103] Siehe section 6.25.17 auf Seite 102

stellt.

Listing 57: typische Größe der arithmetischen Datentypen

```
1 == sizeof(char) <= sizeof(short)
   <= sizeof(int) <= sizeof(long) <= sizeof(long long)
1 <= sizeof(bool) <= sizeof(long)
sizeof(char) <= sizeof(wchar_t) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
sizeof(N) == sizeof(signed N) == sizeof(unsigned N)
```

N kann von einem der Typen: **char**, **short int**, **int**, **long int** oder **long long int** sein.

Tabelle 3: Eingebaute Datentypen und Literale

Typ/Modifier	sizeof()	Wertebereich, Compiler abhängig	Zweck	Konstanten / Literale
bool	?	true, false	Darstellung von Wahrheitswerten	true, false
unsigned char (signed oder unsigned nicht standardisiert)	1	0 - 255	Darstellung von Zeichen (ANSI-Zeichensatz)	'a','9','\n' const char* "chars"
signed char	1	-128 bis 127	Ganzzahlen	
wchar_t typedef (unsigned short)[104]	2	0 - 65535	Unicode Zeichensatz	\42 Oktaler Wert \x40 hex wert wchar_t* L"wide chars"
char16_t	2	NA	Unicode character	u"char16"
char32_t	4	NA	Unicode character	U"char32"
[signed] short	min 2	-32768 bis 32767	Ganzzahlen	
unsigned short	min 2	0 bis 65535	Ganzzahlen	
[signed] int	min 2	-2147483648 bis +2147483647	Ganzzahlen	default Typ dezimal: 12, -3 oktal: 0777 hex: 0xFF
unsigned int	min 2	0 bis 4294967295	Ganzzahlen	12u, 12U, 0xFFU
[signed] long	min 4	-2147483648 bis +2147483647	Ganzzahlen	12L, 12l, -3L Großbuchstaben bevorzugen

[104] siehe section 6.23.2 auf Seite 73

unsigned long	min 4	0 bis 4294967295	Ganzzahlen	12ul, 12UL
[signed] long long	min 8	±9.223E18	Ganzzahlen	3ll, 3LL
unsigned long long	min 8	0 bis 18.447E18	Ganzzahlen	5ull, 5ULL
float	6 Bits	±3.4E+38 1.2E-38 kleinste positive Zahl	Gleitpunktzahlen 6 Dezimalstellen Genauigkeit	5.19f, 12.F
double	10Bits	±1.7E+308 2.3E-308 kleinste positive Zahl	Gleitpunktzahlen 15 Dezimalstellen Genauigkeit	5.19 Default Typ
long double	10Bits	±1.1E+4932 3.4E-4932 kleinste positive Zahl	Gleitpunktzahlen 19 Dezimalstellen Genauigkeit	1.23l, 1.23L

Die Gleitpunktzahlen (**`float`**, **`double`**, **`long double`**) sind nach IEEE 754 / IEC-60559:1989 Standard codiert[105].

6.22.1 Wertebereichsüberschreitungen Overflow/Underflow/Truncation

Der Überlauf von `signed` **`int`** und seinen Verwandten **`short`**, **`long`**, ... (*signed integer types*) führt zu `undefined behavior`[106]. Der Grund dafür sind die verschiedenen Möglichkeiten negative Werte zu repräsentieren. Häufig wird zwar das 2-er Komplement verwendet, bei dem der Überlauf ganz *natürlich* zu einem sogenannten *wraparound* beim Überlauf führt und der Wertebereich von unten wieder betreten wird, aber das ist nicht auf allen Plattformen gegeben. Für `unsigned` Typen wird aber immer die binäre Repräsentation der Werte verwendet und daher kann dieses Verhalten auf allen Plattformen garantiert werden.

Ganzzahlige numerische Fehler können heimtückisch, teuer und für Angriffe ausnutzbar sein. Diese Fehler beinhalten overflows, underflows, verlustbehaftete Abschneidung von Werten (z.B. cast int -> short) und die illegale Anwendung von Operatoren, wie z.B. den Bitshift << Operator.

Das Listing 58 auf der nächsten Seite zeigt einige dieser Ausdrücke und ihr Ergebnis.

[105] https://de.wikipedia.org/wiki/IEEE_754
[106] [WD12]

Listing 58: Ausdrücke die zu Überläufen und undefined behavior führen

```
template<class T> using nl = std::numeric_limits<T>;

nl<unsigned int>::max()+1  // 0 ok, wrapping around
nl<int>::max()+1           // undefined
nl<long>::max()+1          // undefined
nl<short>::max()+1         // nl<short>::max()+1 if nl<int>::max() > nl<short>::max
    ()
                           // sonst undefined
-nl<int>::min()            // undefined
static_cast<char>(nl<int>::max()) // commonly -1
1<<-1                 // undefined
1<<0                  // 1
1<<31                 // undefined (32 Bit integer) seit C++11 vorher nl<int>::min()
1<<32                 // undefined (32 Bit integer)
1/0                   // undefined
nl<int>::min() % -1   // undefined in C++11
```

Der Überlauf von `unsigned int` entspricht dem erwarteten Verhalten, das Ergebnis ist `0` [107]. Programmteile deren Algorithmen darauf basieren sollten entsprechend gekennzeichnet sein! Eine Studie mit der Vorstellung eines Werkzeugs zur Identifikation solcher *Programmierfehler* ist in [WD12] beschrieben.

6.22.2 Datentypen für den embedded Bereich

Mit dem Standard C++11 wurden im Header `<cstdint>`[108] die Datentypen aus dem Standard C99[109] für den embedded Bereich übernommen.

Diese garantieren

- mit `intX_t` einen Typ der exact X Bits belegt, z.B. `int8_t`.
- mit `int_fastX_t` einen Typ der mindestens X Bits enthält und am schnellsten auf der jeweilgen Zielmaschine verarbeitet werden kann, z.B. `int_fast8_t`. Auf einem 16 Bit Rechner könnte das ein 16 Bit Typ sein!
- mit `int_leastX_t` den kleinsten Typ mit dem diese Anzahl Bits repräsentiert werden kann.

Eine ausführliche Diskussion dieses Themas ist bei embeddedgurus[110] zu finden.

[107] C++ Standard §3.9.1.4 Fundamental types
[108] http://en.cppreference.com/w/cpp/types/integer
[109] http://en.cppreference.com/w/c/types/integer
[110] http://embeddedgurus.com/stack-overflow/2008/06/efficient-c-tips-1-choosing-the-correct-integer-size/

6.23 Textkonstante / String Literal

Eine Textkonstante ist eine Zeichenkette die in Hochkomma eingeschlossen ist: `"Textkonstante"`. Der Typ des Ausdrucks ist **`char const *`**, ein Zeiger auf einen konstanten **`char`**. Der Wert ist die Adresse des ersten Zeichens. Es ist ein Fehler, wie in Listing 59, die Textkonstante zu ändern. Die Initialisierung von `p` quitiert der Compiler mit einem Warning. Die Änderung kann zur Laufzeit fatale Auswirkungen haben, z.B. wenn die Textkonstante im ROM abgelegt wird.

Listing 59: Textkonstante ändern

```
1 //warning: ISO C++ forbids converting a string constant to 'char*'
2 char *p = "textkonstante";
3 p[0] = 'E'; // Run-Time Error, textkonstante dürfen nicht verändert werden!
```

Die Lebensdauer der Textkonstanten ist über die gesamte Prozesslaufzeit. Daher können Textkonstanten als Argumente und Rückgabewerte verwendet werden. Ob zwei gleiche Textkonstanten identisch sind, also nur einmal Speicher belegen oder nicht ist Implementierungsabhängig.

Mit C++11 ist es möglich raw string und multibyte/wide-character string Literale zu definieren, sowie Benutzer definierte suffixe.

6.23.1 Raw String Literale[111]

Sollen in einem Literal der Form: `"Hello"`, Sonderzeichen in der Zeichenkette verwendet werden, müssen diese mit einem Backslash `\` maskiert werden. Im Falle von Regulären Ausdrücken oder Dateipfaden führt das zu unleserlichem und schwer wartbarem Code.

Mit Raw-Strings besteht das Problem nicht, da die Zeichenkette geschrieben werden kann, wie sie gewünscht ist, inclusive Zeilenschaltungen.

Definition: `R"(Das␣ist␣ein␣Raw-String␣mit␣\Sonderzeichen\)"`.

Um auch die runden Klammern in Raw Strings verwenden zu können, kann ein Begrenzungszeichen / delimiter benutzt werden. Die abstrakte Syntax ist: `R"delim(Raw␣String)delim"` wobei der delimiter maximal 16 Zeichen lang sein und nicht die Zeichen backslash `\`, whitespaces und Klammern enthalten darf.

6.23.2 Vordefinierte encoding prefixes

Mit einem *encoding prefix* kann ein bestimmtes character encoding für String Literale festgelegt werden. Die folgenden *encoding prefixes* sind definiert:

- **u8** definiert ein UTF-8 string Literal mit dem Zeichentyp **`const char`**
- **u** definiert ein string Literal mit dem Zeichentyp `char16_t`

[111] [Jos12] New String Literals

- **U** definiert ein string Literal mit dem Zeichentyp `char32_t`
- **L** definiert ein wide string Literal mit dem Zeichentyp `wchar_t`

Zum Beispiel definiert `L"Hello"` ein `wchar_t` string Literal.

6.23.3 Benutzerdefinierte suffixes C++11[112]

Mit dem **Literal Operator** `operator""_suffix` können für die eingebauten Datentypen Integer, Floatingpoint, Strings und char Benutzer definierte Suffixes für Literale definiert werden, die daraus Literale Benutzer definierter Typen machen.

Das Listing 60 zeigt die Definition eines Literal Operators für doubles die Kilometer darstellen. Der Literal Operator soll mit einem Unterstrich beginnen (km), um Konflikte mit zukünftigen standard Suffixes zu vermeiden.

Listing 60: Benutzer definierte suffixes

```
1  class Distance{
2     long double d;
3     char const* unit;
4  public:
5     constexpr Distance(long double d, char const* u):d(d), unit(u){}
6     friend
7     std::ostream& operator<<(std::ostream& out, Distance const& distance){
8        out << distance.d << " " << distance.unit;
9        return out;
10    }
11 };
12
13 constexpr Distance operator""_km(long double d){ //cooked-form
14    return Distance(d, "km");
15 }
16 Distance operator""_cm(char const* dc){ //uncooked-form
17    std::cout << "operator\"\"_cm(char const* dc): " << dc << std::endl;
18    long double d = std::atof(dc);
19    return Distance(d, "cm");
20 }
21 void demoDistance(){
22    cout << "demoDistance()" << endl;
23
24    cout << 5.0_Km << endl;
25    cout << 5.2_cm << endl;
26 }
27 Ausgabe:
28 demoDistance()
29 5 Km
30 operator""_cm(char const* dc): 5.2
31 5.2 cm
```

[112][Gri12]

Der Parameter Typ des Literal Operators kann für Integer und Floatingpoint Literale als `char const* literal` (uncooked-form) oder als den zugrundeliegenden Datentyp wie in Listing 60 auf der vorherigen Seite (cooked-form) deklariert werden. Für String und `char` kann nur die cooked-form verwendet werden. Die uncooked-form hat die höhere Priorität.

Tabelle 4: Parameter Typen der Literal Operatoren

Datentyp	uncooked-form	cooked-form
Integer Typen	`const char*`	unsigned long long
Floating Point Typen	`const char*`	long double
Strings	`(const char*, size_t)`	
char	char	

6.24 Statements / Anweisungen, die Struktur eines Programmes

Die konventionellen und **grundlegenden syntaktischen Elemente eines C++ Programmes sind Anweisungen und Ausdrücke**. Alles ist entweder eine Anweisung oder ein Ausdruck[113].

Anweisungen / Statements werden wegen ihrer Effekte bei der Ausführung durch den Rechner verwendet, sie haben keine Werte. Ein objektorientiertes Programm besteht aus Objekten die sich Nachrichten senden, der Code der die Nachrichten versendet und verarbeitet, sind Ausdrucks-Anweisungen.
Ein **Programm** ist eine Folge von Anweisungen, die nacheinander ausgeführt werden. In multithreaded Programmen können Anweisungen auch gleichzeitig ausgeführt werden, aber innerhalb eines Threads auch nur eine nach der anderen.

Im Folgenden sollen einige Anweisungen von C/C++ vorgestellt werden.

Von der Sprache C übernommene Anweisungen:

expression statement / Ausdrucksanweisung Ein Ausdruck der mit einem Semikolon abgeschlossen wird:`a = b + c;`

Compound Statement (Block) / Zusammengesetzte Anweisung `{...}`, geschwungene Klammern, alle Anweisungen innerhalb werden syntaktisch als eine Anweisung betrachtet. Ein Compound Statement eröffnet einen neuen Scope.

Selection / Auswahl, Verzweigung `if/else, switch`

Iteration / Wiederholung `for, while, do`

empty statement / Leeres Statement ein Semicolon `;`

C++-Erweiterungen

[113] siehe section 6.25 auf Seite 81

declaration statement / Deklaration `int i;`

throw Statement löst eine Exception aus

try-block, catch-block `try{ ...}catch(...){ }` Die Struktur um ein Statement, das eine Exception werfen könnte und diese gegebenenfalls auffängt[114]

6.24.1 Ausdrucksanweisung, leere und zusammengesetzte Anweisung

Die Berechnung eines Ausdrucks ist die am häufigsten verwendete Anweisung, (Ausdruck mit Semikolon). Solche Anweisungen sind normalerweise Zuweisungen oder Funktionsaufrufe. Alle Nebenwirkungen des Ausdrucks werden abgeschlossen, bevor die nächste Anweisung ausgeführt wird.

Listing 61: Ausdrucksanweisungen

```
1 a = b + c;
2 objekt.nachricht(); zeiger->nachricht();
```

Fehlt der Ausdruck, wird sie als leere Anweisung bezeichnet. In einer for Schleife kann die ganze Arbeit schon im letzten Statement erledigt sein, da die **for** Schleife aber mindestens eine Anweisung benötigt, kommt in Listing 62 die leere Anweisung zur Verwendung.

Listing 62: Empty Statement

```
1 for( init; condition; statement)
2    ; // empty Statement
```

Damit mehrere Anweisungen verwendet werden können, wo eine einzelne Anweisung erwartet wird, gibt es die zusammengesetzte Anweisung `{ }`, die auch als Block bezeichnet wird. Die Verzweigungen und Wiederholungen in C++ binden genau eine Anweisung (syntax). Werden mehrere Anweisungen benötigt, muss eine zusammengesetzte Anweisung verwendet werden. Der Funktionskörper einer Funktionsdefinition ist ein Block.

Listing 63: Funktionskörper

```
1 void f()
2 {
3    // Funktionskörper
4 }
```

6.24.2 Bedingungsausdrücke, Condition

In Wiederholungen und Verzweigungen lassen sich komplexe Ausdrücke[115] als Bedingungen formulieren.

[114] siehe section 6.34 auf Seite 143
[115] siehe section 6.25 auf Seite 81

Der Ergebnistyp muss ein bool oder ein in `bool` konvertierbarer Typ sein. Alle ganzzahligen Typen und Zeiger können nach bool konvertiert werden, wobei der Wert 0 für **`false`** und alle anderen Werte für **`true`** stehen.

6.24.3 Verzweigungen, Auswahlanweisungen

`if / else, switch / case` sind die Auswahlanweisungen. Eine **`switch`**-Anweisung testet einen **ganzzahligen Wert** gegen einen Menge von **Konstanten**. Diese müssen verschieden sein. Wenn der getestete Wert mit keiner Fallkonstanten übereinstimmt, wird der **`default`**-Zweig ausgewählt. Der **`default`**-Zweig ist optional, sollte aber per Codierrichtlinie immer vorhanden sein.

Listing 64: switch/case Anweisung

```
1  int var = 2;
2  switch(var) {
3     case 1:
4        cout << "Eins!" << endl;
5        break;
6     case 2:
7        cout << "Zwei!" << endl;
8        [[fallthrough]]; // Attribut seit C++17 -> kein warning
9     case 3:
10       cout << "Zwei oder Drei!" << endl;
11       break;
12    default:
13       cout << "default ist optional" << endl;
14 }
```

Ohne **`break`** würden alle nachfolgenden Anweisungen nach dem ausgewählten **`case`** ausgeführt. **`break`** verzweigt ans Ende der `Switch` Anweisung.

`if` erlaubt komplexe Bedingungsausdrücke in den runden Klammern. Ist der Wert **`true`**, wird die Anweisung die zu **`if`** gehört ausgeführt, ist er **`false`**, wird die Anweisung die zu **`else`** gehört, ausgeführt. Der **`else`** Zweig ist optional. **`if`** und **`else`** binden genau eine Anweisung. Sollen mehrere Anweisungen gebunden werden, muss ein Compound-Statement, ein Block verwendet werden.

Listing 65: if/else Anweisung

```
1  if(Bedingungsausdruck)
2     eine Anweisung;
3  else // ist optional
4     eine Anweisung;
```

6.24.4 Wiederholungen, break und continue

`for, while` und **`do while`** binden wie **`if/else`** eine Anweisung. Sollen mehrere Anweisungen gebunden werden, muss ein Block verwendet werden. Wiederholun-

gen werden auch Schleifen genannt. Der Schleifenkörper wird wiederholt ausgeführt, solange der Bedingungsausdruck den Wert **true** liefert.

Listing 66: Iteration Statements / Wiederholungsanweisungen

```
1  while(Bedingungsausdruck) // Ausführungshäufigkeit: 0 - n
2     eine Anweisung; // Schleifenkörper
3
4  do // Ausführungshäufigkeit: 1 - n
5     eine Anweisung; // Schleifenkörper
6  while(Bedingungsausdruck);
7
8  // Ausführungshäufigkeit: 0 - n
9  for(for-init-Anweisung; Bedingungsausdruck;letzte-for-Anweisung)
10    eine Anweisung; // Schleifenkörper
```

Mit **break** wird der Schleifenkörper verlassen und zur Anweisung nach der Wiederholungsanweisung verzweigt. Mit **continue** wird zum Bedingungsausdruck verzweigt.

Listing 67: break und continue

```
1  for(int i=0; i < 5; ++i) {
2     if(i == 1)
3        continue;
4     if(i == 4)
5        break;
6     cout << "i:" << i;
7  }
```

Die Ausgabe in Listing 67 wird `i:0 i:2 i:3` sein. Die letzte Anweisung `++i` wird bei **continue** ausgeführt.

6.24.5 Range-Based `for` Schleife

Mit C++11 kommt eine neue Art der For-Schleife dazu, mit der über alle Elemente eines Ranges, eines nativen Arrays, über STL-Container, `std::string`s und die neuen Datentypen `std::array` und Initialisiererlisten iteriert werden kann. [116]

Die allgemeine Form ist in Listing 68 abgebildet. Wobei `decl` die Deklaration der Laufvariablen ist, die bei jedem Schleifendurchlauf mit dem nächsten Element aus `coll` initialisiert wird und in `statement` über den deklarierten Namen zur Verfügung steht.

Listing 68: Range Based For Loop Syntax

```
1  for(decl : coll){
2     statement
3  }
```

[116] [Gri12] und [Jos12]

Der Basistyp der Laufvariablen ist der Elementtyp der Menge. Durch entsprechende Modifier kann er, wie in Listing 69, angepasst werden.

In Listing 69 wird die die Laufvariable als Referenz deklariert (Modifier &), wodurch die Elemnente in der Menge verändert werden können.

Listing 69: Range-basierte For-Schleife über C-Style Array

```
1 int ai[] = {0, 8, 15};
2 for (int& x : ai) x *= 2;
3 for (int x : ai) cout << x << " ";
4 cout << endl;
```

Die Initialisiererliste in Listing 70 ermöglicht auf einfache Weise über eine vorgegebene Menge beliebiger Werte zu iterieren.

Listing 70: Range-basierte For-Schleife über Initialisiererliste

```
1 for (auto x : {2, 3, 7, 19, 27})
2    std::cout << x << ", ";
3 std::cout << std::endl;
```

Die Semantik der Range-basierten For-Schleife in Listing 71 ist equivalent zur Darstellung in Listing 72.

Listing 71: Range-basierte For-Schleife über Vektor

```
1 vector<string> coll = {"zwo-und-dreissig", "sechzehn", "acht", "Rosie"};
2 for (auto x : coll)
3    std::cout << x << " ";
4 cout << endl;
```

Listing 72: Range-basierte For-Schleife equivalent

```
1 vector<string> coll = {"zwo-und-dreissig", "sechzehn", "acht", "Rosie"};
2 for (decltype(coll)::iterator pos = coll.begin(), end = coll.end();
3    pos != end;
4    ++pos)
5 {
6    auto x = *pos;
7    std::cout << x << " ";
8 }
9 cout << endl;
```

Dies entspricht auch dem Vorgehen des Compilers beim Auflösen einer solchen Schleife. Zuerst wird versucht, die Iteratoren mit den Membermethoden des Mengenobjekts `coll.begin()` und `coll.end()` zu erzeugen. Wenn das fehlschlägt, werden die globalen Funktionen `begin(coll)` und `end(coll)`, die das Mengenobjekt als Argument übergeben bekommen, verwendet, um die Iteratoren zu erzeugen.

In Listing 73 auf der nächsten Seite wird über eine Map iteriert, der Typ der Laufvariablen ist entsprechend ein `std::pair<...>`.

Listing 73: Range-basierte For-Schleife Map

```
map<string, string> phonebook = {
   {"Bjarne Stroustrup", "+1 (212) 555-1212"},
   {"Gerd Hirsch", "+49 (7062) 930951"},
   {...}
};
for{auto val : phonebook}
   cout << val.first << ": " << val.second << endl;
cout << endl;
```

Benutzerdefinierte Typen und Range-Based-For-Loop

Soll über einen benutzer definierten Typ, z.B. `Range` aus Listing 74, der nicht über die Memberoperationen `begin/end` verfügt, iteriert werden, kann der Iterator non-invasiv durch die Überladung der globalen Funktionen `begin(Range&)` und `end(Range&)` oder dieser Funktionen im selben namespace, in der der Typ definiert ist, zur Verfügung gestellt werden, wie in Listing 74 gezeigt.

Listing 74: Range-basiert For non-invasiv benutzerdefinierter Typ

```
namespace myns{
class Range{
   int lower;
   int upper;
public:
   Range(int lower, int upper):lower(lower),upper(upper){}
   int getLower(){return lower;}
   int getUpper(){return upper;}
};

class RangeIterator{
   int value;
public:
   RangeIterator(int value): value(value){}
   RangeIterator& operator++() {
      ++value;
      return *this ;
   }
   int operator*(){
      return value;
   }
   bool operator!=(RangeIterator& rhs){
      return value != rhs.value;
   }
};
RangeIterator begin(Range& beg){
   return RangeIterator(beg.getLower());
}
RangeIterator end(Range& end){
   return RangeIterator(end.getUpper());
}
} // end namespace myns
```

```
33 void rangeDemo(){
34    myns::Range r(3,7);
35
36    for(auto m : r)
37       cout << m << " ";
38    cout << endl;
39 }
40
41 Ausgabe:
42 3 4 5 6
```

Die einzige Bedingung ist, die Funktionen müssen ein Objekt zurückliefern, das die Schnittstelle eines Forward Iterators hat.

Alternativ könnte der benutzer definierte Typ `Range` wie in Listing 75 mit den Operationen `begin()` und `end()` ausgestattet werden, die genauso implementiert werden wie ihre globalen Verwandten.

Listing 75: Range-basierte For-Schleife – Invasiv

```
1  class Range{
2     int lower;
3     int upper;
4  public:
5     Range(int lower, int upper):lower(lower),upper(upper){}
6     int getLower(){return lower;}
7     int getUpper(){return upper;}
8
9     RangeIterator begin(){ return RangeIterator(lower);}
10    RangeIterator end(){ return RangeIterator(upper);}
11 };
```

6.24.6 Deklaration

C-Programmierung: Am Anfang des Blockes, vor der ersten Anweisung können Deklarationen erfolgen, danach Anweisungen.

C++-Programmierung: Deklarationen sind Anweisungen, können also überall stehen, wo Anweisungen erlaubt sind. Deklarieren Sie Objekte erst, wenn diese sinnvoll initialisiert werden können.

6.25 Ausdrücke und Operatoren

In diesem Kapitel wird das grundlegende Konzept der Ausdrücke / Expressions, Operatoren und der Überladung / Overloading von Operatoren vorgestellt[117].

[117] siehe section 6.24 auf Seite 75

6.25.1 Allgemeines

Ausdrücke / **Expressions** werden berechnet! ...und können Seiteneffekte haben! Das Ergebnis der Berechnung, der Wert, liegt in der Form eines Typs, dem **Ergebnistyp** der Operation oder des Operators vor. Alles was der Compiler zu sehen bekommt und nicht eine Anweisung ist, ist ein Ausdruck.
Jeder Ausdruck ist charakterisiert durch zwei voneinander unabhängigen Eigenschaften: dem *type* und der *value category*. Jeder Ausdruck hat einen *non-reference type* und gehört zu genau einer der drei primären *value categories*[118].

Die Kategorien werden durch zwei Eigenschaften bestimmt: Der Ausdruck

1. hat eine Idendität: es ist möglich zwei Ausdrücke miteinander zu vergleichen und festzustellen, ob sie auf dieselbe *Entität* verweisen, z.b. durch Vergleich der Adressen
2. ist *movable*: es existiert ein move constructor und/oder ein move assignment operator

Die Kombination dieser beiden Eigenschaften bestimmt die *value category* des Ausdrucks.

- hat eine Identität und ist nicht movable: -> value category: lvalue expression (left value)
- hat eine Identität und ist movable -> value category: xvalue expression (eXpiring value). Ein Ausdruck, der zwar eine Adresse hat, aber innerhalb der nächsten Anweisungen seine Gültigkeit verliert (expire)
- hat keine Identität und ist movable -> value category: prvalue (pure rvalue) expression
- hat keine Identität und ist nicht movable: nicht benutzt

Für die weitere Betrachtung in diesem Script, genügt meistens folgende vereinfachte Anschauung: Ausdrücke können von der Kategorie **R-Value** oder **L-Value** sein. L-Value Ausdrücke haben einen Platz im Haupspeicher, eine **L**ocation. Von ihnen kann mit dem Adress Operator[119] (&) die Adresse ermittelt werden. R-Value Ausdrücke befinden sich häufig im **R**egister, sind also temporäre Objekte von denen keine Adresse ermittelt werden kann oder sind Literale oder `constexpr`. L-Value Ausdrücke können rechts und links einer Zuweisung stehen, solange sind nicht immutable sind, R-Value Ausdrücke nur rechts.

Bsp.:
```
int a, b = 1, c = 2; //Deklaration mit (teilweiser)Initialisierung
a = b + c; //Expression Statement
```
`b + c` ist ein R-Value Ausdruck, dem Ergebnis kann nichts zugewiesen werden und seine Adresse kann nicht ermittelt werden, der Ausdruck `&(b + c)` ist ungültig und wird vom Compiler zurückgewiesen,
`a` ist ein L-Value Ausdruck, dem Namen `a` ist Speicherplatz zugeordnet in den geschrieben werden kann, der Ausdruck `&a` ist gültig.

[118] weitere Details: http://en.cppreference.com/w/cpp/language/value_category
[119] section 6.28 auf Seite 107 und section 6.25.3 auf Seite 86

Operatoren werden dazu verwendet, **Ausdrücke** zu formulieren. Der Typ der **Operanden** bestimmt, welche Operatoren auf sie angewendet werden können. Operatoren die nur einen Operanden haben, werden unäre / unary Operatoren genannt. Es gibt unäre Operatoren die vor (prefix `-a` unäres minus) und welche die nach (postfix `a++`) dem Operand stehen. Binäre / binary Operatoren haben zwei Operanden und stehen zwischen (infix `a+b`) den Operanden. Es gibt einen ternären Operator mit 3 Operanden, den Konditionaloperator[120], der es ermöglicht, eine Verzweigung als Ausdruck zu formulieren: `int a = b < c ? b : c;`

6.25.2 Vorrangregeln, Priorität und Assoziativität

Die Tabelle 5 stellt die Priorität (Spalte P) der Operatoren von C++ in absteigender Reihenfolge dar. Je kleiner die Zahl, desto höher die Priorität.

Sind an einem komplexen Ausdruck Operatoren mit gleicher Priorität beteiligt, entscheidet die Assoziativität (Bindungsrichtung, Spalte A) über die Auswertungsreihenfolge. Die unären Operatoren und die Zuweisungsoperatoren sind rechts bindend (R), alle anderen links bindend (L). In section 6.25.5 auf Seite 89 sind einige anschauliche Beispiele mit arithmetischen Ausdrücken beschrieben.

In der Spalte „überladen (Ü)“ bedeutet

- N: kann nicht überladen werden,
- M: kann nur als Member überladen werden
- ohne: kann global oder als Member überladen werden

Tabelle 5: Vorrangsregeln und Assoziativität der Operatoren

P	A	Beschreibung	Operator	Ü
1	L	Bereichsauflösung	`Klassenname::element`	N
		Bereichsauflösung	`namensbereichsname::element`	N
		global	`::name`	N
		global	`::qualifizierter-name`	N
2	L	Elementselektion (Auswahloperator)	`objekt.element`	N
		Elementselektion (Auswahloperator)	`zeiger->element`	M
		Indizierung	`zeiger[ausdruck]`	M
		Funktionsaufruf	`ausdruck(ausdrucksliste)`	M
		Werterzeugung (Konstruktor)	`typ(ausdrucksliste)`	M
		Postinkrement	`lvalue++`	
		Postdekrement	`lvalue--`	
		Typidentifikation	`typeid(typ)`	
		Laufzeit-Typinformation	`typeid(ausdruck)`	
		zur Laufzeit geprüfte Konvertierung	`dynamic_cast<typ>(ausdruck)`	

[120] siehe section 6.25.16 auf Seite 102

		zur Übersetzungszeit geprüfte Konvertierung	`static_cast<typ>(ausdruck)`	
		ungeprüfte Konvertierung	`reinterpret_cast<typ>(ausdruck)`	
		const-Konvertierung	`const_cast<typ>(ausdruck)`	
3	L	Objektgröße	`sizeof objekt`	N
	L	Typgröße	`sizeof (typ)`	N
	R	Präinkrement	`++lvalue`	
	R	Prädekrement	`--lvalue`	
	R	Komplement (bitwise not)	`~ausdruck`	
	R	Nicht (logisches not)	`!ausdruck`	
	R	unäres Minus	`-ausdruck`	
	R	unäres Plus (der Vollständigkeit halber)	`+ausdruck`	
	R	Adresse	`&lvalue`	
	R	Dereferenzierung	`*ausdruck`	
	L	Erzeugung (Belegung)	**new** `typ`	
	L	Erzeugung(Belegung und Initialisierung)	**new** `typ(ausdrucksliste)`	
	L	Erzeugung(Platzierung)	**new**`(ausdrucksliste)typ`	
	L	Erzeugung(Platzierung und Initialisierung)	**new**`(ausdrucksliste)typ(ausdrucksliste)`	
	R	Zerstörung(Freigabe)	`delete zeiger`	
	R	Feldzerstörung	`delete[] zeiger`	
	R	Cast(Typkonvertierung (C-Style))	`(typ)ausdruck`	
4	L	Elementselektion (Auswahloperator)	`objekt.*zeiger-auf-element`	N
	L	Elementselektion (Auswahloperator)	`zeiger->*zeiger-auf-element`	M
5	L	Multiplikation	`ausdruck * ausdruck`	
	L	Division	`ausdruck / ausdruck`	
	L	Modulo (Restwertoperator) nur für ganzzahlige Operanden	`ausdruck % ausdruck`	
6	L	Addition	`ausdruck + ausdruck`	
	L	Subtraktion	`ausdruck - ausdruck`	
7	L	Lschieben (bitoperation)	`ausdruck << ausdruck`	
	L	Rschieben (bitoperation)	`ausdruck >> ausdruck`	
8	L	Kleiner als	`ausdruck < ausdruck`	
		Kleiner gleich	`ausdruck <= ausdruck`	
		Größer als	`ausdruck > ausdruck`	
		Größer gleich	`ausdruck >= ausdruck`	
9	L	Gleich	`ausdruck == ausdruck`	
	L	Ungleich	`ausdruck != ausdruck`	
10	L	Bitweises Und	`ausdruck & ausdruck`	
11	L	Bitweises Exklusiv-Oder	`ausdruck ^ ausdruck`	
12	L	Bitweises Oder	`ausdruck | ausdruck`	
13	L	logisches Und	`ausdruck && ausdruck`	

14	L	logisches Oder	`ausdruck || ausdruck`	
15		Bedingte Zuweisung	`ausdruck ? ausdruck : ausdruck`	

16	R	Einfache Zuweisung
		Multiplikation und Zuweisung
		Division und Zuweisung
		Modulo und Zuweisung
		Addition und Zuweisung
		Subtraktion und Zuweisung
		Lschieben und Zuweisung
		Rschieben und Zuweisung
		bitweises Und und Zuweisung
		bitweises Oder und Zuweisung
		bitweises exclusiv Oder und Zuweisung
			`lvalue = ausdruck`
			`lvalue *= ausdruck`
			`lvalue /= ausdruck`
			`lvalue %= ausdruck`
			`lvalue += ausdruck`
			`lvalue -= ausdruck`
			`lvalue <<= ausdruck`
			`lvalue >>= ausdruck`
			`lvalue &= ausdruck`
			`lvalue |= ausdruck`
			`lvalue ^= ausdruck`	

17	R	Ausnahmen werfen	**`throw`** `ausdruck`	
18	L	Komma(Sequenzoperator)	`ausdruck, ausdruck`	

Operatoren der Form: `lvalue @= ausdruck` werden als compound assignment Operatoren bezeichnet. Die Semantik ist `lvalue = lvalue @ ausdruck`.

6.25.3 Einschränkungen beim Überladen von Operatoren

Nur die vordefinierten Operatoren können beim Überladen für Benutzer definierte Typen verwendet werden, es ist nicht möglich, neue Operatoren zu definieren (z.B.: ** für Potenzieren) oder die Operatoren für die vordefinierten Datentypen selbst zu definieren[121].
Ein Operator kann nur entsprechend der für ihn, in der Grammatik definierten Syntax, deklariert werden[122], d.h. ein binärer Operator kann nicht unär oder ternär definiert werden.
Ein binärer Operator kann durch eine nicht statische Elementfunktion / Methode mit genau einem Parameter oder durch eine Nichtelementfunktion (globale Funktion) mit 2 Parametern definiert werden. Für jeden binären Operator @ kann der Ausdruck `lhs@rhs` als `lhs.operator@(rhs)` oder als `operator@(lhs, rhs)` interpretiert werden. Im ersten Fall entspricht **`this`** dem Argument für `lhs`.
Überladene Operatoren sind Funktionen mit dem Namen `operator@(..)`! Sind beide Operator Funktionen definiert, kommen die Auflösungsregeln beim Überladen zum tragen.

Folgende Operatoren **können nicht** überladen werden:

- `::` socpe resolution / Bereichsauflösung

[121] http://en.cppreference.com/w/cpp/language/operators
[122] [Str00a] 11.2 Operatorfunktionen

- `.` element selection / Elementauswahl via Objekt/Referenz
- `.*` element selection via Objekt/Referenz und element pointer / Elementauswahl durch einen Elementzeiger
- `?:` conditional operator
- `sizeof, dynamic_cast, static_cast, reinterpret_cast, const_cast, typeid , decltype`

Folgende Operatoren **müssen als Elementfunktionen** implementiert werden, damit garantiert ist, dass ihr erster Operand ein L-Value Ausdruck (**`this`**) ist:

- `=, +=, ...` Zuweisungsoperator, Compoundassignment
- `[]` Indexoperator
- `()` Funktionsaufrufoperator
- `->` Elementselektion über einen Pointer
- `->*` Element selection via Objektzeiger und element pointer / Elementauswahl durch einen Elementzeiger
- `type` Konvertierung in den `type` (siehe section 6.37 auf Seite 157)

Folgende Operatoren **sollten nie** überladen werden:[123].

- `&&` logisches Und
- `||` logisches Oder
- `,` der Sequence- oder Kommaoperator, uuh, ah!
- `&` Adressoperator, ermittelt die Adresse eines L-Value Ausdrucks[124]

Ausdrücke mit den logischen Operatoren (Und, Oder) unterliegen der so genannten **`short-circuit evaluation`**. Das bedeutet, wird das Ergebnis des gesamten Ausdrucks durch den ersten Operanden bestimmt, wird der zweite Operand nicht mehr ausgewertet! Bei Und ist das der Fall, wenn der erste Operand **`false`** ist, bei Oder, wenn er **`true`** ist.
Dadurch ist es möglich Bedingungsausdrücke wie folgt zu formulieren:
`if``(p && p->isValid())...` zuerst den Zeiger auf `p != 0` prüfen, dann die Nachricht `isValid()` an das Objekt senden. Ist `p == nullptr` wird der Ausdruck `p->isValid()` nicht mehr ausgewertet. Für `p == nullptr` würde die Auswertung von `p->isValid()` zu `undefined behavior`, bzw. zu einer Null-Pointer Exception führen! Das ist das Verhalten, das Programmierer erwarten.

Im Falle der Überladung des Operators durch eine **Funktion**, gibt es keine short-circuit evaluation, alle Argumente des Funktionsaufrufs werden ausgewertet, bevor die Funktion gerufen wird. Was im Code so aussieht:
`if``( expression1 && expression2)` ... sieht für den Compiler so aus:
`if``( expression1.operator&&(expression2))` ... für Member Funktionen oder so
`if``( operator&&( expression1, expression2)` ... für globale Funktionen.

[123] [Mey99] Item 7 Never overload `&&`, `||` or `,`
[124] http://manderc.com/operators/addressoperator/

Es werden zuerst beide Argumente ausgewertet, dann wird die Operatorfunktion `operator&&(...)` aufgerufen!

Die **short-circuit** Semantik der logischen Operatoren stellen eine Ausnahme gegenüber den anderen Operatoren dar! Sowohl bei einem Funktionsaufruf als auch bei der Auswertung eines Ausdrucks an dem andere Operatoren beteiligt sind, **ist die Reihenfolge der Auswertung der Operanden** durch die Sprache **nicht definiert!**[125];

Listing 76: Der **Komma** oder **Abfolge Operator**

```
1 void reverse(char s[]){
2    for(int i = 0, j = strlen(s)-1; // initialisierung
3    i < j;   // Laufbedingung
4    ++i, --j) { // Aah, der Komma Operator
5       int c = s[i];
6       s[i] = s[j];
7       s[j] = c;
8    }
9 }
```

Im letzten Teil der **for** Schleife ist genau eine Ausdrucksanweisung erlaubt, sollen hier mehrere Anweisungen stehen, können diese durch den Komma Operator getrennt aufgelistet werden.
Der Ausdruck in der Anweisung wird von links nach rechts ausgewertet, das Ergebnis ist der Wert des rechts stehenden Ausdrucks.

```
int i = a, b; //i wird der Wert von b zugewiesen
```

Auch dieses Verhalten kann nicht durch eine Operator Funktion realisiert werden. Deshalb sollten diese Operatoren nicht überladen werden.

6.25.4 Sequence point

Der Begriff Sequence Point oder „Sequencing“ ab C++11 hat große Bedeutung im Zusammenhang der Sprachspezifikation und des Verständnisses von Programmiersprachen im Allgemeinen und im Besonderen für die Sprachen C/C++.

Ein Sequence Point definiert einen Punkt im Sourcecode (Programm), an dem garantiert ist, dass **alle Seiteneffekte** der vorangegangenen Auswertung von Ausdrücken **vollständig abgeschlossen sind** und noch keine von nachfolgenden Auswertungen Auswirkungen hervorgerufen haben[126].

Daraus ergeben sich 3 Möglichkeiten:

- die Reihenfolge ist festgelegt (Reihenfolge wie im Sourcecode: a sequenced before b)

[125] section 6.25.4
[126] https://en.wikipedia.org/wiki/Sequence_point

- die Reihenfolge ist nicht festgelegt, aber es gibt Eine! Der Compiler oder die Laufzeitumgebung(!) kann frei wählen (indeterminately sequenced), es gibt keine Überlappungen
- die Reihenfolge ist „unsequenced“ spezifiziert: Ein Teil des einen und ein Teil des anderen Ausdrucks kann abwechselnd ausgewertet werden (overlap)

Die Auswertung von Ausdrücken, die nicht in einer bestimmten Reihenfolge definiert ist (unsequenced), können überlappen, was zu „undefined behavior“ führt, wenn die Reihenfolge der Auswertung Auswirkung auf das Ergebnis hat.

Die Reihenfolge der Auswertung[127]

- der Operanden fast aller Operatoren
- der Funktionsargumente in einem Funktionsaufruf
- von Unterausdrücken (Subexpressions)

ist nicht spezifiziert.
Die Reihenfolge kann bei jeder Ausführung auf Grund von Optimierungen variieren! Das Listing 77 soll die Auswirkung dieser Regel verdeutlichen.

Listing 77: Auswertungsreihenfolge der Operanden und Argumente

```
1  int b() {
2     x = 1;
3     return x;
4  }
5  int c() {
6     x++;
7     return x;
8  }
9  int x = 0; // globales Objekt
10 void f(int, int);
```

In dem Ausdruck
`a = b()+ c();`
ist nicht definiert, ob zuerst die Funktion `b()` und dann die Funktion `c()` aufgerufen wird oder umgekehrt[128]!

`a == 3` für die Reihenfolge `b(), c()` oder `a == 2` für die Reihenfolge `c(), b()`.

Ebenso beim Aufruf der Funktion:
`f( b(), c());`
ist nicht definiert in welcher Reihenfolge die Argumente `b()` und `c()` ausgewertet werden.

[127] http://en.cppreference.com/w/cpp/language/eval_order
[128] nicht zu verwechseln mit der Reihenfolge der Auswertung der Operatoren selbst siehe section 6.25.5 auf der nächsten Seite

6.25.5 Arithmetische Ausdrücke und implizite Typkonvertierung

Am Beispiel der arithmetischen Operatoren und der eingebauten arithmetischen Datentypen sollen in den folgenden Kapiteln die Regeln dargelegt werden, die bei der Auswertung von komplexen Ausdrücken zur Anwendung kommen und was beachtet werden muss, wenn sie überladen werden sollen.

Einfache arithmetische Ausdrücke

Listing 78: Einfache Ausdrücke

```
5 // rvalue ausdruck Ganzzahlkonstante typ: int, wert: 5
'a' // rvalue ausdruck Zeichenkonstante typ: char, wert:'a' oder 97
3.5e2 // rvalue ausdruck Gleitpunktkonstante typ:double, wert: 350
"textkonstante" // rvalue ausdruck typ: char const *, wert: "textkonstante"
```

Einfache Konstanten sind Ausdrücke. Sie auszuwerten bedeutet sie zur Verarbeitung in ein Register zu laden.

Variablen und Funktionen als Ausdrücke

Listing 79: Variablen und Funktionen als Ausdrücke

```
int a = 3; //Deklaration ist eine Anweisung (C++)
int f(); // Funktionsdeklaration
a // lvalue ausdruck typ: int wert: 3
f() // rvalue ausdruck, typ: int, wert: ?
```

Einfache Ausdrücke mit Operatoren

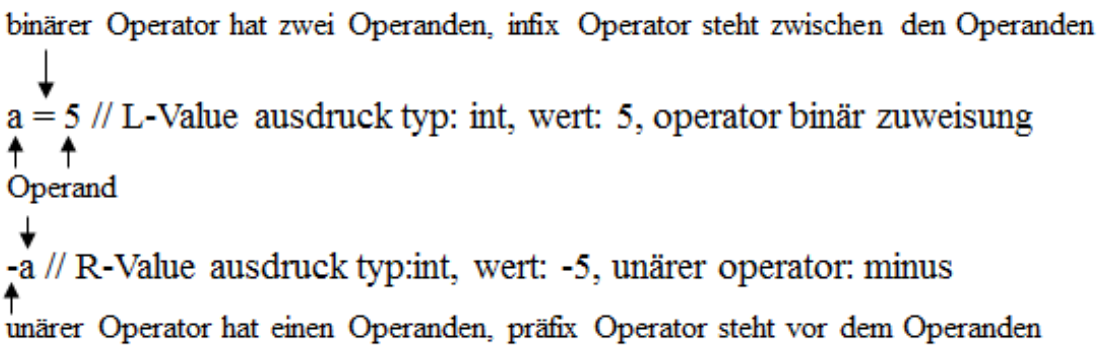

Abbildung 2: Einfache Ausdrücke mit Operatoren

Listing 80: Einfache Ausdrücke mit Operatoren

```
int a = 5;
a < 5    // rvalue ausdruck typ: (C:int, wert 0) bool,
      // wert: false, binärer operator: kleiner
a + 1 // rvalue ausdruck typ: int, wert: 6, binärer operator: plus
```

Reihenfolge der Berechnung von Ausdrücken

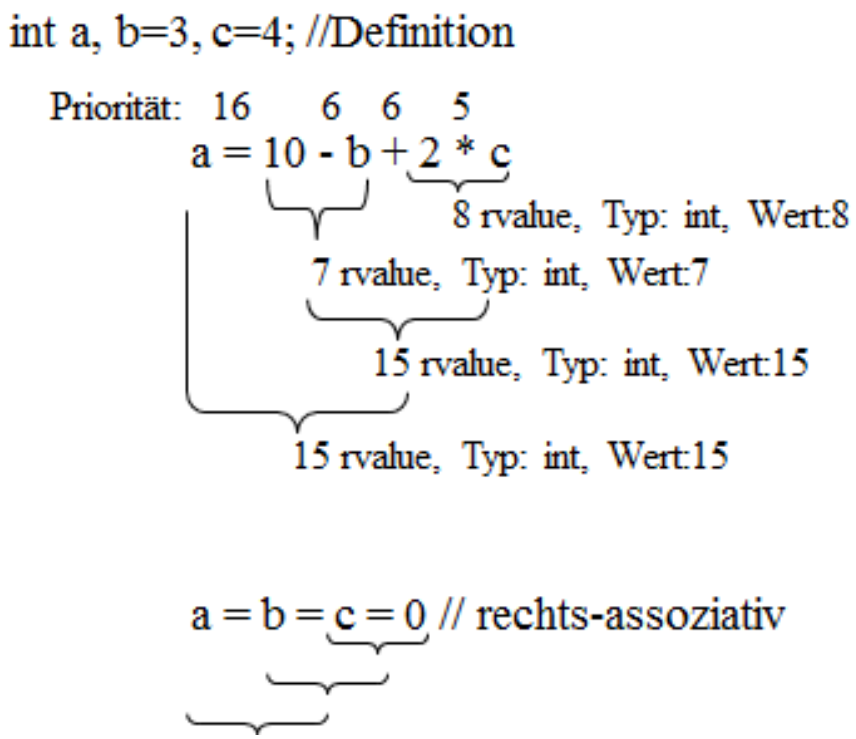

Abbildung 3: Auswertung komplexer / zusammengesetzter Ausdrücke

Regel: Die Reihenfolge der Auswertung wird durch die Priorität der beteiligten Operatoren bestimmt. Ist die Priorität gleich, bestimmt die Assoziativität die Auswertungsreihenfolge[129].

Aufgrund der Möglichkeit, Zuweisungen zu verketten, sollte der Zuweisungsoperator `operator=(...)` immer eine Referenz auf das aktuelle Objekt (**`return *this`**) zurückgeben um dieser Konvention zu folgen.

Achtung: Die Reihenfolge in der die Operanden ausgewertet werden ist nicht definiert[130].

Implizite Typkonvertierung in komplexen Ausdrücken und bei der Zuweisung

Regel: Konvertiert wird in den komplexesten Typ der an dem Ausdruck beteiligt ist.

Regel bei der Zuweisung: Konvertiert wird in den Typ des lvalue-Ausdrucks.

Typen die ohne Verluste konvertierbar sind, werden durch den Compiler bei Bedarf automatisch konvertiert, das wird unter impliziter Typkonvertierung verstanden.

Regel: integral promotion (Ganzzahlerweiterung)
Bei Datentypen die kleiner sind als **`int`** wird als erstes die so genannte **Integral Promotion** / Ganzzahl Erweiterung vorgenommen, bevor der Ausdruck berechnet wird.

[129] Priorität und Assoziativität siehe section 6.25.2 auf Seite 83
[130] section 6.25.4 auf Seite 87

```
int a; //Deklaration  ist eine Anweisung  (C++)
double b=3.5;
float c=4;
```

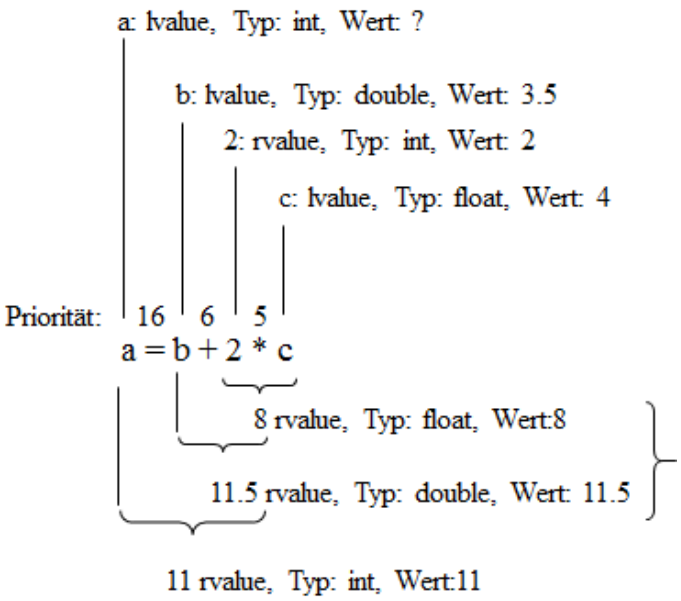

Abbildung 4: Regel zur Typkonvertierung

`bool`, `signed` **`char`**, `unsigned` **`char`**, **`short`** wird zu **`int`** konvertiert.

`unsigned` **`short`** zu **`int`** wenn **`int`** größer als **`short`** ist, sonst zu `unsigned` **`int`**, treten danach noch Operanden mit unterschiedlichen arithmetischen Typen auf, werden weitere implizite Typanpassungen durchgeführt.

In komplexen arithmetischen Ausdrücken, an denen verschiedene Datentypen beteiligt sind, wird der einfachere Operand in den komplexeren Typ, der an der Operation beteiligt ist konvertiert.

Achtung: Wenn nur die „signedness“ verschieden ist, also der gleiche Typ einmal `signed` und einmal `unsigned` in einem Ausdruck beteiligt ist, wird in den `unsigned` Typ konvertiert! Das Listing 81 demonstriert das Ergebnis: aus einem negativen Wert wird ein positiver!

Listing 81: Typkonvertierung signed unsigned

```
1 unsigned int ui = 0;
2 int i = -1; // Bitmuster 0xFFFFFFFF für einen 4 Byte int
3 std::numeric_limits<unsigned int>::max() == ui + i
```

6.25.6 Arithmetische Operatoren `+, -, *, /, %`

Die arithmetischen Operatoren (`+, -, *, /, %`) und alle anderen infix Operatoren können nicht als Member implementiert werden, wenn gemischte Operationen mit eingebauten Datentypen oder anderen Klassen möglich sein sollen.

`a = b + 42; //ok, entspricht b.operator+(42)`

`a = 42 + b; //Error, es existiert kein operator 42.operator+(b)` der ein Widget als Argument erwartet

Die Operatoren dürfen ihre Operanden nicht verändern! Daher werden diese als `const&` übergeben.

Die Operatoren sollten auf der Basis der zusammengesetzten Zuweisungsoperatoren +=, *=, usw. (compound asignment operator) implementiert werden, was bedeutet, dass diese ebenfalls existieren müssen. Damit ist gewährleistet, dass sich diese Operatoren konsistent zueinander verhalten.

Die impliziten Typkonvertierungsregeln in arithmetischen Ausdrücken sind in section 6.25.5 auf Seite 89 beschrieben.

Listing 82: Arithmetische Operatoren

```
1  class Widget {
2     int i;
3  public:
4     Widget(int i=0):i(i){}
5     Widget& operator+=(Widget const& rhs) {
6        i += rhs.i;
7        return *this;
8     }
9  };
10 // für a = b + c;
11 const Widget operator+(Widget const& lhs, Widget const& rhs) {
12    Widget ret(lhs)
13    ret += rhs;
14    return ret;
15 };
16 // für a = 42 + b;
17 const Widget operator+(int lhs, Widget const& rhs) {
18    Widget ret(lhs)
19    ret += rhs;
20    return ret;
21 };
22
23 // für a = b + 42;
24 const Widget operator+(Widget const& lhs, int rhs) {
25    Widget ret(rhs)
26    ret += lhs;
27    return ret;
28 };
```

Wenn die Klasse `Widget` entsprechende Konvertierungskonstruktoren[131] für die Typen hat, mit denen sinnvolle Verknüpfungen möglich sind, in diesem Fall `Widget (int i)`, dann wird nur der erste `operator+(Widget const&, Widget const&)` benötigt, ansonsten müssen für alle Kombinationen Operatoren wie in Listing 82 zur Verfügung gestellt werden.

[131] section 6.37.1 auf Seite 157

6.25.7 Die relationalen Operatoren >, >=, <, <=, ==, !=

Die Tabelle 6 zeigt die relationalen Operatoren und das Ergebnis der Auswertung für die Werte -1, 0 und 1 für x.

Tabelle 6: Die relationalen Operatoren

Priorität:	8	8	8	8	9	9
C/C++ Operatoren:	x>0	x>=0	x<0	x<=0	x==0	x!=0
x	x>0	$x \geq 0$	x<0	$x \leq 0$	x=0	$x \neq 0$
-1	f	f	w	w	f	w
0	f	w	f	w	w	f
+1	w	w	f	f	f	w

6.25.8 Die logischen Operatoren &&, ||, !

Die Tabelle 7 zeigt die Anwendung der logischen Operatoren und das Ergebnis der Auswertung der Ausdrücke.

Die Operatoren logisches *Und* (&&) und logisches *Oder* (||) werden mit der *short-circuit-evaluation* ausgewertet, d.h. der rechte Operand wird nicht mehr ausgewertet, wenn der linke Operand das Ergebnis des gesamten Ausdrucks bestimmt. Bei logischem *Und* ist das bei false und bei logischem *Oder* bei true der Fall! Darum sollten sie nicht überladen werden, siehe section 6.25.3 auf Seite 85.

Der linke Operand wird vor dem rechten vollständig ausgeführt (sequenced before) und alle Seiteneffekte kommen zum tragen, bevor der rechte Operand ausgewertet wird.

Tabelle 7: Die logischen Operatoren

Priorität:				13		14	3
C/C++ Operatoren:				`a && b`	`a || b && !(a && b)`	`a || b`	`!a`
Beispiele a,b		x > 0	y > 0				
x	y	a	b	$a \wedge b$	a xor b	$a \vee b$	$\neg a$
-1	-1	f	f	f	f	f	w
-1	+1	f	w	f	w	w	w
+1	-1	w	f	f	w	w	f
+1	+1	w	w	w	f	w	f

6.25.9 Die bitwise Operatoren &, |, ^, ~

Die bitwise Operatoren können nur auf ganzzahlige / integrale Datentypen angewendet werden. Sie verknüpfen alle korrespondierenden Bits der Operanden miteinander. Die Tabelle 8 auf der nächsten Seite zeigt die Anwendung der bitwise Operatoren auf zwei Variablen a und b und das Ergebnis der Auswertung für jeweils ein einzelnes Bit.

Tabelle 8: Die bitwise Operatoren

Priorität:		10	11	12	3
C/C++ Operatoren:		`a & b`	`a ^ b`	`a | b`	`~a`
a	b	a $\wedge$ b	a xor b	a $\vee$ b	$\neg a$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	0	1	0

Die bitwise Operatoren werden häufig in Zusammenhang mit Bitmasken verwendet: `constexpr unsigned char Bit3 = 0x08`, das Bit 3 ist 1 alle anderen sind 0[132]. Die Bits der Bitmaske Bit3 sind nach dieser Initialisierung wie folgt belegt: `0000 1000`.

Die bitwise Operatoren können wie folgt verwendet werden:

- Filtern eines bestimmten Bits: `a & Bit3 != 0` ist wahr, wenn Bit 3 gesetzt ist, die Variable bleibt unverändert
- Einschalten eines bestimmten Bits: `a |= Bit3` Bit 3 ist dannach 1, alle anderen Bits bleiben unverändert
- Ausschalten eines bestimmten Bits: `a &= ~Bit3` Bit 3 ist dannach 0, alle anderen Bits bleiben unverändert
- wechseln / toggeln eines bestimmten Bits: `a^=Bit3` Bit 3 ist 1 wenn es 0 war und 0 wenn es 1 war, alle anderen Bits bleiben unverändert.
- tauschen ohne temp: `a^=b; b^=a; a^=b;` die Inhalte von a und b sind nach diesen 3 Anweisungen vertauscht

6.25.10 Bitshift Operator <<, >>

Die Bitshift Operatoren shift left und shift right schieben das Bitmuster in einem ganzzahligen Datentyp um eine Anzahl Bits.
`c << n` verschiebt das Muster von c um n Stellen nach links bzw. `c >> n` nach rechts.
In Listing 83 auf der nächsten Seite wird die Variable `c` mit einem Wert in hexadezimaler Schreibweise initialisiert. Das Bitmuster ist im Kommentar dargestellt und wird in den darauf folgenden Anweisungen eine Stelle nach links, anschließend zwei Stellen nach rechts und wieder in die Ausgangsposition verschoben. Die Bits an den Rändern werden mit Nullen aufgefüllt, die Einsen gehen verloren.

[132] Bits werden von rechts, dem least signifcant bit (lsb) mit 0 beginnend nummeriert

Listing 83: Die Bitshift Operatoren «

```
unsigned char c = 0xDB; // 1101 1011
c <<= 1; // c == 1011 0110
c >>= 2; // c == 0010 1101
c <<= 1; // c == 0101 1010
```

`c <<= 1` entspricht `c = c << 1` und verschiebt das Muster ein Bit nach links, entsprechend verschiebt `c >>= 2` und das Muster um 2 nach rechts.

Der bitwise rechts- bzw. linksshift Operator ist für Streams und die eingebauten Datentypen als Ausgabe- und Eingabeoperator überladen. Soll dieser Operator zur Ausgabe oder zum Einlesen für eigene Klassen überladen werden, muss er immer eine Referenz auf den Stream zurückgeben und als globaler Operator und nicht als Member überladen werden, damit die übliche Verkettung der Aufrufe möglich ist.

Listing 84: Die Verkettung des Shift Operators

```
Widget a, b;
cout << "a:" << a << "b:" << b << endl;
```

Reicht die öffentliche Schnittstelle der Klasse für die Ausgabe nicht aus, kann der Operator als friend in der jeweiligen Klasse deklariert werden, damit hat er Zugriff auf die privaten Member der Klasse und kann diese ausgeben. Typischer weise wird der Operator aber mit einer geeigneten print-Operation der Klasse implementiert, die gegebenenfalls virtual ist.

Listing 85: Shift Operator überladen

```
class Widget {
int i;
public:
   ...
   virtual ostream& print(ostream& os) const {
      os << i;
      return os;
   }
   // Alternativ:
   friend ostream& operator<<(ostream&, Widget const&);
};

ostream& operator<<(ostream& lhs, Widget const& rhs){
   return rhs.print(lhs);
}
```

Eine Funktion, die innerhalb einer Klassendefinition `friend` definiert wird, ist obwohl sie innerhalb der Klassendefinitioin definiert wird, kein Member der Klasse, sondern eine globale Funktion.

Listing 86: Der globale friend Operator Shift

```
class Widget{
   int data;
public:
   friend std::ostream& operator<<(std::ostream& os, Widget const& w){
      os << w.data;
      return os;
   }
};
```

6.25.11 Copy- und Move- Zuweisung / assignment `operator=()` [133]

Aufgrund der Möglichkeit, Zuweisungen zu verketten, sollte der Zuweisungsoperator

```
T& operator=(T const & rhs)& { ... return *this; }
```

immer eine Referenz auf das aktuelle Objekt (`return *this`) zurückgeben um dieser Konvention zu folgen.

Die Zuweisung an temporäre Objekte (rvalues) sollte verhindert werden. Daher sollte der Copy- und der Move- Assignment Operator für lvalues qualifiziert werden[134]. Dadurch kann er nicht auf rvalue Ausdrücke angewandt werden.

Bei der Zuweisung sollte fast immer ein Schutz vor der Selbstzuweisung eingebaut werden!

```
If(this == &rhs)return *this;
```

Vor dem Hintergrund von Ausnahmen, die beim Kopieren auftreten können, sollte der Zuweisungsoperator atomar sein, d.h. entweder wird die Zuweisung vollständig durchgeführt oder gar nicht. Eine mögliche Implementierung für eine Klasse `Widget` wäre:

Listing 87: Assignment Operator und swap

```
Widget& operator=(Widget rhs){ // call by value
   swap(rhs); // tauscht Inhalt von this und dem temporären Objekt rhs
   return *this;
}
```

Bei dem Beispiel wird das Argument by value übergeben, das Kopieren des Objekts wird daher vom copy-constructor durchgeführt. Tritt dabei eine Exception auf, wird die Zuweisung nicht ausgeführt. Die Funktion swap() muss dann nur noch die Inhalte des temporären Objekts `rhs` mit den Inhalten von this austauschen. Der Destruktor von rhs räumt anschließend die Resourcen von `this` auf. Als Nebeneffekt wird dadurch ein Schutz vor Selbstzuweisung erzielt, weil bei der Zuweisung eine Kopie erstellt wird. Der Aufwand dafür kann in den meisten Fällen

[133] [Mey06a] Item 10 - 12
[134] section 6.32.10 auf Seite 135

vernachlässigt werden, weil Selbstzuweisungen nur selten oder überhaupt nicht vorkommen.

Nur zwei Operationen sollten Objekte kopieren. Der copy constructor und der copy assignment operator. Sie müssen gewährleisten, dass alle Attribute kopiert werden. In Vererbungshierarchien muss also die jeweilige Kopieoperation der Basisklasse aufgerufen werden. Der `operator=()` kann dazu entweder `Base::operator=(rhs)` aufrufen oder Base definiert eine **`protected`** `swap()` Operation, die in `Widget::swap()` aufgerufen werden kann.

6.25.12 Element Auswahl `operator->()` und Dereferenzierungs- `operator*()`

Die typische Anwendung für diese Operatoren sind so genannte SmartPointer[135] und Iteratoren[136]. Das sind Objekte, die eine Schnittstelle wie Pointer haben, aber darüber hinaus weitere Dienste anbieten, z.B.: Resourcenverwaltung. In Listing 88 ist die Implementierung dieser Operatoren skizziert.

Der Derefereinzierungs- `operator*()` ist das Gegenstück zum Adress- `operator &()`[137]

Listing 88: Resourcenverwaltung und RAII

```
1  class Widget{...};
2  class SmartPointer{
3     Widget* pWidget;
4  public:
5     explicit SmartPointer(Widget* pWidget):pWidget(pWidget) {}
6     ~SmartPointer() { delete pWidget; }
7     ...
8     Widget& operator*() { return *pWidget; } // Dereferenzierung
9     Widget* operator->() { return pWidget; } // Elementauswahl
10 };
11
12 int main(){
13    SmartPointer p(new Widget);
14    p->operation(); // äquivalent zu: (p.operator->())->operation()
15    (*p).operation();
16    // delete erledigt p
17 }
```

Repository:Cpp-Basics/SmartPointer

Will man solches Verhalten für beliebige Typen implementieren, ist diese Klasse als Template zu realisieren, wie es die STL in verschiedener Ausprägung anbietet. (auto_ptr, shared_ptr, weak_ptr, iteratoren der STL)

[135] section 6.17 auf Seite 54
[136] [GHJV95] Iterator Pattern und section **??** auf Seite ??
[137] section 6.25.3 auf Seite 86

Der Elementauswahl- `operator->()` hat gegenüber den anderen Operatoren, die überladen werden können, eine Besonderheit[138]: Die Operatorfunktion wird vom Compiler so lange angewendet, bis der Ergebnistyp ein nativer Pointer ist, dann wird das Element aus dem Typ, auf den der Pointer zeigt, ausgewählt!

Mit dieser Technik kann ein SmartPointer ein Objekt anstatt eines nativen Pointers zurückliefern, das den `operator->()` ebenfalls überlädt. Die Lebensdauer dieses Objekts beginnt mit dessen Erzeugung wenn es zurückgeliefert wird:
`return Wrapper(pointee);` und endet nach dem Aufruf `p->operation();`. Der Aufruf könnte alternativ auch mit `p.operator->().operator->()->operation()` codiert werden, wobei der Ausdruck `p.operator->()` vom Typ `Wrapper` ist.

Der Wrapper kann dafür verwendet werden, vor und nach dem Aufruf einer Operation etwas zu tun, z.B. in einer multithreaded Umgebung einen Lock anfordern und diesen im Destruktor wieder frei geben. Mehrere Wrapper können verkettet werden um verschiedene Aufgaben vor oder nach dem Aufruf der eigentlichen Operation des Objekts zu erledigen. Das entspricht ungefähr dem Decorator Pattern der Gof[139]. Leider kann auf die Übergabeargumente und den Rückgabewert nicht zugegriffen werden, was die Anwendung des Mechanismus z.B. für Logging, einschränkt.

Das Listing 89 skizziert diese Technik.

Listing 89: Ein Wrapper um den Aufruf einer Operation

```
1  struct Widget{ void operation(){ cout << "Widget.operation()" << endl; } };
2  struct Wrapper{
3     Wrapper(Widget* pointee):pointee(pointee){}
4     ~Wrapper(){
5        std::cout << "Wrapper::~Wrapper()" << std::endl;
6     }
7     Widget* operator->(){
8        std::cout << "Wrapper.operator->()" << std::endl;
9        return pointee;
10    }
11    Widget* pointee;
12 };
13 struct SmartPointer{
14    SmartPointer(Widget* pointee) : pointee(pointee){}
15    Wrapper operator->(){ return Wrapper(pointee); }
16    Widget* pointee;
17 };
18 int main(){
19    cout << "PointerElementAuswahlOperator" << endl;
20    Widget b;
21    SmartPointer p(&b);
22    cout << "=== main() p->operation()" << endl;
23    p->operation();
24 }
25 //Ausgabe:
```

[138] [Str00b]
[139] [GHJV95]

```
26 PointerElementAuswahlOperator
27 === main() p->operation()
28 Wrapper.operator->()
29 Widget.operation()
30 Wrapper::~Wrapper()
```

6.25.13 Index `operator[]`

Der Index Operator ist ein binärer Operator. Auf Pointer und Namen von Arrays angewandt, führt er folgende Berechnung durch: Adresse + index * sizeof(Type). Er liefert eine Referenz auf das ausgewählte Objekt. An dieses Verhalten sollte man sich bei der Implementierung eines überladenen Index Operators halten.

Listing 90: Index Operator[]

```
1  class SmartIntPointer {
2     int* buffer;
3     std::size_t count;
4     SmartIntPointer(int buffer[], std::size_t count):buffer(buffer), count(count){}
5     int& operator[](std::size_t idx) {
6        if(idx < 0 || idx >= count)
7           throw std::out_of_range();
8        return buffer[idx];
9     }
10    // für const Objekte
11    const int& operator[](std::size_t idx) const {
12       if(idx < 0 || idx >=count)
13          throw std::out_of_range();
14       return buffer[idx];
15    }
16 };
```

6.25.14 Funktionsaufruf `operator()()`

Ein Funktionsaufruf `ausdruck(ausdrucks-liste)` kann als binärer Operator mit `ausdruck` als linkem Operand und `ausdrucks-liste` als rechtem Operand interpretiert werden[140]. Objekte von Klassen die diesen Operator überladen werden auch als `SmartFunction`, `FunctionObject` oder kurz `Functor` bezeichnet. Die STL macht von dieser Möglichkeit umfangreich gebrauch um damit Algorithmen zu parametrieren und um die Ordnung der Elemente in sortierten Containern festzulegen. Diese Functors werden als Prädikate bezeichnet.

Mit der Einführung von Lambdas[141] in C++11 hat ihre Bedeutung etwas abgenommen.

[140] [Str00a] 11.9 Funktionsaufruf und section 6.32.5 auf Seite 128
[141] section 6.32.11 auf Seite 136

Listing 91: Funktionsaufrufoperator

```
class Add{
   complex wert;
   Add(complex c):wert(c) {}
   void operator()(complex& c) const { c += wert; }
};

void demo(vector<complex>& v, list<complex>& l){
   for_each( v.begin(), v.end(), Add(complex(2,3)) );
   for_each( l.begin(), l.end(), Add(complex(5,1)) );
}
```

Dem Algorithmus wird ein temporäres Objekt von `Add` übergeben. Dieses ist mit dem Wert, der zu allen in den Containern enthaltenen Objekten dazu addiert werden soll, initialisiert. Ein solches Verhalten ist mit puren Funktionen nicht elegant erreichbar.

Ein Prädikat kann ein Functor mit dem Prototyp `bool operator ()(const T& o1, const T& o2);` sein. Er muss die Äquivalenz Regeln einhalten, also das Verhalten des Operators „<“:
`!(o1 < o2)&& !(o2 < o1)` für Gleichheit.

6.25.15 Inkrement / Dekrement `++operator()--`

Die Inkrement und Dekrement Operatoren sind eine Ausnahme, weil sie sowohl als Prä- als auch als Postfix Operatoren vorkommen.

Die Postfix Version des Operators liefert den letzten Wert des Objekts und verändert das Objekt im Sinne des Operators. Dazu wird ein temporäres Objekt benötigt das per Value zurückgegeben wird. Die Präfix Version verändert das Objekt und liefert dieses im neuen Zustand als Referenz zurück. Die Semantik der Operatoren `++/--` ist wie folgt:

```
int i, a = 0;
i = ++a; //Präfix entspricht
a = a + 1; i = a;
i = a++; //Postfix entspricht
i = a; a = a + 1;
```

Hier wird klar, dass die Präfix Version überall wo es möglich ist, der Postfix Version vorzuziehen ist, weil keine temporäre Speicherung des alten Wertes von a notwendig ist und sie daher geringere Laufzeitkosten hat. Zur Unterscheidung von Prä- und Postfix wird der Dummy Parameter `int` für den Postfix verwendet.

Listing 92: Operatoren `++/--`

```
class Widget {
public:
   explicit Widget(int i):i(i) {}

   Widget() = default;
```

```
6     Widget(Widget const&) = default;
7
8     //assignment only for l-values
9     Widget& operator=(Widget const&) & = default;
10    Widget& operator=(Widget &&) & = default;
11
12
13    Widget& operator++() { // Präfix
14       ++i;
15       return *this;
16    }
17
18    Widget operator++(int) { // Postfix
19       Widget temp(*this);
20       ++i;
21       return temp;
22    }
23
24    Widget& operator--() { /*dto.*/ } // Präfix
25
26    Widget operator--(int){ /*dto.*/ } // Postfix
27 private:
28    int i;
29 };
30 void f(Widget&& rW);
31 void f(Widget const& w);
32 int main(){
33    cout << "IncrementDecrementOperatoren" << endl;
34
35    Widget w, w2;
36    f(w); // calls f(Widget const&)
37    f(w++); // calls f(Widget&&)
38    w++ = w2; // error: passing 'Widget' as 'this' argument discards qualifiers
39 }
```

Die Präfix Operatoren sollten die Referenz auf das Objekt selbst zurück liefern, damit der Ergebnistyp des Operators ein L-Value Ausdruck ist[142]. Das entspricht dem Verhalten dieser Operatoren für eingebaute Datentypen.

Die Postfix Operatoren sollten in C++03 ein konstantes Objekt (`const Widget operator++(int)`) zurückliefern, damit der Ergebnistyp des Operators ein R-Value Ausdruck ist.
In C++11 ist wird das nicht mehr empfohlen, da mit dieser Signatur eine Funktion `void f(Widget&&)` nicht aufgerufen werden kann: `f(w++)`. Um das Verhalten wie es von eingebauten Datentypen gewohnt ist zu erzielen, kann der copy und der move assignment Operator `...operator=(...)&` für lvalue qualifiziert definiert werden und der für rvalues nicht. Damit ist eine Anweisung wie in der letzten Zeile in Listing 92 auf der vorherigen Seite nicht mehr möglich.

[142] [Sut01] Lektion 20

6.25.16 Der Conditional Operator

Allgemein: `bedingung ? ausdruck1 : ausdruck2;` ist die Bedingung wahr, liefert der Konditionaloperator den ausdruck1 ansonsten den ausdruck2.

Mit dem Conditional Operator kann eine `if(Condition)max=a; else max=b;` Anweisung durch einen Ausdruck ersetzt werden: `max = Condition ? a : b;`

Die Ausdrücke `a, b` müssen vom selben Typ sein oder in einen Typ `T` implizit konvertiert werden können. Ausserdem kann ein Zweig wie in Listing 93 eine Exception werfen[143].

Bedingte Ausdrücke (Conditonal Expression) können in `constexpr` Ausdrücken verwendet werden (siehe section 6.33 auf Seite 140).

Listing 93: Exceptions in Conditional Expressions

```
1 void function(int* p){
2    int i = p ? *p : throw std::runtime_error("unexpected nullptr");
3    ...
4 }
```

6.25.17 Der sizeof Operator

Die Größe eines Ausdrucks oder eines Types in Byte kann mit dem `sizeof` Operator ermittelt werden. Der Ausdruck `sizeof(Type)` oder `sizeof(expression)` wird zur compile time berechnet und ist konstant!

Listing 94: sizeof Operator

```
1  class A{...};
2  A a;
3  int ai[3];
4  double f(); // Funktionsdeklaration
5  sizeof(A); // Abhängig von der Definition von A
6  sizeof(a); // dto wie bei A
7  sizeof(ai) == 3 * sizeof(int);
8  sizeof(f()) == sizeof(double);
9  sizeof("Hello") == 6; // null terminierter C-String
10 sizeof(void); // Error: void ist ein illegaler sizeof Operand
```

Die Funktion `f()` in Listing 94 wird bei `sizeof(f())` nicht aufgerufen, sie muss auch nicht definiert sein! Diese Eigenschaft von sizeof, eine Compiletime Konstante zu sein, wird in der generischen Programmierung ausgenutzt um z.B: zu prüfen, ob ein Typ von einem anderen abgeleitet ist oder nicht.

Seit C++11 gibt es einen weiteren `sizeof..(parameter-pack)` Operator. Dieser liefert die Anzahl der Argumente im parameter-pack. Siehe section 11 auf Seite 204.

[143] [Str13] S.275 11.1.3 Conditional Expressions

6.25.18 Der Operator typeid und type_info [144]

Der Operator `typeid(expr)` oder `typeid(type)` liefert ein Objekt vom Typ type_info zurück. type_info stellt die Vergleichsoperatoren `==` und `!=` zur Verfügung, eine Operation `name()`, die im schlechtesten Fall einen leeren String zurückliefert und eine Operation `before()` die eine Ordnungsbeziehung zwischen type_info Objekten einführt.

type_info Objekte können nicht kopiert werden, was die Verwendbarkeit erschwert. Zeiger auf diese Objekte können aber gespeichert werden, weil die Objekte, die Speicherklasse static haben. Allerdings ist es nicht gewährleistet, dass bei zwei Aufrufen von `typeid` mit demselben Typ, dasselbe `type_info` objekt zurückgeliefert wird. Soll ein Vergleich durchgeführt werden, müssen die Zeiger dereferenziert und die Objekt miteinander verglichen werden. typeid kann z.B. in der generischen Programmierung beim testen verwendet werden um zu überprüfen, ob ein bestimmter Typ erzeugt wurde.

6.26 cast Operatoren

C++ definiert vier Cast Operatoren. `static_cast, const_cast, dynamic_cast` und `reinterpret_cast`. Sie sind dem C-style cast `(type)` oder `type(expression)` aus folgenden Gründen vorzuziehen:

1. Sie sind für Mensch und Maschine leichter zu identifizieren
2. Sie sind jeweils nur für einen spezifischen Zweck gestaltet
3. Sie sind sichere Casts, sie ermöglichen dem Compiler cast Fehler zu entdecken
4. Sie sind schwierig zu tippen! ;-)

Anwendung `static_cast<type>(expression)`

```
int i=3, j=4;

double result = static_cast<double>(i)/j;
```

C-Style:

```
double result = double(i)/j; //sieht aus wie eine Funktion
```

`static_cast` stellt dieselben Möglichkeiten wie sein Vorfahre `(double)` zur Verfügung und unterliegt denselben Restriktionen. So kann ein `struct` nicht in einen `int` gecastet werden oder ein `double` nicht in einen Pointer. Der `static_cast` lässt darüber hinaus, das Wegcasten von `const` nicht zu.

Dafür gibt es den `const_cast<type>(expression)`.Mit diesem cast operator kann `const` oder `volatile` entfernt werden. Andere Anwendungen sind nicht zulässig und werden vom Compiler zurückgewiesen. Wird die constness eines Objekts

[144] [Ale09] 2.8 A Wrapper Around type_info

weggecastet gibt es keine Garantien für das weitere Systemverhalten, wenn der Compiler ein const Objekt in einen readonly memory Bereich legt.

Im Zusammenhang mit Vererbung bzw. multipler Vererbung sind die Begriffe **upcast, downcast** und **crosscast** von Bedeutung. In der UML werden Basisklassen typischer Weise oben und abgeleitete Klassen darunter, wie in Diagramm 3, dargestellt. Daraus haben sich die Begriffe up- down- und crosscast entwickelt.

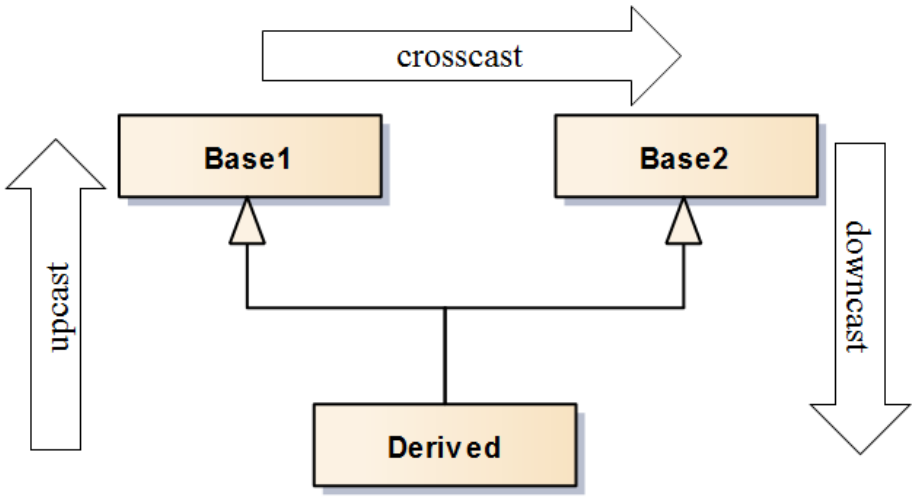

Diagramm 3: upcast, downcast, crosscast und die UML

Das Listing 95 zeigt denselben Sachverhalt in C++.

Der `dynamic_cast<type>(expression)` ermöglicht sicheres **down** und **crosscasten** in einer Vererbungshierarchie. Er kann nur auf polymorphe Klassen angewandt werden. Also auf Klassenhierarchien die mindestens eine `virtual` Operation in der Basis aufweisen. Ist der cast nicht möglich, liefert `dynamic_cast` bei Zeigern 0 und bei Referenzen wirft er eine `bad_cast` Exception. Ein Beispiel für eine sinnvolle Anwendung des `dynamic_casts` ist das acyclic-Visitor-Pattern[145].

Listing 95: upcast, downcast, crosscast

```
1  class Base1 {
2     public virtual ~Base() noexcept;
3  };
4
5  class Base2 {...}
6
7  class Derived:public Base1, public Base2 {...}
8
9  Derived *pD = new Derived();
10
11 Base1 *pB1 = pD; // impliziter upcast, zuweisungskompatibler Typ Base1
12
13 Base2 *pB2 = dynamic_cast<Base2*>(pB1); // crosscast
14
```

[145] http://www.objectmentor.com/resources/articles/acv.pdf

```
15 pD = dynamic_cast<Derived*>(pB1); // downcast
16
17 delete pB1; // Destruktor von Derived, Base2, Base1 wird gerufen
```

Ist Vererbung **nicht** im Spiel, kommt der `static_cast<type>(expression)` zum Einsatz.

`reinterpret_cast<type>(expression)` Das Ergebnis dieses Operators ist fast immer Compiler abhängig und nicht portabel. Die häufigste Anwendung ist der cast zwischen Funktionszeiger Typen oder in der Hardware nahen Programmierung der cast von feststehenden Speicheradressen von sogenannten Memory-mapped-Registern in T* wobei T der Typ des Registers ist, das adressiert wird z.B.: (`unsigned int`).

Bsp:

```
unsigned int controlRegisterAdresse = 0xFFFF0000;

unsigned int * cRAdresse = reinterpret_cast<unsigned int*>(controlRegisterAdresse
);
```

6.27 Alignment

Alignment bezeichnet die Notwendigkeit Objekte eines bestimmten Typs an Adressen mit bestimmten Eigenschaften im Hauptspeicher zu platzieren[146].
Diese Adressen sind immer Potenzen von 2. und können mit wenigen Speicherzugriffen verarbeiten werden[147].

Zur Ermittlung des Alignment stellt C++ den Operator `alignof(type-id)` zur Verfügung[148]. Um für einen Benutzer definierten Typ ein bestimmtes Alignment zu erzwingen den Spezifier[149] `alignas(...)`.
Seit C++11 gibt es in der STL die Funktion `void* align(..)`, die einen Pointer auf einen korrekt ausgerichteten Speicherbereich in einem übergebenen Speicherbereich zurückliefert[150] bzw. das Template
`template<size_t len, size_t Align> struct aligned_storage`, mit dem korrekt ausgerichteter uninitialisierter roher Speicher für die Anzahl `len` Elemente reserviert werden kann.

Das Beispiel in Listing 96 auf der nächsten Seite zeigt die Verwendung von `void* align(..)` im Zusammenspiel mit dynamisch angefordertem Speicher. Die Parameter `first` und `byteSize` werden by reference übergeben und von `align(..)` auf korrekte Werte gesetzt, wenn es möglich ist. Der Rückgabewert entspricht dann dem Wert von `first`. Wenn es nicht möglich ist, wird `nullptr` zurückgeliefert und die beiden Argumente nicht verändert.

[146] http://en.cppreference.com/w/cpp/language/object#Alignment
[147] https://de.wikipedia.org/wiki/Speicherausrichtung
[148] http://en.cppreference.com/w/cpp/language/alignof
[149] http://en.cppreference.com/w/cpp/language/alignas
[150] http://en.cppreference.com/w/cpp/memory/align

Listing 96: Alignment und dynamischer Speicher

```
template<class T>
class Vector{
public:
   using iterator = T*;
   Vector(std::size_t firstCapacity = 10)
   : capacity(firstCapacity),
   byteSize(sizeof(T) * firstCapacity + (alignof(T)-1) ),
   data(operator new(byteSize)),
   first(data),
   nextPos(reinterpret_cast<T*>(std::align(
                        alignof(T),
                        sizeof(T),
                        first, // by reference
                        byteSize) // by reference
                        )
         )
   {
      std::cout << "Vector() Capacity: " << capacity << std::endl;
      std::cout << "data: " << data << " nextPos: " << nextPos << std::endl;
   }
   template<class ...Params>
   iterator emplace(Params...params){
      if(size() >= capacity){
         resize();
      }
      return new(nextPos++) T(std::forward<Params>(params)...);
   }
   ~Vector(){
      for(--nextPos; nextPos >= static_cast<T*>(first); --nextPos)
         nextPos->~T(); //dtor für alle Elemente in umgekehrter reihenfolge
      operator delete(data); // Speicher freigeben
   }
   std::size_t size(){
      return nextPos - static_cast<T*>(first);
   }
private:
   void resize(){
      // Übung: hier die vergrößerung implementieren
   }
   std::size_t capacity;
   std::size_t byteSize;
   void* data;
   void* first;
   T* nextPos;
};

void demoVector(){
   Vector<A> vA;
   (*vA.emplace()).opConst();
   (*vA.emplace(43)).operation();
```

```
51
52    cout << "vA.size(): " << vA.size() << endl;
53 }
54 Ausgabe:
55 Vector() Capacity: 10
56 data: 0x20075d68 nextPos: 0x20075d68
57 A::A()
58 A::opConst() i: 42
59 A::A(int i: 43)
60 A::operation() i: 43
61 vA.size(): 2
62 A::~A() i: 43
63 A::~A() i: 42
```

6.28 Zeiger und Arrays

Die Bedeutung von nativen Zeigern und Arrays hat in C++ gegenüber C stark an Bedeutung verloren. Sie werden ersetzt durch die Mengenobjekte (Container) und SmartPointer der STL. Hier soll ihre Semantik dargestellt werden, um diese beim Überladen der entsprechenden Operatoren berücksichtigen zu können.

6.28.1 Definition und Initialisierung von Arrays

Arrays oder Felder sind (homogene) Mengenobjekte, sind Ansammlungen von Objekten gleichen Typs die hintereinander im Hauptspeicher liegen. Bei benutzerdefinierten Typen werden alle Elemente mit dem default Konstruktor erzeugt.

In Listing 97 werden verschiedene Möglichkeiten ein Feld zu definieren gezeigt.

Listing 97: Definition eines Feldes

```
1 typ feldname[konstanter Ausdruck];
2 int ia[3]; // uninitialisiertes Feld mit 3 ints
3
4 // Mit Initialisierung:
5 typ feldname[size]= { initialisierungsliste };
6 int ia[3] = { 1, 2 }; // nur die ersten Elemente initialisiert
7 int ia[ ] = {1 , 2, 3 };
```

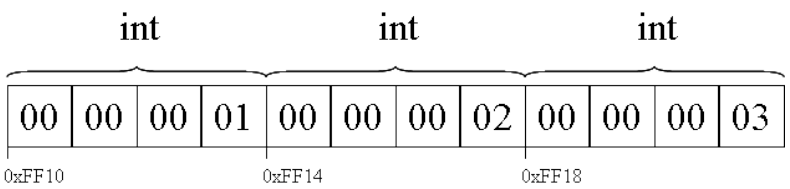

Abbildung 5: Schematische Darstellung eines eindimensionalen Arrays

Durch die Definition eines Feldes wird Speicherplatz für eine konstante Anzahl Elemente reserviert[151]. Jedes Element ist vom Typ `typ` der bei der Deklaration angegeben wird.

Der Name eines Feldes ist ein Ausdruck der in den Typ: Zeiger auf ein Element (Bsp: int*) konvertiert wird (decay) und hat den Wert: Adresse des ersten Elements.

Bsp:
`ia == &ia[0] == 0xFF10`.

Der **konstante Ausdruck** in den eckigen Klammern bei der Definition gibt die Anzahl der Elemente in dem Feld an. `size` kann weggelassen werden, wenn eine `initialisierungsliste` vorhanden ist; `size` entspricht dann der Anzahl der Elemente in der `initialisierungsliste`. Wird eine `size` Größer als die Anzahl der Elemente angegeben, werden nur die vorderen Elemente mit Werten belegt. Wird `size` explizit angegeben, ist es ein Fehler, mehr Elemente in der `initialisierungsliste` anzugeben.

6.28.2 Zugriff auf Arrayelemente

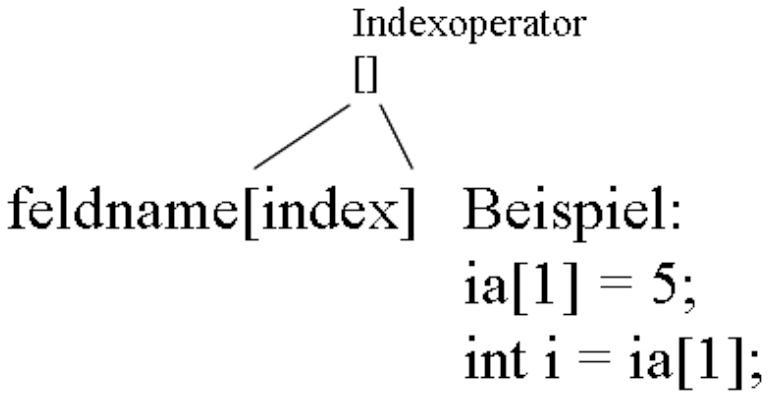

Abbildung 6: Der Indexoperator

Der Wertebereich von `index` ist: `0 bis size-1`

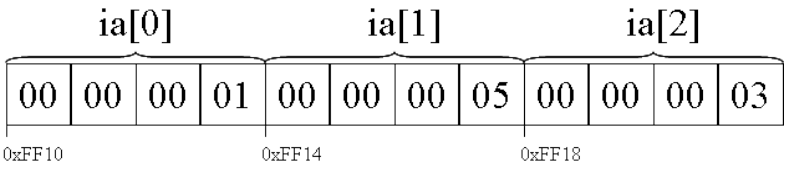

Abbildung 7: Array und Adressierung der Elemente

feldname[index] ist ein Ausdruck vom Typ Referenz auf den Elementtyp eines Elementes und hat den Wert des jeweiligen Elements. Lies: `ia[1]` -> ia an der Stelle 1.

[151] Die konkrete Abbildung von int ist von der jeweiligen Rechnerarchitektur abhängig (little/big endian)

`ia[1]` ist vom Typ int& und hat nach der obigen Zuweisung den Wert 5.

Der Indexoperator ermittelt den Wert des Elements an der Stelle:
AnfangsAdresse + index * sizeof(typ)
Bsp: feldname == 0xFF10, index == 1, sizeof(int) == 4 Byte
0xFF10 + 1 * 4 = 0xFF14 also das 2.te Element in dem Feld. (Das erste hat den Index 0!)

Zuweisungen zu Feldern sind nicht möglich:
`ia = { 1, 2, 3}; //Fehler`

6.28.3 Mehrdimensionale Arrays

Definition und Initialisierung:
```
int ia2D[][] = {{1,2 ,3},{4, 5, 6}};
```

Mehrdimensionale Arrays werden als Arrays von Arrays abgelegt. Der Typ von `ia2D[0]` ist ein Zeiger auf ein Array von **int** das die Elemente 1, 2, 3 enthält. Der Ausdruck `ia2D[0][1]` hat also den Wert 2.

6.28.4 Definition und Initialisierung von Zeigern

Listing 98: Definition und Initialisierung von Zeigern

```
1 Typ * name ; // ohne Initialisierung ist ein Programmierfehler!
2 Typ * name = nullptr; // Initialisierung, zeigt nicht auf ein Objekt
3 Typ * name = wert; // Initialisierung
4
5 // Bsp.:
6 int * pi = nullptr;
7 int i = 3;
8 pi = &i;
```

Der * ist bei der Deklaration/Definition ein Typmodifier. Er deklariert eine Variable `pi` vom Typ Zeiger auf ein Objekt vom Typ **int**. `pi` kann die Adresse eines **int** aufnehmen. Das & ist in diesem Zusammenhang der Adressoperator. Er ermittelt die Adresse des Objektes `i`, die `pi` zugewiesen wird. Lies: &i Adresse von i. `pi` kann auf Elemente in dem Array `ia` zeigen.
Bsp.:
`pi = ia; //der name des feldes wird in einen zeiger int* konvertiert (decay)`

Jetzt zeigt `pi` auf das erste Element in dem Array `ia`.
Alternativ: `pi = &ia[0]; //lies: pi gleich Adresse von ia an der Stelle null`

6.28.5 nullptr und std::nullptr_t

Mit dem Schlüsselwort[152] `nullptr` und dem als fundamental geltenden Datentyp `std::nullptr_t` wurde die Mehrdeutigkeit des Wertes „Null 0“ zwischen integralen Typen und Pointern aufgelöst bzw. ein eigener Wert für „ungültige“ Pointer geschaffen.

Listing 99: nullptr

```
1 void f(int);
2 void f(void*);
3
4 f(0);       // calls f(int)
5 f(NULL);    // calls f(int) if NULL ist 0, ambiguous otherwise
6 f(nullptr); // calls f(void*);
```

`nullptr` ist ein Schlüsselwort das automatisch in jeden Pointertyp konvertiert wird, aber nicht in den integralen Wert Null 0. Der Typ von `nullptr` ist `std::nullptr_t` der in `<cstddef>` definiert ist.

Damit ist es möglich Operationen zu überladen, die einen Pointer erwarten (Listing 99) und sie mit einem ungültigen Pointer aufzurufen, ohne dass dabei Mehrdeutigkeiten zu Funktionen, die einen integralen Typ erwarten, entstehen.

6.28.6 Zugriff auf den Inhalt

`*pi = 3;`
Dereferenzierungsoperator: **`*pi` ist ein Ausdruck** vom Typ **`int&`** und hat den Wert des Objekts auf das `pi` zeigt. Mit `*pi` lässt sich genauso bequem auf den Speicher zugreifen wie mit einer „normalen“ Variablen. Das erste Element in `ia` hat nach der Zuweisung den Wert 3.

6.28.7 Zeiger Arithmetik und random_access_iterator

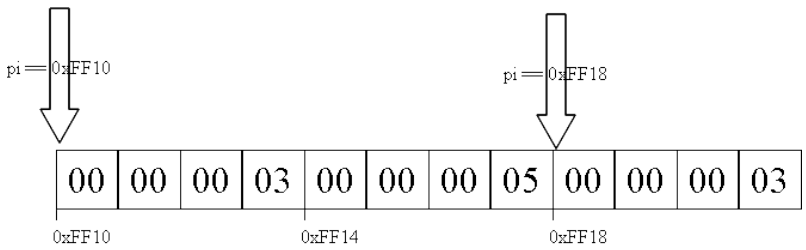

Abbildung 8: Zeiger Arithmetik

Nach der Zuweisung `pi = ia;` zeigt `pi` auf das erste Element in `ia`. `pi` hat den Wert: `&ia[0]`.

[152][Jos12] New Language Features

Der Typ des Zeigers (int) ermöglicht eine sinnvolle Anwendung der arithmetischen Operatoren auf einen Zeiger. `pi = pi + 2` entspricht der Berechnung `pi + 2*sizeof(int)`. Dadurch zeigt `pi` nach der Addition auf den Anfang des entsprechenden int-Objekts in dem Feld ia! Diese Semantik haben auch die `random_access_iterator`en der STL. Mit Zeigern und `random_access_iterator`en lassen sich alle arithmetischen Operationen in diesem Sinne durchführen.

6.29 Referenzen

Referenzen werden hauptsächlich als Parameter von Funktionen verwendet. Bsp.:
`void setName(std::string const& name);`
Der Parameter `name` ist eine Referenz auf ein konstantes string Objekt[153]. Sie wird beim Funktionsaufruf mit dem übergebenen Objekt verknüpft.
Eine Referenz auf ein konstantes Objekt ist ähnlich mit einem Call by Value, die gerufene Funktion kann das übergebene Objekt nicht verändern, die Kosten für eine Kopie entfallen aber. Das ist insbesondere für Klassen mit hohen Kopierkosten von Bedeutung. Bei primitiven Datentypen ist Call by Value die günstigere Variante, da eine Referenz in irgendeiner Weise über einen Zeiger realisiert werden muss. Der Vorteil von Referenzen gegenüber der Übergabe der Adresse eines Objekts ist, der Aufrufer kann dieselbe Syntax verwenden wie bei einem call by value.
Das Kaufmannsund / ampersand (`T&`) ist bei der Definition ein Typemodifier und deklariert eine Referenz auf einen L-Value Ausdruck, eine sogenannte lvalue reference (einen Aliasnamen).
Mit dem Standard C++11 kommen zwei weiter dazu: rvalue references und forwarding references. Beide werden mit einem doppelten Ampersand (`T&&`) deklariert.
Rvalue references können nur mit temporären Ausdrücken verknüpft werden.
Forwarding references kommen nur in Templates vor und **sollten ausschließlich mit `std::forward` verwendet werden**, da sie zu unterschiedlichen Typen evaluieren[154]. In diesem Fall sprach Scott Meyers von einer „Universal Reference“, da sie sowohl für L-Value als auch für R-Value Ausdrücke zur Anwendung kommt[155]. Im Standard wird von „forwarding reference“ gesprochen[156]. Bis zur vollständigen Überarbeitung werden in diesem Text beide Begriffe verwendet.

Die neue Syntax wurde eingeführt um die sogenannte „Move Semantik“[157] und „Perfect Forwarding“[158] implementieren zu können.

Referenzen müssen bei der Definition mit dem zu referenzierenden Objekt verknüpft werden. Nachträglich können Referenzen nicht mehr mit anderen Objekten verknüpft werden.

[153] von rechts nach links lesen hilft
[154] section 6.30 auf Seite 113
[155] see https://isocpp.org/files/papers/N4164.pdf
[156] http://en.cppreference.com/w/cpp/language/reference#Forwarding_references
[157] Siehe section 6.9 auf Seite 42
[158] siehe [Bec13] und [Mey12]

Listing 100: Referenzen

```
int i = 0; // Definition eines int Objektes mit Initialisierung
int k = 3; // Definition eines int Objektes mit Initialisierung
int &ri = i; // Definition einer Referenz auf ein int Objekt mit Verknüpfung
ri = k; // Zuweisung des Werts von k an i über die Referenz ri
int &&rri1 = 42; // Definition einer R-Value Referenz auf eine Konstante
int &&rri2 = i; // Error, invalid initialization
```

Listing 101: forwarding references

```
class Widget{};

Widget createWidget(){ return Widget(); }

void rValueReference(Widget&&){ // kann nur für R-Value Ausdrücke aufgerufen werden
   cout << "rValueReference(Widget&&)" << endl;
}

template<class T>
void forwardingReference(T&& arg){
   cout << "forwardingReference";

   if(std::is_lvalue_reference<decltype(arg)>::value)
      cout << "(T& arg) arg is L-Value Expression of type L-Value Reference" <<
          endl;

   if(std::is_rvalue_reference<decltype(arg)>::value)
      cout << "(T&& arg) arg is L-Value Expression of type R-Value Reference" <<
          endl;
}

int main(){
   cout << "Forwarding References" << endl;
   cout << "===main Widget widget;" << endl;
   Widget widget; // widget is L-Value Expression of type Widget

   cout << "===main auto && ref1 = makeWidget();" << endl;
   auto && ref1 = makeWidget();
   if(is_rvalue_reference<decltype(ref1)>::value)
      cout << "ref1 is L-Value Expression of type R-Value Reference" << endl;

   cout << "===main auto && ref2 = widget;" << endl;
   auto && ref2 = widget;
   if(is_lvalue_reference<decltype(ref2)>::value)
      cout << "ref2 is L-Value Expression of type L-Value Reference" << endl;

   // error: cannot bind 'Widget' lvalue to 'Widget&&'
   //rValueReference(widget);

   cout << "===main rValueReference(makeWidget());" << endl;
   rValueReference(makeWidget());

```

```
41    cout << "===main forwardingReference(widget);" << endl;
42    forwardingReference(widget);
43    cout << "===main forwardingReference(makeWidget());" << endl;
44    forwardingReference(makeWidget());
45 }
```

Die Funktion `rValueReference(..)` kann nur mit einem R-Value Ausdruck aufgerufen werden.

Listing 102: Forwarding References Ausgabe

```
1  Forwarding References
2  ===main Widget widget;
3  ===main auto && ref1 = makeWidget();
4  ref1 is L-Value Expression of type R-Value Reference
5  ===main auto && ref2 = widget;
6  ref2 is L-Value Expression of type L-Value Reference
7  ===main rValueReference(makeWidget());
8  rValueReference(Widget&&)
9  ===main forwardingReference(widget);
10 forwardingReference(T& arg) arg is L-Value Expression of type L-Value Reference
11 ===main forwardingReference(makeWidget());
12 forwardingReference(T&& arg) arg is L-Value Expression of type R-Value Reference
```

6.30 Automatische Typerkennung (type deduction)

Dieses Kapitel basiert auf [Mey15] CHAPTER 1 Deducing Types. Es ist eine knappe Zusammenfassung der darin enthaltenen Aussagen.
Einige Kapitel in diesem Script enthalten redundante Aussagen zu diesem Thema, diese werden nach und nach konsolidiert und hier eingearbeitet.

Mit den Versionen C++11 ff. ergeben sich weitreichende Konsequenzen die dieses Kapitel notwendig machen.
Vor C++11 gab es eine kleine Menge Regeln zur automatischen Typerkennung bei Funktionstemplates[159].
Mit C++11 kommen zwei weitere hinzu, für `auto` und für `decltype`.
C++14 erweitert die Anwendungsmöglichkeiten von `auto`, `decltype` und `decltype(auto)`[160].
C++17 erweitert die Typerkennung von Funktionstemplates auf Konstruktoren von Klassentemplates. Dadurch werden verschiedene Hilfsfunktionen wie `std::make_pair` überflüssig[161].

Damit wird C++ leichter anpassbar, weil ein Typ nur an einer Stelle festgelegt werden muss. Der Code wird dadurch aber teilweise auch schwerer verständlich. Ohne ein solides Verständnis, wie die automatische Typerkennung arbeitet, ist effective Programmierung mit modernem C++ nicht möglich.

[159] section 3.12 auf Seite 27
[160] http://en.cppreference.com/w/cpp/language/auto
[161] http://www.bfilipek.com/2017/01/cpp17features.html#template-argument-deduction-for-class-templates

6.30.1 type deduction und Funktionstemplates

Abstrakt kann ein Funktionstemplate wie in Listing 103 beschrieben werden.

Listing 103: Pseudocode Funktionstemplate

```
template<typename T>
void f(ParamType param);
```

Ein Aufruf könnte so aussehen: `f(ausdruck);`.

ParamType kann folgende drei Ausprägungen haben:

1. Referenz oder Zeiger aber keine forwarding reference
2. forwarding reference
3. weder Referenz noch Zeiger

Der erste Fall mit Referenzen ist am einfachsten:

```
// mit ParamType: T&
template<typename T>
void f(T& param);

int x = 27;
int const cx = x;
int const & rx = x;

f(x);    // T: int,        ParamType: int&
f(cx);   // T: int const,  ParamType: int const&
f(rx);   // T: int const,  ParamType: int const&

// oder mit ParamType: const T&
template<typename T>
void f(T const& param);

f(x);    // T: int, ParamType: int const&
f(cx);   // T: int, ParamType: int const&
f(rx);   // T: int, ParamType: int const&
```

Dass `rx` eine Referenz ist, wird ignoriert.

Für Zeiger gilt daselbe analog:

```
template<typename T>
void f(T* param);

int x = 27;
int const *px = &x;

f(&x);   // T: int,        ParamType: int*
f(px);   // T: const int,  ParamType: int const *
```

Der zweite Fall mit forwarding references ist weniger offensichtlich:

```
template<typename T>
void f(T&& param);

int x = 27;       // lvalue
const int cx = x; // lvalue
const int& rx = x;// lvalue

f(x);    // T: int&,       ParamType: int&
f(cx);   // T: const int&, ParamType: const int&
f(rx);   // T: const int&, ParamType: const int&
// 27 ist rvalue!
f(27);   // T: int,        ParamType: int&&
```

Bei forwarding references (`T&&`) wird zwischen rvalue und lvalue Argumenten unterschieden, der ParamType wird bei einem rvalue Argument zu einer rvalue reference (`int&&`)!

Beim dritten Fall wird das Argument by Value übergeben:

```
template<typename T>
void f(T param);

int x = 27;       // lvalue
const int cx = x;    // lvalue
const int& rx = x;   // lvalue

f(x);    // T: int, ParamType: int
f(cx);   // T: int, ParamType: int
f(rx);   // T: int, ParamType: int
```

`const` und `volatile` wird nur bei Value Parametern ignoriert!

Arrays als Argumente: By Value, der Arrayname wird zu einem Pointer (decay) mit dem Wert der Adresse des ersten Elements.

```
template<typename T>
void f(T param);

const char name[] = "Gerdi"; // type is const char[6]
const char * ptrToName = name; // decay to pointer

f(name);    // T: char const*, ParamType: char const*
```

Arrays als Argumente: By Reference

```
// return size of an array as a compile-time constant
template<typename T, std::size_t N>
constexpr auto arraySize( T(&)[N] ){ return N; }

int keyVals[] = {1, 3, 7};
std::array<int, arraySize(keyVals)> mappedVals;
```

Parameter können keine echten Arrays sein, aber Referenzen auf Arrays sind möglich. Der Parametertype `T(&)[N]` ist eine Referenz auf ein Array mit `N` Elementen. Der Parametername ist weggelassen, weil in diesem Beispiel nur die Anzahl `N` der Elemente von Interesse ist. Der Funktion `arraySize(T(&)[N])` können nur Arrays übergeben werden.

Funktionen als Argumente:

```
1  template<typename T>
2  void f1(T param);
3
4  template<typename T>
5  void f2(T& param);
6
7  void someFunc(int, double); // type: void(int,double)
8
9
10 f1(name);   // T, ParamType: void(*)(int, double) Pointer
11 f2(name);   // T, ParamType: void(&)(int, double) Reference
```

Bei `f1(T param)` wird der ParamType zu **`void(*)(int, double)`**, ein Zeiger auf eine Funktion, der Funktionsname wird zu einem Pointer auf die Funktion (decay), der Wert ist die Funktionsadresse.
Bei `f2(T&)` wird der ParamType zu **`void(&)(int, double)`**, eine Referenz auf eine Funktion, die nichts zurückliefert und als Argument einen **`int`** und einen **`double`** erwartet.

Zusammenfassung:

- Während der Template Type Deduction werden Referenzenvariablen behandelt wie nicht Referenzvariablen
- Bei forwarding reference parameter werden lvalue argumente speziell behandelt
- Bei by-value parametern wird **`const`** und **`volatile`** ignoriert
- Array- oder Funktionsnamen (*decay*) werden zu Zeigern, außer wenn sie Referenzen initialisieren

6.30.2 type deduction und Klassentemplates (C++17)

Bei Klassentemplates werden die Template Argumente wie bei Funktionstemplates aus den Call Argumenten des Konstruktors erkannt[162], vorausgesetzt, der Konstruktor hat für alle Template Parameter einen Call Parameter:
`std::pair p(42, 'z')`, der Typ von p ist `std::pair<int, char>`.

[162] http://www.bfilipek.com/2017/01/cpp17features.html#template-argument-deduction-for-class-templates

6.30.3 auto type deduction

Bei `auto` type deduction gelten dieselben Regeln wie bei Templates. Das Schlüsselwort `auto` übernimmt die Rolle des Template Parameters `T`, der Variablenname die Rolle des Parameternamens und der Ausdruck zur Initialisierung die Rolle des Arguments.

```
auto x = 27;          // case 3, type of x is int
const auto cx = x;    // case 3, type of cx is const int
const auto& rx = x;   // case 1, type of rx is const int &
```

Es gelten dieselben Ausprägungen für den ParamType wie bei Templates. Der case 1 und 3 sind bereits dargelegt.

Case 2 funktioniert wie erwartet, auch für Arrays und Funktionen.

```
auto&& fref1 = x; // int& lvalue reference
auto&& fref2 = cx;// int const& const lvalue reference
auto&& fref3 = 27;// int&& rvalue reference
```

Bis C++17: Bei der Initialisierung von Variablen mit der (ab C++11) einheitlichen Initialisierungssyntax:[163], `auto x3 = {27}`, wie in Listing 104 ergibt sich nicht der erwartete Typ mit `auto`.
Ab C++17: ist x3 und x4 ein `int`. Bei einer Liste mit Werten in geschwungenen Klammern (braced-init-list) mit nur einem Element, wird der Typ des Elements erkannt, eine Liste mit mehr als einem Element ist ungültig (ill-formed).

Listing 104: auto und braced-init-list

```
int x1 = {27}; // x1 ist int
int x2{27};    // x2 ist int
auto x3 = {27};// Bis C++17: x3 ist std::initializer_list<int>, value 27
auto x4{27};   // x4 wie x3

//Bis C++17
template<typename T>
void f(T param);

f({27, 30}); // error! can't deduce type for T
```

Der Templateparamter wird nicht als `std::initializer_list<int>` interpretiert. Es müsste zuerst der Typ `int` für `std::initializer_list<T>` und anschließend diese als der Typ `T` des Templates ermittelt werden.

C++14 erlaubt mit dem Schlüsselwort `auto` die Typerkennung beim Rückgabetyp einer Funktion und die Typerkennung bei Parametern von Lambdas. Hier gelten aber die Regeln der template type deduction und nicht die von `auto`. Daher wird `newValue` beim Aufruf nicht zu einer `std::initializer_list<int>`.

```
auto createInitList(){
   return {1, 2, 4}; // error: can't deduce type for {1, 2, 4}
```

[163] section 6.14 auf Seite 47

```
}

std::vector<int> v;
...
auto resetV =
                  //vector::operator=(std::initializer_list<T>)
   [&v](const auto& newValue){ v = newValue; };
   ...
resetV({1, 2}); // error: can't deduce type for {1, 2}
```

Zusammenfassung:

- `auto` und template type deduction ist gleich, mit Ausnahme einer Werteliste in geschwungenen Klammern `{ initializer-value-list }`
- `auto` in function return type oder lambda parametern wird behandelt wie template type deduction

6.30.4 Das Schlüsselwort decltype C++11

Mit dem Operator `decltype(expression)` kann der genaue Typ eines Namens oder eines Ausdrucks ermittelt werden. Der Ausdruck in den runden Klammern wird nicht ausgewertet, es wird lediglich der Ergebnistyp vom Compiler während der compile time ermittelt[164].
Dieses Schlüsselwort ersetzt die inkonsistente non-standard Erweiterung `typeof` verschiedener Compiler.
Die primäre Anwendung in C++11 ist die Ermittlung des Rückgabetyps der von Argumenten abhängig ist bei Funktionstemplates[165].

Listing 105: Beispiel decltype vs auto

```
std::map<std::string, float> coll;

// decltype(coll) is std::map<std::string, float>
decltype(coll)::value_type elem;

double& f();

decltype(f()) rd = f(); // type of rd is double&
auto d = f();           // type of d is double
```

Mit C++14 wurde der Standard um die Erkennung des Rückgabetyps aller Funktionen, wie schon vorher bei lambdas[166], erweitert inclusive von Funktionen mit mehreren **`return`** Statements, solange alle zu demselben Typ ausgewertet werden.

[164] siehe http://en.wikipedia.org/wiki/Decltype
[165] section 6.32.2 auf Seite 126
[166] section 6.32.11 auf Seite 136

Listing 106: Ermittlung des Rückgabetyps mit auto C++14

```
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i){
   authenticateUser();
   return c[i];
}
...
authAndAccess(container, 5) = 42; // error: rvalue expression
```

In Listing 106 ist eine Funktion abgebildet, die vor dem Zugriff auf einen `Container` symbolisch die Benutzerrechte prüft und dann das Objekt zurückliefert. Der Index Operator[167] liefert typischer Weise eine Referenz auf das Element zurück. Mit type deduction `auto` würde die Funktion aber den Wert by Value zurückliefern und damit wäre eine Zuweisung nicht möglich.

Soll die Funktion den genauen Typ des Ausdrucks zurückliefern, muss die Syntax mit *specifier*: `decltype(auto)` wie in Listing 107 verwendet werden. Das teilt dem Compiler mit, ermittle den Typ (`auto`) verwende aber die Regeln von `decltype`.

Listing 107: Ermittlung des Rückgabetyps mit decltype(auto) C++14

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i){
   authenticateUser();
   return c[i];
}

Widget w;
Widget const& crw = w;
auto aW = crw; // type of aW is Widget
decltype(auto) crw2 = crw; // type of crw2 is Widget const&
```

ACHTUNG:
Wenn ein lvalue Ausdruck komplexer ist als nur ein Name[168], liefert `decltype` eine Referenz (`T&`) auf den Typ!
Mit `int x = 0;` ist `x` der Name einer Variablen und `decltype(x)` liefert `int`. Aber der lvalue Ausdruck `(x)` ist ein komplizierterer Ausdruck als ein Name und daher liefert `decltype( (x)) int&` als Ergebnistyp! Mit fatalen Auswirkungen im Zusammenhang von `decltype(auto)` bei Funktionen.

Listing 108: Probleme mit decltype(auto) C++14

```
decltype(auto) f(){ // return type is int&
   int x = 0;
   return (x);
}
```

Die Funktion `f` in Listing 108 liefert eine Referenz auf ein lokales Objekt, das nach dem Aufruf nicht mehr existiert[169]!

[167] section 6.25.13 auf Seite 99
[168] [Mey15] Item 3 S. 29
[169] O-Ton: "That´s the kind of code that puts you on the express train to undefined behavior"

Zusammenfassung:

- `decltype` liefert normalerweise den genauen Typ des Ausdrucks
- für komplexere lvalue Ausdrücke `(x)` liefert `decltype T&` (außer für einfache Namen)
- C++14 `decltype(auto)` type deduction nach den Regeln von `decltype`

6.30.5 Reference Collapsing Rules C++11

Zur Erinnerung: Referenzen auf Referenzen `Widget & & rrw = ...` sind in C++ nicht erlaubt und verursachen einen Compilerfehler.

In Templates wurde daher vor C++11 so etwas wie `remove_reference<T>::type` an den entsprechenden Stellen verwendet.

Mit C++11 wurden folgende 2 „Reference Collapsing Rules“ definiert[170].

1. Eine Forwarding Referenz auf eine R-Value Referenz wird zu einer R-Value Referenz
   - `A&& &&` wird zu `A&&`
2. Alle anderen Referenzen auf Referenzen werden zu einer L-Value Referenz
   - `A& &` wird zu `A&`
   - `A& &&` wird zu `A&`
   - `A&& &` wird zu `A&`

Diese Regeln werden im Zusammenhang mit automatischer Typerkennung (type deduction) bei Templates und `auto` deklarierten Variablen sowie bei der Definition und Anwendung von `typedef`s und in abgewandelter Form bei `decltype`, angewandt[171].

Ausserdem wurde eine spezielle Argument Deduction Rule für Templatefunktionen definiert, die einen Forwarding Referenz Parameter haben:

Listing 109: Templatefunktionen mit Forwarding Referenz Parametern

```
1 template<typename T>
2 void foo(T&&);
```

In Listing 109 wird folgendes angewendet:

1. Wenn foo mit einem **L-Value Argument** von `A` aufgerufen wird, wird `T` zu `A&` und darus resultiert `foo(A& &&)`. Mit den Reference Collapsing Rules wird der Parametertyp effektiv zu `A&`.
2. Wenn foo mit eine **R-Value Argument** von `A` aufgerufen wird, wird `T` zu `A` und darus resultiert `foo(A &&)`. Der Parametertyp wird `A&&`

[170] [Mey12]
[171] [Mey15] Chapter 1 Deducing Types

6.31 `std::move(..)` und `std::forward<..>()` C++11

`std::move(expression)` verschiebt nichts sondern castet einen Ausdruck bedingungslos in einen R-Value Ausdruck[172]! Dadurch wird beim Aufruf von überladenen Funktionen, die Funktion mit einem R-Value Referenz Parameter, soweit vorhanden, ausgewählt, z.B. der Move Konstruktor in section 6.9 auf Seite 42.
`std::forward<type>(..)` castet in den angegebenen Typ.

Die Eigenschaft, ein R-Value Ausdruck zu sein ist sehr flüchtig. Sobald ein Parameter beim Funktionsaufruf mit einem temporären Objekt initialisiert ist, ist der Parametername kein R-Value Ausdruck mehr sonder ein L-Value Ausdruck, es läßt sich davon immmer die Adresse ermitteln: `&paramName`. Deswegen sollten R-Value Referenzen immer via `std::move` weitergegeben werden und Forwarding Referenzen via `std::forward`, wenn die Eigenschaft, ein *R-Value Ausdruck* zu sein, erhalten bleiben soll[173].

Im Folgenden soll die Verwendung und Auswirkung von `std::move` und `std::forward` im Kontext von „normalen“ Funktionen, Templates und Lambdas mit `T&&` gezeigt werden[174].
Dazu werden in Listing 110 die Klassen `Base` und `Derived` definiert, mit der überladenen reference qualified Operation **`void`** `rqOp()`[175] und einer nicht reference qualified Operation **`void`** `op()`, sowie der überladenen Funktion **`void`** `f(Base` **`const`**`&)` für L-Value Ausdrücke und **`void`** `f(Base &&)` für R-Value Ausdrücke.

Listing 110: Überladungen mit R- und L-Value

```
1  class Base{
2  public:
3     virtual ~Base() = default;
4     virtual
5     void rqOp() &&;
6     virtual void rqOp() &;
7     virtual void op();
8
9  };
10 class Derived : public Base{
11 public:
12    //virtual
13    void rqOp() &&; //override;
14    virtual void rqOp() & override;
15    virtual void op() override;
16 };
17
18 void Base::rqOp() &&{
19    std::cout << "Base::rqOp() &&" << std::endl;
20 }
21 void Base::rqOp() &{
22    std::cout << "Base::rqOp() &" << std::endl;
```

[172] [Mey15] Item 23 Understand std::move and std::forward
[173] [Mey15] Item 25 Use std::move on rvalue references, std::forward on universal references
[174] **`return`** `std::move(expression)` siehe section 6.9 auf Seite 42
[175] section 6.32.10 auf Seite 135

```
}
void Base::op(){
   std::cout << "Base::op()" << std::endl;
}
// dasselbe für Derived:: ...(){ "Derived::...()" }

//-----------------------------------------------------------------------
//Überladene Funktion f() für L- und R-Value Ausdrücke
void f(Base const&){ // 1 für L-Value Ausdrücke
   cout << "f(Base const&)" << endl;
}
void f(Base&&){      //2 für R-Value Ausdrücke
   cout << "f(Base&&)" << endl;
}
```

In in Listing 111 werden die Funktionen aus Listing 110 auf der vorherigen Seite verwendet. Der Aufruf der Funktion `f(t)` oder der Aufruf der reference qualified überladenen Operation `t.rqOp()` unter Verwendung des Funktionsparameter `t` führt nie zur Auswahl der R-Value reference qualified Methode, weil der Parameter selbst ein L-Value Ausdruck ist. Darum muss `std::move(..)` verwendet werden, um den Parameter in einen R-Value Ausdruck zu casten, oder `std::forward<..>(..)` bei Forwarding Referenzen in den ursprünglichen Typ.

Listing 111: Anwendung von std::move() und std::forward<..>(..)

```
// Überladene Funktion die f(t) und t.rqOp() aufruft
// Item 25 Use move on rvalue references, std::forward on universal references
void callF(Base&& t){   // 3
   f(std::move(t));
   std::move(t).rqOp();
   std::move(t).op();
}
void callF(Base& t){ // 4
   f(t);
   t.rqOp();
   t.op();
}
// template mit forwarding Reference
// das f(t) und t.rqOp() aufruft
template<class T>
void callFT(T&& t){  // 5
   f(std::forward<T>(t));
   std::forward<T>(t).rqOp();
   std::forward<T>(t).op();
}
// lambda mit forwarding Reference
// das f(t) und t.rqOp() aufruft
auto callFL = [](auto&& t){ // 6
   f(std::forward<decltype(t)>(t));
   std::forward<decltype(t)>(t).rqOp();
   std::forward<decltype(t)>(t).op();
};
```

```
28
29 Derived rValue(){ return Derived(); } // Aufruf ist ein Derived R-Value Ausdruck
```

Der Typ des Template Type Parameters T bzw. `auto` wird durch die Typededuction Rules bei Templates und die Reference Collapsing Rules[176] festgelegt[177]

Ohne `std::move` bei 3 bzw. `std::forward` bei 5 und 6 würde jeweils `f(Base const &)` und `rqOp()&` gerufen. Für die nicht reference qualified operation `op()` spielt es keine Rolle, ob `std::forward` verwendet wird, da sie sowohl für L- als auch für R-Value Ausdrücke gerufen wird.

Achtung: Das Objekt das durch `t` referenziert wird, kann jeweils nach dem ersten `std::move` oder `std::forward` nicht mehr benutzt werden. Dieses Beispiel soll nur zeigen, welche Operationen gerufen werden. In section 6.9 auf Seite 42 ist der Move Konstruktor und seine Auswirkungen beschrieben.

Die Verwendung ist in Listing 112 dargestellt:

Listing 112: Die Anwendung der überladenen Operationen

```
1  int main(){
2     cout << "PerfectForwarding" << endl;
3
4     int no = 0;
5     Derived widget;
6     Base& w = widget;
7     Base&& baseRValueReference = rValue();
8
9     cout << "----------------------" << endl;
10    cout << "callF(w); //" << ++no << endl;
11    callF(w);
12    cout << "----------------------" << endl;
13    cout << "callF(rValue()); //" << ++no << endl;
14    callF(rValue());
15    cout << "----------------------" << endl;
16    cout << "callF(std::forward<Base&&>(baseRValueReference)); //" << ++no  << endl;
17    callF(std::forward<Base&&>(baseRValueReference));
18    cout << "----------------------" << endl;
19    cout << "callFL(w); //" << ++no << endl;
20    callFL(w);
21    cout << "----------------------" << endl;
22    cout << "callFL(rValue()); //" << ++no<< endl;
23    callFL(rValue());
24    cout << "----------------------" << endl;
25    cout << "callFT(w); //" << ++no << endl;
26    callFT(w);
27    cout << "----------------------" << endl;
28    cout << "callFT(rValue()); //" << ++no << endl;
29    callFT(rValue());
30    cout << "----------------------" << endl;
```

[176] section 6.30.5 auf Seite 120
[177] [Mey15] Chapter 1 und Item 28

```
31 }
```

Listing 113: Ausgabe

```
1  PerfectForwarding
2  ----------------------
3  callF(w); //1
4  f(Base const&)
5  Derived::rqOp() &
6  Derived::op()
7  ----------------------
8  callF(rValue()); //2
9  f(Base&&)
10 Derived::rqOp() &&
11 Derived::op()
12 ----------------------
13 callF(std::forward<Base&&>(baseRValueReference)); //3
14 f(Base&&)
15 Derived::rqOp() &&
16 Derived::op()
17 ----------------------
18 callFL(w); //4
19 f(Base const&)
20 Derived::rqOp() &
21 Derived::op()
22 ----------------------
23 callFL(rValue()); //5
24 f(Base&&)
25 Derived::rqOp() &&
26 Derived::op()
27 ----------------------
28 callFT(w); //6
29 f(Base const&)
30 Derived::rqOp() &
31 Derived::op()
32 ----------------------
33 callFT(rValue()); //7
34 f(Base&&)
35 Derived::rqOp() &&
36 Derived::op()
37 ----------------------
```

Achtung: Durch eine andere Aufrufsyntax (`std::forward`) wird eine andere Methode oder Funktion gerufen, bzw. ohne diese, nicht die erwartete für ein temporäres Objekt.

Die hier vorgestellte Technik mit `std::forward` wird häufig in der Kombination mit Varidic Templates bei verschiedenen `make_...(Types&&... args)` Funktionen und bei den Container Operationen `container.emplace(Args&&... args)` verwendet um überflüssige Kopien zu vermeiden.

6.32 Funktionen, Operationen und Methoden

Die folgenden Ausführungen gelten sowohl für globale Funktionen als auch für Operationen und Methoden in Klassen wenn nicht explizit auf Operationen Bezug genommen wird. In diesem Absatz wird immer von Funktion gesprochen. Funktionen die `constexpr` deklariert sind siehe section 6.33 auf Seite 140.

6.32.1 Deklaration

Eine Funktion kann nur aufgerufen werden, wenn sie vorher deklariert wurde. Eine Funktionsdeklaration kann aus einer Vielfalt verschiedener specifiers und modifiers bestehen.

Prototyp, Signatur einer Funktion mit prefix return type:

```
[inline] [constexpr] type identifier( parameterliste )[const][throw-spec];
```

Der Typ einer Funktion besteht aus dem return Typ und der Parameterliste.

- `inline`, eine Aufforderung an den Compiler keinen Functioncall zu generieren (section 6.32.6 auf Seite 129)
- type, `auto`, `decltype(auto)` required: Der Typ des Rückgabewertes ist der Typ des Funktionsaufrufes / der Nachricht
  Dieser kann seit C++14 automatisch erkannt werden
- identifier, required: Name der Funktion/Operation
- parameterliste, required: Liste der Typen der Übergabeparameter, Identifier für Parameter sind optional, die Namen der Identifier geben aber einen Hinweis auf die Bedeutung der Parameter und unterstützen damit die Dokumentation
- `constexpr`, die Funktion kann zur compile time evaluiert werden, wenn ihre Argumente ebenfalls `constexpr` sind (section 6.33 auf Seite 140)
- **`static`**, eine linkage specification (section 6.21 auf Seite 62)
- throw-spec: (C++11 deprecated) Schlüsselwort throw(Exceptionlist) Eine leere throw-spec garantiert, dass die Funktion keine Exceptions wirft.
- `noexecpt` (C++11) Anstelle der leeren throw-spec **`throw()`** die nicht zur Compile Time überprüft wird, sollte `noexecpt` verwendet werden. Das ermöglicht dem Compiler Optimierungen durchzuführen. (section 6.34.5 auf Seite 151)

Memberfunktionen können spezifiziert werden als:

- `virtual`, kann in abgeleiteten Klassen überschrieben werden (section 6.32.8 auf Seite 132)
- `override`, muss eine `virtual` Operation der Basisklasse überschreiben (section 6.32.9 auf Seite 133)
- **`final`**, `virtual` Operation kann nicht mehr überschrieben werden (section 6.32.9 auf Seite 133)

- mit einem *reference qualifier* `&` oder `&&` (section 6.32.10 auf Seite 135)
- **`static`**, eine Klassenoperation ohne Objektkontext (kein **`this`** Pointer)
- **`const`**, wird eine Operation mit dem Modifier const versehen, kann das Objekt innerhalb der Methode nicht verändert werden, weil der this Pointer vom Typ T const * const ist (section 6.36 auf Seite 153)

Listing 114: Funktions Deklaration

```
return-type identifier(parameterliste)
parameterliste:
   type [identifier][=defaultargument]
   type [identifier][=defaultargument], parameterliste

// Bsp:
long berechneSumme(int op1, int op2=0);
long berechneSumme(int, int);

// Member einer Klasse:
struct S{
   [[noreturn]]
   virtual inline auto f(unsigned long int const * const) -> void const noexcept;
};
```

Das Attribute noreturn

Das Konstrukt `[[...]]` wird *attribute* genannt und kann überall in der C++ Syntax verwendet werden. Durch ein attribute wird eine Eigenschaft für das nachfolgende syntaktische Element definiert. Es gibt nur zwei standard attribute, `[[noreturn ]]` und `[[carries_dependency]]`. Die genaue Bedeutung ist Implementierungs abhängig. Am Anfang einer Funktionsdeklaration bedeutet `[[noreturn]]`, dass die Funktion nicht zurückkehrt (z.B. `[[noreturn]] void exit(int)`). Dieser Hinweis ist für das Verständnis und die Codeerzeugung hilfreich.
Kehrt eine so deklarierte Funktion wider Erwarten zurück, ist das Systemverhalten nicht definiert.

Default Argumente

Defaultargumente müssen bei der Übersetzung bekannt sein. Wird für einen Parameter ein Defaultargument festgelegt, müssen für alle rechts folgenden Parameter ebenfalls Defaultargumente festgelegt werden.

Defaultargumente werden zur Übersetzungszeit statisch gebunden, virtuelle Operationen aber zur Laufzeit. Darum sollten in abgeleiteten Klassen die Defaultargumente nie überschrieben werden [178].

6.32.2 Alternative Deklarationssyntax

Prototyp, Signatur einer Funktion mit suffix return type:
`auto identifier( parameterliste )->type [const][throw-spec];`

[178] [Mey06a] Item 37 Never redefine a function´s inherited default parameter value

In manchen Situationen hängt der Typ einer Funktion von einem Ausdruck ab, an dem die Funktionsargumente beteiligt sind. In Listing 115 ist ein Beispiel dafür abgebildet.

Listing 115: Alternative Deklarationssyntax für Funktionen

```
1 template<typename T1, typename T2>
2 decltype(t1+t2) add(T1 t1, T2 t2); // error t1, t2 sind in diesem scope nicht
      deklariert
3
4 template<typename T1, typename T2>
5 auto add(T1 t1, T2 t2) -> decltype(t1+t2);
```

In der ersten Deklaration sind die Namen `t1, t2` noch nicht bekannt, mit der zweiten Form wird eine Deklaration in Abhängigkeit des Typs des Ausdrucks `t1+t2` möglich. Das Schlüsselwort `auto` hat diesem Zusammenhang nichts mit type-deduction[179] zu tun, sondern ist lediglich eine syntaktische Notwendigkeit, da der Typ durch den suffix return type angegeben ist.

6.32.3 Definition

Eine Funktion, die aufgerufen wird muss irgendwo genau einmal definiert sein. Eine Funktionsdefinition ist eine Funktionsdeklaration, bei der der Funktionskörper angegeben wird. Der **Funktionskopf** muss mit dem **Prototyp** genau übereinstimmen.

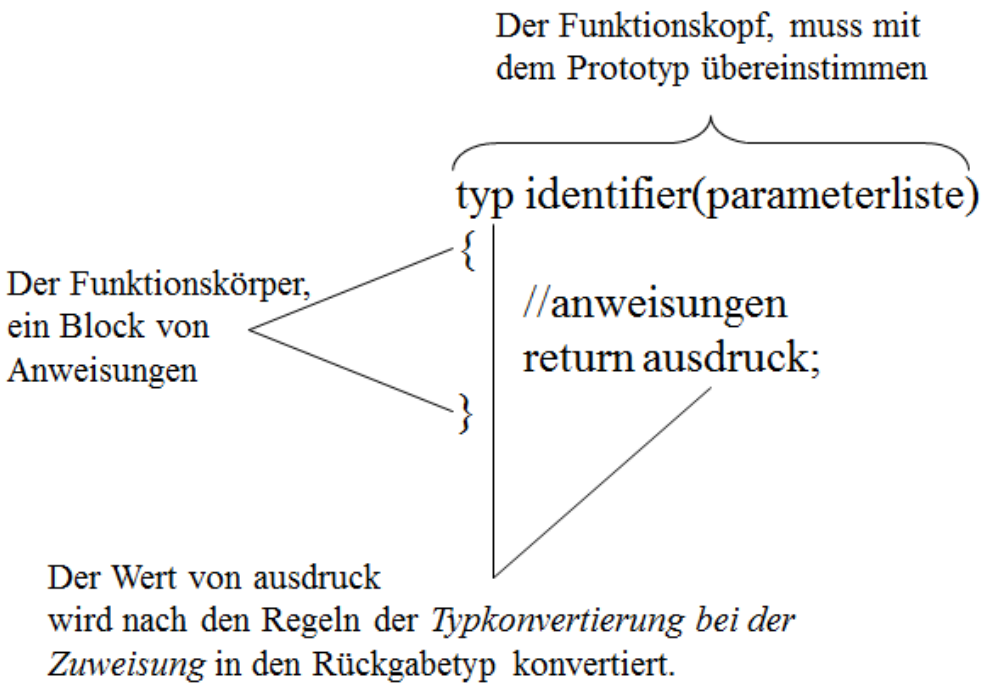

Abbildung 9: Funktionsdefinition

[179] section 6.30.3 auf Seite 117

Parameterliste: Liste der Typen der Übergabeparameter. Die Identifier gewähren den Zugriff auf die übergebenen Argumente, sie werden beim Aufruf mit den **Werten** der korrespondierenden Argumente **initialisiert.** Dabei kommen die Regeln der **Typkonvertierung** bei der **Zuweisung** zum tragen.

Listing 116: Funktions Definition

```
1 long berechneSumme(int op1, int op2=0) {
2    return static_cast<long>(op1) + op2;
3 }
```

6.32.4 Das Schlüsselwort `this`

Innerhalb einer Methode steht über das Schlüsselwort **`this`** die Adresse des aktuellen Objekts zur Verfügung. **`this`** hat den Typ `T * const this`, mit `T` als die Klasse zu der die Methode gehört.

6.32.5 Funktionsaufruf

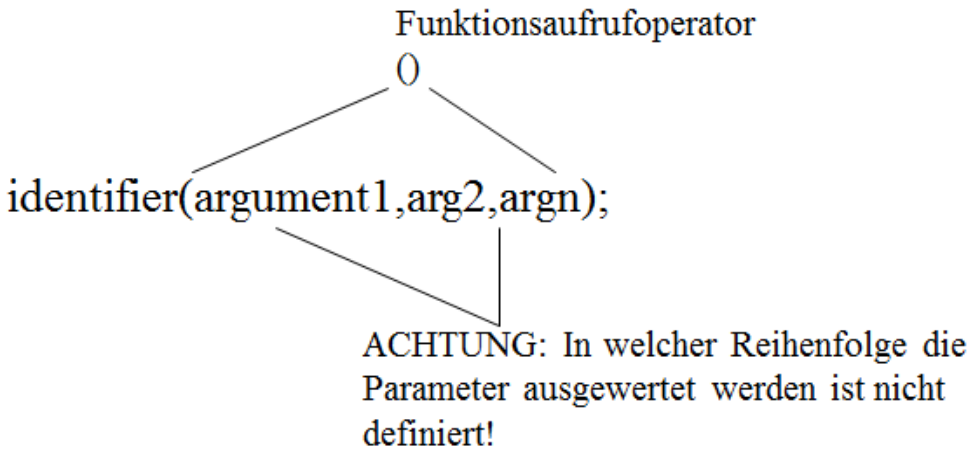

Abbildung 10: Funktionsaufruf

Eine Funktion kann nicht aufgerufen werden, solange sie nicht vorher deklariert wurde [180].

Ein **Funktionsaufruf ist ein Ausdruck** vom Typ des Rückgabewertes der Funktion und hat den Wert, den die Funktion zurückliefert.

```
long result = berechneSumme(42, 43);
```

Der **Name der Funktion ist ein Ausdruck** vom Typ Referenz auf eine Funktion mit diesem Prototyp, der implizit in einen Zeiger auf eine Funktion mit diesem Prototyp konvertiert wird und hat den Wert **Adresse der Funktion.** Diese **implizite Konvertierung von Funktionsnamen** (und Arraynamen) wird in der Fachliteratur

[180] [Str00a] Kapitel 7.1 Funktionsdeklarationen

häufig auch **decay** genannt[181] (Abfall en decay). Der eigentliche Typ (Reference) verfällt zu einem Pointer.

Bsp.:

`long (*pLongFunction)(int, int);` Deklaration eines Objekts mit dem Namen pLongFunction vom Typ: Zeiger auf eine Funktion die `long` zurück gibt und zwei `int` Objekte als Argumente erwartet.

```
pLongFunction = &berechneSumme; //decay bei der Zuweisung

pLongFunction(5, 3); //Aufruf von berechne Summe via Funktionszeiger
```

op1 und **op2** werden mit den Werten 5 und 3 **initialisiert**.

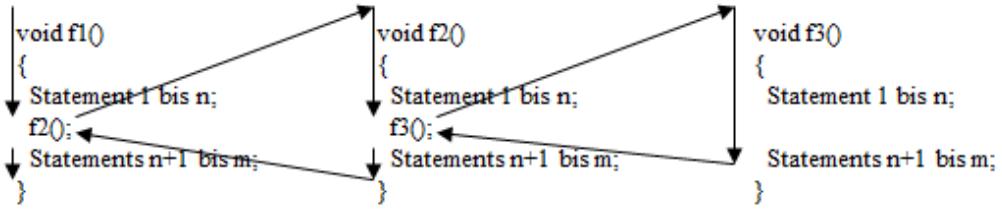

Abbildung 11: Programmablauf beim Aufruf mehrerer Funktionen

Ein Funktionsaufruf verzweigt an die Stelle, wo der Code der Funktion liegt und kehrt an die Stelle wo sie gerufen wurde zurück. Die Rücksprung Adresse wird auf dem Stack verwaltet. siehe section 6.34 auf Seite 143.

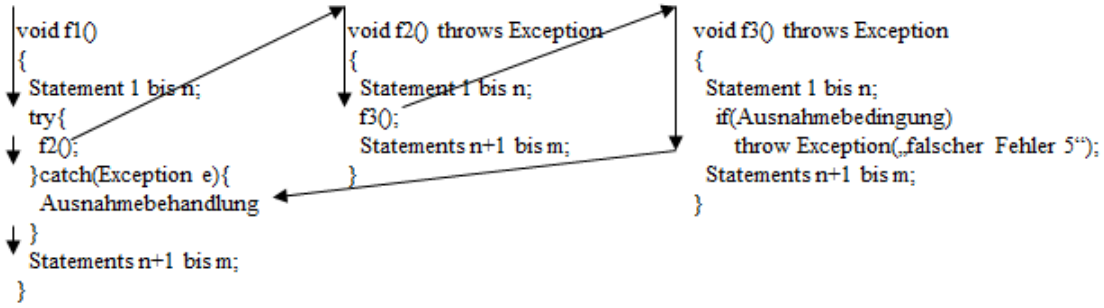

Abbildung 12: Programmablauf im Falle einer Ausnahme (Exception)

6.32.6 Das Schlüsselwort inline

Funktionen können `inline` definiert werden. Das Schlüsselwort `inline` ist ein Hinweis an den Compiler, den Funktionsrumpf anstelle des Funktionsaufrufes in den Code einzufügen, dadurch wird der Runtime-Overhead für den Funktionsaufruf gespart, der Binärcode wird unter bestimmten Umständen allerdings größer. Damit eine Funktion `inline` behandelt werden kann, muss der Funktionskörper bei der Übersetzung bekannt sein. **Methoden** die **in der Klassendefinition** enthalten sind, **sind `inline` ohne Angabe des Schlüsselworts**.

[181] [VJ03] 22.2 Pointers and References to Functions

Listing 117: Inline Funktionen

```
1  class A{
2     void f1() { /*methodenkörper*/ } //implizit inline
3     void f2();
4  };
5
6  // out of class Memberdefinition im Header explizit inline
7  inline
8  void A::f1() { /*methodenkörper*/ }
9
10 // globale inline funktion
11 inline
12 void f2() { /*funktionskörper*/ };
```

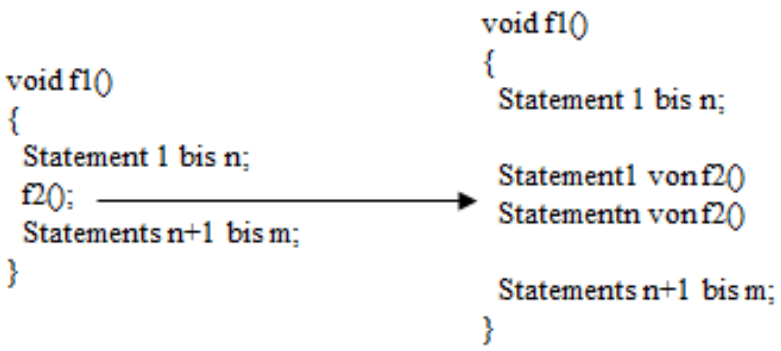

Abbildung 13: Programmablauf beim Aufruf von inline Funktionen

Der Compiler fügt den Code aus der Inline Funktion dort ein, wo sie aufgerufen wird. Der Scope der lokalen Variablen, der aufgerufenen Funktion (`f2`) bleibt dabei erhalten.

Wird eine Methode oder eine Funktion im Header außerhalb der Klassendefinition definiert, muss sie mit dem Schlüsselwort inline gekennzeichnet werden.

Methoden die nicht inline deklariert sind, müssen in einer eigenen Übersetzungseinheit definiert werden, weil es sonst mehrere Funktionsdefinitionen gibt, wenn der Header in verschiedenen Übersetzungseinheiten includiert wird, was zu einem Fehler beim Linken des Programms führt.

Auf der Basis von Templates und inline Funktionen lassen sich hervorragend Abstraktionen bilden, bei gleichzeitig höchster Effizienz des Compilats, da die Abstraktionen vom Compiler beim übersetzten eliminiert werden. Es werden also keine virtuellen Operationen benötigt, es werden keine *jumps* aus den Funktionsaufrufen, die im Sourcecode vorhanden sind, generiert.

6.32.7 Function Overload Resolution und Template Argument Deduction [182]

Funktionen und Funktionstemplates können überladen werden. Es kann also mehrere Funktion und gleichzeitig mehrere Funktionstemplates geben, die denselben Namen tragen, aber unterschiedliche Parameterlisten haben. Die Parameterlisten können sich in der Anzahl der Parameter und in den Typen der Parameter unterscheiden. Die Anwendung der Auflösungsregeln beim Überladen (**Overload Resolution)** ist nur ein Teil eines Funktionsaufrufs. Abstrakt könnte der Aufruf einer Funktion mit ihrem Namen wie folgt beschrieben werden:

- Alle Funktionen mit dem Namen werden gesucht und eine erste Menge der in Frage kommenden Funktionen und Funktionstemplates (`overload set`) gebildet.
- Wenn notwendig wird die Menge bearbeitet, z.B.: template argument deduction
- Alle Funktionen die nicht passen, werden entfernt (nach dem implizite Typkonvertierungen und default Argumente überprüft wurden). Das ist die Menge der brauchbaren Funktionen (`viable function candidates`).
- **Overload Resolution** wird durchgeführt um die Funktion deren Prototyp am besten passt (`best candidate`) zu finden. Gibt es eine, wird sie ausgewählt, gibt es mehrere, ist der Funktionsaufruf mehrdeutig (`ambiguous`).
- Die ausgewählte Funktion wird überprüft. Z.B.: wenn sie nicht im Zugriff ist (**`private, protected`**) wird eine Fehlermeldung ausgegeben.

Jeder dieser Schritte ist auf seine Weise hinlänglich komplex, wobei Overload Resolution der schwierigste Schritt ist.

Argument Deduction: Wird nur **bei Funktionstemplates** durchgeführt. Wird ein Funktionstemplate mit bestimmten **call Argumenten** aufrufen, werden die **template Argumente** durch die Typen der call Argumente vom Compiler bestimmt. Implizite (automatische) **Typkonvertierungen werden dabei nicht durchgeführt**! Jeder template Parameter muss exakt übereinstimmen (`exact match`). Bsp.:

Listing 118: Call Argument exact match

```
1 template<typename T>
2 inline T const& max(T const& a, T const& b);
3 max(4, 7); // ok : T ist int für beide Argumente
4 max(4, 4.2); // Error: das erste T ist int, das zweite ist double
```

Wenn der Aufruf mit unterschiedlichen Typen erfolgen soll,

1. können die call Argumente explizit gecastet werden
   `max(static_cast<double>(4), 4.2);`
2. oder T explizit spezifiziert werden
   `max<double>(4, 4.2);`

[182] [VJ03] Appendix B und 2.2 Argument Deduction

3. oder das Template mit unterschiedlichen Parametern ausgestattet werden
   ```
   template<typename T1, typename T2>
   T1 const& max(T1 const& a, T2 const& b);
   ```
   dabei stellt sich aber die Frage, welchen Typ hat der Rückgabewert der Funktion (T1 oder T2)?

Soll ausschließlich das Funktionstemplate ausgewählt werden, kann der Typ in den spitzen Klammern weggelassen werden:

```
int i = max<>(3, 4);
```

6.32.8 Das Schlüsselwort `virtual`

Mit dem Schlüsselwort `virtual` lassen sich polymorphe Operationen definieren. Siehe section ?? auf Seite ??, section 6.15 auf Seite 49, section 6.26 auf Seite 103.

Eine typische Realisierung polymorpher Operationen erfolgt auf der Basis sogenannter virtueller Funktionstabellen (virtual function table vtbl). Das hier beschriebene ist nicht C++ Standard und dient der Veranschaulichung. Die meisten Compiler verwenden eine ähnliche Art der Realisierung!
Das Diagramm 4 auf der nächsten Seite zeigt auf der linken Seite eine Klasse Base mit den virtuellen Operationen op1() und op2() und einer nicht virtuellen Methode mf().
Darunter eine Klasse Derived, die von Base abgeleitet ist, die Operation op1() redefiniert (überschreibt) und eine weitere virtuelle Operation op3() definiert.
In der Mitte sind die Funktionstabellen mit den symbolischen Adressen der Operationen (z.B. &op1) und rechts die dazugehörigen Methoden der beiden Klassen.

Wenn eine Klasse virtuelle Operationen hat, wird für sie eine Funktionstabelle erzeugt. In diese Tabelle werden alle Adressen der Operationen eingetragen, die mit dem Schlüsselwort `virtual` deklariert werden.
Der Index der Operation in der Tabelle entspricht der Position der Operation im Header. Eine nachträgliche Änderung der Reihenfolge der Operationen im Header einer Klasse kann daher fatale Folgen haben, die Klasse muss bevor sie benutzt wird, neu übersetzt werden!

Wird von einer solchen Klasse abgeleitet, wird für die abgeleitete Klasse (Derived) eine Kopie der Funktionstabelle angelegt. Die Einträge der Methoden, die in der abgeleiteten Klasse redefiniert werden, werden durch die Adressen der Methoden der abgeleiteten Klasse überschrieben.

Objekte dieser Klassen benötigen `sizeof(void*)` mehr Platz, als es sich aus der Summe der in den Klassen deklarierten Attribute ergibt, weil in jedem Objekt die Adresse der Funktionstabelle gespeichert wird.

Das Diagramm 5 auf Seite 134 zeigt auf der linken Seite jeweils ein Objekt von Base (b) und Derived (d) und symbolisch die Speicherstruktur der Objekte. In der Mitte sind die Funktionstabellen mit den Adressen der Methoden und rechts die Methoden auf die die Zeiger zeigen.

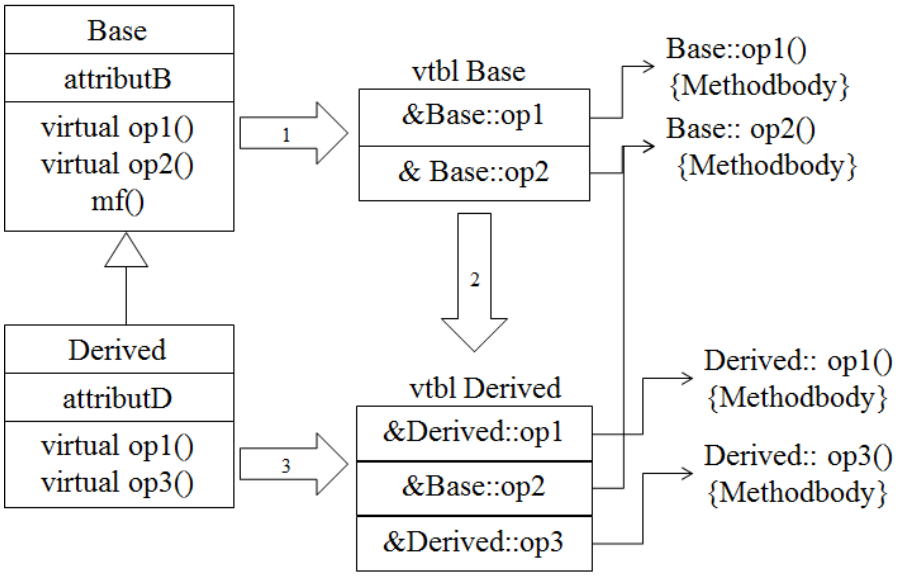

Diagramm 4: Polymorphe Operationen und virtual

Bei der Konstruktion der Objekte wird zuerst der Base Anteil und anschließend der Derived Anteil initialisiert und umgekehrt, bei der Zerstörung der Objekte wird zuerst der Derived Anteil und anschließend der Base Anteil zerstört. Vor dem Konstruktor der Base Klasse wird dem Zeiger auf die vtbl des Objekts, die Adresse der vtbl der Base Klasse zugewiesen und vor dem Konstruktor der Derived Klasse, die Adresse der vtbl der Derived Klasse. Bei der Zerstörung der Objekte wird jeweils vor dem Aufruf des Destruktors, die Adresse der jeweiligen Funktionstabelle in den Speicherbereich des Objekts kopiert.
Das erklärt warum in Konstruktoren und Destruktoren nicht die speziellsten polymorphen Methoden, sondern nur die Methoden der jeweiligen Klasse gerufen werden und warum Objekte polymorpher Klassen nicht in den ROM Bereich geladen werden können, auch wenn sie **`const`** deklariert sind.

An der Stelle eines Funktionsaufrufs (obj.op1()), wird vom Compiler code generiert, der die Adresse der Funktionstabelle aus dem Objekt ermittelt, über den Index der Operation die Adresse der Methode aus der Funktionstabelle liest und anschließend die Methode über ihre Adresse aufruft.
In einer Funktion **`void`** `f(Base& obj){ obj.op1()}` wird dadurch immer die zum jeweiligen Objekt passende Methode ausgewählt.
Dieser Mechanismus benötigt immer dieselbe Anzahl Verarbeitungsschritte, unabhängig von der Tiefe der Vererbungshierarchie. Das Diagramm 5 auf der nächsten Seite zeigt die beteiligten Datenstrukturen.

6.32.9 Die Identifier final und override C++11

`override` und **`final`** sind zwei Bezeichner die eine spezielle Bedeutung im Kontext von `virtual` Operationen haben.

Mit dem Identifier `override` sollte eine Methode gekennzeichnet werden, die eine Methode aus einer Basisklasse überschreibt. Der Compiler hat dadurch die

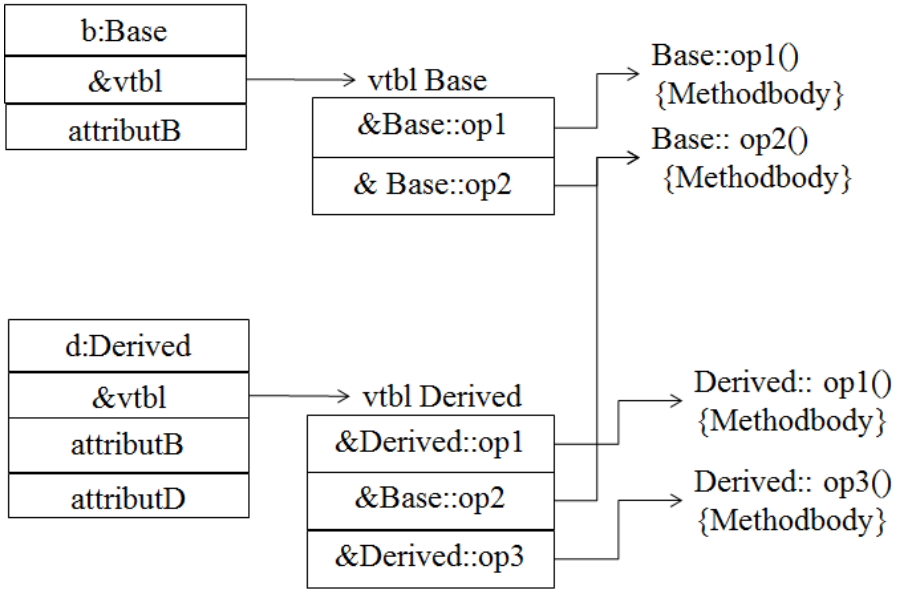

Diagramm 5: Aufruf virtual Function

Möglichkeit einen Hinweis auszugeben, wenn diese Methode nicht mit der Signatur einer `virtual` Methode der Basisklasse übereinstimmt. Die Signatur beinhaltet hier den Rückgabetyp, den Namen der Operation, die Parameterliste, der Modifier **`const`** und die reference Qualifier[183]

Mit dem Identifier **`final`** können leider nur `virtual` deklarierte Operationen als nicht mehr weiter überschreibbar gekennzeichnet werden.

In Listing 119 sind die verschiedenen Möglichkeiten und Fehlermeldungen dargestellt. Wenn `Op5` nachträglich in der Klasse `Base` eingefügt wird, nachdem diese bereits in `Derived virtual` existiert, wird das vom Compiler nicht bemängelt.

Listing 119: Keyword override und final

```
1  class Base{
2  public:
3     virtual void Op1() {}
4     virtual void Op2() const {}
5     virtual void Op3() final {}
6
7     //mit reference-qualifier überladen
8     virtual void Operation() &; // wird angewendet wenn *this ein lvalue ist
9     virtual void Operation() &&; // wird angewendet wenn *this ein rvalue ist
10
11    void nonvirtualOp() {}
12
13    //error:'void Base::nonvirtualOp()' marked final, but is not virtual
14    //void nonvirtualOp() final {}
15
16    // error: 'virtual void Base::Op4()' marked override, but does not override
17    //virtual void Op4() override {}
18
```

[183] section 6.32.10 auf der nächsten Seite

```
19    void Op5() {}
20 };
21 class Derived : public Base{
22    void Op1() override final {}
23    //error: 'virtual void Derived::Op2()' marked override,
24    //but does not override => non const
25    //virtual void Op2() override {}
26
27    void Op2() const override {} // ok
28
29    //error: 'void Derived::nonvirtualOp()' marked override, but does not override
30    //void nonvirtualOp() override {}
31
32
33    //error: overriding final function 'virtual void Base::Op3()
34    //void Op3() {}
35
36    virtual void Op5() final {}
37 };
38 class D2 : public Derived{
39    //error: overriding final function 'virtual void Derived::Op1()
40    //virtual void Op1() override {}
41 };
```

Von Klassen die mit **final** gekennzeichnet sind, kann nicht abgeleitet werden: **class** FinalClass **final** { /*Definition */};.

6.32.10 Reference Qualified Operations

Mit einer Operation, die mit einem Reference Qualifier (Operation()&) überladen ist, kann unterschiedliches Verhalten für R- und L-Value Ausdrücke definiert werden. Ein Beispiel ist in section 6.31 auf Seite 121 zu finden[184]. Wird eine Operation mit einem Reference Qualifier definiert, kann es keine nicht Reference qualifizierte überladene Operation mit demselben Namen geben. Wird eine Operation nur für l-values definiert, kann sie nicht auf r-values, also auf temporäre Objekte angewandt werden und umgekehrt.

Listing 120: Reference Qualified Operations

```
1  class Widget{
2     ...
3     void Operation() &; // for l-values
4     void Operation() &&; // for r-values
5
6     // error: 'void Operation()' cannot be overloaded
7     // with 'void Operation() &'
8     void Operation(); // error
9     ...
10 };
```

[184] siehe auch section 6.32.9 auf Seite 133

Um eine RValue reference qualified Operation aufzurufen, wird mit einer Ausnahme immer `std::forward<...>`[185] benötigt[186].
Die Ausnahme ist `Derived().Operation()`. In diesem Fall ist `Derived()` ein RValue Objekt.

6.32.11 Lambda Expression C++11/14

Lambda Expressions oder kurz Lambdas[187] ermöglichen die kompakte Definition von lokalen Funktionsobjekten, soganannte Closures, die als Argumente übergeben oder als lokale Objekte verwaltet werden können.

Begriffe im Zusammenhang mit Lambdas:

lambda expression ist ein Ausdruck! Z.B. der fettgedruckte dritte Parameter von `find_if`:

```
find_if(container.begin(), container.end(),
[](int val){ return 0 < val && val < 10; }
);
```

ist ein Ausdruck der zu einem closure evaluiert, das ist eine häufige Verwendungsart für Lambdas.

ein closure ist das run-time Objekt das durch das Lambda erzeugt wird. Abhängig vom capture mode enthält das Objekt Werte von oder Referenzen auf andere Objekte

eine closure class ist eine Klasse, von der das closure Objekt erzeugt wird. Jede Lambda Expression veranlasst den Compiler dazu, ein unikat einer Klasse zu erzeugen. Der Funktionskörper des Lambdas ist die Methode des `return-type operator()(...)` dieser Klasse.

Die Syntax ist
entweder: `[...] (...)mutable throwSpec ->retType {...}`
mit Parameterliste, wobei `mutable throwSpec ->retType` jeweils optional sind.
oder: `[...] {...}`
ohne Parameterliste.

Im einzelnen sind das:

- *lambda introducer*: Die eckigen Klammern `[...]` am Anfang. Hier können die Namen von lokalen Variablen angegeben werden (Captures) auf die innerhalb des Funktionskörpers Zugriff benötigt wird.
  - `[]`, leere eckige Klammern: auf keine der lokalen Variablen kann im Funktionskörper zugegriffen werden
  - `[=]` default capture mode **„by value“**: auf die Werte aller lokalen Variable kann im Funktionskörper zugegriffen werden.
  - `[&]` default capture mode **„by reference“**: auf alle lokalen Variable kann im Funktionskörper zugegriffen werden

[185] [Mey15] Item 23
[186] http://stackoverflow.com/questions/35246947/makes-it-any-sense-to-declare-rvalue-methods-e-g-void-operati
[187] [Jos12] und [Gri12]

  - `[a, &b]` a „by value“, b „by reference“, auf alle anderen besteht kein Zugriff
  - Wird das Lambda innerhalb einer Klasse definiert, besteht mit den default capture modes `[=]`, `[&]` Zugriff auf **`this`**, Attribute des Objekts sind dadurch innerhalb des Funktionskörpers sichtbar.
    Achtung: Beim Zugriff auf die lokalen Objekte und auf Attribute des Objekts muss auf die Lebensdauer z.B. von **`this`** geachtet werden[188].
  - Alle globalen und statischen Objekte, deren Namen an der Stelle sichtbar sind, an der das Lambda definiert wird, können innerhalb des Funktionskörpers referenziert werden.

- `(parameterliste)`: wie bei einer normalen Funktion, in runden Klammern
- `mutable`, wenn innerhalb des Funktionskörpers die Variablen, die „by value“ bei den Captures angegeben sind, änderbar sein sollen
- Der Rückgabetyp wird vom Ausdruck des return Statements abgeleitet. Dabei kommen dieselben Regeln zur Anwendung, wie bei Funktionen mit dem return-type `auto`.
- Funktionskörper der Lambda, ein Block `{...}` von Anweisungen

Der Typ einer Lambda

ist ein anonymes Funktionsobjekt, das für jede Lambda einzigartig ist. Um Objekte dieses Types zu deklarieren wird entweder ein template benötigt oder das Schlüsselwort `auto`.

In Listing 121 ist die gleiche Funktionalität zur Anschauung was eine Lambda ist, einmal mit einer Lambda und mit einem Functor realisiert. Die Werte der „by value“ übergebenen Variablen werden in der Lambda gespeichert, wie in der Membervariablen `inc` im Functor `AccumTemp`. Die Lambda entspricht dem `operator ()(int)const`. Daher dürfen die Member innerhalb nicht verändert werden. Verwirrend ist hier die Möglichkeit, über den Member **`int`** `& sum` das gleichnamige referenzierte lokale Objekt **`int`** `sum` zu verändern.

Listing 121: Beispiele für Lambdas

```
1  void demoAccumLambda(){
2     class AccumTemp{
3        int & sum;
4        int inc;
5     public:
6        AccumTemp(int& sum, int inc):sum(sum), inc(inc){}
7
8        int operator()(int v) const {
9           return sum += v + inc;
10       }
11    };
12    vector<int> intVec{1, 2, 3, 4};
13    int sum = 0;
```

[188] [Mey15] Item 31 Avoid default capture modes

```
14    int inc = 2;
15
16    for_each(intVec.begin(), intVec.end(),
17          [&sum, inc](int v) { return sum += v +inc; }
18    );
19    cout << "sum: " << sum << endl;
20
21    sum = 0;
22    for_each(intVec.begin(), intVec.end(),
23          AccumTemp(sum, inc)
24    );
25    cout << "sum: " << sum << endl; // sum: 18
26 }
```

Listing 122: Lambdas speichern die Werte

```
1  void demoLambdaMemorize(){
2     vector<int> intVec{1, 2, 3, 4};
3
4     string separator = "-";
5     auto sepHyphen = [separator](int i)
6        {cout << i << separator;};
7
8     separator = ", ";
9     auto sepComma = [separator](int i)
10       {cout << i << separator;};
11
12    for_each(intVec.begin(), intVec.end(),sepHyphen);
13    cout << endl;
14    for_each(intVec.begin(), intVec.end(),sepComma);
15    cout << endl;
16 }
17 Ausgabe:
18 1-2-3-4-
19 1, 2, 3, 4,
```

Move Semantik mit init capture[189]

Manchmal wird move Semantik anstatt by-value oder by-reference capture benötigt. Z.B. wenn ein `std::unique_ptr` oder ein `std::future` in der closure verwaltet werden soll.
Ab C++14 steht dafür eine spezielle Syntax zur Verfügung wie sie in Listing 123 beschrieben ist.

Listing 123: Move Semantik mit init capture

```
1  class Widget{
2     ...
3     bool isValid() const;
4     bool isArchived() const;
5     ...
```

[189][Mey15] Item 32: Use init capture to move objects into closures.

```
6     calculatecSomething()
7 };
8
9 void f(){
10    auto pw = std::make_unique<Widget>(); // erzeugt ein Widget auf dem Heap
11    ...
12    pw->calculatecSomething();
13    auto isValidAndArchivedL =
14       [pw = std::move(pw)]{ return pw->isValid() && pw->isArchived();};
15    ...
16    if(isValidAndArchivedL()) ...
17 }
```

Das Capture `[pw = std::move(pw)]` initialisiert die Variable `pw` des closures.

Links von der Initialisierung (=) stehen der Name des Data Members (hier: `pw`) der closure class, die definiert und initialisiert werden, rechts davon ein Ausdruck, mit dessen Wert er initialisiert wird.
Das `pw` in `std::move(pw)` ist das lokale der Funktion `f()`.

Mit einem init capture ist es möglich,

1. den Namen eines Members der closure class und
2. einen Ausdruck zur Initialisierung desselben

zu spezifizieren.

Steht diese Syntax noch nicht zur Verfügung, hilft eine konventionelle Functor Klasse, wie sie in Listing 124 beschrieben ist oder `std::bind()`. Ein Konstruktor nimmt eine R-Value Referenz auf das Objekt entgegen und moved es in eine Membervariable. Die `operator()(...)` Funktion nutzt den Member.

Listing 124: Lambdas und Move vor C++14

```
1  //Definition der Functor Klasse
2  struct IsValidAndArchived{ // Functor Klasse
3     using DataType = std::unique_ptr<Widget>;
4     explicit IsValidAndArchived(DataType&& ptr) // ctor mit R-Value Referenz
5     : pw(std::move(ptr)){}
6     bool operator()() const { return pw->isValid() && pw->isArchived();}
7
8  private:
9     DataType pw;
10 }
11 void f(){
12    auto pw = std::make_unique<Widget>();
13    ...
14    // Objekt erzeugen
15    IsValidAndArchived isValidAndArchivedF(std::move(pw)); // ctor
16    ...
17    // Anwenden
18    if(isValidAndArchivedF()) ... // operator() F Functor
19
```

```
20    // mit std::bind
21    auto isValidAndArchivedB = std::bind( // B Bind
22       [](std::unique_ptr<Widget> const & pw) // C++11 lambda
23       { return pw->isValid() && pw->isArchived();},
24       std::move(pw)
25    );
26    ...
27    if(isValidAndArchivedB()) ...
28 }
```

Sowohl das Objekt der Functor Klasse `isValidAndArchivedF` als auch das mit `std::bind` erzeugte Objekt `isValidAndArchivedB` speichern `pw`. Letzteres übergibt dem Closure beim Aufruf das zweite Argument. Der Parameter des Closures ist **`const`**`&` deklariert, damit wird dieselbe Semantik wie bei einem Closure, das **nicht** `mutable` deklariert ist, erreicht.

6.33 Das Schlüsselwort constexpr C++11

Mit dem Schlüsselwort `constexpr` gekennzeichnete Identifier und Funktonsdefinitionen können zur **compile time** ausgewertet werden. Sie sind also **compile-time constant**! Im Gegensatz zu Deklarationen mit dem Schlüsselwort **`const`**[190].

`constexpr` deklarierte Identifier können nicht `extern` deklariert werden. Ihre Definition muss bei ihrer Verwendung in der Übersetzungseinheit bekannt sein. Solche Funktionen sind implizit `inline`. Die wichtigste Anwendung von einzelnen Konstanten ist die Vermeidung von **magic numbers**, die Verwendung von Literalen (42) im Code. **`const`** deklarierte Symbole werden zur Spezifikation von Schnittstellen benötigt.

Parameter von Funktionen können nicht `constexpr` deklariert werden.
`int f(constexpr int i){};//error: a parameter cannot be declared 'constexpr'`

Listing 125: Das Schlüsselwort constexpr

```
1  constexpr int global{42};
2  //error: declaration of constexpr variable 'globalExtern' is not a definition
3  //extern constexpr int globalExtern;
4  extern int nonConstGlobal;
5
6  //warning: inline function 'constexpr int f()' used but never defined
7  constexpr int f();
8
9  // constexpr Funktionsdefinition
10 constexpr int square(int x){ return x*x; }
11
12 constexpr int globalAccess(int const& i){
13    return global + i;
14 }
```

[190] siehe section 6.36 auf Seite 153

```
15 void f(int n){
16    int nonConstlocal{1};
17    constexpr int constLocal{1};
18    //error: uninitialized const 'i1'
19    //constexpr int i1;
20    // error: 'n' is not a constant expression
21    //constexpr int i1 = n;
22
23    int square5 = square(5); // wird möglicherweise zur compile time ausgewertet
24    constexpr int square2 = square(2); // wird sicher zur compile time ausgewertet
25    int squareN = square(n); // wird zur runtime ausgewertet, n ist eine Variable
26    float fa[square(9)]; // fa mit 81 Elementen
27
28    constexpr int i2 = globalAccess(constLocal);
29
30    // error: the value of 'nonConstlocal' is not usable in a constant expression
31    //constexpr int i3 = globalAccess(nonConstlocal);
32
33    // OK: runtime evaluation
34    cout << globalAccess(nonConstlocal) << endl;
35
36    // error: 'constexpr int f()' used before its definition
37    //constexpr int i4 = f();
38
39    // error: the value of 'nonConstGlobal' is not usable in a constant expression
40    //constexpr int i5 = globalAccess(nonConstGlobal);
41
42    // OK: runtime evaluation
43    cout << globalAccess(nonConstGlobal) << endl;
44 }
```

Allgemein gilt: Funktionen können nicht zur compile time ausgewertet werden und daher kann ihr Aufruf nicht an Stellen verwendet werden, wo compile time Konstanten erwartet werden, z.B. als Größenangaben für Arrays[191].

Der Funktionsaufruf `square(9)` in Listing 125 auf der vorherigen Seite kann aber zur compile time ausgewertet werden, weil das Argument ebenfalls eine `constexpr` ist (die Konstante 9) und an dieser Stelle, als Größenangabe für das Array `fa`, muss `square(9)` auch zur compile time ausgewertet werden.

Damit ist es möglich, einen Ausdruck wie z.B. `std::numeric_limits<short>::max()` als integrale Konstante zu verwenden was die Verwendung solcher Konstrukte in der Metaprogrammierung eröffnet.

Im Gegensatz dazu führt die Deklaration in Listing 126 auf der nächsten Seite mit **`const`** zu einer Konstante die erst zur Laufzeit initialisiert werden muss[192].

[191] siehe section 6.28.1 auf Seite 107
[192] siehe section 6.36 auf Seite 153

Listing 126: const vs. constexpr

```
1 void f(int i){
2    // ci wird zur Laufzeit definiert und kann danach nicht verändert werden
3    const int ci = i;
4    //constexpr int cei = ci; //  error: the value of 'ci' is not usable in a
         constant expression
5    const int ci2 = 5;
6    constexpr int cei2 = ci2; // OK: ci2 ist mit einer Konstanten initialisiert
7 }
```

Die Konstante `cei` in Listing 126 kann nicht mit einem **`const`** deklarierten **L-Value Ausdruck** initialisiert werden, der mit einer Variablen initialisiert ist.

6.33.1 Einschränkungen von `constexpr` Funktionen (C++11)[193]

Mit C++14 wurden die folgenden Einschränkungen gelockert![194] außerdem werden `constexpr` deklarierte Memberfunktionen nicht mehr als **`const`** angenommen[195].

Funktionen, die zur compile time ausgewertet werden sollen (Schlüsselwort `constexpr`) dürfen nur ein Statement enthalten: **`return`** `expression;`.

Sie unterliegen daher folgenden Einschränkungen:

- dürfen keine lokalen Variablen haben
- dürfen keine Kontrollstrukturen enthalten (Verzweigungen, Wiederholungen) aber conditional expressions und Rekursion ist erlaubt
- dürfen keine Seiteneffekte haben
- dürfen nur auf globale `constexpr` deklarierte Objekte zugreifen die im Scope sind (nicht `extern`)
- dürfen nur ein return Statement enthalten
- dürfen nicht **`void`** als Rückgabe Typ haben

Die Regeln für `constexpr` Konstruktoren[196] sind entsprechend. Es sind nur einfache Initialisierungen der Member erlaubt.

Genauso wie `inline` Funktionen unterliegen `constexpr` Funktionen der „One Definition Rule (ODR)“, die Definition in verschiedenen Übersetzungseinheiten muss übereinstimmen. Man kann `constexpr` Funktionen als `inline` Funktionen betrachten, die stärkeren Einschränkungen unterliegen.

Allerdings sind Rekursionen und Conditonal Expressions[197] in `constexpr` Funktionen erlaubt. Damit kann fast alles berechnet werden, aber die daraus resultierende Dauer der Übersetzung wird entsprechend länger und debugging wird

[193] [Str13] S.311 12.1.6 constexpr Functions
[194] [Mey15] Item 15, S.100
[195] https://akrzemi1.wordpress.com/2013/06/20/constexpr-function-is-not-const/
[196] siehe section 6.12 auf Seite 46
[197] section 6.25.16 auf Seite 102

erschwert. Verwenden Sie `constexpr` Funktionen nur für einfache Berechnungen, wofür Sie auch gedacht sind.

6.33.2 Conditional Evaluation

In einer `constexpr` Funktion wird der Zweig einer Conditional Expression, der nicht ausgewählt wird auch nicht ausgewertet. Daraus ergibt sich, dass für diesen Zweig eine runtime Auswertung / Evaluation erforderlich sein kann. Je nach dem welcher Zweig ausgewählt wird, wird die Funktion entweder zur compile time oder zur runtime ausgewertet.

Listing 127: Ein Zweig einer Conditional Expression erfordert runtime Evaluation

```
1 constexpr int low = 0;
2 constexpr int high = 99;
3 constexpr int check(int i){
4    return (low <= i && i < high) ? i : throw out_of_range();
5 }
```

6.34 Exceptions[198]

Die C++ Ausnahmebehandlung / Exceptionhandling ermöglicht den Umgang mit Ausnahmen ohne dazu das Interface der Funktionen (Rückgabewert und Argumente) zu überfrachten. Exceptions bieten einen alternativen Ausstieg zu **`return`** aus einer Funktionsaufrufhierarchie.

6.34.1 Programmablauf im Falle einer Ausnahme

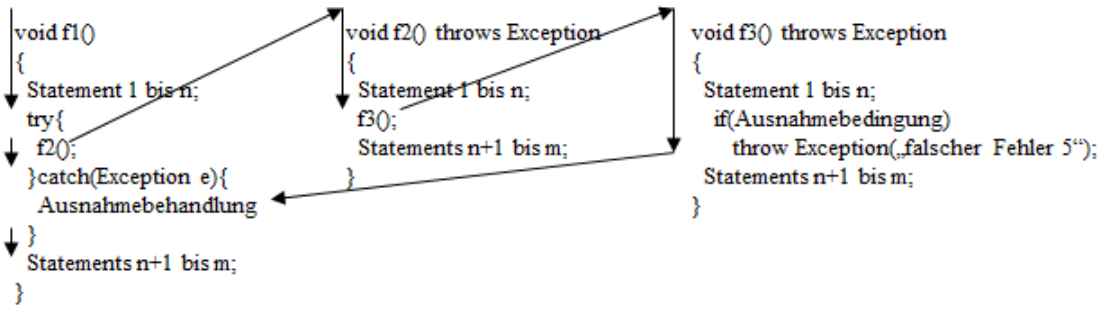

Abbildung 14: Programmablauf im Falle einer Ausnahme (Exception)

Wird in einer Funktion (`f3`) eine Exception geworfen (**`throw`** `Exception`), wird der gesamte Aufrufstack abgewickelt (**stack unwinding**), bis die Exception aufgefangen (**`catch`**) und behandelt wird (in `f1`). Wird die Exception nicht aufgefangen wird `main()` verlassen und damit das Programm beendet. Alle lokalen Objekte auf dem Stack werden zerstört, d.h. der Destruktor wird gerufen.

[198] [Str00a] Kapitel 2.2.3 und Kapitel 3.3

Als Exception können alle Arten von Objekten geworfen werden, also auch die eingebauten Typen. Besser ist es aber, eine Exception Hierarchie zu verwenden, so dass der Auslöser der Ausnahme (`f3`) der Ausnahmebehandlung (`f1` **`catch`**`()`) Informationen über den Grund der Ausnahme mitteilen kann.

Die **`catch`**`-clause` muss hierarchisch gestaffelt werden. Dabei müssen speziellere Exceptions vor den generellen Exceptions, innerhalb einer Exception Klassenhierarchie, aufgelistet werden, da sonst der Ausnahmehandler des speziellen catch Blocks nie erreicht wird.

Mit der Ellipse **`catch`**`(...)` werden alle Arten von Ausnahmen aufgefangen.

Ausgehend von einer Vererbungshierarchie:

```
class BaseException : public std::exception {...};
class Derived1Exception : public BaseException {...};
class Derived2Exception : public BaseException {...};
```

würden die Exceptionhandler wie folgt erreicht:

Listing 128: Exceptionhandler

```
1  try {
2     if(meinProblem)
3        throw Derived1Exception("Mein Problem");
4     else if(anderesProblem)
5        throw Derived2Exception("anderes Problem");
6  }
7
8  catch(const Derived1Exception& e){...} // fängt Derived1Exception auf
9  catch(const BaseException& e){...} // fängt Derived2Exception auf
10 catch(const std::exception& e){...}
11 catch(...){...} // hier alles andere behandeln
```

Exceptions sollten immer **`catch`** `by reference` aufgefangen werden um das **type slicing Problem** zu umgehen [199] (Zuweisung eines abgeleiteten Objektes einem Objekt der Basisklasse mit Verlust des statischen Types und der dazugehörigen Eigenschaften). siehe section 6.16.4 auf Seite 53.

Die Elipse **`catch`**`(...)` fängt alle Exceptions auf.

throw Specification Deprecated Die **`throw`**`(...)` Spezifikation (`f3`, `f2`) bei Funktionen und Methoden besagt, dass nur die aufgelisteten Exceptions geworfen werden. Eine leere **`throw`**`()` clause garantiert, dass keine Exception geworfen wird. Dieses Feature ist deprecated in C++11.

Wird die **exception specification** verletzt, also eine andere Exception geworfen, als die aufgelisteten, wird die Funktion `unexpected()` gerufen, die ihrerseits die Funktion `terminate()` aufruft und diese beendet das Programm. Listet die Funktion **`void`** `f()`**`throw`**`(MyException, std::bad_exception)` in ihrer **`throw`**-clause auf, wird die Funktion `unexpected()` aufgerufen, die die unerwartete Exception durch `bad_exception` ersetzt.

[199] [Mey99] Item 13 Catch exceptions by reference

6.34.2 Exceptions in Konstruktoren

Wenn bei der Initialisierung eines Objekts eine Exception auftritt, werden nur die Destruktoren der bereits konstruierten Basisklassen und Member gerufen. Der Destruktor des Objekts wird nicht gerufen. Der Funktionskörper des Konstruktors des Objekts wird nicht erreicht.

Die Initialisiererliste des Konstruktors kann, wie in Listing 129 im Konstruktor `A(char)`, in einen try/catch Block eingeschlossen werden. Damit lassen sich zwar die Exceptions der Konstruktoren der Basisklassen und Member auffangen, das Objekt wird aber trotzdem in einem ungültigen Zustand sein und es wird an dieser Stelle nicht möglich sein, das Objekt in einen gültigen Zustand zu versetzen. Daher ist das default Verhalten, - anders als in ordinären Funktionen, - die Exception wird am Ende der **catch** Klausel wieder geworfen (rethrow)[200].

Repository: Cpp-Basics/ExceptionsCtorDelegation

Listing 129: Ausnahmen in Konstruktoren

```
1  class A {
2     B b;
3     C c;
4
5  public:
6     A(bool throwException = true) : b(), c(throwException) {
7        std::cout << "A::A(bool throwException = "
8              << std::boolalpha << throwException
9              << std::noboolalpha << ")" << std::endl;
10    }
11
12    A(int i) : A(false) {
13       std::cout << "A::A(int i)" << std::endl;
14       throw std::runtime_error("A(int i)");
15    }
16
17    A(char)
18    try
19    : A()
20    {
21       std::cout << "A(char) try : A()" << std::endl;
22    }
23    catch (std::exception& e) {
24       std::cout << "A(char c) catch exception from: " << e.what() << std::endl;
25       //throw std::logic_error("replaced in catch Block");
26    }
27
28    ~A() { std::cout << "A::~A()" << std::endl; }
29 };
```

Der Konstruktor `A(int): A(false)` delegiert die Konstruktion an den Konstruktor `A(bool)` und wirft dann eine Exception. Das Objekt ist nach dem ersten Konstruktor

[200] [Str13] Section 13.5.2.4

vollständig konstruiert, daher wird auch der Destruktor von A gerufen[201].

Die Klasse B erzeugt lediglich eine Ausgabe im Konstruktor und im Destruktor. Die Klasse C wirft bedingt eine Exception im Konstruktor. Die Anwendung demonstriert den Aufruf der Konstruktoren und Destruktoren, sowie die Weiterleitung der Exception (rethrow) nach der **catch** Klausel. Die Anwendung erzeugt jeweils ein Objekt der Klasse A mit den verschiedenen Konstruktoren.

Die im **catch** Block aufgefangene Exception kann durch eine andere ersetzt werden. Im A(**char**) Konstruktor ist im **catch** Block eine auskommentierte **throw** Anweisung die zu der als Kommentar gekennzeichneten Ausgabe in Listing 131 auf der nächsten Seite führt, wenn sie einkommentiert wird.

Listing 130: Ausnahmen in Konstruktoren Main

```
1  class B {
2  public:
3     B() { std::cout << "B::B()" << std::endl; }
4
5     ~B() { std::cout << "B::~B()" << std::endl; }
6  };
7
8  class C {
9  public:
10    C(bool throwException = true) {
11       std::cout
12          << "C::C(bool throwException = "
13          << std::boolalpha << throwException
14          << std::noboolalpha << ")" << std::endl;
15       if(throwException)
16          throw std::runtime_error("C(bool throwException)");
17    }
18
19    ~C() { std::cout << "C::~C()" << std::endl; }
20 };
21
22 int main(){
23    cout << "=== CtorWithException ===" << endl;
24
25    cout << endl << "### main: A a;" << endl;
26    try {
27       A a;
28    }
29    catch (exception& e) {
30       cout << "main catch exception from: " << e.what() << endl;
31    }
32
33    cout << endl << "### main: A a(int);" << endl;
34    try {
35       A a(23);
36    }
```

[201] siehe section 6.11 auf Seite 45

```
37    catch (exception& e) {
38       cout << "main catch exception from: " << e.what() << endl;
39    }
40
41
42    cout << endl << "### main: A a(char);" << endl;
43    try {
44       A a('c');
45    }
46    catch (exception& e) {
47       cout << "main catch exception from: " << e.what() << endl;
48    }
49 }
```

Die Ausgabe von Listing 130 auf der vorherigen Seite ist in Listing 131 abgebildet.

Listing 131: Ausnahmen in Konstruktoren Ausgabe

```
1  === CtorWithException ===
2
3  ### main: A a;
4  B::B()
5  C::C(bool throwException = true)
6  B::~B()
7  main catch exception from: C(bool throwException)
8
9  ### main: A a(int);
10 B::B()
11 C::C(bool throwException = false)
12 A::A(bool throwException = false)
13 A::A(int i)
14 A::~A()
15 C::~C()
16 B::~B()
17 main catch exception from: A(int i)
18
19 ### main: A a(char);
20 B::B()
21 C::C(bool throwException = true)
22 B::~B()
23 A(char c) catch exception from: C(bool throwException)
24 main catch exception from: C(bool throwException)
25 //main catch exception from: replaced in catch Block
```

6.34.3 Die Standard Exception Klassen

1. Exceptions für den Sprach Support
2. Exceptions für die STL
3. Exceptions für Errors außerhalb des Scopes eines Programms

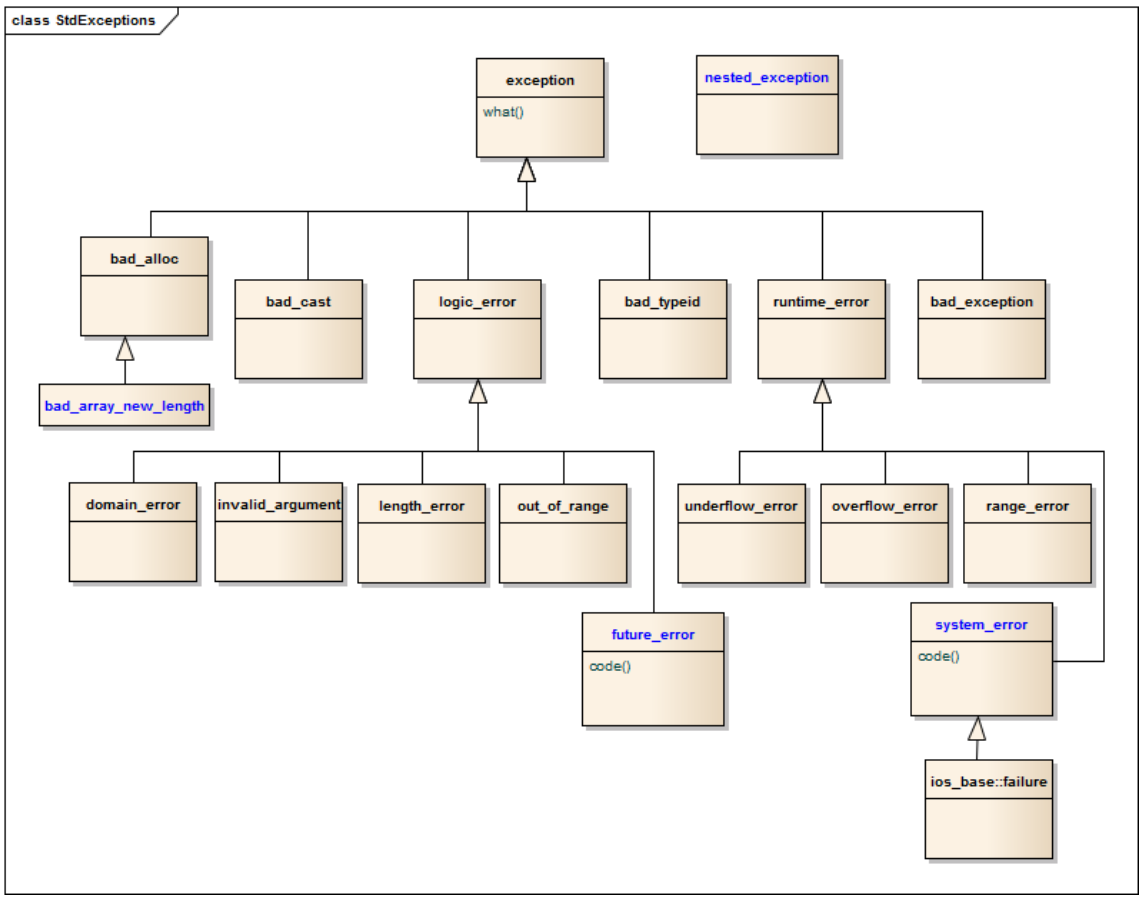

Diagramm 6: Exception Hierarchie der Standard Exceptions

Die Exceptions für den Sprach Support:

1. `bad_alloc`, von **`new`** wenn kein Speicher verfügbar ist
2. `bad_cast`, von `dynamic_cast` bei Referencen die nicht passen
3. `bad_exception`, siehe oben
4. `bad_typid`, von `typid` wenn das Argument ein Null-Pointer ist

Die Exceptions für die STL:

1. `invalid_argument`, ungültige Argumentwerte
2. `length_error`, wenn eine maximale Größe überschritten wird, z.B. wenn die maximale Länge eines Strings überschritten wird.
3. `out_of_range`, z.B.: wenn der Index eines Vectors nicht im Bereich 0 <= idx < length liegt
4. `domain_error`, um einen Anwender Fehler zu Reporten.
5. `ios_base::failure`, von streams bei Fehlern oder end-of-file EOF

Die Klasse `domain_error` ist von `logic_error` abgeleitet. Objekte der Klasse `logic_error` werden geworfen, wenn die Ursache für das Problem durch den Anwender verursacht ist, zum Beispiel, die Nichteinhaltung von Preconditions.

Die Exceptions für außerhalb des Scopes eines Programmes `range_error`, Range Error bei der internen Verarbeitung

`overflow_error`, `underflow_error`, arithmetische Über- oder Unterläufe.

Die STL wirft folgende Exceptions:

Direkte Exceptions: `range_error`, `out_of_range`, `invalid_argument`,

indirekte: `bad_alloc` oder user spezifische Exceptions

6.34.4 Exception Sicherheit und Exception Neutralität

„Exception-Behandlung und Templates sind zwei der mächtigsten C++ Merkmale. Exception-sicheren Code zu schreiben kann jedoch recht schwierig sein – besonders in einem Template, wenn man keinen Anhaltspunkt hat, welche Exception wann in einer bestimmten Funktion ausgeworfen wird„[202]

Exception-neutral bedeutet, dass ein Modul alle Exceptions die es nicht behandeln kann nach außen weiterleitet und keine „verschluckt“.

Was bedeutet Exception Sicherheit und welche Garantien können gegeben werden? Bei den Garantien werden 3 Stufen unterschieden:

1. Grundlegende Garantie: In Gegenwart einer Exception geht kein Speicher verloren, das Programm bleibt in einem konsistenten Zustand
2. Hohe Garantie (strong guarantee): In Gegenwart einer Exception bleibt das Programm in einem unveränderten Zustand, die Operation (Nachricht) hat also keinen Effekt
3. Absolute Garantie: (`noexept` guarantee) es werden keine Exceptions geworfen

Exception-Sicherheit kann nur durch entsprechendes Design erlangt werden! Voraussetzung für die Programmierung von Exception-sicherem Code sind die (basic requirements) Anforderungen an die verwendeten Klassen:
Destruktoren dürfen keine Exceptions werfen!

Das folgende Beispiel leistet auf den ersten Blick hohe Garantie, zwingt den Anwender des Stacks aber sehr subtil dazu, unsicheren Code zu schreiben:

Listing 132: Exception Sicherheit

```
1 template<class T> class Stack {
2 public:
3    ...
4    T pop();
5 private:
6    T* v; //Zeiger auf Speicher
7    size_t vsize;
8    size_t vused; // anzahl der Elemente auf dem Stack
9 };
```

[202] [Sut01] Lektion 8ff siehe : http://www.stlport.org/doc/exception_safety.html

```
10
11 template<class T>
12 T Stack<T>::pop(){
13    if(vused == 0){
14       throw "pop bei leerem stack";
15    }
16    else{
17       T result = v[vused-1];
18       --vused;
19       return result;
20    }
21 }
```

Ist der Stack leer, wird eine entsprechende Exception geworfen. Im anderen Fall wird die erste Kopie des Objekts erzeugt und der Stack Zähler aktualisiert. Wird bei der ersten Kopie eine Exception ausgelöst, bleibt der Stack in seinem ursprünglichen Zustand (hohe Garantie). Zum Schluss wird das Objekt by Value zurückgegeben. Das Problem tritt erst in der Anwendung auf, wenn bei der zweiten Kopie, die im return Statement entstehen kann, eine Exception auftritt.

```
string s1(s.pop());
string s2;
s2 = s.pop();
```

Dann ist der Stack bereits aktualisiert, das Objekt ist aber für immer verloren. Der Anwender ist damit gezwungen Exception unsicheren Code zu schreiben.

Die Operation pop() hat zwei Verantwortlichkeiten! Das ist das eigentliche Problem. pop() muss das oberste Element entfernen und dessen Wert zurückliefern. Verteilt man diese Verantwortlichkeiten auf zwei Operationen wie in Listing 133 ist das Problem eliminiert und Exception-sicherer Code ist möglich. `top()` liefert das oberste Objekt, und `pop()` entfernt es.

Listing 133: Exception Sicherheit 2

```
1  template<class T>
2  void Stack<T>::pop() {
3     if(vused == 0) {
4        throw "pop bei leerem stack";
5     } else {
6        --vused;
7     }
8  }
9
10 template<class T>
11 T Stack<T>::top() {
12    if(vused == 0) {
13       throw "top bei leerem stack";
14    } else {
15       return v[vused-1];
16    }
17 }
```

Kohäsion: Jeder Teil des Systems, jedes Modul, jede Klasse, Methode sollte genau eine Verantwortlichkeit haben, genau eine Aufgabe lösen. Dadurch entsteht Kohäsion, was in diesem Zusammenhang bedeutet, dass alle Elemente, Konstanten, Variablen, Anweisungen, usw. in engem Zusammenhang stehen, nämlich diese Verantwortlichkeit zu erfüllen.

Diese Schnittstelle (`top/pop`) finden wir auch in der STL. Eine Alternative wäre `pop(T& result)` zu definieren.

6.34.5 Das Schlüsselwort noexcept (ab C++11)[203]

Eine Funktion die keine Exceptions wirft, kann mit dem Schlüsselwort `noexcept` spezifiziert werden: `void f()noexcept;`. Diese Spezifikation ist eine Kurzform von `void f()noexcept(true);` Wirft eine so spezifizierte Funktion trotzdem eine Exception, wird `std::terminate()` aufgerufen, die `std::abort()` aufruft und das Programm terminiert.

Das Schlüsselwort `noexcept` löst einige Probleme die sich mit den alten (leeren) Exception Spezifikationen ergaben.

- Runtime checking: C++ exception Spezifikationen werden zur Laufzeit und nicht zur Übersetzungszeit geprüft. Dadurch besteht keine Garantie, dass alle Exceptions behandelt werden.
- Runtime overhead: Für die Überprüfung zur Laufzeit muss der Compiler zusätzlichen Code produzieren, der Opimierungen behindert.
- Unbrauchbar in generischem Code: Innerhalb von generischem Code sind die zu erwartenden Exceptions unbekannt, daher ist eine genaue throws-Spezifikation nicht möglich.

Daher ist nur von Interesse, ob überhaupt Exceptions geworfen werden oder nicht. Letzteres wird mit dem Schlüsselwort `noexcept(true)` zum Ausdruck gebracht.

Wie in Listing 134 gezeigt, kann die `noexcept([true|false])` Spezifikation bedingt erfolgen. Hier wird mit dem Operator `noexcept(t1.swap(t2))` ermittelt, ob der Ausdruck `t1.swap(t2)` keine Exception wirft, wenn ja, liefert der Operator `true` und die generierte Funktion `swap` für diesen Typ, ist `void swap(...)noexcept`.

Listing 134: Bedingte noexcept Spezifikation global swap

```
1 template<class T>
2 void swap(T& t1, T& t2) noexcept(noexcept(t1.swap(t2))){
3    t1.swap(t2);
4 }
```

Ein anderes Beispiel für eine bedingte `noexcept` Spezifikation ist in Listing 135 auf der nächsten Seite. Hier wird das type trait `is_nothrow_move_assignable` verwendet, um zu überprüfen, ob mit den übergebenen Typen move assignment ohne Exceptions möglich ist.

[203] [Jos12] Keyword noexcept

Listing 135: Move Assignment für pair C++11

```
template<class T1, class T2>
struct pair{
...
pair& operator=(pair&& rhs)
   noexcept(is_nothrow_move_assignable<T1>::value) &&
   noexcept(is_nothrow_move_assignable<T2>::value)
...};
```

Im Standard wird nach [N3279:LibNoexcept], die Verwendung von `noexcept` zurückhaltend festgelegt. Die folgende Auflistung stellt eine Skizze der Spezifikation dar.

- Jede Bibliotheksfunktion die keine Exceptions wirft und kein „undefined behavior“ zeitigt, sollte als bedingungslos noexcept deklariert werden.
- Die Bibliotheksfunktionen swap, move constructor, move assignment operator, für die mit dem Operator `noexcept(...)` nachgewiesen werden kann, dass sie keine Exception wirft, sollte als bedingt noexcept deklariert werden.
- Kein Destruktor der Standardbibliothek darf eine Exception werfen.
- Bibliotheksfunktionen die für die Kompatibilität mit C code gestaltet sind, können als bedingungslos noexcept deklariert werden.

Zu beachten: `noexcept` wird bewußt nicht auf Bibliotheksfunktionen angewendet, die eine Precondition / Vorbedingung voraussetzen, die - wenn nicht erfüllt - zu undefiniertem Verhalten führen kann. Damit ist es möglich, „safe mode“ Implementierungen zur Verfügung zu stellen, die eine „precondition violation“ Exception werfen.

In den meisten Beispielen in diesem Script wurde die Exceptionsspezifikation weggelassen.

6.35 Das Schlüsselwort typedef und using

Mit der typedef Anweisung wird ein Aliasname für einen Typ definiert. Der Aliasname und der Name des Typs sind gleichwertig. Seit C++11 kann dafür auch das Schlüsselwort `using newname = oldname` verwendet werden[204]. Mit der neuen Syntax ist auch die partielle Ersetzung von Parametern für Templates möglich, wie in Listing 137 auf der nächsten Seite gezeigt.

Listing 136: typedef

```
// type alias declarations
typedef unsigned int UINT;
typedef std::vector<int> IntVector;
// alternative since C++ 11
using UINT = unsigned int;
using IntVector = std::vector<int>;
```

[204] siehe section 9.10.1 auf Seite 193 und section 6.18 auf Seite 55

```
// type alias usage
UINT ui = 0;
IntVector intVector;
// Ist äquivalent zu
unsigned int ui = 0;
std::vector<int> intVector;
```

Listing 137: Das Schlüsselwort using

```
template<class T1, class T2>
class MyTemplate;

template<class T>
using AndererName = MyTemplate<int, T>;

AndererName<double> var; // var of type MyTemplate<int, double>
```

6.36 Das Schlüsselwort `const` und `mutable`[205]

Der *type qualifier*[206] **const** ist linksbindend, er bezieht sich auf den Typ der links davon steht, außer es steht nur rechts ein Typ, dann bezieht sich **const** auf den Typ. Deshalb ist die Definition
const `std::string& cs` und
`std::string` **const**`& cs` gleichwertig. Lies: „cs ist eine Referenz auf ein konstantes Stringobjekt.

Machen Sie alles was möglich ist, **const**! Konstanz / constness ist Bestandteil des Vertrages der durch eine Schnittstelle angeboten wird. Bsp.: Eine Operation die in einer Klasse Widget wie folgt deklariert ist: `string` **const**`& getName()` **const**`;` liefert eine Referenz auf ein konstantes String Objekt das vom Client nicht verändert werden darf und garantiert dem Client, dass das Widget Objekt, dem die Nachricht `getName()` gesendet wird, durch die Methode nicht verändert wird. Der Programmierer der Methode wird vom Compiler unterstütz, diesen Vertrag auch einzuhalten, der **this** Pointer ist vom Typ: `Widget` **const** `*` **const this**, Attribute des Objekts können in dieser Methode nicht verändert werden. Das wird durch **const** hinter dem Operationsnamen ausgesagt.

Der Aufruf der Operation ist ein Ausdruck (Rückgabetyp) vom Typ **Referenz auf ein konstantes string Objekt**.

In Parametern garantiert **const**, dass das übergebene Objekt vom Server nicht verändert wird. Bsp.:

```
void setName(string const& name);
```

`name` ist vom Typ **Referenz auf ein konstantes string Objekt** das innerhalb der Methode nicht verändert werden kann.

[205] [Mey06a] Item 3 Accustoming Yourself to C++
[206] http://en.cppreference.com/w/cpp/language/cv

6.36.1 Konstante Objekte und konstante Operationen

Werden Objekte via `const&` übergeben, können auf sie nur `const` Operationen angewendet werden.

Listing 138: const

```
void f(Widget const& w) {
   // der Aufruf ist nur möglich wenn getName() const deklariert ist.
   cout << w.getName();
}
```

Steht eine Operation sowohl als `const` und non-const Version zur Verfügung, wird für konstante Objekte die const-Version und für nicht konstante Objekte die non-const Version ausgewählt.

Listing 139: Konstante Objekte und konstante Operationen

```
class Widget
{
public:
   void op()       { cout << "op()" << endl; }
   void op() const { cout << "op() const" << endl; }
};

void f(Widget& w, Widget const& cw){
   w.op();  // calls op()
   cw.op(); // calls op() const
}
int main(){
   Widget w;
   f(w, w);
}
```

6.36.2 Unterscheidung physikalische und logische Konstantheit

[207] Anwendung des Schlüsselworts **mutable**. Es gibt Fälle, in denen das äußere Verhalten eines Objekts unverändert ist (`const`) aber innerhalb der Verarbeitung einer Nachricht Teile des Objekts verändert werden müssen. Diese Teile müssen `mutable` deklariert werden.

Bsp.: Eine Klasse `Datum` stellt eine Operation

```
string const& toString()const;
```

zur Verfügung. Wenn die Erzeugung der Stringrepräsentation sehr kostspielig ist, könnte diese erst bei Bedarf berechnet werden und für ungültig erklärt werden, wenn sich das Objekt, also das Datum, verändert. Bei erneuter Anfrage der Stringrepräsentation muss sie wieder erneut berechnet werden. Der Cache für

[207] [Str00a] 10.2.7.1 Physikalische und logische Konstantheit

die Stringrepräsentation muss also auch für Objekte der Art: `const Datum d;` veränderbar sein.

Listing 140: class Datum und mutable member

```
class Datum {
   mutable bool cacheGueltig;
   mutable string cache;
   void ermittleCacheWert() const;
   int tag, monat, jahr;
public:
   string const& toString() const;
   void setDatum(int tag, int monat, int jahr);
};

void Datum::setDatum(int tag, int monat, int jahr) {
   this->tag = tag;
   this->monat = monat;
   this->jahr = jahr;
   cacheGueltig = false;
}

const string& Datum::toString() const {
   if(!cacheGueltig)
      ermittleCacheWert(); // verändert cache und cacheGueltig
   return cache;
}

void Datum::ermittleCacheWert() const {
   ... // Stringrepräsentation berechnen
   cacheGueltig = true;

}
```

Listing 141: const_cast<..>

```
String const& Datum::toString() const {
   if(!cacheGueltig) {
      Datum * pD = const_cast<Datum*>(this);
      pD->ermittleCacheWert(); // Aufruf einer non-const Methode
   }
   return cache;
}
```

Durch einen cast wie in Listing 141 könnte die `constness` ebenfalls übergangen werden, `ermittleCacheWert()` könnte non-const deklariert werden und die Attribute ändern. Wenn Compiler konstante Objekte in einen readonly Speicherbereich legen, wird das aber nicht funktionieren und `undefined behavior` nach sich ziehen. Mit `mutable` bekommt der Compiler den Hinweis, dass diese Objekte (`cache` und `cacheGueltig`) auch für konstante Objekte änderbar sein müssen.

6.36.3 const und Pointer

Im Zusammenhang mit Pointern[208] gibt es vier Möglichkeiten mit const umzugehen wie in Tabelle 9 gezeigt. Zeiger auf konstante Objekte können immer auch auf

Tabelle 9: const und Pointer

	Zeiger auf nicht konstante Objekte	Zeiger auf konstante Objekte
Nicht konstante Zeiger	int *	int const *
konstante Zeiger	int * const	int const * const

Objekte die nicht const sind, zeigen, umgekehrt aber nicht. Ein Zeiger, über den ein Objekt verändert werden kann, darf nicht auf ein konstantes Objekt zeigen.

Um die Syntax leichter zu verstehen kann die Deklaration von rechts nach links wie folgt gelesen werden. In den Namen der Bezeichner ist in den folgenden Beispielen der Typ codiert. Dabei steht i für int, c für const und p für Pointer.

6.36.4 Variablen und Konstanten

```
int i(41); //eine Variable, Typ int, wert 41
int const ci(42); //eine Konstante, Typ int, wert 42
```

lies: `i` ist ein **`int`**

lies: `ci` ist ein konstanter **`int`**

6.36.5 Zeiger auf nicht konstante Objekte

```
int * pi; //pi ist ein Zeiger auf einen int
pi = &i; //Adresse von i pi zuweisen, pi zeigt auf i
*pi = 3; //i den Wert 3 über pi zuweisen
//pi = &ci; ERROR, pi kann nur auf non-const Objekte zeigen
```

lies: `pi` ist ein Zeiger auf einen **`int`**

Über diesen Zeiger kann das Objekt auf den er zeigt, verändert werden. Daher kann `pi` nicht auf die Konstante `ci` zeigen. Oder anders ausgedrückt: Der Typ von (lies: adresse von ci) `&ci` ist **`int const*`** und nicht **`int*`**.

6.36.6 Zeiger auf konstante Objekte

```
int const * pci; //pci ist ein zeiger auf einen konstanten int
pci = &i; //ok, i kann nicht über pci verändert werden
```

[208] section 6.28 auf Seite 107

```
//*pci = 3; ERROR, pci ist ein Zeiger auf ein konstantes Objekt

pci = &ci; //dto wie bei &i
```

6.36.7 Konstante Zeiger auf nicht konstante Objekte

```
int * const cpi1 = &i; //cpi1 ist ein konstanter Zeiger auf einen int

*cpi1 = 3; //Den Wert von i über cpi1 verändern

//int * const cpi2 = &ci; ERROR cpi2 kann nur auf non-const Objekte zeigen.
```

`ci` und `cpi1` sind Konstanten! Eine Konstante muss bei der Deklaration auch initialisiert werden weil sie danach nicht mehr verändert werden kann. `cpi1` kann also nicht mehr auf ein anderes Objekt zeigen. `cpi1` zeigt aber auf ein non-const Objekt. Das Objekt auf das `cpi1` zeigt, kann über `cpi1` verändert werden! Daher kann `cpi2` nicht auf `ci` zeigen.

6.36.8 Konstante Zeiger auf konstante Objekte

```
int const * const cpci1 = &i; //konstanter zeiger auf konstanten int

int const * const cpci2 = &ci;

//*cpci2 = 3; ERROR cpci2 zeigt auf ein konstantes objekt
```

`cpci1` kann nicht mehr verändert werden und das Objekt auf den `cpci1` zeigt, kann über `cpci1` nicht verändert werden. `cpci2` kann daher auf `ci` zeigen, weil `ci` über `cpci2` nicht verändert werden kann.

6.37 Typkonvertierung benutzerdefinierter Typen

Benutzer definierte Typen können durch Konstruktoren und cast Operatoren konvertiert werden. Konstruktoren erzeugen Objekte des Typs aus anderen Objekten und cast Operatoren konvertieren die Objekte des Typs in Objekte eines anderen Typs.

Eine benutzer definierte Konvertierung wird nur angewandt, wenn sie eindeutig ist und nur eine Konvertierung notwendig ist.

6.37.1 Implizite Konvertierung

Ein Konstruktor mit nur einem Parameter, wie in Listing 142 auf der nächsten Seite oder mehreren Parametern, die default Argumente haben, wird vom Compiler als impliziter Konvertierungskonstruktor verwendet. Seit der Einführung der einheitlichen Initialisierungsliste gilt dies auch für mehrere Parameter wie in section 6.14 auf Seite 47 beschrieben.

Ein Operator der den Namen eines Typs hat, ist ein cast Operator in diesen Typ. Die Syntax ist von anderen Operatoren abweichend, der Rückgabetyp ist implizit und kann nicht angegeben werden! In Listing 142 hat die Klasse `B` einen cast `operator A()` und die Klasse `A` einen cast `operator int()`, mit denen aus einem `B` Objekt ein `A` Objekt und aus einem `A` Objekt ein `int` Objekt erzeugt werden kann.

Listing 142: benutzer definierte Typ Konvertierung

```
1  class A {
2  public:
3     A(int i); // Konvertierungskonstruktor
4     operator int(); // cast Operator int Rückgabe Typ implizit!
5     ...
6  };
7
8  class B{
9  public:
10    operator A();
11 };
12
13 void f1(A);
14 void f2(A const&);
15 void f3(A&);
16 void f4(int)
17
18 void f() {
19    // a wird mit temporärem A, das mit A(int i) erzeugt wird, initialisiert
20    A a = 42; // 1
21    f1(42); // 2
22    f2(42); // 3
23    f3(42); // 4 Errror
24    f4(a); //  5 Operator int()
25    B b;
26    f4(b); //  6 Error b -> A -> int nur eine implizite Konvertierung
27 }
```

In den ersten 3 Fällen erzeugt der Compiler ein temporäres Objekt A aus dem int 42 und initialisiert damit bei 1 die lokale Variable a, bei 2 den Parameter von f1 und bei 3 die Referenz auf ein A. Dieses Verhalten wird erzielt durch Konstruktoren mit nur einem Parameter und Konstruktoren mit mehreren Parametern, wenn für die Parameter ein Defaultwert definiert ist.

Im Fall 4 ist keine implizite Konvertierung durch den Compiler möglich, da f3 keine konstante Referenz auf ein A (`A const&`), sondern eine Referenz auf ein A (`A&`), erwartet. Das übergebene Objekt könnte also durch f3 verändert werden, was im Falle eines temporären Objekts keine Auswirkung hätte. Darum wird keine implizite Konvertierung von 42 in ein A durch den Compiler vorgenommen. siehe section 6.29 auf Seite 111.

Im Fall 5 erwartet `f4(int)` ein `int` Objekt als Argument, aber ein `A` wird übergeben. Die Klasse `A` definiert mit `operator int()` die Konvertierung in `int`.

Im Fall 6 wird `f4(b)` aufgerufen. Für die Klasse `B` ist eine Konvertierung nach `A`,

und für A ist eine Konvertierung nach **int** definiert, aber es wird nur eine implizite Konvertierung durch den Compiler angewendet.

6.37.2 Das Schlüsselwort explicit

Ist die implizite Konvertierung durch einen Konstruktor oder einen cast Operator nicht gewünscht, müssen diese mit dem Schlüsselwort `explicit` deklariert werden.

Listing 143: Das Schlüsselwort explicit

```
1 class A {
2 public:
3    explicit A(int i); // Konvertierungskonstruktor
4    explicit operator int(); // cast Operator int Rückgabe Typ implizit!
5    ...
6 };
```

Sie müssen dann explizit aufgerufen werden:

```
A a(42); //1 explizite Initialisierung

f1(A(42)); //2 explizite Konvertierung dto für f2

f4(static_cast<int>(a)); //5 explizite Konvertierung
```

Wird der Kopiekonstruktor `explicit` deklariert, kann keine Argumentübergabe by value mehr erfolgen!

7 Memory Management

7.1 Allgemeines

Die Speicherverwaltung in C++ wird von den Operatoren **new** und `delete` erledigt. Soll die Speicherverwaltung an spezielle Bedürfnisse angepasst werden, können diese Operatoren überladen werden[209]. Entweder global oder für eine bestimmte Klasse. Dabei müssen bestimmte Konventionen eingehalten werden[210].

7.1.1 Die Anwendung von new

Der Operator **new** beschafft Speicher für genau ein Objekt und initialisiert diesen mit dem ausgewählten Konstruktor.

```
Widget* pW = new Widget(42); //int Ctor
```

Soll ein Array von Objekten dynamisch erzeugt werden, kommt der *array new* Operator **new**`[]` zum Einsatz. Die eckigen Klammern sind dabei Teil des Funktionsnamens.

```
Widget* pWA = new Widget[10]; //10 x default Ctor
```

Bei Arrays wird für alle Elemente der default Konstruktor aufgerufen.
Die Objekte müssen mit `delete` bzw. mit *array delete* `delete[]` wieder zerstört werden, ansonsten kann der Speicher nicht wieder verwendet werden (memory leaks).
Es ist ein Fehler, Speicher, der mit **new**`[]` angefordert wurde, nicht mit *array delete* `delete[]` sondern mit `delete` wieder freizugeben.
Arrays im allgemeinen und dynamische Arrays im besonderen haben in C++ keine große Bedeutung und sollten wenn möglich vermieden werden, da die STL Container komfortablere Möglichkeiten zur Verwaltung von Mengen zur Verfügung stellen.

Alles was im folgenden bzgl. Operator **new** gilt, gilt auch für *array new*: **new**`[]`, wenn nichts anderes beschrieben ist.

7.1.2 Der Ausdruck new/new[]

Der Compiler erzeugt aus dem new bzw. delete etwa die Logik in Listing 144.

Listing 144: Die Logik von new und delete

```
1 Widget *w = new Widget;
2 //pseudocode:
3 Widget *w = operator new(sizeof(Widget)); // raw memory allocation
4 Widget(this=w) Ctor aufrufen
5 //-------------------------
6 delete w;
7 //pseudocode:
```

[209] section 7.2 auf Seite 165
[210] [Mey99] Item 8 und [Mey06a] Chapter 8

```
8 w->~Widget();  // dtor aufrufen
9 operator delete(w);  //Speicher freigeben
```

Der Operator `delete` zerstört nur ein Objekt, daher ist es wichtig, den richtigen Operator `delete[]` zu verwenden, der ruft für alle Objekte den Destruktor auf!

Ein Ausdruck der Form: **`new`** `Widget`
hat den Typ `Widget*`, liefert die Adresse des neuen Objekts und hat als Nebeneffekte:

1. Der `operator` **`new`**`(sizeof(Widget))` wird aufgefordert, Speicher in der Größe `sizeof(Widget)` zu beschaffen
2. Der Konstruktor der Klasse `Widget` wird aufgerufen, der **`this`** Pointer ist mit der Adresse, die der `operator` **`new`**`(std::size_t size)` geliefert hat, initialisiert.

Dieses Verhalten kann nicht geändert werden. Die Art, wie Speicher beschafft wird, kann aber durch den `operator` **`new`** an die Bedürfnisse angepasst werden. Kommt der `operator` **`new`**`[]` zum Einsatz, wird der Default Konstruktor für alle Objekte aufgerufen, der Wert des Ausdrucks ist die Adresse des ersten Elements im Array.

7.1.3 Der new Handler

Der new Handler hat die Aufgabe, auf Anforderung z.B. von **`new`**, eventuell nicht mehr benötigten Speicher freizugeben.
`typedef` **`void`** `(*new_handler)();` ist ein `typedef` für einen Zeiger auf eine Funktion die nichts zurückliefert und keine Argumente erwartet. Das ist die erwartete Signatur für den new Handler.
Mit `std::new_handler set_new_handler(std::new_handler newHandler);` kann eine Funktion als new Handler gesetzt werden, die bei Bedarf gerufen wird. `set_new_handler (..)` liefert den letzten `new_handler` zurück. Seit C++11 gibt es auch die Funktion `new_handler get_new_handler()`.
Bei Programmstart ist kein `new_hanlder` installiert.
Wenn der `operator` **`new`**`(..)` keinen Speicher in der angeforderten Größe beschaffen kann, ruft er den installierten `new_hanlder` auf und versucht danach, wieder Speicher zu beschaffen. Das wird solange wiederholt, bis entweder Speicher zur Verfügung steht oder kein `new_hanlder` mehr installiert ist. Wenn kein Speicher beschafft werden kann, wird eine `std::bad_alloc` Exception geworfen.
Dieses Verhalten des `operator` **`new`**`(..)` sollte beim Überladen berücksichtigt werden[211]. In section 7.1.5 auf Seite 163 sind weitere Randbedingungen genauer beschrieben.

Ein ordentlicher new Handler muss daher eines der folgenden Dinge tun:

- Speicher zur Verfügung stellen, meistens indem nicht mehr benötigter Speicher dem System zurückgegeben wird. Damit ist der `operator` **`new`** beim nächsten Versuch in der Lage, die Anforderung zu erfüllen

[211] [Mey06a] Item 51 Adhere to convention when writing **`new`** and `delete`

- einen anderen new Handler installieren, der eventuell Speicher freigeben kann. Eine Variation dieses Themas ist das eigene Verhalten zu ändern und beim nächsten Aufruf etwas anderes zu tun. Da der new Handler nicht weis, welche Größe der angeforderte Speicher haben muss, könnte er versuchen, zuerst kleinere und dann zunehmend größere Speicherblöcke freizugeben, um eine Fragmentierung zu vermeiden
- den new handler deinstallieren: `set_new_handler(nullptr)`, dann wirft der `operator new` eine Exception
- Eine `bad_alloc` Exception oder eine Ableitung davon werfen. Diese wird von `operator new` nicht aufgefangen und damit zum Aufrufer von `new` propagiert
- nicht zurückkehren indem `std::abort()` oder `std::exit()` aufgerufen wird

7.1.4 Die Signaturen von new und delete

Die Operatoren `new` und `delete` werden vom Standard mit folgenden Signaturen **global** zur Verfügung gestellt. Der zum `operator new` paarweise passende `operator delete` ist jeweils nach dem new Operator in der Liste aufgeführt.

- `void* new(size_t size)throw(std::bad_alloc);`
- `void delete(void* memory)throw(); //globales delete`
- `void delete(void *memory, size_t size)throw(); //Klassenspezifisch`
- `void* new(size_t size, void* location)throw(); //placement new`
- `void delete(void *memory, void* location)throw();`
- `void* new(size_t size, const std::nothrow&)throw();`
- `void* delete(void* memory const std::nothrow&)throw();`

Eine vollständige Liste mit der Zuordnung zu den verschiedenen Standards[212] findet sich für

- `operator new`: http://en.cppreference.com/w/cpp/memory/new/operator_new
  http://www.cplusplus.com/reference/new/operator%20new/?kw=operator%20new

- `operator delete`: http://en.cppreference.com/w/cpp/memory/new/operator_delete
  http://www.cplusplus.com/reference/new/operator%20delete/?kw=operator%20delete

wieder.
Der erste Parameter muss bei allen Versionen immer `size_t`, die angeforderte Größe des Speichers in Byte sein. Er wird dem Operator `new` vom Compiler automatisch übergeben. Die Exception `bad_alloc` wird geworfen, wenn kein Speicher beschafft werden konnte. Die `throw`-Specification ist ab C++11 deprecated. Eine leere `throw`-Specification wird durch `noexept` ersetzt.

[212] C++98/03, C++11, C++14

Aus Gründen der Kompatibilität zu früheren Standards gibt es das `nothrow` **new**, das einen null Pointer zurückliefert, anstatt eine Exception zu werfen, wenn kein Speicher beschafft werden konnte.

Die klassenspezifische Variante des Operator `delete` hat als zweites Argument, die Größe des zu zerstörenden Objekts, die der Compiler ermittelt und automatisch übergibt.

7.1.5 Überladen von new

Wird der Operator **new** überladen sind einige nicht triviale Randbedingungen zu beachten, die im Folgenden skizziert werden sollen.

- Der globale Operator ::**new** sollte nur in Umgebungen überladen werden, in denen keine Konflikte mit anderen Bibliotheken oder Modulen auftreten können, da er von Allen verwendet wird. Für das Überladen des globalen Operator **new** gelten etwas strengere Regeln als für einen klassenspezifischen Operator **new**.
- Für jeden Operator **new** muss ein passender Operator `delete` zur Verfügung stehen. Dieser wird automatisch gerufen, wenn im Konstruktor eine Exception auftritt. Existiert kein passender Operator `delete` entsteht ein memory leak. Der passende Operator delete für den Operator new
  void* **new**(size_t size)**throw**(std::bad_alloc); ist
  void operator delete(**void** *rawMemory)**throw**();
- Die Anforderung `size == 0` muss wie `size == 1` behandelt werden!
- kann kein Speicher beschafft werden, sollte der `new_handler` gerufen werden[213], wenn einer vorhanden ist.

Das Listing 145 skizziert eine Implementierung des `operator` **new**.

Listing 145: new überladen

```
1  void* operator new(std::size_t size){
2     void* retVal = nullptr;
3     size = size == 0 ? 1 : size;
4
5     while(true){
6        retVal = allocateMemory(size);
7        if(retVal != nullptr)
8           return retVal;
9
10       new_handler globalHandler = std::get_new_handler();
11       if(globalHandler)
12          globalHandler();
13       else
14          throw std::bad_alloc();
15    }
16 }
```

[213] section 7.1.3 auf Seite 161

7.1.6 Placement new

Um Objekte in einem bestimmten Speicherbereich platzieren zu können, steht ein Operator **new** mit einem zweiten Parameter **void*** location im Standard zur Verfügung.

```
void* new(size_t size, void* location)noexcept; //placement new
```

location muss auf Speicher ausreichender Größe zeigen und das korrekte Alignment haben. Die Implementierung für den Operator sieht ungefähr wie folgt aus:

```
void* operator new(size_t , void *location){ return location; }
```

Soll z.B. ein Objekt in einem shared memory Bereich liegen, könnte dieser über entsprechende Funktionen verwaltet werden:

```
void* mallocShared(size_t size);
void freeShared(void* memory);
```

Ein Objekt wird dann in diesem Speicherbereich wie in Listing 146 platziert:

Listing 146: Anwendung des Placement operator new

```
1 void* sharedMemory = mallocShared(sizeof(Widget));
2 Widget pw = new(sharedMemory) Widget; //ctor aufrufen
3 ...
4 pw->~Widget(); // dtor aufrufen
5 freeShared(sharedMemory);
```

delete pw; wäre ein Fehler (undefined), da dieser Speicher nicht vom Operator **new** kommt, sondern von mallocShared(size_t)!

In der Hardware nahen Programmierung kann diese Technik angewendet werden, um ein Register, das im Hauptspeicher gemapped ist, mit einem Register-Objekt zu verwalten. Hier wird die Adresse des Registers, die als ganzzahlige Konstante feststeht, dem Placement new übergeben.

Beispiel:

Listing 147: Placement operator new zur Abbildung von Registern

```
1 class ControlRegister{ unsigned int register; ... //zugriffsoperationen };
2 unsigned int * cRAdresse = reinterpret_cast<unsigned int*>(0xFFFF0000);
3 ControlRegister *cR = new(cRAdresse) ControlRegister;
4 cR->setBit(Bit8);
```

Voraussetzung dafür ist, dass das Speicherimage von ControlRegister exakt mit der Größe des Speicherbereichs, auf den das Register gemapped ist, übereinstimmt.

Der Begriff „Placement new“ ist überladen und wird für jeden Operator new, der zusätzliche Argumente erwartet, verwendet.
Der Operator new könnte z.B. klassenspezifisch wie folgt überladen werden:
operator **new**(size_t size, std::ostream& logStream)

um Ausgaben für die Analyse der Speichernutzung zu erzeugen. Die Speicheranforderung wird einfach an den globalen Operator new weitergeleitet und davor oder danach Ausgaben erzeugt. Der dazugehörige Operator delete sieht wie folgt aus[214]:

```
operator delete(void* pMemory, std::ostream& logStream)
```

7.2 Benutzerdefinierte Speicherverwaltung

7.2.1 Das globale new

Der globale operator **new** ist für alle möglichen Speicheranforderungen zuständig, die Implementierung ist entsprechend komplex, der Aufwand zur Laufzeit ebenfalls. Er muss beispielsweise große und kleine Speicherbereiche verwalten können, er muss Speicher der lange belegt ist ebenso wie Speicher der häufig angefordert und wieder freigegeben wird, verwalten und dabei eine Fragmentierung des gesamten Speichers vermeiden. Sein **Laufzeitverhalten ist nicht deterministisch**!, Für konkrete Anwendungen ist es möglich wesentlich effizientere Strategien zur Speicherverwaltung zu nutzen, als es der allgemeine Operator **new** tun kann. Wenn z.B. bekannt ist, dass der Speicher immer in umgekehrter Reihenfolge freigegeben wird, wie er angefordert wurde (LIFO), könnte darauf basierend eine vereinfachte Verwaltung, die schneller und deterministisch ist, implementiert werden, genauso wäre eine Vereinfachung möglich, wenn bekannt ist, dass die Speicherblöcke immer die gleiche Größe haben, usw.

Eine effiziente Speicherverwaltung ist z.B. in „Composing High-Performance Memory Allocators“[215], eine Einführung in das Thema in „Policy-Based Memory Allocation"[216] und ein Allocator für Multithreaded Umgebungen ist in „Hoard: A Scalable Memory Allocator for Multithreaded Applications“[217] beschrieben.
In „Reconsidering Custom Memory Allocation“[218] kommen die Autoren zu dem Schluss, dass eine gute allgemeine Speicherverwaltung in den überwiegenden Fällen die beste Lösung ist und dass es nur in sehr wenigen extremen Situationen möglich ist, eine bessere Speicherverwaltung zu implementieren, als es die verfügbaren, über mehrere Generationen und Anwendungen hinweg erpropten Speicherverwaltungen sind. Darüber hinaus haben die angepassten Speicherverwaltungen weitere Nachteile wie z.B. Inkompatibilitäten mit der allgemeinen Speicherverwaltung, Unzugänglichkeit für bestimmte Tools, usw.

[214] siehe section 7.1.5 auf Seite 163
[215] https://people.cs.umass.edu/~emery/pubs/berger-pldi2001.pdf
[216] https://erdani.com/publications/cuj-2005-12.pdf
[217] www.cs.umass.edu/~emery/hoard/asplos2000.pdf
[218] www.cs.umass.edu/~emery/.../berger-oopsla2002.pdf

7.2.2 Vorbereitungen

Bevor es in Betracht gezogen werden sollte, eine eigene Speicherverwaltung zu schreiben, sollten intensive Untersuchungen in Form von konkreten Messungen durchgeführt werden.
Was ist zu tun und wann ist es sinnvoll, eine eigene Speicherverwaltung zu schreiben?

1. Statistik Informationen gewinnen, als Entscheidungsgrundlage zur Erstellung einer eigenen Speicherverwaltung
   (a) Art und Weise Speicherbeschaffung und Freigabe (z.B.:LIFO)
   (b) Dauer der Belegung
   (c) Angeforderte Größen ermitteln
   (d) Gesamtbedarf (Maximum)
2. Effizienz Verbesserung
   (a) Fragmentierung vermeiden
   (b) Deterministisches Laufzeitverhalten für bestimmte Klassen schaffen
   (c) Performance Verbesserung bzgl. Laufzeit und Speicherbedarf
3. Fehler erkennen
   (a) fehlendes delete: memory leaks,
   (b) mehrfaches delete: undefined behavior
   (c) Speicherbereichsüberschreitungen: undefined behavior

Für die Punkte unter 3.) gibt es hervorragende Libraries, wie in den obigen Literaturhinweisen beschrieben.

Teil III

Templates

8 Template Basics [219]

8.1 Verwendung von Templates

8.1.1 Klassen- und Funktionstemplates

Klassen- und Funktionstemplates waren ursprünglich zur generischen Definition von Klassen (Datenstrukturen, z.B. list, queue, stack, ...) und Funktionen (allgemeine Algorithmen, z.B. transform, find_if, ...) für verschiedene Typen gedacht. Dazu wird der gewünschte Typ erst bei der Anwendung des Templates spezifiziert und eine Klasse[220] oder Funktion[221] aus dem Template erzeugt. Aus der Klasse kann dann, wie aus einer normalen Klasse, ein Objekt erzeugt werden oder die Funktion aufgerufen werden. Die Container und Algorithm der STL[222] entsprechen dieser Art der Verwendung.
Das Modell[223] besteht aus 3 Ebenen: Templates, Klassen und Objekte.

Ohne die Unterstützung durch Templates ist dieses Ziel ebenfalls erreichbar, aber die Ansätze dazu haben alle ihre Nachteile, die C++ Templates nicht haben. Einige Beispiele:

1. Für jeden Typ einen eigene Implementierung zur Verfügung stellen. Die Konsequenzen sind Code Duplikation und ihre Folgen
2. Eine gemeinsame Basis schaffen. Alle Typen die einen Algorithmus nutzen wollen müssen von dieser Basis abgeleitet sein. Die Typsicherheit geht vollständig verloren oder muss zur Laufzeit gewährleistet werden
3. Präprozessor Makros verwenden. Was zu unübersichtlichem Code führt. Der Code wird von einem simplen Textersetzer erzeugt, der von Scope und Typen nichts weis.

Templates haben diese Probleme nicht. Sie sind Funktionen oder Klassen für verschiedene Typen, die während der Definition des Templates noch nicht spezifiziert sind. Der Compiler bekommt diese Definitionen zu sehen und kann auf syntaktische Korrektheit überprüfen. Erst wenn das Template benutzt wird, wird der gewünschte Typ entweder vom Anwender explizit oder durch die Argumente des Funktionstemplates[224] (Argumentdeduction) implizit angegeben. Der Compiler erzeugt dann aus dem Template und dem Typ die geeignete Implementierung.

[219] [VJ03] Basics und [Jos04] New Language Features
[220] siehe section 9 auf Seite 182
[221] section 10 auf Seite 197
[222] section 17 auf Seite 304
[223] section 8.7 auf Seite 174
[224] section 10.3 auf Seite 198

Dieser Vorgang wird auch Instanziierung des Templates genannt. Anschließend wird der Code übersetzt, wie von Hand geschriebener Code.

Templates sind in diesem Sinn Codegeneratoren, also Programme die der Compiler ausführt um Code zu erzeugen. Tatsächlich sind sie **turing-complete**, was bedeutet, dass damit alles berechnet werden kann was berechenbar ist. Sie sind eine funktionale Sprache, eingebettet in die objektorientierte Sprache C++.

Die Konsequenz daraus ist, zur Übersetzungszeit muss dem Compiler die Definition / Implementierung des Templates zur Verfügung stehen. Die einzige portable Möglichkeit, Templates zur Verfügung zu stellen, ist sie in Header Files zu implementieren. (Im Standard gibt es eine Spezifikation für ein *template compilation model* mit dem Schlüsselwort `export`, aber dafür gibt es bis heute nur eine Implementierung.) Im neuen Standard soll das Schlüsselwort in diesem Zusammenhang entfernt, bzw. deprecated erklärt werden[225].

8.1.2 Statische Polymorphie

Um das OCP[226] zu erreichen, kann eine geeignete Abstraktion in Form einer Basisklasse (dynamische Polymorphie) oder als Template Parameter (statische Polymorphie) modelliert werden. Sollen die Ausprägungen zur Laufzeit variiert werden, muss dynamische Polymorphie verwendet werden, kann eine Ausprägung zur compile time festgelegt werden, kann statische Polymorphie verwendet werden. Je früher eine Ausprägung festgelegt werden kann, desto effizienter kann das OCP bzgl. Laufzeit und Speicher realisiert werden.

8.1.3 Policy based design

Policy based design[227] ermöglicht die Erweiterung einer sogenannten Host Klasse um weitere Eigenschaften die von den Clients der Host Klasse und der Host Klasse selbst, im Sinne des Strategie Patterns, genutzt werden kann. Dafür erbt die Host Klasse von ihren Template Parametern.

8.1.4 Typisierung von Konstanten

Mit einem Template wie in Listing 148 auf der nächsten Seite können Konstanten typisiert und für diese Typen Funktionen überladen werden[228]. Jede Verwendung von `Int2Type<Wert>` erzeugt für jeden Wert einen eindeutigen Typ.

[225] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf
[226] section **??** auf Seite ??
[227] section 13.2 auf Seite 208
[228] [Ale09] 2.4 Mapping Integral Constants to Types

Listing 148: Typisierung von Konstanten

```
template<int V>
struct Int2Type{...};

//Deklaration
void f(Int2Type<1>);
void f(Int2Type<2>);
...
//Aufruf
f(int2type<1>());
```

8.1.5 type to type mapping

Mit einem Template wie in Listing 149 können alternative leichtgewichtige Typen definiert werden und für diese Typen Funktionen überladen werden[229]

Listing 149: type to type mapping

```
template<int N>
struct Type2Type{};

//Deklaration
template<class T, class U>
std::unique_ptr<T> Create(Type2Type<T>, U const & arg);
template<class U>
std::unique_ptr<Widget> Create(Type2Type<Widget>, U const & arg);
...
//Aufruf
std::unique_ptr<String> = Create(Type2Type<String>(), ‘‘Hello’’);
std::unique_ptr<Widget> = Create(Type2Type<Widget>(), 100);
```

oder wie in Listing 150 Templates spezialisiert werden. Das Template `struct SS` fasst zwei Typen zu einem zusammen.

Listing 150: SimpleSelector

```
class MyType1;
class MyType2;

template<class T1, class T2>
struct SS{...};

// Primary
template<class T>
struct Component;

//Specialization
template<>
struct Component<SS<MyType1, MyType2>>{...};
```

[229][Ale09] 2.5 Type-to-Type Mapping

8.1.6 *type-function*

*type-function*s sind eine weitere Möglichkeit, Klassen Templates zu verwenden. Dabei wird nicht die aus dem Template erzeugte Klasse genutzt, sondern der Typ, den das Template via `typedef ... type` definiert[230].
Die Grundlage für diese Art der Verwendung ist die Möglichkeit, Klassen Templates zu spezialisieren.

8.1.7 *type-method*

Wird eine *type-function* als Member Template in einem Typ oder einem Template realisiert, kann von *type-method*s gesprochen werden. Damit besteht die Möglichkeit, einem Template einen Typ zu übergeben, dessen inneres Template dazu benutzt wird, einen Typ zu erzeugen. Wie bei globalen *type-function*s, wird der Typ, den das innere Template via `typedef ... type` definiert, genutzt[231]. Der Vorteil gegenüber globalen *type-function*s ist die Möglichkeit, die zu verwendende *type-function* zu variieren.

8.1.8 *Selector*

Im Zusammenhang mit *type-function*s und *type-method*s können Typen als *Selector*en zur Auswahl der Spezialisierung verwendet werden. Diese müssen dafür lediglich deklariert werden:
`struct HMI;` Auf dieser Basis kann eine Spezialisierung einer *type-function* definiert werden.

8.1.9 *type-object*

Templates können mit konstanten Werten erzeugt werden.
Sind die Konstanten die Adressen von Memory Mapped Registern, können die daraus erzeugten Klassen als Objekte betrachtet werden, **`this`** wird durch die Konstante ersetzt. Diese Templates können als Klassen und die Template Argument Liste als der Konstruktor betrachtet werden[232].

8.1.10 Statische Functoren

Templates, die innere Klassen Templates definieren, können als statische Functoren betrachtet und verwendet werden. Die inneren Templates haben zugriff auf die Parameter des umgebenden Templates und können darauf basierend wiederum Typen erzeugen.

[230] siehe section 13.4 auf Seite 217
[231] siehe section 13.4 auf Seite 217
[232] siehe section 13.3 auf Seite 215

8.2 Arten von Templates

- Funktions-Templates
- Klassen-Templates
- Template - Alias (ab C++11)
- Variablen Template (ab C++14)

8.2.1 Variablen Template, C++14

Das Listing 151 zeigt die Definition des Variablen Templates `pi` und dessen Anwendung. Die allgemeine Syntax ist
`template < parameter-list > variable-declaration`
Siehe auch http://en.cppreference.com/w/cpp/language/variable_template.

Listing 151: Variablen Template

```
1 template<class T>
2 constexpr T pi = T(3.1415926535897932385);
3
4 //Anwendung:
5 template<class T>
6 T circular_area(T r) // function template
7 {
8     return pi<T> * r * r; // pi<T> is a variable template instantiation
9 }
```

Variablen Templates könnnen nicht als Argumente für Template Template Parameter verwendet werden.

8.3 Keywords, Expressions und Templates

Dieses Kapitel soll einen Überblick über die wichtigsten Schlüsselworte im Zusammenhang mit Templates geben, sie haben in verschiedenen Kontexten unterschiedliche Bedeutung. Diese wird in den jeweiligen Kapiteln erklärt.

Keywords, Expressions und Templates:

- template
- class / struct
- typedef / using
- typename
- this-> / type::
- decltype(type/object/auto)

8.4 Template Parameter

Template Parameter werden in der Form `template<comma-separated-list>` spezifiziert. Template Parameter können Werte, Typen oder Templates sein:

- Werte / Value (**template value parameter**)
  - Ganzzahlige Konstanten
  - Enumerationen
  - Zeiger mit external linkage
  - Funktionen (Zeiger/Referenzen) mit internal linkage (ab C++11)
- Typen (**template type parameter**)
  - eingebaute Typen **`int`**, **`double`**, **`char`**, `bool`, `...`
  - benutzerdefinierte Typen / Klassen, lokale Typen
  - die Schlüsselworte `typename` und **`class`** sind an dieser Stelle gleichwertig
- Templates (**template template parameter**)
  - benutzerdefinierte Templates
  - Traits und Policies
  - die Kombination der Schlüsselworte template<. . . > class ist notwendig

Für Parameter, die Templates sind, wird der Begriff **template template parameter** verwendet, analog könnte man für Typen **template type parameter** und für Werte **template value paramter** verwenden. Für Werte wird auch häufig *template non type parameter* verwendet [233]. Der Scope der Template Parameter erstreckt sich über die gesamte Definition des Templates.

Seit C++11 sind Templates mit variabler Argumentliste möglich. Die Syntax für den Parameter ist:
`template<class... Parameterpack> class tuple{...}`

8.5 Deklaration von Templates

Listing 152: Template Deklarationen und Parameter

```
1 template <int N>
2 class TemplateWithValueParameter;
3
4 template <typename T>
5 class TemplateWithOneTypeParameter;
6
7 template <template <typename CP> class TP>
8 class TemplateWithTemplateParameter;
```

[233] [VJ03]

```
9
10 namespace std{
11 template<
12    class T,
13    std::size_t N
14 > struct array;
15 }
```

Das Listing 152 auf der vorherigen Seite zeigt ein Template mit einem Template Value Parameter `int N`, ein Template mit einem Template Type Parameter `typename T`, und ein Template mit einem Template Template Parameter `template< typename CP> class TP`.

Bei Template Type Parametern kann anstatt das Schlüsselwort `typename` das Schlüsselwort `class` in der Parameterliste verwendet werden. Diese sind in diesem Kontext gleichwertig[234].

Bei Template Template Parametern muss der "Typ" des Arguments exakt mit dem Parameter passen. Das bedeutet, wenn der Parameter ein Template beschreibt, das einen Typ als Argument erwartet, dann muss auch das als Argument übergebene Template genau einen Typ als Argument erwarten. In dem obigen Bespiel würde das `TemplateWithOneTypeParameter` für den Parameter `TP` passen. Darum wäre folgende Anwendung syntaktisch korrekt:

```
TemplateWithTemplateParameter<TemplateWithOneTypParameter> variable;
```

`TP` ist die Deklaration eines Templates das einen Type als Parameter hat, genau so wie das Template `TemplateWithOneTypeParameter`.

Seit C++11:
Das Template `array` im `namespace std` ist ein Beispiel aus der STL für die Anwendung von type und value Parametern. Die Semantik und Performance ist dieselbe wie ein C-style `struct` das als einzigen Member ein natives Array `T t[N];` deklariert, kombiniert mit den Vorzügen eines STL Containers[235].

8.6 Generische Programmierung

Bei der konventionellen Verwendung von Templates werden diese als Vorlagen für Klassen und Funktionen verwendet. In der generischen Programmierung, wie in Listing 153 auf der nächsten Seite, werden Klassen Templates als *compile time* Funktionen verwendet, die einen Typ (via `typedef ... type;` C++11: `using type = ...;`) oder einen Wert (via `enum {value}`) erzeugen.

Das Beispiel in Listing 153 auf der nächsten Seite soll einen Eindruck dieser Technik vermitteln. Es berechnet die Fakultät von n und soll verdeutlichen, dass eine Berechnung stattfindet, also ein Programm vom Compiler ausgeführt wird, wenn ein Template angewendet wird. Der Ausdruck `Factorial<5>::value` wird vom Compiler berechnet und ist zur Laufzeit eine Konstante.

[234] siehe section 8.8 auf Seite 176
[235] http://en.cppreference.com/w/cpp/container/array

Das Klassen Template `Factorial` mit allen Spezialisierungen muss als eine Funktion, die zur compile time ausgewertet wird, betrachtet werden.

Listing 153: Die compile time Funktion Factorial

```
template<unsigned int n> // primäres Template
struct Factorial {
   enum { value = n * Factorial<n-1>::value};
};

template<> // Spezialisierung für den wert 0
struct Factorial<0> {
   enum { value = 1};
};

main() {
   // Anwendung des Templates
   std::cout << Factorial<5>::value; // prints 120
   std::cout << Factorial<10>::value; // prints 3628800
}
```

Das primäre Template wird vom Compiler angewendet, wenn es für den aktuellen Wert von n, kein spezielles Template gibt. Das primäre Template wendet sich selbst auf n-1 an. Dadurch wird das allgemeine Template so lange angewendet, bis n einen Wert annimmt, für den es ein spezielles Template gibt. Dann wird das spezielle Template angewendet. In diesem Beispiel gibt es nur das spezielle Template für den Wert 0.

Templates sind eine funktionale Sprache! Sprache ist Medium und Werkzeug zugleich! Mit funktionalen Sprachen zu programmieren erfordert eine völlig andere Art zu denken als mit prozeduralen oder objektorientierten Sprachen. Es gibt keine Variablen, nur Konstanten, es gibt keine Zuweisung, keine Schleifen, alles wird als eine Funktion zum Ausdruck gebracht. Das Template `Factorial` mit allen Spezialisierungen ist eine Funktion! Das ist letztlich die größte Hürde, die es bzgl. Templates zu nehmen gilt. Es lohnt sich, sich mit einer anderen funktionalen Sprache, wie z.B.: Haskell, auseinander zu setzen, um den Einstieg in diese Welt zu erleichtern[236].

Durch die Einführung von `constexpr` Funktionen hat die Bedeutung solcher Templates als *compile time functions* zur Berechnung von Werten stark abgenommen, weil diese mit konventioneller Syntax und `constexpr` Funktionen erstellt werden können. Typen können aber nur auf diese Weise erstellt werden.

8.7 Templates und die UML

Das Diagramm 7 auf der nächsten Seite und das Listing 154 auf Seite 176 stellen denselben Sachverhalt dar, einmal in UML und einmal in C++.

[236] http://de.wikipedia.org/wiki/Haskell_(Programmiersprache)

Bei der konventionellen Verwendung von Templates, werden diese als Vorlagen für Klassen und Funktionen verwendet werden. Ein Template (`MyTemplate`) wird in der UML durch eine parametrisierte Klasse mit den Parametern (hier nur `T`) rechts oben in einem Rechteck dargestellt.

Der Begriff *Instance* eines Templates wird für eine Klasse die aus einem Template erzeugt wurde, verwendet[237]. Die *Instantiation* des Templates ist die Erzeugung einer Klasse aus einem Template. Die Instanziierung von Klassen aus dem Template wird in der UML durch eine gestrichelte Linie mit einer unausgefüllte Pfeilspitze[238] dargestellt, die mit dem Stereotype `<<bind>>` versehen ist. Das Binding der Parameter an die Argumente wird in spitzen Klammern an die Linie angetragen:
z.B. `<T->MyType>`. In C++ wird aus einem Satz von gleichen Argumenten jeweils ein eigener Typ erzeugt.

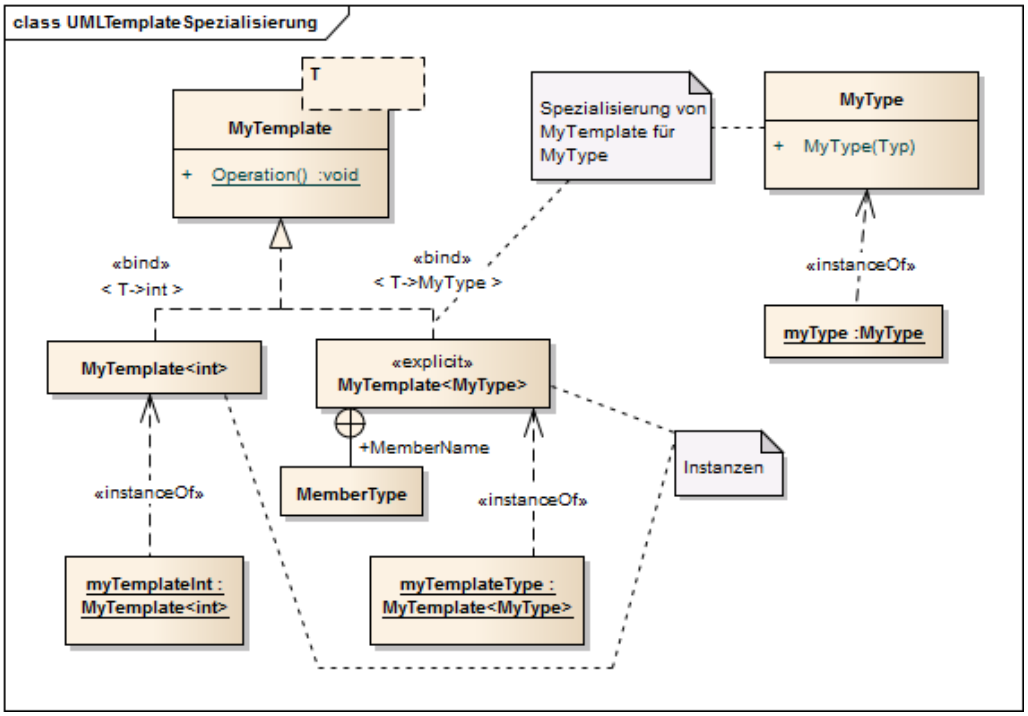

Diagramm 7: Templates und die UML

Durch die explizite Spezialisierung eines Templates[239] kann die Erzeugung der Instanz abweichend vom primary Template beeinflußt werden. In Listing 154 auf der nächsten Seite ist das Template `MyTemplate<..>` für das Argument `MyType` explizit spezialisiert. In diesem Zusammenhang wird die *generierte Spezialisierung* im Falle von `myTemplateInt` und die *explizite Spezialisierung* im Falle von `myTemplateType` unterschieden[240]. Die explizite Spezialisierung wird in Diagramm 7 mit dem Stereotyp `explicit` dargestellt.

Die Klasse `MyTemplate<MyType>` hat eine *member type definition* `typedef MemberType MemberName` die in der UML als nestet Associations modelliert werden. Der Name

[237] [VJ03] siehe section 9.2 auf Seite 184
[238] wie `<<implements>>`
[239] siehe section 9.3 auf Seite 185
[240] [Str13] S. 742 26.2 Template Instantiation

wird als Rollenname angetragen.

Listing 154: Die Instanzen eines Templates: explizite und generierte Spezialisierung

```
// primary Template
template<class T>
struct MyTemplate {...};

struct MyType{...};
MyType myType;

// die Spezialisierung MyTemplate<int> wird vom Compiler generiert
MyTemplate<int> myTemplateInt; // Variable vom Typ MyTemplate<int>

struct MemberType{...};

//explizite Spezialisierung
template<>
struct MyTemplate<MyType>{
   typedef MemberType MemberName; // member type definition
   // C++11 alternativ
   using MemberName = MemberType; // member type definition
};

// die Spezialisierung MyTemplate<MyType> wird vom Compiler ausgewählt
MyTemplate<MyType> myTemplateType;
```

In Diagramm 7 auf der vorherigen Seite sind die Variablen `myTemplateInt, myTemplateType` der *Instanzen* von `MyTemplate<..>` und die Variable `myType` der Klasse `MyType` dargestellt.

Es existieren drei Ebenen des Modells:

1. Templates
2. Klassen
3. Objekte

Das entspricht der “üblichen” Verwendung von Templates: Ein Typ, z.B. `Stack<T>`, wird abstrakt durch ein Template beschrieben. Wird für einen bestimmten Typ, z.B. **`int`**, ein Stack benötigt, wird aus dem Template durch Binding `<T->int>` des Parameters die dafür notwendige Klasse erzeugt. Diese kann zur Definition der benötigten Variablen z.B. `Stack<int> intStack;` benutzt werden.

8.8 Das Schlüsselwort `typename`

`typename` wurde im Standard eingeführt um einen Namen innerhalb eines Templates, als den Namen eines Typs spezifizieren zu können. Generell werden alle Namen innerhalb eines Templates als Wert / Value angesehen, außer sie werden

mit dem Schlüsselwort `typename` für Typen oder `template` für Templates[241] qualifiziert. `typename` darf nur innerhalb von Templates verwendet werden.

Listing 155: Das Schlüsselwort `typename`

```
template <class T>
class Widget{
   typename T::SubType * ptr;
   ...
}
```

Ohne `typename` ist in Listing 155 nicht klar, was die Anweisung `T::SubType * ptr;` ist. Es könnte die Multiplikation eines Klassenelements (**`static`**) von `T` mit dem Namen `SubType` mit dem Objekt `ptr` sein. Durch das Schlüsselwort `typename` wird klargestellt, dass `T::SubType` ein Typ ist und damit `ptr` ein Objekt vom Typ „Zeiger auf ein `T::SubType` “ ist[242].

Widget kann nur auf Typen angewendet werden, die einen entsprechenden SubType definieren. Bsp.:

Listing 156: Beispiel: SubType

```
Widget <Q> q;// funktioniert nur wenn Q wie folgt definiert ist
class Q{
   typedef int SubType; // ptr wäre int*
   //oder auch
   class SubType; //ptr wäre SubType*
   ...
};
```

Bei der Deklaration von Template Type Parametern `template<class T>` kann alternativ zum Schlüsselwort **`class`** das Schlüsselwort `typename` verwendet werden. Die Verwendung von `typename` für alle Typen, also Klassen und eingebaute Datentypen, und die Verwendung von **`class`**, wenn nur benutzerdefinierte Typen als Template Argumente verwendet werden können, wird als Konvention von verschiedenen Autoren vorgeschlagen[243].

8.9 Verwendung von `this->` und `::`

Der Scope von Namen innerhalb einer Klasse ist eingebettet in den Scope der Basisklasse[244]. Daher kann auf Member einer Klasse oder auf **`protected`** oder **`public`** Member der Basisklasse mit dem Namen zugegriffen werden wie in Listing 157 auf der nächsten Seite.

[241] section 9.9.1 auf Seite 192
[242] [VJ03] Chapter 5
[243] z.B. [Ale09] Seite 107 Fußnote
[244] section 6.20 auf Seite 60

Listing 157: Verwendung von **this** in Klassen

```
// global Function
void operation();

class Base{
   ...
protected:
   void operation();
   ...
};
class Derived : public Base{
   ...
   void Op(){
      operation(); // calls Base::operation()
      this->operation(); // calls Base::operation()
   }
};
```

Bei Templates ist das nicht der Fall. Der Name muss voll qualifiziert werden, entweder durch das Schlüsselwort **this**-> oder mit dem Scope Resolution Operator `Base::` wie in Listing 158.

Listing 158: Verwendung von **this** und `::` in Templates

```
// global Functions
void operation();
void classOperation();

template<typename T>
class Base{
   ...
protected:
   static void classOperation();
   void operation();
   ...
};
template<typename T>
class Derived : public Base<T>{
   typedef Derived<T> this_type;
   ...
   void Op(){
      operation(); // calls global operation()
      classOperation(); // calls global operation()
      this->operation() // calls Base::operation()
      this->classOperation(); // calls Base::classOperation()
   }
   static void classOp(){
      classOperation(); // calls global operation()
      this_type::classOperation() // calls Base::classOperation()
   }
};
```

In Templates sollte deshalb immer der voll qualifizierte Name verwendet werden.

8.9.1 Injected Class Names in Templates

Das Listing 159 zeigt die Verwendung des template Namens `C`, einmal als vereinfachter Name der Klasse `C<T>` und einmal als Name des Templates. Je nach dem in welchem Kontext der Name verwendet wird. Das ist etwas verwirrend und gewöhnungsbedürftig[245].

Empfehlung: Immer den vollständigen Namen der Klasse verwenden, wo die aus dem Template instanziierte Klasse gemeint ist, z.B. der Parameter Typ des Kopiekonstruktors, Movekonstruktors oder bei der Deklaration von Zeigern oder Referenzen.

In Listing 159 wird zuerst das Template `gh::C` deklariert. Dann das Primary Template Widget definiert, das ein Template mit einem Typ Parameter als Argument erwartet. Dieses wird anschließend für `gh::C` spezialisiert!

In der Implementierung von `C` wird `Widget` verwendet, wobei in `gh::C` die Spezialisierung und im globalen `C` das Primary Template von `Widget` herangezogen wird, was in der Ausgabe zu erkennen ist.

Im globalen template `C` wird auf den Namen des Templates einmal ohne und einmal mit dem Scope Resolution Operator `::` zugegriffen, das Leerzeichen nach der spitzen Klammer auf ist notwendig, da `<:` ein sogenanntes digraph token als Alternative für die eckige Klammer auf (`[`) ist.

Listing 159: Injected Class Names und Templates

```
1  // forward declaration
2  namespace gh{
3     template<class T> class C;
4  }
5
6  //primary template
7  template<template<class> class TT>
8  struct Widget{
9  public:
10    void operation(){
11       cout << "Widget<TT>" << endl;
12    }
13 };
14 //spezialisierung for Template gh::C
15 template<>
16 struct Widget<gh::C>{
17    void operation(){
18       cout << "Widget<gh::C>" << endl;
19    }
20 };
21
```

[245] [VJ03] 9.2.3 Injected Class Names

```
22 namespace gh{
23
24 template<class T> class C{
25    // Achtung: Name des Templates C entspricht hier dem Namen der Klasse C<T>
26    C *pC1;     // ok, entspricht aber: C<T> *pC;
27    C<T> *pC2;     // dasselbe wie pC1
28
29    // Achtung: Name des Templates C entspricht hier dem template C
30    Widget<C> w1;
31    Widget<gh::C> w2; // dasselbe wie w1
32 public:
33    C() : pC1(this){ pC2 = this; } // default Ctor
34    C(C const&) {/* copy Ctor entspricht: C(C<T> const& rhs) */}
35    void operation(){
36       cout << "gh::C<T>::operation()" << endl;
37       w1.operation(); // spezialisierung wird verwendet
38    }
39 };
40
41 } // end namespace
42
43 template<class T> class C{
44    Widget<C> w1;
45    Widget< ::C> w2; // dasselbe wie w1
46    //Widget<::C> w2; // error: '<::' cannot begin a template-argument list
47 public:
48    void operation(){
49       cout << "C<T>::operation()" << endl;
50       w1.operation(); // primary Template wird verwendet
51    }
52 };
53
54 int main(){
55    cout << "Injected Class Name" << endl;
56
57    cout << "gh::C<void> c1;" << endl;
58    gh::C<void> c1;
59    cout << "c1.operation();" << endl;
60    c1.operation();
61    gh::C<void> c1a(c1); // copy Ctor
62
63    cout << "C<void> c2;" << endl;
64    C<void> c2;
65    cout << "c2.operation();" << endl;
66    c2.operation();
67 }
68 //Ausgabe
69 Injected Class Name
70 gh::C<void> c1;
71 c1.operation();
72 gh::C<T>::operation()
```

```
73 Widget<gh::C>
74 C<void> c1;
75 c2.operation();
76 C<T>::operation()
77 Widget<TT>
```

8.10 Indirektion, die Lösung aller Probleme

TODO: Hier die verschiedenen Möglichkeiten statische Indirektion zu realisieren.

- `Type or template with InnerTemplate`
- `template<T1, T2, ..> using templateName = OtherTemplate<T1, Default, T2 >`
- `using typeName = type or templateInstance;`
- `using NS = NamespaceName;`

9 Klassen Templates

9.1 Definition eines Klassen Templates Stack

Stack[246] ist eine der typischen Datenstrukturen, die immer wieder für verschiedenste Typen benötigt wird. Durch das Template kann die dafür notwendige Implementierung unabhängig von einem konkreten Typ definiert werden und bei Bedarf, wenn das Template angewendet wird, mit dem Typ, der als Argument übergeben wird, ergänzt werden. Dieses Beispiel dient dazu die Syntax zur Definition von Templates und der Member darzustellen.

In Listing 160 wird entsprechend der in section 8.8 auf Seite 176 beschriebenen Konvention `typename` für die Parameter Deklaration verwendet, da alle Typen als Argument für das Template verwendet werden können.

Der prefix `template<typename T>` oder alternativ `template<class T>`, spezifiziert, dass ein Template deklariert oder definiert wird und dass innerhalb der Definition `T` als Typ benutzt wird. Nach der Einführung des Template Parameternamens `T`, kann er als der Name eines Typs benutzt werden. Der Scope der Template Parameter erstreckt sich über die gesamte Definition des Templates.

Listing 160: Definition des Klassen Templates Stack

```
1  template<typename T>
2  class Stack {
3  private:
4     std::vector<T> elems;
5  public:
6     void push(T const& element){ elems.push_back(element); }
7     void pop();
8     T const& top() const;
9     //... weitere Operationen
10 };
```

Die Methodenkörper der Operationen sind diesem Beispiel zum Teil weggelassen und werden außerhalb der Templatedefinition definiert.

Die Stack Klasse ist mit `std::vector` implementiert. Darum wird kein Konstruktor benötigt. Zuweisen, Kopieren erfolgt alles korrekt durch die vom Compiler erzeugten Methoden, die die Methoden der verwendeten Klassen (`std::vector`) aufrufen.
Ein schönes Beispiel [247] für eine „has-a“ Beziehung in der Problem Domain oder „is-implemented-in-terms-of“ Beziehung auf der Implementierungsebene, die durch private Ableitung implementiert werden muss, wenn Zugriff auf protected member der Implementierungsklasse (hier: `std::vector`) benötigt wird. Die dafür notwendige Definition ist in Listing 161 auf der nächsten Seite abgebildet.

[246] [VJ03] Chapter 3
[247] [Mey06a] Item 38, 39

Listing 161: Definition Klassen Template mit private Vererbung

```
template <typename T>
class Stack : private  vector<T>{
   /* hier besteht zurgriff auf die protected Member von vector*/
   ...
}
```

9.1.1 Out of template member definition

Das Listing 162 zeigt zwei Methodendefinitionen für das Klassen Template `Stack`. Der Rückgabetype von top ist `T`, der nicht spezifizierte Typ, der erst bei Anwendung des Templates angeben wird. Der Typ der Klasse ist `Stack<T>`. Damit wird, wie bei normalen Klassen, die Zugehörigkeit der Methode zu diesem Template angeben.

Listing 162: Out of template member Definition

```
inline               // inline Aufforderung
template<typename T> // Template Parameter
T                    // return value
Stack<T>::           // Template-ID
top() const          // Methoden Kopf
{                    // Methoden Körper
   if(elems.empty())
      throw out_of_range("empty Stack");
   return elems.back();
}
inline
template<typename T>
void Stack<T>::pop(){
   elems.pop_back();
}
```

Die Parameterliste muss vollständig wiederholt werden und die Template-Id muss auf der Basis der Parameternamen angegeben werden.

9.1.2 Konstruktoren und Assignment Operator

Listing 163: Konstruktoren und Assignment Operator

```
template<typename T> class Stack {
   ...
   Stack();
   Stack(Stack<T> const&)
   Stack<T>& operator=(Stack<T> const&);
   ...
};
```

Beim Kopiekonstruktor und Assignmentoperator in Listing 163 auf der vorherigen Seite muss der Übergabeparameter `Stack<T>`, der Typ der Klasse sein. Für den Namen des Konstruktor muss der Name des Templates (`Stack`) verwendet werden.

9.2 Anwendung des Klassen Templates Stack, Template Instanziierung, Template-Id

Bei der Verwendung eines Templates werden die Template-Parameter durch die Template-Argumente ersetzt und daraus eine Klasse oder eine Funktion[248] erzeugt. Dieser Vorgang wird als *template-instantiation* bezeichnet. Das Ergebnis dieses Vorgangs ist eine Klasse.

Die Definition des Templates und alle Spezialisierungen müssen vor ihrer Verwendung für den Compiler sichtbar sein, das bedeutet, die Header, in denen das Template und die Spezialisierungen definiert sind, müssen in dem Header oder der Übersetzungseinheit (cpp), in der sie verwendet werden, `#include`d sein.

Für jede Kombination von Template-Argumenten wird jeweils eine eigene Template-Instanz, eine eigene Klasse, erzeugt. Der Parameter `T` in Listing 164 ist ein *template type parameter*. Es werden insgesamt vier Klassen aus dem Klassen Template `Stack` erzeugt.

Der Name eines Klassen Templates gefolgt von der Argumentliste in spitzen Klammern `Stack<...>` ist die Template-Id der Klasse. Sie kann wie der Name einer “normalen"Klasse benutzt werden. Die Identität der daraus entstehenden Klasse ist der Templatename mit den ersetzten Parametern, die Typ Argumente, hier **`int`**, **`double`**, `Widget` und `std::string` für den Parameter `T`.

Listing 164: Anwenden des Klassen Templates Stack

```
1  voidf(){
2     Stack<int> intStack; // Template-Id == Stack<int>
3     Stack<double> doubleStack; // Template-Id == Stack<double>
4
5     class Widget{...};
6     Stack<Widget> widgetStack; // Template-Id == Stack<Widget>
7
8     Stack<std::string> stringStack; // Template-Id == Stack<std::string>
9     intStack.push(5);
10    cout << intStack.top() << endl;
11 ...
12 }
```

Für Templates wird nur Code für die benutzten Member erzeugt. Das Binary ist entsprechend kleiner, als bei einer Klasse, wenn nicht alle Member benutzt werden. Außerdem können Typen als Argumente verwendet werden, die keine vollständige Schnittstelle zur Verfügung stellen, wenn keine Template Methoden verwendet werden, die diesen Teil der Schnittstelle verwenden.

[248] siehe section 10 auf Seite 197

9.3 Spezialisierung des Klassen Templates

9.3.1 Begriffe

Bei der *expliziten Spezialisierung* eines Klassen Templates wird bei der Deklaration oder Definition der Spezialisierung eine Argumentliste angegeben, für die die Spezialisierung des Templates verwendet werden soll, wenn in der Argumentliste bei der Anwendung diese Kombination von Argumenten verwendet wird.
Von *generierter Spezialisierung* wird gesprochen, wenn der Compiler bei der Anwendung eines Templates mit Argumenten (*template-instantiation*), aus dem Template eine konkrete Klasse erzeugt.

Listing 165: Template Parameter und Argument Liste

```
1 template<parameter-liste> class Widget<argument-liste> {/*definition*/};
```

9.3.2 Primary Template

Als **primary template** wird das Template bezeichnet, das keine Argumentliste hat. `template<typename T> class Widget{...}`.
Das Primary Template wird vom Compiler in allen Fällen angewendet, für die es keine passende explizite Spezialisierung gibt. Er erzeugt dabei eine sogenannte *implizite Spezialisierung*. Das primary Template muss vor allen Spezialisierungen deklariert werden. Die Definition des primary Templates kann weggelassen werden, wenn es keine sinnvolle generelle Implementierung gibt.

Listing 166: Primary Template

```
1 // Deklaration ohne Definition für das Primary template
2 template <typename T> struct MyTemplate;
3
4 // Mit Definition {...} des Primary Templates
5 template<class T1, class T2>
6 struct Widget{...}; // Mit Definition
```

9.4 Vollständige Spezialisierung des Klassen Templates

Bei der vollständigen Spezialisierung eines Klassen Templates, werden für alle Parameter die Argumente festgelegt. Die Parameterliste bleibt leer (`template<>`), in der Argumetliste werden die speziellen Argumente (Typen, Werte) festgelegt (`class Stack<std::string>{...}`). Wird das Template mit diesen Argumenten angewendet, wird die *explizite Spezialisierung* vom Compiler ausgewählt.

Listing 167: Vollständige Spezialisierung des Klassen Templates

```
template<>              // leere Parameterliste
class Stack<std::string>// Argumentliste
{                       //Definition
   // hier kann eine vollständig vom primären Template
   // verschiedene Implementierung erfolgen
   void push(std::string const& element);
   ...
};
```

Die vollständige Spezialisierung ist nur global oder in Namespaces möglich, aber nicht als Member einer Klasse oder eines Klassentemplates.

9.4.1 out of class Member Definition

Member von spezialisierten Templates können außerhalb der Template Definition wie in Listing 168 definiert werden:

Listing 168: Definition Member vollständige Spezialisierung

```
template<>
void Stack<std::string>::push(std::string const& element){
   container.push_back(element);
}
```

Die Parameterliste bleibt leer und die Template-Id muss auf der Basis der spezialisierten Argumente angegeben werden.

9.5 Partielle Spezialisierung

Werden nicht für alle Parameter die Argumente festgelegt, wird von partieller Spezialisierung gesprochen. Die Parameterliste enthält nur die Parameter für die nicht spezialisierten Argumente. Die Position der spezialisierten Argumente in der Argumentliste bestimmt den spezialisierten Parameter.

Listing 169: Partielle Spezialisierung

```
// Primary Template
template<typename T1, typename T2> class Widget {...};
// spezialisierung für gleiche Argumente
template<typename T> class Widget<T, T>{...};
// spezialisierung für T, int
template<typename T> class Widget<T, int>{...};
// für Pointer
template<typename T1, typename T2> class Widget<T1*, T2*>{...};
void f(){
   Widget<int, float> wif; // benutzt Widget<T1, T2>
   Widget< float, float> wff; // benutzt Widget<T, T>
   Widget< float, int > wfi; // benutzt Widget<T, int>
```

```
13    Widget< float*, int* > wfpip; // benutzt Widget<T1*, T2*>
14
15    /* error: ambiguous class template instantiation for
16     * 'class Widget<float*, float*>'
17     * ../Widget.h:20:28: error: candidates are: class Widget<T, T>
18     * ../Widget.h:32:41: error:                 class Widget<T1*, T2*>
19     */
20    Widget< float*, float* > wfpfp;
21 }
```

Bei der Spezialisierung kann es zu Mehrdeutigkeiten kommen.
Der Versuch `Widget< float*, float* >` aus Listing 169 auf der vorherigen Seite zu instanziieren, wird vom Compiler mit der Fehlermeldung `error: ambiguous class template instantiation` quitiert. Die Auflösung der Mehrdeutigkeit kann auf verschiedene Weise erfolgen. In diesem Fall könnte ein Template `template<T > class Widget<T*, T*>{...}` definiert werden, das die dafür passende Implmentierung zur Verfügung stellt.

9.5.1 out of class Member Definition

Member von partiell spezialisierten Templates können außerhalb der Template Definition wie in Listing 170 definiert werden:

Listing 170: Definition Member partielle Spezialisierung

```
1 template<typename T>
2 void Widget<T, int>::print(){ std::cout << "class Widget<<T, int>" << std::endl; }
```

Die Parameterliste enthält nur die nicht spezialisierten Parameter und die Template-Id muss auf der Basis der Parameternamen und der spezialisierten Argumente angegeben werden.

9.6 Spezialisierung mit Templates

Das Listing 171 zeigt die Spezialisierung für ein Template, dessen Argumente nicht festgelegt sind, sondern in der Parameterliste aufgeführt werden. Damit besteht die Möglichkeit, das Template aus Listing 169 auf der vorherigen Seite, das als Primary Template zwei Paramter hat, mit einem weiteren Template aufzurufen und der Spezialisierung mehr Typen zu übergeben, als das Primary Template erwartet. Die Anzahl der Template Argumente ist weiterhin zwei: `Another<T1, T2>` und `T3`.

Listing 171: Spezialisierung für ein Template

```
1 template<class T1, class T2>
2 struct Another;
3
4 //Spezialisierung für Another<T1, T2>, T3
5 template<class T1, class T2, class T3>
```

```
6  struct Widget<Another<T1, T2>, T3>{...};
7
8  //out of class Member Definition
9  template<class T1, class T2, class T3>
10 void Widget<Another<T1, T2>, T3>::operation(){...}
11
12 //Anwendung
13 Widget<Another<int, double>, char> variable;
```

Im Listing 171 auf der vorherigen Seite ist Another nur deklariert, nicht definiert. Die Template Argumente stehen nicht fest. Nur das erste Argument ist mit `Another<T1, T2>` spezialisiert. Bei der Anwendung des Templates muss dann als erster Parameter `Another` angegeben werden. `Another` muss aber nicht definiert sein. Die Typen T1 bis T3 stehen in der Spezialisierung von `Widget<...>` zur Verfügung.

In Listing 172 wird diese Technik mit Variadic Templates (C++11) angewandt, um eine Liste von Typen `typelist` einem Template als Argument zu übergeben. Das Template `Typelist` ist ohne Definition und wird nur zur Spezialisierung des Templates (`Widget`) benötig, das eine Liste von Typen verarbeiten können soll.

Listing 172: Typelist, einen Parameterpack als Template Argument

```
1  template<class ...Types> struct Typelist; // without any definition
2
3  class A; class B; class C;
4
5  using typelist = Typelist<A, B, C>;
6
7  //primary Template without definition
8  template<class T> class Widget;
9
10 template<class Base>
11 struct Printer : Base{
12    using base_type = Base;
13
14    Printer(){ base_type::print(); }
15 };
16
17 template<class...Types>
18 class Widget<Typelist<Types...>> : Printer<Types>...
19 { };
20
21 //Typedefinitions
22 class A{
23 public:
24     static void print(){ std::cout << "A::print" << std::endl; }
25 };
26 class B, C { dto. };
27
28 void demoTypelist(){
29    //usage:
```

```
30    Widget<typelist> widget;
31 }
```

9.7 Default Template Argumente

Default Template Argumente können für Klassen-Templates und seit C++11 auch für Funktions-Templates angegeben werden. Das Template Stack aus Listing 160 auf Seite 182 nutzt zur Verwaltung der Objekte den Container `std::vector<T>`. Damit die Nutzer des Stacks die Möglichkeit haben, den Containter zur Verwaltung der Objekte selbst zu bestimmen, wird das Template mit einem weiteren Parameter, dem Typ zur Verwaltung der Objekte, ausgestattet.
`template<typename T, class Container> class Stack{...}`. Bei der Verwendung des Stacks muss damit aber jedesmal die Klasse des Containers angegeben werden. Mit Default Argumenten kann das Template Stack verwendet werden wie vorher, aber der Nutzer hat die Möglichkeit, den Container bei Bedarf selbst zu bestimmen.

Das default Argument in Listing 173 ist `std::deque` wie in der STL für den Parameter `Container`. Wird bei der Verwendung des Templates nur ein Typ angegeben, wird der zweite Template Parameter `Container` ein `std::deque<T>` sein.

Listing 173: Beispiel: Default Template Argumente

```
1 template <class ElementType, class Container = std::deque<ElementType> >
2 class Stack{
3    Container container;
4 ...
5 };
```

Bsp.: `Stack<int> intStack;`
ist äquivalent zu: `Stack<int, std::deque<int> > intStack;`
Die default Argumente können mit Bezug auf vorherige Parameter (`ElementType`) definiert werden. Soll ein anderer Typ von Container verwendet werden, muss das Template Argument zweimal angegeben werden:
`Stack<int, std::vector<int> > intStack;`

9.7.1 out of class Member Definition

Member von Templates mit mehreren *template type parametern* können außerhalb der Templates Definition wie in Listing 174 definiert werden:

Listing 174: Definition Member mit mehreren template type parametern

```
1 template<typename ElementType, typename Container>
2 void Stack<ElementType, Container>::push(ElementType const& element){
3    container.push_back(element);
4 }
```

Die Parameterliste muss vollständig wiederholt werden und die Template-Id muss auf der Basis der Parameternamen angegeben werden.

9.8 Template Template Parameter

Die Wiederholung des Template Parameter `ElementType` kann vermieden werden, indem dem Template als Parameter ein Template übergeben wird und nicht eine Klasse. Der Template Parameter ist also selbst ein Template! Ein *Template Template Parameter*. In Listing 175 ist die Definition eines solchen Klassen Templates skizziert.

Listing 175: Template Template Parameter

```
template <  typename ElementType,
            template <typename> class Container = std::deque >
class Stack{
   Container <ElementType> container;
...
};
```

Da `Container` jetzt ein Template ist, muss bei der Instanzierung des Templates in der Klasse `Stack` ein Typ als Argument übergeben werden:
`Container<ElementType> container`.

Bis C++17: **Das Schlüsselwort `class` vor dem Parameternamen `Container` kann hier nicht** durch das Schlüsselwort `typename` **ersetzt werden**, da `Container` die Deklaration eines Klassen Templates sein muss.
Das wurde mit C++17 aufgehoben:
`template<template<typename>typename name> class templateName{/*definition*/};`

Beispiel Deklaration eines Klassen Templates Container:
`template<typename ElementType> class Container;` die Parameterdeklaration entspricht genau dieser Deklaration. Der Parameter `ElementType` wird in der Parameterliste von `Stack` nicht benutzt und kann daher weggelassen werden:
`template<typename> class Container`.
Übersetzen lässt sich das aber noch nicht, weil `std::vector` mehr als einen Template Parameter hat.

Die Version die übersetzbar sein sollte ist in Listing 176 abgebildet.

Listing 176: Template Stack mit template template parameter

```
template<typename ElementType,
         template<typename T, typename = std::allocator<T> >
         class Container = std::vector >
class Stack{
   Container <ElementType> container;
public:
   void push(ElementType const&);
   ...
};
```

Damit ist `Container` als ein Template mit 2 *template type parametern* deklariert und passt mit `std::vector` zusammen. Die *template template argumente* müssen exakt mit den *template template parametern* übereinstimmen.

Default Argumente der *template template argumente* werden nicht berücksichtigt, können aber in der Parameterliste (`typename = std::allocator<T>`) angegeben werden.

Das Konzept der *template template parameter* ist wenig flexibel, da die Argumente mit dem Parameter exakt übereinstimmen müssen. Besser ist es, einen Typ zu übergeben, einen generischen Functor[249], der ein Membertemplate definiert, das ein weiteres Template anwendet. Wenn dieses eine abweichende Anzahl Parameter hat, können diese durch den Functor ergänzt bzw. weggelassen werden.

9.8.1 out of class Member Definition

Member von Templates mit *template template parametern* können außerhalb der Template Definition wie in Listing 177 definiert werden:

Listing 177: Definition Member mit template template parametern

```
1 template<typename ElementType, template<typename, typename> class Container>
2 void Stack<ElementType, Container>::push(ElementType const& element){
3    container.push_back(element);
4 }
```

Die Parameterliste muss vollständig wiederholt werden und die Template-Id muss auf der Basis der Parameternamen angegeben werden.

9.9 Member Template Class

Sowie Klassen und Klassen Templates in Klassen definiert werden können, können auch Klassen und Klassen Templates in Klassen Templates definiert werden. Werden Klassen Templates in Klassen Templates definiert, besteht Zugriff auf die Template Parameter des umgebenden Templates.

Damit ist es möglich, ein Template mit vorkonfigurierten Parametern zur Verfügung zu stellen, ähnlich wie es mit der `using` Definition ab C++11 möglich ist[250].

In Listing 178 auf der nächsten Seite soll die dafür notwendige Syntax vorgestellt werden. Bei 1 wird ein Template `Implementation` definiert, das einfach seine Parameter durch die `typedef`s zur Verfügung stellt.

Das `OuterTemplate` bei 2 hat einen Parameter `OuterParamType` und ein inneres Class Template `InnerTemplate` bei 3 das ebenfalls einen Parameter `InnerParamType` hat. Beide stehen in `InnerTemplate` zur Verfügung und werden als Argument für das Template `Implementation` genutzt.

[249] section 13.5.1 auf Seite 224
[250] siehe section 9.10 auf Seite 193

Listing 178: Member Class Template und Zugriff auf Parameter

```
1  template<typename T1, typename T2>
2  class Implementation{   // 1
3  public:
4     typedef T1 FirstType;
5     typedef T2 SecondType;
6  };
7  //================================================
8  template <class OuterParamType>
9  class OuterTemplate{ // 2
10 public:
11    template <class InnerParamType>
12    class InnerTemplate{ // 3
13    public:
14       typedef Implementation<OuterParamType, InnerParamType> type;
15    };
16 };
17 //================================================
18 template<class TemplateWithMemberTemplate, class MyParam>
19 class UsingMemberTemplate{ // 4
20 public:
21    typedef typename TemplateWithMemberTemplate::template InnerTemplate<MyParam>::
          type type;
22 };
```

In Listing 179 wird

- `var11` den Typ **`int`** haben
- `var12` den Typ **`double`** haben
- `var21` den Typ **`int`** haben
- `var22` den Typ **`char`** haben

Listing 179: Anwendung des Member Class Templates

```
1  void demoMemberClassTemplate(){
2     typedef OuterTemplate<int> Outer;
3     typedef Outer::InnerTemplate<double>::type Type1;
4     Type1::FirstType var11{};
5     Type1::SecondType var12{};
6
7     typedef UsingMemberTemplate<Outer, char>::type Type2;
8     Type2::FirstType var21{};
9     Type2::SecondType var22{};
10 }
```

9.9.1 Das Keyword type::template

Das Template `UsingMemberTemplate` in Listing 178 bei 4 zeigt die notwendige Syntax wenn innerhalb eines Templates auf ein Membertemplate eines anderen Typs

zugegriffen werden soll. Das Schlüsselwort `template` ist hier ähnlich wie `typename` [251]. Es stellt klar, dass der Name `InnerTemplate` nicht einen Wert und nicht einen Typ, sondern ein Template adressiert.

9.10 Template Aliasse

9.10.1 Template Aliasse und das Schlüsselwort `using` ab C++11

Mit dem Keyword `using` kann wie mit `typedef` ein Alias Name für einen Typ vereinbart werden: `using myType = int;`. Darüber hinaus ermöglicht `using` die Vereinbarung eines Alias Namens für ein Template, bei dem für bestimmte Parameter bereits Argumente eingesetzt sind, wie in Listing 180. Für das Alias Template können wie für Klassen Templates, Default Argumente angegeben werden `template< class T = char> ...`. Ist das Template, für den ein Alias definiert ist, für bestimmte Argumente spezialisiert, wird diese Spezialisierung ausgewählt, wenn der Alias mit entsprechenden Argumenten angewendet wird.

Listing 180: Template Alias und das Schlüsselwort using

```
1  //primary Template
2  template<class T1, class T2>
3  class MyTemplate;
4
5  //Spezialisierung
6  template<class T1>
7  class MyTemplate<T1, double>;
8
9  //Aliasname
10 template<class T = char>
11 using AndererName = MyTemplate<int, T>;
12
13 // Spezialisierung MyTemplate<T1, double> wird ausgewählt
14 AndererName<double> var1; // Template-Id: MyTemplate<int, double>
15
16 // primary Template MyTemplate wird verwendet
17 AndererName<> var2; // Template-Id: MyTemplate<int, char>
```

Aliasse die mit `using` deklariert werden, können nicht spezialisisert werden[252].

9.10.2 Template Aliasse mit Vererbung vor C++11

Mit Vererbung kann fast derselbe Effect wie mit `using` Aliassen erzielt werden.

Für das Template `Widget` in Listing 181 auf der nächsten Seite sind zwei Alias Namen deklariert. Die Namen der Aliasse sind in diesem Beispiel durch die Imple-

[251] siehe section 8.8 auf Seite 176
[252] [Gri12] S.206 9.3 Aliase Templates

mentierungstechnik der Aliasse bestimmt und dienen nur der Veranschaulichung des Mechanismus.

Das Template `Widget` in Listing 181 ist für **`double`** und **`char`** spezialisiert.

Listing 181: Template Aliasse mit Vererbung

```
template<typename T=int>
struct Widget{
   void print() {
      cout << "Widget<T=int>: " << typeid(T).name() << endl;
   }
};
template<>
struct Widget<double>{
   void print() { cout << "Widget<double>" << endl;}
};
template<>
struct Widget<char>{
   void print() { cout << "Widget<char>" << endl;}
};

template<typename T=double>
struct AliasInheritWidget : Widget<T>{};

template<>
struct AliasInheritWidget<int>{
   void print() { cout << "AliasInheritWidget<int>" << endl;}
};

template<typename T = char>
using AliasUsingWidget = Widget<T>;

template<class T>
void print(T const & t){
   cout << "print(T const & t): ";
         t.print();
}
template<class T>
void print(Widget<T> const & t){
   cout << "print(Widget<T> const & t): ";
   t.print();
}
```

Mit der “Alias” Technik via Vererbung und Delegation, können Aliasse spezialisiert werden.

Das Template `AliasInheritWidget` hat ein Default Argument und delegiert durch Vererbung an das Template `Widget`. Der Effect ist derselbe wie mit dem Schlüsselwort `using` bei dem Template `AliasWidgetChar`. Dieses kann aber nicht spezialisiert werden.

Die Spezialisierung `AliasInheritWidget<int>` erbt nicht von `Widget<T>` sondern

stellt eine andere Implementierung bereit.

Es gibt einen kleinen Unterschied zwischen Aliassen die mit `using` deklariert werden und Aliassen, die via Vererbung definiert werden: Aliasse die via Vererbung definiert werden, definieren einen neuen Typ, der von dem ursprünglichen Template erbt, das ist bei Aliassen via `using` nicht der Fall, diese haben denselben Typ, der durch die Anwendung des originalen Templates entstehen würde.
Daher wird die Überladung des Funktions Templates `print(T const & t)` verwendet, wenn die Funktion mit einem `AliasInheritWidget<..>` Objekt aufgerufen wird aber die Überladung `print(Widget<char> const& t)` für `print(aliasChar)`, einem Objekt des Alias `AliasUsingWidget<>`.
Fehlt die Überladung für `print(T const &)`, führt der Compiler einen impliziten *upcast* durch und ruft `print(Widget<T> const & t)` für Objekte von `AliasInheritWidget<..>` auf. Die Ausgabe mit der auskommentierten Überladung von `print(T const &)` ist in Listing 184 auf der nächsten Seite abgebildet.
Der Versuch, `print(aliasInt)` aufzurufen, führt dann zu der dargestellten Fehlermeldung, weil die Spezialisierung von `AliasInheritWidget<int>` nicht von `Widget` erbt und deshalb auch nicht implizit konvertiert werden kann.

Listing 182: Anwendung Template Aliasse

```
1  void demoAlias(){
2     cout << "demoAlias" << endl;
3     AliasInheritWidget<> aliasDouble;
4     aliasDouble.print();
5     print(aliasDouble);
6
7     AliasInheritWidget<int> aliasInt;
8     aliasInt.print();
9     //error: no matching function for call to 'print(AliasInheritWidget<int>&)'
10    //wenn die Überladung print(T const & t) nicht existiert
11    print(aliasInt);
12
13    AliasInheritWidget<long> aliasLong;
14    aliasLong.print();
15    print(aliasLong);
16
17    AliasUsingWidget<> aliasChar;
18    aliasChar.print();
19    print(aliasChar);
20 }
```

Die Methode `print()` des Primary Templates `Widget` gibt den Namen des Typs des Parameter `T` aus. Für `long` liefert die Gnu Implementierung den Namen `l`, ein kleines L. Für `long` existiert keine Spezialisierung, deshalb wird das Primary Template von `Widget` vom Compiler verwendet und `Widget<long>` erzeugt. Die Ausgabe von `aliasLong.print()` ist entsprechend.

Listing 183: Ausgabe Template Aliasse

```
demoAlias
Widget<double>
print(T const & t): Widget<double>
AliasInheritWidget<int>
print(T const & t): AliasInheritWidget<int>
Widget<T=int>: 1
print(T const & t): Widget<T=int>: 1
Widget<char>
print(Widget<T> const & t)Widget<char>
```

Spezialisierung von Aliasnamen ist mit Vorsicht zu genießen, da dieses System von Templates, Aliassen und Spezialisierungen nicht offensichtlich und schwer zu durchschauen ist.

Listing 184: Ausgabe Template Aliasse ohne Überladung von print

```
demoAlias
Widget<double>
print(Widget<T> const & t): Widget<double>
AliasInheritWidget<int>
Widget<T=int>: 1
print(Widget<T> const & t): Widget<T=int>: 1
Widget<char>
print(Widget<T> const & t): Widget<char>
```

In Listing 184 ist die Ausgabe ohne die Überladung von `print(T const & t)` und dem auskommentierten Aufruf von `print(aliasInt)` dargestellt.

10 Funktions Templates

Funktions Templates definieren eine Familie von Funktionen. Sie werden mit Typen oder Werten parametriert. Die Template Parameter sind Platzhalter für die Template Argumente und können verwendet werden wie konkrete Typen. Funktions Templates können überladen werden und mit "normalen" Funktionen koexisitieren.

10.1 Deklaration und Definition eines Funktions Templates

Die Deklaration in Listing 185

Listing 185: Deklaration eines Funktions Templates

```
1 template<class T>
2 inline T const& max(T const& a, T const& b);
```

teilt dem Compiler mit, es gibt etwas mit dem Namen `max`, das ein Funktions Template ist, eine `T const&` zurückliefert, zwei `T const&` als Argumente erwartet und wenn möglich inline realisiert werden soll.

Die Definition (ist immer auch eine Deklaration) in Listing 186

Listing 186: Definition und Deklaration eines Funktions Templates

```
1 template<class T>
2 inline const T& max(const T& a, const T& b) {
3    return a < b ? b : a ;
4 }
```

definiert eine Familie von Funktionen, die das Maximum von zwei Werten zurückliefert. Voraussetzungen für T in diesem Fall (implizites Interface): Der relationale Operator **<** (less than) muss für T definiert sein.

Template Parameter (`template<comma-separated-list>`) werden in den spitzen Klammern angegeben.

- Hier : `class T` oder gleichbedeutend: `typename T`
- `T` ist Platzhalter für einen konkreten Typ und kann verwendet werden wie ein konkreter Typ. Die Schlüsselworte `typename` und `class` sind an dieser Stelle (template parameter) gleichwertig.

Call Parameter `function-name(type-list)` werden in runden Klammern angegeben.

```
type-list : type | type, type-list
```

- Hier : `T const& a, T const& b`
- Wie bei “normalen“ Funktionen
- Können default Werte haben

10.2 Funktions Templates und `inline`

Kurze Funktionen sollten `inline` deklariert werden. Der Specifier `inline` teilt dem Compiler mit, dass, wenn möglich, anstelle eines kostspieligen Funktionsaufrufes, der Funktionskörper an der Stelle des Funktionsaufrufes eingefügt werden soll. Der Scope lokaler Variablen bleibt dabei erhalten.

Sowohl `inline` deklarierte Funktionen als auch **Funktions Templates können in mehreren Übersetzungseinheiten definiert werden,** im Gegensatz zur allgemein gültigen *one definition rule*. Dadurch könnte der Eindruck entstehen, dass Funktions Templates implizit `inline` sind, was aber nicht der Fall ist. Sollen Funktions Templates `inline` behandelt werden, muss der Specifier wie in Listing 186 auf der vorherigen Seite verwendet werden.

10.3 Aufruf eines Funktions Templates

Beim Aufruf eines Funktions Templates werden die Template Parameter durch die Template Argumente ersetzt und daraus eine Funktion erzeugt. Dieser Prozess wird *instantiation* genannt.

Template Argumente

- sind die Typen, durch die die Template-Parameter (hier nur T) ersetzt werden.

Call Argumente

- sind die Werte, mit denen die Call-Parameter initialisiert werden.

Template-Argument-Deduction

- ist die automatische Bestimmung der Template-Argumente auf der Basis der Typen der Call-Argumente
- Funktions Templates können daher aufgerufen werden wie „normale“ Funktionen, die Template Argumente werden automatisch erkannt.

Bsp.:

```
int result = max(5,3);
```

Default Typ für ganzzahlige Konstanten ist int. Das ist der Typ der Call-Argumente 5 und 3. Das Template-Argument `T` ist daher **`int`**. Damit wird die Funktion `max<int>(int, int)` aus dem Funktions Template erzeugt und mit den Werten 5 und 3 aufgerufen.

Die Argumente müssen bei der Verwendung von Templates genau übereinstimmen. Es werden (nicht wie bei normalen Funktionen) keine Typkonvertierungen durchgeführt. Der Aufruf

```
double dr = max(3.1, 3);
```

ist daher ein Fehler, weil die Argumente unterschiedliche Typen haben, es werden aber gleiche Typen erwartet. Bei einer Funktion mit der Signatur

```
double max(double a, double b);
```

wäre der Aufruf erfolgreich, weil 3 implizit vom Compiler nach **double** konvertiert werden würde. Diese Funktion könnte als Überladung mit dem Funktions Template koexistieren und würde dann verwendet werden. Gibt es diese Funktion aber nicht und soll das Template angewandt werden, kann das Template Argument explizit angegeben werden:

```
double dr = max<double>(3.1, 3);
```

. . . ok, das Template Argument **double** wurde explizit angegeben, die Funktion `max<double>(double, double)` wird aus dem Funktions Templates erzeugt und jetzt kann eine implizite Typkonvertierung durchgeführt werden.
Alternativ kann das Call-Argument explizit gecastet werden.

```
double dr = max (3.1, static_cast<double>(3));

string hello("hello");
string templates("templates");
string sr = max(hello, templates); //ok, < ist für string definiert
```

Die Anforderung, die an den Typ T gestellt wird (implizites Interface),
ist das Vorhandensein des Operators <. Das Template `max` kann also nur auf Typen angewandt werden, für die dieser Operator definiert ist, ansonsten gibt es einen Syntax Error.

Anmerkung: das Prädikat `less_than<typename T>` wird als default Argument in sortierten Containern verwendet und nutzt ebenfalls (wie hier max) den kleiner Operator „<“ für den Vergleich.

10.4 Überladen von Funktions Templates

Funktionen können überladen werden, das gilt genau so für Funktions Templates. Normale Funktionen und Funktions Templates können koexistieren.

Überladen für einen bestimmten Typ:

```
double max(double a, double b){...};
```

Überladen für Pointer:

```
template<typename T>
T const& max(T* const& a, T* const& b){
   return *a < *b ? b : a;
};
```

Für Pointer ist die allgemeine generische Lösung sinnlos, ob ein Objekt an einem Speicherplatz mit einer höheren Adresse existiert als ein anderes Objekt, wird in den meisten Fällen nicht von Interesse sein. Meistens sollen die Inhalte verglichen werden um dann die Adresse des größeren Objektes als Ergebnis zu erhalten.

Überladen für eine verschiedene Anzahl von Parametern:

```
template<typename T>
T const& max(T* const& a, T* const& b, T* const& c){
   return max(max(a, b), c);
}
```

Soll nur das Funktions Template bei der Auswahl der Kandidaten berücksichtigt werden, kann eine leere Argumentliste angegeben werden: `max<>(3, 5)`. Bei diesem Aufruf werden “normale” Funktionen nicht berücksichtigt.

10.5 Default Template Argumente ab C++11

Die Syntax für Default Argumente von Funktions Templates ist in Listing 187 abgebildet. Es ist dieselbe wie für Klassen Templates. Das Funktions - Template `demoDefaultArguments()` hat sowohl ein default Template Argument `MyClass` als auch ein default Call Argument `T()`.

Listing 187: Default Template Argumente für Funktions Templates

```
1  struct MyClass{
2     void print(){ cout << "MyClass" << endl; }
3  };
4
5  template<typename T = MyClass>
6  void demoDefaultArguments(T  t = T()){
7     t.print();
8  }
9
10 int main(){
11    cout << "TemplateFunktionDefaultArgument" << endl;
12    demoDefaultArguments();
13 }
```

Die Funktion `demoDefaultArguments()` in `main()` wird mit dem Template Argument `MyClass` Instanziiert, wenn das Template Argument weggelassen wird. Das Default Call Argument wird ein Objekt (`t = T()`) des Default Template Arguments `MyClass` sein.

10.6 Convenience Functions (`make_pair`)

Bei Klassen Templates gibt es keine Argument Deduktion. Die gewünschten Typen müssen immer explizit angegeben werden. Deshalb werden häufig sogenannte **convenience functions** wie z.B.: `std::make_pair()` eingeführt.

Listing 188: Convenience Functions (`make_pair`

```
1  template<T1, T2>
2  inline pair<T1, T2> make_pair(const T1& value1, const T2& value2){
3     return pair<T1,T2>(value1, value2);
4  }
```

Beim Aufruf werden T1 und T2 durch die Typen der call Argumente value1 und value2 ersetzt. Argument Deduktion bei Funktionen wird genutzt um eine Klasse zu erzeugen.

Für den Aufruf:

```
make_pair(42, 24.0);
```

erzeugt diese Funktion eine Klasse mit der Template-Id `pair<int, double>`, legt ein Objekt mit den call Argumenten `value1` und `value2` an und liefert dieses zurück.

10.7 Member Function Templates

Methoden in Klassen können als Templates definiert werden. Diese können aber nicht virtual deklariert werden und können keine default Argumente haben. Eine typische Anwendung von Member Templates ist die Unterstützung automatischer Typkonvertierung bei der Zuweisung (assignment operator=) und der Initialisierung (copy constructor).

Beachte: Der Template Kopiekonstruktor oder der Template assignment operator in Listing 189 ersetzen nicht die Typ spezifischen Operationen, diese werden vom Compiler erzeugt, auch wenn ein template dafür vorhanden ist. Die Prüfung auf Selbstzuweisung wird daher in einem Template Assignment Operator nicht benötigt, Exceptionsicher sollte aber auch dieser sein. Soll das Typ spezifische Kopierverhalten definiert werden, müssen die Typ spezifischen Operationen zur Verfügung gestellt werden[253].

Listing 189: Member Function Template

```
1  template<typename ElementType,
2           typename Container = std::deque<ElementType> >
3  class Stack{
4     Container container;
5  public:
6      // Assignment
7      // 1 damit können nur Stack<E, C> zugewiesen werden
8      template<typename ElemType, typename ContType>
9      Stack<ElementType, Container>& operator=(Stack<ElemType, ContType> rhs);
10
11     // 2 damit kann jedes Objekt das die Operationen
12     // top, pop und empty
13     // implementiert zugewiesen werden
14     template<class AnotherType>
15     Stack<ElementType, Container>& operator=(AnotherType rhs);
16     ...
17 };
```

Der Typ des Übergabeparameters bei 1 von
`operator=(Stack<ElemType, ContType> rhs)` weicht vom Typ des Klassen Templates
`Stack<ElememmentType, Container>` ab. Daher kann in der Methode des `operator=` nicht auf **`private`** Member von `rhs` zugegriffen werden, es kann nur die **`public`**

[253] siehe: section 6.5 auf Seite 38 Operationen die der Compiler zur Verfügung stellt.

Schnittstelle verwendet werden!
Mit einer `friend` Deklaration:
`template<class ET, class CT> friend class Stack;` im Klassentemplate Stack kann die Klasse `Stack<ElemType, ContType>` den Zugriff auf ihre privaten Member, gewähren.

Dieser Operator erwartet eine Stack Klasse, die aus demselben Template erzeugt wurde. Stack Objekte der Klasse aus Listing 175 auf Seite 190 können mit diesem Zuweisungsoperator nicht zugewiesen werden.

Der Typ des Übergabeparameters bei 2 von
`operator=(Another rhs)` ist von dem Klassen Template
`Stack<ElememementType, Container>` vollständig unabhängig. Mit diesem Operator können alle Objekte dem Stack zugewiesen werden, wenn sie das implizite Interface `top()`, `pop()` und `empty()` implementieren, das in der Methode des Operators in Listing 190 benutzt wird.
Eine `friend` Deklaration in der Klasse `Stack<ElememementType, Container>` wie oben, hätte keine Auswirkung auf die Zugriffsrechte des übergebenen Objekts.

10.7.1 out of class Member Definition

Member Function Templates können außerhalb der Template Definition wie in Listing 190 definiert werden.

Listing 190: Definition Member Function Template

```
1  // 1
2  template<typename ElementType, class Container>
3  template<typename ElemType, class ContType>
4  Stack<ElementType, Container>&
5  Stack<ElementType, Container>::operator=(Stack<ElemType, ContType> rhs){
6        Container temp;
7        while(!rhs.empty()){
8           temp.push_front(rhs.top());
9           rhs.pop();
10       }
11       container.swap(temp);
12
13       return *this;
14 }
15 // 2
16 template<typename ElementType, class Container>
17 template<class AnotherStack>
18 Stack<ElementType, Container>&
19 Stack<ElementType, Container>::operator=(AnotherStack rhs){
20     //implementierung wie bei 1
21       return *this;
22 }
```

Die Parameterliste des umgebenden Klassen Templates wird zuerst aufgeführt,

dannach folgt die Parameterliste des Member Funktions Templates. Die Template-Id muss auf der Basis der Parameternamen angegeben werden.

Der Methode des `operator=` wird eine Kopie `rhs` (by value) übergeben, da das übergebene Objekt anschließend leer ist `rhs.pop()`. Sie legt einen temporären `Container` an und fügt diesem am Anfang `temp.push_front(rhs.top())` das aktuelle Objekt von `rhs` ein. Dadurch bleibt die Reihenfolge des Stacks erhalten. Sollte während dieser Operationen eine Exception auftreten, ist der ursprüngliche Stack, dem etwas zugewiesen werden sollte, noch unverändert. Zum Schluss wird der Inhalt des temporären Containers mit dem eigenen Container getauscht `container.swap(temp)`. Dabei kann keine Exception auftreten.

Wird eine Stack Klasse mit einem `std::vector` als Container erzeugt (`Stack<int, std::vector<int>>`), kann der Template `operator=` nicht angewandt werden, weil `std::vector` die Operation `push_front` nicht zur Verfügung stellt. Solange keine Zuweisung versucht wird, wird dieser Operator aber nicht erzeugt und der Code erzeugt daher keinen Fehler[254].

[254] [VJ03] 5.3 Member Templates

11 Templates mit variabler Anzahl Argumente, Variadic Templates C++11

Als variadisch werden Argumentlisten mit unbestimmter Arität bezeichnet. Das bedeutet, **die Anzahl der Argumente wird nicht** durch korrespondierende Parameter **festgelegt**, sondern durch die *Ellipse ...* offen gelassen.

In Listing 191 wird das überladene Funktionstemplate `print` definiert.

Der Aufruf in `f()` verwendet `print(First t, Rest...rest)` bei 2. Bei diesem Aufruf werden die Argumente auf die Parameter von `print(First t, Rest...rest)` verteilt. `First` wird zu **`int`** ausgewertet und `t` wird mit `1` initialisiert. Die restlichen Argumente werden in den Parameter Pack[255] `rest` gepackt.
Die Syntax **`class`**`...Rest` definiert keinen *template type parameter* `Rest` sondern einen *template parameter pack*, der zur Deklaration eines *Parameter Pack* `Rest ...rest` verwendet wird[256]. Dieser repräsentiert alle bei der Verwendung übergebenen Typargumente.
Die Ellipse steht jeweils vor dem Parameternamen.

Bei der Verwendung des Parameters in 5 wird der Parameter Pack mit dem Wiederholungsoperator `rest...` entpackt. Die Ellipse steht dabei nach dem Parameternamen.
Innerhalb von `print` wird bei 3 und 4 das überladene `print(T t)` aufgerufen und bei 5 rekursiv das `print` von 2 mit einer verkürzten Liste der Argumente aufgerufen. Der `rest...` wird dabei auf `First t, Rest...rest` verteilt.

Listing 191: Variadische Templates

```
1  template<class T>
2  inline
3  void print(T t){  // 1
4     std::cout << t;
5  }
6  template<class First, class ...Rest> // 2
7  inline
8  void print(First t, Rest ... rest){
9     print(t);   //3
10    print(", "); // 4
11    print(rest...); // 5
12 }
13
14 //Anwendung
15 void f(){
16    print(1, 42.1, 'c', "Hello Variadic Templates");
17 }
18 //Ausgabe
19 1, 42.1, c, Hello Variadic Templates
```

[255] http://en.cppreference.com/w/cpp/language/parameter_pack
[256] https://akrzemi1.wordpress.com/2013/06/05/intuitive-interface-part-i/

Innerhalb eines variadic Templates liefert `sizeof...(rest)` die Anzahl der verbleibenden Argumente. Der Aufruf von `printNumArgs(..)` aus Listing 192 mit denselben Argumenten liefert die Ausgabe `Anzahl Argumente: 4`.

Listing 192: sizeof...()

```
1 template<class ...Types>
2 void printNumArgs(Types ...args){
3    std::cout << "Anzahl Argumente: " << sizeof...(args) << std::endl;
4 }
```

Die STL nutzt diesen Mechanismus für verschiedene Factory Methoden, wie z.B. `make_shared` oder `make_unique` und Verwandte.

12 Nontype Template Parameter

Es ist auch möglich Templates mit Konstanten zu parametrieren. Bsp.:
`std::bitset<32> flags32;`
`std::bitset<50> flags50;`
Die Konstanten sind Bestandteil des Typs der erzeugt wird. Die Objekte `flags32` und `flags50` haben also unterschiedliche Typen! Daher ist es auch nicht möglich, sie miteinander zu vergleichen oder zuzuweisen außer es sind entsprechende Operatoren dafür definiert.

Template Parameter, die keine Typen sind müssen entweder:

- Ganzzahlige Konstanten
- Enumerationen
- Zeiger mit external linkage
- Funktionen (Zeiger/Referenzen) mit internal linkage (ab C++11)

Nicht erlaubt sind Gleitpunktkonstanten oder Objekte von Klassen. Das Listing 193 skizziert die Definition eines Klassen Templates Stack, die eine natives Array zur Verwaltung der Elemente benutzt. Über den Parameter `N` kann der Nutzer die Größe des Stacks festlegen. Das Klassen Template `template<typenmae T, std::size_t N> struct array;`[257] ist auf diese Weise implementiert.

Listing 193: Definition fixed sized Stack

```
1  template<typename E, std::size_t N>
2  class Stack{
3     E container[N];
4     std::size_t numElements;
5  public:
6     Stack();
7     void push(E const& e);
8     void pop();
9     E top() const;
10    bool empty() const { return numElements == 0; }
11    bool full() const { return numElements == N; }
12 };
13
14 template<typename E, std::size_t N>
15 Stack<E, N>::Stack() : numElements(0){/*nothing to do*/}
16
17 template<typename E, std::size_t N>
18 void Stack<E, N>::push(E const& e) {
19    if(full()) throw std::logic_error("Stack<>::push() overflow");
20
21    container[numElements++] = e;
22 }
23
```

[257] seit C++11

```
24 template<typename E, std::size_t N>
25 E Stack<E, N>::top() const {
26    if(empty()) throw std::logic_error("Stack<>::top() underflow");
27
28    return container[numElements-1];
29 }
30
31 template<typename E, std::size_t N>
32 void Stack<E, N>::pop() {
33    if(empty()) throw std::logic_error("Stack<>::pop() underflow");
34
35    --numElements;
36 }
```

Zuweisen, Vergleichen von Stacks aus Listing 193 auf der vorherigen Seite ist nur möglich, wenn sowohl der Typ `E` als auch die Größe `N` der Stack Objekte übereinstimmen. Stacks mit gleichen Elementtypen aber unterschiedlicher Größe sind verschiedene Typen. Die Zuweisung eines Stacks aus Listing 193 auf der vorherigen Seite an einen Stack aus Listing 189 auf Seite 201 ist aber möglich, da dieser einen Template Assignment Operator definiert und der fixed sized Stack die erwartete Schnittstelle implementiert.

Teil IV

Generische Programmierung

13 Generische Programmierung

Dieses Kapitel soll einen ersten Eindruck von generischer Programmierung vermitteln. Es ist ein Auszug aus dem Script “Generische Programmierung mit C++ Templates” Einige Absätze wurden gekürzt oder zur Verständlichkeit ergänzt.

13.1 Grundlagen Traits & Policies

Traits und Policies sind technisch dasselbe, es sind Klassen Templates, sie unterscheiden sich in ihrer Verwendung, in ihrem Zweck. Traits können als *type functions*[258] betrachtet werden, Policies werden häufig als Template Argument übergeben.

Traits sind meist von Template Parametern abhängig, sie definieren spezifische Typen und Konstanten für den Typ des Template Arguments. Durch Spezialisierung der Traits für bestimmte Typen kann die Definition, der durch die Traits definierten Typen und Konstanten an den jeweiligen Typ des Template Arguments angepasst werden. Die STL verwendet diesen Mechanismus der generischen Programmierung für die Spezialisierung der Algorithmen mit Hilfe der `iterator_traits<Iterator>`. Seit C++11 steht eine umfangreiche `type_traits` Bibliothek[259] im gleichnamigen Header zur Verfügung.

Eine Ausnahme von dieser Regel ist `string char_traits<Char_Type>`. Dieses Traits Template definiert zusätzlich Operationen für den Vergleich (`compare`), das Verschieben (`move`) und Auffinden (`find`) von Zeichen (**`char`**/`wchar`)

Policies sind meist von den Typen der Template Argumente unabhängig. Sie definieren Verhalten, also Operationen und sind oft normale Klassen. Der Standard `allocator<T>` ist ein Beispiel aus der STL.

13.2 Policy Based Design

Policy Based Design wurde zum ersten mal von Andrei Alexandrescu in [Ale09] beschrieben. Am Bespiel eines Joysticks der auf einer Platine verbaut ist, soll hier Policy Based Design veranschaulicht werden.

Der Joystick in Listing 196 auf der nächsten Seite löst fünf Ereignise aus. Je eines wenn er nach links, nach rechts, usw. gedrückt wird. Es soll möglich sein, den

[258] siehe section 13.4 auf Seite 217

[259] http://en.cppreference.com/w/cpp/header/type_traits

Joystick so zu konfigurieren, dass zur Laufzeit die Handler, die für diese Ereignisse gerufen werden, verändert werden können. Wenn das nicht benötigt wird, soll es möglich sein, dem Joystick die Handler statisch zur Verfügung zu stellen, so dass sie `inline` implementiert werden können. Die Operationen `simulateLeft()` und `simulateRight()` simulieren die Hardware Ereignisse und zeigen die Verwendung der Policy. Um die fünf Policies zu bündeln, werden sie in einem Repository zusammen gefasst.

13.2.1 Static Binding

Listing 194: Die DefaultHandler Policy

```
1 struct DefaultHandler{
2     static void Handle(){}
3 };
```

Listing 195: Das DefaultJoystickRepository

```
1  template<
2      class LeftHandler = DefaultHandler,
3      class DownHandler = DefaultHandler,
4      class RightHandler = DefaultHandler,
5      class UpHandler = DefaultHandler,
6      class PushHandler = DefaultHandler
7          >
8  struct DefaultJoystickRepository{
9      typedef LeftHandler Left;
10     typedef DownHandler Down;
11     typedef RightHandler Right;
12     typedef UpHandler Up;
13     typedef PushHandler Push;
14 };
```

Das erste Beispiel verwendet statisches Binding. Die Konfiguration des Joysticks erfolgt über das `DefaultJoystickRepository` in Listing 195, das den `DefaultHandler` aus Listing 194 für jedes Ereignis als Defaultargument definiert. Der `DefaultHandler` stellt eine leere Methode `Handle()` zur Verfügung. Diese ist `inline` und wird vom Compiler eliminiert.

Listing 196: Der Joystick

```
1  template<class Repository=DefaultJoystickRepository<> >
2  class Joystick : public Repository {
3  public:
4      typedef Joystick<Repository> this_type;
5
6      static void simulateLeft(){
7          std::cout << "Joystick::simulateLeft()" << std::endl;
8          this_type::Left::Handle();
9      }
10     static void simulateRight(){ /* dto. für alle Ereignisse */ }
```

```
};
```

Auf die Member (z.B. `Left`) der Basisklasse in Listing 196 auf der vorherigen Seite kann nicht direkt zugegriffen werden, weil sie zur Übersetzungszeit des Templates noch nicht sichtbar sind. Erst bei der Instanziierung des Templates ist die Information über die geerbten Member verfügbar. Darum muss der Zugriff entweder durch **`this`**`->` in nicht statischen Methoden oder durch den eigenen Typ (`this_type::`) in statischen Methoden qualifiziert werden, um den Scope, in dem der Name gesucht wird, festzulegen.

Listing 197: Die statischen Handler

```
struct LeftHandler{
    static void Handle(){
        cout << "static LeftHandler::Handle()"  << endl;
    }
};
struct RightHandler{ /* dto.*/ };
```

Listing 198: Die Verwendung des Joysticks StaticBound

```
void demoStaticBound(){
    cout << "StaticBound" << endl;
    typedef DefaultJoystickRepository<
        LeftHandler,
        DefaultHandler,
        RightHandler> JoystickRepository;
    typedef Joystick<JoystickRepository> StaticBound;

    StaticBound::simulateLeft();
    StaticBound::simulateRight();
    StaticBound::simulatePush();
}
```

Die aus der Verwendung des Joysticks resultierende Ausgabe wird in Listing 199 gezeigt.

Listing 199: StaticBound Output

```
StaticBound
Joystick::simulateLeft()
static LeftHandler::Handle()
Joystick::simulateRight()
static RightHandler::Handle()
Joystick::simulatePush()
```

Für das Ereignis Push ist kein Handler definiert, bzw. der `DefaultHandler`. Daher erfolgt auch keine Ausgabe.

13.2.2 Dynamic Binding

Das Beispiel in Listing 203 auf der nächsten Seite zeigt die dynamische Verwendung des Joystick. Die Handler können während der Laufzeit ausgetauscht werden. Zur Vereinfachung werden im Beispiel Zeiger auf Funktionen verwendet. Eine echte Joystick Implementierung würde eher auf eine `Function<...>` Implementierung zurückgreifen, wie sie in [MC++D] beschrieben ist, so dass alle Callable Entities von C++ als Handler verwendet werden könnten. Also globale Funktionen, Memberfunktionen und Funktoren und seit C++11 auch Lambda Ausdrücke.

Listing 200: Die DefaultDynamicHandler Policy

```
1  template<int num>
2  struct DefaultDynamicHandler{
3      typedef void(*PF)();
4      static void setHandler(PF handler){
5              Handler() = handler;
6      }
7      static PF getHandler(){ return Handler();}
8
9      static void Handle(){
10         if(Handler()){
11          Handler()();
12         }
13     }
14 private:
15     static PF& Handler(){
16         static PF pf = nullptr;
17         return pf;
18     }
19 };
```

Der `DefaultDynamicHandler` definiert zusätzlich zu der `Handle()` Methode eine `setHandler(PF handler)` und eine `PF getHandler()` Methode.
Der Zeiger auf die Handler Function wird in einer lokalen **`static`** Variablen in der **`private`** `PF& Handler()` Methode verwaltet, die eine Referenz darauf zurückliefert. Damit wird die Notwendigkeit der Definition eines Klassenattributs in einer eigenen Übersetzungseinheit vermieden.

Die `Handle()` Methode wendet auf den Rückgabewert der `Handler()` Methode den Funktionsaufrufoperator an: `Handler()()` und ruft damit die Funktion auf, auf die `pf` zeigt.

Damit nicht für jedes Ereignis ein dynamischer Handler geschrieben werden muss, ist der `DefaultDynamicHandler` als Template ausgestaltet, das einen **`int`** als Parameter erwartet, der nicht benutzt wird. Der Parameter dient lediglich dazu jeweils einen eigenen Typ für die verschiedenen Ereignisse zu erzeugen, der seine eigenen statischen Member hat.

Der Joystick erbt von seinem Repository, dadurch erbt er die notwendigen Datenstrukturen zur Verwaltung der Handler und die Schnittstelle `setHandler(...)` von

den DefaultDynamicHandlern. Durch die Policies wird die Schnittstelle des Joysticks erweitert[260].

Mit dieser Implementierungstechnik würden sich auch gemischte Handler, also Handler die zur Laufzeit verändert werden können und Handler die statisch gebunden sind, realisieren lassen.

Listing 201: Das DynamicJoystickRepository

```
template<
    class LeftHandler = DefaultDynamicHandler<0>,
    class DownHandler = DefaultDynamicHandler<1>,
    class RightHandler = DefaultDynamicHandler<2>,
    class UpHandler = DefaultDynamicHandler<3>,
    class PushHandler = DefaultDynamicHandler<4>
        >
struct DynamicJoystickRepository
      : DefaultJoystickRepository<
        LeftHandler,
        DownHandler,
        RightHandler,
        UpHandler,
        PushHandler
        >
      {};
```

Das DynamicJoystickRepository in Listing 201 erbt vom DefaultJoystickRepository aus Listing 195 auf Seite 209 und delegiert die Typen lediglich an dieses weiter. Das gewährleistet, dass für die Handler die gleichen Namen verwendet werden. Lediglich die default Typen sind andere.

Listing 202: Die dynamischen Handler

```
void leftHandler(){
    cout << "dynamic leftHandler():" << endl;
}
void rightHandler(){ dto. }
```

Die dynamischen Handler aus Listing 202 sind einfache C-Funktionen.

Listing 203: Die Verwendung des Joysticks DynamicBound

```
void demoDynamicBound(){
    cout << "DynamicBound" << endl;
    typedef Joystick<DynamicJoystickRepository<> > DynamicBound;
    DynamicBound::simulateLeft();
    DynamicBound::Left::setHandler(leftHandler);
    DynamicBound::simulateLeft();
    DynamicBound::Right::setHandler(rightHandler);
    DynamicBound::simulateRight();
}
```

[260][Ale09]

Beim ersten Ereignis Left ist noch kein Handler angemeldet, entsprechend ist die Ausgabe in Listing 204.

Listing 204: DynamicBound Output

```
DynamicBound
Joystick::simulateLeft()
Joystick::simulateLeft()
dynamic leftHandler()
Joystick::simulateRight()
dynamic rightHandler()
```

13.2.3 Kompatibilität

Listing 205: Die Host Klasse

```
template<class T, class CheckPolicy>
class Host : public CheckPolicy{
public:
   Host(T* pointer):pointer(pointer){
      CheckPolicy::check(pointer);
   }

   template<class T2, class CP>
   Host(Host<T2, CP> const& rhs) :
      CheckPolicy(rhs),
      pointer(rhs.getPointer())
   {
      CheckPolicy::check(pointer);
   }

   T* getPointer()const{ return pointer; }
private:
   T* pointer;
};
```

Die Host Klasse erbt von der Policy. Für die Konvertierung aus abweichenden Policies und Ts wird ein templatized Konvertierungskonstruktor definiert:
`Host(Host<T2, CP> const&)` der ein anderes Host Objekt entgegen nimmt. Die Konvertierung ist nur erfolgreich, wenn die `pointer` und die Policies kompatibel sind[261].

Listing 206: Die Policies

```
template<class T>
struct NoCheck{

   static void check(T const* p){}
};

```

[261] [Ale09] 1.11 Compatible and Incompatible Policies

```
7  template<class T>
8  struct Check{
9     Check(){}
10
11    template<class T1>
12    explicit Check(NoCheck<T1> const&){
13       std::cout << "Check(NoCheck<T> const&)" << std::endl;
14    }
15    static inline void check(T const* p)
16    {
17       std::cout << "Check::check(T const* p)" << std::endl;
18       if(!p)
19       {
20          throw std::runtime_error("NullPointer Exception: Pointer must not be null!
                ");
21       }
22    }
23 };
```

Die Policy, die mit einer anderen kompatibel sein soll, definiert einen Konvertierungskonstrutor
`explicit Check(NoCheck<T> const&)` für die Policy, mit der sie kompatibel ist. Dieser wird im Konstruktor von Host verwendet.

Listing 207: Anwendung der kompatiblen Host und Policy Klassen

```
1  int main(){
2     Base b;
3     Derived d;
4
5     typedef Host<Derived, NoCheck<Base> > DerivedNoCheck;
6     DerivedNoCheck dnc1(&d); // init
7     DerivedNoCheck dnc2(dnc1); // copy ctor
8     DerivedNoCheck dnc3(0); // noCheck
9     dnc3 = dnc1;
10
11    typedef Host<Base, Check<Base> > BaseCheck;
12    BaseCheck bc1(&d); // init
13    BaseCheck bc2(dnc1); // convert ctor Derived->Base NoCheck->Check
14
15    // NullPointer Exception: Pointer must not be null!
16    //BaseCheck bc3(0); // Check
17
18    // darf nicht möglich sein
19    //typedef Host<Base, NoCheck<Base> > BaseNoCheck;
20    //error: no matching function for call to 'NoCheck<Base>::NoCheck(const Host<
          Base, Check<Base> >&)
21    //no known conversion for argument 1 from 'const Host<Base, Check<Base> >' to '
          const NoCheck<Base>&'
22    //BaseNoCheck bnc4(bc1);
23 }
```

13.3 Templates sind Klassen, Klassen sind Objekte

Klassen, die aus Templates mit konstanten Adressen wie in Listing 208 skizziert, oder aus Typen, die die Register zusammenfassen, generiert werden, können **als Objekte** betrachtet werden, weil aus jedem Template, das mit einem unterschiedlichen Satz von Argumenten verwendet wird, jeweils ein eigener Typ entsteht, mit einem eigenen Satz an Konstanten und **static** Membern. Ein praktisches Beispiel dafür sind sogenannte Memory Mapped Register von MicroControllern.

Von diesen Klassen werden keine Objekte benötigt, weil die notwendigen Informationen bereits durch den Typ festgelegt sind. Als Analogie zu den *type functions* aus section 13.4 auf Seite 217 könnte von diesen Instanzen der Templates von *type objects* gesprochen werden.

Listing 208: Ein Template als Klasse

```
// ======================================================
// MemoryMappedRegister
// Abstrahiert ein im Memory gemapptes Prozessorregister
// ======================================================
template<   class HAL_Traits,
          class Address, // RegisterAddress
          typename RegisterType_ = typename HAL_Traits::RegisterType>
struct MemoryMappedRegisterImpl
{
   typedef RegisterType_ RegisterType;
   typedef RegisterType* RegisterPointer;
   typedef typename HAL_Traits::ValueType ValueType;

   // ab C++11: address kann hier initialisiert werden
   constexpr static RegisterPointer address =
         reinterpret_cast<RegisterPointer>(Address::value);

   // ab C++11: konstante kann verwendet werden
   static void setBit(ValueType b){ *address |= b; }
   static void clearBit(ValueType b){ *address &= ~b; }
   static void clear(){ ...}
   static ValueType readBit(ValueType b){...}
   static ValueType read(){ ... }

   // bis C++11: der Integerwert muss gecastet werden
   static void set(ValueType value){
      *reinterpret_cast<RegisterPointer>(Address::value) = value;
   }
};
```

Die Member können alle **static** als *class member* und meistens `constexpr` deklariert werden oder es sind selbst nur Typen die als Objekte verwendet werden können. Der **this** Pointer ist implizit durch die Konstanten geben.

Aus dieser Perspektive können diese **Templates als Klassen** betrachtet werden, die Instanziierung der Templates mit den Argumenten, als der Aufruf des Kon-

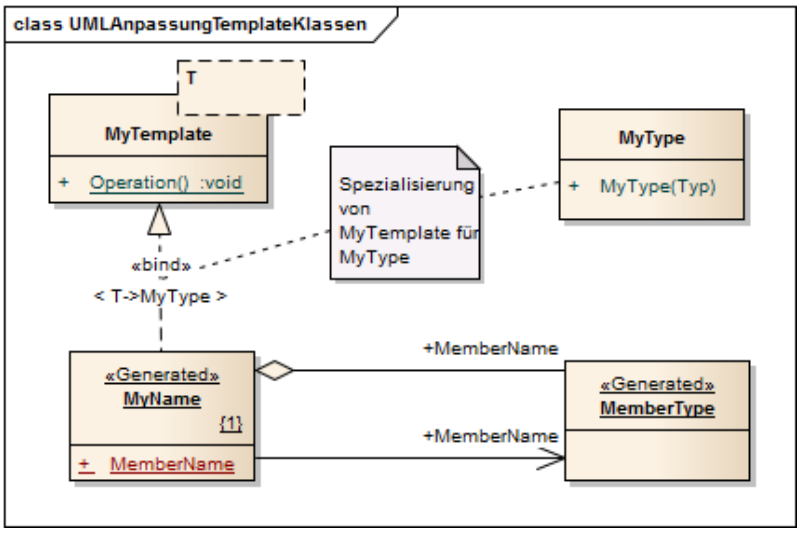

Diagramm 8: Anpassung UML: Templates als Klassen, Klassen als Objekte

struktors. In Diagramm 7 auf Seite 175 sind zur Darstellung dieser Analogie, auf der rechten Seite die Beziehung des Objekts `myType` zu seiner Klasse und auf der linken Seite die Beziehung der Template Instanzen zu ihrem Template, aus dem sie erzeugt werden, nebeneinander gestellt.

Um diesen Sachverhalt darzustellen wird in diesem Werk die Notation in Diagramm 8 verwendet. Die Templates werden mit den *class member* Operationen[262] dargestellt. Die Bezeichner der daraus generierten Klassen werden ebenfalls unterstrichen dargestellt, wie Objekte im Objektdiagramm und die Klasse wird mit dem Stereotyp `<<Generated>>` und der Multiplizität `1` ausgezeichnet.

Die Template Binding Notation `<<bind>> <T->MyType>` tritt an die Stelle der `<<InstanceOf>>` Dependency.

Die Multiplizität ist redundant, sie wird weggelassen und dient hier nur zur Darstellung des Sachverhaltes, dass es dieses “Objekt”, repräsentiert durch den Typ, nur einmal gibt.

*member type definition*s werden als Aggregate bzw. als navigable Associations dargestellt, wenn diese Typen ebenfalls als Objekt betrachtet werden können. Innere Typen werden als **`public static`** Attribute dargestellt, zur Unterscheidung von Aliasnamen von *member type definition*s via `typedef` oder `using`.

Die Semantik der Beziehungen ist dieselbe wie im Objektdiagramm. Die Kardinalität wird durch die Anzahl der gezeichneten “Objekte” repräsentiert.

Die Anpassung der UML ist notwendig, weil ein und dasselbe Konstrukt: C++ Templates, zu vollständig anderen Zwecken verwendet wird und darum auch anders gedacht werden muss, was durch eine andere Darstellung erleichtert wird.

[262] UML: unterstrichen, C++ **`static`**

13.4 *type function*s, Klassen Templates als Funktionen

*type function*s sind Klassen Templates die einen Typ erzeugen. Bei der Verwendung eines Klassen Templates als *type function* wird nicht die aus der Template Instanziierung resultierende Klasse verwendet, sondern ein *member type* der innerhalb dieser Klasse definiert ist und der per **Konvention** mit dem Namen `type` benannt wird.

`Component<..>` in Listing 209 auf der nächsten Seite ist eine solche *type function*, sie erzeugt in diesem Fall einen Typ, der dem Namen `Type67` zugewiesen wird.

Templates mit dieser Verwendung werden immer als *type function* bezeichnet und folgen der Konvention
`template<parameter-list> struct typeFunctionName{type}` und werden mit
`typedef typeFunctionName<argument-list>::type TypeName;` aufgerufen.

Ein Aliasname der mit `typedef` deklariert wird, kann als die Initialisierung einer Referenz `TypeName` auf den resultierenden Typ angesehen werden.
Seit C++11 existiert eine alternative Syntax für diese “Initialisierung”, des Aliasnamens, die die Verwandschaft noch deutlicher macht:
`using TypeName = typeFunctionName<argument-list>::type;`.
Mit der neuen Deklarations Syntax
`auto& varName = objectFunction();` kann die Analogie am besten verdeutlicht werden.
`using` tritt an die Stelle von `auto&`,
der Name der Referenz entspricht dem Aliasnamen des Typs und der *result type* der *type function* dem Objekt das `objectFunction()` zurückliefert.

Die Auswertung des “Ausdrucks”
`typeFunctionName<argument-list>::type` erfolgt zur compile time, der Ausdruck ist eine *compile time expression*, das Ergebnis ist ein Typ.

Wird ein Alias für ein Template via `using` definiert, kann die Reference auf den nestet `::type` vermieden werden und in Templates wird das Schlüsselwort `typename` nicht benötigt[263].

13.4.1 *type function*s und Konstanten

Das Listing 209 auf der nächsten Seite zeigt eine einfache *type function*, die für eine bestimmte Konstante einen Typ erzeugt. Das Template `TypeFunction<int selector>` und alle Spezialisierungen zusammen sind eine Funktion.

Der Compiler wählt auf Grund der Argumente die entsprechende Spezialisierung des Templates `TypeFunction` aus. Die Anwendung des Templates (`TypeFunction<'C'>::type`) erzeugt einen Typ der bei einer Variablendeklaration verwendet werden kann wie jeder andere Typ auch.

Das Beispiel ergibt so noch keinen Sinn, skizziert aber die Grundlage für einen erweiterbaren Generator.

[263] [Mey15] Item 9 Prefer alias declarations ...

Wird keine passende Spezialisierung gefunden, wird das Primary Template im folgenden, die *primary type function* genannt, verwendet. Diese ist diesem Fall nicht definiert. Die Anwendung der *type function* mit einem **int** z.B. mit 68, dem Wert für 'D', würde zu einer Fehlermeldung führen, weil die *primary type function* nicht definiert ist.

Listing 209: type function mit Konstanten

```
1  //Types
2  class A{};
3  class B{};
4  class C{};
5  //============================
6  // Eine Funktion (TypeFunction) die einen Typ als Ergebnis liefert
7  // Primary template ohne Definition
8  template <int selector> struct TypeFunction;
9
10 // Spezialisierungen
11 template <>
12 struct TypeFunction<'A'>{
13     typedef A type;
14 };
15 template <>
16 struct TypeFunction<'B'>{
17     typedef B type;
18 };
19 template <>
20 struct TypeFunction<'C'>{
21     typedef C type;
22 };
23
24 void f(A const &){ cout << "f(A)" << endl; }
25 void f(B const &){ cout << "f(B)" << endl; }
26 void f(C const &){ cout << "f(C)" << endl; }
27
28 int main(){
29    typedef TypeFunction<67>::type Type67;
30    // alternativ seit C++11
31    using Type67 = TypeFunction<67>::type;
32
33    f(TypeFunction<'A'>::type());
34    f(Type67());
35
36    //error: incomplete type 'TypeFunction<68>' used in nested name specifier
37    //f(TypeFunction<68>::type());
38 }
39 \\Ausgabe
40 f(A)
41 f(C)
```

13.4.2 *type-function*s und Kontrollstrukturen

In Listing 210 ist die *type-function* IF abgebildet. Wie der Name vermuten lässt, wird in Abhängigkeit einer Bedingung, der eine (`THEN` bei true) oder der andere Typ (`ELSE` bei false) zurückgeliefert. Dazu wird das template `IF_Function` partiell für den Wert **`false`** spezialisiert, die anderen Parameter werden offen gelassen. Das primary Template wird angewendet wenn die `Condition` true ist. Seit C++11 gibt es eine solche *type-function* unter dem Namen `conditional`[264]

Listing 210: Die *type-function* IF

```
1  //primary template für true
2  template<bool condition, class THEN, class ELSE>
3  struct IF_Function{
4     using type = THEN;
5  };
6  template<class THEN, class ELSE>
7  struct IF_Function<false, THEN, ELSE>{
8     using type = ELSE;
9  };
10
11 template<bool condition, class THEN, class ELSE>
12 using IF = typename IF_Function<condition, THEN, ELSE>::type;
```

Ein Beispiel für die Anfwendung ist in Listing 211 abgebildet. Die Funktion `f(..)` und die Typen `A, B, C` sind dieselben aus Listing 209 auf der vorherigen Seite.

Listing 211: Die Anwendung von IF

```
1  void demoIFTypeFunction(){
2     cout << endl << "demoIFTypeFunction()" << endl;
3
4     f(IF<true, A, B>());
5     f(IF<false, A, B>());
6  }
7  //Ausgabe:
8  demoIFTypeFunction()
9  f(A)
10 f(B)
```

13.4.3 *type-function*s und Vererbung, Überladen und Spezialisieren

Die in Listing 212 auf der nächsten Seite abgebildete *type function* `Component` ist für die Typen `S1` und `S3` spezialisiert. Sie erzeugen jeweils einen Implementation Typ A oder C. Die Typen `S1` bis `S3` stehen in einer einfachen Vererbungsbeziehung zu einander in Beziehung und werden im weiteren als Selectoren bezeichnet, weil sie als Argument verwendet, die Spezialisierung der *type function* auswählen.

Die *primary type function* erzeugt den Typ `ImplementationError`.

[264] http://en.cppreference.com/w/cpp/types/conditional

Listing 212: type functions und Vererbung

```
// Selectoren S1 bis S3
struct S1{};
struct S2 : S1{};
struct S3 : S2{};

// Implementierungen die erzeugt werden sollen
class ImplementationA{};
class ImplementationC{};
class ImplementationError{};

//Primary type function
template<typename Selector>
struct Component{
   typedef ImplementationError type;
};

template<>
struct Component<S1>{
   typedef ImplementationA type;
};
template<>
struct Component<S3>{
   typedef ImplementationC type;
};
```

In Listing 213 werden mit `Component` die Typen `TypeS1` bis `TypeS3` erzeugt und die überladene Funktion `f(..)` mit jeweils einem Objekt dieser Typen aufgerufen.

Zur Gegenüberstellung des Verhaltens von überladenen Funktionen und spezialisierten Templates ist die Funktion `f(..)` in Analogie zur Spezialisierung von `Component` ebenfalls für die Typen `S1` und `S3` überladen und wird mit einem Objekt von `S2` und `S3` aufgerufen.

Die Ausgabe zeigt den Unterschied. Die Spezialisierung für `S1` wird für das Template Argument `S2` von `Component` nicht verwendet,
der *result type* von `Component<S2>` ist `ImplementationError` aber die Überladung für `S1` von `f()` wird für das Call Argument `S2` ausgewählt. Der Compiler führt einen impliziten *upcast* von `S2` auf eine **`const`** Referenz auf `S1` für das temporäre Objekt beim Aufruf der Funktion durch.

Listing 213: Anwendung type functions und Vererbung

```
void f(ImplementationA const &){ cout << "f(ImplementationA)" << endl; }
void f(ImplementationC const &){ cout << "f(ImplementationC)" << endl; }
void f(ImplementationError const &){ cout << "f(ImplementationError)" << endl; }

void f(S1 const&){ cout << "f(S1)" << endl; }
void f(S3 const&){ cout << "f(S3)" << endl; }

void demoComponent(){
   typedef Component<S1>::type TypeS1;
```

```
10    typedef Component<S2>::type TypeS2;
11    typedef Component<S3>::type TypeS3;
12
13    f(TypeS1{});
14    f(TypeS2{});
15    f(TypeS3{});
16
17    f(S2{});
18    f(S3{});
19 }
20 // Ausgabe
21 f(ImplementationA)
22 f(ImplementationError)
23 f(ImplementationC)
24 f(S1)
25 f(S3)
```

Das Diagramm 9 zeigt die Selectoren und die *type function* `Component` aus Listing 212 auf der vorherigen Seite.

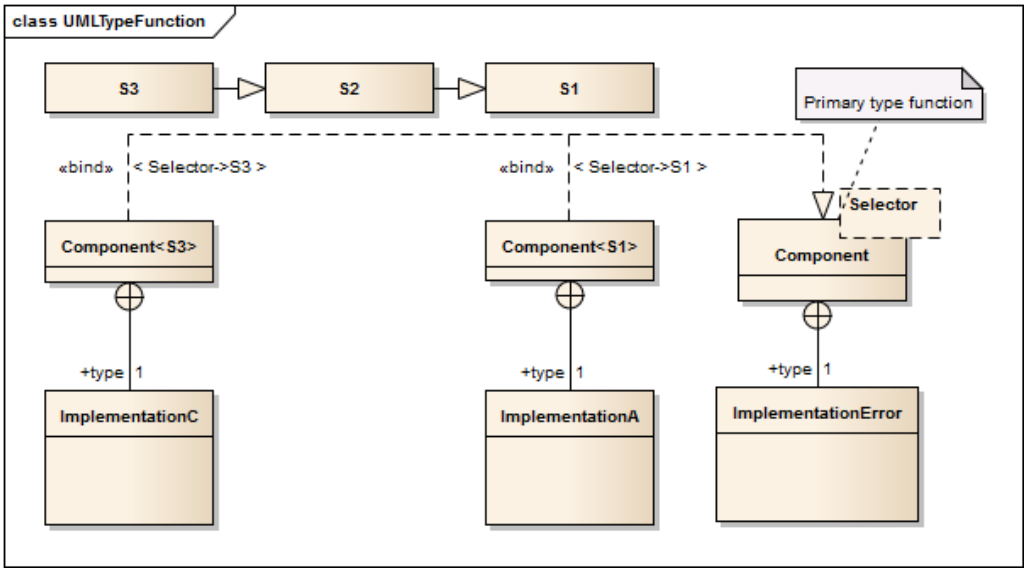

Diagramm 9: UML vollständige Darstellung type function

Eine Komponente wird durch die drei Beteiligten `Selector`, explizite Spezialisierung der *type function* `Component` und des Typs für die Implementation definiert: `using Implementation = Component<Selector>::type;` oder mit `typedef Component<Selector>::type Implementation;`.
Für die *type function* `Component` wird dazu eine Spezialisierung wie in Listing 212 auf der vorherigen Seite benötigt. Existiert keine Spezialisierung, wird keine Implementierung für die Komponente adressiert, bzw. ist das Ergebnis der Typ der durch die *primary type function* definiert wird.

Durch geeignete Template Programmierung kann eine *type function* `Generator ::Create<Selector>::type` erstellt werden, mit der das Verhalten der *type function* `Generator::Component` ähnlich ist wie es von überladenen Funktionen bekannt ist. Die Selectoren müssen dafür ihren Basistyp definieren, die *type functi-*

on `Generator::Create<Selector>::type` sucht nach einer passenden `Component` für den aktuellen Selector oder einen Basistyp des Selectors.

Da im Zusammenhang mit solchen *type function*s die Instanz der Spezialisierung der *type function* `Component` und die Selectoren von untergeordneter Bedeutung sind, werden die Selectoren und die Spezialisierungen zusammengefasst und mit einem Stereotype `<<Selector>>` gekennzeichnet wie in Diagramm 10 dargestellt. Die Vererbungsbeziehung der Selectoren ist eine Implementierung des `Generator::SelectorBase<..>` und keine echte Vererbungsbeziehung. Sie wird wie in Listing 214 dargestellt, etabliert und mit dem stereotype `Basistyp` gekennzeichnet. Diese Information ist redundant, und kann bei der Modellierung weggelassen werden weil Selectoren immer über diese spezielle Vererbungsbeziehung in Beziehung zu einander stehen.

Listing 214: Die Vererbungsbeziehung mit SelectorBase<..>

```
1 struct S1 : Generator::SelectorBase<>{};
2 struct S2 : Generator::SelectorBase<>{S1};
3 struct S3 : Generator::SelectorBase<>{S2};
```

Der *result name* der *type function* ist als Rollenname an die *nested* Association modelliert, die Kardinalität ist 1. Diese Information ist redundant und kann bei der Modellierung weggelassen werden, weil die *type function* `Component` mit einem Selector als Argument immer genau einen Ergebnistyp unter dem Namen `type` veröffentlicht.

Die *nested* Association wird dafür verwendet, um zu zeigen, dass es sich um einen Typ handelt, der nicht in dem Selector aggregiert ist, sondern das Ergebnis der Anwendung der *type function* auf den Selector ist. Im Gegensatz zu Diagramm 8 auf Seite 216, das Typen und Beziehungen dieser Typen zeigt, die aus Templates erzeugt wurden um sie als *type object*s zu verwenden.

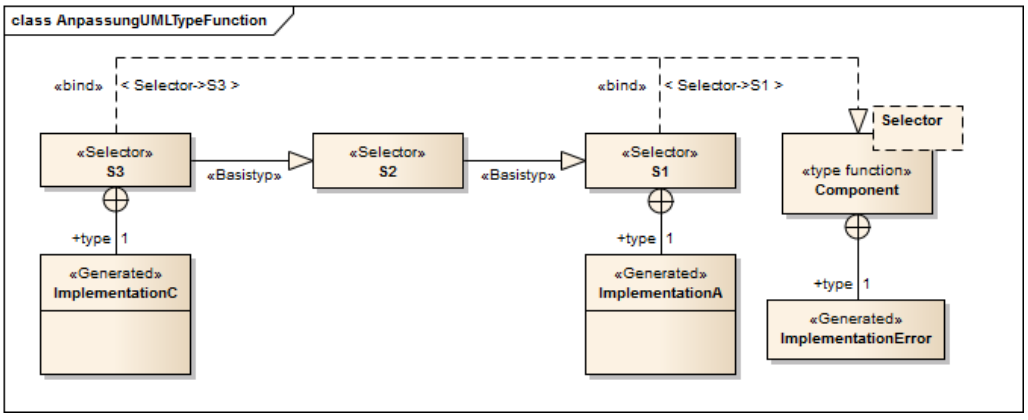

Diagramm 10: UML angepasste Darstellung type function

13.4.4 *member type function*s

Werden *type function*s innerhalb eines Templates als Member Klassen Templates definiert, kann von *member type function*s oder *member type method*s gesprochen werden.

Innere Klassen und Templates haben Zugriff auf die Parameter, die Typdefinitionen und `enum`s des umgebenden Templates. Sie können als Methoden der generischen Programmierung betrachtet werden.

Das umgebende Template kann als Repository[265] betrachtet werden, das verschiedene Typen für die inneren Templates zur Verfügung stellt.

Eine Klasse oder ein Template mit *member type function*s kann auch als *Functor* der generischen Programmierung betrachtet werden, wenn es als Argument an ein Template übergeben wird und die Membertemplates innerhalb verwendet werden.

Template Template Parameter werden in diesem Zusammenhang zu "Funktionszeigern" auf *type function*s.

Die Spezialisierung von Member Templates ist auf partielle Spezialisierung beschränkt. Full Class Template Specialization ist nur im `namespace` Scope zulässig. Templates mit **gleichen Namen** in **verschiedenen `namespace`s** sind verschiedene Templates. Wird ein Template mit dem qualifizierten Namen z.B. `Mock::BaseType <..>` verwendet, werden nur die Spezialisierungen in dem `namespace Mock` berücksichtigt. Dasselbe gilt wenn das Template `Mock::BaseType` als Argument einem anderen Template übergeben wird und innerhalb aufgerufen wird. Der Template Template Paramter kann als *type function pointer* betrachtet werden. Der `namespace` kann in diesem Zusammenhang wie eine Klasse und die Templates wie Methoden betrachtet werden. Die Typen und `enum`s des `namespace` sind innerhalb der Templates im Zugriff, wenn ihre Definition vor dem Template erfolgt.

13.4.5 *value function*s

*value function*s sind das Gegenstück zu *type function*s. Sie berechnen einen Wert und liefern diesen mit dem *return value name* `value` zurück. Alle *value function*s folgen der Konvention
`template<param-list> struct valueFunctionName{value}` und werden mit
`valueFunctionName<argument-list>::value` aufgerufen. Eine sinnvolle Anwendung ist z.B. die Berechnung von Bitpositionen oder Nummern von Devices in der Hardware nahen Programmierung.
Ihre Bedeutung hat mit der Einführung des Schlüsselworts `constexpr` stark abgenommen.

13.5 Einen Typ erzeugen

Das folgende Beispiel basiert auf einigen Ideen und Implementierungen aus der Bibliothek Loki von A. Alexandrescu[266]. Die Techniken, die in Loki vorgestellt sind, helfen Variadic Parameterlisten und deren Verarbeitung leichter zu verstehen und können angewendet werden wenn Variadic Templates nicht zur Verfügung stehen.

[265] Beispiel: https://github.com/GerdHirsch/Cpp-VisitorFrameworkCyclicAcyclic
[266] [Ale09]

13.5.1 Generische Functors

Klassen oder Klassentemplates die eine innere *type-function* definieren können als Functoren der generischen Programmierung betrachtet werden. Das Listing 215 zeigt einen möglichen Functor. Die *type-function* `apply` entspricht dem Function-call `operator()(...)` der konventionellen Programmierung.

Listing 215: Ein Functor der generischen Programmierung

```
1  //forward declaration of used Template in apply
2  template<class ToVisit, class Base>
3  struct InheritFrom;
4
5  struct MyFunctor{
6     template<class Type, class Base>
7     struct apply{
8        typedef InheritFrom<Type, Base> type;
9     };
10 };
```

Als erstes Argument erwartet `apply` einen Typ und als zweites das Ergebnis der vorherigen Anwendung von `apply`. Sie liefert einen Typ `type` als Ergebnis zurück: `apply<A, apply<B, apply<c, EmptyType>::type >::type >::type`

13.5.2 Generische Bausteine zur Erzeugung von Typen

Das Template `InheritFrom`, das `apply` anwendet, ist in Listing 216 abgebildet. Es ist für `Loki::EmptyType` spezialisiert. Das Primary Template erbt von seinem zweiten Parameter. Die Spezialisierung wird zur Basisklasse des erzeugten Typs. Sie hat selbst keine Basisklasse, der Typ `EmptyType` repräsentiert das Ende der Liste. Beide definieren die Operation `visit(ToVisit&)= 0;`.
Ohne die Spezialisierung würde `EmptyType` zur Basisklasse des erzeugten Typs werden.

Listing 216: Der Baustein aus dem der Visitor gebaut wird

```
1  template<class ToVisit, class Base>
2  struct InheritFrom : public Base {
3  public:
4     virtual void visit(ToVisit& visitable) = 0;
5
6     using Base::visit;
7  };
8
9  template<class ToVisit>
10 struct InheritFrom<ToVisit, Loki::EmptyType> {
11 public:
12    virtual void visit(ToVisit& visitable) = 0;
13    virtual std::string toString() const = 0;
14 };
```

13.5.3 Den Typ erzeugen

Genutzt werden kann ein solcher Functor, wenn ein Algoritmus `ForEachClassIn<TypeList<..>, Functor>` zur Verfügung steht, der über seinen ersten Paramter, eine Liste von Typen, iteriert und für jeden Typ in der Liste die *type-function* des Functors aufruft. Als erstes Argument erwartet `apply` den aktuellen Typ in der Liste und als zweites das Ergebnis der vorherigen Anwendung von `apply`.

Die Anwendung auf eine `Typelist` ist in Listing 217 abgebildet.

Listing 217: Die Anwendung des Functors mit ForEachClassIn

```
1  //forward declaration of types
2  class A; class B; class C;
3
4  // Create Typelist with A, B, C
5  typedef
6     MakeTypelist<A, B, C>::type
7  typelist;
8
9  // Iterate over typelist and call apply
10 typedef
11    ForEachClassIn
12    <
13       typelist,
14       MyFunctor
15    >::type
16 VisitorBase;
```

Der Algoritmus `ForEachClassIn` ist ebenfalls eine *type-function*, die das Ergebnis des Functors zurückliefert.

13.5.4 Der erzeugte Typ und seine Verwendung

Der Typ `VisitorBase` ist eine lineare Vererbungsstruktur wie sie in Listing 218 skizziert ist.

Listing 218: Die Vererbungsstruktur der Klasse VisitorBase

```
1  class InheritFrom<C, EmptyType> {
2  public:
3     virtual void visit(C&) = 0;
4     virtual std::string toString() = 0;
5  };
6  class InheritFrom<B, InheritFrom<C, EmptyType>>
7  :public InheritFrom<C, EmptyType>
8  {
9  public:
10    using InheritFrom<C, EmptyType>::visit;
11    virtual void visit(B&) = 0;
12 };
```

```
13
14 //entspricht VisitorBase
15 class InheritFrom<A, InheritFrom<B, InheritFrom<C, EmptyType>>>
16 :public InheritFrom<B, InheritFrom<C, EmptyType>>
17 {
18 public:
19    using InheritFrom<B, InheritFrom<C, EmptyType>>::visit;
20    virtual void visit(A&) = 0;
21 };
22
23 // ein spezieller Visitor
24 class MyVisitor : public VisitorBase{
25 public:
26    void visit(A& a){...}
27    void visit(B& a){...}
28    void visit(C& a){...}
29 };
```

Mit der `using Base::visit` deklaration in `InheritFrom`, wird die `visit` Operation der Basisklasse in den abgleiteten Klassen sichtbar.

13.5.5 Eine Liste von Typen

Das Listing 219 zeigt die `Loki::Typelist`. Ein Template, das als erstes Argument einen Typ und als zweites eine `Loki::Typelist` erwartet. Der Typ, den `MakeTypelist<A, B, C>::type` erzeugt, ist in der letzten Zeile skizziert.

Listing 219: Die Loki::Typelist

```
1 template <class T, class U>
2 struct Typelist
3 {
4    typedef T Head;
5    typedef U Tail;
6 };
7
8 typedef Typelist<A,
9       Typelist<B,             //    C     B     A
10      Typelist<C, Loki::NullType>::type>::type>::type
11 typelist;
```

13.5.6 Die *type-function* ForEachClassIn

Listing 220: Die *type-function* ForEachClassIn

```
1  //Primary Template ohne Definition mit Default Argument Loki::EmptyType
2  template
3      <
4          class TList,
5          class Functor,
6          class Root = Loki::EmptyType
7      >
8      struct ForEachClassIn;
9
10     //============================================================
11     template
12     <
13         class T1,
14         class T2,
15         class Functor,
16         class Root
17     >
18     struct ForEachClassIn<Loki::Typelist<T1, T2>, Functor, Root>
19     {
20             typedef typename ForEachClassIn<T2, Functor, Root>::type BaseType;
21             typedef typename Functor::template apply<T1, BaseType>::type type;
22     };
23
24     //============================================================
25     template
26     <
27         class T,
28         class Functor,
29         class Root
30     >
31     struct ForEachClassIn< Loki::Typelist<T, Loki::NullType>, Functor, Root>
32     {
33         typedef Root BaseType;
34         typedef typename Functor::template apply<T, BaseType>::type type;
35     };
```

Die *type-function* `ForEachClassIn` besteht aus

- dem Primary Template, das nicht definiert ist und das das default Argument `Loki::EmptyType` für den Parameter `Root` festlegt
- der Spezialisierung für `Loki::Typelist<T1, T2>`[267]
- und der Spezialisierung für `Loki::Typelist<T, Loki::NullType>`

Wird `ForEachClassIn` mit einer `Typelist` wie in Listing 219 auf der vorherigen Seite skizziert aufgerufen, ist
`T1=A, T2=Typelist<B, Typelist<C, NullType>>`. Deshalb ruft sich der Algoritmus als erstes selbst mit
`T1=B, T2=Typelist<C, NullType>` und dann mit

[267] section 9.6 auf Seite 187

`T1=C, T2=NullType` auf. Wenn `T2=Loki::NullType` ist, wird die entsprechende Spezialisierung verwendet. Diese wendet `apply<T, BaseType>::type` an und definiert den Membertype `type` damit. Das entspricht in diesem Fall `InheritFrom<C, EmptyType>{...}`. Dieser wird zu `BaseType` in der Spezialsierung `Typelist<T1, T2>` die damit den Functor `apply<T1=B, BaseType>::type` aufruft. Die Rekursion wird wieder abgewickelt und der erste Aufruf von `ForEachClassIn` definiert das Ergebnis der *type-function* `ForEachClassIn<..>::type` mit `apply<A, BaseType>::type`.

Die *type-function* `ForEachClassIn` ist inspiriert von der *type-function* `GenLinearHierarchy`[268]. Diese wird aber Teil des erzeugten Typs und legt fest, wie mit den Typen in der `Typelist` umgegangen wird. Die *type-function* `ForEachClassIn` iteriert lediglich über die Liste der Typen, sie wird nicht Teil des erzeugten Typs und wie mit den Typen in der Liste umgegangen wird, wird durch den Functor festgelegt.

13.5.7 Das Schlüsselwort template und typename

Das Schlüsselwort `template` bei der Anwendung[269] `typename Functor::template apply<T, BaseType>::type` ist notwendig, um dem Compiler mitzuteilen, dass es sich bei dem Namen `apply` um den Namen eines Templates handelt. Das Schlüsselwort `typename` bezieht sich auf den Namen `::type` und ist notwendig[270], um dem Compiler mitzuteilen, dass es sich bei dem Namen um einen Typ handelt, ansonsten würde der Namen wie ein **`static`** deklariertes Klassenattribut behandelt. Die Schlüsselworte `typename` und `template` sind in diesem Zusammenhang nur innerhalb von Templates notwendig und zulässig.

13.5.8 Template Functors

Sollen bei der Verarbeitung der Typelist weitere Typen oder eine weitere Typelist zur Anwendung kommen, kann der Functor diese innerhalb definieren oder er wird als Template ausgelegt und bekommt die weiteren Typen vor seiner Verwendung übergeben.
Die Parameter des Functors stehen in `apply` zur Verfügung[271]. In Listing 221 auf der nächsten Seite hat der Functor nicht nur einen weiteren Typparameter `SystemTraits` sondern auch noch ein Template das 3 Argumente erwartet und das in `apply` aufgerufen wird. `ForEachClassIn` ruf `apply` mit zwei Argumenten auf, der TemplateFunctor delegiert den Aufruf an sein Argument `Creator` und ergänzt dabei den Aufruf um das Argument `SystemTraits`. `Creator` kann als Zeiger auf ein Template betrachtet werden. Wenn das Template, auf das `Creator` zeigt, spezialisiert ist, werden die Spezialisierungen entsprechen ausgewählt.

[268] [Ale09] 3.13.3 Generating Linear Hierarchies
[269] section 9.9.1 auf Seite 192
[270] section 8.8 auf Seite 176
[271] section 9.9 auf Seite 191

Listing 221: Ein Template als Functor

```
template<typename SystemTraits,
        template<typename TYPE, typename BASE, typename SYSTEMTRAITS> class Creator
            >
struct TemplateFunctor{

    template<typename Type, typename Base>
    struct apply{
        typedef typename Creator<Type, Base, SystemTraits>::type type;
    };
};
```

13.5.9 Variadic Klassentemplate MakeTypelist

Die *type-function* `MakeTypelist` wurde in Loki auf der Basis einer großen Anzahl Template Typeparametern realisiert, die alle mit `Loki::NullType` als Defaultargument ausgestattet waren. Mit Variadic Templates seit C++11 ist das nicht mehr nötig. Das Listing 222 zeigt eine mögliche Implementierung.

Listing 222: Die type-function MakeTypelist

```
template<class T, class ...List>
struct MakeTypelist{
   typedef typename MakeTypelist<List...>::type tail;
   typedef Loki::Typelist<T, tail> type;
};

template<class T>
struct MakeTypelist<T>
{
   typedef Loki::Typelist<T, Loki::NullType> type;
};
```

Die *type-function* `MakeTypelist` hat einen *type-parameter* `class T` und einen Parameter Pack als Parameter `class... List`. Sie wendet sich selbst auf den Parameter Pack an bis nur noch ein Typ darin enthalten ist: `MakeTypelist<List...>::type`. Dabei wird bei jedem Aufruf jeweils der erste Parameter `T` mit dem ersten Element der Liste und `...List` mit den verbleibenden Elementen verknüpft[272]. Die Spezialisierung für nur einen Typ beendet die Rekursion und definiert das Ende der Typlist mit dem `Loki::NullType`.

Die Bedeutung der `Loki::Typelist` wird aber durch Parameter Packs veringert. Mit einem Template wie in Listing 223 auf der nächsten Seite können Templates für diesen Typ spezialisiert werden und können damit eine an anderer Stelle definierte `Typelist` als Argument erwarten.

[272] initialisiert könnte mißverstanden werden

Listing 223: Typelist mit Parameter Pack

```
template<class ...Types>
struct Typelist;

// primary Template
template<class ...MyTypes
struct MyTemplate{
   /* some definitions */
};

// using Typelist zur Spezialisierung
templates<class ...MyTypes>
struct MyTemplate<Typelist<MyTypes...>
//delegates to primary
: MyTemplate<MyTypes...>{};


class T1; class T2; class T3;
using typelist = Typelist<T1, T2, T3>;

...
MyTemplate<typelist> myTemplates;
```

Die Spezialisierung von
`MyTemplate<Typelist<MyTypes...> : MyTemplate<MyTypes...>{};` erbt vom primary Template, es delegiert damit die Implementierung an das primary Template[273].

13.5.10 Variadic Baustein zur Erzeugung von Typen

Mit Variadic Template Parametern, wird die Erzeugung der Basisklasse für den *cyclic visitor* etwas einfacher. Mit Variadic Templates wird die *type-function* `ForEachClassIn` und der Functor mit `apply` ersetzt.

Das dafür notwendig Template ist in Listing 224 skizziert.

Listing 224: Inherit from Pack und visits

```
template<class ToVisit, class... Rest>
struct InheritFromPack : public InheritFromPack<Rest...>{
public:
   virtual void visit(ToVisit& v) = 0;
   using InheritFromPack<Rest...>::visit;
};
template<class ToVisit>
struct InheritFromPack<ToVisit>{
public:
   virtual void visit(ToVisit& v) = 0;
   virtual std::string toString() const = 0;
};
```

[273] https://github.com/GerdHirsch/Cpp-VisitorFrameworkCyclicAcyclic/wiki/Typelist

```
13 template<class ToVisit, class...Rest>
14 using visits = InheritFromPack<ToVisit, Rest...>;
```

Der Name des Templates `InheritFromPack` ist an der verwendeten C++ Technik orientiert. Der `using` Alias `visits` ist an der Problem Domain ausgerichtet. Die Funktionsweise ist dieselbe wie bei dem Template `InheritFrom` aus Listing 216 auf Seite 224. Die Spezialisierung für ein Argument erbt von keiner Basisklasse, damit endet die Rekursion.

Die Anwendung zeigt Listing 225. VisitorBase hat in diesem Fall dieselbe Struktur wie in Listing 218 auf Seite 225 skizziert.

Listing 225: Die Anwendung der generierten Basisklasse

```
1  class A; class B; class C;
2  using VisitorBase = visits<A, B, C>;
3  //================================================
4
5  #include "MyTypes.h"
6
7  class MyVisitor : public VisitorBase {
8  public:
9     void visit(A& v) override {
10       std::cout << "MyVisitor::visit(" << v.toString() << "& a)" << std::endl;
11    }
12    void visit(B& v) override {
13       std::cout << "MyVisitor::visit(B&)" << std::endl;
14    }
15    void visit(C&) override {
16       std::cout << "MyVisitor::visit(C&)" << std::endl;
17    }
18
19    std::string toString() const override{ return "MyVisitor"; }
20 };
```

Für die Definition von `VisitorBase` wird nur eine *forward declaration* der Typen benötigt. Der konkrete `MyVisitor` benötigt für den Zugriff auf die Schnittstelle in `visit(ToVisit&)override` den Typ der besucht wird.

Der *override specifier* zeigt, dass tatsächlich alle Operationen in der Basisklasse vorhanden sind.

Das Listing 226 zeigt die Verwendung von MyVisitor. Die Definition des Adapters wird in section 15.2 auf Seite 246 beschrieben.

Listing 226: MyVisitorAdapterApplication

```
1  void demoGenerateVisitorsCyclic(){
2     using namespace std;
3
4     cout << "=== demoGenerateVisitorsCyclic()" << endl;
5     MyVisitor myVisitor;
6     A a; B b; C c;
7
```

```
8     MyAdapter<A> adapterA(a);
9     MyAdapter<B> adapterB(b);
10    MyAdapter<C> adapterC(c);
11
12    adapterA.accept(myVisitor);
13    adapterB.accept(myVisitor);
14    adapterC.accept(myVisitor);
15
16    cout << "===  end demoGenerateVisitors()" << endl;
17 }
18 /*
19 DemoForEachClassIn
20 === demoGenerateVisitorsCyclic()
21 CyclicAdapter::A::accept: MyVisitor
22 MyVisitor::visit(A& a)
23 CyclicAdapter::B::accept: MyVisitor
24 MyVisitor::visit(B&)
25 CyclicAdapter::C::accept: MyVisitor
26 MyVisitor::visit(C&)
27 ===  end demoGenerateVisitors()
28 */
```

Anstatt die `visit` Operationen pur virtuell zu deklarieren, könnten sie auch wie in Listing 227 definiert werden. Damit müssten die Visitoren nicht für alle visit Operationen eine Methode zur Verfügung stellen. Die Ausgabe sollte aber nicht wie im Beispiel, “fest verdrahtet” sein, sondern über eine geeignete LoggingPolicy vom Benutzer bestimmt werden können.

Listing 227: Default Methoden für visit

```
1  template<class ToVisit, class... Rest>
2  struct InheritFromPack : public InheritFromPack<Rest...>{
3     virtual void visit(ToVisit& visitable){
4        std::cout
5           << this->toString()
6           << "::visit("
7           << typeid(visitable).name()
8           << " &) is not implemented!"
9           <<
10       std::endl;; }
11    ...
12 };
```

Die Ausgabe für eine nicht implementierte Methode `visit(B&)` könnte wie in Listing 228 auf der nächsten Seite aussehen. `typeid(..).name()` liefert einen kryptischen Namen für B.

Listing 228: Die Ausgabe der visit default Methoden

```
DemoForEachClassIn
=== demoGenerateVisitorsCyclic()
CyclicAdapter::A::accept: MyVisitor
MyVisitor::visit(A& a)
CyclicAdapter::B::accept: MyVisitor
MyVisitor::visit(N12_GLOBAL__N_11BE &) is not implemented!
CyclicAdapter::C::accept: MyVisitor
MyVisitor::visit(C&)
===  end demoGenerateVisitors()
```

Teil V

Design Pattern

14 Template Factory Method

14.1 Template-Method

In diesem Kapitel wird die Motivation für die beiden Patterns Template-Method und Factory-Method zusammen beschrieben, weil beide mit dem gleichen Beispiel motiviert werden können und weil sie häufig zusammen verwendet werden. Der Begriff *Template* in diesem Zusammenhang muss von demselben Begriff im Kontext von C++ unterschieden werden.

14.1.1 Name, Kategorie, Synonyme

Name: Template Method

Kategorie: Class Behavioral

14.1.2 Problembeschreibung

Intention
Die Struktur eines Algorithmus in einer sogenannten Template Methode festlegen und gleichzeitig die einzelnen Schritte variabel gestalten. Subklassen können die Ausprägung der einzelnen Schritte, die der Algorithmus verwendet, durch überschreiben der polymorhen Methoden, der so genannten Hook-Methods[274], definieren.

Motivation
Ein Framework zur Verwaltung und Bearbeitung von Dokumenten kann die allgemeinen Benutzerinteraktionen definieren, kennt aber nicht alle Details, wie diese ausgeführt werden sollen.

Ein Dokument neu erzeugen ist eine solche Aktion bei der klar ist, dass das Dokument in der Liste der geöffneten Dokumente erscheinen sollte und wann es erzeugt werden muss, usw. aber nicht, wie ein solches Dokument erzeugt werden kann, bzw. nicht von welcher konkreten Klasse das Objekt sein soll.

Die Template Methode newDocument() definiert welche Schritte in welcher Reihenfolge ausgeführt werden, die Subklassen (MyApplication, MyDocument) definieren wie die Schritte ausgeführt werden. Die Template-Methode newDocument() der Basisklasse erzeugt ein neues Dokument durch die Factory-Methode

[274] [Pre95]

createDocument(), die ein Objekt der Klasse MyDocument, die das Interface „Document“ implementiert, zurückliefert. Die Template-Methode und die Factory-Methode gehören zu demselben Objekt (der Klasse MyApplication), weil die Klasse MyApplication eine Subklasse der Klasse Application ist, in der die Template-Methode definiert ist. Eine Methode wie die Factory-Methode, die von der Basisklasse aufgerufen wird und in den abgeleiteten Klassen spezialisiert wird, wird auch als Hook-Methode bezeichnet, weil sie sich wie ein Haken in den Algorithmus einhakt. Hook-Methoden sind konventionellen Programmierern als Callback Funktionen bekannt.

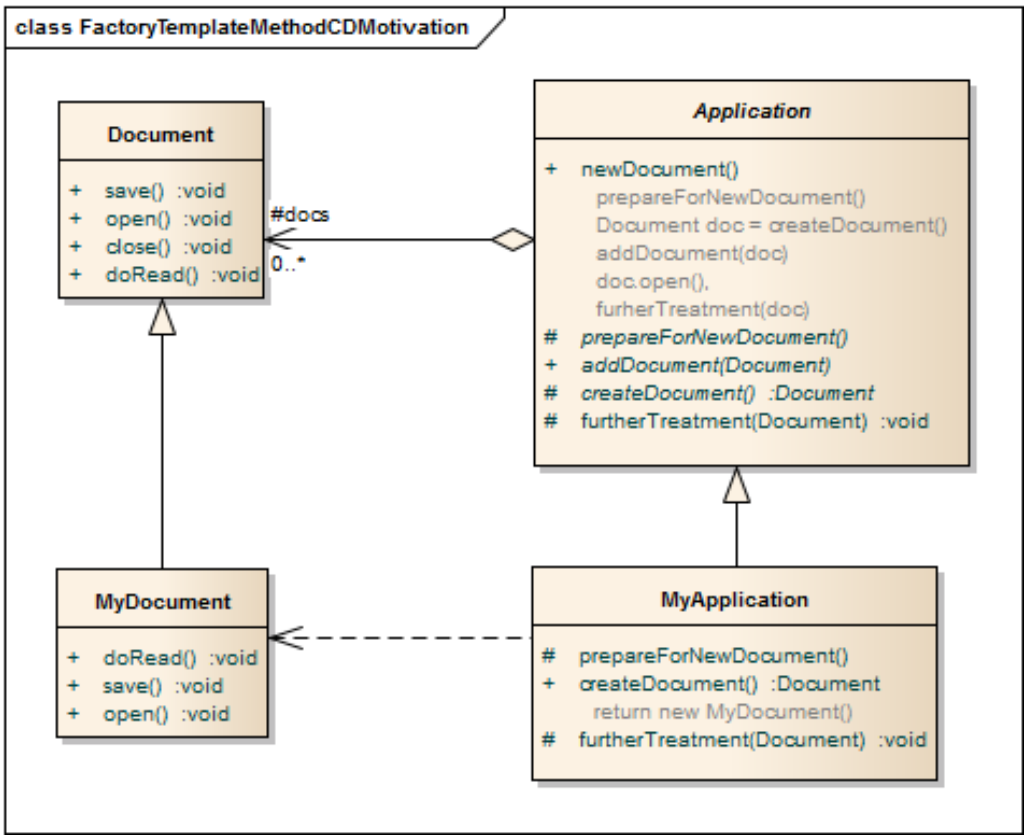

Diagramm 11: TemplateMethod und FactoryMethod ergänzen sich

Eine andere Möglichkeit ist, die Template-Methode und die Hook-Methoden in verschiedenen Objekten unter zu bringen. Die Template-Methode (newDocument) verwendet ein Interface (Document), das durch eine spezialisierte Klasse (MyDocument) implementiert wird. Das Objekt, das von createDocument() erzeugt wird, kann geöffnet, gespeichert, kopiert, usw. werden. Das spezielle Objekt der Klasse MyDocument weis, wie es sich öffnen kann, die Template-Methode weis wann es sich öffnen soll.

14.1.3 Lösungsbeschreibung

Die Struktur

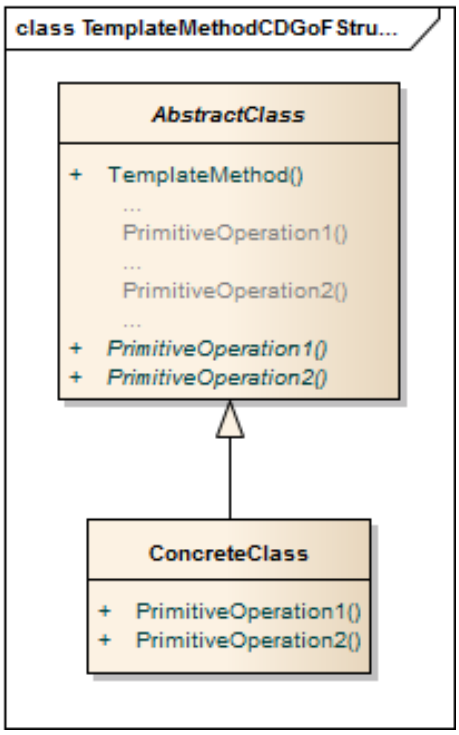

Diagramm 12: GoF Template-Method

Die Beteiligten
AbstractClass: definiert einige Operationen und Implementiert einen Algorithmus auf der Basis dieser Operationen, die Template-Methode.

ConcreteClass: implementiert einige Operationen.

Die Zusammenarbeit
Die abgeleiteten Klassen stellen für die verschiedenen Schritte des Algorithmus die Methoden zur Verfügung. Die Template-Methode führt diese in der gewünschten Reihenfolge aus.

Template-Methoden rufen typischer Weise folgende Methoden auf:

- Nicht polymorphe Methoden, die etwas konkretes tun, das nicht variiert werden darf
- Polymorphe Methoden (einfache Hook-Methoden), die etwas Konkretes tun, das variiert werden können soll, eine default-Implementierung steht zur Verfügung.
- Abstakte polymorphe Methoden, für die eine konkrete Implementierung zur Verfügung gestellt werden muss, weil es keine sinnvolle default-Implementierung gibt. Daraus ergibt sich, dass die Basisklasse abstrakt ist.
- Andere polymorphe Template-Methoden, die einen konkreten Ablauf definieren, so dass dieser von den Clients variiert werden kann
- Factory-Methoden, die abstrakt Objekte erzeugen (polymorpher Konstruktor)

14.1.4 Implementation

Mit C++ Templates kann das Pattern mit statischer Polymorphie realisiert werden[275].

14.1.5 Konsequenzen

Template-Method ist eine grundlegende Technik zur Wiederverwendung von Code insbesondere in Frameworks. Sie definieren das allgemeine Verhalten und ermöglichen Spezielles hinzu zufügen. Die Kontrollstruktur wird dadurch invertiert, was manchmal auch als „das Hollywood Prinzip“ bezeichnet wird (don´t call, wait for calling). Die Basisklasse bestimmt den Zeitpunkt, wann eine Operation gerufen wird, die Spezialisierung stellt nur die notwendige Methode zur Verfügung und wartet darauf, dass sie gerufen wird.

14.1.6 Bekannte Anwendungen

Sämtliche Frameworks basieren mehr oder weniger auf dieser Technik

14.1.7 Kombinationsmöglichkeiten

Factory-Method (section **??** auf Seite ??) als Hook-Method zur Erzeugung von Objekten.

Strategie (section 16.2 auf Seite 285) verwendet Delegation anstelle von Vererbung um dasselbe zu erreichen. Strategie ist auch während der Laufzeit variabel.

14.2 Template und Factory Method mit C++ Templates

Das Diagramm 13 auf der nächsten Seite skizziert eine mögliche Implementierung in C++ des Template Method Patterns. Die Basisklasse ist als C++ Template ausgelegt, das den Typ der Spezialisierung als Parameter erhält (Listing 229 auf Seite 241). Die Spezialisierungen erben von der Template Instanz:
`class Application : public Implementation<Application, MyDocument>{..};` (Listing 230 auf Seite 242)
Das entspricht dem sogenannten Curiosly Recuring Template Pattern (CRTP).

14.2.1 Aufruf der Hook Methoden der Spezialisierung

Die Methode `Implementation::TemplateMethod` in Listing 229 auf Seite 241 demonstriert,

[275] https://github.com/GerdHirsch/Cpp-TemplateFactoryMethod

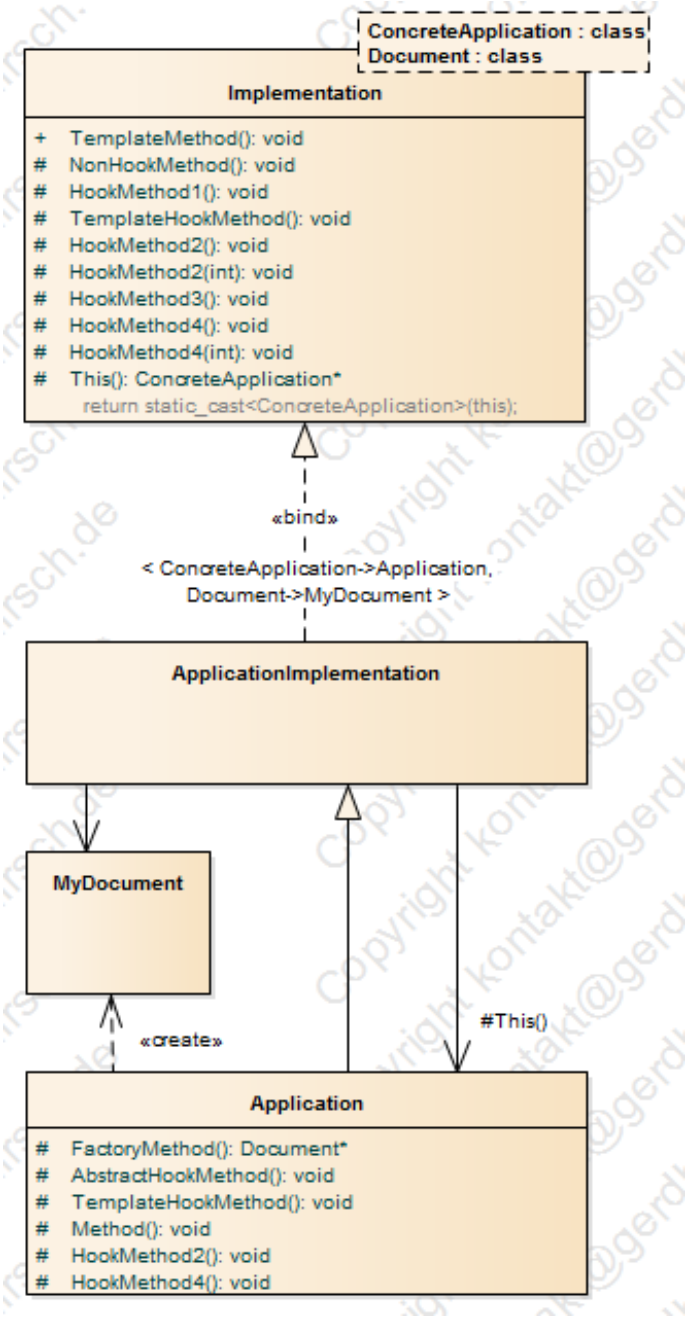

Diagramm 13: Statische Polymorphie mit C++ Templates

- wie Methoden aufgerufen werden (`this->NonHookMethod()`), die nicht in der Spezialisierung überschrieben werden können
- wie abstrakte nicht `virtual` Hook Methoden aufgerufen werden (`This()->AbstractHookMethod()` und `This()->FactoryMethod()`)
- und wie **nicht** `virtual` Hook Methoden der Spezialisierung aufgerufen werden können.

Wenn die Hook Methoden in der Spezialisierung nicht redefiniert sind, wie die Methode `This()->HookMethod1()`, wird die geerbte Methode der Basisklasse verwendet, ansonsten die der Spezialisierung z.B.: die Methode

`This()->TemplateHookMethod()`.

14.2.2 Der Scope

In den Methoden der Basisklasse `Implementation<..>` wird anstatt **`this`** die Memberfunktion `This()` verwendet.
Sie castet **`this`** in den Typ `ConcreteApplication`. Der Scope, in dem der Name der Methode gesucht wird ist dadurch der Typ `ConcreteApplication`. Weil der konkrete Typ bekannt ist, werden keine `virtual` Operationen benötigt, alle Methoden können von `inline` profitieren.

An die Stelle von `virtual` tritt Namensüberdeckung.
Darum muss für überladene Operationen wie z.B. `HookMethod2`, für die in der Spezialisierung nicht alle überladenen Varianten redefiniert sind, eine using Deklaration erstellt werden: `using HookMethod2`. Ohne die using Deklaration sind die anderen Überladungen der Methoden aus der Basisklasse in der Spezialisierung nicht sichtbar und der Compiler quitiert den Aufruf in `Implementation::TemplateMethod()` mit einer Fehlermeldung der Art: `error: no matching function for call to 'Application::HookMethod2(int)'`.

Die Sichtbarkeitsbereiche / Scopes sind ineinander eingebettet. Der Scope einer abgeleiteten Klasse ist in den Scope der Basisklasse eingebettet. Die Suche nach einem Namen beginnt in dem Scope, der beim Start der Suche explizit oder implizit vorgegeben wird. Mit `This()->` ist das der Scope der Spezialisierung `Application`. Die Suche bricht ab, wenn der Name gefunden ist. Die Suche erfolgt nicht auf der Basis der Signatur sondern basiert ausschließlich auf dem Namen. Der Name kann also durch jede Art von Element, z.B. eine Konstante, überdeckt werden, es muss nicht eine Art von Element (z.B. Methode) sein, nach der gesucht wird. Wird ein Element mit dem Namen gefunden, z.B. eine Memberfunktion mit einer falschen Signatur, endet die Suche und die Fehlermeldung ist das Ergebnis. Siehe auch section 6.20 auf Seite 60.

Wird der Name nicht gefunden, wird der nächste Scope, in dem der Aktuelle eingebettet ist, durchsucht. Das wird fortgesetzt bis zum globalen Scope als dem letzten der durchsucht wird.

Im Fall der `HookMethod4` ist keine using Deklaration notwendig, weil die nicht sichtbare Überladung `HookMethod4(int)` nur in der Methode `Implementation::TemplateHookMethod` verwendet wird. Für diese wird kein Code generiert, weil sie nicht benutzt wird. An ihrer Stelle wird die Methode mit demselben Namen aus der Spezialisierung benutzt und diese verwendet die Überladung der `HookMethod4(int)` nicht. Trotzdem sollte für alle Namen aus der Basisklasse, die in der Spezialisierung redefiniert werden, eine using Deklaration eingefügt werden.

In Templates verhält sich die Namensauflösung anders, der Scope eines Templates das von einem anderen erbt ist nicht in den Scope seiner Basisklasse eingebettet. Siehe auch section 8.9 auf Seite 177.

14.2.3 Abstrakte polymorphe Methoden

Methoden, für die keine sinnvolle default Implementierung zur Verfügung gestellt werden kann, werden im Template `Implementation<..>` einfach garnicht deklariert, wie z.B. `FactoryMethod():MyDocument` oder `AbstractHookMethod()`. Sie können aber trotzdem für den Typ
`ConcreteApplication` aufgerufen werden, wenn sie dort definiert sind. Ansonsten kann die Klasse `Application` nicht erzeugt werden. Das entspricht einer pure virtual Operation der dynamischen Poliymorhpie mit `= 0;` am Ende der Deklaration. Wird dabei in der Ableitung keine Methode für diese Operation definiert, können keine Objekte von dieser Klasse erzeugt werden.

14.2.4 Nicht polymorphe Methoden

Methoden die nicht redefiniert werden können sollen, also keine Hook Methoden sind, werden nicht mit `This()->NonHookMethod()` sondern mir `this->NonHookMethod()` in `Implementation` aufgerufen. Der Scope ist damit die Basisklasse, eine Methode mit gleichem Namen in der Spezialisierung wird nicht berücksichtigt. Das entspricht einer non `virtual` Methode.

14.2.5 Template Methoden als Hook Methoden

Eine Template Methode definiert einen Algorithmus als Abfolge von Aufrufen von Hook Methoden. Soll diese Abfolge variiert werden können, werden die Template Methoden ebenfalls als Hook Methoden zur Verfügung gestellt. Häufig wird dabei auf default Implementierungen anderer Hook Methoden aus der Basisklasse zurückgegriffen. Die Methode `Application::TemplateHookMethod` demonstriert, wie Methoden der Basisklasse aus der Application aufgerufen werden können. Dazu wird der Alias
`using base_type = Implementation<...>;` definiert, mit dem die Methoden vollqualifiziert ausgewählt und aufgerufen werden können: `base_type::HookMethod2()`. Die Methode `HookMethod3()` ist nicht "überschrieben"[276] und muss daher in `Application` nicht qualifiziert werden.

14.2.6 protected Hook Methoden

Die Hook Methoden sind meistens **`protected`** oder **`private`**. Damit diese Methoden aus `Application` von `Implementation<..>` aufgerufen werden können, wird diese als `friend` **`class`** `Implementation<..>` in der abgeleiteten Klasse (z.B. `Application`) deklariert.

[276] korrekt wäre: der Name ist nicht überdeckt!

14.2.7 Die Implementierung als Template

Das Template `Implementation<..>` in Listing 229 zeigt die Basisklasse als Template. Sie ist der Codegenerator für eine spezielle Applikation.

Listing 229: Template Method, die Basisklasse als Template

```
template<class ConcreteApplication, class Document> class Implementation{
   using this_type = ConcreteApplication;
public:
   void TemplateMethod(){
      std::cout << "Implementation::TemplateMethod()" << std::endl;
      this->NonHookMethod();
      This()->AbstractHookMethod();
      This()->HookMethod1();
      This()->HookMethod2(42);
      This()->HookMethod4();

      Document d = This()->FactoryMethod();
      This()->TemplateHookMethod();
   }
protected:
   this_type* This(){
      return static_cast<this_type*>(this);
   }
   void HookMethod1(){
      std::cout << "Implementation::HookMethod1()" << std::endl;
   }
   void TemplateHookMethod(){
      std::cout << "Implementation::TemplateHookMethod()" << std::endl;

      This()->HookMethod2();
      This()->HookMethod4(43);
   }
   void HookMethod2(){
      std::cout << "Implementation::HookMethod2()" << std::endl;
   }
   void HookMethod2(int){
      std::cout << "Implementation::HookMethod2(int)" << std::endl;
   }
   void HookMethod3(){
      std::cout << "Implementation::HookMethod3()" << std::endl;
   }
   void HookMethod4(){
      std::cout << "Implementation::HookMethod4()" << std::endl;
   }
   void HookMethod4(int){
      std::cout << "Implementation::HookMethod4(int)" << std::endl;
   }
   void NonHookMethod(){
      std::cout << "Implementation::NonHookMethod()" << std::endl;
   }
```

```
};
```

Listing 230: Die Spezialisierung

```
class MyDocument{
public:
   MyDocument(){
      std::cout << "MyDocument::MyDocument()" << std::endl;
   }
};

class Application : public Implementation<Application, MyDocument>{
   using base_type = Implementation<Application, MyDocument>;
   friend base_type;
protected:
   MyDocument FactoryMethod(){
      return MyDocument();
   }
   void TemplateHookMethod(){
      std::cout << "Application::TemplateHookMethod()" << std::endl;
      HookMethod2();
      base_type::HookMethod2();
      Method();
      HookMethod3();
   }
   void Method(){
      std::cout << "Application::Method()" << std::endl;
   }
   using base_type::HookMethod2;
   void HookMethod2(){
      std::cout << "Application::HookMethod2()" << std::endl;
   }
   void HookMethod4(){
      std::cout << "Application::HookMethod4()" << std::endl;
   }
   void NonHookMethod(){
      std::cout << "Application::NonHookMethod()" << std::endl;
   }

};
```

Listing 231: Die Anwendung und Ausgabe

```
int main(){
   cout << "TemplateFactoryMethod" << endl;

   Application app;
   app.TemplateMethod();
}
//Ausgabe:
=== TemplateFactoryMethod ===
Implementation::TemplateMethod()
```

```
10 Implementation::NonHookMethod()
11 Application::AbstractHookMethod()
12 Implementation::HookMethod1()
13 Implementation::HookMethod2(int)
14 Application::HookMethod4()
15 MyDocument::MyDocument()
16 Application::TemplateHookMethod()
17 Application::HookMethod2()
18 Implementation::HookMethod2()
19 Application::Method()
20 Implementation::HookMethod3()
```

15 Visitor

15.1 Acyclic Visitor

15.1.1 Name, Kategorie

Name: Acyclic Visitor

Kategorie: Object Behavior

15.1.2 Problembeschreibung

Intention
Der Acyclic Visitor eliminiert die Nachteile, die sich aus den zyklischen Abhängigkeiten des Visitor Patterns ergeben und ermöglicht das Hinzufügen von neuen Klassen in die Objektstruktur, ohne alle bestehenden Visitor-Klassen anpassen zu müssen.

15.1.3 Lösungsbeschreibung

Die Struktur

Der Kern des Acyclic Visitors basiert auf Mehrfachvererbung und auf dynamischer Typprüfung.

Die Beteiligten
Visitor ist ein degeneriertes Interface ohne Operationen. Es dient lediglich als Parametertyp für die Operation `accept(Visitor v)` im Interface Visitable.

ConcreteElement: jede Klasse definiert ein Visitor Interface mit einer `visit(ConcreteElement )` Operation. Diese Interfaces sind vollständig voneinander unabhängig.

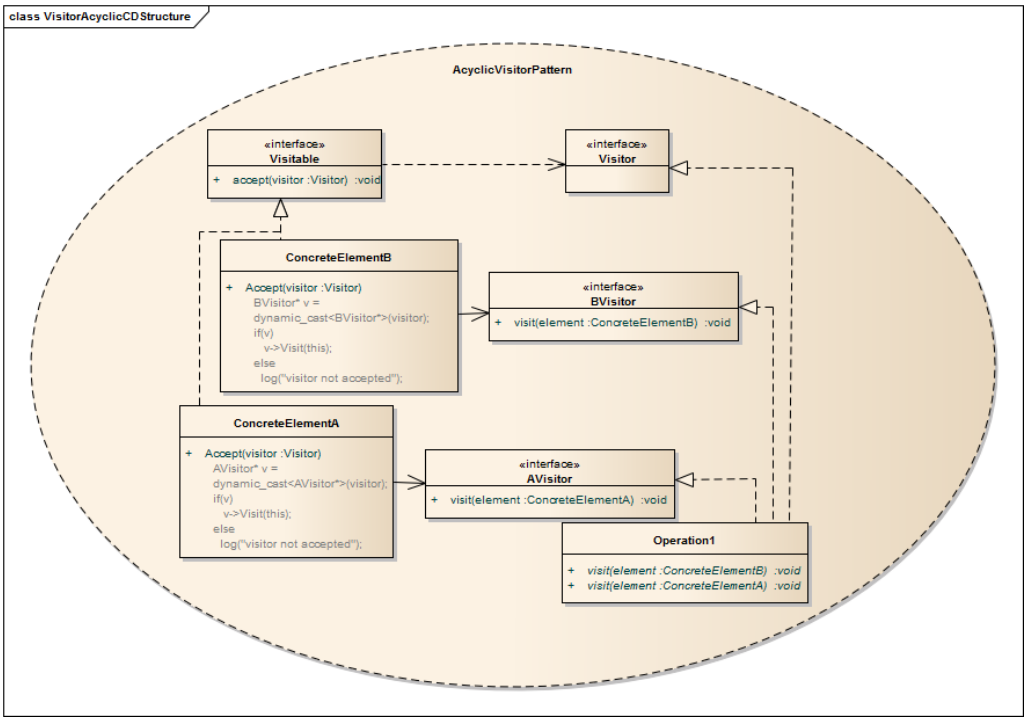

Diagramm 14: Acyclic Visitor Pattern

Operation1: ist ein konkreter Visitor der für die Klassen, die er besuchen möchte, die jeweiligen Interfaces implementiert. Außerdem implementiert er das leere Interface Visitor um von den Visitables akzeptiert zu werden.

Die Zusammenarbeit
Wie beim GoF Visitor muss über die Menge der Objekte iteriert werden und jedem Objekt in der Menge die Nachricht `accept(visitor)` gesandt werden. Die Objekte versuchen den Visitor in einen diesem Elementtyp entsprechenden Visitor zu wandeln (`dynamic_cast<>`). Wenn das möglich ist, wird dem Visitor die Nachricht `visit(this)` gesandt.

AdapterVisitor

Mit Hilfe eines Adapters lassen sich Objektstrukturen besuchen, die nicht für den Visitor vorbereitet sind.

15.1.4 Implementation

Mit C++ Templates kann ein Framework gemäß dem Template Method Pattern[277] erstellt werden.

15.1.5 Konsequenzen

Die positiven Konsequenzen sind dieselben wie für den GoF-Visitor.

[277] section 14.1 auf Seite 234 siehe Projekte Visitor_ im Workspace DesignPatternWorkspace

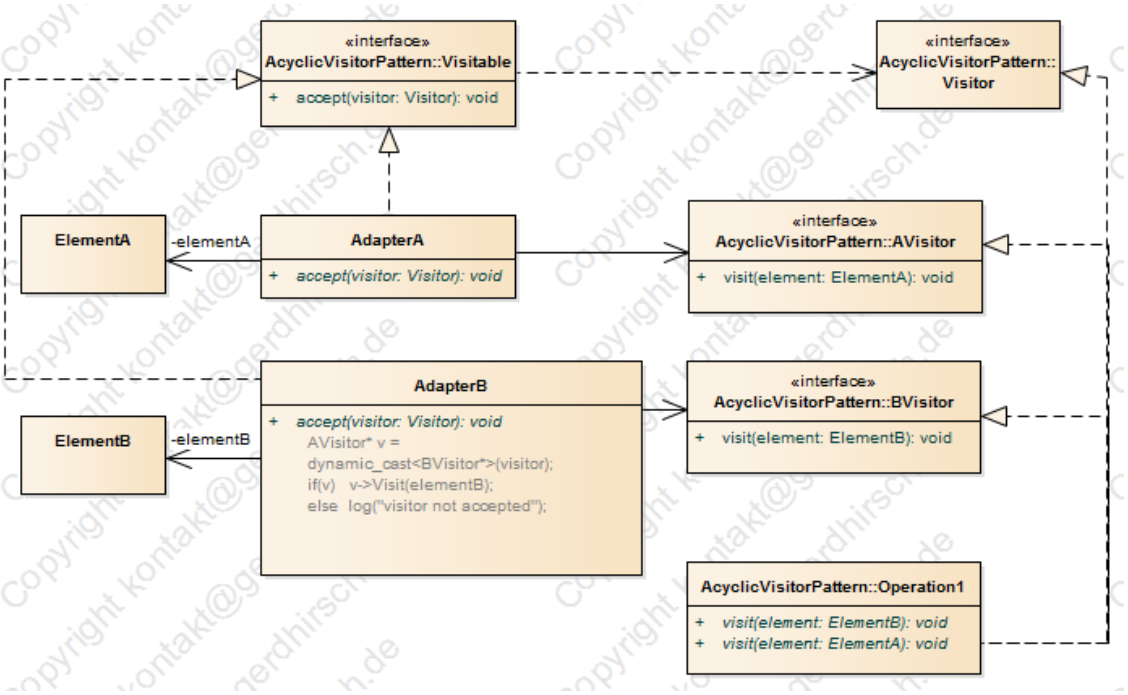

Diagramm 15: Adapter ermöglicht den Visitor

Der Acyclic-Visitor ist aber leichter erweiterbar als sein GoF Vertreter. Die konkreten Visitor-Klassen müssen allerdings ebenfalls angepasst werden, wenn sie neu hinzugekommene Visitable-Elemente besuchen sollen. Das ist eine Fehlerquelle, da an dieser Stelle keine statische Compiler Unterstützung vorhanden ist. Die Objekte müssen in irgendeiner Form den Besuch eines Visitors, der nicht akzeptiert wurde, zur Laufzeit protokolieren. Das ist der Preis für die dynamischen Konzepte: keine statische Prüfung. Beim GoF-Visitor müsste eine neue Operation Visit(neuerTyp) in der abstrakten Basisklasse Visitor deklariert werden und der Compiler würde jeden vorhandenen Visitor, der keine Methode dafür zur Verfügung stellt, als eine abstrakte Klasse behandeln.

Andererseits ist es mit dem Acyclic-Visitor möglich, die Objektstruktur nur partiell zu besuchen, indem nur für die Elemente an denen Interesse besteht, das Visitor Interface implementiert wird. Mit dem GoF Visitor Pattern müssten leere Methoden zur Verfügung gestellt werden.

Java: `AVisitor v = obj instanceof AVisitor ? Avisitor(obj): null;`

C++: In der Visitor-Klasse muss der Destruktor `virtual` deklariert werden um den Cross-Cast mit dem `dynamic_cast` zu unterstützen.
`dynamic_cast<AVisitor*>(&visitor)` kann bezogen auf die Laufzeit teuer sein. Die Kosten können je nach Klassenhierarchie variieren. Daher ist das Pattern für kritische Echtzeit-Anwendungen nicht geeignet. In manchen Umgebungen steht RTTI und/oder Multiple Inheritance nicht zur Verfügung, wodurch dieses Pattern nicht anwendbar ist. Der CrossCast in den accept Methoden kann wie folgt begründet werden: Das OCP bleibt gewährleistet! Sowohl

1. neue visitable Klassen als auch
2. neue Visitor Klassen

können hinzugefügt werden ohne bestehenden Code ändern zu müssen.

15.2 Acyclic Visitor in C++

Am Beispiel von verschiedenen Modems, die für verschiedene Umgebungen konfiguriert werden sollen, wird in diesem Kapitel, eine generisches Framework für das VisitorPattern entwickelt.
Am Ende steht ein kompaktes templatebasiertes Framework, das die meisten Bedürfnisse bzgl. des Visitor Patterns bedient und das Anpassungen an weitere Bedürfnisse durch ganz unterschiedliche *Hooks* auf verschiedenen Ebenen erlaubt.

15.2.1 Die Interfaces

Die benötigten Interfaces in Listing 232 sind in den Headern `Visitor.h` und `Visitable.h` definiert. Das Interface `Visitor` ist leer. `toString()` dient nur der Nachvollziebarkeit für die Ausgaben der Beispielprogramme.

Listing 232: Acyclic Visitor, Interfaces

```
1  class Visitor
2  {
3  public:
4     virtual ~Visitor(){};
5     virtual std::string toString() const = 0;
6  };
7  class Visitable
8  {
9  public:
10    virtual ~Visitable(){}
11    virtual void accept(Visitor& visitor) = 0;
12 };
```

15.2.2 Visitable, eine einfache Implementierung

Eine visitable Klasse könnte wie in Listing 233 auf der nächsten Seite implementiert werden. Sie definiert das Interface, das von einem Visitor erwartet wird: `class Visitor{..}` als innere Klasse und die Methode `accept(::Visitor&)`. Die Methode muss immer auf dieselbe Weise implementiert werden:

1. den `::Visitor` in den speziellen `ConcreteVisitable::Visitor` für dieses Element casten
2. bei Erfolg dem visitor eine Nachricht senden: `v->visit(*this)`, zur besseren Nachvollziehbarkeit, vorher eine Action ausführen, z.B. einen log schreiben
3. `else`: bei Misserfolg eine Action durchführen, z.B. einen log schreiben

Das entspricht einer Template Method aus section 14.1 auf Seite 234, bisher ohne Hook Methoden.

Listing 233: Ein konkretes Visitable

```
class HayesModem :
   public Visitable
{
public:
   using ConcreteVisitable = HayesModem;

   class Visitor
   {
   public:
      virtual void visit(ConcreteVisitable& modem) = 0;
   };

   virtual void accept(::Visitor& visitor)
   {
      ConcreteVisitable::Visitor* v =
         dynamic_cast<ConcreteVisitable::Visitor*>(&visitor);
      std::cout << std::endl;
      if(v)
      {
         std::cout << toString() << " accepted: " << visitor.toString()
            << std::endl;
         v->visit(*this);
      }
      else
      {
         std::cout << toString() << " did not accept " << visitor.toString()
            << std::endl;
      }
   }

   std::string toString() const
   {
      return "HayesModem";
   }
};
```

Die Methode `accept(::Visitor&)` könnte nicht in der Basisklasse `Visitable` implementiert werden, weil dort **`this`** vom Typ `Visitable*const` wäre und nicht vom Typ des konkreten Visitables (z.B. `HayesModem*`). Damit würde der Overload Mechanismus für Operationen nicht greifen: `visit(HayesModem&)`.

15.2.3 Visitor, Implementierung

Ein konkreter Visitor könnte wie in Listing 234 auf der nächsten Seite implementiert werden. In der Liste der Basisklassen wird das Interface `Visitor` und alle `Visitable::Visitor` Klassen gelistet, deren Objekte besucht werden sollen.

Da der konkrete Visitable Typ bekannt ist (z.B. `HayesModem`), muss die Methode `modem.toString()` nicht `virtual` sein.

Listing 234: Ein konkreter Visitor

```
class ConfigureMacModemVisitor :
   public Visitor,
   public HayesModem::Visitor
{
public:
   virtual void visit(HayesModem& modem)
   {
      std::cout << "-> " << toString() << " visits " << modem.toString()
         << std::endl;
   }

   virtual std::string toString() const
   {
      return "ConfigureMacModemVisitor";
   }
};
```

15.2.4 Die Anwendung des Acyclic Visitors

Die Anwendung ist in Listing 235 gezeigt. Dem Algorithmus `for_each` wird eine Lambda Expression übergeben die jeweils `visitable->accept(visitor)` für das übergebene `Visitable` Objekt aufruft.

Listing 235: Die Anwendung der Acyclic Visitors

```
using SharedPointer = std::shared_ptr<Visitable>;
using Visitables = std::vector<SharedPointer>;

void demoVisitor(Visitor& visitor, Visitables& visitables){
   std::cout << std::endl << "==== " << visitor.toString() << " ====" << std::endl;

   std::for_each(visitables.begin(), visitables.end(),
         [&visitor](SharedPointer& visitable){ visitable->accept(visitor); });
}

int main()
{
   cout << "Visitor AcyclicVisitor" << endl;
   Visitables visitables;

   visitables.push_back(SharedPointer(new ElsaModem));
   visitables.push_back(SharedPointer(new HayesModem));
   visitables.push_back(SharedPointer(new ZoomModem));

   ConfigureDOSModemVisitor dosVisitor;
   ConfigureUnixModemVisitor unixVisitor;
```

```
22    ConfigureMacModemVisitor macVisitor;
23
24    demoVisitor(dosVisitor, visitables);
25    demoVisitor(unixVisitor, visitables);
26    demoVisitor(macVisitor, visitables);
27 }
```

In Listing 236 ist die Ausgabe des Programms abgebildet. Die `accept` Methoden der verschiedenen `Visitable`s geben die jeweigen Konfigurationsdaten vor und nach dem Besuch aus, soweit welche vorhanden sind.

Listing 236: Die Ausgabe

```
1  Visitor AcyclicVisitor
2  ElsaData::ElsaData() m_Value: default
3  ZoomData::ZoomData() m_Value: default
4
5  ==== ConfigureDOSModemVisitor ====
6
7  ElsaModem accepted ConfigureDOSModemVisitor
8  => Data: default
9  -> ConfigureDOSModemVisitor visits ElsaModem
10 => Data: DOS Configuration
11
12 HayesModem accepted: ConfigureDOSModemVisitor
13 -> ConfigureDOSModemVisitor visits HayesModem
14
15 ZoomModem accepted ConfigureDOSModemVisitor
16 => Data: default
17 -> ConfigureDOSModemVisitor visits ZoomModem
18 => Data: DOS Configuration
19
20 ==== ConfigureUnixModemVisitor ====
21
22 ElsaModem did not accept ConfigureUnixModemVisitor
23
24 HayesModem accepted: ConfigureUnixModemVisitor
25 -> ConfigureUnixModemVisitor visits HayesModem
26
27 ZoomModem accepted ConfigureUnixModemVisitor
28 => Data: DOS Configuration
29 -> ConfigureUnixModemVisitor visits ZoomModem
30 => Data: UNIX Configuration
31
32 ==== ConfigureMacModemVisitor ====
33
34 ElsaModem did not accept ConfigureMacModemVisitor
35
36 HayesModem accepted: ConfigureMacModemVisitor
37 -> ConfigureMacModemVisitor visits HayesModem
38
39 ZoomModem did not accept ConfigureMacModemVisitor
```

15.2.5 Visitable, Zugriff auf nicht öffentliche Elemente

Um nur den konkreten `Visitor`en Zugriff auf die nicht öffentlichen Elemente der `Visitable`s zu gewähren, können **`protected`** Klassenoperationen (**`static`**) in den `ElementVisitor`s wie in Listing 237 `setElsaData(modem, value)` zur Verfügung gestellt werden.
Die konkreten Visitoren erben von den Interfaces und können damit auf die protected Elemente zugreifen. Der Kontext, in diesem Fall das Modem, wird als erster Parameter übergeben. Damit wird den Visitoren Zugriff auf die nicht öffentlichen Elemente der Visitables gewährt.

Listing 237: Schnittstelle für nicht öffentliche Elemente der Visitables

```
class ElsaModem :
   public Visitable
{
public:
   using ConcreteVisitable = ElsaModem;

   ElsaModem();

   class Visitor
   {
   public:
      virtual void visit(ConcreteVisitable& modem) = 0;

   protected:
      static void setElsaData(ConcreteVisitable& modem, const std::string& value)
      {
         modem.setElsaData(value);
      }
   };

   virtual void accept(::Visitor& visitor);

   std::string toString() const { ... }
private:
   void setElsaData(const std::string& value);
   ...
};
```

Und der dazugehörige Client ist in Listing 238 abgebildet.

Listing 238: Der Zugriff auf nicht öffentliche Elemente der Visitables

```
class ConfigureDOSModemVisitor :
   public Visitor,
   public HayesModem::Visitor,
   public ZoomModem::Visitor,
   public ElsaModem::Visitor
{
public:
   virtual void visit(HayesModem& modem);
```

```
9     virtual void visit(ZoomModem& modem);
10    virtual void visit(ElsaModem& modem)
11    {
12       std::cout << "-> " << toString() << " visits " << modem.toString()
13                 << std::endl;
14       setElsaData(modem, "DOS Configuration");
15    }
16    ...
17 };
```

15.3 Die Entwicklung eines Framework für das Acyclic Visitor Pattern

Die Implementierung auf der Visitable Seite ist immer gleich! Auf der Basis von templates kann sie daher generiert werden.

15.3.1 Die generischen Interfaces

Das Listing 239 zeigt die benötigten Interfaces für das Acyclic Visitor Pattern. Das Interface `Visitor` und `Visitable` sind dieselben wie vorher.

Listing 239: Die generische Interfaces

```
1  class Visitor
2  {
3  public:
4     virtual ~Visitor();
5     virtual std::string toString() const = 0;
6  };
7
8  class Visitable
9  {
10 public:
11    virtual ~Visitable() = 0;
12    virtual void accept(Visitor& visitor) = 0;
13 };
14 // -------- generisches Visitor Interface -----------
15 class DefaultAccessor{};
16
17 template<class VISITABLE, class ACCESSOR = DefaultAccessor>
18 class ElementVisitor : protected ACCESSOR
19 {
20 public:
21    virtual ~ElementVisitor(){}
22    virtual void visit(VISITABLE& visitable) = 0;
23 };
```

Das Template `ElementVisitor` erzeugt das Interface für ein konkretes Visitable von dem der konkrete Visitor erbt, wenn er Objekte dieser Klasse besuchen möchte. Dadurch entsteht eine Vererbungshierarchie wie in Diagramm 16 dargestellt. Ein Beispiel für die Anwendung des Templates ist in Listing 241 auf Seite 254 dargestellt: `using Visitor = ...`
Die Implementierung des Visitors ist dieselbe wie in Listing 234 auf Seite 248.

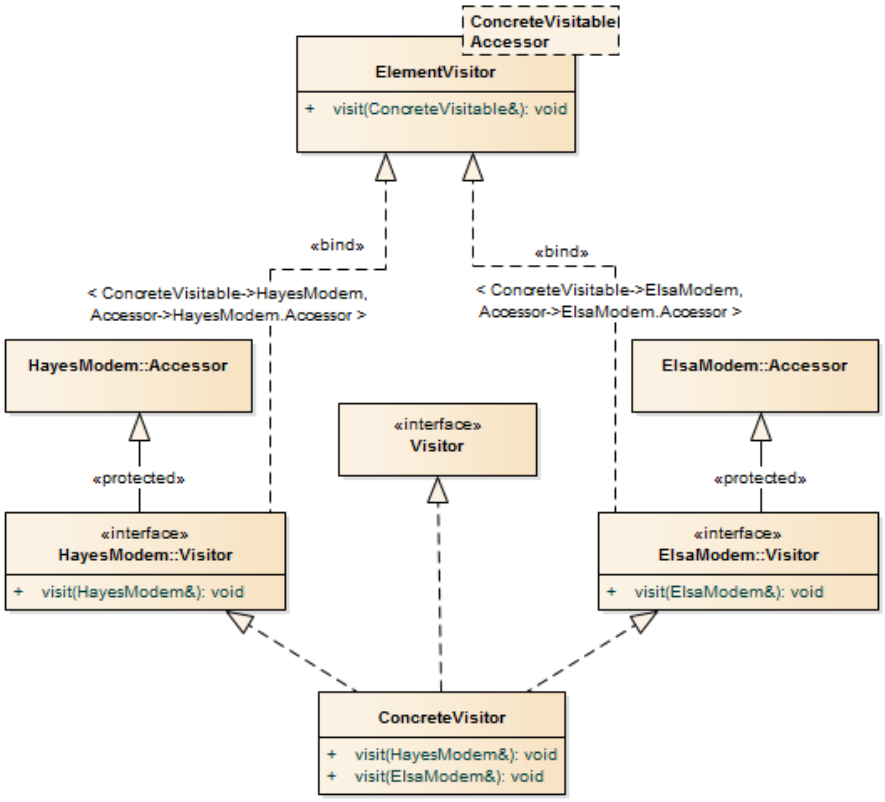

Diagramm 16: Die Vererbungshierarchie der konkreten Visitors

15.3.2 Protected Inheritance

Der ACCESSOR ist eine Klasse die dem konkreten Visitor Zugriff auf die nicht öffentliche Schnittstelle des VISITABLEs über **`protected`** Klassenmethoden ermöglicht.

Ein ElementVisitor ist aber kein Accessor! Deshalb wird hier **`protected` Inheritance** verwendet. Die konkreten Visitoren haben Zugriff auf den Accessor und auf dessen protected Methoden, ein unabsichtlicher cast `Accessor* a = &visitor;` ist aber nicht möglich und daher wird auch kein `virtual` Destruktor in den Accessorklassen benötigt.

15.3.3 VisitableImpl<..>, eine erste generische Implementierung

Da die Methode `accept(Visitor&)` immer gleich implementiert werden muss, kann sie in einer templatisierten Basisklasse `VisitorImpl<ConcreteVisitable>` wie in

Listing 240 definiert werden. Das wäre in einer nicht templatisierten Basisklasse nicht möglich, weil **this** vom Typ `VisitableImpl * const` ist und damit die überladene Operation im Visitor `visit(ConcreteVisitable&)` nicht ausgewählt werden würde.
Der Typ von **this** muss daher in den Typ `ConcreteVisitable`, der als Templateparameter übergeben wird, gecastet werden:
`ConcreteVisitable* visitable = getVisitable();`. Die Methode `getVisitable()` führt den cast durch:
return `static_cast<ConcreteVisitable*>(this)`. Das entspricht einem Downcast, da `VisitableImpl<ConcreteVisitable>` die Basisklasse von `ConcreteVisitable` ist, z.B. in Listing 241 auf der nächsten Seite.
Trotzdem genügt ein `static_cast<ConcreteVisitable*>` da an dieser Stelle der konkrete Typ von **this** bekannt ist.

Der Parameter `LoggingPolicy` wird in section 15.6 auf Seite 275 ausführlicher behandelt. Er stellt die Operationen `logAccepted(..)` und `logNotAccepted(..)` die in `accept(..)` verwendet werden zur Verfügung.

Listing 240: eine einfache generische Visitable Imlementierung

```
template<
   class ConcreteVisitable_,
   class LoggingPolicy = DefaultLoggingPolicy
   >
class VisitableImpl : public Visitable, public LoggingPolicy{
public:
   using ConcreteVisitable = ConcreteVisitable_;

   ConcreteVisitable* getVisitable() {
      return static_cast<ConcreteVisitable*>(this);
   }

   void accept(::Visitor& visitor){
      using Visitor = typename ConcreteVisitable::Visitor;

      Visitor* v = dynamic_cast<Visitor*>(&visitor); //crosscast
      ConcreteVisitable* visitable = getVisitable();
      if(v){
         this->logAccepted(*visitable, visitor);
         v->visit(*visitable);
      } else {
         this->logNotAccepted(*visitable, visitor);
      }
   }
};
```

15.3.4 Ein Name für einen Typ als Hook

Der Name `Visitor` für das Interface ist Konvention und muss von den Klassen, die das Template als Basisklasse verwenden, eingehalten werden. Der Zugriff

auf diesen Typ in der Methode `accept(..)` erfolgt über den konkreten Typ des Visitables: `ConcreteVisitable`. Über diesen Mechanismus können Typen für die weitere Verarbeitung in Templates zur Verfügung gestellt werden.

15.3.5 Ein konkretes Visitable mit VisitableImpl<..>

Ein konkretes Visitable könnte damit wie in Listing 241 implementiert werden. Die innere Klasse `ElsaModem::Accessor` hat Zugriff auf die nicht öffentlichen Elemente der umgebenden Klasse `ElsaModem`. Der Zugriff wird an den konkreten Visitor über die **protected** Klassenmethoden weiter gegeben, weil `ElementVisitor<..>` von Accessor und der konkrete Visitor von `ElsaModem::Visitor` erbt.

Die Implementierung der konkreten Visitors sieht immer noch wie in Listing 238 auf Seite 250 aus.

Listing 241: Anwendung des generischen Visitable

```
1  class ElsaModem :
2     public VisitableImpl<ElsaModem>
3  {
4  public:
5     class Accessor
6     {
7     protected:
8        static void setElsaData(ElsaModem& modem, const std::string& value);
9        static std::string const& getElsaData(ElsaModem& modem);
10    };
11    ElsaModem();
12    ...
13    using Visitor = ElementVisitor<ElsaModem, Accessor>;
14    ...
15 private:
16    void setElsaData(const std::string& value);
17    ...
18 };
```

Das spezielle Visitable `ElsaModem` erbt die Implementierung für `accept(::Visitor&)` von der Basisklasse `VisitableImpl<ElsaModem>`. Die Basisklasse wird über den Typparameter `ConcreteVisitable` mit den dafür notwendigen Typinformationen ausgestattet.

`ElsaModem` definiert auf der Basis des Templates `ElementVisitor` das von einem konkreten Visitor erwartete Interface:
`using Visitor = ElementVisitor<ElsaModem, ElsaAccessor>;` In diesen Typ wird das Argument `visitor` in der `accept(..)` Methode gecastet. Der Zugriff auf diesen Typ erfolgt über den qualifizierten Namen `ConcreteVisitable::Visitor`.

In Listing 240 auf der vorherigen Seite entspricht
der Typ `ConcreteVisitable` dem Typ `ElsaModem` und
der Typ `ConcreteVisitable::Visitor` dem Typ `ElsaModem::Visitor`

wenn `VisitableImpl<..>` angewendet wird wie in Listing 241 auf der vorherigen Seite.

15.3.6 Visitable, eine erweiterte Implementierung

Die Definition des speziellen Visitor Interfaces für die konkreten Visitor Klassen `using Visitor = ElementVisitor<ElsaModem, ElsaAccessor>;` kann in die Basisklasse `VisitabelImpl<..>` verschoben werden. Dazu wird ein weiterer Typ Parameter in diesem Template benötigt, wie in Listing 242: der `Accessor`.

Listing 242: generisches Visitable mit Accessor

```
template< class ConcreteVisitable_,
   class Accessor = DefaultAccessor,
   class LoggingPolicy = DefaultLoggingPolicy
   >
class VisitableImpl : public Visitable, public LoggingPolicy{
public:
   using ConcreteVisitable = ConcreteVisitable_;

   using Visitor = ElementVisitor<ConcreteVisitable, Accessor>;

   // alternative Möglichkeit:
   // error: invalid use of incomplete type 'class ElsaModem'
   using Visitor =
      ElementVisitor<ConcreteVisitable, typename ConcreteVisitable::Accessor>;

   void accept(::Visitor& visitor){
      using Visitor = typename ConcreteVisitable::Visitor;
      ... wie vorher
   }
};
```

Die Anwendung dieses Templates ist in Listing 243 gezeigt. Die Accessorklasse muss vor der Visitable Klasse definiert werden, damit sie als template Parameter für die Basisklasse verwendet werden kann. Das bevölkert leider den Scope mit einem weiteren Klassennamen. Um der Accessorklasse weiter Zugriff auf die nicht öffentlichen Elemente von ElsaModem zu gewähren muss sie `friend class ElsaAccessor;` in ElsaModem deklariert werden.

Listing 243: Ein konkretes Visitable

```
class ElsaAccessor
{
protected:
   static void setElsaData(ElsaModem& modem, const std::string& value);
   static std::string const& getElsaData(ElsaModem& modem);
};
class ElsaModem :
   public VisitableImpl<ElsaModem, ElsaAccessor>
{
```

```
10    friend class ElsaAccessor;
11 public:
12    ElsaModem();
13    ...
14 private:
15    void setElsaData(const std::string& value);
16    ...
17 };
```

Eine alternative Möglichkeit, die Accessorklasse zur Verfügung zu stellen ist, den Namen `Accessor` als Konvention in dem Typ `ConcreteVisitable` zu vereinbaren, genauso wie der Name `Visitor` per Konvention vereinbart ist.

Der Compiler quitiert den Versuch, auf diesen eingebetteten Typ `ConcreteVisible ::Accessor` an dieser Stelle im Code in Listing 242 auf der vorherigen Seite zuzugreifen, mit der Fehlermeldung:
`error: invalid use of incomplete type 'class␣ElsaModem'`
Der Zugriff auf den Typ `ConcreteVisitable::Visitor` in der Methode `access(..)` ist jedoch erfolgreich, weil der Typ `ElsaModem` schon vollständig konstruiert ist, wenn die Methode verwendet wird!

15.3.7 Ein weiterer Name für einen Typ als Hook

Der Typ `Visitor` wird aber erst bei der Implementierung der konkreten Visitor Klassen verwendet, wie in Listing 238 auf Seite 250. Daher kann die Auflösung der Namen bis zu diesem Zeitpunkt verschoben werden, in dem sie erst im Template `ElementVisitor<..>` erfolgt.

Das Template sieht dann wie in Listing 244 aus.
Für den Parameter Typ von `visit(..)` wird der Typ des konkreten Visitables benötigt. Der Name wird ebenfalls per Konvention auf `ConcreteVisitable` festgelegt. Die Schnittstelle für das Template `ElementVisitor<..>` sind die Namen `ConcreteVisitable` und `Accessor`.

Listing 244: ElementVisitor revisited

```
1 template<class VisitableImpl>
2 class ElementVisitor : protected VisitableImpl::Accessor
3 {
4 public:
5    virtual void visit(typename VisitableImpl::ConcreteVisitable& visitable) = 0;
6 };
```

Für Visitable Klassen, die keinen Accessor benötigen, wird in der Basisklasse `VisitableImpl<..>` in Listing 245 auf der nächsten Seite eine leere Klasse `class Accessor{};`, die default Implementierung, definiert.

Der Name `Visitor` für das Interface ist Konvention und wird von den Visitable Klassen eingehalten weil sie den Typ mit dieser Implementierung erben.
Der Zugriff in `accept(..)` erfolgt weiterhin über den qualifizierten Namen

`ConcreteVisitable::Visitor`. Damit könnte eine Klasse, die von `VisitableImpl <..>` erbt, den Visitor spezifisch implementieren, der generierte Visitor aus dem Template `VisitableImpl` würde überdeckt werden, weil über den konkreten Typ darauf zugegriffen wird! Die Methode `access` würde aber weiter funktionieren.

Listing 245: generische Visitable Implementierung mit Accessor

```
template<
   class ConcreteVisitable_,
   class LoggingPolicy = DefaultLoggingPolicy
   >
class VisitableImpl : public Visitable, public LoggingPolicy{
public:
   using ConcreteVisitable = ConcreteVisitable_;
   class Accessor{};
   using Visitor = ElementVisitor<ConcreteVisitable>;

   void accept(::Visitor& visitor){
      using Visitor = typename ConcreteVisitable::Visitor;
      ... wie vorher
   }
};
```

Ein konkretes Visitable ist in Listing 246 skizziert. Die Klasse `ElsaModem::Accessor` wird anstelle der Klasse mit demselben Namen aus der Basisklasse (`VisitableImpl<ElsaModem>::Accessor`) benutzt, weil `ConcreteVisitable` ein Aliasname für `ElsaModem` ist und der Visitor damit definiert wird:
`using Visitor = ElementVisitor<ConcreteVisitable>;`.

Listing 246: Ein konkretes Visitable auf Framework Basis

```

class ElsaModem :
   public VisitableImpl<ElsaModem>
{
public:
   class Accessor{...};
   ...
};
```

Der Name `Accessor` ist ein weiterer Hook, mit dem sich die Anwendung im Visitor Pattern Framework einhängen und das Verhalten anpassen kann. Das Framework definiert angemessenes default Verhalten, eine konkrete Visitable Klasse kann es spezialisieren.

15.3.8 Anwendung und Visitor

Die Implementierung der konkreten Visitors sieht immer noch wie in Listing 238 auf Seite 250 aus und die Anwendung wie in Listing 235 auf Seite 248. Die konkreten Visistables müssen aber nur noch von der Implementierung `VisitableImpl <..>` erben und bei Bedarf einen Accessor zur Verfügung stellen.

15.4 Visitable Adapter

Sollen Objekte von Klassen besucht werden, die nicht gemäß dem Visitor Pattern gestaltet sind, wie in Listing 247, kann ein Adapter für diese Objekte erstellt werden. Diese Objekte können nur über ihre öffentliche Schnittstelle manipuliert werden, ein Accessor steht dafür meistens nicht zur Verfügung.

Listing 247: Ein einfacher non visitable Typ

```
1  class HayesModem
2  {
3  public:
4     HayesModem(){};
5
6     std::string toString() const
7     {
8        return "HayesModem";
9     }
10 };
```

15.4.1 Eine Adapter Implementierung auf der Basis des Frameworks

Das Listing 248 zeigt eine mögliche Implementierung des Adapters auf der Basis des Tempaltes `VisitableImpl`.

Listing 248: Ein Adapter auf der Basis von VisitableImpl

```
1  template<class Adaptee>
2  class VisitableAdapter :
3        public VisitableImpl<VisitableAdapter<Adaptee>>
4  {
5  public:
6     VisitableAdapter(Adaptee& element): adaptee(element){}
7
8     std::string toString() const {
9        std::string message("Adapter0::");
10       message += adaptee.toString();
11       return message;
12    }
13
14    Adaptee& getAdaptee() const { return adaptee;}
15 private:
16    Adaptee& adaptee;
17 };
```

Ein Visitor, der mit einem Adapter aus Listing 248 zusammen arbeitet, muss wie in Listing 249 auf der nächsten Seite implementiert werden. Der Parameter Typ der Methode `visit(..)` ist `VisitableAdapter<ConcreteVisitable>&` anstatt des Typs des Objekts, das bearbeitet werden soll. Das eigentliche Objekt muss der Visitor erst vom Adapter beschaffen: `modem = adapter.getAdaptee()`.

Listing 249: Ein Visitor für einen Adapter

```
class ConfigureMacModemVisitor :
   public Visitor,
   public VisitableAdapter<HayesModem>::Visitor
{
public:
   virtual void visit(VisitableAdapter<HayesModem>& adapter)
   {
      std::cout << "-> ConfigureMacModemVisitor visits " << adapter.toString()
      << std::endl;
      HayesModem& modem = adapter.getAdaptee();
      ...
   }
   // wurde ersetzt ...
   virtual void visit(HayesModem& modem){ ... }
   ...
};
```

15.4.2 Eine unabhängige Implementierung des Adapters

Soll der Methode visit ein Element der Menge (z.B. ein `HayesModem`) übergeben werden und nicht der Adapter, kann die alternative Implementierung aus Listing 250 verwendet werden.

Der Adapter implementiert das Interface `Visitable` selbst. Der gesamte Code des Templates `VisitorImpl<..>` muss wiederholt werden.

Listing 250: Eine unabhängige Implementierung des Adapters

```
template<
   class Adaptee,
   class LoggingPolicy = DefaultLoggingPolicy>
class VisitableAdapter1 : public Visitable, public LoggingPolicy{
public:
   VisitableAdapter1(Adaptee& adaptee): adaptee(adaptee){}

   // Schnittstelle für ElementVisitor
   class Accessor{};
   using ConcreteVisitable = Adaptee;
   using VisitableImplementation = VisitableAdapter1<Adaptee, LoggingPolicy>;
   // Interface Definition für die Visitors
   using Visitor = ElementVisitor<VisitableImplementation>;

   ConcreteVisitable* getVisitable() {
      return &adaptee;
   }

   void accept(::Visitor& visitor){
      using Visitor = VisitableImplementation::Visitor;
      Visitor* v = dynamic_cast<Visitor*>(&visitor); //crosscast
```

```
22       ConcreteVisitable* visitable = getVisitable();
23       if(v){
24          this->logAccepted(*visitable, visitor);
25          v->visit(*visitable);
26       } else {
27          this->logNotAccepted(*visitable, visitor);
28       }
29    }
30
31    std::string toString() const {...}
32 private:
33    Adaptee& adaptee;
34 };
```

Der Visitor für diesen Adapter ist in Listing 251 abgebildet.
Die Methode `visit(ConcreteVisitable&)` erwartet keinen Adapter sondern ein Objekt des zu besuchenden Typs. Der Visitor muss aber die Unterscheidung zwischen “echten” Visitables (`HayesModem::Visitor`) und adaptierten Elementen (`VisitableAdapter<HayesModem>::Visitor`) treffen.

Listing 251: Ein weiterer Visitor für einen Adapter

```
1  class ConfigureDOSModemVisitor :
2     public Visitor,
3     public VisitableAdapter1<HayesModem>::Visitor,
4     public VisitableAdapter1<ZoomModem>::Visitor,
5     public VisitableAdapter1<ElsaModem>::Visitor
6  {
7  public:
8     virtual void visit(HayesModem& modem)
9     {
10       std::cout << "-> ConfigureDOSModemVisitor visits " << modem.toString()
11          << std::endl;
12    }
13    ...
14    virtual std::string toString() const
15    {
16       return "ConfigureDOSModemVisitor";
17    }
18 };
```

Die Anwendung der beiden Adapter und deren Ausgabe ist in Listing 252 abgebildet. Die fehlenden Codezeilen sind dieselben wie in Listing 235 auf Seite 248.

Listing 252: Die Anwendung der Adapter

```
1  int main()
2  {
3     cout << "Visitor_AdaptedV2" << endl;
4     Visitables visitables;
5
6     HayesModem hayes;
7
```

```
8     visitables.push_back(SharedPointer(new VisitableAdapter1<HayesModem>(hayes)));
9     visitables.push_back(SharedPointer(new VisitableAdapter<HayesModem>(hayes)));
10
11    ConfigureDOSModemVisitor dosVisitor;
12    ConfigureMacModemVisitor macVisitor;
13
14    demoVisitor(dosVisitor, visitables);
15    demoVisitor(macVisitor, visitables);
16 }
17 //Ausgabe:
18 Visitor_AdaptedV2
19
20 ==== ConfigureDOSModemVisitor ====
21 HayesModem accepted ConfigureDOSModemVisitor
22 -> ConfigureDOSModemVisitor visits HayesModem
23 Adapter0::HayesModem did not accept ConfigureDOSModemVisitor
24
25 ==== ConfigureMacModemVisitor ====
26 HayesModem did not accept ConfigureMacModemVisitor
27 Adapter0::HayesModem accepted ConfigureMacModemVisitor
28 -> ConfigureMacModemVisitor visits Adapter0::HayesModem
```

15.4.3 Harmonisierung der Implementierungen

Den Versuch, die beiden Implementierungen auf abstrakte Weise gleich zu gestalten, auch wenn bestimmte Abstraktionen in der einen oder anderen Version überflüssig erscheinen, bezeichne ich als Harmonisierung. Der Ergebnis ist in Listing 253 und Listing 250 auf Seite 259 zu sehen. Der Code sieht fast identisch aus und kennzeichnet die Unterschiede.

Beim Vergleich der harmonisierten Implementierung von `VisitableImpl<..>` aus Listing 253 mit der Adapter Implementierung aus Listing 250 auf Seite 259 wird die Notwendigkeit der Unterscheidung zwischen dem Typ des Elements, das besucht wird (`visit(visitable)`) und dem Typ, der die Infrastruktur für das Visitor Pattern zur Verfügung stellt (`VisitableImplementation`), deutlich.
Zur Infrastruktur gehört das Interface `Visitor` für die Implementierung der konkreten Visitors und den cast sowie die Typdefinitionen `ConcreteVisitable` und `Accessor` die das Template `ElementVisitor<VisitableImplementation>` benötigt. Das Interface Visitor wird auf der Basis des Templates `ElementVisitor` definiert.

Listing 253: Die Implementierung von VisitableImpl<..>

```
1 template<
2    class ConcreteVisitable_,
3    class LoggingPolicy = DefaultLoggingPolicy>
4 class VisitableImpl : public Visitable, public LoggingPolicy{
5 public:
6
7    // Schnittstelle für ElementVisitor
8    class Accessor{};
```

```
9     using ConcreteVisitable = ConcreteVisitable_;
10    using VisitableImplementation = ConcreteVisitable_;
11    // Interface Definition für die Visitors
12    using Visitor = ElementVisitor<VisitableImplementation>;
13
14    ConcreteVisitable* getVisitable() {
15       return static_cast<ConcreteVisitable*>(this);
16    }
17    void accept(::Visitor& visitor){
18       using Visitor = typename VisitableImplementation::Visitor;
19       Visitor* v = dynamic_cast<Visitor*>(&visitor); //crosscast
20       ConcreteVisitable* visitable = getVisitable();
21       if(v){
22          this->logAccepted(*visitable, visitor);
23          v->visit(*visitable);
24       } else {
25          this->logNotAccepted(*visitable, visitor);
26       }
27    }
28 };
```

15.4.4 Ein template basiertes Framework für das Acyclic Visitor Pattern

Wird das Template `VisitableImpl<..>` wie in Listing 246 auf Seite 257 verwendet, ist der Typ `ConcreteVisitable` und der Typ `VisitableImplementation` identisch. Das default Argument für den Parameter
`VisitableImplementation_ = ConcreteVisitable_` in Listing 254 auf Seite 264 zeigt diesen Zusammenhang.
Das Diagramm 17 auf der nächsten Seite zeigt die UML Sicht auf diesen Sachverhalt. Die Aliasnamen (`using ..`) und die inneren Typdefinitionen (**`class ..`**) sind mit der nested Beziehung modelliert. Die Aliasnamen sind als Rollen an den nested Beziehungen angetragen, wenn der Typ mehrere Rollen annehmen kann. Auf diese Namen beziehen sich die Argumente der Templatebindings.

Im Falle des Adapters müssen die beiden Typen `ConcreteVisitable` und `VisitableImplementation` unterschieden werden. Um sie in `VisitableImpl` in Listing 254 auf Seite 264 unterscheiden zu können, wird ein weiterer Type Parameter
`VisitableImplementation_` eingeführt.

Das `ConcreteVisitable` wird mit dem `Adaptee` und `VisitableImplementation` mit dem Adapter belegt.

Das Diagramm 18 auf der nächsten Seite zeigt die UML Sicht auf diesen Sachverhalt.
Die default Implementierungen, die leere Klasse `Accessor` und die Methode `getVisitable ():ConcreteVisitable`, werden durch das Template `VisitableImpl<..>` des Frameworks zur Verfügung gestellt.
Das ElsaModem in Diagramm 17 auf der nächsten Seite überschreibt diesen Typ

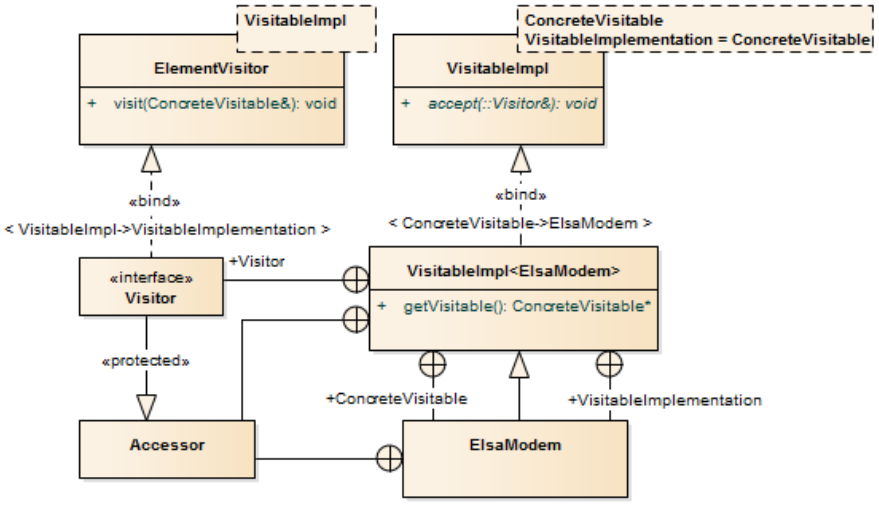

Diagramm 17: Nested Types und Template Parameter

durch einen eigenen Accessor.
Der Adapter überschreibt die Methode `getVisitable()`. Das Listing 254 auf der

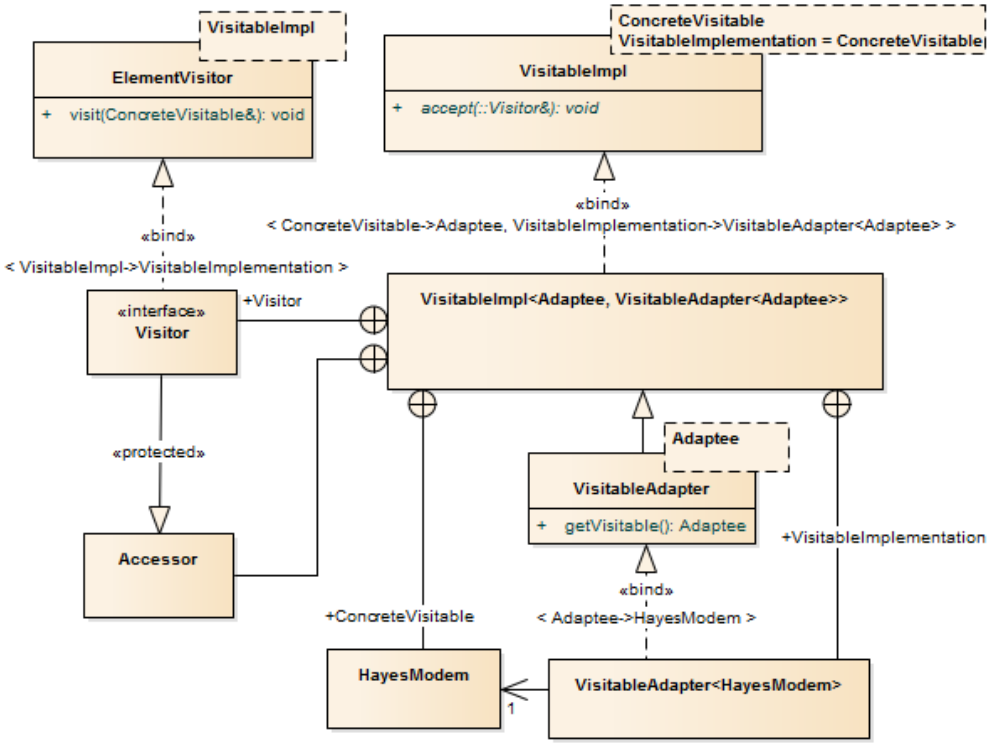

Diagramm 18: Nested Types und der Adapter

nächsten Seite zeigt eine mögliche Implementierung die diese Unterscheidung trifft.

Listing 254: Die Zusammenführung der Implementierungen

```
template<
   class ConcreteVisitable_,
   class VisitableImplementation_ = ConcreteVisitable_,
   class LoggingPolicy = DefaultLoggingPolicy
   >
class VisitableImpl : public Visitable, public LoggingPolicy{
public:
   // default Accessor für ElementVisitor,
   // kann von einem konkreten Visitable überschrieben werden
   class Accessor{};

   // Parametertyp für visit in ElementVisitor
   using ConcreteVisitable = ConcreteVisitable_;

   // Typ der das Interface Visitable implementiert!
   // stellt Infrastruktur
   // für ElementVisitor und für VisitableImpl<..> zur Verfügung:
   // Accessor, ConcreteVisitable, Visitor, VisitableImplementation, getVisitable()
   using VisitableImplementation = VisitableImplementation_;

   // default Visitor, Interface Definition für die konkreten Visitors
   // kann von einem konkreten Visitable überschrieben werden
   using Visitor = ElementVisitor<VisitableImplementation>;

   // Muss überschrieben werden wenn ConcreteVisitable und
   // VisitableImplementation nicht übereinstimmen
   ConcreteVisitable* getVisitable() {
      return static_cast<ConcreteVisitable*>(this);
   }
   ConcreteVisitable const* getVisitable() const {
      return static_cast<ConcreteVisitable*>(this);
   }

   // liefert this als Pointer auf die Spezialisierung
   VisitableImplementation* This(){
      return static_cast<VisitableImplementation*>(this);
   }

   void accept(::Visitor& visitor){
      using Visitor = typename VisitableImplementation::Visitor;
      Visitor* v = dynamic_cast<Visitor*>(&visitor); //crosscast
      ConcreteVisitable* visitable = This()->getVisitable();

      if(v){
         this->logAccepted(*visitable, visitor);
         v->visit(*visitable);
      } else {
         this->logNotAccepted(*visitable, visitor);
      }
   }
```

```
51 };
```

Ein weiterer Hook
In der Methode `accept(..)` wird über die Methode `getVisibable()` das zu besuchende Objekt ermittelt. Der cast `static_cast<VisitableImplementation*>(this)` und die Verwendung beim Aufruf von `This()->getVisible()` ist notwendig, um die richtige Methode aus der Spezialisierung von `VisitableImpl<..>` auszuwählen.
An dieser Stelle ersetzt wieder Namensüberdeckung das Schlüsselwort `virtual`, weil der konkrete Typ in der generischen Programmierung immer bekannt ist! Das ist statische Polymorhie! Die Methode `accept(..)` ist die Template Method und die Methode `getVisitable()` ist die Hook Method gemäß dem Template Method Pattern[278].
Die Methoden `logAccepted(..)` und `logNotAccepted(..)` sind keine Hook Methoden gemäß dem Template Method Pattern, da sie über `this->` und nicht über `This()->` aufgerufen werden. Sie können aber gemäß dem OCP variiert werden, indem die DefaultLoggingPolicy ausgetauscht wird.

15.4.5 Eine Adapterimplementierung

Der Adapter kann damit wie in Listing 255 implementiert werden.

Listing 255: Der Adapter für das Acyclic Visitor Pattern

```
1  template<class Adaptee>
2  class VisitableAdapter :
3        public VisitableImpl<Adaptee, VisitableAdapter<Adaptee>>
4  {
5  public:
6     VisitableAdapter(Adaptee& adaptee): adaptee(adaptee){}
7     Adaptee* getVisitable() { return &adaptee; }
8  private:
9     Adaptee& adaptee;
10 };
```

Mit einem Adapter wie in Listing 255 gezeigt, ist eine direkte Implementierung des Interfaces `Visitable` nur noch für Typen wie z.B. `ElsaModem` notwendig, deren öffentliche Schnittstelle nicht ausreichend für die Bearbeitung durch die Visitors ist. Jeder andere Typ kann über den generischen Adapter besuchbar (`Visitable`) gemacht werden.

15.4.6 Die Adapter StoragePolicy

Der Adapter wie er in Listing 255 abgebildet ist, verwaltet die Adaptees via Reference auf die Objekte `Adaptee&`. Damit bürdet der Adapter die Verwaltung der Lebensdauer dem Client auf. Eleganter wäre die Auswahlmöglichkeit einer entsprechenden `StoragePolicy`, mit der die Clients bestimmen können, wie die Adaptees verwaltet werden, z.B. via `std::weak_ptr<Adaptee>`.

[278] section 14.1 auf Seite 234

Ein erster Versuch ist in Listing 256 abgeblidet. Der Adapter hat einen weiteren Parameter `StoragePolicy` von dem er erbt. Der Member `adaptee` ist in der Policy definiert und wird im Konstruktor initialisiert. Mit `get()` wird in `getVisitable()` darauf zugegriffen.

Listing 256: Ein Adapter mit StoragePolicy

```
template<
      class Adaptee,
      class StoragePolicy = StorageByReference<Adaptee>>
class VisitableAdapter :
      public VisitableImpl<Adaptee, VisitableAdapter<Adaptee, StoragePolicy>>,
      public StoragePolicy
{
public:
   using StorageType = typename StoragePolicy::StorageType;
   using ReturnType = typename StoragePolicy::ReturnType;

   using Visitor = typename Adapter<Adaptee>::Visitor;

   VisitableAdapter(StorageType element): StoragePolicy(element){}

   ReturnType getVisitable() { return this->get(); }

   std::string toString() const {...}
};
```

Zwei möglich StoragePolicies aus dem Header `StoragePolicies.h` sind in Listing 257 abgebildet. Mit der Policy `StorageByReference` verhält der Adapter sich wie vorher.
Die Policy `StorageByWeakPointer` speichert das Adaptee in einem Weakpointer und liefert in der Methode `get()` einen `std::shared_ptr<Adaptee>` zurück, der über `adaptee.lock()` vom Weakpointer angefordert wird. Damit ist gewährleistet, dass das Objekt während der Bearbeitung durch den Visitor existiert, wenn `lock()` einen gültigen Sharedpointer liefert. In der Methode `VisitableImpl<..>::access(..)` kann der Pointer auf das `visitable` auf Gültigkeit geprüft werden: `if(!visitable)return;`. Das ist auch mit einem nativen Pointer möglich, aber ob das Objekt noch existiert, ist bei einem nativen Pointer nicht festzustellen (dangling Pointer).

Listing 257: Zwei mögliche StoragePolicies

```
template<class Adaptee>
struct StorageByReference{
   using StorageType = Adaptee&;
   using ReturnType = Adaptee*;

   StorageByReference(StorageType adaptee):adaptee(adaptee){}

   ReturnType get(){ return &adaptee; }
protected:
   StorageType adaptee;
};
template<class Adaptee>
```

```
13 struct StorageByWeakPointer{
14    using StorageType = std::weak_ptr<Adaptee>;
15    using ReturnType = std::shared_ptr<Adaptee>;
16 
17    StorageByWeakPointer(StorageType adaptee):adaptee(adaptee){}
18    ReturnType get(){ return adaptee.lock(); }
19 protected:
20    StorageType adaptee;
21 };
```

Leider funktionieren mit dem Adapter aus Listing 256 auf der vorherigen Seite die Visitors nicht mehr, weil zwei Adapter mit unterschiedlicher StoragePolicy verschiedene Typen sind und damit ist deren nestet Type `Visitor` ebenfalls verschieden, auch wenn der Typ `Accessor` und der Typ des Parameters für `visit(ConcreteVisitable)` identisch sind. Der `dynamic_cast` in `access(..)` schlägt fehl und der Visitor wird nicht benachrichtigt, bzw. das aktuelle `Visitable` nicht besucht.
In Listing 258 erbt der Vsitor bei 1 von einem Adapter mit der default StoragePolicy und bei 2 wird die StoragePolicy explizit angegeben. Beide Fälle sind nicht sinnvoll verwendbar, weil der Visitor nur Elemente die mit genau diesem Adapter gekappselt sind, verarbeitet. An dieser Stelle darf aber die vom Benutzer verwendete Policy nicht bekannt sein müssen. Sonst müsste in der Liste der Basisklassen jede Kombination von Adpapter und StoragePolicy für jeden Typ aufgelistet werden und das ist nicht praktikabel und steht im Widerspruch zum OCP.

Listing 258: Visitor erbt von falschem Typ

```
1  class DemoVisitor : public Visitor,
2     // 1
3     public VisitableAdapter<NonVisitable>::Visitor,
4     // 2
5     public VisitableAdapter<NonVisitable, StorageByWeakPointer<NonVisitable>>::
          Visitor
6  {
7  public:
8  
9     virtual void visit(NonVisitable& nv)
10    {
11       std::cout << "-> " << toString() << " visits " << nv.toString() << std::endl;
12    }
13 
14    virtual std::string toString() const {...}
15 };
```

Die fehlenden Codefragmente in Listing 259 sind dieselben wie in Listing 235 auf Seite 248.

Listing 259: Eine Anwendung mit Adapter mit StoragePolicies

```
1 ...
2 template<class Adaptee>
3 using AdapterWeak = VisitableAdapter<Adaptee, StorageByWeakPointer<Adaptee>>;
4 template<class Adaptee>
```

```
using AdapterReference = VisitableAdapter<Adaptee, StorageByReference<Adaptee>>;

int main()
{
   cout << "Visitor TemplatedAcyclicAdvanced" << endl;
   Visitables visitables;

   NonVisitable nv;
   shared_ptr<NonVisitable> p(new NonVisitable);
   visitables.push_back(SharedPointer(new AdapterWeak<NonVisitable>(p)));
   visitables.push_back(SharedPointer(new AdapterReference<NonVisitable>(nv)));

   DemoVisitor v;
   demoVisitor(v, visitables);
```

15.4.7 Eine Adapter neutrale ElementVisitor Definition

Was benötigt wird, ist eine Implementierungs neutrale `ElementVisitor<..>` Definition, bei der nur der Typ, der besucht werden soll, bekannt sein muss und nicht, wie der Besuch realisiert wird.

Listing 260: Eine Skizze eines Visitors

```
class ConfigureDemoVisitor : public Visitor,
   public implementsVisitor<ElsaModem>,
   public implementsVisitor<NonVisitable>,
   public implementsVisitor<NonVisitableWithAccessor>
{
public:
   virtual void visit(ElsaModem& modem)
   {
      getVisitor<ElsaModem>* This = this;

      std::cout << "-> " << toString() << " visits " << modem.toString()
      << std::endl;
      This->setData(modem, "DemoVisitor");
      std::cout << "Data: " << This->getData(modem) << std::endl;
   }
   virtual void visit(NonVisitable& nv)
   {
      std::cout << "-> " << toString() << " visits " << nv.toString()
      << std::endl;
   }
   virtual void visit(NonVisitableWithAccessor& visited)
   {
      getVisitor<NonVisitableWithAccessor>* This = this;

      std::cout << "-> " << toString() << "
      visits " << visited.toString() << std::endl;
      std::cout << "Data: " << This->getData(visited) << std::endl;
```

```
28      This->setData(visited, "DemoConfiguration");
29      std::cout << "Data: " << This->getData(visited) << std::endl;
30   }
31   virtual std::string toString() const {return "ConfigureDemoVisitor";}
32 };
```

Das Listing 260 auf der vorherigen Seite skizziert einen Visitor, der unabhängig davon ist, ob der Typ der besucht wird, das Interface Visitable selbst implementiert oder durch einen Adapter gekappselt ist. Wird z.B. die Klasse `NonVisitable` geändert und auf der Basis des Frameworks als Visitable implementiert, bleibt der Code aller Visitors davon unberührt.

Die *type-function* `getVisitor<Visitable>` erzeugt einen Typ Visitor der für das übergebene Visitable passend ist und liefert diesen zurück!

Der Zugriff auf die nicht öffentlichen Elemente z.B. von ElsaModem, wird wie bisher über eine Accessorklasse realisiert.

Treten dabei Namenskonflikte auf, weil die Accessors der besuchten Typen dieselben Signaturen oder dieselben Namen verwenden, kann der qualifizierte Name verwendet werden: `This->getData(modem);`
mit dem *upcast* `getVisitor<ElsaModem> * This = this;`.

Die Qualifizierung ist auch notwendig, wenn nur der gleiche Name, z.B. `setData(..)` aber unterschiedliche Signaturen verwendet werden, weil Operationen mit gleichem Namen in verschiedenen Basisklassen überdeckt und bei der Overload Resolution nur berücksichtigt werden, wenn sie mit einer entsprechenden `using Baseclassname::operationName` Klausel in den Namensraum der abgeleiteten Klasse eingeführt werden. Die Operationen der Accessors werden aber jeweils nur in den Methoden der dazugehörigen Typen benötigt. `This->` ist eine einfache lokale Lösung für das Problem.

Listing 261: ElementVisitor Adapterneutral

```
1  template<class Adaptee>
2  struct Adapter{
3     // Interface für ElementVisitor
4     using ConcreteVisitable = Adaptee;
5     class Accessor{};
6
7     using this_type = Adapter<Adaptee>;
8     //Visitor erzeugen
9     using Visitor = ElementVisitor<this_type>;
10 };
```

Listing 262: VisitorAdapter Adapterneutral

```
1  template<
2        class Adaptee,
3        class StoragePolicy = StorageByReference<Adaptee>>
4  class VisitableAdapter :
5        public VisitableImpl<Adaptee, VisitableAdapter<Adaptee, StoragePolicy>>,
6        public StoragePolicy
```

```
{
public:
   ...
   using Visitor = getVisitor<Adaptee>;
   ...
};
```

Das Template `Adapter<Adaptee>` in Listing 261 auf der vorherigen Seite ist von dem eigentlichen `VisitableAdapter` in Listing 262 auf der vorherigen Seite unabhängig, die StoragePolicy hat keinen Einfluss auf den Typ von `Visitor`.

Der eigentliche `VisitableAdapter` stellt lediglich die Infrastruktur zur Verfügung und definiert das Interface `Visitor` auf derselben Basis wie die konkreten Visitors in Listing 260 auf Seite 268 mit `getVisitor<Adaptee>`. Er überschreibt damit den Namen Visitor aus der Basisklasse `VisitorImpl<..>` und dadurch wird dieser Typ in `VisitorImpl<..>::accept(..)` verwendet: `VisitableImplementation::Visitor`.

15.5 Das Visitor Interface für Visitables erzeugen

15.5.1 Eine einfache type-function

Die *type-function* `getVisitorType<Visitable>::type` und der dazugehörige Alias `getVisitor<Visitable>` sind im Header `VisitorImplementation.h` definiert.

Ein Ausschnitt daraus ist in Listing 263 abgebildet.

Alle Templates `getVisitorType` zusammen bilden die *type-function*.

Listing 263: type-function getVisitor erster Versuch

```
template <typename T> struct void_type { typedef void type; };

template<class ToVisit, class = void>
struct getVisitorType{
   using type = typename VisitorInterface<ToVisit>::Visitor;
};
template<class ToVisit>
struct getVisitorType<ToVisit,
                      typename void_type<typename ToVisit::Visitor>::type>
{
   using type = typename ToVisit::Visitor;
};
template<typename ToVisit>
using getVisitor = typename getVisitorType<ToVisit>::type;
template<typename ToVisit>
using implementsVisitor = typename getVisitorType<ToVisit>::type;
```

Das primäre Template `getVisitorType<..>` ist mit 2 Type Paramtern ausgestattet. Der zweite Parameter mit default Argument (`class = void`) hat keinen Namen weil er nur zur Auswahl der Spezialisierung benötigt wird. Die Spezialisierung von

`getVisitorType<..>` ist für `void_type<ToVisit::Visitor>::type` spezialisiert und das evaluiert immer zu **`void`**.

Das Template `void_type<typename ToVisit::Visitor>::type` kann aber nur angewendet werden, wenn der Typ `ToVisit` einen nestet Type `Visitor` hat und nur dann wird auch die Spezialisierung erzeugt.

Die Unmöglichkeit der Ersetzung eines Parameters einer Spezialisierung durch ein Argument führt nicht zu einem Compilerfehler. Dieses Prinzip wird abgekürzt: **SFINAE**[279], Substitution Failure is not an Error, genannt.

Beim Aufruf von `getVisitorType<..>` mit nur einem Argument, wird das default Argument **`void`** verwendet und das führt zur Auswahl der Spezialisierung mit **`void`** als zweitem Argument, wenn sie vorhanden ist. Sie erzeugt den Typ `ToVisit::Visitor`, der spezifisch für `ToVisit` ist.
Existiert `ToVisit::Visitor` nicht, wird die Spezialisierung nicht erzeugt und das Primary Template angewendet, das den Typ `VisitorInterface<ToVisit>::Visitor` erzeugt.
Für die bequeme Anwendung sind zwei Aliasee definiert: `getVisitor<..>` und `implementsVisitor<..>`

15.5.2 Erweiterte type-functions

Anstatt nur zu prüfen, ob der Typ `ToVisit` einen nested type `Visitor` hat, könnte in einem nächsten Schritt, wenn er nicht vorhanden ist, geprüft werden, ob der Typ `Accessor` vorhanden ist und damit den Visitor ausstatten, der mit dem VisitorInterface erzeugt wird. Wenn auch kein Accessor vorhanden ist, wird ein Visitor mit dem `EmptyAccessor` erzeugt.

Das VisitorInterface in Listing 264 ist dafür mit einem weiteren Type Parameter Accessor ausgestattet mit dem default Argument EmptyAccessor. Er wird im Header `VisitorImplementation.h` definiert.

Listing 264: Visitor mit Accessor erzeugen

```
1  struct EmptyAccessor{
2     std::string toString(){ return "EmptyAccessor";}
3  };
4  template<class ToVisit, class Accessor_ = EmptyAccessor>
5  class VisitorInterface{
6  public:
7     // Interface für ElementVisitor
8     using ConcreteVisitable = ToVisit;
9     using Accessor = Accessor_;
10
11    using this_type = VisitorInterface<ToVisit, Accessor>;
12    //Visitor erzeugen
13    using Visitor = ElementVisitor<this_type>;
14 };
```

[279] SFINAE (Substitution Failure Is Not An Error; always hilariously pronounced, sometimes ßss fee nay")

Mit den spezifischen Type-Traits `hasVisitor` und `hasAccessor` aus Listing 265 kann festgestellt werden, ob ein Typ diesen nestet Type hat (`hasVisitor<NonVisitable>::value`) und wenn ja, welcher Typ es ist (`hasVisitor<NonVisitable>::type`). Das Prinzip ist dasselbe wie aus Listing 263 auf Seite 270.

Listing 265: Type-Traits hasMember

```
template<class T, class = void>
struct hasVisitor{
   enum {value = false};
   using type = void;
};
template<class T>
struct hasVisitor<
      T,
      typename void_type<typename T::Visitor>::type>{
   enum {value = true};
   using type = typename T::Visitor;
};
template<class T, class = void>
struct hasAccessor{
   enum {value = false};
   using type = void;
};
template<class T>
struct hasAccessor<T, typename void_type<typename T::Accessor>::type>{
   enum {value = true};
   using type = typename T::Accessor;
};
```

In Listing 266 ist der benötigte Code für `getVisitorType` skizziert.
Der Aufruf `getVisitorType<ConcreteVisitable>::type` bzw. der Alias `getVisitor<ConcreteVisitable>` sollte zu einem Visitor evaluieren, der zu dem jeweiligen Argument passt. Dafür sind die einzelnen Spezialisierungen von `getVisitorType<..,hasVisitor<ToVisit>::type>` bzw. `..<..,hasAccessor<..>::type>` zuständig.

Die Frage ist, welchen Typ sollte das default Argument `class = ???` des Primary Templates haben?

Listing 266: type-function getVisitor, eine Skizze

```
// Die Implementierung der type-function getVisitor<..>
//Primary Template
template<class ToVisit, class = ???
struct getVisitorType{
   // kein Visitor und Accessor in ToVisit verfügbar
   using type = typename VisitorInterface<ToVisit>::type;
};
//Spezialisierung für den Typ ToVisit::Visitor
template<class ToVisit>
struct getVisitorType<ToVisit, typename hasVisitor<ToVisit>::type>{
   // ToVisit definiert seinen Visitor
   using type = typename hasVisitor<ToVisit>::type;
```

```
13 };
14 //Spezialisierung für den Typ ToVisit::Accessor
15 template<class ToVisit>
16 struct getVisitorType<ToVisit, typename hasAccessor<ToVisit>::type>{
17    // ToVisit definiert nur einen Accessor
18    using type = typename VisitorInterface<ToVisit, typename ToVisit::Accessor>::
          type; };
19 // Der Alias, das Interface der type-function
20 template<typename ToVisit>
21 using getVisitor = typename getVisitorType<ToVisit>::type;
22 template<typename ToVisit>
23 using implementsVisitor = typename getVisitorType<ToVisit>::type;
```

Die Antwort: Jeweils den, des nestet types, der für das Argument von ToVisit definiert ist oder EmptyAccessor! Wobei der Typ Visitor Vorrang vor Accessor hat. Je nach dem, mit welchem Argument die *type-function* aufgerufen wird, muss das default Argument ein anderes sein.

Mit der *type-function* aus section 13.4.2 auf Seite 219 in Zusammenarbeit mit den Type Traits `has<Membername>`, kann der Typ des default Arguments berechnet werden.

Die Anwendung zur Berechnung des default Arguments ist in Listing 267 gezeigt. Das erste `IF` hat als Condition `hasVisitor<ToVisit>::value`. Wenn das zu true evaluiert, wird T11 zum Ergebnistyp ansonsten T12. T12 ist das Ergebnis des zweiten `IF`. Wenn die Condition `hasAccessor<ToVisit>::value` true liefert, wird T12 zu T21 ansonsten zu `EmptyAccessor`. Die `>` spitze Klammer am Ende gehört zum ersten `IF`.

Listing 267: Ein Beispiel mit IF

```
1    using type =
2       IF<hasVisitor<ToVisit>::value, // Condition
3          // T11
4          hasVisitor<ToVisit>::type,
5       // T12
6       IF<hasAccessor<ToVisit>::value, // Condition
7          // T21
8          hasAccessor<ToVisit>::type,
9          // T22
10         EmptyAccessor> // T12
11      >;
```

Damit sieht das primary Template `getVisitorType<..>` wie in Listing 268 auf der nächsten Seite aus. Es wird angewendet, wenn weder ein Visitor noch ein Accessor definiert ist. Ansonsten werden die entsprechenden Spezialisierungen aus Listing 266 auf der vorherigen Seite für Visitor oder Accessor, im ersten Schritt erzeugt und dann verwendet. Wenn es sowohl einen Visitor als auch einen Accessor gibt, wird der Visitor zum default Argument und damit wird die Spezialisierung für Visitor ausgewählt, wenn die *type-function* mit nur einem Argument aufgerufen wird: `getVisitor<HayesModem>`.

Der Aufruf der *type-function* sieht aus wie ein Funktionsaufruf! Die runden Klammern werden durch die spitzen ersetzt.
Das Interface der Funktion wird durch den Alias
`template<..> using getVisitor = ..::type;` beschrieben. Der Funktonsaufruf ist ein Ausdruck der den Typ des jeweiligen Visitors repräsentiert!
Die Funktion wird gemeinsam durch die Spezialisierungen des Klassentemplates `getVisitorType` implementiert.
Jeder Typ z.B. `HayesModem` kann durch das Framework um die weitere Eigenschaft `Visitalbe` *non-invasiv* erweitert werden.

Listing 268: Die Berechnung des default Arguments

```
1  //Primary Template
2  template<class ToVisit, class =
3                          IF<hasVisitor<ToVisit>::value,
4                             typename hasVisitor<ToVisit>::type,
5                          IF<hasAccessor<ToVisit>::value,
6                             typename hasAccessor<ToVisit>::type,
7                             EmptyAccessor>
8                          >
9        >
10 struct getVisitorType{
11    // kein Visitor und Accessor in ToVisit verfügbar
12    using type = typename VisitorInterface<ToVisit>::type;
13 };
```

15.5.3 Meta Programmierung

Abhängig vom Typ `ToVisit` wird jeweils ein Satz von vollständig verschiedenen Templates erzeugt und diese dann angewendet. Man könnte hier von einer MetaMetaProgrammierung sprechen, da zuerst der Code für die MetaProgrammierung erzeugt wird und dann der Code der übersetzt wird.

Die Konsequenzen die sich daraus ergeben sind:

1. Objekte von jedem Typ können in einer Menge als Visitables verwaltet werden.
2. Ein Typ der als Visitable konzipiert ist, kann durch einfache Vererbung als Visitable definiert werden.
3. Über die Definition eines nestet Types Accessor kann in beiden Fällen einem Visitor Zugriff auf nicht öffentliche Elemente gewährt werden.

In Listing 269 sind die verschiedenen Visitables abgebildet.

Listing 269: Verschiedene Visitables

```
1  struct NonVisitable{
2     std::string toString() const { return "NonVisitable"; }
3  };
4
5  struct NonVisitableWithAccessor{
```

```
6     using this_type = NonVisitableWithAccessor;
7
8     NonVisitableWithAccessor():data("default"){}
9
10    std::string toString() const {
11       return "NonVisitableWithAccessor";
12    }
13    class Accessor{
14    protected:
15       static void setData(this_type& This, std::string data){
16          This.data = data;
17       }
18       static std::string getData(this_type& This){
19          return This.data;
20       }
21    };
22 private:
23    std::string data;
24 };
25
26 class ElsaModem :
27    public VisitableImpl<ElsaModem>
28 {
29 public:
30    class Accessor
31    {
32    protected:
33       static void setData(ElsaModem& modem, const std::string& value);
34       static std::string const& getData(ElsaModem& modem);
35    };
36    std::string toString() const { return "ElsaModem"; }
37
38    std::string const& getModemData();
39 private:
40    void setElsaData(const std::string& value);
41    ...
42 };
```

Der Visitor sieht immer noch aus wie in Listing 260 auf Seite 268.

15.6 Die Logging Policy

Mit Policies oder auch Strategies soll das Verhalten von Objekten variiert werden können[280]. Dazu wählen die Anwender eines Templates einen Typ aus, der die jeweilige Policy in der gewünschten Weise implementiert.
Im Falle des Visitor Pattern Frameworks, kann die LoggingPolicy aber nicht von den Anwendern des Templates `VisitableImpl<..>` ausgewählt werden, weil damit die Policy festgelegt wäre. Das Ziel der LoggingPolicy in diesem Fall ist, zwischen

[280] section 13.2 auf Seite 208

verschiedenen Policies umschalten zu können um bei Erweiterungen des Systems um weitere ConcreteVisitables, fehlende Schnittstellenimplementierungen in den konkreten Visitoren erkennen und beheben zu können und im Release den effektivsten Code zu produzieren. Die LoggingPolicy könnte z.B. alle nicht akzeptierten Visitors in eine Datei schreiben.

In Ermangelung einer besseren Idee zur Implementierung, habe ich auf ein Makro `LOGGING_POLICY` zurückgegriffen. Die Entscheidung, ein Makro in der heutigen Zeit zu verwenden, fällt mir äußerst schwer. Wenn aber Makros verwendet werden, sollten sie so einfach wie nur möglich sein[281].

Wenn das Makro nicht definiert ist, wird die Deklaration
`using DefaultLoggingPolicy = EmptyLoggingPolicy` verwendet, ansonsten
`using DefaultLoggingPolicy = LOGGING_POLICY;`.
Der Header `DefaultLoggingPolicy.h` ist in Listing 270 abgebildet.

In den Beispielen in diesen Kapiteln wurde die globale Makrodefinition `LOGGING_POLICY=DemoLoggingPolicy` verwendet.

Die Methoden in den konkreten Logging Policies sind inline, leere Methoden werden vom Compiler vollständig eliminiert. Die Methode `accept(visitor)` wird damit reduziert auf den teueren Cast des Arguments in den spezifischen Visitor
`v = dynamic_cast<Visitor>(visitor)`
und den Aufruf der virtuellen Operation `Visitor::visit(..)`
`if(v) v->visit(visitable);`

Listing 270: Der Header DefaultLoggingPolicy.h

```
1  struct EmptyLoggingPolicy{
2     template<class Visitable, class Visitor>
3     static void logNotAccepted(Visitable const& visitable, Visitor const& visitor){}
4     template<class Visitable, class Visitor>
5     static void logAccepted(Visitable const& visitable, Visitor const& visitor){}
6  };
7  struct DemoLoggingPolicy{
8     template<class Visitable, class Visitor>
9     static void logNotAccepted(Visitable const& visitable, Visitor const& visitor){
10       std::string message = visitable.toString();
11       message += " did not accept ";
12       message += visitor.toString();
13       std::cout << message << std::endl;
14 //    std::clog << message << std::endl;
15    }
16    template<class Visitable, class Visitor>
17    static void logAccepted(Visitable const& visitable, Visitor const& visitor){
18       std::string message = visitable.toString();
19       message += " accepted ";
20       message += visitor.toString();
21       std::cout << message << std::endl;
22    }
23 };
```

[281] [Ale09] S. 251 ...efficiency is reason enough for relying on macros from time to time ...

```
24
25 #ifndef LOGGING_POLICY
26 using DefaultLoggingPolicy = EmptyLoggingPolicy;
27 #else
28 using DefaultLoggingPolicy = LOGGING_POLICY;
29 #endif
```

15.7 Visitor Summary

Mit den Interfaces `Visitor, Visitable, ElementVisitor<..>` aus Listing 271, den Implementierungen `VisitableImpl<..>` aus Listing 254 auf Seite 264 und `VisitableAdapter<..>` aus Listing 272 auf der nächsten Seite und der *type-function* `getVisitor<ConcreteVisitable>` steht ein robustes Framework zur Implementierung des Visitor Patterns in C++ zur Verfügung. Eine UML Übersicht ist in Diagramm 19 auf Seite 279 dargestellt. Die Elemente des Frameworks sind rötlich eingefärbt, die Klassen, die durch die Benutzer erstellt werden müssen, grün und die Elemente, die durch die Benutzer generiert werden müssen, blau. Der Rest wird durch das Framework erstellt. Werden die Accessors nicht durch den Benutzer definiert, stellt das Framework einen leeren default Accessor zur Verfügung.

Das Framework ist gemäß dem Template Method Pattern implementiert, wobei die Methode `accept(..)` die Template Methode ist, bzw. das Klassen Template `VisitableImpl<..>` die Infrastruktur aus Typen, Datenstrukturen und Methoden zur Verfügung stellt, wie die `AbstractClass` aus dem Template Method Pattern in section 14.1 auf Seite 234.

Der Weg zu diesem Template Framework, der hier gegangen wurde, ist typisch für die Entwicklung von Frameworks im Allgemeinen und Templates im Besonderen. Ausgehend von konkreten Einzelfällen, den einzelnen Visitables und den Visitors, wurden nach und nach die passenden Abstraktionen identifiziert und mit geeigneten sprachlichen Mitteln realisiert.

Die dafür notwendigen Irrwege wurden hier nicht alle aufgezeigt, der tatsächliche Weg war etwas länger, als der hier skizzierte;-).

Listing 271: Interfaces für das Visitor Framework

```
1  class Visitor
2  {
3  public:
4     virtual ~Visitor();
5     virtual std::string toString() const = 0;
6  };
7  class Visitable
8  {
9  public:
10     virtual ~Visitable() = 0;
11    virtual void accept(Visitor& visitor) = 0;
12 };
13 template<class VisitableImpl>
```

```
14 class ElementVisitor : public VisitableImpl::Accessor
15 {
16 public:
17    virtual void visit(typename VisitableImpl::ConcreteVisitable& visitable) = 0;
18 };
```

Listing 272: Der Visitable Adapter

```
1  template<
2        class Adaptee,
3        class StoragePolicy = StorageByReference<Adaptee>>
4  class VisitableAdapter :
5        public VisitableImpl<Adaptee, VisitableAdapter<Adaptee, StoragePolicy>>,
6        public StoragePolicy
7  {
8  public:
9     using StorageType = typename StoragePolicy::StorageType;
10    using ReturnType = typename StoragePolicy::ReturnType;
11
12    using Visitor = getVisitor<Adaptee>;
13
14    VisitableAdapter1(StorageType element): StoragePolicy(element){}
15
16    ReturnType getVisitable() { return this->get(); }
17 };
```

Die Hooks in das Framework sind:

- `Visitor` und `getVisitor<..>` für konkrete Visitoren die Objekte besuchen wollen
- `VisitableAdapter<Adaptee>` für non Visitable Typen
- `getVisitable():ConcreteVisitable` wenn `VisitableImplementation` vom Parametertyp von `visit(ConcreteVisitable&)` abweicht, z.B. beim Adapter
- `Accessor` wenn den konkreten Visitors von den konkreten Visitables spezifischer Zugriff auf nicht öffentliche Elemente gewährt werden muss
- `acccept(..)` wenn diese Methode spezifisch implementiert werden muss. Zum Beispiel, wenn Pre- und Post- Conditions geprüft werden sollen. Das könnte aber auch als Erweiterung des Frameworks realisiert werden.

Übung: Erweitern Sie das Framework so, dass bei Bedarf Pre- und Post- Conditions überprüft werden können, wenn das nicht notwendig ist, sollte das Compilat nicht davon belastet sein.

Das Diagramm 19 auf der nächsten Seite zeigt eine mögliche UML Repräsentation des Visitor Pattern Frameworks. Die linke und rechte Seite entsprechen sich jeweils. Ein Beispiel für den unscheinbaren `ConcreteVisitor` in der Mitte ist in Listing 260 auf Seite 268.

Auf der linken Seite der Typ `HayesModem`, der einen eigenen `Accessor` definiert und durch einen Adapter `Visitable` gemacht wird. Der `VisitableAdapter` überschreibt

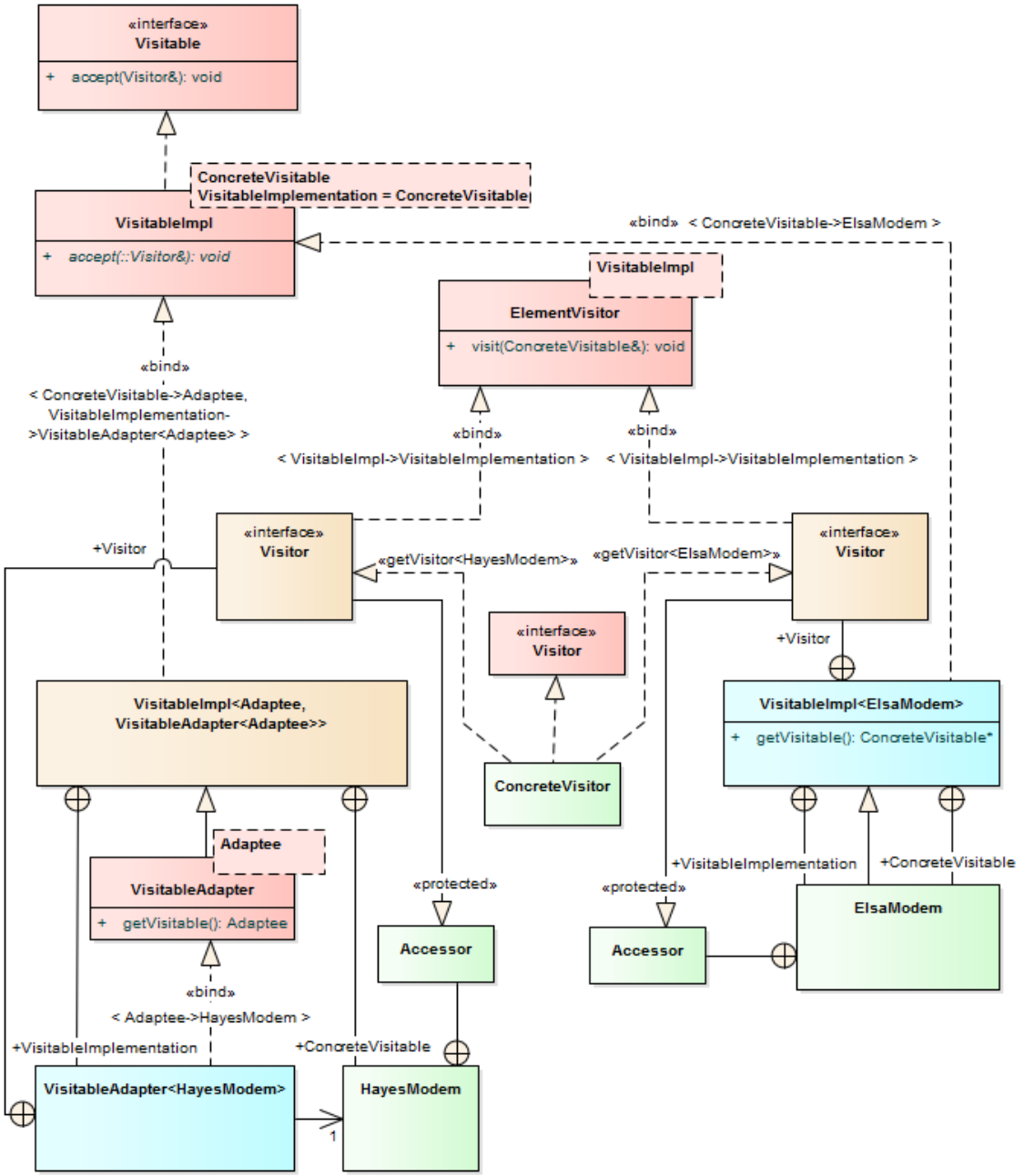

Diagramm 19: Die UML Repräsentation des Acyclic Visitor Pattern Frameworks

den default `Visitor` des Frameworks durch einen mit `getVisitor<Adaptee>` erzeugten. Genau diesen verwendet auch der `ConcreteVisitor`, sonst wäre der cast in `VisitableImpl<..>::accept(..)` nicht erfolgreich.

Auf der rechten Seite der Typ `ElsaModem`, der von `VisitableImpl<..>` erbt und ebenfalls einen eigenen Accessor definiert.

16 Weitere Pattern

16.1 Observer

16.1.1 Name, Kategorie, Synonyme

Name: Observer Pattern

Kategorie: Object Behavioral

Synonyme: Dependents, Publish-Subscribe (Peter Coad), Listener (Java)

16.1.2 Problembeschreibung

Intention Es soll eine 1:m Beziehung zwischen einem Objekt und anderen Objekten etabliert werden, um die Änderungen an einem Objekt an die anderen zu publizieren, ohne dass das eine Objekt von den anderen abhängig ist.

Motivation Wird Verantwortlichkeit geteilt, besteht die Notwendigkeit, das System über alle beteiligten Objekte hinweg, in einem konsistenten Zustand zu halten. Das Design sollte nicht zu einer gegenseitigen Abhängigkeit führen, da sonst Wiederverwendbarkeit und Erweiterbarkeit nicht gewährleistet sind.

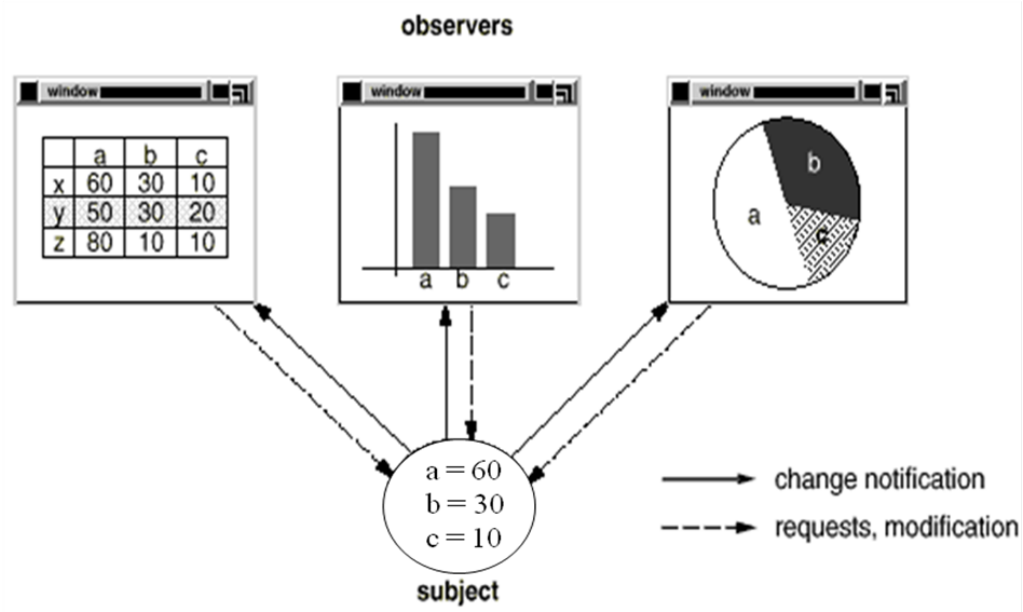

Abbildung 15: Observer Pattern und MVC

Das *subject* hat die Verantwortung für die Verwaltung der Informationen. Die Informationen sollen auf verschiedene Weise dargestellt werden. Als Tabelle mit den Werten, als absolutes Säulendiagramm und als relatives Tortendiagramm, weitere könnten hinzukommen. Änderungen in der Tabelle sollten sofort in den Diagrammen sichtbar sein. Die Diagramme sind nur eine andere Darstellung derselben Informationen und hängen von dem Objekt das diese verarbeitet ab. Das

Observer Pattern beschreibt, wie diese Beziehungen etabliert werden können, die wichtigsten Aspekte sind Subject und Observer. Die Kapselung dieser Aspekte in Objekte erlaubt eine flexible und unabhängige Wiederverwendung und Kombination der Objekte.

Anwendbarkeit Das Observer Pattern ist anwendbar, wenn

- ein Problem in zwei Aspekte zerlegt werden kann und der eine Aspekt vom anderen abhängig ist.
- Die Zustandsänderung des einen Objekts die Änderung von anderen Objekten erfordert.
- ein Objekt andere Objekte über seine Zustandsänderung informieren muss, ohne die anderen zu kennen.

16.1.3 Lösungsbeschreibung

Die Struktur

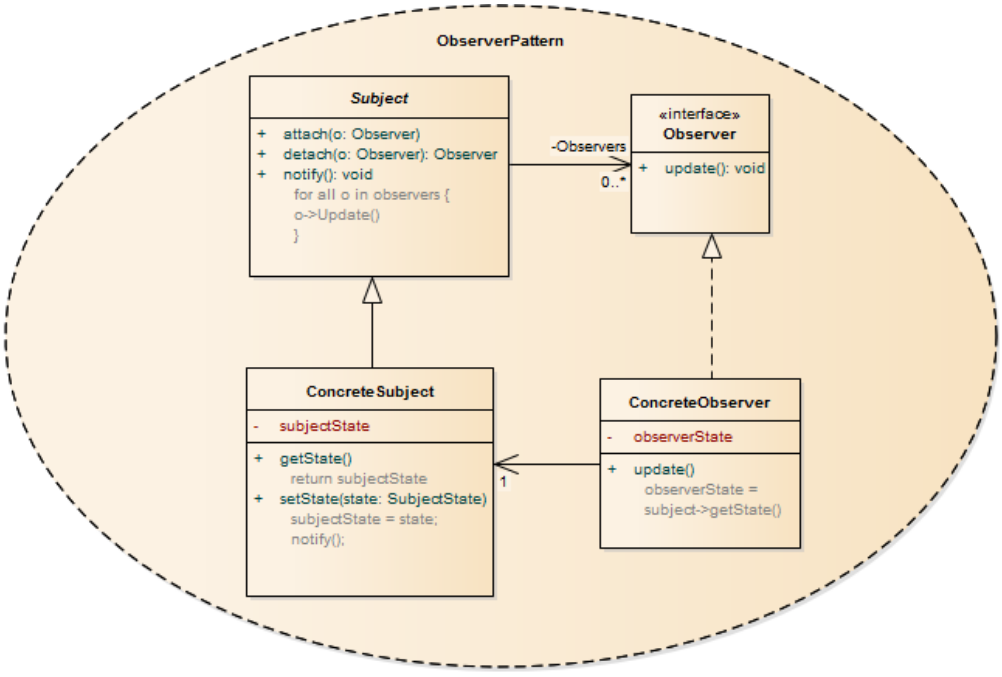

Diagramm 20: Observer Pattern Structure

Die Beteiligten

Das **abstrakte Subject** stellt den Dienst zur An- und Abmeldung (`attach, detach`), sowie zur Benachrichtigung (`notify`) der Observer über Änderungen zur Verfügung.

Das `ConcreteSubject` erbt diesen Dienst (Implementation Inheritance) und verwaltet spezifische Informationen. Ändert sich das `ConcreteSubject`, informiert es seine `Observer` über diese Änderung (`notify, update`).

Der `Observer` stellt eine Schnittstelle zur Benachrichtigung zur Verfügung (update).

Die Objekte (`ConcreteObserver`) die über Änderungen informiert werden wollen, müssen das `<<Interface>> Observer` implementieren und beim `ConcreteSubject` angemeldet (`attach`) sein. Sie kennen das `ConcreteSubject` und holen sich bei Bedarf die aktuellen Informationen.

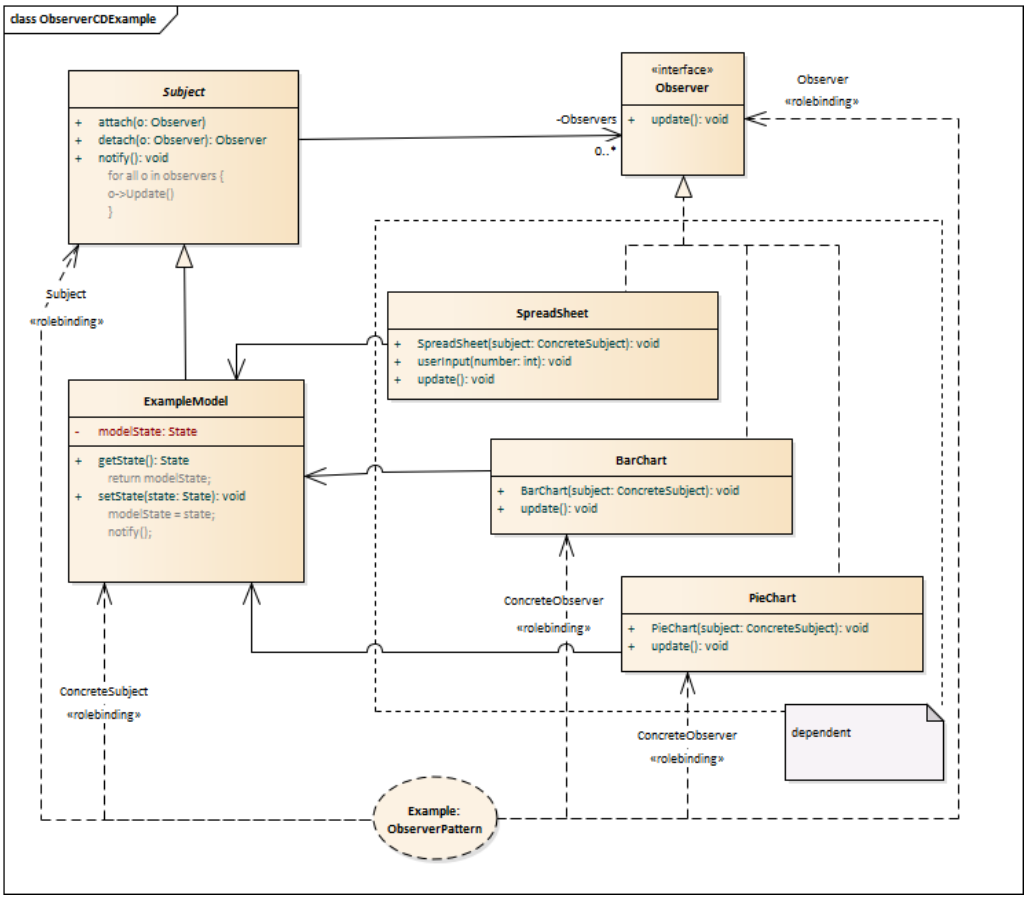

Diagramm 21: Observer Pattern: Subject mit 3 Views

Die Klassen außerhalb der gestrichelten Linie (*dependent*) sind vollständig von den konkreten Observern unabhängig.

Die Zusammenarbeit

Das Diagramm 22 auf der nächsten Seite zeigt eine Sequenz von Nachrichten zur Initialisierung der Observer (`sheet, barChart, pieChart`) mit dem Subject und die Anmeldung der Observer bei dem Subject.

Eine Benutzereingabe im `sheet` Objekt ändert das `subject`. Das `subject` propagiert die Änderung an alle angemeldeten `Observer`. Die `Observer` fragen den aktuellen Zustand des `subjects` ab und aktualisieren ihre Anzeige.

Realisierung

Bestandteil des Vertrages zwischen Subject und Observer ist:

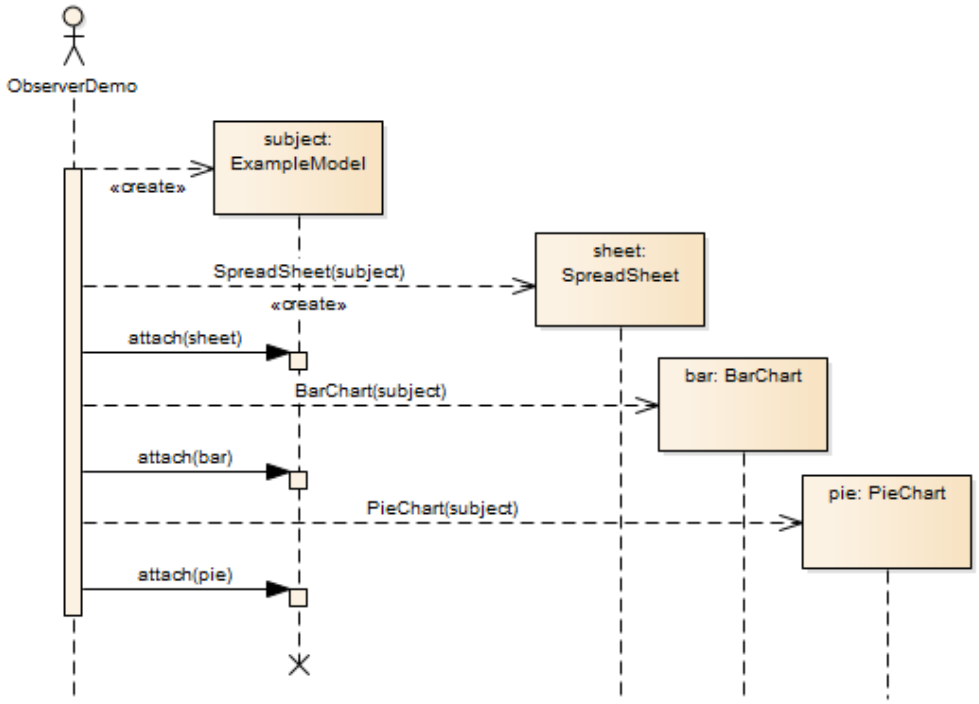

Diagramm 22: Observer Pattern Initialisierung, attach

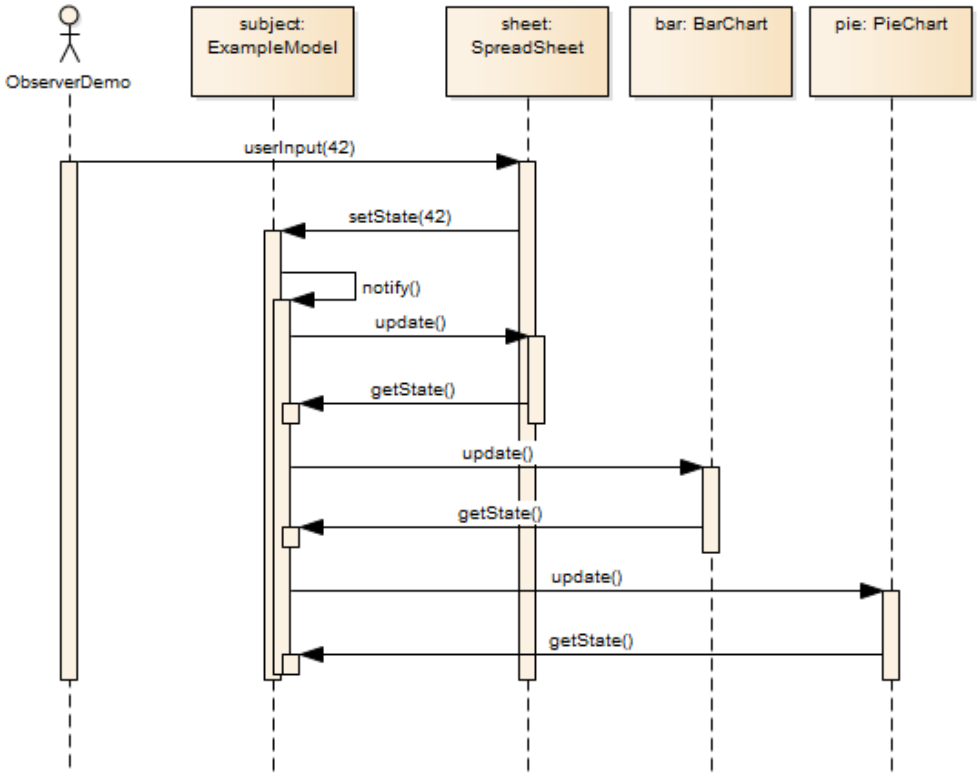

Diagramm 23: Observer Pattern, notify/update

- dass die `Observer` in der `update()` Methode den State des `ConcreteSubject` nicht ändern, da dies zu einer endlosen Rekursion führen würde.

- dass `update()` nur gesendet wird, wenn das `Subject` in einem konsistenten Zustand ist.
- die Veröffentlichung der Operationen des Subjects, die `notify()` auslösen.

Zuordnung Observer zu Subject:

- durch direkte Referenzen auf `Observer` im `Subject`
- durch ein Mapping-Objekt, das sich die Zuordnungen merkt, um Platz in den Subjects zu sparen. Die Zeitkosten für `notify()` werden dadurch aber größer.

Mehrere gleiche Subjects beobachten:

- je einen Adapter beim Subject anmelden, der die Nachricht `update()` an eine spezielle Operation des `Observer`s weiterleitet.
- als Argument von `update()` ein Event Objekt, das detaillierte Informationen über die Änderungen enthält (push-model), oder das `Subject` selbst mit senden, so dass die Quelle von `update()` ermittelt werden kann und nicht alle `Observer getState()` (pull-model) aufrufen müssen.

Implementation:

- In Java wird der Observer als <Event>Listener Interface realisiert. Die attach/detach Oparationen werden add-/ remove<Event>Listener genannt. Die Operationen des Listener Interface haben alle einen spezifischen Event als Parameter. Die <Event>Listener werden häufig als innere Klassen realisiert, um sich bei mehreren gleichen Subjects anzumelden (z.B. Button).
- In C++ besteht die Möglichkeit, die Observer als abstrakte Klasse zu realisieren. Als Klebeschicht zwischen den Observern und dem Subject, kann ein AdapterTemplate zur Verfügung gestellt werden, das die Nachrichten (update) an die übergebenen Operationen void (C::*op)(void) weiterleitet.

Wer ist für notify() verantwortlich?

- das `Subject`, bei jeder Zustandsänderung werden `update()` Nachrichten an alle `Observers` gesandt, was hohe Kosten verursachen kann.
- der Client, der die Änderung hervorruft, so dass der Zeitpunkt von `notify()` nach einer Reihe von Änderungen gewählt werden kann. Ist aber fehlerträchtiger, weil der Client `notify()` vergessen kann.

Wer räumt auf?

- wird ein `Subject` zerstört, müssen alle `Observer` darüber benachrichtig werden.
- `Subjects` sind nicht für die Zerstörung der `Observer` zuständig, da diese auch noch bei anderen `Subject`s angemeldet sein können.

16.1.4 Konsequenzen

Die `Subject`s sind nur von der Abstraktion `Observer` abhängig und kennen die konkreten Observer nicht. Die `Observer` wissen von den anderen nichts und können die Kosten für eine Änderung am `Subject` nicht abschätzen (broadcast update).

16.1.5 Bekannte Anwendungen

- MVC Model View Controller Architektur in Smalltalk
- Listener in Java

16.1.6 Kombinationsmöglichkeiten

- Mediator als ChangeManager zur Verwaltung der Observer
- State zur Verwaltung der Zustände des ChangeManager um update() nur einmal zu versenden

16.2 Strategy

Das Strategy Pattern und das State Pattern haben große Verwandtschaft. Die Implementierung ist fast identisch und entspricht dem Meta Pattern *1:1 Connection Pattern*. Die Strategy ist aber häufig, im Gegensatz zum State, länger mit dem Kontext verbunden. Außerdem könnte sie zustandsbehaftet sein.

Die States wechseln häufiger während der Laufzeit. Das State Pattern könnte auch als Strategy interpretiert werden. Ein State ist die Strategie zur Verarbeitung der Nachrichten in dem jeweiligen State.

16.2.1 Name, Kategorie, Synonyme

Name: Strategie

Kategorie: Object Behavioral

Also Known As: Policy

16.2.2 Problembeschreibung

Intention
Eine Familie von Algorithmen soll gekapselt werden, so dass sie zur Laufzeit und vom Client unabhängig ausgetauscht werden können. Kann die Strategie zur

Compile Time bereits ausgewählt werden, kann diese auch statisch gebunden werden[282].

Motivation
Je nach Situation kann ein bestimmter Algorithmus sinnvoller sein oder schneller oder weniger Platz benötigen. Wenn ein Algorithmus direkt in einer Methode realisiert wird, kann er nicht ausgetauscht werden. Zum Beispiel ist es bei geringer Anzahl von Objekten oft günstiger einen Vector zu nehmen, wenn die Anzahl einen bestimmten Wert übersteigt kann zu einer anderen Strategie der Verwaltung übergegangen werden und eine Map verwendet werden.

16.2.3 Lösungsbeschreibung

Die Struktur

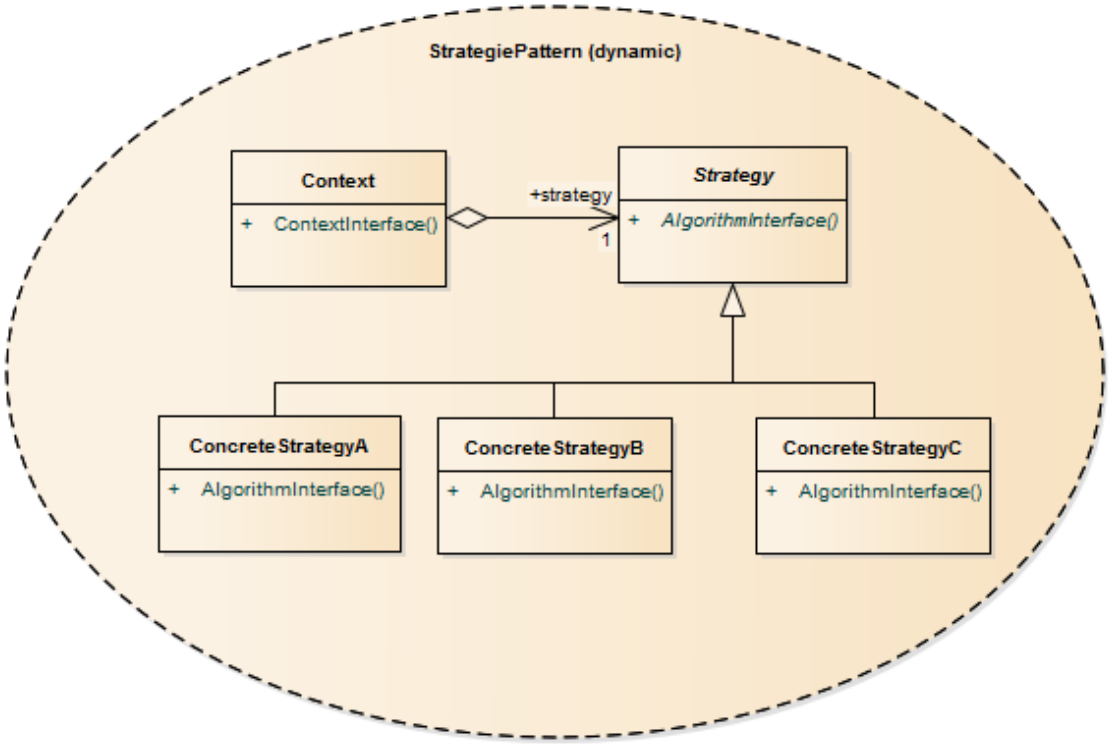

Diagramm 24: Strategy Pattern Structure

Die Beteiligten
Das Interface Strategy definiert einen Satz von Operationen die allen verwandten Algorithmen gemeinsam ist.

Die ConcreteStrategies implementieren die korrespondierenden Methoden und stellen Datenstrukturen, Attribute zur Verfügung.

Der Context ist mit einer Strategy konfiguriert und nutzt die Strategy zur Erfüllung seiner Verantwortung.

[282] Policy based Design siehe [Ale09]

Die Zusammenarbeit
Die Clients kommunizieren nur mit dem Context. Typischer Weise delegiert der Context bestimmte Nachrichten an die Strategy. Benötigen die einzelnen Strategies Zugriff auf den Context, muss dieser sich mit jeder Nachricht versenden.

16.2.4 Konsequenzen

Die Alternative wäre die Bildung von Unterklassen, dabei könnten aber während der Laufzeit die Algorithmen nicht ausgetauscht werden.

Das Pattern ist eine gute Alternative zu switch/case um ein bestimmtes Verhalten auszuwählen.

16.2.5 Implementierung

Benötigt die Strategie Zugriff auf den Context[283], kann dieser sich bei der Delegation selbst übergeben: `strategie->AlgorithmInterface(this)`.

Wenn die Verbindung zwischen Kontext und Strategie über die gesamte Laufzeit bestehen bleibt und zur Compile Time ausgewählt werden kann, kann statische Polymorphie[284] mit C++ Templates als effektive Implementierung gewählt werden, wie in Diagramm 25 gezeigt. Der Context enthält die Strategie als Member oder

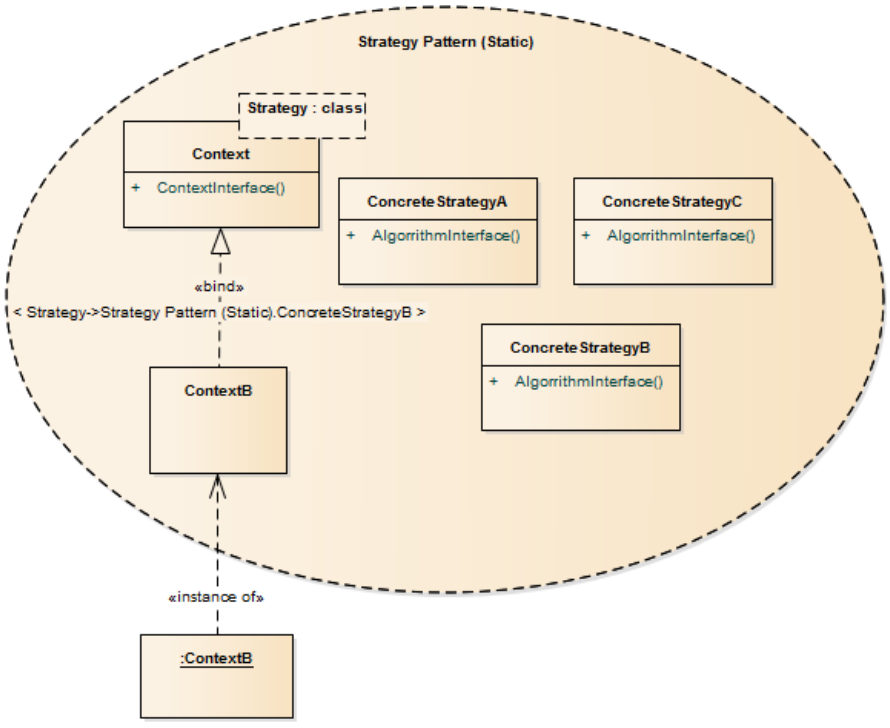

Diagramm 25: Das Strategie Pattern: statisch gebundene Strategie

[283] siehe auch section 16.3 auf der nächsten Seite
[284] section **??** auf Seite ??

erbt von dieser. Erbt der Context von seinem Template Parameter, entspricht das Policy Based Design[285]. Zur Laufzeit wird nur ein Objekt benötigt, das die Nachrichten an den Member bzw. an sich selbst delegiert.

16.2.6 Kombinationsmöglichkeiten

Flyweight (section 16.6 auf Seite 301): Strategy Objekte sind häufig Kandidaten für Flyweights.

Template Method (section 14.1 auf Seite 234) zur Erzeugung der speziellen Strategie.

State (section 16.3) ist eine spezielle Form der Strategie.

16.3 State

16.3.1 Name, Kategorie, Synonyme

Name: State

Kategorie: Object Behavioral

Also Known As: Objects for States, Role (Peter Coad)

16.3.2 Problembeschreibung

Intention
Zur Laufzeit soll das Verhalten der Objekte in Abhängigkeit ihres Zustands verändert werden.

Motivation
Ein TCPConnection Objekt muss, je nach dem in welchem Zustand es sich befindet, auf die Nachrichten unterschiedlich reagieren.

Anwendbarkeit
Das State Pattern ist anwendbar, wenn

- einfache (nicht hierarchische) Zustandsgraphen variiert werden sollen

Das State Pattern ist nicht anwendbar, wenn

- Das State Pattern eignet sich nicht zur Implementierung hierarchischer State-Machine Diagramme, wie sie mit der UML definiert werden könnnen.

[285] [Ale09]

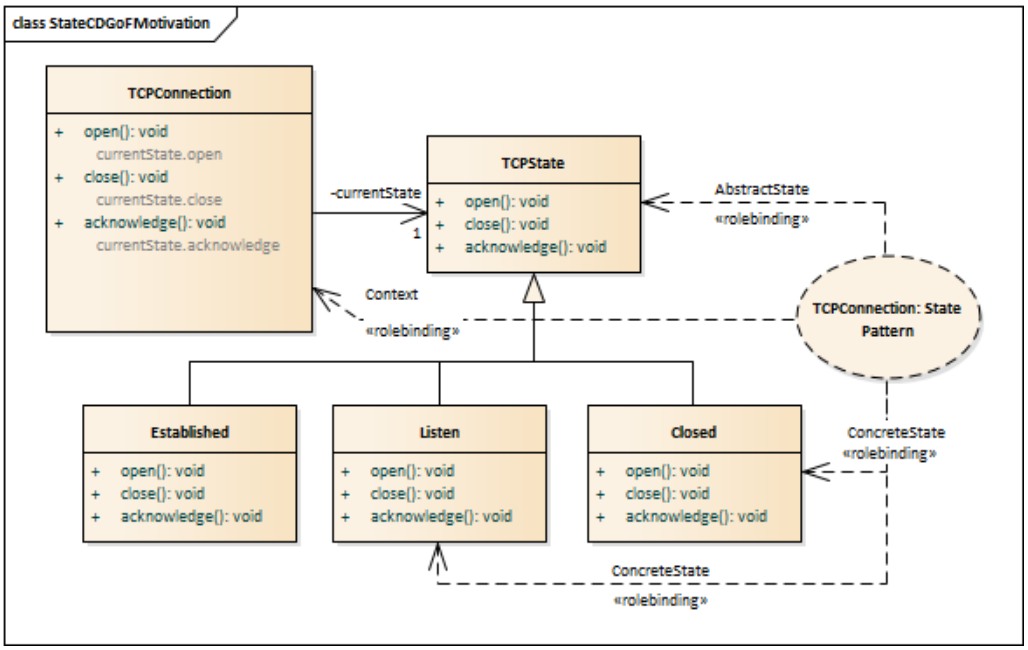

Diagramm 26: TCP Connection und ihre Zustände

16.3.3 Lösungsbeschreibung

Die Struktur

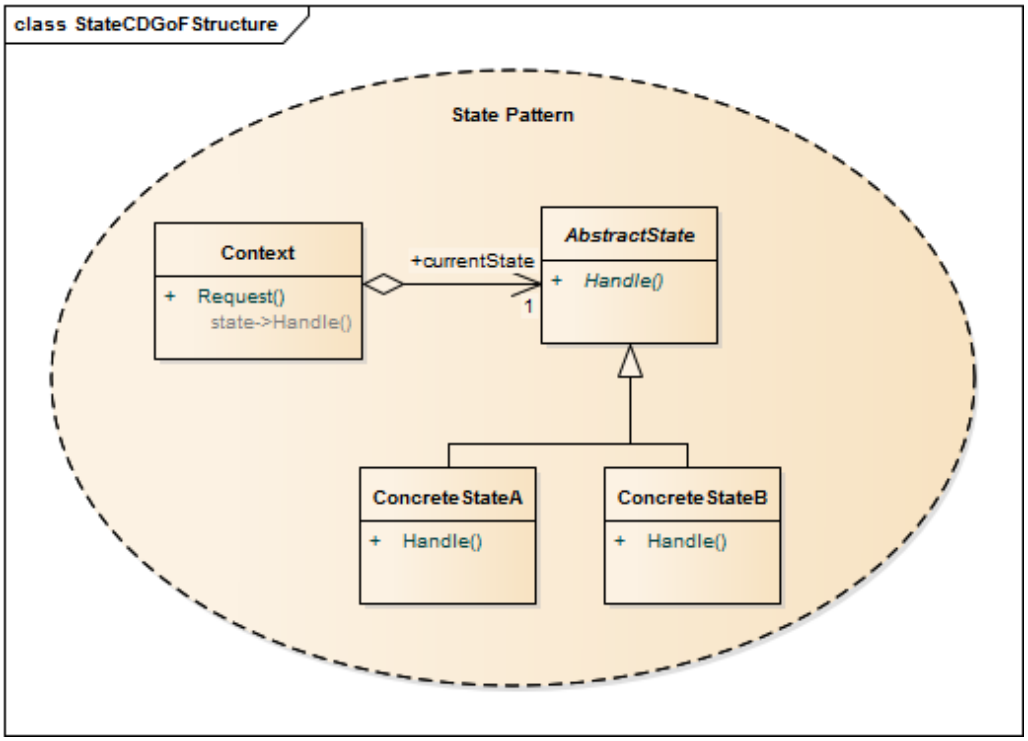

Diagramm 27: GoF State Pattern

Der Context hält eine Reference auf den aktuellen State und delegiert die Nachrichten an diesen. Die konkreten States implementieren zustandsspezifisches

Verhalten auf diese Nachrichten.

Die Beteiligten
Context (TCPConnection) definiert das Interface für die Clients und hält eine Referenz auf das aktuelle Stateobjekt.

State (TCPState) stellt dasselbe Interface zur Verfügung.

ConcreteState (Established, Listen, ...) implementiert das abstrakte State Interface mit dem Verhalten, das von diesem State erwartet wird.

Die Zusammenarbeit
Der Context leitet die Nachrichten an seinen aktuellen State weiter. Dieser verarbeitet die Nachrichten entsprechend und manipuliert dabei gegebenenfalls den Context.

16.3.4 Konsequenzen

Durch die Abstraktion der Zustände kann der Zustandsgraph explizit durch Klassen abgebildet werden. Das Verhalten kann variieren und mit einer Konfiguration bestimmt werden. Die State Objekte benötigen gegebenenfalls Zugriff auf einige Member des Contextes, die öffentliche Schnittstelle kann in manchen Fällen dazu nicht ausreichen.

16.3.5 Implementation

Das Ziel einer Implementierung des Patterns in einer Sprache ist es, den speziellen State Klassen den notwendigen Zugriff zu ermöglichen ohne die Schnittstelle des Kontextes aufzuweichen, bzw. eine falsche Benutzung zu ermöglichen.
In C++ können dazu der Context und die Klasse AbstractState sich wechselseitig als friend deklarieren. In Java werden die beiden in ein `package` gelegt. Die Klasse AbstractState stellt für die Spezialisierungen Klassenoperationen (`static`) für den Zugriff auf den Context zur Verfügung.

Der Vertrag kann durch eine Schicht von Zustandsklassen durchgesetzt werden, deren transitionsauslösende Operationen `final` deklariert werden.

16.3.6 Bekannte Anwendungen

TCP connection protocol, Verkehrssteuerung

16.3.7 Kombinationsmöglichkeiten

Flyweight (section 16.6 auf Seite 301): Werden die States zustandslos implementiert, können die Kontexte sich die State Objekte teilen.

Abstract Factory (section **??** auf Seite ??) um die State Objekte zu erzeugen.

16.4 Lösung Verkehrssteuerung, Varianz des Kommunikationsprotokolls

Um die Ampel Testen zu können, wird eine entsprechende Abstraktion ihrer Partner, den Lampen benötigt. Die Lampe in Diagramm 28 wird als Interface modelliert, die Spezialisierungen implementieren das gewünschte Protokoll bzw. unterstützen die Tests.

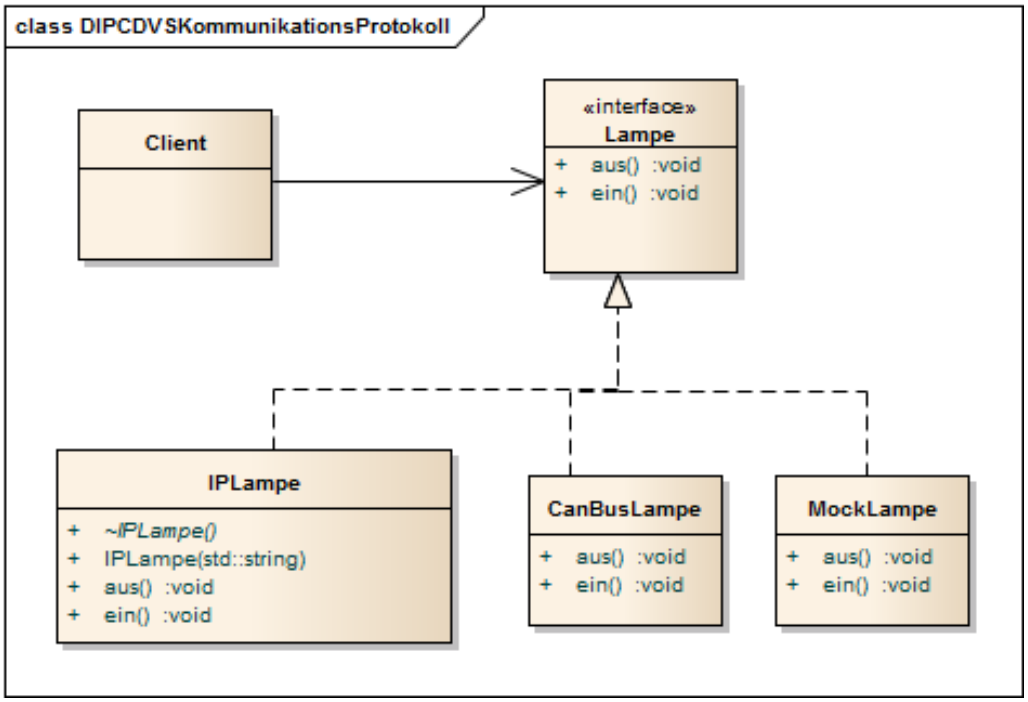

Diagramm 28: Spezialisierungen implementieren verschiedene Kommunikationsprotokolle

16.5 Lösung Verkehrssteuerung, Varianz des Verhaltens

16.5.1 Varianz des Verhaltens, Realisierung mit Switch

Die Veränderung der bestehenden Ampel durch weitere cases, welche die länderspezifischen Verhaltensweisen realisieren, soll nicht weiter diskutiert werden, die Auswirkungen auf die Wartbarkeit lassen sich leicht vorstellen und sind aus der Vergangenheit bekannt. Das Listing 273 zeigt einen Ausschnitt aus der Methode `umschalten` der Klasse Ampel die auf diese Weise realisiert ist.

Listing 273: Ampel State Machine mit Switch

```
1 void umschalten(){
2    switch(state){
3    case ROT:
4       switch(region){
5       case SOUTHERNEUROPE:
6          state = GRUEN;
```

```
7           rot.aus(); gelb.aus(); gruen.ein();
8           break;
9        default:
10          state = ROTGELB;
11          rot.ein(); gelb.ein(); gruen.aus();
12          break;
13    }
14    break;
15    case GRUEN:
16       state = GELB;
17       switch(region){
18       case SCANDINAVIA:
19          rot.aus(); gelb.ein(); gruen.ein();
20          break;
21       default:
22          rot.aus(); gelb.ein(); gruen.aus();
23       }
24    }
25    break;
26    ... // weitere Cases
27 }
```

16.5.2 Verletzung des LSP wegen Protokoll Verletzung

Der Code in Listing 273 auf der vorherigen Seite hat nicht nur Probleme mit der Übersichtlichkeit, der Erweiterbarkeit, usw. Hier wird auch der Vertrag (Vier-Phasen-Protokoll), den die Ampel ihren Partnern anbietet, verletzt. Die Kreuzung würde nach dieser Änderung nicht mehr funktionieren da die Ampel bereits nach 3 Nachrichten `umschalten()` wieder in ihrem Ausgangszustand ist.

Dasselbe Problem tritt auf, wenn von der Klasse Ampel eine SouthernEuropeAmpel abgeleitet und die Operation `umschalten()` so überschrieben werden würde, dass es den Zustand RotGelb nicht mehr gibt. Damit hätten wir das Protokoll verändert, würden wir uns nicht mehr an den Vertrag halten und damit würde der Client, die Kreuzung, nicht mehr korrekt arbeiten. Außerdem wäre fast der gesamte Code in umschalten() redundant, eine Verletzung des *„DRY“*[286] Prinzips.

16.5.3 Eine weitere Abstraktion

Die länderspezifischen Ampeln müssen also alle dieses vier Phasenprotokoll unterstützen, wenn das LSP, der Vertrag, nicht verletzt werden soll. Die Namen der States passen damit aber nicht mehr zu den Wertekombinationen der Lampen. Das Protokoll muss abstrakter gehalten werden.

[286] [HT03] “Don´t repeat yourself”

Die Abstraktion dieses Protokolls sind vier Betriebszustände: Stehen, Start vorbereiten, Fahren und Anhalten, sowie Achtung, die den Zuständen Rot, RotGelb, Grün, Gelb und Blinkend in Mitteleuropa entsprechen. Das Diagramm 29 spiegelt diese Sachverhalte wider.

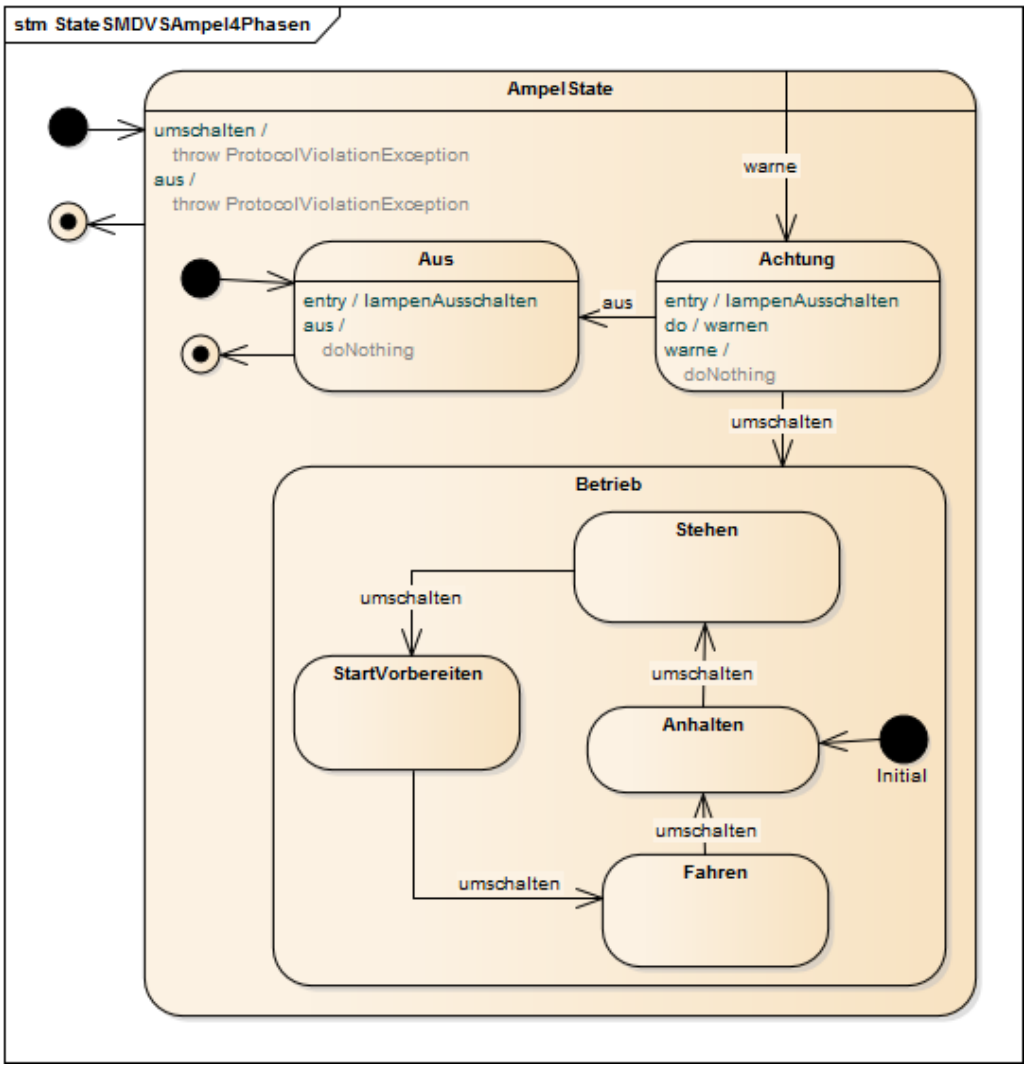

Diagramm 29: Zustände Ampel, weitere Generalisierung und Abstraktion des vier Phasen Protokolls

Welche Lampen in den Zuständen leuchten ist damit offen und den länderspezifischen Zuständen überlassen.

Damit sind wir den Weg von etwas sehr konkretem, der deutschen Ampel :-), zu der Abstraktion eines vier Phasen Protokolls gegangen, mit dem es möglich ist, beliebige Kreuzungssteuerungen zu bauen und diese mit verschiedenen Ampeln auszustatten ohne die Kreuzungen dazu verändern zu müssen!

Durch das Diagramm 29 wird explizit zum Ausdruck gebracht, dass

- alle Zustände auf `warne()` in den Zustand Achtung übergehen
- `umschalten()` und `aus()` nicht immer erlaubt sind und zu einer Ausnahme führen können

- es vier Betriebszustände gibt und zuerst der Zustand Anhalten betreten wird
- die Ampel nicht einfach ausgeschaltet werden kann, sondern zuerst in den Achtung Zustand versetzt werden muss

16.5.4 Spezialisierung der Ampel

Eine mögliche Implementierung, um das OCP zu erreichen, ist die Spezialisierung der Ampeln für die verschiedenen Regionen. Die Basisklasse implementiert den Zustandsgraphen weiterhin als **switch**, delegiert aber die `entry` Actions an die abgeleiteten Klassen gemäß dem *Template Method Pattern*[287].

Das Listing 274 zeigt eine mögliche Methode in der Basisklasse `Ampel`. Die Basisklasse stellt default Implementierungen für die `virtual entry...()` Methoden zur Verfügung, die von den Spezialisierungen bei Bedarf überschrieben werden.

Listing 274: Ampel::umschalten als Template Method

```
1  void umschalten(){
2     switch(state){
3     case AUS:
4        throw ProtocolViolationException("‘umschalten in AUS"’);
5     case Stehen:
6        state = StartVorbereiten;
7        entryStartVorbereiten();
8        break;
9     case Fahren:
10       state = Anhalten;
11       entryAnhalten();
12       break;
13    ... // weitere Cases
14    }
15 }
```

Das Diagramm 30 auf der nächsten Seite zeigt die daraus resultierende Vererbungshierarchie.

Konsequenzen
Werden verschiedene Kombinationen des selben Verhaltens benötigt, z.B. von StartVorbereiten aus der SouthernEuropeAmpel und entryAnhalten aus ScandinavianAmpel, muss der Code im schlechtesten Fall in einer sehr großen Anzahl von Klassen wiederholt werden. Die Anzahl ergibt sich aus der Kombination der möglichen Verhalten, das wird auch als kombinatorische Klassenexplosion bezeichnet.

16.5.5 Zustände als Klassen

Durch die Abstraktion der Zustände der Ampel kann ihr Zustandsgraph explizit als Klassen abgebildet werden.

[287] section 14.1 auf Seite 234

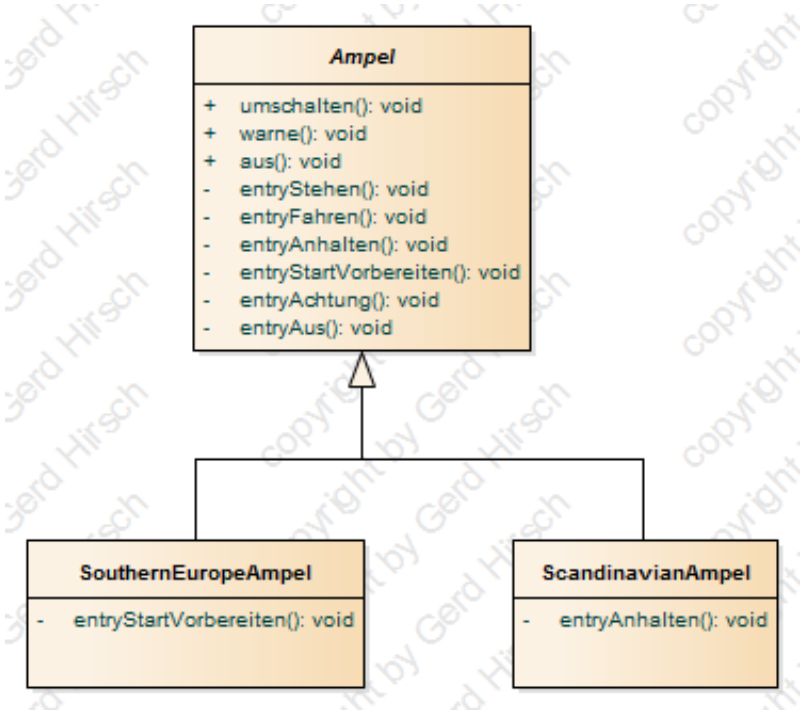

Diagramm 30: Ampel mit Template Methods

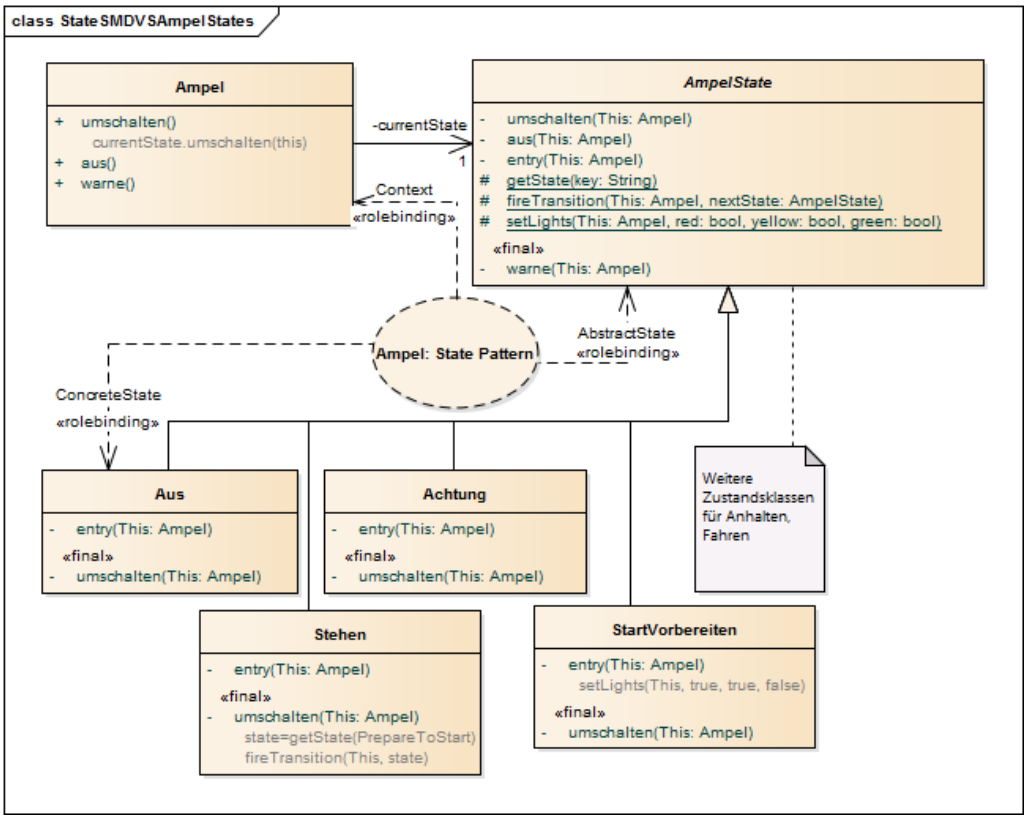

Diagramm 31: Zustände als Klassen, das State Pattern

Angenommen im Diagramm 31 sei die Ampel im Zustand Stehen. Sie delegiert daher die Nachricht `umschalten()` an ein Objekt dieser Klasse. Die Methoden `umschalten()` und `entry()` sind in der Ampel und in den jeweiligen Zustandsklas-

sen abgebildet. Der Parametername `This` ist bewußt so gewählt, da die Methoden in den States ja inhärenter Bestandteil der Ampel sind, bzw. die Ampel der Context, das eigentliche Objekt ist, auf dem die Methoden der States operieren.

Die Zustandobjekte selbst sind zustandslos und operieren auf dem Kontext, der übergebenen Ampel (`This: Ampel`). Für jeden Zustand wird daher nur ein Zustandsobjekt benötigt. Die Zustandsobjekte könnten über eine Factory verwaltet werden (Flyweight Pattern, siehe section 16.6 auf Seite 301).

Die Erweiterung erfolgt durch hinzufügen einer Klasse `SouthernEurope::StartVorbereiten` :-), deren `entry(This:Ampel)` Methode die gelbe Lampe nicht einschaltet, bzw. `Scandinavia::Anhalten`, bei dem die grüne Lampe nicht ausgeschaltet wird.

Die Zusammenarbeit
Diagramm 32 auf der nächsten Seite ist ein Vorgriff auf das nächste Kapitel. Es zeigt die Interaktion des Clients (Kreuzung) mit dem Server (Ampel).
Die Ampel delegiert die Verarbeitung der Nachrichten an ihren aktuellen Zustand und übergibt sich selbst als Parameter (z.B. `umschalten(`**`this`**`)`).
Die Methode `umschalten()`, der Klasse `Stehen` ermittelt das Objekt des nächsten Zustands mit Hilfe der StateFactory und lässt die Transition feuern:

```
fireTransition(This, state)
```

`fireTransition` setzt den Zustand der Ampel auf den übergebenen State (`startVorbereiten`) und sendet die Nachricht `entry(This)` an den aktuellen Zustand der Ampel.
Die `entry(This:Ampel)` Methode der Klasse `StartVorbereiten` schaltet die Lampen der übergebenen Ampel (`This`) entsprechend.

Die Transitionsauslösenden Operationen, `umschalten, warne, aus`, sind **`final`** gekennzeichnet und können daher nicht überschrieben werden. Damit wird der Vertrag, das vier Phasenprotokoll durchgesetzt. Die Spezialisierungen können nur die `entry(...)` oder `exit(...)` Methoden überschreiben.

Das generelle Verhalten auf die Ereignisse `umschalten(), aus()` und `warne()` wird in den entsprechenden default Methoden der Klasse `AmpelState` definiert. Sie müssen durch die speziellen Betriebszustände überschrieben werden.

- `aus()` und
- `umschalten()` werfen eine Exception und werden jeweils spezialisiert
- `warne()` ist final und überführt immer in den Zustand Achtung.
- **`static`** `fireTransition(This:Ampel, nextState:AmpelState)` weist der Ampel ihren neuen Zustand zu und sendet die Nachricht `entry(This:Ampel)` an den neuen Zustand
- **`static`** `setLights(This:Ampel, red:bool, yellow:bool, green:bool)` schaltet die jeweilige Lampe ein oder aus.
- Die übergebene Ampel (**`this`**), der Context, wird durch den Parameter `This` in den Methoden der States repräsentiert.

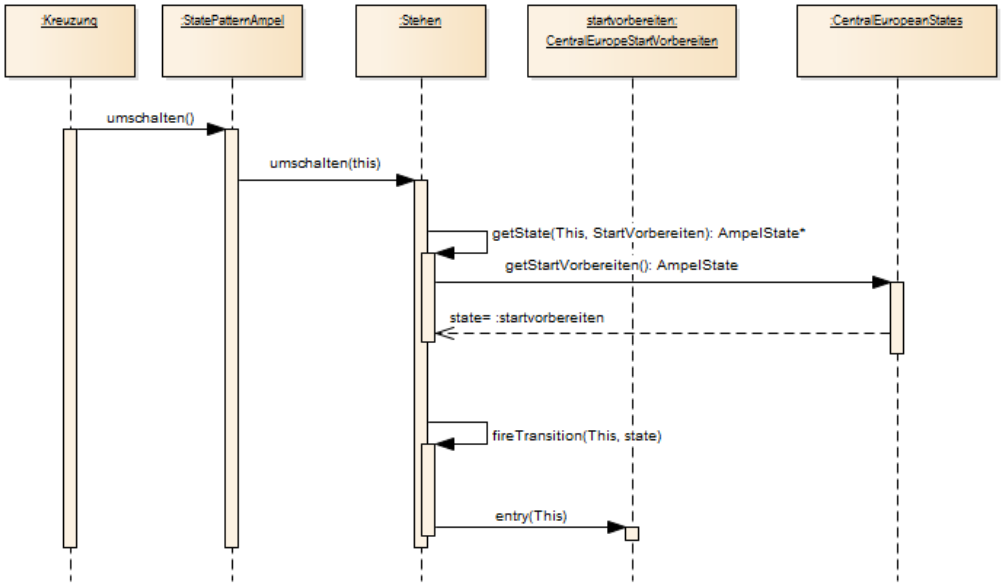

Diagramm 32: State Pattern Delegation der Nachrichten an den aktuellen State

16.5.6 Konsequenzen

Auf der Basis des State Patterns lässt sich das Verhalten bedarfsgerecht konfigurieren ohne die Klassen `Ampel` oder `AmpelState` verändern zu müssen. Das OCP bzgl. der Varianz des Verhaltens der Ampel ist damit vollständig erreicht.

Von außen betrachtet, scheint die Ampel ihren Typ zu ändern, da sie ja jedesmal auf die Nachrichten, z.B.: umschalten auf andere Weise reagiert.

Die Konstruktion **`switch`**`(state)/`**`case`** in den Methoden der Ampel ist vollständig eliminiert. Das Verhalten der Ampel wird durch den aktuellen State (`currentState`) bestimmt. Jeder State enthält genau die Algorithmen, die zu den jeweiligen Nachrichten in diesem State passen.

Die Zustandsübergänge sind durch die Klassenstruktur abgebildet. Damit ist die StateMachine explizit durch den Programmtext ausgedrückt.

Ein Zustandsübergang kann in multithreaded Umgebungen atomar gestaltet werden, da dieser immer in der Methode `fireTransition(...)` der Basisklasse `AmpelState` durchgeführt wird und über den Kontext, den Parameter `This:Ampel` synchronisiert werden kann.

Werden die State Klassen zustandslos implementiert, können die Objekte von mehreren Kontexten (Ampeln) gemeinsam benutzt werden, es wird also von jeder State Klasse jeweils nur ein Objekt benötigt (Flyweight Pattern, siehe section 16.6 auf Seite 301).

Die States benötigen dafür aber den Zugriff auf **`private`**`/`**`protected`** Elemente des Kontextes. Diese einfach **`public`** zu machen würde zu einer Verletzung des Geheimnisprinzips / der Kapselung führen. Jeder beliebige Client könnte die Integrität des Kontextes zerstören.

z.B.: `ampel.rot.aus()`

Die Basisklasse `AmpelState` der States, stellt deshalb zur Manipulation der `Ampel` **`protected`** Methoden zur Verfügung. Mit entsprechenden sprachlichen Mitteln kann die Klasse `Ampel` der Klasse `AmpelState` den Zugriff auf ihre geschützten Member gewähren. Die darus resultierende zyklische Abhängigkeit zwischen den Klassen `Ampel` und `AmpelState` ist aus Designsicht fragwürdig und muss erst einmal begründet werden:

1. Die Klasse `Ampel` ist der Kontext, das Gedächtnis, für die Daten (rot, gelb, grün) und ihren Zustand (`currentState`). Sie ist für ihre eigene Konsistenz (Invarianz) verantwortlich, ihre Member müssen daher **`private`** deklariert sein, damit sie von außen nicht manipuliert werden können (Geheimnisprinzip). Sie delegiert alle Nachrichten an ihren `currentState`.

2. Die Klasse `AmpelState` und ihre Spezialisierungen gehen aus dem Kontext hervor, sie sind das Herzstück der Abstraktion `Ampel`: ihr Verhalten!
   Sie würden ohne den Kontext nicht existieren. Sie kapseln das zustandsbedingte Verhalten auf die Nachrichten, in den Methoden die die Ampel an ihren `currentState` delegiert, z.B. `umschalten()` und den entry- und exitActions und benötigen dafür den Zugriff auf den Kontext, auf die privaten Member der `Ampel`.

3. Einem `AmpelState` sollte niemand Nachrichten senden können und damit eine `Ampel` manipulieren können, ausser der `Ampel` selbst.

   z.B.: `state.umschalten(ampel)` oder `state.entry(ampel)`

   Daher sind die Nachrichten, die die `Ampel` an den `currentState` delegiert, in `AmpelState` so zu schützen, dass nur die `Ampel` darauf zugreifen kann.

4. Es können weitere State Klassen hinzugefügt werden ohne die Klassen `Ampel` oder `AmpelState` ändern zu müssen. Damit ist das Geheimnisprizip gewährleistet und das Wichtigste, das OCP erreicht.

5. Alle aufgeführten Punkte zusammen sollten ausreichen, die zyklische Abhängigkeit zu rechtfertigen.

16.5.7 Implementation

Das Ziel einer Implementierung des Patterns in einer Sprache ist es, den speziellen State Klassen den notwendigen Zugriff auf den Kontext zu ermöglichen ohne die Schnittstelle des Kontextes aufzuweichen, bzw. eine falsche Benutzung zu ermöglichen[288]. z. B.: dass ein Client fälschlicherweise einem State Objekt eine Ampel übergibt:

`state.umschalten(ampel)` oder `state.entry(ampel)`
oder die Lampen der Ampel direkt manipulieren kann:
`ampel.rot.aus()`. Das sollte nur innerhalb der `Ampel` bzw. innerhalb der Klasse `AmpelState` möglich sein.

[288] section **??** auf Seite ??

Die Member der `Ampel` und die Operationen, die die Ampel an ihren `currentState` delegiert, in der Klasse `AmpelState` werden daher **`private`** deklariert.

Der daraus resultierende notwendige wechselseitige Zugriff auf **`private`** Member kann in *C++* durch wechselseitige `friend` Deklarationen in der `Ampel` und der dazugehörigen `AmpelState` Klasse ermöglicht werden. Mit den `friend` Deklarationen wird ein klassenübergreifender Raum für den Zugriff auf **`private`** oder **`protected`** Elemente geschaffen, auf die der Zugriff von Außen nicht möglich ist.

Listing 275: class Ampel State Pattern

```
1  //Ampel.h
2  class AmpelState; // forward declaration
3
4  class Ampel { //Context
5  public: // Client Interface
6     void umschalten();
7     void aus();
8     void warne();
9  private:
10    friend class AmpelState;
11    AmpelState* currentState;
12    Lampe *rot, *gelb, *gruen;
13    void setLights(bool red, bool yellow, bool green);
14 };
15 inline
16 void Ampel::setLights(bool red, bool yellow, bool green){
17 // hier die Lampen schalten
18       if(red) rot->ein();
19       else  rot->aus();
20       ...
21 }
22 //Ampel.cpp
23 void Ampel::umschalten() {
24    currentState->umschalten(this); // delegation
25 }
26 // aus(), warne() dto.
```

Listing 276: class AmpelState private Ampel Interface

```
1  class AmpelState{
2     ...
3  private:
4     // Interface für die Ampel
5     friend class Ampel;
6     virtual void umschalten(Ampel * const This) { throw ProtocolViolationException;
          }
7     virtual void aus(Ampel * const This) { throw ProtocolViolationException; }
8     virtual void warne(Ampel * const This) {
9        fireTransition(This, getState(StateName::Achtung))
10    }
11
12    // State Machine Operations
```

```
13    virtual void entryAction(Ampel * const This) = 0
14    virtual void exitAction(Ampel * const This) = 0
15 };
```

Das Listing 276 auf der vorherigen Seite zeigt die mit den Operationen in der `Ampel` korrespondierenden Operationen in `AmpelState`. Sie sind **`private`**, so dass ein Client nicht irrtümlich direkt einem State eine `Ampel` übergibt und diese in einen anderen Zustand überführt. Nur die `Ampel` selbst kann diese Nachrichten an ihren `currentState` senden, weil sie `friend` deklariert ist.

Listing 277: class AmpelState protected Operations

```
1  // AmpelState.h
2  #include "Ampel.h"
3
4  class AmpelState{
5  public:
6     static StateFactory* stateFactory; // wird vom Builder initialisiert
7     ...
8  protected: // Access Operations
9     static void fireTransition(Ampel * const This, AmpelState * const nextState) {
10       This->currentState->exitAction(This);
11       This->currentState = nextState;
12       nextState->entryAction(This);
13    }
14    static void setLights(Ampel * const This, bool rot, bool gelb, bool gruen)
15    {  This->setLights(rot, gelb, gruen); }
16
17    // State Machine Operation
18    static AmpelState* getState(Ampel* This, StateName key)
19    { return stateFactory->getState(key);}
20 };
```

Da die friend Declaration nicht vererbt wird, definiert die Klasse `AmpelState` für den Zugriff auf die **`private`**/**`protected`** Elemente der `Ampel` zwei Klassenoperationen (`fireTransition()`, `setLights()`). Diese sind **`protected`** deklariert, können also nur von abgeleiteten Klassen verwendet werden.
Das Listing 277 zeigt die Methoden von `AmpelState` die auf die privaten Elemente der `Ampel` (`currentState` und `This->setLights(..)`) zugreifen können, weil `AmpelState` in `Ampel` `friend` deklariert ist. Die Spezialisierungen von `AmpelState` können die **`protected`** Klassenoperationen (**`static`**) nutzen um den Kontext zu manipulieren, z.B.: die Lampen ein, bzw. auszuschalten oder eine Transition auslösen. Die Objekte für den `nextState` von `fireTransition` werden über `getState (key)` ermittelt. Die abgeleiteten Klassen von `AmpelState` benötigen kein Wissen darüber, wie diese Operationen implementiert sind. Das entspricht dem Demeter Prinzip[289].

Dadurch wird das OCP vollständig erreicht: es können weitere Stateklassen hinzugefügt werden, das Verhalten also variiert werden, ohne dass die `Ampel` oder die Basisklasse `AmpelState` verändert werden muss.

[289] [HT03]

In diesem Beispiel, wird ein Klassenattribut `stateFactory` verwendet, an das `AmpelState::getState()` die Anfrage delegiert. Damit werden sich alle Ampeln gleich verhalten. Sollte die Verkehrssteuerung an der Grenze mit Ampeln auf beiden Seiten zum Einsatz kommen, kann die StateFactory von jeder Ampel verwaltet werden und die Anfrage wird nicht direkt an die StateFactory weitergeleitet, sondern an den aktuellen Kontext, die Ampel. Dafür muss der Operation `AmpelState::getState(This:Ampel, key){ return This->getState(key);}` ein weiterer Parameter, die Ampel hinzugefügt werden.

In Java kann die Verletzung der Kapselung ebenfalls vermieden werden. Anstatt einer wechselseitigen `friend` Deklaration müssen die beiden Klassen `Ampel` und `AmpelState` in demselben **package** liegen und die Member entsprechend package scope bzw. **protected** deklariert werden. Die Zugriffsoperationen auf die package scope Member der `Ampel` müssen genauso gestaltet werden wie in C++.

Ohne diese Zugriffsoperationen müssten die Stateklassen in Java in dasselbe package gelegt werden und in C++ müssten entsprechende friend Deklarationen für alle Stateklassen in der Ampel eingefügt werden. Was im Widerspruch zum OCP steht.

16.6 Flyweight

16.6.1 Name, Kategorie, Synonyme

Name: Flyweight

Kategorie: Object Structural (Creational)

16.6.2 Problembeschreibung

Intention

Eine große Menge fein granularer Objekte soll gemeinsam genutzt werden (shared), so dass es von jeder Ausprägung nur ein Objekt geben muss.

Motivation
Der Speicherbedarf und die Kosten für die Erzeugung der Objekte soll minimiert werden.

16.6.3 Lösungsbeschreibung

Die Struktur
Die Flyweight Factory erzeugt und verwaltet die angeforderten Objekte.

Die Beteiligten
Flyweight definiert ein Interface, über das den konkreten Objekten der externalisierte (`extrinsicState`) Zustand übergeben werden kann.

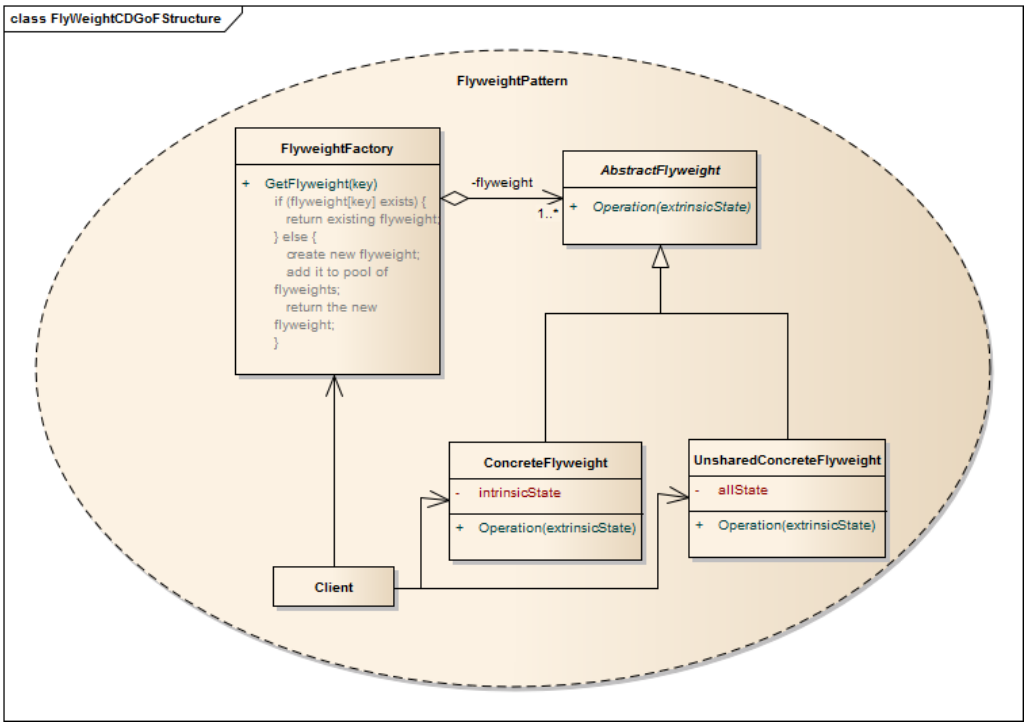

Diagramm 33: FlyWeight Pattern

Beispiel: Ampel `state->umschalten(this)` hier ist die Ampel der Kontext der den extrinsicState repräsentiert. Die einzelnen States haben keinen Zustand (intrinsicState).

ConcreteFlyweight: implementiert das Flyweight Interface und fügt für den intrinsicState Attribute hinzu.

UnsharedConcreteFlyweight: hält typischer Weise andere Flyweight Objekte die gemeinsam genutzt werden. Sie haben jeweils einen eigenen Key und werden nur von einem Client genutzt.

FlyweightFactory: erzeugt und verwaltet die Objekte. Wird ein Objekt mit einem bestimmten Schlüssel (Key) angefordert wird dieses, wenn es noch nicht vorhanden ist, erzeugt und geliefert.

Client: hält Referenzen auf die angeforderten Objekte und verwaltet den externalisierten Zustand.

Die Zusammenarbeit
Die Zustände der Flyweight Objekte müssen nach extrinsic und intrinsic unterschieden werden. Intrinsische Zustände müssen im Flyweight verwaltet werden, die extrinsischen States müssen dem Flyweight vom Client bei Bedarf übergeben werden.

Die Clients dürfen die Flyweight Objekte nicht selbst erzeugen, sondern müssen sich immer an die Factory wenden. Damit ist gewährleistet, dass die Flyweight korrekt gemeinsam genutzt werden können.

16.6.4 Konsequenzen

Es entstehen Kosten für den Transfer und die Berechnung bzw. das Suchen der externalisierten States. Die relativen Kosten für den Speicher werden mit zunehmender Anzahl der gemeinsam genutzten Objekte immer geringer.

16.6.5 Bekannte Anwendungen

Das Konzept Flyweight wurde für einen Editor entwickelt. Der Editor nutze Glyph Objekte um die einzelnen Zeichen zu repräsentieren. Für jedes Zeichen und die dazugehörigen Styles (Font, Color, usw.) wurde ein Glyph Objekt erzeugt. Der intrinsic State der Glyph Objekte beinhaltete das Zeichen und die Style Informationen. Der extrinsische Zustand war lediglich die Position des Zeichens. Der Editor hatte dadurch eine gute Performanz und benötigte wenig Speicher.

16.6.6 Kombinationsmöglichkeiten

State/Strategie (section 16.3 auf Seite 288) um die benötigten State Objekte zu verwalten. Dadurch wird auch die Abhängigkeit der konkreten State Klassen aufgelöst.

Abstract Factory (section ?? auf Seite ??) um die Flyweight Objekte zu erzeugen.

Composite (section **??** auf Seite ??) um eine logische Hierarchie der Flyweights zu verwalten.

Singleton (section **??** auf Seite ??) um den Zugriff auf die einzige Flyweight Factory zu ermöglichen.

Teil VI

STL

17 STL Basics

17.1 Einführung

Was ist die STL? Welche Grundlagen werden zum Verständnis benötigt? [290]
Um die STL zu verstehen, ist ein grundlegendes Verständnis von C++ notwendig. Insbesondere das Konzept von Klassen, Vererbung, Templates, exception handling, Funktionen und Namespaces und deren Verwendung sollte bekannt sein.

Die STL ist ein erweiterbares Framework von Komponenten:

1. Input/Output (I/O) Klassen
2. Strings und Regular Expressions
3. Algorithmen und Datenstrukturen
4. Klassen für Multithreading und Nebenläufigkeit
5. Klassen zur Unterstützung von Internationalisierung
6. Numerische Klassen
7. Eine Fülle von Hilfsklassen

17.2 Algorithmen und Datenstrukturen

Algorithmen und Datenstrukturen (Container) sind Komponenten zur

- Manipulation von Mengen
- Manipulation von darin enthaltenen Objekten
- Abstract spezifiziert durch ihre Eigenschaften
  - Big O-Notation (Angaben oft amortized spezifiziert)
  - Protokolle

17.3 Komplexität und die Big O-Notation

Die O-Notation beschreibt das relative Laufzeitverhalten von Operationen und Algorithmen bezogen auf die Anzahl der Elemente in einer Menge.

[290] empfohlene Nachschlagewerke [Jos04] für C++03 und [Jos12] für C++11 und http://en.cppreference.com/w/

Operationen:

- suchen / finden
- einfügen / löschen
- sortieren
- extrahieren / zusammenführen
- kumulieren / transformieren

Das Ideal ist O(1), was bedeutet, die Anzahl der Elemente hat keinen Einfluss auf die Laufzeit des Algorithmus.

Die Tabelle 10 listet die typischen Werte für die Komplexität und die dazugehörige O-Notation in aufsteigender Reihenfolge.

Tabelle 10: Typische Werte für die Komplexität

Typ	Notation	Bedeutung
Konstant	O(1)	Anzahl der Elemente hat keinen Einfluss auf die Laufzeit
Logarithmisch	O(log(n))	Die Laufzeit wächst logarithmisch mit der Anzahl der Elemente
Linear	O(n)	Die Laufzeit wächst linear mit der Anzahl der Elemente
n-log-n	O(n * log(n))	Die Laufzeit wächst mit dem Produkt aus der Anzahl der Elemente und dem Logarithmus der Anzahl
Quadratisch	O(n^2)	Die Laufzeit wächst quadratisch mit der Anzahl der Elemente

Die Tabelle 11 zeigt die Zunahme von Operationen in Abhängigkeit der Komplexität und der Anzahl der Elemente in einer Menge.

Tabelle 11: Laufzeit in Abhängigkeit der Komplexität und der Anzahl der Elemente

Komplexität		Anzahl der Elemente					
Typ	Notation	1	2	5	10	100	1000
Konstant	O(1)	1	1	1	1	1	1
Logarithmisch	O(log(n))	1	2	3	4	7	10
Linear	O(n)	1	2	5	10	100	1000
n-log-n	O(n * log(n))	1	4	15	40	700	10000
Quadratisch	O(n^2)	1	4	25	100	10.000	1.000.000

Die Big O-Notation berücksichtigt konstante Faktoren der Algorithmen nicht. Es spielt keine Rolle, wie lange ein Algorithmus benötigt. Daher kann ein Algorithmus, der durch die O-Notation schlecht beurteilt wird, in der Praxis ein besseres Ergebnis bei kleinen Mengen erzielen, weil der konstante Faktor bei dem scheinbar besseren Algorithmus erst bei großen Mengen vernachlässigbar ist.

Einige Laufzeit Definitionen der STL sind als *amortized* spezifiziert. Das bedeutet, dass diese Spezifikationen nicht immer eingehalten werden können. Eine einzelne Operation kann mehr Zeit in Anspruch nehmen als es durch die Spezifikation definiert ist. Zum Beispiel ist die Operation `std::vector::push_back` mit O(1) spezifiziert. Reicht der reservierte Platz aber nicht aus, beschafft der Container neuen Speicher und kopiert alle Elemente in den neuen Speicher, die auslösende Operation dauert entsprechend länger.

17.4 Design der STL Container

Das Design der STL Container basiert auf Abstraktion durch Generalisierung.

17.4.1 Design Ziele

Die Ziele sind

- geringe Laufzeit
- geringer Speicherbedarf
- Exceptionsicherheit (soweit möglich)

Das Ideal wären Container neutrale Algorithmen, was aber zu teuer oder nicht machbar ist. Die Austauschbarkeit von Containern war **nicht**[291] das Ziel des Designs. Tatsächlich unterscheiden sich die Container in ihren Schnittstellen erheblich. Es werden jeweils nur die Operationen zur Verfügung gestellt, die bzgl. der Laufzeit sinnvoll implementiert werden können.
So stellt `std::vector` einen Index Operator `operator[index]` zur Verfügung, eine `std::list` aber nicht. Ausserdem werden in der STL, von einigen Ausnahmen abgesehen, keine Bereichsprüfungen vorgenommen, was Laufzeit Kosten verursachen würde, der Programmierer muss selbst darauf achten, z.B. nicht über das Ende eines Vectors hinaus zu greifen, was *undefined behavior* nach sich zieht. Der Index Operator von `std::vector` macht keine Bereichsprüfung aber die Methode `vector::at(index)`, sie wirft eine `std::out_of_range` Exception, was unüblich in der STL ist.

Die Verständlichkeit im Sinne von selbsterklärend kann auch kein Ziel gewesen sein, ohne die Kenntnis der Konzepte ist die Dokumentation nicht sehr hilfreich;-(

17.4.2 Komponenten

Die wesentlichen Kategorien von Komponenten sind

- Container (Objekte zur Verwaltung von Mengen)
- Algorithmen (Funktionen)

[291] [Mey06b] Item 2 “Beware the illusion of container-independent code”

- Iteratoren (Pointer / Interator Pattern)
- Functors (Smart Functions `operator()`)

Die Zusammenarbeit der Algorithmen mit den Containern basiert auf dem Iterator Pattern wie es in section **??** auf Seite ?? beschrieben ist. Die Anpassung an die Benutzer spezifischen Bedürfnisse erfolgt über Callable Entities insbesondere über Functors und seit C++11 über Lambda Expressions.

Die Abbildung 16 zeigt die Komponenten und ihre Beziehungen.

Die RandomAccessIteratoren der STL haben dieselbe Schnittstelle wie native Pointer. Andere Iterator Typen schränken die Pointer Schnittstelle ein.

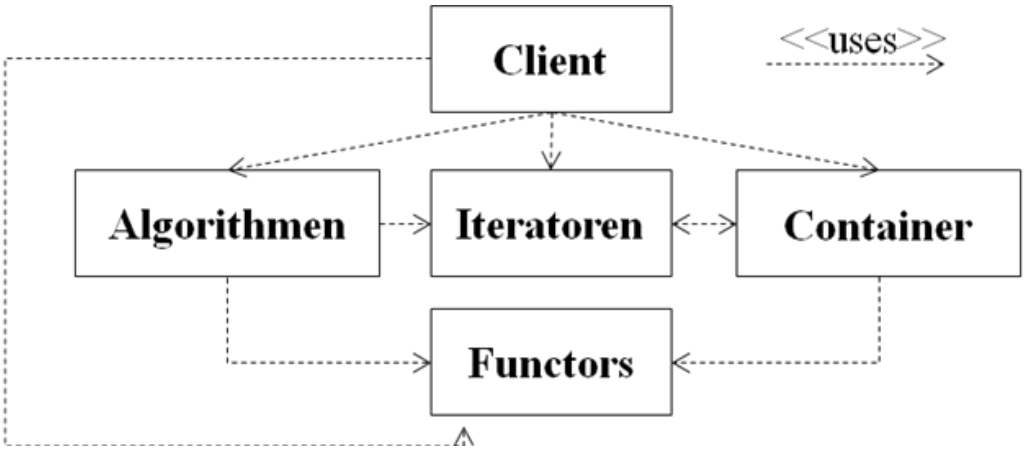

Abbildung 16: Die STL Komponenten und ihre Beziehungen

17.5 Container

Die Container der STL sind alle als Templates realisiert. Die Container unterscheiden sich in der Struktur in der sie die enthaltenen Objekte verwalten. Es gibt Container, die einen oder mehrere zusammenhängende Speicherblöcke zur Verwaltung der Objekte benutzen und es gibt Knoten basierte Container. Seit C++11 sind auch Hashtables in der STL verfügbar. Aus der jeweiligen Struktur ergibt sich das Laufzeitverhalten der Operationen, ausgedrückt in der Big O-Notation.

Die Deklarationen einiger Container Templates der STL sind in Listing 278 abgebildet.

Listing 278: Die Deklarationen der Container der STL

```
1 // a container that encapsulates constant size arrays.
2 template<class T, std::size_t N>
3 struct array;
4
5 //a sequence container that encapsulates dynamic size arrays.
6 template<class T, class Allocator = std::allocator<T>>
7 class vector;
8
9 // an indexed sequence container that allows fast insertion
```

```
// and deletion at both its beginning and its end.
template<class T, class Allocator = std::allocator<T>>
class deque;

// a container that supports constant time insertion and removal
// of elements from anywhere in the container.
template<class T, class Allocator = std::allocator<T>>
class list;

// a container that supports fast insertion and removal of elements
// from anywhere in the container.
template<class T, class Allocator = std::allocator<T>>
class forward_list; // singly-linked list

// associative container that contains a sorted set of unique objects of type Key.
template<class Key, class Compare = std::less<Key>,
         class Allocator = std::allocator<Key>>
class set;

// an associative container that contains a sorted set of objects of type Key.
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class multiset;

// a sorted associative container that contains key-value pairs with unique keys.
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;

// an associative container that contains a sorted list of key-value pairs.
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class multimap;

// an associative container that contains set of unique objects of type Key.
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;

// an associative container that contains set of possibly non-unique objects of
```

```
    type Key.
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_multiset;

// an associative container that contains key-value pairs with unique keys.
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;

// an unordered associative container that supports equivalent keys
// (an unordered_multimap may contain multiple copies of each key value)
// and that associates values of another type with the keys.
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_multimap;

// stores and manipulates sequences of char-like objects.
template<
    class CharT,
    class Traits = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
> class basic_string;
```

Container haben ausschließlich Value Semantik, das bedeutet:

- Die Objekte werden werden kopiert, deshalb sind
- nur homogene Mengen möglich
  bei Objekten abgeleiteter Klassen, wird nur der Teil der Basisklasse kopiert: typeslicing[292]
- die Typen, die in Containern verwaltet werden, müssen bestimmte Eigenschaften haben
- Veränderungen des Containers invalidieren (in den meisten Containern)
  - Iteratoren
  - Referenzen

[292] siehe section 6.16.4 auf Seite 53

- Pointer

17.5.1 Eigenschaften von Container Elementen

Der Typ der Elemente eines Containers steht als Namen zur Verfügung:
`ContainerType<ElementType>::value_type` ist der Typ der Elemente in dem Container.

Die Eigenschaften, die Container Elemente haben müssen sind:

- kopierbar und/oder Moveable (Copy/MoveConstructor) (Move ab C++11)
  - `value_type(value_type const& lValue)`
  - `value_type(value_type && rValue)` (C++11)
  - kopieren sollte billig und muss korrekt sein
  - Die Kopie muss **äquivalent** zum Original sein
- zuweisbar (Assignment Operator)
  - `value_type& operator=(value_type const& lValue)`
  - `value_type& operator=(value_type && rValue)` (C++11)
  - Die Kopie muss **äquivalent** zum Original sein
- zerstörbar
  - `public ~value_type()noexcept()`
  - Destructor darf keine Exception werfen
- erzeugbar (Default Constructible)
  - `public value_type()`
- vergleichbar (Vergleichsoperator)
  - `bool operator==(value_type const& lhs, value_type const& rhs)`
  - `bool operator==(value_type const& rhs)` (Member operator)
- sortierbar (Äquivalenzoperator)
  - default ist der kleiner Operator (**<**)
    repräsentiert durch den Typ `less<T>`[293]

17.5.2 Regeln für den Umgang mit Containern

- `std::vector` als default Container verwenden

[293] Äquivalenz siehe section 6.25.14 auf Seite 99

  - Ausnahme: `vector<bool>` vermeiden[294]! Ist ein non-standard Container, enthält keine boolean Values!
  - Mit `reserve(numElements)` das Vergrößern des Vectors während der Verwendung vermeiden
  - `std::vector<char>` als Ersatz für `std::string`
    * in multi-threaded Environments[295]
    * als C-API Schnittstelle: `lagacyFunction(&v[0], v.size())`
    * Memory layout ist garantiert wie ein Array
  - `std::vector` sortiert als Ersatz für assoziative Container für bessere Performance- und Speicherbedarf[296]
- `std::list` für häufige ein/ausfüge Operationen in der Mitte
- `std::deque` wenn ein/ausfügen meistens am Anfang oder am Ende benötigt wird
- Elemente kopieren sollte billig und muss korrekt sein
- `Container<std::auto_ptr>` sind nicht möglich
- `empty()` der Operation `size()== 0` vorziehen
- range based member Operationen den single element Operationen vorziehen
- bei gleichnamigen Algorithmen, die Operationen des Containers verwenden
- non-standard STL Container sind keine vollwertigen Container (`arrays, bitset, valarray, queue, priority_queue, ...`)
- bei assoziativen Containern keine in-place modification der Elemente (Key)
- `map::operator[key]` erzeugt immer einen Eintrag, wenn noch keiner mit dem Key vorhanden ist[297]

Array basierte Container

`vector, deque` und `string` verwalten ihre Elemente in einem kontinuierlichen Speicherblock. Dadurch ist wahlfreier Zugriff (random access) auf die Elemente möglich, bzw. die Iteratoren sind vom Typ `random_access_iterator`. Die Operationen zum einfügen und entfernen von Elementen in der Mitte sind teuer (O(n)), am Anfang und am Ende bieten sie gute Performance (O(1)), wenn nicht zuerst neuer Speicher angefordert werden muss.

[294] [Mey06b] Item 18
[295] [Mey06b] Item 13
[296] [Mey06b] Item 23
[297] [Mey06b] Item 24

17.5.3 Exception Sicherheit

Die Container der STL überprüfen keine logischen Fehler, wie Bereichsüberschreitung. Ausnahmen davon sind die Operationen `at(index)` von `std::vector` und `std::deque`. Diese werfen eine `std::out_of_range`. Die anderen Operationen sind nicht `noexept` deklariert. Dadurch kann eine bereichsprüfende Version zu debugging Zwecken eingesetzt werden.
Die Container selbst werfen nur `std::bad_alloc` wenn kein dynamischer Speicher mehr angefordert werden kann.
Die Exceptions, die nicht behandelt werden können und Exceptions, die von Container Elementen, den Benutzer definierten Typen, geworfen werden, werden an den Aufrufer weitergeleitet (und nicht "verschluckt"), das wird als **Exception neutral** bezeichnet[298].
Die Container garantieren, dass keine Memory leaks entstehen und dass die Invarianz der Container im Falle einer Exception erhalten bleibt. Das wird als **Basis Garantien** bezeichnet.

Atomare Operationen werden als **erweiterte Garantie** bezeichnet:

- array basierte Container
  - insert nur am Anfang und am Ende
  - insert in der Mitte nur wenn der Kopiekonstruktor und der Assignment Operator keine Exception wirft
- Transaktionssicherheit, commit or rollback, wird nur für bestimmte Operationen garantiert[299]
- bei Knoten basierten Containern (lists, multi-, set, map)
  - Knoten erzeugen
  - bei assoziativen Containern nur einzelne Insert, nicht für range based inserts
  - Knoten entfernen (`~Element()noexept`!)
- bei `std::list` alle Operationen außer
  - `remove(), remove_if(), merge(), sort(), unique()`

17.6 Iteratoren[300]

Iteratoren sind eine Abstraktion von Pointern, mit dem Zweck, das Iterieren über Mengen in Containern zu unterstützen. Sie bilden das Bindeglied zwischen Algorithmen und Containern.
Iterator ist ein Konzept: Alles was sich verhält wie ein Iterator, ist ein Iterator.

[298] siehe section 6.34.4 auf Seite 149
[299] siehe auch section 6.25.11 auf Seite 96
[300] siehe auch section ?? auf Seite ??

Alle Container definieren ihre Iteratoren, es werden keine zusätzlichen Header benötigt. Die Container definieren verschiedene Iteratortypen und stellen **const** und non-const Factory Methoden zur Verfügung. Daneben gibt es Container, die dieser Konvention nicht folgen, sie werden als non-standard Container bezeichnet.

Container und Iteratoren:

- Iterator Typen
  - `Container<ElementType>::iterator`
  - `Container<ElementType>::const_iterator`
  - `Container<ElementType>::reverse_iterator`
  - `Container<ElementType>::const_reverse_iterator`
- Factory Methoden
  - `begin(), end(): iterator, const_iterator`
  - `rbegin(), rend(): reverse_iterator, const_reverse_iterator`
- sie unterstützen das Iterieren über die Objekte im Container
- ein nativer Pointer kann als Iterator verwendet werden
- Iterator Klassen implmentieren teilweise das Pointer Interface
- das Interface wird bestimmt durch die Iterator Kategorie
  - `RandomAccessIterator` verhält sich wie ein Pointer und ist ein `BidirectionalIterator`
  - `BidirectionalIterator` kann nicht wahlfrei bewegt werden, sondern nur jeweils ein Element vor- oder rückwärts (`operater++/--()`) er ist ein `ForwardIterator`
  - `ForwardIterator` read-write repeatedly
  - `InputIterator` read only and only once
  - `OutputIterator` write only and may be only once
- Objekte von Klassen, die einer dieser Kategorien entsprechen, können eingeschränkt verwendet werden wie Pointer

Die Operation `begin()` liefert einen `iterator` der auf das erste Element zeigt.
Die Operation `end()` liefert einen `iterator` der hinter das letzte Element zeigt. Er darf nicht dereferenziert werden.
Die Operation `rbegin()` liefert einen `iterator` der auf das letzte Element zeigt.
Die Operation `rend()` liefert einen `iterator` der vor das erste Element zeigt. Er darf nicht dereferenziert werden[301].

Ist der Container **const**, liefern die Operationen einen `const_iterator`.

[301] siehe http://en.cppreference.com/w/cpp/iterator/reverse_iterator

17.6.1 Iterator Adapter

- Reverse Iteratoren
  - `rbegin(), rend()` der Container
  - bewegen sich rückwärts
  - lassen sich konvertieren Iterator <-> ReverseIterator
    Iter it = revIter.base()
    RevIter revIter(it)
- Insert Iteratoren oder Inserters
  - assignment is insertion
  - `back_insert_iterator, front_insert_iterator, insert_iterator`
  - Container müssen `push_back, push_front` unterstützen
- Stream Iteratoren
  - schreiben / lesen aus einem Stream
  - `ostream_iterator / istream_iterator`

Das Listing 279 zeigt eine Anwendung des `istream_iterators`. Dem Range Konstruktor von `list<..>` wird ein Iterator auf den Anfang der Datei und ein Iterator auf das Ende (`istream_iterator<int>()`) übergeben. Der default Konstruktor erzeugt einen Iterator der das Ende repräsentiert.

Ohne die inneren Klammern um `(istream_iterator<int>(dataFile))` wäre die Anweisung eine Funktionsdeklaration![302]

Listing 279: Anwendung des Stream Iterators

```
1 ifstream dataFile("ints.dat");
2 list<int> data( ( istream_iterator<int>(dataFile) ),
3          istream_iterator<int>() );
```

17.6.2 Iterator Konvertierung

Die verschiedenen Iterator Typen können nur bedingt in den jeweils anderen Konvertiert werden. Die Abbildung 17 auf der nächsten Seite zeigt die möglichen Konvertierungen und die dazugehörigen Funktionen (`base()`). In Listing 280 auf der nächsten Seite ist der dazugehörige C++ Code abgebildet.

Die Operationen `base()` der `reverse_iterator`en liefern einen Iterator, der auf ein Element nach dem `reverse_iterator` zeigt.
Die Initialisierung eines `reverse_iterator`s mit einem `iterator` erzeugt einen `reverse_iterator` der auf ein Element vor dem `iterator`, mit dem er initialsiert wurde, zeigt.

[302] [Mey06b] Item 6 Be alert for C++´s most vexing parse

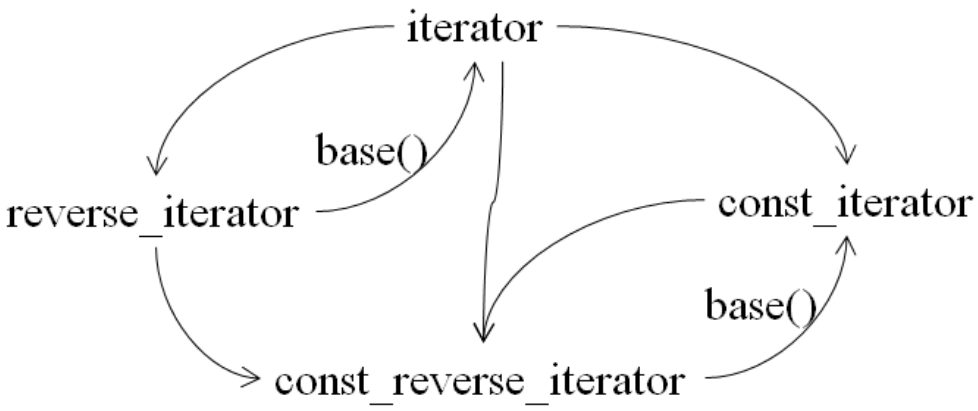

Abbildung 17: Iterator Konvertierungen

Listing 280: Iterator Konvertierungen

```
1  void demoIteratorKonvertierung(){
2     typedef vector<int> CollType;
3
4     typedef CollType::iterator iterator;
5     typedef CollType::const_iterator const_iterator;
6     typedef CollType::reverse_iterator reverse_iterator;
7     typedef CollType::const_reverse_iterator const_reverse_iterator;
8
9      CollType coll;
10    // einen iterator von der collection besorgen
11    iterator i = coll.begin();
12
13    // aus einem iterator einen reverse_iterator gewinnen
14    reverse_iterator ri(i); //Konvertierungskonstruktor
15    //reverse_iterator ri = i; // ist explicit
16    //ri = i; // assignment nicht möglich
17
18    // aus einem reverse_iterator einen iterator gewinnen
19    //iterator i_from_ri_implizit(ri); //Kein Konvertierungskonstruktor
20    //i = ri; // assignement nicht möglich
21    iterator i_from_ri_implizit = ri.base(); //CopyConstructor nicht explicit
22    i = ri.base(); // assignement
23
24    // aus einem iterator einen const_iterator gewinnen
25    const_iterator ci = i; // Konvertierungskonstruktor nicht explicit
26    ci = i; // assignement
27
28    // aus einem const_iterator einen const_reverse_iterator gewinnen
29    const_reverse_iterator cri(ci); //Konvertierungskonstruktor
30    //const_reverse_iterator cri_implizit = ci; // ist explicit
31    //cri = ci; // assignment nicht möglich
32
33    // aus einem const_reverse_iterator einen const_iterator gewinnen
34    // CopyConstructor nicht explicit
```

```
35    const_iterator ci_from_const_reverse = cri.base();
36    ci = cri.base(); // assignement
37
38    // aus einem iterator einen const_reverse_iterator gewinnen
39    const_reverse_iterator cri_from_iterator(i); //Konvertierungskonstruktor
40    //const_reverse_iterator cri_from_iterator = i; //ist explicit
41
42
43    //Konvertierungen die nicht direkt möglich sind
44    //iterator it_from_const(ci);
45    //iterator it_from_const(static_cast<iterator>(ci));
46    //i =  ci;
47    //i = static_cast<iterator>(ci);
48
49    /* das explizite Type Argument const_iterator ist notwendig, weil
50     distance nur mit einem Type Parameter ausgestattet ist
51     und dadurch eine implizite Umwandlung von i in const_iterator
52     erfolgt */
53    advance(i, distance<const_iterator>(i, ci) );
54 }
```

Das Listing 280 auf der vorherigen Seite:
Aliasnamen (`CollType`) für den Typ `vector<int>` und Aliasnamen für die Typen `CollType::iterator, CollType::const_iterator`, usw. werden definiert. Ein Objekt von `CollType coll;` wird angelegt.

Ein Objekt `iterator i = coll.begin();` wird mit dem Iterator, den `coll.begin()` liefert, initialisiert. Daraus wird ein `reverse_interator ri(i)` erzeugt und dieser wieder zurück konvertiert. Mit `i` wird ein `const_interator ci = i;` erzeugt.

Auf diese Weise werden alle möglichen Kombinationen der Iteratoren durch Zuweisung oder durch implzite und explizite Konstruktoren konvertiert.

Der Versuch einen non-const Iterator (letzter Block unten) mit dem const Iterator zu initialisieren scheitert. Der Grund ist, mit dem const Iterator können Objekte, die er referenziert, nicht verändert werden, mit dem non-const Iterator aber schon. Dieselbe Logik gilt auch für const und non-const Pointer[303]. Auch der Versuch, explizit die constness wegzucasten scheitert, wenn die Iteratoren keine nativen Pointer sind.

Die einzige Möglichkeit (letzt Zeile in Listing 280 auf der vorherigen Seite) einen const Iterator in einen non-const Iterator zu konvertieren ist, den non-const Iterator vor den const Iterator zu setzen und ihn solange zu bewegen, bis er dieselbe Position wie der const Iterator hat. Das wird am besten mit den Hilfsfunktionen für Iteratoren durchgeführt. Dem Funktions Template `distance<ConstIter>` muss explizit das Argument übergeben werden, weil die beiden Iteratoren verschiedene Typen haben. Ein non-const Iterator kann implizit in einen **`const`** Iterator konvertiert werden. Das Ergebnis von `distance` ist die Anzahl der Elemente die zwischen den beiden Iteratoren liegen. Mit `advance(..)` wird der non-const Iterator um diese Anzahl weitergeführt.

[303] siehe section 6.36.3 auf Seite 156

Wer nicht in Schwierigkeiten geraten möchte, sollte einen Mix verschiedener Iteratoren vermeiden.

17.6.3 Iterator Hilfsfunktionen

Die STL stellt zur Manipulation von Iteratoren die folgenden Hilfsfunktionen zur Verfügung:

- **void** `advance(iterator, n)` bewegt den Iterator um n Elemente vorwärts
- `distance(iter1, iter2)` liefert die Anzahl der Elemente zwischen den beiden Iteratoren
- `iter_swap(iter1, iter2)` tauscht die Werte der Iteratoren
- `next(Iterator, n=1)` (C++11)
  liefert einen Iterator der um ein oder n Elemente bewegt ist
- `prev` wie `next`

Das Listing 281 zeigt die Deklarationen der Funktionen[304]

Listing 281: Iterator Hilfsfunktionen

```
template< class InputIt >
typename std::iterator_traits<InputIt>::difference_type // return type
    distance( InputIt first, InputIt last );

template< class InputIt, class Distance >
void advance( InputIt& it, Distance n );

template< class ForwardIt1, class ForwardIt2 >
void iter_swap( ForwardIt1 a, ForwardIt2 b );

template< class ForwardIt >
ForwardIt next( ForwardIt it,
                typename std::iterator_traits<ForwardIt>::difference_type n = 1 );

template< class BidirIt >
BidirIt prev( BidirIt it,
              typename std::iterator_traits<BidirIt>::difference_type n = 1 );
```

17.6.4 Iterator Kategorien

Die Iterator Kategorien sind als `..._iterator_tag` in einer linearen Vererbungshierarchie als *flag* oder *label* Interfaces definiert. Jeder Iterator muss den Namen (z.B. via `typedef ... iterator_category`) definieren. Dieser Typ wird bei der generischen Programmierung zur Auswahl von zu dem Iterator am besten passenden

[304] z.B. http://en.cppreference.com/w/cpp/iterator/distance

Algorithmen verwendet. Die Auswahl trifft der Compiler und hat zur Laufzeit keinen Einfluss mehr. In Listing 282 sind die Iterator Kategorien abgebildet und das Listing 284 auf der nächsten Seite zeigt eine Anwendung davon[305].

Listing 282: Iterator Kategorien

```
namespace std{
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag : public input_iterator_tag { };
struct bidirectional_iterator_tag : public forward_iterator_tag { };
struct random_access_iterator_tag : public bidirectional_iterator_tag { };
}
```

17.6.5 Iterator Traits

Die `iterator_traits` in Listing 283 können als *type function* mit mehreren Ergebnistypen betrachtet werden. Sie werden auf einen Iterator Typ angewendet und liefern als Ergebnisse die verschiedenen Typen die im Zusammenhang des jeweiligen Iterators von Bedeutung sind. Das primary Template definiert die Typen auf der Basis des Iterators selbst. Die Spezialisierungen für Pointer und **const** Pointer definieren entsprechende Typen. Ein Beispiel für das non-invasive Ergänzen oder Erweitern eines bestehenden Typs durch Templates.

Listing 283: Iterator Traits

```
template<typename _Iterator>
struct iterator_traits
{
  typedef typename _Iterator::iterator_category iterator_category;
  typedef typename _Iterator::value_type        value_type;
  typedef typename _Iterator::difference_type   difference_type;
  typedef typename _Iterator::pointer           pointer;
  typedef typename _Iterator::reference         reference;
};
 /// Partial specialization for pointer types.
template<typename _Tp>
struct iterator_traits<_Tp*>
{
  typedef random_access_iterator_tag iterator_category;
  typedef _Tp                        value_type;
  typedef ptrdiff_t                  difference_type;
  typedef _Tp*                       pointer;
  typedef _Tp&                       reference;
};

/// Partial specialization for const pointer types.
template<typename _Tp>
struct iterator_traits<const _Tp*>
```

[305] http://en.cppreference.com/w/cpp/iterator/iterator_tags

```
24 {
25   typedef random_access_iterator_tag iterator_category;
26   typedef _Tp                        value_type;
27   typedef ptrdiff_t                  difference_type;
28   typedef const _Tp*                 pointer;
29   typedef const _Tp&                 reference;
30 };
```

17.6.6 Generische Programmierung mit Iterator Traits

Der Funktion `alg` in Listing 284 ist als Funktions Template realisiert und mehrfach überladen[306]. Die erste Überladung wird mit zwei Iteratoren, dem Range vom Anwender aufgerufen. Diese ermittelt mit Hilfe der `iterator_traits` die `iterator_category` und ruft die Funktion `alg` mit einem Objekt dieser Kategorie als drittem Argument auf. Je nach dem, von welcher Kategorie die Iteratoren sind, wird vom Compiler die entsprechende Überladung von `alg` ausgewählt. Der Vorteil, die `iterator_category` über die `iterator_traits` zu ermitteln und nicht die *member type definition* der Iteratoren direkt zur verwenden ist, es können auch native Pointer verwendet werden, die keine *member type definitions* für die `iterator_category` haben.
"Jedes Problem in der Software Entwicklung kann durch eine weitere Ebene der Indirektion gelöst werden" [307].
Auf diese Weise sind die Hilfsfunktionen für Iteratoren realisiert. Die Funktion `std::distance` berechnet für RandomAccess Iteratoren via Pointerarithmetik die neue Position, während für ForwardIteratoren eine Schleife notwendig ist, die den Iterator Schritt für Schritt an die gewünschte Position führt.
Das Beispiel in Listing 284 veranschaulicht eine einfache Form der generischen Programmierung. Die Auswahl des passenden Algorithmus erfolgt zur compile time, zur Laufzeit wird die Funktion, wenn sie `inline` ist, noch nicht einmal gerufen.

Listing 284: Auswahl eines Algorithmus mit der Iterator Kategorie

```
1  #include <iostream>
2  #include <vector>
3  #include <list>
4  #include <iterator>
5
6  template <class BIIter>
7  void alg(BIIter, BIIter, std::bidirectional_iterator_tag)
8  {
9      std::cout << "alg() called for bidirectional iterator" << std::endl;
10 }
11 template <class RAIter>
12 void alg(RAIter, RAIter, std::random_access_iterator_tag)
13 {
14     std::cout << "alg() called for random-access iterator" << std::endl;
15 }
```

[306] Beispiel von http://en.cppreference.com/w/cpp/iterator/iterator_tags
[307] http://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

```
16
17 template< class Iter >
18 void alg(Iter first, Iter last)
19 {
20     alg(first, last,
21         typename std::iterator_traits<Iter>::iterator_category());
22 }
23
24 int main()
25 {
26     std::vector<int> v;
27     alg(v.begin(), v.end());
28
29     std::list<int> l;
30     alg(l.begin(), l.end());
31
32 }
33 // output:
34 alg() called for random-access iterator
35 alg() called for bidirectional iterator
```

17.7 Smart Pointer

Smart Pointer sind eine Abstraktion von Pointern, mit dem Zweck, die Verwaltung von Resourcen zu unterstützen, die nach Gebrauch wieder freigegeben werden müssen[308], z.B. dynamisch auf dem Heap mit `new` erzeugte Objekte. Sie rufen automatisch `delete` oder eine Benutzer definierte Operation auf, die das Objekt zerstören und eventuell Speicher freigeben soll.

Ein Smart Pointer ist ein C++ Klassen Template, das syntatisch die Schnittstelle von Pointern nachahmt[309] und etwas mehr tut als ein “raw” Pointer.

- `T* operator->()` Elementauswahl
- `T& operator*()` Dereferenzierung
- in manchen Fällen `T& operator[index]()`

Objekte dieser Klassen können eingeschränkt verwendet werden wie Pointer. Als Elemente in Containern ist polymorphes Verhalten der Elemente möglich. Sie sind für alle Typen nützlich und werden daher mit dem Typ parametrisiert, den sie verwalten.

Die SmartPointer der STL sind im Header `<memory>` definiert

- `std::shared_ptr<..>`: ein counted Pointer, der bei der Erzeugung des Objekts initialisiert wird (RAII): `shared_ptr<Widget> p(new Widget)`. Weitere dürfen nur als Kopien aus dem Ersten erzeugt werden, der Letzte, der zerstört

[308] section 6.17 auf Seite 54
[309] section 6.25.12 auf Seite 97

wird, zerstört auch das Objekt. Dem Konstruktor kann als zweiten Parameter ein Objekt übergeben werden, das die Resource frei gibt, das default Argument führt `delete` aus.

- `std::weak_ptr<..>`: ein Partner des `shared_ptr` zur Auflösung und Vermeidung von zyklischen Abhängigkeiten
- `std::unique_ptr<..>`: der Highlander, es kann nur einen geben, der auf das Objekt zeigt. Kopieren ist nicht möglich, nur der Move Konstruktor und Move Assignment Operator stehen zur Verfügung. Der zweite template Parameter ist die Storage Policy[310].

Mit der Basisklasse `std::enable_shared_from_this<Widget>` wird die Erzeugung eines `std::shared_ptr` in der Methode der Klasse unter folgenden Voraussetzungen ermöglicht:

- Das Objekt existiert auf dem Heap
- Es wird bereits von einem `std::shared_ptr` verwaltet
- Der Pointer wird mit `shared_from_this()` erzeugt
- `shared_from_this()` kann nicht im Konstruktor verwendet werden

Listing 285: Beispiel enable shared from this

```
1  class Widget : public std::enable_shared_from_this<Widget>
2  {
3  ...
4     Widget(){
5        // Exception what():  bad_weak_ptr
6        // kann nicht im Ctor verwendet werden
7        std::shared_ptr<Widget> p(shared_from_this());
8     }
9     // darf nur auf Objekte die auf dem Heap erzeugt wurden
10    // und bereits von einem std::shared_ptr<Widget> verwaltet werden
11    // angewendet werden
12    std::shared_ptr<Widget> getShared(){
13       return shared_from_this();
14    }
15    std::shared_ptr<Widget> bad(){
16       return std::shared_ptr<Widget>(this);
17    }
18 };
```

Repository:Cpp-Basics/EnableSharedFromThis

Anstatt von der Basisklasse zu erben, kann dem Objekt ein `std::shared_ptr` bei Bedarf übergeben werden.

[310] im Sinne von [Ale09] Section 7.14.1 The Storage policy

17.7.1 raw Pointer v.s. SmartPointer

SmartPointer haben **value Semantik**, das gilt für einige "raw" Pointer nicht[311]. Value Semantik bedeutet, dem Objekt kann ein anderer Wert zugewiesen werden und das Objekt kann kopiert werden. Das gilt auch für raw Pointer die auf ein Element in einem Array zeigen. Sie können durch das Array bewegt werden und ihr Wert kann kopiert werden, um temporäre Ergebnisse zu speichern.

Das gilt für einen raw Pointer der mit
`Widget *p = new Widget;`
initialisiert wird, nicht. Er zeigt nicht nur auf das Objekt, sondern es "gehört" ihm auch, im Sinne, dass das Objekt über diesen Pointer wieder zerstört werden muss! `delete p;` Wird diesem einfach ein neuer Wert zugewiesen:
`Widget *p = nullptr;`
kann das Objekt nicht mehr erreicht werden, der Speicher ist verloren (resource leak). Wird der Pointer einem anderen Pointer zugewiesen, muss mit größter Sorgfalt damit umgegangen werden. Ein Objekt zweimal mit `delete` zu zerstören ist noch katastrophaler als ein resource leak.

17.8 Functors[312]

Functors sind eine Abstraktion von Funktionen, meistens mit dem Zweck, Funktionen ein "Gedächtnis", einen Zustand zu verleihen.

- vordefinierte Functors sind im Header `<functional>` definiert
- Functors sind Klassen, die den `operator()` überladen,
- Objekte dieser Klassen sind *smart functions*, *function objects* oder kurz *functors* sie können verwendet werden wie Funktionen:
  `obj.operator()(argList)` oder kurz
  `obj(argList)`
- haben ihren eigenen Typ
- Prädikate sind Functors mit dem Prototyp
  `bool operator(...)` wie sie von assoziativen Containern als Sortierkriterium und von Algorithmen verwendet werden, sie dürfen nicht zustandsbehaftet implementiert werden
- sind effizienter als Funktionen oder Funktionspointer weil sie `inline` implementiert werden können

17.8.1 Prädikate

Prädikate werden als Äquivalezkriterium in sortierten Containern verwendet und von Algortihmen als Vergleichskriterium. Das default Äquivalezkriterium ist der

[311] [Ale09] Chapter 7
[312] siehe auch section 6.25.14 auf Seite 99

kleiner Operator **<**, repräsentiert durch den Typ `less<T>`, das default Vergleichskriterium ist der Operator **==**.

Gleichheit vs. Äquivalenz

- Gleichheit mit dem Operator **==**
  - `o1 == o2`
  - Algorithmen wie `find, replace_copy,...`
- Äquivalenz mit dem Operator **<**
  - `!(o1 < o2)&& !(o2 < o1)`
  - Assozative Container und verschiedene Algorithmen wie `sort, binary_search,...`
- unäre oder binäre Functors
- `bool` als Rückgabe Typ
- *pure functions*[313]
  - dürfen nicht zustandsbehaftet sein
  - liefern für zwei Aufrufe `op(o1)` oder `op(o1, o2)` immer daselbe Ergebnis
- Sortierkriterien sind binary Prädikate, sie müssen immer die Logik des Operator **<** implementieren (*strict weak ordering*)[314]
  - antisymmetric
    - `x < y == true -> y < x == false`
    - `op(x, y)== true -> op(y, x)== false`
  - transitiv
    - `x < y && y < z -> x < z`
    - `op(x, y)&& op(y, z) -> op(x, z)`
  - irreflexive
    - `x < x == false`
    - `op(x, x)== false`

17.9 Algorithmen

Algorithmen sind Funktions Templates oder Operationen der Container, die die Iteration über einen Range einer Menge zu verschiedenen Zwecken abstrahieren. Sind gleichnamige Operationen in dem verwendeten Container vorhanden, ist diese der globalen Funktion vorzuziehen.

[313] [Mey06b] Item 39 und [Jos04] S. 302
[314] [Mey06b] Item 42

Der Name der Algorithmen gibt einen Hinweis auf den Zweck, wie das bei guten Namen üblich ist. Die meisten Algorithmen sind im Header `<algorithm`. Dieser Header beinhaltet auch einige Hilfsfunktionen wie `max` oder `iter_swap`. Numerische Algorithmen sind im `<numeric>` deklariert. Im Zusammenhang mit Algorithmen werden häufig Functors benötigt, diese sind im Header `<functional` deklariert.

Klassifikation der Algorithmen:

- Nicht modifizierende / nonmodifying
- verändernde / modifying, mutating
- entfernende / removing
- sortierende / sorting
- sorted-range
- Numerische / numeric

Die Liste der einzelnen Algorithmen ist zu lang um hier aufgeführt zu werden.

Die allgemeine Form eines Algorithmus ist
`algoName(begin, end, ...)` oder
`algoName(begin, end, destination, ...)`
Die weggelassenen Parameter (...) sind vom jeweiligen Algorithmus abhängig.

- alle Algorithmen arbeiten mit Ranges
- Ranges werden mit `[begin, end(` spezifiziert
  - halboffene Bereiche: inclusive begin, exclusiv end
  - begin und end sind Iteratoren
  - weitere Ranges (Destinations), werden meist nur mit begin spezifiziert, sie müssen auf ausreichend Platz zeigen oder durch einen Inserter spezifiziert werden
- Benutzer sind für die Gültigkeit der Ranges verantwortlich
  - beide gehören zum selben Container
  - begin ist nicht hinter end
  - ungültige Ranges bedingen undefiniertes Verhalten

17.9.1 Wann welchen Algorithmus einsetzen

Die Tabelle 12 auf der nächsten Seite zeigt die verschiedenen Aufgaben und welche Algorithmen oder Memberfunktionen der Container genutzt werden können[315].

[315] [Mey06b] Item 54

Tabelle 12: Wann welchen Algorithmus einsetzen?

	Algorithmus		Member Function	
Aufgabe	unsorted range	sorted range	set / map	multiset/ multimap
Existiert ein Wert?	find	binary_search	count	find
erstes Objekt mit Wert	find	equal_range	find	find or lower_bound
erstes Objekt mit Wert das kein Vorgänger ist oder das Objekt	find_if	lower_bound	lower_bound	lower_bound
erstes Objekt mit Wert	find_if	upper_bound	upper_bound	upper_bound
wieviel Objekte mit Wert	count	equal_range plus distance	count	count
alle Objekte mit Wert	find wiederholt	equal_range	equal_range	equal_range

17.10 Anwendungbeispiele Iteratoren

Listing 286: Function Template print

```
1  template<typename Element>
2  void printElement(Element const& e){ std::cout << e << ' ';}
3
4  template <typename Iterator>
5  void printValues(Iterator begin, Iterator end, char const * message = ""){
6     typedef typename std::iterator_traits<Iterator>::value_type value_type;
7
8     std::cout << message;
9     std::for_each(begin, end, printElement<value_type>);
10    std::cout << std::endl;
11 }
```

Die Funktion `printValues` in Listing 286 gibt einen Range von `[begin, end(` aus. Die Funktion `printElement` gibt einen einzelnen Wert aus. Sie werden in den folgenden Beispielen verwendet.

17.10.1 native Arrays und Iteratoren

Das Listing 287 auf der nächsten Seite zeigt die Verwendung von nativen Zeigern als Iteratoren. Dieses Beispiel soll die Standard Algorithmen wie z.B. `distance` und `for_each` expliziten Schleifen gegenüber stellen.

Listing 287: native Arrays und Iteratoren

```
void demoZeigerAlsIteratoren(){
    cout << "Zeiger als iteratoren" << endl;
    int coll[] = { 5, 4, 3, 2, 1 };
    int* end = coll+(sizeof(coll)/sizeof(int));

    cout << "distance( coll, end ): "
        << distance(coll, end) << endl;

    cout << endl << "Ausgabe mit expliziter Schleife: ";
    for(int* it = coll; it != end; ++it)
      cout << *it << ' ';
    cout << endl;

    printValues(coll, end, "Ausgabe mit Funktion: ");

    cout << endl << "Ausgabe mit std Algorithm: ";
    for_each(coll, end, printElement<int> );
    cout << endl;
}
// Ausgabe:
Zeiger als iteratoren
distance( coll, end ): 5

Ausgabe mit expliziter Schleife: 5 4 3 2 1

Ausgabe mit Funktion: 5 4 3 2 1

Ausgabe mit std Algorithm: 5 4 3 2 1
```

17.10.2 std::vector und Iteratoren: reverse_iterator

Das Listing 288 zeigt die Anwendung von `reverse_iterator`.

Listing 288: std::vector und Iteratoren und reverse_iterator

```
void demoVectorUndIteratoren(){
   cout << "vector und iteratoren" << endl;
    typedef std::vector<int> Collection;
   typedef Collection::iterator iterator;
   typedef Collection::reverse_iterator reverse_iterator;

   Collection coll;

   for(int i=0; i < 5; coll.push_back(++i))
      ;
    cout << "distance( coll.begin(), coll.end() ): "
        << distance(coll.begin(), coll.end()) << endl;

    cout << endl << "Ausgabe mit std Algorithm: ";
```

```
15     for_each(coll.rbegin(), coll.rend(), printElement<int>);
16     cout << endl;
17 }
18 // Ausgabe:
19 vector und iteratoren
20 distance( coll.begin(), coll.end() ): 5
21
22 Ausgabe mit std Algorithm: 5 4 3 2 1
```

17.10.3 std::list und Iteratoren: Inserter

Das Listing 289 zeigt die Anwendung der Iteratoradapter Inserter: `front_inserter`.

Listing 289: std::list und Iteratoren

```
1  void demoListUndIteratoren(){
2     cout << "list und iteratoren" << endl;
3      typedef std::list<int> Collection;
4     typedef Collection::iterator iterator;
5     typedef Collection::reverse_iterator reverse_iterator;
6
7     Collection coll;
8     std::front_insert_iterator<Collection>  inserter = std::front_inserter(coll);
9
10    for(int i=0; i < 5; inserter = ++i)
11       ;
12     cout << "distance( coll.begin(), coll.end() ): "
13         << distance(coll.begin(), coll.end()) << endl;
14
15     cout << endl << "Ausgabe mit std Algorithm: ";
16     for_each(coll.begin(), coll.end(), printElement<int>);
17     cout << endl;
18 }
19 // Ausgabe:
20 list und iteratoren
21 distance( coll.begin(), coll.end() ): 5
22
23 Ausgabe mit std Algorithm: 1 2 3 4 5
```

17.11 Anwendungsbeispiele Algorithmen

17.11.1 Filtern und Sortieren mit lower_bound und upper_bound

Das Beispiel in Listing 290 auf der nächsten Seite zeigt die Verwendung und Kombination von `sort`, `vector::erase`, `lower_bound`, `upper_bound` [316].

[316] [Mey06b] Item 45

Listing 290: lower_bound upper_bound

```
void demoAlgo(){
   cout << "demoAlgo" << endl;
   vector<double> v;
   double a[] = { 7, 1, 4.1, 4.9, 2, 5, 5.1, 5.9};
   const size_t N = sizeof(a)/sizeof(double);
   printValues(a, a+N, "Array: ");
   double limit = 5;
   cout << "limit: " << limit << endl;

   copy(a, a+N, back_inserter(v));

   printValues(v.begin(), v.end(), "v unsortiert: ");
   sort(v.begin(), v.end());
   printValues(v.begin(), v.end(), "v   sortiert: ");

   //copy
   vector<double> c(v);

   //entferne alle Elemente die kleiner sind als limit exclusive
   v.erase(v.begin(),lower_bound(v.begin(),v.end(),limit));
   printValues(v.begin(), v.end(), "v   erased lower_bound: ");

   v.clear(); copy(c.begin(), c.end(), back_inserter(v));

   //entferne alle Elemente die kleiner sind als limit inclusive
   v.erase(v.begin(), upper_bound(v.begin(),v.end(),limit));
   printValues(v.begin(), v.end(), "v   erased upper_bound: ");

   cout << endl;
   v.clear(); copy(c.begin(), c.end(), back_inserter(v));

   //entferne alle Elemente die grösser sind als limit exclusive
   v.erase(lower_bound(v.begin(),v.end(),limit), v.end());
   printValues(v.begin(), v.end(), "v   erased lower_bound: ");

   v.clear(); copy(c.begin(), c.end(), back_inserter(v));

   //entferne alle Elemente die grösser sind als limit inclusive
   v.erase(upper_bound(v.begin(),v.end(),limit), v.end());
   printValues(v.begin(), v.end(), "v   erased upper_bound: ");
}
// Ausgabe:
demoAlgo
Array: 7 1 4.1 4.9 2 5 5.1 5.9
limit: 5
v unsortiert: 7 1 4.1 4.9 2 5 5.1 5.9
v   sortiert: 1 2 4.1 4.9 5 5.1 5.9 7
v   erased lower_bound: 5 5.1 5.9 7
v   erased upper_bound: 5.1 5.9 7

```

```
v   erased lower_bound: 1 2 4.1 4.9
v   erased upper_bound: 1 2 4.1 4.9 5
```

17.11.2 Filtern und Sortieren mit equal_range

Das Beispiel in Listing 291 zeigt die Verwendung und Kombination von `equal_range` und der Initializer List (ab C++11).

Listing 291: equal_range

```
void demoEqualRange(){
   cout << "demoEqualRange" << endl;
   typedef vector<double> Container;
   typedef Container::iterator iterator;
   typedef pair<iterator, iterator> Pair;

   Container v{ 7, 1, 4.1, 4.9, 2, 5, 5, 5.1, 5.9};
   double limit = 5;
   cout << "limit: " << limit << endl;

   printValues(v.begin(), v.end(), "v unsortiert: ");
   sort(v.begin(), v.end());
   printValues(v.begin(), v.end(), "v   sortiert: ");

   Pair p = equal_range(v.begin(), v.end(), limit);

   printValues(p.first, p.second, "equal_range: ");

   cout << "Anzahl: " << distance(p.first, p.second) << endl;
}
//Ausgabe
demoEqualRange
limit: 5
v unsortiert: 7 1 4.1 4.9 2 5 5 5.1 5.9
v   sortiert: 1 2 4.1 4.9 5 5 5.1 5.9 7
equal_range: 5 5
Anzahl: 2
```

Teil VII

Übungen

18 Übungen zu C++

18.1 Übung Lebenszyklus von Objekten

Die folgenden Übungen sind dazu gedacht, den Lebenszyklus von Objekten und die Compiler synthetisierten Methoden beobachten und verstehen zu können.

18.1.1 Operationen die der Compiler zur Verfügung stellt

Geben Sie in jeder Methode die Sie in den Übungen erstellen die Signatur der Methode aus wie in Listing 292.

Listing 292: Beispiel Ausgabe der Signatur

```
1 class A{
2    void operation(int i){
3       cout << "void A::operation(int " << i <<")" << endl;
4    }
5 };
```

Führen Sie jeden Schritt der folgenden Übungen einzeln aus und kommentieren Sie gegebenenfalls die vorherigen Codeteile aus, um einen übersichtlichere Ausgabe zu erhalten.

- Geben Sie nach jeder Veränderung der Variablen ihren Wert aus
- Erstellen Sie eine Klasse A mit einem Member int i
  - Erzeugen Sie ein Objekt `A a1` und weisen dem Member einen Wert zu `a1.i = 42;`; Ausgabe `cout << "a1.i:␣"<< a1.i << endl;`
  - Erzeugen Sie zwei weitere Objekte aus dem ersten Objekt `A a2(a1); A a3 = a1;` Ausgabe `a2.i` und `a3.i`
  - Weisen Sie dem Member i in a1 einen anderen Wert zu
  - Erzeugen Sie ein weiteres Objekt und weisen Sie diesem das erste Objekt zu
    `A a4; a4 = a1;`

Wie ist das alles möglich? Der Compiler erzeugt unter bestimmten Umständen die notwendigen Methoden für die Klassen. Die genauen Regeln sind in section 6.5 auf Seite 38 beschrieben.

18.1.2 Benutzer definierte Operationen die der Compiler zur Verfügung stellt

Werden für diese Operationen durch den Benutzer Methoden zur Verfügung gestellt, kann das Verhalten beeinflusst werden.

Beginnen wir noch einmal von vorne! Kommentieren Sie alles aus und nach und nach wieder ein! Führen Sie wieder jeden Schritt einzeln aus! Beobachten Sie was passiert, welche Fehlermeldungen werden ausgegeben und warum?

- Stellen Sie in der Klasse `A` einen Default Konstruktor mit Ausgabe zur Verfügung
- Kommentieren sie den Default Konstruktor aus und stellen einen Konstruktor der einen int als Parameter erwartet zur Verfügung und initialisieren Sie den Member `i` mit dem Argument. Ausgabe wie in Listing 292 auf der vorherigen Seite
- Stellen Sie einen Destruktor mit Ausgabe zur Verfügung
- Machen Sie den Member **`private`**
- Stellen Sie einen getter (`int getI()`) für den Member zur Verfügung und geben Sie den Wert mit Hilfe des Getters aus
- Stellen Sie einen Kopiekonstruktor `A(A const& src)` und einen Assignment Operator `A& operator=(A const& src)` mit Ausgabe zur Verfügung
- Stellen Sie einen Move Konstruktor `A(A && src)` und einen Move Assignment Operator `A& operator=(A && src)` mit Ausgabe zur Verfügung
- Deklarieren Sie den Copy-Konstruktor `explicit`
- Deklarieren Sie den Move-Konstruktor `explicit`

18.1.3 Temporäre Objekte bei Funktionsaufrufen

- Schreiben sie eine überladenen Funktion wie in Listing 293 und rufen Sie diese mit `f(a1)` und mit `f(A())` auf
- Ändern Sie die Signatur in `void f(A const& a)`
- Ändern sie den Getter `int getI()const`

Listing 293: Übung Temporäre Objekte

```
1 void f(A a){
2    cout << "f(A a) a.i: " << a.getI() << endl;
3 }
4 void f(A && a){
5    cout << "f(A && a) a.i: " << a.getI() << endl;
6 }
```

18.1.4 Lebenszyklus leere Klasse

Die Übungen in den folgenden Kapitel bestehen aus einem übersetzbaren Codefragment und der Frage „Was ist die Ausgabe des Programms?“ In `main()` wird jeweils die Funktion `lebenszyklus()` aufgerufen.

Die Beispiele zeigen Syntax, Deklaration und Implementierung der verschiedenen Operationen und Methoden inclusive der in C++11 hinzugekommenen.

Die Puzzles sind mit der gnu version g++ 4.7.2 und der Option -std=c++11 übersetzt. So weit die neuen Sprachfeatures zur Verfügung waren, wurden diese verwendet. z.B. zur Initialisierung von Variablen wurde die einheitliche Initialisierung `int i{}` verwendet.

Listing 294: Eine einfache Klasse A (A1Empty.h)

```
#ifndef A1EMPTY_H_
#define A1EMPTY_H_

class A
{
public:
   int i = 42;
};
#endif /* A1EMPTY_H_ */
```

Die Funktion `lebenszyklus()` in Listing 295 lässt sich mit der Definition der Klasse `A` aus Listing 294 übersetzen und ausführen. Welche Ausgabe erzeugt dieses Programm?

Listing 295: Lebenszyklus von Objekten auf dem Stack

```
// Operationen die der Compiler zur Verfügung stellt //
#include <iostream>
#include <utility>

#include "A1Empty.h"
//#include "A2.h"
//#include "A3MemberBC.h"
//#include "A4BaseBC.h"
//#include "A5MemberbBaseC.h"

using namespace std;

void f(A&& a){ cout << "void f(A&& a)" << endl; }
void f(A& a){ cout << "void f(A& a)" << endl; }
void f(A const& a){ cout << "void f(A const& a)" << endl; }

void lebenszyklus()
{
   cout << "=== begin Lebenszyklus()" << endl;
   {
      cout << "=== begin Block \n{" << endl;
```

```
22
23      cout << "=== A a1;" << endl;
24      A a1;
25      cout << "a1.i: " << a1.i << endl;
26
27      cout << "=== A a2 = a1;" << endl;
28      A a2 = a1;
29      cout << "a2.i: " << a2.i << endl;
30
31      cout << "=== A a3(a1);" << endl;
32      A a3(a2);
33      cout << "a3.i: " << a3.i << endl;
34
35      a1.i = 43;
36      cout << "=== a1.i = 43 " << a1.i << endl;
37
38      cout << "=== a2 = a1" << endl;
39      a2 = a1;
40      cout << "a2.i: " << a2.i << endl;
41
42      cout << "=== A a4(move(a1));" << endl;
43      A a4(move(a1));
44      cout << "a1.i: " << a1.i << endl;
45      cout << "a4.i: " << a4.i << endl;
46
47      cout << "a4 = move(a3);" << endl;
48      a4 = move(a3);
49      cout << "a3.i: " << a3.i << endl;
50      cout << "a4.i: " << a4.i << endl;
51
52      cout << "=== A a5 = f();" << endl;
53      A a5 = f();
54      cout << "a5.i: " << a5.i << endl;
55
56      cout << "=== f(A());" << endl;
57      f(A());
58
59      cout << "=== f(a4);" << endl;
60      f(a4);
61
62      cout << "=== end Block \n}" << endl;
63   }
64   cout << "=== end Lebenszyklus()" << endl;
65 }
```

Die Ausgabe ist in Listing 303 auf Seite 346 dargestellt.

18.1.5 Lebenszyklus Klasse mit Benutzer definierten Operationen

Welche Ausgabe erhält man, wenn die Definition der Klasse A aus Listing 296 verwendet wird?

Listing 296: A mit benutzerdefinierten Operationen (A2.h)

```
#ifndef A2_H_
#define A2_H_

class A
{
public:
   A() { std::cout << "A::A()" << std::endl; }

   //explicit
   A(A const& src): i(src.i) { std::cout << "A::A(A const& src)" << std::endl; }

   //explicit
   A(A && src): i(src.i) {
      src.i = 0;
      std::cout << "A::A(A && src)" << std::endl;
   }

   ~A() { std::cout << "A::~A()" << std::endl; }

   A& operator=(A const& src){
      i = src.i;
      std::cout << "A& A::operator=(A const& src)" << std::endl;
      return *this;
   }
   A& operator=(A && src){
      i = src.i;
      src.i = 0;
      std::cout << "A& A::operator=(A && src)" << std::endl;
      return *this;
   }
   int i = 42;
};
#endif /* A2_H_ */
```

Die Ausgabe ist in section 18.6.2 auf Seite 347 dargestellt.

18.1.6 Lebenszyklus Klasse A mit Klasse B und C als Member

Welche Ausgabe erhält man, wenn die Definition der Klasse A aus Listing 297 auf der nächsten Seite verwendet wird?

Listing 297: A mit Klasse B und C als Member (A3MemberBC.h)

```
#ifndef A3MemberBC_H_
#define A3MemberBC_H_

#include "C.h"
#include "B.h"

class A
{
public:
   A() : c{}, b{} { std::cout << "A::A()" << std::endl; }

   //explicit
   A(A const& src) : c{src.c}, b{src.b}, i{src.i}
      { std::cout << "A::A(A const& src)" << std::endl; }

   //explicit
   A(A && src) : c(std::move(src.c)), b(std::move(src.b)), i{src.i}
   {
      src.i = 0;
      std::cout << "A::A(A && src)" << std::endl;
   }

   ~A() { std::cout << "A::~A()" << std::endl; }

   A& operator=(A const& src){
      c = src.c;
      b = src.b;
      i = src.i;
      std::cout << "A& A::operator=(A const& src)" << std::endl;
      return *this;
   }
   A& operator=(A && src){
      c = std::move(src.c);
      b = std::move(src.b);
      i = src.i;
      src.i = 0;
      std::cout << "A& A::operator=(A && src)" << std::endl;
      return *this;
   }
   C c;
   B b;
   int i = 42;
};

#endif /* A3MemberBC_H_ */
```

Die Ausgabe ist in section 18.6.3 auf Seite 349 dargestellt.

18.1.7 Lebenszyklus Klasse A mit Klasse B und C als Basisklassen

Welche Ausgabe erhält man, wenn die Definition der Klasse A aus Listing 298 verwendet wird?

Listing 298: A mit Klasse B und C als Basisklassen (A4BaseBC.h)

```
#ifndef A4_BASE_BC_H_
#define A4_BASE_BC_H_

#include "C.h"
#include "B.h"

class A : public C, public B
{
public:
   A() { std::cout << "A::A()" << std::endl; }

   //explicit
   A(A const& src) : C{src}, B{src}, i{src.i}
      { std::cout << "A::A(A const& src)" << std::endl; }

   //explicit
   A(A && src) : C(std::move(src)), B(std::move(src)), i{src.i}
   {
      src.i = 0;
      std::cout << "A::A(A && src)" << std::endl;
   }

   ~A() { std::cout << "A::~A()" << std::endl; }

   A& operator=(A const& src){
      C::operator =(src);
      B::operator =(src);
      i = src.i;
      std::cout << "A& A::operator=(A const& src)" << std::endl;
      return *this;
   }
   A& operator=(A && src){
      C::operator =(std::move(src));
      B::operator =(std::move(src));
      i = src.i;
      src.i = 0;
      std::cout << "A& A::operator=(A && src)" << std::endl;
      return *this;
   }
   int i = 42;
};

#endif /* A4_BASE_BC_H_ */
```

Die Ausgabe ist in section 18.6.3 auf Seite 349 dargestellt.

18.1.8 Lebenszyklus Klasse A mit Klasse B als Member und C als Basisklasse

Welche Ausgabe erhält man, wenn die Definition der Klasse A aus Listing 299 verwendet wird?

Listing 299: A mit Klasse B als Member und C als Basisklasse (A5MemberbBaseC)

```
1  #ifndef A5_MEMBER_B_BASE_C_H_
2  #define A5_MEMBER_B_BASE_C_H_
3
4  #include "C.h"
5  #include "B.h"
6
7  class A : public C
8  {
9  public:
10    A() : C{} { std::cout << "A::A()" << std::endl; }
11
12    //explicit
13    A(A const& src) : C{src}, b{src.b}, i{src.i}
14       { std::cout << "A::A(A const& src)" << std::endl; }
15
16    //explicit
17    A(A && src) : C(std::move(src)), b(std::move(src.b)), i{src.i}
18    {
19       src.i = 0;
20       std::cout << "A::A(A && src)" << std::endl;
21    }
22
23    ~A() { std::cout << "A::~A()" << std::endl; }
24
25    A& operator=(A const& src){
26       C::operator =(src);
27       b = src.b;
28       i = src.i;
29       std::cout << "A& A::operator=(A const& src)" << std::endl;
30       return *this;
31    }
32    A& operator=(A && src){
33       C::operator =(std::move(src));
34       b = std::move(src.b);
35       i = src.i;
36       src.i = 0;
37       std::cout << "A& A::operator=(A && src)" << std::endl;
38       return *this;
39    }
40    B b = B();
41    int i = 42;
42 };
43 #endif /* A5_MEMBER_B_BASE_C_H_ */
```

Die Ausgabe ist in section 18.6.3 auf Seite 350 dargestellt.

18.2 Übung RAII und SmartPointer

Ziel der Übung ist die Anwendung der verschiedenen Techniken

- RAII und SmartPointer anwenden
- Copy- und Move- Operationen implementieren
- Überladen von Operatoren (Elementauswahl, Assignment, Increment, Decrement)
- Exceptions in Konstruktoren bearbeiten
- Templates und Membertemplates zur impliziten Konvertierung
- `friend` Deklarationen in Templates
- Variadic Templates

am Beispiel eines SmartPointers, der benutzt werden kann, wie ein nativer Pointer.

Schreiben Sie eine Funktion wie in Listing 39 auf Seite 54. Ersetzen Sie `Widget` durch die Klasse `A` aus Listing 296 auf Seite 334. Lösen Sie in `bar(p)` eine Exception aus.
Wird der Destruktor von `A` nach der Exception gerufen?

18.2.1 RAII

Schreiben Sie eine Klasse, die einen Konstruktor `SmartPointer(A* pA)` hat, der sich das übergebene Objekt, das auf dem Heap erzeugt sein muss, merkt und im Destruktor wieder zerstört. Erstellen Sie alle Methoden inline im Header von `SmartPointer.h`. Ändern sie die Funktion wie in Listing 40 auf Seite 55 und verwenden Sie Ihren SmartPointer anstelle `std::unique_ptr<T>`.
Wird der Destruktor von `A` nach der Exception gerufen?

18.2.2 SmartPointer

Erweitern Sie die Klasse `A` mit einer Operation `void op();` und versuchen Sie die Operation über den SmartPointer aufzurufen `pD->op();` oder `(*pD).op();`! Überladen Sie dazu die Elementauswahl Operatoren aus section 6.25.12 auf Seite 97.

18.2.3 Copy und Move Operationen

Ermöglichen Sie die Initialisierung eines neuen SmartPointers mit einem vorhandenen: `SmartPointer p2(p1);` oder `SmartPointer p2 = p1;`

und die Zuweisung `p2 = p1;`! Damit kann ein SmartPointer aus einer Funktion zurückgegeben werden oder by value einer Funktion als Argument übergeben werden.
Wie muss der Kopiekonstruktor und der Kopieassignment Operator des Smartpointers gestaltet werden?

18.2.4 Exceptions in Konstruktor Initialisierungslisten

Es gibt verschiedene Strategien mehrere Smartpointer Objekte für eine Resource zu verwalten. Die einfachste ist die Erzeugung eines Counters der im Konstruktor des Smartpointers auf dem Heap erzeugt wird. Dabei kann eine `bad_alloc` Exception ausgelöst werden.

Fangen Sie die Exception auf wie in Listing 129 auf Seite 145 und geben Sie im **`catch`** Block die Resource wieder frei.

18.2.5 Increment, Decrement Operatoren

Erstellen Sie eine Klasse `Counter` und überladen Sie den Increment und den decrement operator und verwenden Sie die Klasse als Counter. Werfen Sie eine Exception im Konstruktor des `Counter`s und beobachten Sie das Verhalten Ihres Smartpointers.

18.2.6 Template

Wandeln Sie die Klasse SmartPointer in ein Template um, so dass jeder beliebige Typ von der Klasse verwaltet werden kann.
Welche Änderungen müssen durchgeführt werden?

18.2.7 Template Memberfunktionen

Ermöglichen Sie mit dem `SmartPointer<T>` einen impliziten upcast wie er in Listing 95 auf Seite 104 gezeigt ist. Mit `SmartPointer<Derived> pD(new Derived)` sollte sowohl die Initialisierung `SmartPointer<Base> pB = pD;` als auch die Zuweisung `pB = pD` möglich sein.
Realisieren Sie das mit einem entsprechenden Konstruktur und einem Assignment Operator die als Membertemplate Funktionen ausgelegt sind wie in section 10.7 auf Seite 201 beschrieben.

18.2.8 Resourceleaks und dangling Pointers

Wie können trotz der Anwendung des Prinzips RAII, resource leaks und dangling pointers entstehen? Worauf ist bei der Anwendung des SmartPointers zu achten?

18.2.9 perfect forward und Variadic Templates

Schreiben Sie eine Funktion
`SmartPointer<T> makeSmartPointer<T>(args);`, die ein `T` auf dem Heap anlegt und einen `SmartPointer<T>` darauf zurückliefert. Die Typen der Argumente sollten exakt erhalten bleiben und an den Konstruktor von T weitergegeben werden, so dass gegebenenfalls der Move Konstruktor verwendet werden kann. Testen Sie das indem Sie als Argument ein temporäres Objekt an `makeSmartPointer<A>(createA())` übergeben!

18.2.10 Movesemantik

Stellen Sie einen SmartPointer zur Verfügung, der nur einen Besitzer ermöglicht. Dafür wird das Feature *Movesemantik* von C++11 benötigt.

18.3 UndoRedoFramework

Ziel der Übung ist die Anwendung der verschiedenen Techniken

- Design und Realisierung eines Frameworks (UndoRedoFramework)
- Implementieren vorgegebener Interfaces
- R- und L-Value Referencen unterscheiden und anwenden
- Reference qualified Operations, copy und move Semantik
- Test Design (Template Method, Factory Method) mit C++ Templates
- Test Driven Development / Test First
- Template techniken

18.3.1 Design eines UndoRedoFrameworks

Überlegungen zu einem Framework:

- Wie sollte die Schnittstelle eines UndoRedoManagers aussehen?
- Welche Hooks sollte das Framework zur Verfügung stellen?
- Wie muss mit Exceptions umgegangen werden?
- Welche Tests werden benötigt?
- Copy oder Move?

Implementieren Sie einen ersten Prototyp auf der Basis eines einfachen Calculators.

18.3.2 Schnittstellen und Tests

- Installieren Sie das Plugin CUTE
- clonen sie das Repository Cpp-UndoRedoFramework v0.9: https://github.com/GerdHirsch/Cpp-UndoRedoFramework/releases

In dem Repository befinden sich 5 Projekte in 3 Verzeichnissen,

- das UndoRedoFramework, mit den Interfaces und den referenz Implementierungen
- im Verzeichnis Test ein UndoRedoTestCute Projekt mit den Tests auf der Basis der Interfaces und
- ein UndoRedoDefaultTest Projekt, das die referenz Implementierungen testet
- im Verzeichnis UndoRedoCustomized das Projekt UndoRedoCustomized, das eine Klasse UndoRedoStackImpl mit leeren Methoden definiert und
- ein UndoRedoCustomizedTest Projekt, das die Implementierung des Customized Projekts testet
- in einem weiteren Repository Cpp-UndoRedoDemo, ein Projekt, das die Anwendung des Frameworks skizziert

Für das UndoRedoTestCute Projekt benötigen Sie das Testframework „CUTE Cplusplus Unit Test Easy“ das Sie unter http://cute-test.com/projects/cute/wiki/CUTE_Installation_and_System_Requirements finden. Um auf die URLs zugreifen zu können, muss in Eclipse gegebenenfalls ein Proxy eingetragen werden.

Genaueres ist im Wiki des Repositories beschrieben.

18.3.3 Test Design

Wie sollte ein Test gestaltet werden?
Der Test sollte auf einfache Weise jede Implementierung des Interfaces testen können. Das Diagramm 34 auf der nächsten Seite zeigt die Klassen mit den Testmethoden und ihre Spezialisierungen, die die Implementierungen über die Factory Method `createSUT():unique_ptr<..>` zur Verfügung stellen.

Das Diagramm 35 auf der nächsten Seite zeigt die Basisklasse `UndoRedoTest` für alle Tests. Sie stellt die Infrastruktur, die von allen Tests benötigt wird zur Verfügung.

18.3.4 Implementierung erstellen

- Legen Sie ein C++ Projekt für eine static Library an (z.B. MyUndoRedoImplementation)

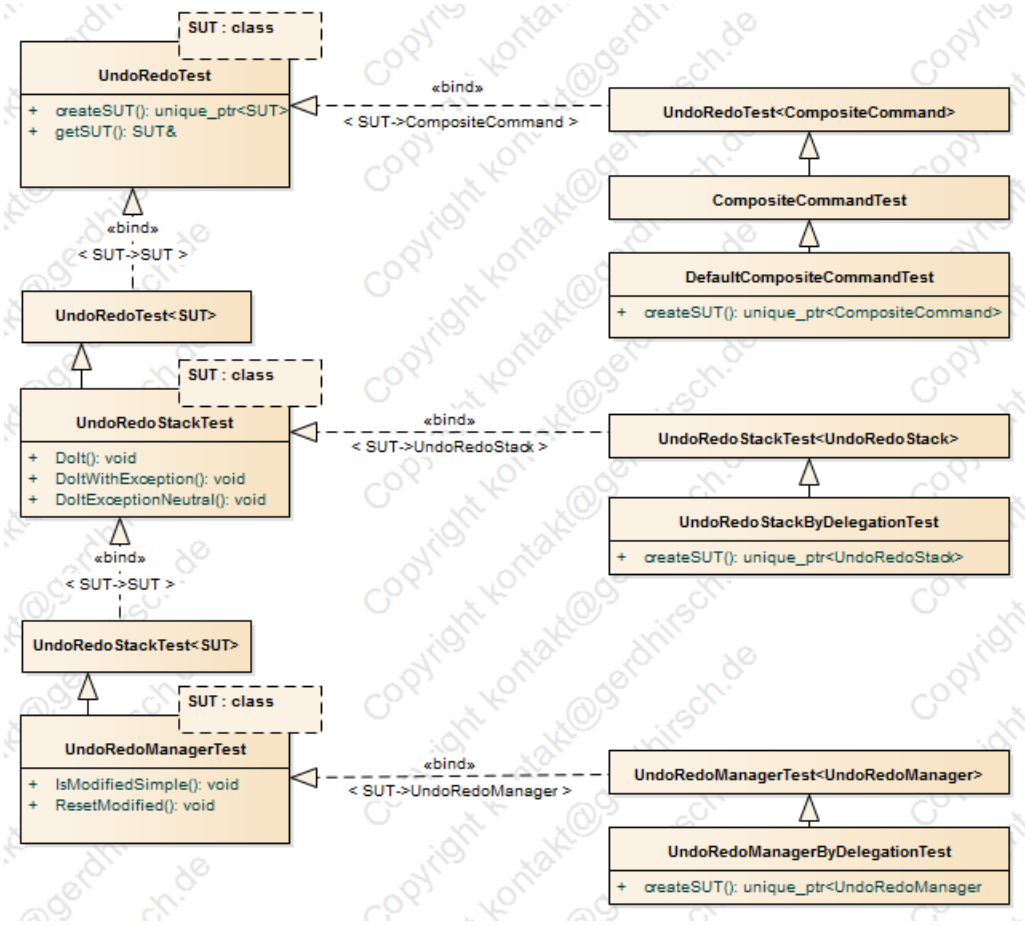

Diagramm 34: Das Test Design gemäß dem Template Method Pattern

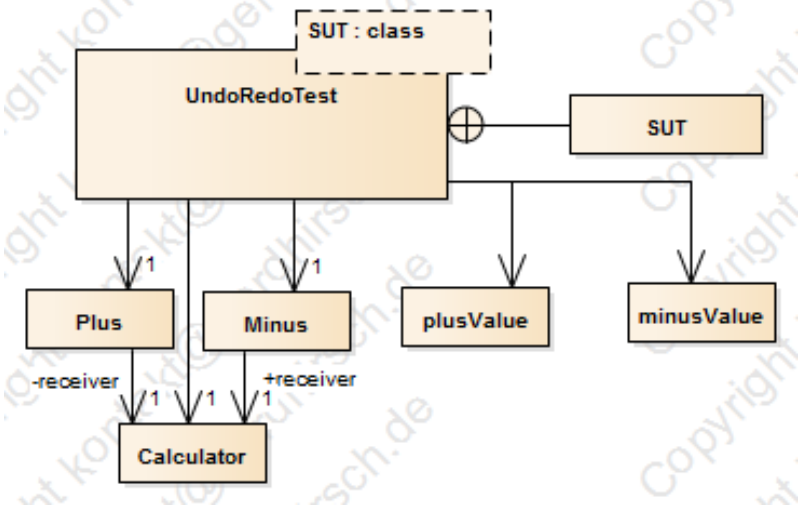

Diagramm 35: Die Infrastruktur für alle Tests

- Fügen Sie in den include Pfad, das Verzeichnis UndoRedoFramework/Include ein
- Implementieren Sie die Interfaces in `UndoRedoStack.h` `UndoRedoManager.h` und `CompositeCommand.h` zuerst leer
- Integrieren Sie Ihre Lib in des Testprojekt UndoRedoTestCute indem Sie für

Ihre Implementierungen einen Test auf der Basis der entsprechenden Tests des Frameworks erstellen und Ihre Tests in der Datei Test.cpp einfügen

- implementieren Sie die Methoden nach und nach bis alle Tests erfolgreich abgeschlossen werden

Arbeiten Sie in kleinen Schritten, so dass Sie immer ein übersetzbares Projekt haben.

18.4 Übung Sichtbarkeit und Lebensdauer

Die folgenden Übungen basieren auf der Anwendung aus section 6.21 auf Seite 62. Experimentieren Sie mit den auskommentierten Zeilen, kommentieren sie diese ein und beobachten die Effekte, wie z.B. Fehlermeldungen des Compilers, des Linkers oder Laufzeitfehler und die Auswirkung auf die Reihenfolge der Initialisierung der Objekte.

Beispiele:

- `gC` in allen Übersetzungseinheiten `extern` deklarieren
- Zeile 21 in Listing 52 auf Seite 65 einkommentieren
- **`static`** vor allen Variblen `mgC` entfernen

18.5 Basic Puzzles

Die Übungen in diesem Kapitel bestehen aus einem übersetzbaren Codefragment und der Frage „Was ist die Ausgabe des Programms?“ In `main()` wird jeweils die `aufgabe()` aufgerufen.

Die Puzzles sind mit der gnu version g++ 4.7.2 und der Option -std=c++11 übersetzt. So weit die neuen Sprachfeatures zur Verfügung waren, wurden diese Verwendet. z.B. zur Initialisierung von Variablen wurde die einheitliche Initialisierung **`int`** `i{}` verwendet.

18.5.1 Increment Pre- und Postfix Operator

Listing 300: Increment Pre- und Postfix Operator

```
1 void aufg0()
2 {
3    cout << "aufg0" << endl;
4
5    int i{}; // = 0;
6
7    cout << "(++i)++ : " << (++i)++ << endl;
8    cout << "i:" << i << endl;
9 }
```

```
10 // Welche Ausgabe erzeugt dieses Programm? //
```

Entfernen Sie die Klammern aus dem Ausdruck `(++i)++`. Welche Änderung können sie beobachten? Lösung? section 18.6.4 auf Seite 350

18.5.2 Call by Value

Listing 301: Call by Value

```
1  void f(int i)
2  {
3     cout << i << endl;
4     if (i) // i != 0
5        f(i-1);
6     cout << i << endl;
7  }
8
9  void aufg0a()
10 {
11    f(2);
12 }
13 // Welche Ausgabe erzeugt dieses Programm ? //
```

Lösung? section 18.6.5 auf Seite 350

18.5.3 Call by Reference vs. by Value

Listing 302: Call by Reference vs. by Value

```
1  namespace{  // anonymer namespace
2  // 1) void f(int i)
3  // 2)
4  void f(int& i)
5  {
6     cout << --i << " ";
7     if (i) // i != 0
8        f(i);
9     cout << i-- << " ";
10 }
11 }//namespace
12
13 void aufg0b()
14 {
15    int i(3); // initialisierung int i = 3;
16    cout << "i:" << i << endl;
17    f(i);
18    cout << endl << "i:" << i << endl;
19 }
20 // Welche Ausgabe erzeugt dieses Programm für 1) und 2)?
```

Lösung? section 18.6.6 auf Seite 350

18.6 Lösungen Basic Puzzles

18.6.1 Lebenszyklus leere Klasse

Lösung zu section 18.1.4 auf Seite 332

Listing 303: Ausgabe mit der leeren Klasse A

```
=== begin Lebenszyklus()
=== begin Block
{
=== A a1;
a1.i: 42
=== A a2 = a1;
a2.i: 42
=== A a3(a1);
a3.i: 42
=== a1.i = 43 43
=== a2 = a1
a2.i: 43
=== A a4(move(a1));
a1.i: 43
a4.i: 43
a4 = move(a3);
a3.i: 42
a4.i: 42
=== A a5 = f();
a5.i: 42
=== f(A());
void f(A&& a)
=== f(a4);
void f(A& a)
=== end Block
}
=== end Lebenszyklus()
```

Der Compiler erzeut alle benötigten Operationen.

- Default Konstruktor
- Copy Konstruktor
- Copy Assignment Operator
- Move Konstruktor
- Move Assignment Operator
- Destruktor

Die Kopie- und Move- Operationen arbeiten Bitweise, der Speicherinhalt wird kopiert.

18.6.2 Lebenszyklus Klasse mit Benutzer definierten Operationen

Lösung zu section 18.1.5 auf Seite 334.

Im Gegensatz zu den Compiler synthetisierten Methoden geben alle Methoden ihre Signatur aus. Die Move Operationen setzen den Wert der Quelle auf 0; (siehe Listing 296 auf Seite 334 Zeile 13 und Zeile 26). Das hat in diesem Fall nur symbolische Bedeutung, es soll darauf hinweisen, dass die Verantwortung für eine Resource übernommen wurde, das Quellobjekt also nicht mehr für deren Freigabe verantwortlich ist.

Listing 304: Ausgabe mit der Klasse mit benutzerdefinierten Operationen

```
1  === begin Lebenszyklus()
2  === begin Block
3  {
4  === A a1;
5  A::A()
6  a1.i: 42
7  === A a2 = a1;
8  A::A(A const& src)
9  a2.i: 42
10 === A a3(a1);
11 A::A(A const& src)
12 a3.i: 42
13 === a1.i = 43 43
14 === a2 = a1
15 A& A::operator=(A const& src)
16 a2.i: 43
17 === A a4(move(a1));
18 A::A(A && src)
19 a1.i: 0
20 a4.i: 43
21 a4 = move(a3);
22 A& A::operator=(A && src)
23 a3.i: 0
24 a4.i: 42
25 === A a5 = f();
26 A::A()
27 a5.i: 42
28 === f(A());
29 A::A()
30 void f(A&& a)
31 A::~A()
32 === f(a4);
33 void f(A& a)
34 === end Block
35 }
36 A::~A()
37 A::~A()
38 A::~A()
39 A::~A()
```

```
A::~A()
=== end Lebenszyklus()
```

18.6.3 Lebenszyklus Klasse A mit Member/Base Klassen

Die Funktion `lebenszyklus()` aus Listing 295 auf Seite 332 erzeugt für die Klasse `A` aus Listing 297 auf Seite 335, sowie Listing 298 auf Seite 336 und Listing 299 auf Seite 337 stets dieselbe Ausgabe aus Listing 305. Wie diese Ausgabe zustande kommt unterscheidet sich jedoch zwischen den einzelnen Implementierungen der Klasse `A` und wird im Anschluss getrennt beschrieben.

Listing 305: Ausgabe der Klasse A mit Member/Base Klassen

```
=== begin Lebenszyklus()
=== begin Block
{
=== A a1;
C::C()
B::B()
A::A()
a1.i: 42
=== A a2 = a1;
C::C(C const& src)
B::B(B const& src)
A::A(A const& src)
a2.i: 42
=== A a3(a1);
C::C(C const& src)
B::B(B const& src)
A::A(A const& src)
a3.i: 42
=== a1.i = 43 43
=== a2 = a1
C& C::operator=(C const& src)
B& B::operator=(B const& src)
A& A::operator=(A const& src)
a2.i: 43
=== A a4(move(a1));
C::C(C && src)
B::B(B && src)
A::A(A && src)
a1.i: 0
a4.i: 43
a4 = move(a3);
C& C::operator=(C && src)
B& B::operator=(B && src)
A& A::operator=(A && src)
a3.i: 0
a4.i: 42
=== A a5 = f();
```

```
38 C::C()
39 B::B()
40 A::A()
41 a5.i: 42
42 === f(A());
43 C::C()
44 B::B()
45 A::A()
46 void f(A&& a)
47 A::~A()
48 B::~B()
49 C::~C()
50 === f(a4);
51 void f(A& a)
52 === end Block
53 }
54 A::~A()
55 B::~B()
56 C::~C()
57 A::~A()
58 B::~B()
59 C::~C()
60 A::~A()
61 B::~B()
62 C::~C()
63 A::~A()
64 B::~B()
65 C::~C()
66 A::~A()
67 B::~B()
68 C::~C()
69 === end Lebenszyklus()
```

Lebenszyklus Klasse A mit Klasse B und C als Member

Lösung zu section 18.1.6 auf Seite 334.

Die Klasse A aus Listing 297 auf Seite 335 hat je einen Member der Klasse `B b;` und `C c;`

Wird ein Objekt von A erzeugt oder kopiert, müssen diese Member ebenfalls erzeugt bzw. kopiert und zu gegebender Zeit in umgekehrter Reihenfolge wieder zerstört werden.

Entfernen Sie die Methoden aus der Klasse A. Wie wird sich die Ausgabe verändern?

Lebenszyklus Klasse A mit Klasse C und B als Base

Lösung zu section 18.1.7 auf Seite 336.

Die Klassen C und B sind Basisklassen der Klasse A aus Listing 298 auf Seite 336.

Wird ein Objekt von A erzeugt oder kopiert, muss der Anteil der jeweiligen Basis-

klasse in der Reihenfolge (von links nach rechts) wie sie in der Klassendefinition aufgeführt sind ebenfalls erzeugt bzw. kopiert und zu gegebender Zeit in umgekehrter Reihenfolge wieder zerstört werden.

Siehe section 6.16.3 auf Seite 52.

Lebenszyklus Klasse A mit Klasse B als Member und C als Base

Lösung zu section 18.1.8 auf Seite 337.

Das Beispiel aus Listing 299 auf Seite 337 zeigt, zuerst werden die Konstruktoren der Basisklassen, anschliesend die Konstruktoren der Member und zuletzt der Konstruktor der Klasse gerufen wird.

18.6.4 Increment Pre- und Postfix Operator

Lösung für section 18.5.1 auf Seite 343.

Listing 306: Ausgabe Increment Pre- Postfix

```
1 aufg0
2 (++i)++ : 1
3 i:2
```

Wird die Klammer entfernt, wird zuerst der Postinkrement Operator angewendet. Dieser evaluiert zu einem R-Value Ausdruck, da er ein temporäres Objekt zurück liefert, auf das dann der Präinkrement Operator angewendet wird, was zu einer entsprechenden Fehlermeldung führt: `error: lvalue required as increment operand`. Beide Operatoren erwarten einen L-Value Ausdruck. siehe section 6.25.15 auf Seite 100.

18.6.5 Call by Value

Lösung für section 18.5.2 auf Seite 344.

Listing 307: Ausgabe Call by Value

```
1 2 1 0 0 1 2
```

Die Funktion ruft sich selbst rekursiv auf. Der Parameter `int i` ist lokal und wird bei jedem Aufruf mit dem übergebenen Wert initialisiert. Die Änderungen an `i` sind nur innerhalb der Funktion sichtbar. siehe section 6.32.5 auf Seite 128.

18.6.6 Call by Reference vs. by Value

Lösung für section 18.5.3 auf Seite 344.

Listing 308: Ausgabe Call by Reference vs. by Value für 1

```
1 i:3
2 2 1 0 0 1 2
3 i:3
```

Die Funktion **void** f(**int** i) ruft sich selbst rekursiv auf. Der Parameter **int** i ist lokal und wird bei jedem Aufruf mit dem übergebenen Wert initialisiert. Die Änderungen an i sind nur innerhalb der Funktion sichtbar.

Listing 309: Ausgabe Call by Reference vs. by Value für 2

```
1 i:3
2 2 1 0 0 -1 -2
3 i:-3
```

Die Funktion **void** f(**int**& i) ruft sich selbst rekursiv auf. Der Parameter **int**& i ist eine Referenz[317] auf ein **int** Objekt und wird bei jedem Aufruf mit dem übergebenen Objekt initialisiert. Die Änderungen an i werden an dem referenzierten Objekt, der lokalen Variablen i aus der Funktion **void** aufgOb(), wirksam. siehe section 6.32.5 auf Seite 128

[317] siehe section 6.29 auf Seite 111

Teil VIII

Anhang

Literatur

[Ale09] ALEXANDRESCU, Andrei: *Modern C++ Design, Generic Programming and Design Patterns Applied*. 17th Printing. Addison Wesley, 2009 (C++ In-Depht Series - Bjarne Stroustrup). – ISBN 0–201–70431–5

[Ast03] ASTELS, David: *Test-Driven Development, A Practical Guide*. Prentice Hall, 2003. – ISBN 0–13–101649–0

[Bec03] BECK, Kent: *Test-Driven Development, By Example*. Addison Wesley, 2003. – ISBN 0–321–14653–0

[Bec13] BECKER, Thomas: C++ Rvalue References Explained. In: *www.thbecker.net* (2013). – http://thbecker.net/articles/rvalue_references/section_01.html

[GHJV95] GAMMA, E. ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995 (Professional Computing Series). – ISBN 0–201–63361–2

[Gri12] GRIMM, Rainer: *C++11*. Addison Wesley, 2012. – ISBN 978–3–8273–3088–8

[Här07] HÄRDTLEIN, Jochen: *Moderne Expression Templates Programmierung*. SCS Publishing House, 2007 (Advances in simulation). – ISBN 978–3–93615–0–513

[HT03] HUNT, Andrew ; THOMAS, David: *Der Pragmatische Programmierer*. Carl Hanser Verlag, 2003. – ISBN 3–446–22309–6

[Jos04] JOSUTTIS, Nicolai M.: *The C++ Standard Library, A Tutorial and Reference*. 13th Printing. Addison Wesley, 2004. – ISBN 0–201–37926–0

[Jos12] JOSUTTIS, Nicolai M.: *The C++ Standard Library,Second Edition, A Tutorial and Reference*. 1st Printing. Addison Wesley, 2012. – ISBN 0–321–62321–5

[Mey99] MEYERS, Scott: *More Effective C++, 35 New Ways to Improve Your Programs and Designs*. First Edition. Addison Wesley, 1999 (Professional Computing Series). – ISBN 0–201–63371–X

[Mey06a] MEYERS, Scott: *Effective C++, 55 Specific Ways to Improve Your Programs and Designs*. Third Edition. Addison Wesley, 2006 (Professional Computing Series). – ISBN 0–321–33487–6

[Mey06b] MEYERS, Scott: *Effective STL, 50 Specific Ways to Improve Your Use of the Standard Template Library*. 9th Printing. Addison Wesley, 2006 (Professional Computing Series). – ISBN 0–201–74962–9

[Mey12] MEYERS, Scott: Universal References in C++. In: *www.isocpp.org* (2012). – http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers

[Mey15] MEYERS, Scott: *Effective Modern C++, 42 Specific Ways to Improve Your Use of C++11 and C++14*. Fifth Release. O´Reilly, 2015. – ISBN 978–1–491–90399–5

[Pre95] PREE, Wolfgang: *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995. – ISBN 0–201–42294–8

[Str00a] STROUSTRUP, Bjarne: *Die C++ Programmiersprache*. 4. aktualisierte Auflage. Addison Wesley, 2000 (Professional Computing Series). – ISBN 3–8273–1756–8. – deutsche Übersetzung von Nicolai Josuttis und Achim Lörke

[Str00b] STROUSTRUP, Bjarne: Wrapping calls to member functions. In: *C++ Report, June 2000* (2000)

[Str13] STROUSTRUP, Bjarne: *the C++ Programming Language*. Fourth Edition. Addison Wesley, 2013 (Professional Computing Series). – ISBN 978–0–321–56384–2

[Sut01] SUTTER, Herb: *Exceptional C++*. Addison Wesley, 2001. – ISBN 3–8273–1711–8

[VJ03] VANDEVOORDE, David ; JOSUTTIS, Nicolai M.: *C++ Templates, The Complete Guide*. 3th Printing. Addison Wesley, 2003. – ISBN 0–201–73484–2

[WD12] WILL DIETZ, John Regehr Vikram A. Peng Li L. Peng Li: Understanding Integer Overflow in C/C++. In: *Proc. ICSE 2012*, 2012

Abbildungsverzeichnis

Diagrammverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Weitere nützliche Quellen

Tabelle 13: Weitere nützliche Quellen

Beschreibung	Verweis
Eine Online Reference für die C++ Standard Library	http://en.cppreference.com/w/Main_Page
Fragen und Antworten zu verschiedenen Themen	http://stackoverflow.com/
UML Diagramme	http://www.uml-diagrams.org/
Eclipse	http://eclipse.org/downloads/
Eclipse Wiki	http://wiki.eclipse.org/Main_Page
CDT Dokumentation	www.eclipse.org/cdt/documentation.php
Help für Juno, Kepler, usw	help.eclipse.org/juno/index.jsp

Index