

ODEMx0x

**Verwendung neuer Sprachmerkmale aus C++11 in einer
Prozesssimulationsbibliothek**

**Studienarbeit am Institut der Informatik der Humboldt-Universität zu
Berlin**

Betreuer: Dr. Klaus Ahrens (ahrens@informatik.hu-berlin.de)

Magnus Müller

Berlin, den 30. Juli 2012

Inhaltsverzeichnis

Einführung	v
1. Überblick	v
1.1. Die Programmiersprache C++	v
1.2. Die Prozesssimulationsbibliothek ODEMx	v
1.3. Ziel und Aufbau der Arbeit	vii
1.4. Anmerkungen zur Darstellung	viii
 1. Der Sprachstandard C++11	 1
1.1. Initialisierung in C++	2
1.1.1. Braced Initialization	5
1.1.2. Initialisierungslisten	8
1.2. Rvalue Referenzen und Move Semantic	10
1.2.1. Perfect Forwarding	16
1.3. Automatische Typableitung	20
1.4. Iteration über Mengen	22
1.5. Umgang mit speziellen Methoden	23
1.5.1. Explizite Nutzung oder Löschung impliziter Methoden	24
1.6. Eine Konstante für den Nullzeiger	26
1.7. Stark typisierte Aufzählungen	26
1.8. Variadic Templates	28
1.8.1. Eine typsichere <code>printf</code> Funktion	28
1.8.2. Tuple in C++11	29
1.9. Hülltypen für Zeiger	31
1.10. Weitere neue Sprachmerkmale	32
 2. Verwendung der neuen Features in ODEMx	 35
2.1. Das PortT Warteschlangenkonzept in ODEMx	35
2.1.1. Implementation des Kommunikationspuffers	36
2.1.2. Mängel der Portimplementation	38
2.1.3. Lösungsimplementation	40

2.1.4. Performanz der Änderung	42
2.2. Prädikate und Gewichtsfunktionen	43
2.3. Process::wait	46
2.3.1. Vereinfachung durch C++11	47
2.3.2. Implementation in ODEMX	47
2.3.3. Performanz	48
3. Fazit und Ausblick	53
3.1. Ausblick	53
3.1.1. Weitere neue Sprachmerkmale und Änderungen der Standardbibliothek	54
3.1.2. Vereinigung BinT/ResT	54
3.1.3. Performanzmessung der Bibliothek	55
3.1.4. Simulationszeit als Template Parameter für Simulationen	55
3.1.5. Ersetzen von Aufzählungstypen in ODEMX	56
3.2. Fazit	56
A. Tabellarische Übersicht der ausgelassenen Neuerungen	59
B. Ausgewählte Grammatikregeln	61
B.1. Braced initialization	61
C. Hilfsprogramme und -algorithmen	63
C.1. Type Traits	63
D. Programme zur Messung von Laufzeitverhalten	65
D.1. Portable Benchmark	65
D.2. Flexiblere Benchmarkklasse	67
D.3. Unterschied move gegen copy semantic bei swap	69
D.3.1. Einfacher Benchmark	69
D.3.2. Hinnant's Benchmark	71
D.4. Performanz der veränderten Portimplementation	73
Literaturverzeichnis	79

Einführung

1. Überblick

Diese Studienarbeit setzt sich mit Sprachstandard C++11 der Programmiersprache C++ im Kontext der Prozesssimulationsbibliothek ODEMX auseinander. Um dem Leser den Einstieg in die zwei Themenbereiche C++11 und ODEMX zu erleichtern, wird dieses Kapitel eine kurze Einführung in diese geben. Um den Rahmen der Studienarbeit nicht zu sprengen wird, wo nötig, auf Primärliteratur verwiesen, die dem interessierten Leser genauere Details vermittelt und Grundlagen ausführlicher erklärt. So wird beispielsweise die Verwendung der Programmiersprache C++ nicht eingeführt, sondern auf Lehrliteratur wie zum Beispiel [Str00] verwiesen.

1.1. Die Programmiersprache C++

C++ ist eine der wenigen Programmiersprachen, die bereits über einen sehr langen Zeitraum genutzt wird und weiterhin beliebt ist. Dies ist besonders beeindruckend, da C++ eine steile Lernkurve aufweist und aufgrund der vielen vorhandenen Paradigmen als sehr komplex gilt. Die Anfänge von C++ gehen bis in das Jahr 1979 zurück [Str96], als Bjarne Stroustrup eine Programmiersprache schaffen wollte, welche die Ordnungsstrukturen aus der Programmiersprache Simula mit der Effizienz der Sprache C vereinigen sollte. Nach sehr starkem Wachstum in den 1980er Jahren und linearem Wachstum in den 1990er Jahren befindet sich C++ heute laut dem TIOBE-Index [BV11] weiterhin unter den meistgenutzten Programmiersprachen. 2011 wurde ein neuer Standard C++11 veröffentlicht, der die Sprache um weitere Merkmale ergänzt und Probleme im alten Standard behebt.

1.2. Die Prozesssimulationsbibliothek ODEMX

Um zu verstehen, wozu eine Prozesssimulationsbibliothek benötigt wird, müssen wir den eigentlichen Begriff zuerst in seine Teilbegriffe zerlegen und analysieren, was die Teilbegriffe jeweils bedeuten. Dieser Abschnitt betrachtet also die Begriffe Simulation, Prozess und

Bibliothek, bevor ein kurzer Überblick über ODEMx, ergänzt um Primärliteratur, gegeben wird.

Was ist Simulation?

Wenn wir den Begriff Simulation verwenden, meinen wir im Allgemeinen die *Computer-simulation*, also die Ausführung eines Simulators, der ein formales Modell eines Systems implementiert. Wir bezeichnen hierbei ein *System* als eine Menge von Objekten, real oder nicht-existent, die miteinander in irgendeiner Verbindung stehen und von einer Systemumgebung getrennt betrachtet werden können. Zwischen dem System und der Systemumgebung darf es Kopplungen, aber keine Rückkopplungen, geben. In [FA96] wird der Begriff System genauer dargestellt und analysiert. Für unsere Betrachtung reicht es aus zu sagen, dass ein System eine Menge von *Phänomenen*¹ ist, aus der ein Modell gebildet werden kann. Ein Modell wiederum ist eine Abstraktion, also eine Vereinfachung, die nur bestimmte Teile des Originalsystems abbildet. Fischer und Ahrens betrachten den System- und Modellbegriff in [KFA07a, S. 2] genauer.

Die Erstellung von Modellen bietet sich immer dann an, wenn nur bestimmte Aspekte eines komplexen Systems genauer analysiert werden sollen. Im Falle von einfachen Modellen kann dann eine analytische Lösung für das Modell erzeugt werden, um das *Verhalten* des Modells über der Zeit zu betrachten. Dies ist ein Forschungsgebiet der Mathematik. Es zeigt sich jedoch, dass bereits einfach wirkende Modelle eine Komplexität aufweisen können, die eine analytische Lösung nicht oder nur schwer umsetzbar machen. An dieser Stelle setzt die Simulation an: Ein formales Modell, das in einen Simulator überführt wurde, kann auf einem Rechner schrittweise in einer beliebigen Geschwindigkeit ausgeführt werden.² Die Lösung beziehungsweise das Verhalten eines dynamischen Systems wird damit nicht analytisch berechnet, sondern durch die Ausführung des Simulators approximiert.

Eine generische Prozesssimulationsbibliothek

ODEMx ist eine generische, objektorientierte Prozesssimulationsbibliothek zur Ausführung von Simulationen auf Computern. Sie bietet dem Nutzer verschiedene Komponenten an, um zielgerichtet ein dynamisches System modellieren und ausführen zu können. Wir unterscheiden in einem System *aktive* und *passive* Komponenten: Aktive Komponenten sind Objekte, die einen eigenen Lebenslauf, also ein Verhalten über Zeit, aufweisen. Passive Komponenten sind hingegen Objekte, deren Zustandsänderung nur durch Einwirkung aktiver Komponen-

¹Hierunter verstehen wir ein wahrnehmbares oder gedachtes Ereignis.

²Wobei die mögliche Geschwindigkeit natürlich durch die vorhandenen Rechenressourcen und der Komplexität des Modells eingeschränkt wird.

ten ausgelöst werden können. ODEMx nutzt die Abstraktion des *Prozesses* zur Darstellung aktiver Komponenten in dem Modell eines Systems. Ein Prozess ist hierbei kein aus dem Umfeld von Betriebssystemen bekannter Prozess, sondern ein an das Aktor-Modell erinnernde ausführbare Einheit, die mit anderen Prozessen in Verbindung stehen kann. Prozesse können Nachrichten austauschen, neue Prozesse schaffen und Prozesse beenden. Das Verhalten eines durch ODEMx implementierten Modells wird durch das Verhalten der ausgeführten Prozesse bestimmt. [Ger03] bietet eine ausführliche Einführung in die Prozesssimulation, weshalb im Folgenden nur einige Grundlagen dargestellt werden.

ODEMx Prozesse sind Koroutinen [Con63], die über verschiedene Mittel kommunizieren können. So bietet ODEMx zum Beispiel Synchronisationsobjekte an, um den Nachrichtenaustausch zwischen Prozessen zu realisieren. Die Ausführung der Prozesse findet serialisiert statt. In diesem als *next-event simulation* bekannten Konzept werden die Koroutinen in einen „Kalender“ eingefügt und der Reihe anhand von Prioritäten ausgeführt. Ein Prozess kann bei seiner Ausführung die Kontrolle abgeben und sich an einen späteren Zeitpunkt im Kalender eintragen. Nähere Informationen hierzu finden sich in [KFA07b] und werden hier aus Platzgründen ausgelassen.

ODEMx bietet neben der Grundfunktionalität der *next-event simulation* auch weitere Komponenten an, die dem Nutzer die Implementation von Simulationen erleichtern sollen. Hierzu gehören Klassen zur Statistikbildung, Synchronisationsklassen zur Kommunikation und Austausch von Daten zwischen Prozessen und Klassen zur Datenhaltung, die den Zugriff auf Datenbanken abstrahieren oder auch die Generierung von Berichten über einen Simulationslauf vereinfachen. Damit bietet die Prozesssimulationsbibliothek Lösungen für viele Probleme an, die bei der Modellierung von dynamischen Systemen auftreten.

1.3. Ziel und Aufbau der Arbeit

Nachdem in den vorherigen Abschnitten die Grundlagen für diese Arbeit eingeführt wurden, können nun die eigentlichen Ziele der Arbeit festgelegt werden. Das Ziel der Arbeit ist die Analyse einer Auswahl neuer Sprachmerkmalen und Erweiterungen der Standardbibliothek von C++ im Kontext der Prozesssimulationsbibliothek ODEMx. Die neuen Sprachmerkmale werden motiviert und einige Einsatzszenarien anhand von Fallbeispielen in ODEMx skizziert. Zudem sollen, wo möglich, Performanzmessungen die neuen Sprachmerkmale in Relation zu bisherigen Lösungen mit älteren Sprachmitteln stellen. In Kapitel 1 führen wir eine Auswahl neuer Sprachmerkmale ein. Kapitel 2 stellt Teile der Bibliothek ODEMx auf, an denen neue Sprachmerkmale eingesetzt werden und zeigt mittels Messungen, wo diese gewinnbringend waren und wo nicht. Abschließend ziehen wird in Kapitel 3 ein Fazit gezogen und durch einen Ausblick ergänzt.

1.4. Anmerkungen zur Darstellung

Im Kapitel 1 werden vielfach Abschnitte aus C++11 [ISO11] und C++98/03 [ISO03] zitiert. Um diese leicht erkennbar zu gestalten, werden Zitate aus dem neuen Standard C++11 wie unten dargestellt blau hinterlegt und Zitate aus dem vorherigen Standard durch grünen Hintergrund angezeigt. Hierbei befindet sich der Abschnitt des Standards, in dem sich das jeweilige Zitat befindet, in der oberen rechten Ecke.

C++11 [§1.1p2]

C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:1999 Programming languages — C (hereinafter referred to as the C standard). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

C++98/03 [§1.1p2]

C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:1990 Programming languages – C (1.2). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, inline functions, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

Die Darstellung des Abschnitts erfolgt hierarchisch: Kapitel, Abschnitte und Unterabschnitte werden durch Punkte getrennt, ein Paragraph wird durch ein vorstehendes *p* angezeigt und ein Listenelement wird durch ein vorstehendes *i* bestimmt. So ist §14.3.2p1i2 als Kapitel 14, Abschnitt 3, Unterabschnitt 2, Paragraph 1, Listenelement 2 zu verstehen.

1. Der Sprachstandard C++11

Um C++ durch mehr Konsistenz und Vereinheitlichungen zu vereinfachen, die Sprache von neuen Methoden des Compilerbaus profitieren zu lassen, und um auf Änderungen der Hardwarearchitektur reagieren zu können, wird C++ im Laufe der Zeit um neue Sprachmittel und neue Methoden in der Standardbibliothek erweitert. Diese Entwicklungen finden in Standardisierungsgremien statt, um eine offene und portable Sprache anzubieten. Der letzte veröffentlichte C++-Standard war ISO/IEC 14882:1998 (abgekürzt C++98). Dieser Standard ist zum Beginn dieser Arbeit bereits 13 Jahre alt und merklich in die Jahre gekommen. So betrachtet dieser beispielsweise nur Programme mit einem Kontrollfaden, während *threads*¹, atomare Operationen oder Synchronisation nicht erwähnt werden. Das Fehlen dieser Konzepte und weiterer erweckte den Wunsch nach einem neuen Sprachstandard, der C++98 um neue Funktionen ergänzt und Fehler innerhalb des Standards eliminiert. Damit soll der neue Standard eine zukunftsfähige Sprache hervorbringen, die die Konkurrenz mit neuen Programmiersprachen nicht zu scheuen braucht. Neue Sprachmerkmale wie anonyme Funktionen und Erweiterungen der Standardbibliothek, zum Beispiel um Mittel zur generischen Typtransformation durch Templates, bezwecken eine Vereinfachung der Sprache und Ausstattung des Anwenders mit Mitteln, um auch aktuelle Probleme der Softwareentwicklung elegant und mit möglichst wenig Performanzverlust lösen zu können.

Aufgrund der Popularität, der weiten Verbreitung und des Alters von C++ mussten bei Überarbeitung des Sprachstandards viele Randbedingungen beachtet werden, um möglichst wenig alten Code zu brechen und trotzdem neue Sprachmittel zur Verfügung stellen zu können. Stroustrup beschreibt in seiner FAQ² [Str11b] die Ziele bei der Entwicklung des neuen Sprachstandards genauer:

The overall aims of the C++0x effort was to strengthen that:

- *Make C++ a better language for systems programming and library building – that is, to build directly on C++'s contributions to programming, rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development).*

¹Threads sind nebenläufig ablaufenden Kontrollfäden, also Ausführungseinheiten, die durch einen Prozessor ausgeführt werden können.

²FAQ ist eine Abkürzung für *frequently asked questions*, was übersetzt *häufig gestellte Fragen* bedeutet.

1. Der Sprachstandard C++11

- *Make C++ easier to teach and learn – through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts).*

*Naturally, this is done under very stringent compatibility constraints. Only very rarely is the committee willing to break standards conforming code, though that's done when a new keyword (e.g. `static_assert`, `nullptr`, and **`constexpr`**) is introduced.*

Erwartungsgemäß ist der neue Standard sehr umfangreich geworden. Die Vielzahl neuer Elemente erzwingt uns, nur eine Auswahl genauer zu betrachten. Diese Auswahl erfolgte unter den Aspekten *Nützlichkeit* und *vorhandene Implementation*. Alle gewählten Elemente werden in GCC³4.7 unterstützt. Andere Compiler wurden nicht zugrunde gelegt, da GCC zum Zeitpunkt der Auswahl mit Abstand die meisten Sprachmerkmale implementierte. Zu einem späteren Zeitpunkt könnten andere Compiler nachgezogen haben.⁴

Nachfolgend wird eine Auswahl von Sprachmerkmalen beschrieben. Hierbei geben wird jeweils eine kurze Motivation des jeweiligen Merkmals und eine Darstellung der Verwendung gegeben. Einige der Sprachmerkmale lassen einen Performanzverlust oder -gewinn in der Ausführung vermuten. Diese Vermutungen werden anhand von Messungen widerlegt oder bestätigt.

1.1. Initialisierung in C++

C++ gilt seit langem als Sprache mit einer steilen Lernkurve. Die Syntax bietet viele Schwachstellen, über die sowohl Anfänger als auch fortgeschrittene Entwickler öfter stolpern. Ein spezielles Problem der komplexen Syntax ist die Initialisierung beliebiger Objekte der Sprache. C++ kennt eine Vielzahl von Initialisierungsmöglichkeiten von Objekten und Variablen, deren Uneinheitlichkeiten gerade Anfänger oft irritieren. Stroustrup beschreibt diese Problematik in [Str09, S. 3]. Hierzu einige Initialisierungsbeispiele:

```
int a; // a hat den Wert 0, ausser wenn a lokal definiert ist
int b = 10; // b hat den Wert 10 durch die Initialisierung
int c = {1}; // c hat den Wert 1 durch die Initialisierung

int *p; // Ein Zeiger auf eine Ganzzahl.
int *p2 = new int(10); // p2 zeigt auf eine Ganzzahl mit dem Wert 10
```

Listing 1.1: Initialisierung einfacher Typen

³GCC ist eine Abkürzung für *Gnu Compiler Collection*, eine Menge von Compilern der *Free Software Foundation*. Der für C++ zuständige Compiler in dieser Kollektion ist `g++`

⁴Eine aktuelle Übersicht für Clang findet sich unter http://clang.llvm.org/cxx_status.html und für Microsoft® Visual Studio unter <http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx>.

Dieses Beispiel enthält bereits 4 verschiedene Arten der Initialisierung: Initialisierung mit einem Standardwert, Initialisierung mit einem nutzerdefinierten Wert, Listeninitialisierung⁵ und Initialisierung eines Zeigers mit der Adresse auf einen Speicherplatz, der wiederum mit einem nutzerdefinierten Wert initialisiert wurde. Werden nun weitere C++-Konzepte wie Klassen und Felder hinzugenommen, so wird die Situation weiter verkompliziert. Hierzu ein Beispiel:

```

/// Eine einfache Klasse
struct S {
    int a, b;
};

class T {
    int a;

public:
    T() {} // Standardkonstruktor
};

void f( S* esses ); // Funktionsdeklaration

/* Feld mit 6 Ganzzahlen und Initialisierung der Elemente */
int array[6] = {1,2,3,4,5,6};

/* Initialisiert ein Objekt vom Typ s und setzt a = 1, b = 2 */
S s = {1,2};
T t = {1}; // #1 Geht nicht.

S esses[] = { {1,2}, {3,4} }; // #2 Initialisiere ein Feld wie oben
f( esses ); // funktioniert
f( { {1,2},{3,4} } ); // Geht nicht #3

```

Listing 1.2: Initialisierungsmöglichkeiten bei Klassen und Feldern

Vergleicht man obige Initialisierungen, so fällt es schwer, eine Regelmäßigkeit zu entdecken: Felder und einfache Typen können über Listen initialisiert werden, wobei für einfache Typen wie `s` folgende Einschränkung erfüllt sein muss [Str09, S. 4],

When does `S s = { 1, 2, 3 };` actually work? It works in C++98 iff `S` is a struct (or an array) with at least three members that can be initialized by an int and iff `s` does not declare a constructor.

Da der Typ `T` einen Konstruktor definiert, kann er in C++98 nicht durch eine Liste initialisiert werden. Daher funktioniert der mit #1 markierte Fall nicht, was sich aber nur durch die Semantik, nicht aber bereits durch die Syntax erschließt.

Auch die Initialisierung und Verwendung von Feldern ist uneinheitlich. So kann man ein Feld von `s` Objekten zwar durch eine Liste initialisieren und dieses Feld dann auch an eine Funktion

⁵C und C++98 erlauben die Initialisierung einfacher Typen durch eine Liste, wie die Initialisierung von `c` zeigt.

1. Der Sprachstandard C++11

übergeben (vgl. Markierung #2), die einen Zeiger erhalten will, aber der Funktionsaufruf kann nicht durch ein direkt initialisiertes Feld erfolgen. In C++98 besteht also bzgl. der Initialisierung ein Ungleichgewicht zwischen einfachen Typen (*concrete types*) und komplizierteren Strukturen. So erscheint es nicht logisch, dass ein Feld von Ganzzahlen durch eine Liste initialisiert werden kann, aber ein `std::vector< int >` nicht:

```
int a[] = {1,2,3,4}; // Geht
std::vector< int > is = {1,2,3,4}; // Geht nicht
```

Damit ist der Typ `std::vector< int >` gegenüber einem internen Typ benachteiligt. Dieses Problem geht der neue Sprachstandard mit einer oftmals unter dem Begriff *uniform initialization* einhergehenden Änderungen an: Durch eine Vereinheitlichung der Syntax soll eine Initialisierung auf den ersten Blick als solche erkennbar sein. Zudem werden Konstrukte eingeführt, die es dem Nutzer erlauben, auch eigene Typen um Möglichkeiten der Initialisierung mit einer Liste zur Angleichung an sprachdefinierte Typen zu ermöglichen.

Ein weiteres Problem liegt bei der gleichzeitigen Deklaration und Wertinitialisierung einer Variablen vor. C++ erlaubt die Initialisierung mit einem Standardwert durch die Verwendung der leeren Klammer `()`, zum Beispiel in `x x = x()`. Da sowohl in C als auch C++ eine Funktionsdeklaration⁶ auch funktionslokal durchgeführt werden kann, muss aber folgende Eigenheit der Sprache beachtet werden:

C++11 [§8.5p10]

[Note: Since `()` is not permitted by the syntax for initializer,

```
x a();
```

is not the declaration of an object of class `x`, but the declaration of a function taking no argument and returning an `x`. The form `()` is permitted in certain other initialization contexts (5.3.4, 5.2.3, 12.6.2). — end note]

Nach der obigen Darstellung verschiedener Schwierigkeiten im Kontext der Initialisierung werden im folgenden Teil die *braced initialization* und `std::initializer_list` betrachtet, welche als Lösung der obigen Probleme in den neuen Standard aufgenommen wurden.

⁶Also die Definition einer Signatur, ohne Definition (Implementation) der Funktion selbst.

1.1.1. Braced Initialization

Um die Initialisierungssyntax zu vereinfachen, führt C++11 die *braced initialization*, also die Initialisierung mit den geschweiften Klammern '{' und '}', ein. Es ist anzumerken, dass auch die bisherigen Möglichkeiten der Initialisierung zwecks Abwärtskompatibilität möglich bleibt. Der Zweck der *braced initialization* ist, dass man eine Initialisierung „auf den ersten Blick“ als solche erkennt.

Die Initialisierung von Bezeichnern bei deren Deklaration oder in anderen syntaktischen Kontexten wird in C++11 mit [ISO11, §8.5] definiert.⁷ Hierbei wird zwischen Initialisierung von *aggregates* [ISO11, §8.5.1] (Feld oder Klasse ohne nutzerdefinierten Konstruktor), *character arrays* [ISO11, §8.5.2], *Referenzen* [ISO11, §8.5.3] und der *braced initialization* [ISO11, §8.5.4] unterschieden. Die Initialisierung von Aggregaten erlaubt im obigen Beispiel die Initialisierung eines Objekts vom Typ `s` unter Nutzung einer Liste. Die Initialisierung von *character arrays* bezieht sich auf die aus C bekannte Initialisierung `char* string = "hello world";`. Auf die Initialisierung von Referenzen wird im Abschnitt 1.2 zu *rvalue Referenzen* eingegangen.

Die *braced initialization* wird unter dem Begriff *list initialization* in Abschnitt §8.5.4 des Standards [ISO11] eingeführt:

C++11 [§8.5.4]

List-initialization is initialization of an object or reference from a braced-init-list. Such an initializer is called an initializer list, and the comma-separated initializer-clauses of the list are called the elements of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called direct-list-initialization and list-initialization in a copy-initialization context is called copy-list-initialization.

Entsprechend der im Anhang B.1 beigelegten Grammatikregeln zur Spezifikation diesen Teils der Sprache kann eine Initialisierung mit geschweiften Klammern nach einem Gleichheitszeichen oder ähnlich der aus C++98 bekannten Initialisierung `S s(1,2,3);` mit `S s{1,2,3};` erfolgen. Um dies zu veranschaulichen, enthält der Abschnitt über zur *list initialization* eine Reihe von einführender Beispiele:

⁷Dieser Ausarbeitung liegt eine Arbeitsversion des Standards zugrunde.

```

int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once", "upon", "a", "time"}; // 4 string elements
f( {"Nicholas","Annemarie"} ); // pass list of two elements
return { "Norah" }; // return list of one element
int* e {}; // initialization to zero / null pointer
x = double{1}; // explicitly construct a double
std::map<std::string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };

```

Die erste Zeile des Beispiels ist die bekannte Initialisierung eines einfachen Typs. Neu ist die Initialisierung von `z` durch `z{1,2}`, durch die der Konstruktor `std::complex(double, double)` aufgerufen wird. Wie in der darauffolgenden Zeile zu sehen, können auch komplexe Container mit Listen initialisiert werden. Die genaue Umsetzung hiervon wird im nachfolgenden Unterabschnitt 1.1.2 dargestellt. Wie an der letzten Zeile des Beispiels zu sehen, kann die Initialisierung auch verschachtelt werden. Hierbei wird eine `std::map<std::string, int>` mit einer Menge von Paaren initialisiert, wobei jedes Paar wiederum durch *braced initialization* initialisiert wird.

In C++ wird an sehr unterschiedlichen Stellen initialisiert: bei der Definition von Variablen, bei Funktionsaufrufen, bei Rückgabe von Rückgabewerten u.s.w. Um die verschiedenen erlaubten Orte der *braced initialization* übersichtlicher zusammenzufassen, hält [ISO11, §8.5.4] in einer Notiz eine Übersicht der erlaubten Stellen bereit:

Note: List-initialization can be used

- as the initializer in a variable definition (8.5)
- as the initializer in a new expression (5.3.4)
- in a return statement (6.6.3)
- as a function argument (5.2.2)
- as a subscript (5.2.1)
- as an argument to a constructor invocation (8.5, 5.2.3)
- as an initializer for a non-static data member (9.2)
- in a mem-initializer (12.6.2)
- on the right-hand side of an assignment (5.17)

Die Initialisierung Die Semantik der Initialisierung wird in [ISO11, §8.5.4p3] festgelegt. Gegeben sei eine Listeninitialisierung eines Objekts oder einer Referenz des Typs T . Dann gilt:

- Wenn die Initialisierungsliste leer ist (`{}`) und T eine Klasse mit einem Standardkonstruktor ist, dann wird das Objekt wertinitialisiert durch Aufruf des Standardkonstruktors.
- Wenn T ein Aggregat ist, dann wird eine Aggregatinitialisierung durchgeführt. Dies entspricht dem ursprünglichen Fall der Initialisierung eines Feldes durch `int[] a = {1, 2, 3, 4};`, erlaubt aber auch die Initialisierung durch `int[] a{ 1, 2, 3, 4 };`
- Wenn der Initialisierer eine Spezialisierung des Typs `std::initializer_list< E >` ist, so wird aus dem Initialisierer ein Objekt des Typs `initializer_list` geschaffen und der im Abschnitt 1.1.2 erklärte *initializer-list constructor* verwendet.
- Wenn T ein Klassentyp ist, so werden die Konstruktoren betrachtet. Falls die Klasse einen *initializer-list constructor* (siehe wieder Abschnitt 1.1.2) besitzt, so wird dieser aufgerufen. Gibt es diesen nicht, so wird der Konstruktor gewählt, der die Elemente der Initialisierungsliste als einzelne Argumente annehmen kann. Hierbei ist zu beachten, dass keine *narrowing conversion* erlaubt ist.⁸
- Der Spezialfall der Initialisierung von Referenzen wird im nächsten Paragraph angesprochen.

Diese Definition betrifft bereits einen großen Teil der möglichen Initialisierungen. Im gleichen Abschnitt werden weitere semantischen Besonderheiten spezifiziert, die hier nicht wiedergeben werden sollen. Stroustrup gibt in [ISO11, S. 4] einige Beispiele an, die den Umgang mit der neuen Syntax illustrieren.

Sonderfall Referenzen In der aktuellen Ausarbeitungsversion des Standards findet sich in der Definition der Initialisierung über Listen eine Stelle, die dem einheitlichen Schema widerspricht: Die Initialisierung von Referenzen. Betrachte folgendes Beispiel:

```
class Y {};  
  
int main()  
{  
    Y y;  
    Y &ref{y};  
}
```

Listing 1.3: Referenzen können nicht durch *uniform initialization* initialisiert werden.

⁸So erlaubt dies zum Beispiel nicht eine implizite Konvertierung von `double` nach `int`. Die Definition der *narrowing conversion* findet sich in [ISO11, §8.5.4p6].

1. Der Sprachstandard C++11

GCC gibt einen Fehler aus beim Versuch der Compilation:

```
g++ -std=c++0x    initialization.cpp    -o initialization
initialization.cpp: In function 'int main()':
initialization.cpp:9:13: error: invalid initialization of non-const reference of type
                        'Y&' from an rvalue of type '<brace-enclosed initializer list>'
```

Dies erscheint unlogisch, denn die Initialisierung sollte ähnlich der einer Ganzzahl `int a{10};` erfolgen können. Die Antwort auf dieses Problem findet sich in folgendem Abschnitt des Standards:

C++11 [§8.5.4p3i5]

Otherwise, if T is a reference to class type or if T is any reference type and the initializer list has no elements, a prvalue temporary of the type referenced by T is list-initialized, and the reference is bound to that temporary. [Note: As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. — end note]

Zum aktuellen Zeitpunkt ist nicht klar, ob es sich dabei um einen Defekt handelt.⁹

1.1.2. Initialisierungslisten

Nachdem im vorherigen Abschnitt die einheitliche Initialisierung durch Verwendung von `{ }` eingeführt wurde, soll nun noch geklärt werden, wie man Containertypen um die Möglichkeit der Initialisierung durch Zuweisung einer Liste erweitern kann.

In C++98 liegt eine weitere Asymmetrie vor, wie bereits weiter oben angemerkt wurde:

```
/* Arrays koennen durch Listen initialisiert werden.. */
int[] a = { 1, 2, 3, 4 };

/* .. nutzerdefinierte Typen aber nicht */
std::vector< int > b = { 1, 2, 3 };
```

Um dem Nutzer ein Mittel zu geben, ein Objekt durch beliebig viele Argumente gleichen Typs initialisieren zu können¹⁰ führt C++ den generische Datentyp `std::list_initializer< T >` ein. Hierzu spezifiziert der Standard:

⁹Vergleiche hierzu einen Bugreport im GCC Bugtracker http://gcc.gnu.org/bugzilla/show_bug.cgi?id=50024 sowie eine kurze Diskussion auf der GCC-Mailingliste gcc-help@gcc.gnu.org <http://gcc.gnu.org/ml/gcc-help/2011-07/msg00049.html>

¹⁰Der oben beschriebene Aufruf eines Konstruktors durch *braced initialization* schließt dies aus, da eine Funktion nur eine beschränkte Anzahl von Parametern haben kann. Ausnahmen sind die *Auslassung (ellipsis "...")* und *variadic templates*. Variadic templates werden in Abschnitt 1.8 betrachtet.

C++11 [§8.5.4p2]

A constructor is an initializer-list constructor if its first parameter is of type `std::initializer_list<E>` or reference to possibly cv-qualified `std::initializer_list<E>` for some type E, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Note: Initializer-list constructors are favored over other constructors in list-initialization (13.3.1.7). — end note] The template `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `std::initializer_list` — even an implicit use in which the type is not named (7.1.6.4) — the program is ill-formed.

Der zitierte Abschnitt des Standards zeigt bereits auf, wie die `std::initializer_list<T>` verwendet wird. Wichtig bei der konkreten Implementation ist die genannte Wahl bei Mehrdeutigkeiten: Falls zwischen einem normalen Konstruktor und einem Konstruktor mit `std::initializer_list< T >` gewählt werden muss, so wird der Konstruktor mit `std::initializer_list< T >` gewählt werden. Zum Abschluss dieses Abschnitts über die Vereinheitlichung der Initialisierungssyntax betrachten wir ein kurzes Beispiel einer Hüllklasse eines Ganzzahlfelds und demonstrieren damit die Verwendung der Klasse `std::initializer_list<T>`:

```
#include <initializer_list>
#include <algorithm>
/// http://iap.cse.tamu.edu/IAP_Initializers, Folie 13
/// Vervollstaendigt um header Dateien und Namensraeume

class IntegerVector {
    int *el_;

public:
    IntegerVector( std::initializer_list< int > elements ) // #1
        : el_( new int[ elements.size () ] ) {
        std::uninitialized_copy( elements.begin(), elements.end(), el_ );
    }

    IntegerVector( int a, int b ) // #2
        : el_( new int[ 2 ] ) {
        el_[0] = a;
        el_[1] = b;
    }
    ~IntegerVector() {
        delete[] el_;
    }
};

int main ()
{
    IntegerVector vec1 = { 1, 2 }; // calls #1
    IntegerVector vec2 { 1, 2 }; // calls #1
}
```

1. Der Sprachstandard C++11

```
IntegerVector vec3( 1, 2 ); // calls #2  
}
```

Listing 1.4: Verwendung der Initialisierungsliste

Der Typ `std::initializer_list< T >` wird wie einer gewöhnlicher Containertyp verwendet, wobei die Elemente solch einer Liste nicht verändert werden dürfen und nur durch Iteratoren erreichbar sind. Die verfügbaren Operationen für eine `std::initializer_list` sind in [ISO11, §18.9] zu finden.

Initialisierungslisten heben sich durch ein Detail von anderen Containern der Standardbibliothek ab: wenn zwischen mehreren Konstruktoren gewählt werden muss und ein Konstruktor mit einer Initialisierungsliste passt, dann wird dieser bei der Auswahl vorgezogen. Daher wird im Beispiel der mit #1 markierte Konstruktor in den ersten beiden Zeilen der Funktion `main` genutzt. Wenn man den Aufruf eines anderen Konstruktors forcieren will, so kann die aus C++98 bekannte Initialisierung von Objekten mit runden Klammern `()` verwendet werden, wie oben durch die letzte Zeile der Funktion `main` dargestellt.

1.2. Rvalue Referenzen und Move Semantic

Bei der Auswertung temporärer Ausdrücke in C++98 werden *temporaries* gebildet, also *flüchtige* Ergebnisse, die als Initialisierer fungieren können. Da diese temporären Ergebnisse zumeist auf der rechten Seite einer Zuweisung stehen, werden diese auch als *rvalue* bezeichnet und setzen sich dadurch von *lvalues* ab, die auf der linken Seite einer Zuweisung stehen dürfen. So sind zum Beispiel konstante *rvalues* und dürfen nicht auf der linken Seite einer Zuweisung stehen. Somit wäre `5 = x;` ein statischer Fehler, den der Compiler bemängeln muss.

Ein *lvalue* kann immer in ein *rvalue* überführt werden, aber ein *rvalue* üblicherweise¹¹ nicht in einen *lvalue*.

Nach dieser informellen Einführung wird nun der Zusammenhang zwischen *lvalue*, *rvalue* und Ausdrücken näher dargestellt, um später die neuen *rvalue Referenzen* motivieren zu können. Ein Grundverständnis über die Kategorisierungen von Ausdrücken in die genannten Klassen ist hierbei nötig, um den *rvalues* motivieren zu können.

Der Standard C++11 kategorisiert Ausdrücke in [ISO11, §3.10] in verschiedene Klassen, die wir hier kurz betrachten wollen. Vergleiche hierzu auch Abbildung 1.1.

¹¹ Ausnahmen von dieser Regel sind zum Beispiel Referenzen, die als Ergebnis einer Funktion zurückgeliefert werden.

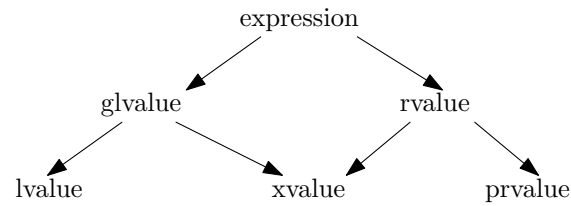


Abbildung 1.1.: Kategorisierung von Ausdrücken, [ISO11, §3.10]

Ein Ausdruck, der ein Objekt oder eine Funktion benennt, als *lvalue*. Ein *xvalue*¹² ist ein Objekt, das sich kurz vor dem Ende seiner Lebenszeit befindet, weshalb die durch das Objekt gehaltenen Ressourcen bewegt, also dass diese dem Objekt “weggenommen”, werden dürfen. *lvalue* und *xvalue* können in der Kategorie *generalized lvalue* (*glvalue*) zusammengefasst werden. Zudem besteht ein *xvalue* aus *rvalue* Referenzen, weshalb ein *xvalue* auch ein *rvalue* ist.

Ausdrücke, die weder *xvalue* noch *lvalue* sind, werden als *pure rvalue* (*prvalue*) bezeichnet. Ein Beispiel hierfür sind Rückgabewerte von Funktionen, die keine Referenzen sind.

Abschließend muss festgehalten werden, dass jeder Ausdruck in C++ genau einer der drei Kategorien *lvalue*, *xvalue*, *prvalue* angehört.

C++98 erlaubt keine Veränderung temporärer Ausdrücke. Somit kann zum Beispiel der Rückgabewert einer Funktion nicht geändert werden. Hierzu spezifiziert [ISO03, §3.10p10]:

C++11 [§3.10p10]

An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [Example: a member function called for an object (9.3) can modify the object.]

Das bedeutet, dass zur Modifikation eines Objekts ein *lvalue* des Objekts vorliegen muss, oder die Veränderungen durch den Effekt eines Memberfunktionsaufrufs erzeugt wird. Ein Objekt, das nicht als *lvalue* vorliegt, kann also nur durch die Memberfunktionen geändert werden.

Ein temporärer Ausdruck ist ein *rvalue*, weshalb er nicht modifiziert werden darf. Um also ein *rvalue* zu modifizieren und das Ergebnis nicht sofort wieder zu verwerfen, muss in C++98 zuerst kopiert werden. Dies gilt auch dann, wenn klar ist, dass der temporäre Ausdruck nachfolgend zerstört wird, also ein *prvalue* ist, und somit auch ohne weitere Einschränkungen

¹²Oder auch *eXpiring value*.

1. Der Sprachstandard C++11

modifizierbar wäre. Um den Performanzverlust durch eine unnötige Kopie zu verhindern und trotzdem die Nichtmodifizierbarkeit eines *rvalues* zu erhalten, kann in C++98 für einen Typ T eine Referenz `const T&` an einen *rvalue* Ausdruck diesen Typs gebunden werden. Das Ergebnis des temporären Ausdruck wird erst dann zerstört, wenn der Geltungsbereich der konstanten Referenz verlassen wird.

Die Folge obiger Einschränkung von temporären Ausdrücken ist das oftmalige Auftreten unnötiger Kopien. Dieses Problem wird in Compilern teilweise durch die Verwendung von *RVO* (*return value optimization*) behoben. Bei dieser Optimierung wird der temporäre Ausdruck direkt im Speicher einer nachfolgenden Kopie erstellt und damit das eigentliche Kopieren vermieden, was zudem den Aufwand einer zusätzlichen Allokationsoperation einspart. Dieses als *elision* bezeichnete Konzept ist durch [ISO03, §12.8p15] spezifiziert:

C++11 [§12.8p15]

When certain criteria are met, an implementation is allowed to omit the copy construction of a class object, even if the copy constructor and/or destructor for the object have side effects. In such cases, the implemen-[sic] tation treats the source and target of the omitted copy operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization.[Fußnote ausgespart])

Jedoch kann der Compiler nicht immer eindeutig entscheiden, wann dies möglich ist. Zudem hat RVO einen Nebeneffekt: Ein Compiler kann sich entscheiden, eine Kopieraktion zu sparen, obwohl sich dadurch der Ablauf des Programms ändert. Insbesondere gewollte Seiteneffekte können der Optimierung zum Opfer fallen.

Um nun das Problem unnötiger Kopien ohne Änderung des Programmablaufs lösen zu können, führt C++11 Referenzen auf *rvalues* ein. Diese werden für einen Typ T durch $T\&\&$ bezeichnet.¹³ Der temporäre Ausdruck, der an solch ein $T\&\&$ gebunden wird, wird erst dann abgeräumt, wenn auch die Referenz ihren Gültigkeitsbereich verlässt. Rückgabewerte einer Funktion sind immer *rvalues*, falls der Rückgabetypp keine Referenz darstellt, wie in der Bemerkung zur Kategorie *prvalue* in [ISO11, §3.10i4] festgehalten wird:

¹³Zur Erinnerung: Für einen Typ T ergibt sich eine Referenz auf diesen Typ durch Hinzufügen von $\&$. Die Referenz hat dementsprechend den Typ $T\&$.

C++11 [§3.10i4]

A prvalue (“pure” rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5, or true is also a prvalue. – end example]

Das erlaubt das Binden einer *rvalue Referenz* an den Rückgabewert einer Funktion zu binden, wie zum Beispiel in `int&& n = std::rand();`. Zudem können *rvalue Referenzen* aber nicht nur zum Binden von Rückgabewerten genutzt werden, sondern auch als Parameter von Funktionen:

```
int f( int&& a ) {
    return a + 10;
}

int main() {
    f( 10 ); // #1 Ok.
    int b = 20;
    f( b ); // #2 ERROR.
}
```

So führt #1 zum Aufruf von `f` mit einer *rvalue Referenz*, während #2 einen Fehler darstellt. [ISO11, 8.3.2p2] legt fest, dass *lvalue Referenzen* und *rvalue Referenzen* verschiedene Typen sind. Eine Zuweisung einer *lvalue Referenz* an eine *rvalue Referenz* und umgekehrt ist also nicht möglich.

Das obige Beispiel illustriert, dass Funktionen auch *rvalue Referenzen* als Parameter erhalten können. Auch spezielle Memberfunktionen von Klassen wie der Konstruktor oder Zuweisungsoperatoren können damit *rvalue Referenzen* entgegennehmen. C++11 definiert hierfür auch weitere spezielle Memberfunktionen, die, falls nicht anders gewollt, explizit generiert werden. So werden automatisch ein *move constructor* und ein *move assignment*-Operator definiert. Diese speziellen Memberfunktionen werden dann genutzt, um die Ressourcen aus *rvalue Referenzen* in ein neues oder bestehendes Objekt zu *bewegen*, was als *move semantic* bezeichnet wird. Dave Abrahams spezifiziert die Aufgabe dieser speziellen Memberfunktionen folgendermaßen [Abr09]:

It's the job of a move constructor or assignment operator to “steal” resources from its argument, leaving that argument in a destructible and assignable state.

Dies ist möglich, da eine *rvalue Referenz* flüchtig ist und diese somit ungültig gemacht werden können. In Abschnitt 1.5 werden die Handhabung dieser und weiterer spezieller Memberfunktionen genauer betrachtet.

1. Der Sprachstandard C++11

Im Falle der Klasse `std::vector`¹⁴ muss also nicht wie bei einer Kopie jedes Element des Containers kopiert werden, sondern nur deren interner Zeiger auf den Zeiger des Felds aus dem *rvalue* Ausdruck gesetzt werden.

Für Funktionen mit *rvalue Referenzen* als Parameter gilt, dass der referenzierte Wert im Laufe der Funktion ungültig werden kann. Will ein Nutzer also die Invalidierung des referenzierten Wertes verhindern, so muss er eine Funktion nutzen, die eine *lvalue Referenz* als Argument entgegen nimmt.

std::move zur Invalidierung Wie oben bereits geschrieben, kann eine *lvalue Referenz* nicht in eine *rvalue Referenz* überführt werden. Falls dies aber gewollt ist, weil zum Beispiel klar ist, dass ein Objekt bald nicht mehr benötigt wird und daher nicht eine Kopie erzeugt sondern die Ressourcen weg vom Objekt bewegt werden sollen, so kann der Nutzer hierfür die Funktion `std::move< T >()` aus der Header-Datei `<utility>` verwenden. Eine Initialisierung wie zum Beispiel `x x2{ std::move(x1) }` sorgt dafür, dass der *move constructor* von `x`, also der Konstruktor mit einer *rvalue Referenz* als Parameter, für `x2` gerufen wird, und nicht der *copy constructor*. Somit können auch im späteren Verlauf eines Programms Ressourcen ohne zu kopieren bewegt werden. [HSK08] nutzt als Beispiel die Implementation von `std::swap`:

```
template <class T> swap(T& a, T& b)
{
    T tmp(a);    // now we have two copies of a
    a = b;       // now we have two copies of b
    b = tmp;     // now we have two copies of tmp (aka a)
}
```

Bereits in diesem kurzen Beispiel wird mehrfach kopiert, obwohl eigentlich nur die Werte von `a` und `b` ausgetauscht werden sollten. Mit C++11 wird folgendes, auch aus [Hin06] stammende, Beispiel ermöglicht:

```
template <class T> swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Zu keinem Zeitpunkt innerhalb der Funktion `std::swap` sind zwei Kopien einer der Variablen `a, b` vorhanden.

Dies führt aber nicht zwingend zu einem Geschwindigkeitsgewinn, wie durch Abbildung

¹⁴Diese Klasse ist eine Hüllklasse für Felder, um deren Speicherverwaltung nicht selbst übernehmen zu müssen. Für weitere Informationen sei auf [Str00, S. 52] verwiesen.

1.2 demonstriert wird. Diese stellt die benötigte Zeit dar, welche zum Vertauschen von N Ganzzahlen unter Verwendung von `std::swap` benötigt wird.¹⁵

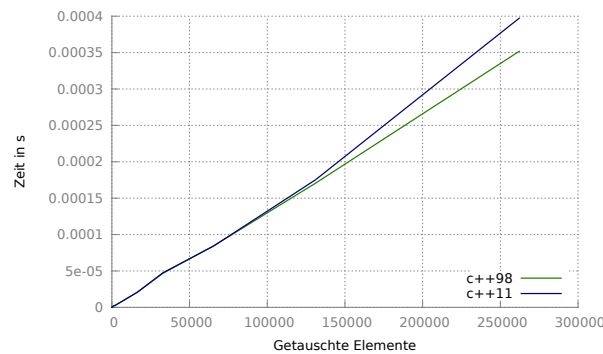


Abbildung 1.2.: Vertauschen von Ganzzahlen durch `std::swap`

Es zeigt sich, dass *move semantic* nicht zwingend schneller als *copy semantic* ist, sondern auch bei einem einfachen Testaufbau langsamer sein kann. Wie sich weiter unten zeigen wird, ist dies aber nicht als repräsentativ anzusehen, da *move semantic* in dieser Messung nicht nur mit der *copy semantic* sondern auch mit Optimierungen durch den Compiler konkurriert. Teilweise kann der Geschwindigkeitsgewinn mithilfe der *move semantic* gravierend sein und gegenüber C++98 eine Beschleunigung um den Faktor zwei oder mehr bedeuten.

Auch im Falle von Geschwindigkeitsverlusten ist *move semantic* sinnvoll, da die Programmausführung damit trotz Optimierungen deterministischer und besser nachvollziehbar wird.

Geschwindigkeitsgewinn mit move semantic Howard E. Hinnant beschreibt in [Hin06] ein einfaches Bewertungsprogramm um *move semantic* *copy semantic* gegenüberzustellen. Die Ausführung des Programms auf einem aktuellen Rechner mit einer CPU Intel Core i5-520M und 4 GB Arbeitsspeicher führt zu starken Laufzeitunterschieden.

Abbildung 1.3 zeigt die für das in Anhang D.3.2 dargestellte Testprogramm benötigte Zeit. Es zeigt sich, dass *move semantic* enorme Geschwindigkeitsgewinne erbringen kann. Dies steht im Kontrast zu Abbildung 1.2, wo kein direkter Gewinn erkennbar war. Hinnant achtet in seinem Testprogramm darauf, Compileroptimierung wie RVO nicht zu erlauben. Dies ist einer der Gründe, weshalb die Unterschiede so stark ausfallen. Zudem ist die verwendete Datenstruktur aufwändiger (ein Vektor von Mengen von Zufallszahlen), wodurch sich Unterschiede stärker ausprägen.

Man kann somit festhalten, dass *move semantic* auch ohne Eingreifen des Programmierers einen

¹⁵Vergleiche dazu auch den zugrunde liegenden Quelltext in Anhang D.3.1.

1. Der Sprachstandard C++11

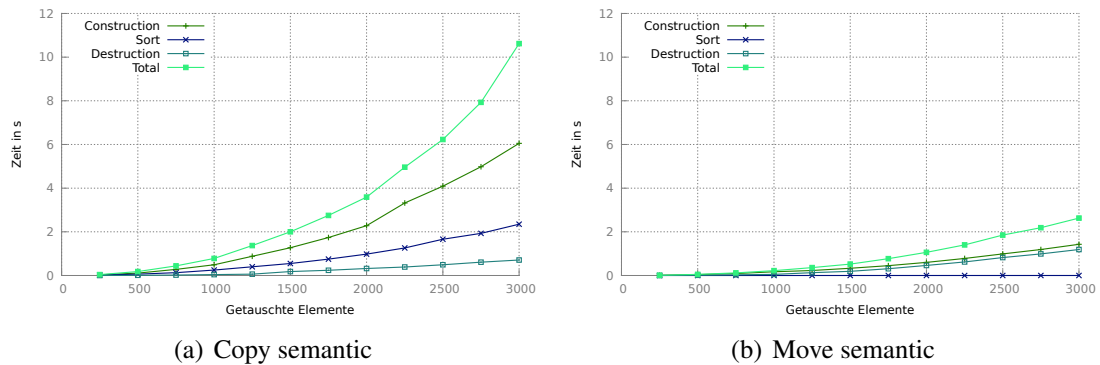


Abbildung 1.3.: Besonders bei Verwendung von Standardalgorithmen auf verschachtelten Strukturen können sich durch move semantic starker Laufzeitunterschiede ergeben. Diese Abbildung zeigt das Ergebnis eines Testlaufs des in Appendix D.3.2 angegebenen Testprogramms. In diesem wird ein Vektor mit mehreren Mengen von Zufallszahlen gefüllt und dann sortiert und rotiert. Gemessen wird die benötigte Zeit um den Vektor zu erstellen, zu sortieren, rotieren, ihn abzubauen und die Gesamtzeit für den Durchlauf. Diese Abbildung zeigt nicht die benötigte Zeit zur Rotation des Vektors, da diese in beiden Fällen jeweils nah 0 Sekunden liegt.

Performanzgewinn bringen kann. Die Auswirkung dieser Änderung auf die Prozesssimulationsbibliothek ODEmx ist jedoch schwer zu erfassen. Wie im Ausblick 3.1 beschrieben, fehlt ein Framework um Performanzänderungen der Bibliothek konsistent erfassen zu können.

1.2.1. Perfect Forwarding

Wie in [HDA02] beschrieben, tritt in C++98/03 ein Problem bei Verwendung generischer Funktionen auf: Nach Ableitung der generischen Parameter können diese nur noch als *lvalue* an andere generische Funktionen übergeben werden. Das Beispiel in Listing ?? illustriert dies. Listing ?? erzeugt folgende Ausgabe:

```
X()
X(const X&)
```

Somit wird zuerst der Standardkonstruktor `X()` aufgerufen und danach der *copy constructor* `X(const X&)`.

In diesem Beispiel wird die Funktion `foo` mit dem Ergebnis des Standardkonstruktors, also mit einem *rvalue* des Typs `x`, aufgerufen. Der nachfolgende Aufruf der Funktion `bar` erfolgt hingegen mit einem *lvalue*, wie an der Ausgabe erkannt werden kann. Somit werden während


```

#include <iostream>

struct X {
    X() {
        std::cout << "X()" << std::endl;
    }
    X( const X& ) {
        std::cout << "X(const X&)" << std::endl;
    }
};

template< class T >
void bar( T t ) { }

template< class T >
void foo( T t )
{
    bar( t );
}

int main()
{
    foo( X {} );
}

```

Listing 1.5: *Rvalue Referenzen degenerieren in C++98 zu lvalue Referenzen.*

der Ausführung des Programms mehrfach unnötige Kopien erstellt, denn der eigentliche Wert, auf dem die generischen Funktionen ausgeführt werden sollen, ist ein *prvalue*. Allgemeiner formuliert schreiben Abrahams, Dimov und Hinant in [HDA02] dazu:

The general form of the forwarding problem is that in the current language,

For a given expression $E(a_1, a_2, \dots, a_n)$ that depends on the (generic) parameters a_1, a_2, \dots, a_n , it is not possible to write a function (object) f such that $f(a_1, a_2, \dots, a_n)$ is equivalent to $E(a_1, a_2, \dots, a_n)$.

Auf das obige Beispiel übertragen ergibt sich also, dass es in C++98 keine Möglichkeit gibt eine Funktion `foo` zu schreiben, die ein *rvalue* Argument als *rvalue* Argument an die Funktion `bar` weiterleitet. Dieses Problem wird als das *forwarding problem* bezeichnet, da es um die Weiterleitung von Typinformation geht. Hieraus ergeben sich aber nicht nur unnötige Kopien. Auch Factoryfunktionen¹⁶ können nur erschwert implementiert werden und Lösung des Problems durch Überladung ist nur unter Verwendung exponentiell vieler überladender Funktionen möglich. [HSK08] gibt einige Beispiele zur weiteren Motivation.

¹⁶Also Funktionen, die Objekte generieren und zurückgeben.

1. Der Sprachstandard C++11

Das forwarding problem besteht laut [HDA02] aus zwei Teilaspekten: Der Weiterleitung von Argumenten und der Weiterleitung des Rückgabetyps. Wir beschäftigen uns im Folgenden mit dem ersten Fall, da der zweite Fall trivial durch Verwendung von *rvalue Referenzen* abgedeckt ist.¹⁷

Die Lösung dieses Problems wird als *perfect forwarding* bezeichnet und durch *rvalue* Referenzen ermöglicht. Der konkreten Lösung liegt eine Änderung der Typableitung bei generischen Funktionen zugrunde: Falls das Argument einer Templatefunktion eine *rvalue* Referenz $T\&\&$ eines Templateparameters T ist, dann wird dieser Parameter beim Aufruf mit einer *lvalue* Referenz in den Typ T überführt (ist also keine *rvalue Referenz* mehr) und damit der Templateparameter T zu $T\&$ abgeleitet. Falls ein *rvalue* als Argument übergeben wird, so wird der Parameter $T\&\&$ nicht angefasst und der Typ T bleibt T . Dieses Verhalten ist in [ISO11, §14.8.2.5p10] spezifiziert und zugegebenermaßen kompliziert nachzuvollziehen. Da die Spezifikation der Ableitung von Templateparametern länglich ist, sei hier nur auf die angegebene Quelle verwiesen. Folgendes Beispiel aus [ISO11, §14.8.2.5p10, S. 388] macht die Auswirkung der Typableitung verständlicher:

```
template <class T> void f(T&&);
template <> void f(int&) { } // #1
template <> void f(int&&) { } // #2
void g(int i) {
    f(i); // f<int&>(int&), i.e., #1
    f(0); // f<int>(int&&), i.e., #2
}
```

Der Typ $T\&\&$ ist der oben beschriebene Parameter in Form einer *rvalue Referenz*. Dieser wird, falls die Funktion mit einem *lvalue* aufgerufen wird, zu T transformiert, und T wiederum dann durch den Referenztyp des *lvalue* abgeleitet. Somit wird beim Aufruf $f(i)$ der generische Parameter $T\&\&$ in den Parameter T überführt und damit T dann zu int abgeleitet. Da für int eine Überladung der Funktion vorhanden ist, wird diese vorgezogen werden.

Die Ableitung von Templatetypen kann nun verwendet werden, um das *forwarding problem* zu lösen. Bei direktem Aufruf einer generischen Funktion, wie im obigen Beispiel, wird bereits die korrekte Funktion verwendet. Somit muss also nur noch der indirekten Aufruf in unserem ursprünglichen Beispiel oben betrachtet werden.

¹⁷Wir erinnern uns, dass der Rückgabewert einer Funktion implizit als *rvalue* betrachtet wird, falls es sich dabei nicht um einen Referenztyp handelt. Damit ist die Weiterleitung des Rückgabetyps trivial, denn falls eine Referenz und damit ein *lvalue* gewollt ist, drückt der Rückgabetypp dies bereits ohne extra forwarding Information aus.

```

template <class T> T&& forward(typename remove_reference<T>::type& t) noexcept
{ return static_cast< T&& >( t ); }
template <class T> T&& forward(typename remove_reference<T>::type&& t) noexcept;
{ return static_cast< T&& >( t ); }

```

Listing 1.6: Forward und move

Beim indirekten Aufruf einer Funktion durch eine andere Funktion wird aus der ursprünglichen *rvalue Referenz* ein *lvalue*. Um dafür zu sorgen, dass der indirekte Aufruf wieder mit einem *rvalue* erfolgt, wird die Funktion `std::forward` aus `<utility>` verwendet. Die Definition dieser Funktion ist denkbar einfach (vergleiche [ISO11, §20.2.3]) und findet sich in Listing 1.6. Diese Funktionen erlauben das Weiterreichen einer *rvalue* oder *lvalue Referenz*, ohne zusätzliche Kopien zu erzeugen.

Trotz ähnlicher Implementation dürfen die Standardfunktionen `std::move` und `std::forward` nicht verwechselt werden. `std::move` dient dazu, grundsätzlich eine *rvalue Referenz* zu bilden, während `std::forward` den Referenztyp erhält.¹⁸ Der Rückgabotyp von `std::forward` wird damit zu dem Typ des Arguments abgeleitet, weshalb weiterhin keine unnötigen Kopien erzeugt werden. Dies wird an einem Beispiel demonstriert.

```

void overloaded( X& x ); // #1
void overloaded( X&& x ); // #2

template< class T >
void forwarder( T&& t ) {
    overloaded( std::forward< T >( t ) );
}

int main () {
    X x;
    forwarder( x ); // calls #1
    forwarder( X{} ); // calls #2
}

```

Listing 1.7: Beispiel für *perfect forwarding*

Hiermit kann *perfect forwarding* erreicht werden. Abhängig vom Referenztyp der Argumente wird durch die generische Funktion die richtige Überladung gewählt. Würde `forward` nicht genutzt, so würden die Aufrufe von `forwarder` immer zum Aufruf von `void overloaded(X&);` führen.

¹⁸Howard Hinnant führt dies in einem Diskussionsbeitrag auf der Mailingliste der GNU Implementation der C++ Standardbibliothek unter <http://gcc.gnu.org/ml/libstdc++/2007-08/msg00037.html> näher aus.

1.3. Automatische Typableitung

Die automatische Ableitung von Typen ist ein lange gefordertes Sprachmerkmal zur Vereinfachung der Programmiersprache C++. Bjarne Stroustrup implementierte laut [Str09, S. 2] bereits im Winter 1982 die automatische Ableitung des Typs einer Variable anhand des Initialisierers durch Verwendung des Schlüsselworts **auto** als Typbezeichner. Er verwarf die Implementierung jedoch, um nicht durch die Neuinterpretation von **auto** eine Inkompatibilität zu C zu erzeugen. Typableitung reduziert explizit anzugebende Typabhängigkeiten und reicht die Aufgabe der Typisierung stärker an den Compiler weiter, der dies aber aufgrund der statischen Typisierung der Sprache bereits tun muss.

Reduktion explizit anzugebender Typen Das Schlüsselwort **auto** wird anhand eines einführenden Beispiels motiviert. Hierzu dient eine einfache Schleife, die mit einem Iterator über einen Vektor von Ganzzahlen iteriert:

```
std::vector< int > v = { 1, 2, 3, 4 }; // #1

for( std::vector< int >::iterator it = v.begin(); it != v.end(); ++it ) // #2
{
    [...]
}
```

Zwischen der Zeile #1 und #2 besteht eine Typabhängigkeit, da der Typ des Iterators `it` vom Typ der Variable `v` abhängt. Sollte der Typ der Variable `v` zu einem anderen Container, zum Beispiel `std::set`, geändert werden, so muss also auch der explizit angegebene Typ der Variable `it` geändert werden. Der Entwickler interessiert sich unter Umständen aber nicht für den eigentlichen Typ des Iterators, solange dieser die geforderten Funktionen anbietet. C++11 bietet über das Schlüsselwort **auto** eine Möglichkeit an, den Typ einer Variable durch den Initialisierer zu definieren und nicht explizit angeben zu müssen. Damit kann obige Schleife vereinfacht werden:

```
std::vector< int > v = { 1, 2, 3, 4 }; // #1

for( auto it = v.begin(); it != v.end(); ++it ) // #2
{
    [...]
}
```

Der Typ des Iterators `v` ergibt sich damit aus dem Rückgabetypp der Funktion `std::vector<int>::begin()` und muss nicht explizit angegeben werden. Sollte nun der Typ des Objekts `v` geändert werden, die Deklaration von `it` nicht angepasst werden, außer die Funktionen `begin()` und `end()` existieren auf dem neuen Typ nicht.

auto kann um **const** erweitert werden, um festzuhalten, dass der abgeleitete Typ konstant ist. Zudem kann auch das kaufmännische Und & zur Bildung einer Referenz ergänzt werden:

```
int a = 10;
const auto& reference = a; // Referenz auf ein konstantes int
```

Wir werden das Referenzieren eines abgeleiteten Typs mit & in 1.4 benötigen, um in einer Menge von Objekten die enthaltenen Objekte ändern zu können.

Typableitung durch einen Ausdruck C++11 definiert **auto** in [ISO11, §7.1.6.2]:

C++11 [§7.1.6.2]

The auto type-specifier signifies that the type of a variable being declared shall be deduced from its initializer or that a function declarator shall include a trailing-return-type.

Oben wurde der erste Teil dieser Definition betrachtet, die Ableitung des Typs einer Variable. Der zweite Teil bezieht sich auf die Angabe des Rückgabewerts einer Funktion. C++11 erlaubt die nachfolgende Definition des Rückgabewerts über

```
auto function( parameters ) -> return-type;
```

Dies ist bei Abhängigkeit des Rückgabetyps von den Parametern oder Templateparametern einer Funktion sinnvoll.

Um den Rückgabewert einer Funktion dynamisch abhängig von Parametern oder Templateparametern abzuleiten, kann ein weiteres neues Sprachmerkmal von C++11 verwendet werden: Die Typableitung anhand eines Ausdrucks mittels eines neuen Schlüsselworts **decltype**.

```
int a = 10;
char b = 20;
decltype( a + b ) sum = a + b;
```

Der Typ der Variable `sum` ist der Typ des Ausdrucks `a + b`. Dies ist semantisch äquivalent zu der Schreibweise `auto sum = a + b;`. **auto** kann aber nur dann verwendet werden, wenn eine Initialisierung vorliegt, oder die Ableitung des Rückgabetyps einer Funktion später erfolgen soll. Falls der Typ eines Ausdrucks abgeleitet werden soll, muss **decltype** ([ISO11, §7.1.6.2p4]) verwendet werden.

Um nun den Rückgabetypp einer Funktion aus einem von Parametern abhängigen Ausdruck

1. Der Sprachstandard C++11

ableiten zu lassen, kann die Syntax `auto function(parameters) -> decltype(operation(parameters));` verwendet werden. Dies erlaubt die Vereinfachung von generischer Funktionen mit variablem Rückgabetyt. Dies erlaubt zum Beispiel folgende Funktion, die zwei Variablen verträglichen Typs addiert.

```
template< class T, class S >
auto add_two( T t, S s ) -> decltype( t + s )
{ return t + s; }
```

In C++98/03 ist eine solche Funktion nur schwerlich unter der Verwendung von Templatemetaprogrammierung zu implementieren, da der Rückgabetyt durch einen *type trait* wie zum Beispiel *common type* gebildet werden kann. Dies ist zwar möglich, erschwert Bibliotheksentwicklern aber die Arbeit und ist nicht trivial.

1.4. Iteration über Mengen

Neben expliziter Typisierung von Objekten kann auch die Verwendung von Iteratoren die Menge an Code in C++98/03 stark steigern. Die Verwendung von Iteratoren ist sinnvoll um die Generizität von Code auszunutzen und möglichst wenig Code doppelt schreiben zu müssen. Stroustrup motiviert dies in [Str95]. Iteratoren sind trotz ihrer Allgemeinheit effizient, da sie typspezifisch sind und gute Grundlagen zur Optimierung bieten. Jedoch kann die Verwendung von Iteratoren für den Nutzer einen Mehraufwand bei der Implementierung bedeuten, besonders falls nur eine Iteration über eine Menge von Objekten durchgeführt werden soll, ohne dass der Inhalt des Iterators selbst dabei eine Rolle spielt. Die in Abschnitt 1.3 dargestellten Möglichkeiten erlauben die vereinfachte Realisierung von Iterationen durch Typableitung der Laufvariable:

```
for( auto i = vector.begin(); i != vector.end(); ++i )
    *i = 10; // initialize all elements with 10
```

In diesem Beispiel besteht keine Abhängigkeit zwischen den verschiedenen Iterationen und der Iterator wird nur als reine Laufvariable verwendet. Nur das durch den Iterator `i` referenzierte Objekt `*i`¹⁹ wird verändert. C++11 erlaubt die Verwendung einer *range-based for*-Schleife, die für solche Fälle eingesetzt wird und keinen Iterator benötigt:

```
for( auto& v : vector )
    v = 10;
```

¹⁹Iteratoren bieten die Methode `T operator*()` an, um das durch den Iterator referenzierte Objekt zu erhalten.

Wie in Abschnitt 1.3 angedeutet, müssen wir hier **auto** um **&** ergänzen, um die Werte zu referenzieren.

Der Vorteil dieser Schreibweise besteht in ihrer Lesbarkeit. Während die explizite Schreibweise einer for-Schleife mit Iteratoren für den Fall der einfachen Vorwärtsiteration viel Code benötigt, sind range-based for-Schleifen kurz und zeigen direkt, in welcher Form die Iteration vonstatten geht. Intern wird aus einer range-based for-Schleife wiederum eine for-Schleife unter Verwendung von Iteratoren gebildet. [ISO11, §6.5.4] stellt diese Transformation dar:

C++11 [§6.5.4]

```
{
    auto && __range = range-init;
    for ( auto __begin = begin-expr, __end = end-expr; __begin != __end;
          ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Die genauen Details hierzu werden nicht zitiert und können in der Originalliteratur nachgelesen werden.

Diese neue Variante der Iteration kann sowohl für Container-Typen T , für Felder mit fester Größe, als auch für Initialisierungslisten verwendet werden. Container-Typen müssen hierfür durch *argument-dependant lookup* eine Spezialisierung von `begin(T)` und `end(T)` anbieten.²⁰ Für Felder fester Größe wird `begin-expr` als `range` definiert, also den Beginn des Arrays, und `end-expr` als `range + bound`, und somit das dem letzten Feldelement folgende Element.

1.5. Umgang mit speziellen Methoden

Der C++ Standard definiert verschiedene spezielle Methoden, die unter den in [ISO03, §12] genannten Bedingungen automatisch durch den Compiler generiert werden müssen. So werden zum Beispiel ein parameterloser *default constructor*, ein *copy constructor* und der *copy assignment*-Operator automatisch generiert. Ein Entwickler kann die Standardimplementation durch eine eigene überschreiben, falls diese nicht ausreicht. Der *default constructor* wird dabei nur dann generiert, wenn kein anderer nutzerdefinierter Konstruktor vorhanden ist.

²⁰C++ nutzt verschiedene Methoden, um den Gültigkeitsbereich eines Namens festzustellen. [ISO11, §3.4] spezifiziert unter anderem das *argument-dependant lookup*, welches in *assozierten Namensräumen* nach einem Namen suchen kann. Die genaue Funktionsweise wird hier ausgelassen und findet sich erklärt in [Str00, S. 177].

A default constructor for a class X is a constructor of class X that can be called without an argument. If there is no user-declared constructor for class X, a default constructor is implicitly declared.

Ähnliche Regeln gibt es auch für die anderen speziellen Methoden, die hier aber nicht wiedergegeben werden sollen. Für Informationen zu diesen kann [ISO03, §12] oder einführende Literatur zu C++ wie [Str00] herangezogen werden.

C++11 ergänzt die obige Liste der implizit definierten speziellen Memberfunktionen aufgrund der neuen *rvalue Referenzen* um zwei weitere: *move constructor* und den *move assignment-Operator*. Somit werden dem Entwickler nun automatisch unterschiedliche spezielle Memberfunktionen zur Verfügung gestellt, die nur bei Bedarf selbst implementiert werden müssen. Die Definition einer dieser speziellen Methoden durch den Nutzer darf aber nur erfolgen, wenn die entsprechende Methode für diese Klasse auch durch den Nutzer deklariert wurde. Enthält also die Klasse `x` nicht die Signatur des Standardkonstruktors `x()`, so darf dieser auch nicht außerhalb der Klasse definiert werden.

1.5.1. Explizite Nutzung oder Löschung impliziter Methoden

C++98 bietet kein Sprachmittel an, um explizit die Nutzung der implizit generierten Methoden zu veranlassen, und auch nicht um diese speziellen Methoden explizit nicht generieren zu lassen. Das zweite Problem lässt sich syntaktisch in C++98 lösen: Wie oben angesprochen, wird eine spezielle Methode nur dann generiert, wenn deren Signatur nicht bereits durch den Nutzer angegeben wurde. Dies kann nun durch Verwendung des Zugriffsattributes **private**²¹ ausgenutzt werden. Indem die spezielle Methode als privat deklariert wird, kann sie nur durch Objekte der Klasse selbst genutzt werden. Da diese dies nicht tun, kann die Definition der Methode ausgelassen werden. Damit wird der Zugriff auf diese Methode verhindert. Der Compiler generiert die spezielle Methode wiederum nicht, da er eine Definition an anderer Stelle erwartet. Sollte die Methode trotzdem durch ein Objekt der Klasse genutzt werden, so führt dies zu einem *Linkerfehler*.

Dies ist der übliche Weg in C++98, um Methoden zu löschen. C++11 bringt mit dem Schlüsselwort **delete** ein neues Sprachmerkmal, um eine Methode explizit zu löschen: [ISO11, §8.4.1].

```
class X {
```

²¹Diese Zugriffsattribute werden in [Str00, S. 225] eingeführt. Mit ihnen kann festgelegt werden, welche Objekte auf welche Namen einer Klasse zugreifen dürfen. **private** lässt nur den Zugriff durch ein Objekt der Klasse selbst zu.


```
public:
    X() = delete; // #1 loesche den Standardkonstruktor
};
```

Durch #1 wird der Defaultkonstruktor explizit gelöscht. Diese Syntax ähnelt der Deklaration einer *pure virtual function*, also einer polymorphen Funktion ohne Implementation.

Die gleiche Syntax wird in C++11 verwendet, um auch das verbleibende und in C++98 nicht gelöste Problem der expliziten Generierung von speziellen Methoden zu lösen:

```
class Y {
public:
    Y() = default; // #1 Biete explizit den Standardkonstruktor an
    Y(int); // #2 Biete einen Konstruktor an, der eine Ganzzahl erwartet
};
```

Durch #1 wird der Defaultkonstruktor generiert, obwohl mit #2 ein anderer durch den Nutzer bereitgestellter Konstruktor vorhanden ist.

Löschen geerbter Funktionen Während sich das Attribut `default` nur auf spezielle Methoden bezieht [ISO11, §8.4.2p1], kann das Attribut `delete` laut [ISO11, §8.4.3] zum Löschen einer beliebigen Methode verwendet werden. Dies kann genutzt werden, um eingerbte Methoden zu löschen:

```
struct Y {
    void foo() {}
};

struct X : public Y {
    void foo() = delete;
};

int main() {
    X x;
    x.foo(); // error: use of deleted function 'void X::foo()'
}
```

Listing 1.8: Löschen geerbter Methoden

Auch eine Methode, die nicht in der Vererbungshierarchie der Klasse auftaucht, kann gelöscht werden. Dies scheint jedoch wenig sinnvoll zu sein, da dann auch in dieser Klasse aufgrund der *one definition rule*²² keine Implementation dieser Methode mehr erfolgen kann.

²² Die one definition rule, oder auch *odr*, ist in [ISO11, §3.2] definiert:

No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.

1.6. Eine Konstante für den Nullzeiger

C++98 spezifiziert keine spezielle Konstante für den Nullzeiger, also für den Inhalt eines Zeigers, der nicht auf eine valide Speicherstelle zeigt. Um den Nullzeiger darzustellen wird wie in C99 die Ganzzahl `0` verwendet [SS03]. Somit wird `0` unterschiedlich verwendet, was zu einem großen Problem führt [SS03, S. 1]: `0` und der Nullzeiger können bei Überladung von Funktionen nicht unterschieden werden. Somit ist für zwei Funktionen `void foo(int)` und `void foo(int*)` beim Aufruf `foo(0)` nicht klar, welche aufgerufen werden soll. Daher wird von vielen Entwicklern seit langem eine spezielle Konstante für den Nullzeiger gefordert, um die Sprache einfacher erlernbar zu machen. Zwar existiert mit dem Makro `NULL` schon eine durch den Namen erkennbare Konstante, jedoch ist diese nicht typsicher. Das Einführen einer Konstante für den Nullzeiger soll generische Programmierung vereinfachen und generell die Implementation von Bibliotheken vereinfachen, da ein Nutzer sich keine Gedanken wegen dem Problem der Überladung machen muss.

Der Standard sieht vor, dass die Nullpointerkonstante `nullptr` ein neues Schlüsselwort ist und von einem Typ ist, der sowohl zur Ableitung eines Templatearguments als auch zur Überladung von Templates verwendet werden kann [SS03, S. 2]. `nullptr` darf hingegen nicht in einem arithmetischen Ausdruck verwendet, einer Ganzzahl zugewiesen oder mit einer Ganzzahl verglichen werden. Zudem darf `nullptr` nicht in irgendeinen anderen Typ konvertiert werden.

Der Typ von `nullptr` ist spezifiziert als ein neuer Datentyp `nullptr_t`. `nullptr` ist die einzig legale Instanz dieses Datentyps. Der Standard definiert den Typ `nullptr_t` daher auch über die Konstante `nullptr`:

C++11 [§18.2p9]

`nullptr_t` is defined as follows:

```
namespace std {  
    typedef decltype(nullptr) nullptr_t;  
}
```

1.7. Stark typisierte Aufzählungen

C++98 bietet die aus C übernommenen Aufzählungstypen `enum` an. Ein Nutzer der Sprache kann damit einen Typ definieren, dessen Instanzen als Wert eine der vordefinierte Konstanten

aus einer Menge annehmen dürfen. In nachfolgendem Beispiel wird ein Aufzählungstyp `enum Color` definiert, dessen Instanzen einen der Werte `Red`, `Blue`, `Green` annehmen können.

```
enum Color { Red, Blue, Green };
Color a = Red;
Color b = Green;
if( a == b ) std::cout << "Equal" << std::endl;
else std::cout << "Not Equal" << std::endl; // Ausgabe wird "Not Equal" sein
```

Listing 1.9: Beispiel zu einem C-style enum

Die Variable `a` vom Typ `Color` kann den Wert `Red` annehmen. Der Variablen `b` wird `Green` zugewiesen und aufgrund des Tests auf Gleichheit wird die Zeichenkette `Not Equal` ausgegeben.

Aufzählungen (im Folgenden als *enum* abgekürzt) bringen in ihrer ursprünglichen Variante drei Probleme mit sich, die in [EMSS07] aufgeführt werden:

1. enums werden implizit nach `int` konvertiert.
2. Die in einem enum definierten Konstanten werden in den umgebenden Namensraum exportiert, was zu Namenskonflikten führen kann.
3. Der einem enum zugrunde liegende Typ ist nicht spezifiziert, weshalb Portabilität zum Problem werden kann und Vorwärtsdeklarierung unmöglich wird.

C++11 geht dieses Problem durch Einführung von *scoped enumerations* an. Dieses neue Konzept wird als `enum class` bezeichnet [EMSS07, S. 6] und zeichnet sich mit der einhergehenden starken Typisierung als Lösung für obige Probleme aus. Nachfolgend illustriert ein Beispiel die Verwendung. Eine ausführlichere Motivation der angegebenen Probleme kann im Proposal [EMSS07] nachgelesen werden. Die technische Spezifikation findet sich im Standard [ISO11, §7.2].

```
int A;

enum class E : int; // #1 Vorwaertsdeklaration ist moeglich

enum class E { A, B, C }; // #2 Alte enums bringen Namenskollisionen
E e = E::A; // #3 Konstanten werden durch scope resolution aufgeloeset
if( e > 1 ) { ... } // #4 Eine implizite Konvertierung zu int schlaegt fehl
```

Listing 1.10: Stark typisierte enums beheben Probleme der bisherigen Aufzählungstypen.

Wie an #1 zu sehen, kann durch Angabe des zugrunde liegenden Basistyps eine Vorwärtsdeklaration durchgeführt werden. #2 demonstriert das Zuweisen der Konstanten an `e`. #2 stellt dar, dass es nicht mehr zu Namenskonflikten mit Symbolen im gleichen Namensraum kommt. Ein `enum class` bildet einen neuen Namensraum, der durch den *scope resolution operator* erreicht wird (#3). Eine implizite Konvertierung zu `int` wird nicht mehr durchgeführt, obwohl der Basistyp auch `int` ist.

1.8. Variadic Templates

Variadic templates sind ein weiteres neues Sprachmerkmal, das keine direkte Entsprechung in C++98 hat und auch nicht in der Standardbibliothek implementiert werden kann. *variadic templates* sind Templates, die eine beliebige Anzahl von Templateparametern annehmen können. In seiner Einführung [Gre06] zu *variadic templates* führt Douglas Gregor als Beispiele zur Nutzung die Implementation einer typsicheren `printf` Funktion an, sowie die Möglichkeit, einen verallgemeinerten Typ `tuple` zu implementieren. Diese Beispiele sollen im Folgenden auch genutzt werden, um die Verwendung der *variadic templates* zu illustrieren. Dieses neue Konzept wird in ODEMx zur Implementation der Methode `Process::wait` genutzt, wie in Abschnitt 2.3 dargestellt werden wird.

1.8.1. Eine typsichere `printf` Funktion

Textausgabe auf die Standardausgabe erfolgt in C üblicherweise durch die Funktion `printf` aus der Standardein-/ausgabebibliothek `<stdio.h>`. Die Signatur dieser Funktion zeigt an, dass sie mit beliebig vielen Parametern aufgerufen werden kann: `int printf(const char *format, ...)`; . Die Auslassungspunkte `...` erlauben den Aufruf mit beliebig vielen Parametern, wobei diese über Macros aus `<stdarg.h>` ausgelesen werden können. Der Zeiger `format` zeigt auf einen C-String, der Formatierungszeichen und Platzhalter enthält. Platzhalter werden durch Argumente aus der variablen Argumentenliste befüllt. So gibt zum Beispiel der Platzhalter `%s` in `printf("%s ", string);` an, dass an der ersten Stelle der variablen Argumentenliste eine Zeichenkette erwartet wird, die durch das Nullbyte `\0` terminiert wird. Der Typ eines Arguments in der variablen Argumentliste muss jedoch nicht durch den Compiler geprüft werden. Daher führt das nachfolgende Programm zu einem Speicherzugriffsfehler:

```
int main() {  
    printf("%s", 10);  
}
```

Da `10` keine nullterminierte Zeichenkette ist, wird solange gelesen, bis ein Nullbyte gefunden wurde. Dies führt in diesem Fall zu einem Speicherzugriffsfehler, da Speicherbereiche gelesen werden, die nicht gelesen werden dürfen. Gregor löst dieses Problem in [Gre06] durch eine *variadic template* Funktion, welche nur das Problem möglicher Speicherzugriffe lösen will.

```
template<typename T, typename... Args>  
void printf(const char* s, const T& value, const Args&... args) {  
    while (*s) {  
        if (*s == '%' && *++s != '%') {
```

```

        // ignore the character that follows the '%': we
        // already know the type!
        std::cout << value;
        return printf(++s, args...);
    }
    std::cout << *s++;
}
throw std::runtime_error("extra arguments provided to printf");
}

```

Listing 1.11: Implementation von `printf()` mittels eines *variadic templates*

Die Funktion `printf` durchläuft den Formatierungsstring und gibt jedes Zeichen aus, bis zum ersten mal ein Platzhalter gefunden wird (**if**-Anweisung). Dieser wird verworfen und durch den entsprechenden Wert `value` vom Typ `const T&` ersetzt. Nach der Ersetzung erfolgt ein rekursiver Aufruf der Funktion.

Wie im obigen Beispiel zu sehen war, nutzt C++11 ... zur Definition von *variadic templates*, welche in [ISO11, §14.5.3] definiert sind. Durch `typename ... Args` wird eine Typliste für das Template definiert. Dieses Konstrukt wird als *template parameter pack* bezeichnet und akzeptiert null oder mehr Typargumente. Dem ähnelnd ist das Konstrukt `const Args&... args`, welches als *function argument pack* bezeichnet wird und der Deklaration von Parametern dient. Zusammengefasst werden diese zwei Formen unter dem Begriff *parameter pack*. Nun verbleibt nur noch der Ausdruck `args...` unerklärt. Dies wird als *pack expansion* bezeichnet [ISO11, §14.5.3p4], die rechts des zu entpackenden *pack* stehen muss:

C++11 [§14.5.3p4]

A pack expansion is a sequence of tokens that names one or more parameter packs, followed by an ellipsis. The sequence of tokens is called the pattern of the expansion; its syntax depends on the context in which the expansion occurs.

Erfolgte also der Aufruf der Funktion `printf` mit `printf("a = %d, b = %d", 10, 20)` so ist der nächste rekursive Aufruf der Funktion dann `printf(", b = %d", 20)`.

1.8.2. Tuple in C++11

Variadic templates können auch genutzt werden, um einen allgemeinen Tuple-Typ zu implementieren. Ein Tuple ist eine geordnete Folge mit einer festen Anzahl von Elementen gleichen oder unterschiedlichen Typs. Als einfaches Beispiel eines Tuples kann ein C-**struct** gesehen werden.

1. Der Sprachstandard C++11

In C++98 muss die Anzahl der Templateargumente einer Templateklasse festgelegt sind. Aus diesem Grund kann in C++98 ein allgemeiner Tuple-Typ nur unter Verwendung von Lisp-ähnlichen Listen implementiert werden.²³ Durch variadic templates können Tuple und andere Typen mit variabler Anzahl von Templateargumenten einfach implementiert werden.

```
template< class ... Types > struct tuple;

template<> struct tuple<> {}; // last element in a list of tuples is empty

// recursive type
template< class T, class ... Tail > struct tuple< T, Tail... > {
    T value; // contained value in this index
    tuple< Tail... > tail; // remaining types

    tuple( T t, Tail... ts ) // constructor
        : value{ t }
        , tail{ ts... }
    {}
};

int main() {
    tuple<int, char, double> t {10, 'c', 1.5d};
}
```

Listing 1.12: Implementation eines generischen Tupletyps in C++11.

C++11 erweitert diese einfache Definition in [ISO11, §20.4] und führt auch Methoden ein, um auf die Werte innerhalb des Tuples zugreifen zu können:

```
#include <tuple>
#include <iostream>

int main() {
    std::tuple<int, char, double> t {10,'c',1.5d};

    std::cout << std::get< 1 >( t ) << std::endl; // output: c
}
```

Wie an dem einfachen Beispiel der Tuple ersichtlich wird, können *variadic templates* genutzt werden, um rekursive Datentypen zu bilden, die aus funktionalen Programmiersprachen wie zum Beispiel Haskell bekannt sind. Mit Hilfe von *type traits*²⁴ ergibt sich hieraus eine mächtige Kombination, die auch die Programmierung generischer Bibliotheken vereinfacht.

²³Dies lässt sich durch Rekursion über Templateargumente lösen, indem Listen anhand der aus Lisp bekannte Cons-Operation aufbaut, über die dann iteriert werden kann. Alexandrescu beschreibt dies in [Ale01, S. 49]

²⁴Also Templates, die genutzt werden, um Eigenschaften eines Templateparameters zu bestimmen.

1.9. Hülltypen für Zeiger

C++ erlaubt die Überladung der Operatoren `operator->` und `operator*` zur Bildung von Hülltypen, die eine Verwendung ähnlich eines Zeigers ermöglichen, aber bei der eigentlichen Dereferenzierung weitere Aktionen durchführen können. Stroustrup bezeichnet diese Hülltypen in [Str00, S. 289] als *smart pointers*, also Typen, deren Instanzen sich wie Zeiger *verhalten* und zusätzliches Verhalten mitbringen. Ein klassischer Datentyp dieser Kategorie ist `std::auto_ptr< T >`. Dieser templatebasierte Typ dient zur Kapselung eines Zeigers `T*`, wobei durch der enthaltene Zeiger durch Kopieren invalidiert wird.

Die typische Verwendung von *smart pointers* ist die Verwaltung eines Zeigers auf selbstverwalteten Speicher und Freigabe diesen Speichers, sobald die Instanz des Hülltyps ihren Gültigkeitsbereich verlässt. Mit dem *Technical Report 1* [ISO05]²⁵ wurden neben `std::auto_ptr` noch `std::shared_ptr` zur Referenzzählung und `std::weak_ptr` zum Verweis auf `std::shared_ptr` Instanzen hinzugefügt. C++11 ersetzt den Typ `std::auto_ptr` durch einen neuen Typ `std::unique_ptr`. Die Motivation für diesen Schritt soll im folgenden Abschnitt gegeben werden. Informationen zur Verwendung der anderen *smart pointers* und Motivation für diese finden sich in der Dokumentation zu der C++-Bibliothek *boost* in [CDDA09], welche die Grundlage zu den *smart pointers* in TR1 bilden.

`std::auto_ptr` ist veraltet Der *smart pointer* `std::auto_ptr` dient der Verwaltung von Objekten auf der Halde, deren Speicherplatz durch den Nutzer alloziiert wurde:

C++11 [§D.10.1]

The class template `auto_ptr` stores a pointer to an object obtained via `new` and deletes that object when it itself is destroyed (such as when leaving block scope 6.7).

Ein Zeiger auf Speicher darf höchstens einmal freigegeben werden. Um dies zu gewährleisten, verhindert die Implementation von `std::auto_ptr` das Kopieren des im *smart pointers* enthaltenen Zeigers durch Überschreiben des *copy constructors*:

```
std::auto_ptr< int > ptr( new int( 10 ) );
std::auto_ptr< int > cpy = ptr; // #1 Invalidiert ptr
```

In #1 wird keine Kopie von `ptr` erzeugt, sondern der durch `ptr` verwaltete Zeiger auf eine Ganzzahl wird an `cpy` *weitergereicht*. Der interne Zeiger von `ptr` zeigt danach auf `nullptr`²⁶ und kann nun nicht mehr dereferenziert werden. Der `std::auto_ptr ptr` wird somit *invalidiert*.

²⁵Im Folgenden wird dies mit *TR1* abgekürzt.

²⁶Die Operation `delete` auf 0 oder `nullptr` funktioniert immer.

1. Der Sprachstandard C++11

Die Semantik des Kopieren eines `std::auto_ptr` entspricht damit der *move semantic* und nicht der *copy semantic*. Dies ist kann bei der Verwendung von Standardcontainern und Standardalgorithmen problematisch sein, wie Howard E. Hinnant in [Hin05] beschreibt:

With such a design, one could put `auto_ptr` into a container:

```
vector<auto_ptr<int> > vec;
```

However field experience with this design revealed subtle problems. Namely:

```
sort(vec.begin(), vec.end(), indirect_less());
```

Depending upon the implementation of sort, the above line of reasonable looking code may or may not execute as expected, and may even crash! The problem is that some implementations of sort will pick an element out of the sequence, and store a local copy of it.

```
...  
value_type pivot_element = *mid_point;  
...
```

*The algorithm assumed that after this construction that `pivot_element` and `*mid_point` were equivalent. However when `value_type` turned out to be an `auto_ptr`, this assumption failed, and subsequently so did the algorithm.*

Hinnant folgert hieraus:

One should not move from lvalues using copy syntax. Other syntax for moving should be used instead. Otherwise generic code is likely to initiate a move when a copy was intended.

`std::auto_ptr` verstößt gegen diese Regel und ist damit unsicher. Als Ersatz dient der neue Typ `std::unique_ptr`, der das Verhalten von `std::auto_ptr` nachahmt, aber keine Kopie erlaubt. `std::unique_ptr` bietet nur die Initialisierung durch *move semantic*, nicht aber durch *copy semantic* an.

1.10. Weitere neue Sprachmerkmale

Die in diesem Abschnitt angesprochenen Sprachmittel und Erweiterungen der Standardbibliothek erfassen nur einen eingeschränkten Bereich aller Neuerungen von C++11. So konnte zum

Beispiel nicht auf das neue Schlüsselwort **constexpr** eingegangen werden, mit dem beliebige, zur Compilezeit auswertbare, Ausdrücke als konstant gekennzeichnet und zur Berechnung anderer Ausdrücke zur Compilezeit verwendet werden können. Um die ausgelassenen Merkmale nicht komplett zu ignorieren, wird im Kapitel 3 in Abschnitt 3.1 ein kurzer Ausblick über einige ausgelassene Sprachmerkmale geworfen, die für eine spätere Arbeit sinnvoll sein könnten. Der Anhang A enthält eine tabellarische Übersicht der ausgelassenen Neuerungen inklusive Kurzbeschreibungen zu Motivation und Umsetzungsstand, die auf [Str11a] und [GCC11] zurückgeht.

2. Verwendung der neuen Features in ODEMX

In diesem Abschnitt werden einige Fallbeispiele analysiert, die von C++11 profitieren könnten. Diese Beispiele stammen aus der Prozesssimulationsbibliothek ODEMX und wurden mit Blick auf eine mögliche Verwendung neuer Sprachmerkmale gewählt. Im Ausblick in Abschnitt 3.1 werden weiter überarbeitungswürdige Teile der Bibliothek angesprochen, die ebenfalls von C++11 profitieren könnten aber hier aufgrund des eingeschränkten Raums der Arbeit ausgelassen wurden.

2.1. Das PortT Warteschlangenkonzept in ODEMX

Die Prozesssimulationsbibliothek ODEMX bietet verschiedene Datenstrukturen zur Synchronisation von Prozessen an. Eine dieser Datenstrukturen, genannt *Port*, dient der Kommunikation zwischen Prozessen und ist dem aus Unix bekannten *pipe*-Konzept nachmodelliert. Ein Port ist ein unidirektionaler, asynchroner Kommunikationskanal, über den Objekte eines bestimmten Typs geschickt und entnommen werden können. Der Puffer des Kanals ist größenbeschränkt, wobei die maximale Belegung nutzerdefiniert bei Konstruktion eines Port-Objekts festgelegt werden kann.

ODEMX-Prozesse können an solch einem Kanal warten. Produzenten warten am Endstück des Kanals `PortTail`, während Konsumenten am Kopfstück des Kanals `PortHead` warten können. Durch diese Wartevorgänge findet eine lose Synchronisation zwischen Produzenten und Konsumenten statt.

Nicht immer ist ein Wartevorgang an einem Kommunikationskanal gewollt. Daher bietet das Port-Konzept drei Modi für eine Port-Instanz an:

Waiting mode ist der oben beschriebene Modus, in dem Prozesse an einem Kanal warten. Produzenten warten, falls kein Platz zum Ablegen eines Objekts im Kanal vorhanden ist, und Konsumenten warten, falls kein Objekt zur Entnahme bereitsteht. Es ist zu beachten, dass nur Instanzen von Subklassen der Klasse `Process` an einem Portobjekt warten können.

2. Verwendung der neuen Features in ODEMX

Sollten Objekte anderer Klassen, zum Beispiel Instanzen der Klasse `Event`, zum Warten gezwungen werden, so führt das zu einer Fehlermeldung.

Zero mode In diesem Modus wird `0` zurückgeliefert, falls ein Produzent kein Element im Kanal ablegen oder ein Konsument nicht direkt ein Objekt aus dem Kanal entnehmen kann.

Error mode Dieser Modus verhält sich wie der *zero mode*, gibt aber im Falle von nicht vorhandenem Platz oder bei Fehlen eines zu entnehmenden Objekts eine Fehlermeldung über einen der Fehlerkanäle aus.

Das Portkonzept wurde bereits in der einer Vorgängerbibliothek von ODEMX, genannt ODEM, eingeführt. Die Implementation findet sich in [FA96, S. 213]. Diese konnte nur Zeiger auf Objekte der ODEM-Klasse `Elem` beziehungsweise Zeiger auf Objekte von Ableitung der Klasse `Elem` aufnehmen. Die heutige Implementation basiert auf der ursprünglichen Implementation und umgeht die Restriktionen durch Verwendung von C++-Templates. Der folgende Unterabschnitt erklärt die aktuelle Implementation näher. Abschnitt 2.1.2 motiviert anschließend, weshalb die aktuelle Implementation in ODEMX überarbeitet werden muss.

2.1.1. Implementation des Kommunikationspuffers

Die Implementation des Portkonzepts gliedert sich, wie in Abbildung 2.1 dargestellt, in drei Teilklassen auf:

- `PortHeadT< T >` ist das Ende des Kommunikationspuffers, über das Objekte durch Ruf von `get()` Elemente aus dem Kanal entnommen werden können. Die Klasse bietet eine Methode `PortTailT< T> getTail()` an, um den Anfang des Kanals zu erhalten.
- `PortT< T >` ist der interne Puffer, also der eigentliche Kanal. Er ist vor dem Nutzer verborgen und implementiert die Details des Kommunikationskanals.
- `PortTailT< T >` ist der Eingang des Kanals. Über die Methode `put(T)` können Objekte im Puffer hinterlegt werden. Zudem bietet diese Klasse eine Methode `PortHeadT< T > getHead()` an, um das Ende des Kommunikationskanals zu erhalten.

Die Elemente des Puffers werden in einer durch den Templateparameter `T` parametrisierten Liste abgelegt. Nutzer des Portkonzepts interagieren bei der Verwendung nur mit den Schnittstellen `PortHeadT< T >` und `PortTailT< T >`, wie im nachfolgenden Beispiel dargestellt wird.

2.1. Das PortT Warteschlangenkonzzept in ODEmX

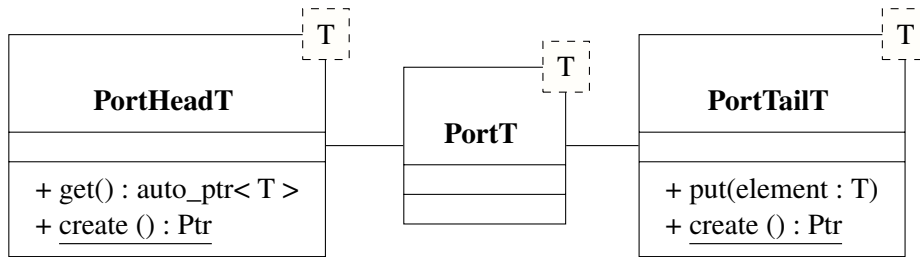


Abbildung 2.1.: PortT< T > und die Nutzerschnittstellen PortHeadT< T > und PortTailT< T >.

```

#include <odemx/odemx.h>
#include <iostream>
using namespace odemx::synchronization;

int main()
{
    auto output = PortHeadT< int >::create( odemx::getDefaultSimulation(), "Port",
        PortMode::ZERO_MODE ); // #1
    auto input  = output->getTail();

    input->put(10); // #2 Gebe das Element in den Puffer
    std::auto_ptr< int > out = output->get(); // #3 Hole das Element aus dem
        Puffer

    std::cout << out->get() << std::endl; // #4 Gebe das Objekt auf der
        Standardausgabe aus

    return 0;
}

```

Listing 2.1: Ein Beispiel zur Verwendung der Portimplementation.

Der Nutzer fügt mit der Methode `put` ein Element in den Anfang `PortTailT< T >` des Puffers ein (#2) und erhält durch Aufruf von `PortHeadT< T >::get()` ein Element aus dem Puffer (#3). Einzig die Konstruktion des Puffers (#1) ergibt sich nicht aus der bisherigen Beschreibung. Ein Port wird durch die Funktionen `PortHeadT< T >::create` bzw. `PortTailT< T >::create` erstellt. Die Konstruktoren der Klassen können nicht direkt verwendet werden, da der Zusammenhang der drei Klassen `PortHead< T >`, `PortTail< T >` und `Port< T >` hergestellt werden muss und somit die Konstruktion einer Instanz einer der drei Klassen die Konstruktion von Instanzen der anderen beiden Klassen nach sich zieht. Die Assoziation der Klassen erfolgt über *smart pointer*, wie durch die Assoziationspfeile aus Abbildung 2.1 angedeutet wird. Durch die Verwendung von *smart pointer* müssen die assoziierten Objekte bei Freigabe eines der Objekte nicht selbst abgeräumt werden; dies wird durch die explizite Assoziation automatisch ausgeführt. Aus der Verwendung der *smart pointer* ergibt sich der Bedarf einer *factory*-Funktion anstatt direkter Verwendung der entsprechenden Konstruktoren: Die zirkuläre Abhängigkeit zwischen den Objekten, wie durch 2.2(a) dargestellt, verhindert das Erkennen von `PortHeadT` und `PortTailT`

2. Verwendung der neuen Features in ODEMX

als *garbage*, da `PortT` weiterhin eine Referenz auf diese enthält. Somit können nicht nur *smart pointer* der Klasse `std::shared_ptr` verwendet, sondern müssen auch *lose Referenzen* durch den Typ `std::weak_ptr< T >` genutzt werden. Diese brechen wie in Abbildung 2.2 dargestellt die zirkuläre Abhängigkeit auf und führen nicht zur Inkrementierung der Referenzzähler.

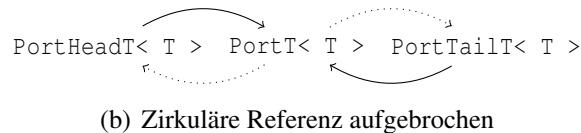
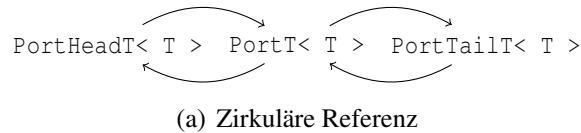


Abbildung 2.2.: Zirkuläre Referenzen verhindern bei Verwendung von Referenzzählern das Abräumen von *garbage*. Die gepunkteten Pfeile in der Abbildung sind einfache Zeiger oder `std::weak_ptr`, während die anderen Pfeile Assoziationen unter Verwendung von `std::shared_ptr` darstellen, die den referenzierten Speicher durch einen Zähler verwalten. Objekte vom Typ `std::weak_ptr` sind lose Referenzen und ähneln damit in ihrer Semantik einfachen Zeigern. Sie können zudem in `std::shared_ptr` überführt werden, um den referenzierten Speicher zu verwalten.

Da ein `std::weak_ptr` durch ein `std::shared_ptr` initialisiert werden muss und während der Konstruktion eines Objekts noch kein `std::shared_ptr` auf dieses Objekt vorhanden sein kann¹, muss eine *factory*-Funktion zur Generierung der Instanz genutzt werden.

2.1.2. Mängel der Portimplementation

In diesem Abschnitt werden zwei Probleme der aktuellen Portimplementation vorgestellt, welche die Verwendung unnötig erschweren. Beide Probleme werden beschrieben und anhand von Beispielen motiviert.

Das erste Problem ist eine Anforderung an den verwendeten Templateparametertyp: Dieser muss *CopyConstructible* sein, also den *copy constructor* anbieten. Dies ist in Hinsicht auf die in C++11 eingeführte *move semantic* problematisch und erweist sich als unnötige Einschränkung.

Das zweite Problem ist der Rückgabetypp von `PortHeadT::get()`. Dieser ist ein `std::auto_ptr` und muss bei Verwendung eines Zeigertypen als Templateparameter doppelt dereferenziert

¹Vergleiche hierzu die Dokumentation von `Boost::weak_ptr`: http://www.boost.org/doc/libs/1_47_0/libs/smart_ptr/weak_ptr.htm, abgerufen am 7. August 2011

werden, um den enthaltenen Zeiger zu erhalten. Dies stellt besonders für Neulinge eine ungewöhnliche Hürde dar und kann zu unnötiger Frustration führen.

In den folgenden Abschnitten demonstrieren wir diese Probleme anhand von Beispielen. In Abschnitt 2.1.3 werden wir eine Lösung vorstellen, die bei näherer Betrachtung ohne Performanzverlust aber mit deutlichem Nutzbarkeitsgewinn implementiert werden kann.

Der Templateparameter muss CopyConstructible sein

Die aktuelle Portimplementation erlaubt nicht die Verwendung eines Typs ohne verfügbaren *copy constructor*. Damit können keine Typen verwendet werden, deren Implementation eine bestimmte Restriktion wie zum Beispiel die nicht-Kopierbarkeit widerspiegeln soll. Das nachfolgende Beispiel erzeugt aus diesem Grund einen Fehler:

```
#include <odemx/odemx.h>
using namespace odemx::synchronization;

class NotCopyConstructible {
public:
    NotCopyConstructible() {}
    NotCopyConstructible( const NotCopyConstructible& ) = delete;
    NotCopyConstructible( NotCopyConstructible&& ) {}
};

int main()
{
    auto input = PortTailT< NotCopyConstructible >::create(
        odemx::getDefaultSimulation(), "Port", PortMode::ZERO_MODE );
    input->put( NotCopyConstructible {} ); // #1 ERROR
    return 0;
}
```

Listing 2.2: Nicht-kopierbare Typen können nicht in einem Port abgelegt werden.

Für die mit #1 markierten Zeile wird ein Fehler ausgegeben: Datentypen, die nur verschoben werden können, dürfen keine Templateparameter des Kommunikationspuffers sein. Dies ist eine unnötige Einschränkung, da der dem Port zugrunde liegende Datentyp² aus der Standard Template Library diese Restriktion nicht aufweist.

Die Verwendung von Zeigertypen ist unhandlich

Wenn wir uns nochmals dem einführenden Beispiel auf Seite 37 zuwenden, dann fällt die bisher nicht angesprochene Markierung #4 auf:

²Aktuell ist dies eine einfache Liste `std::list`.

2. Verwendung der neuen Features in ODEMX

```
std::cout << out->get() << std::endl; // #4 Gib das Objekt auf der
    Standardausgabe aus
```

Wie im Diagramm 2.1 dargestellt ist der Rückgabewert der Methode `PortHeadT<T>::get` nicht `T`, sondern ein `std::auto_ptr< T >`. Falls der Templateparameter `T` ein Zeigertyp `Y*` ist, ergibt sich als Rückgabetypp der Methode `std::auto_ptr< Y* >`. Ein Aufruf der Funktion `std::auto_ptr< T >::get` liefert einen Zeiger auf den enthaltenen Wert zurück, in diesem Falle also einen Zeiger auf `Y*` und damit `Y**`. Dies führte seit der Einführung der Implementation immer wieder zu Verständnisproblemen bei Nutzern, die über diesen Aufruf als Rückgabe den enthaltenen Zeiger erwarten. Somit resultiert aus der Dereferenzierung des Rückgabetypps auch keine Referenz `Y&`, sondern wiederum ein Zeiger `Y*`.

2.1.3. Lösungsimplementation

Zwei Anforderungen sind die Grundlage der Lösung obiger Probleme, die zwingend eingehalten werden sollten. So muss eine Implementation auch weiterhin anzeigen können, ob ein wartender Prozess unterbrochen wurde. Ob dies mittels des Rückgabewerts der Funktion oder durch eine Exception passiert ist hierbei irrelevant. Wir haben uns für eine Implementation über den Rückgabetypp entschieden, da dies in der bisherigen Version von ODEMX auch so gelöst wurde und wir mit dieser Umsetzung nicht brechen wollten.

Neben dieser Anforderung hat unsere Implementation auch die einfache Verwendung im Sinn: Die Lösung soll Nutzern möglichst wenig Zusatzaufwand erzeugen und somit zum Beispiel Typkonvertierungen soweit möglich selbst vornehmen.

Integraler Bestandteil der Implementatin ist die neue generische Klasse `PortElement< T >`, welche in `PortT< T >` enthalten ist. Abhängig davon ob der Templateparameter `T` ein Zeigertyp ist oder nicht, wird der enthaltene Puffer in `PortT` entweder mit dem Zeigertypen oder mit `PortElement< T >` parametrisiert. Der zu verwendende Typparameter wird durch eine templatebasierte Typauswahl ermittelt:

```
typedef typename std::conditional<
    std::is_pointer< ElementType >::value,
    ElementType, PortElement< ElementType >
>::type PortElementType;
```

Für `PortT<T>` ist der `PortElementType` gleich dem Typ `T`, falls `T` ein Zeigertyp ist, und sonst gleich dem Typ `PortElement<T>`. Nachfolgend wird das Interface dieser neuen Klasse vorgestellt.

```
template< class T >
```



```

class PortElement
{
    T t;
    bool valid_;

public:
    constexpr PortElement();
    PortElement( T& t );
    PortElement( const PortElement& ) = delete;
    PortElement( T&& t );
    PortElement( PortElement&& c );
    PortElement& operator=( PortElement&& c );
    T get() const;
    T get();
    bool valid() const;
    T operator*();
    T* operator->();
    const T* operator->() const;
    operator T() const;
    bool operator!=(std::nullptr_t);
    bool operator==(std::nullptr_t);
    bool operator==( const PortElement& c ) const;
};

```

Listing 2.3: Die Klasse `PortElement<T>`

`PortElement< T >` bildet einen Zeigertyp nach, um den Rückgabotyp der Methode `PortHeadT< T >::get()` einheitlicher zu gestalten. Dieser ist damit also nicht zwingend gleich dem Templateparameter `T`, sondern, falls `T` kein Zeigertyp ist, eine Instanziierung des Typs `PortElement< T >`. Durch die Ähnlichkeit in der Verwendung zu einem Zeigertyp kann der Typparameter des Ports von einem Zeigertyp zu einem Nichtzeigertyp und umgekehrt geändert werden, ohne etwas an der Implementation der Objektentnahme an `PortHeadT< T >` ändern zu müssen.

Zudem bietet `PortElement<T>` Konvertierungsmethoden an, um eine Instanz diesen Typs direkt in eine Instanz des Typs `T` zu überführen. Damit kann ein Nutzer das Ergebnis eines Aufrufs von `PortHeadT<T>::get` direkt einem Objekt vom Typ `T` zuweisen, ohne weitere Konvertierungen durchführen zu müssen. Dies stellt also eine Reduktion des Aufwandes für den Nutzer dar, und somit wird damit eine der zwei oberen Anforderungen erfüllt.

Neben dieser Vereinfachung muss auch gewährleistet werden können, dass ein Fehlerfall oder die Unterbrechung des wartenden Prozesses durch die Rückgabe von `PortHeadT<T>::get` angezeigt werden kann. Hierzu dient die Methode `PortElement<T>::operator!=(std::nullptr_t);` und die prüft ob die Variable `PortElement<T>::valid_` den Wert `true` enthält. Nachfolgend wird die Symmetrie bei Verwendung unterschiedlicher Templateparameter gezeigt und damit demonstriert, dass der neu eingeführte Typ zu einer Vereinheitlichung in der Verwendung von `PortHeadT< T >::get()` führt, wie in Listing 2.4 gezeigt wird

Wie an obigem Beispiel ersichtlich werden Zeigertypen und normale Typen symmetrisch auf Gültigkeit geprüft. Der Vergleich mit der Konstanten `nullptr` führt immer zu einer Prüfung

2. Verwendung der neuen Features in ODEMX

```
// define two ports and initialize them
PortHeadT< int* > pints = ...;
PortHeadT< double > doubles = ...;

// try to get two elements
auto pint = pints.get(); // Entnehme einen Zeiger
auto d = doubles.get(); // Entnehme ein verpacktes double

// Ist der Zeiger gueltig?
if (pint != nullptr) { ... }

// Ist das verpackte double gueltig?
if (d != nullptr) { ... }
```

Listing 2.4: Die neue Implementation erlaubt eine einheitliche Verwendung für beliebige Typen.

auf Gültigkeit, womit überprüft werden kann, ob es im Ablauf einen Fehler gab oder ein wartender Prozess unterbrochen wurde. Hierbei ist wichtig, dass die Konstante `nullptr` genutzt wird, da damit für ein in `PortElement< T >` verpacktes Element garantiert ist, dass die Methode `PortElement< T >::operator!=(nullptr_t)` gerufen wird.

Mit dieser Änderung haben wir beide oben gestellten Anforderungen erfüllt. Für Nutzer der Bibliothek ist der Rückgabotyp von `PortHeadT< T >::get()` vereinheitlicht und durch automatische Konvertierungen bei Verwendung von `PortTailT< T >::put(T)` ergibt sich kein Mehraufwand. Zusätzlich können durch die neue generische Klasse auch Typen als Templateparameter genutzt werden, die keine *copy semantic* erlauben und nur bewegt werden dürfen. Im nächsten Abschnitt betrachten wir noch den Aspekt der Performanz und analysieren, welche Auswirkung die eingebrachten Änderungen auf das Laufzeitverhalten des Port haben.

2.1.4. Performanz der Änderung

Die Laufzeitänderung durch die Änderung der Implementation wurde anhand von vier Benchmarks analysiert. Der erste Benchmark (a) misst die Zeit die benötigt wird, um 10000 Objekte in einem Port abzulegen, der zweite (b) die Zeit zur Entnahme von 10000 Objekten und der dritte (c) sowie der vierte (d) Benchmark misst die benötigte Zeit um ein bzw. 1000 Objekte abzulegen und wieder zu entnehmen. Jeder Testlauf wurde 100 mal wiederholt und das Ergebnis gemittelt, um verlässliche Aussagen treffen zu können.

Die Ergebnisse sind in Tabelle 2.1 zusammengefasst. Es zeigt sich, dass bis auf sehr leichte testbedingte Schwankungen praktisch keine Laufzeitunterschiede festzustellen sind. Die verzeichneten Unterschiede befinden sich im Hundertstelsekundenbereich und sind damit eher leicht durch Messfehler auf der Testmaschine erzeugt worden. Anhand dieser Zahlen kann man den Schluss ziehen, dass eine elegante Lösung für die in Abschnitt 2.1.2 dargestellten Probleme

implementiert werden kann, ohne durch die Änderungen größere Geschwindigkeitseinbußen in Kauf nehmen zu müssen.

2.2. Prädikate und Gewichtsfunktionen

Wie oben bereits erwähnt, bietet ODEMX verschiedene Konzepte zur Prozesssynchronisation an. Neben diesem Konzept sind auch Prädikate und Gewichtsfunktionen nötig um Systemmodelle implementieren zu können. So sollte eine Prozesssimulationsbibliothek Mittel anbieten, um aus einem Nachrichtenpuffer bestimmte Nachrichten filtern zu können, oder um den Synchronisationspartner anhand einer gewissen Eigenschaft wählen zu können. ODEMX bietet hierfür drei Funktionszeigertypen zur Abstraktion von einfachen Prädikaten und Gewichtsfunktionen an, die in Synchronisationsmethoden eingesetzt werden sollen:

1. `typedef bool(Process::*Condition)();`
2. `typedef bool(Process::*Selection)(Process* partner);`
3. `typedef double(Process::*Weight)(Process* partner);`

Diese Methodenzeiger drücken verschiedene Semantiken aus: Funktionen vom Typ `Condition` werden genutzt um zu prüfen ob ein Prozess eine Bedingung erfüllt, `Selection` wird genutzt um einen passenden Partnerprozess zur Synchronisation zu finden und `Weight` berechnet das *Gewicht* eines Prozesses um einen Partnerprozess anhand der Wechselwirkung zwischen suchendem und dem zu prüfenden Prozess zu finden.

Funktionshülle zur Verallgemeinerung von Prädikats- und Gewichtsfunktionen Die Typdefinitionen `Condition`, `Selection` und `Weight` definieren jeweils Zeiger auf Methoden der Klasse `Process`. Instanzen dieser Typen müssen damit Methoden des Typs `Process` sein, weshalb Nutzer der Typen eine Typkonvertierung durchführen müssen:

```
// Hinweis: RealBounce ist eine Subklasse von Process
odemx::base::Condition cond = (odemx::base::Condition)&RealBounce::hitGround;
```

Um der Variable `cond` vom Typ `Condition` einen Zeiger auf die Methode `RealBounce::hitGround` zuweisen zu können, muss der Zeiger konvertiert werden, denn in der Klasse `process` existiert die Funktion `hitGround` nicht. Ein Nutzer der Bibliothek ist gezwungen die Zuweisung in dieser Form durchzuführen und die Prädikats- oder Gewichtsfunktion als Methode zu implementieren, obwohl der Zwang eine Methode zu nutzen oftmals nicht der Semantik des Prädikats selbst entspricht. Wenn ein Prädikat beispielsweise keine Aussage über die Eigenschaften eines

2. Verwendung der neuen Features in ODEMx

Objekts trifft sollte es auch keine Methode des Objekts sein. Oftmals reichen für Prädikate eine Funktionszeiger oder *Lambda-Ausdruck* aus. Diese sinnvollerer Konzepte können mit der aktuellen Implementation von `Condition`, `Selection` und `Weight` aber nicht verwendet werden, denn diese schreiben die Verwendung von Methodenzeigern vor.

C++11 bietet ein Konzept, um die Verwendung beliebiger Funktionen als Prädikat zu ermöglichen, falls die Signatur der Funktion mit der Vorgegeben übereinstimmt oder in diese überführbar ist. Dies ist durch eine Funktionshüllklasse `std::function<T>` implementiert, die einen Zeiger auf eine Funktion mit einer Signatur `T` kapselt und durch `operator()` einen Aufruf an die gekapselte Funktion weiterleitet. Sie abstrahiert dadurch verschiedene Funktionskonzepte aus C++ und fasst diese zusammen.³

```
std::function<double (int)> fu = &foo; // Funktionszeiger kapseln

// Auch lambda Ausdrücke koennen gekapselt werden
std::function<double (int)> bar = [] (int) -> double {
    return 0;
};
```

Listing 2.5: `std::function<T>` kapselt aufrufbare Entitäten.

Die durch `std::function` angebotene Abstraktion erlaubt eine verallgemeinerte Implementation der obigen Konzepte und damit die Verwendung verschiedener aufrufbarer Entitäten als Prädikats- und Gewichtsfunktionen. Im folgenden Abschnitt wird demonstriert, wie diese Verallgemeinerung implementiert werden kann.

Der Typ `Weight` wird hierbei aus technischen Gründen übergangen, die am Ende diesen Unterkapitels näher dargestellt werden.

Verwendung in ODEMx

Um von `std::function< T >` Gebrauch zu machen, wurden die zwei Typen `Condition` und `Selection` angepasst:

```
typedef std::function< bool () > Condition;
typedef std::function< bool ( Process &partner ) > Selection;
```

Diese können wie im obigen Beispiel verwendet werden und erlauben damit die Verwendung von Funktionszeigern, Lambdaausdrücken und Funktoren, aber nicht mehr von Zeigern auf Methoden. Diese Restriktion ist unproblematisch, da C++11 mit der Funktion `std::bind` ein

³C++ kennt verschiedene aufrufbare Entitäten: 1. C-artige Funktionen 2. Methoden 3. Aufrufbare Objekte (Funktoren) 4. Lambda-Ausdrücke. Diese Entitäten sind zumeist auch in Form von Zeigern aufrufbar. Näheres dazu und zur Implementation von generischen Funktoren als Abstraktion aufrufbarer Entitäten in C++ finden sich in [Ale01, S. 99].

einfaches Mittel bietet, um Parameter einer Funktion fest an ein Argument zu binden und damit das aus funktionalen Sprachen bekannte Konzept der partiellen Funktionsanwendung realisiert. Für den Einsatz von Methodenzeigern als Prädikate vom Typ `Condition` und `Selection` führt dies zu einem Aufruf ähnlich folgendem Beispiel.

```
// Wir nehmen an, dass diese Methode X::predicate bereits definiert
//      vorliegt
bool X::predicate();

// Um dieses Praedikat zu ueberfuehren, muss das erste Argument
// fest an den Zeiger auf das Objekt gebunden werden
Condition cond = std::bind( &predicate, this );
```

Die Funktion `std::bind` bindet den Zeiger auf das Objekt `this` an den ersten Parameter der Funktion `predicate`. Der erste Parameter einer Methode ist der implizite Zeiger auf das Objekt, auf welchem die Funktion ausgeführt werden soll. Dieser wird in C++ bei der Deklaration der Funktion nicht mit angegeben. Hiermit liefert `std::bind` nun ein aufrufbares Funktionsobjekt zurück, das einer Funktion mit der Signatur `bool()` entspricht, und damit auch wieder dem Funktionshüllobjekt vom Typ `Condition` zugewiesen werden kann.

Diese Änderung verallgemeinert also die in ODEMX vorhandenen Prädikate derart, dass diese für den Nutzer einfacher und vielseitiger genutzt werden können. Die Konvertierung einer Methode in ein Funktionshüllobjekt ist für Anwender ohne Kenntnisse der funktionalen Programmierung oder mit fehlendem C++ Hintergrund unter Umständen schwer verständlich; in diesen Fällen bietet sich die Einführung eines einfachen Macros an, das die Konvertierung übernimmt.

Keine Adaption von `odemx::base::Weight` Die Schnittstelle für Gewichtsfunktionen `Weight` wurde außen vorgelassen und muss weiterhin mit Methoden verwendet werden. Der Grund hierfür liegt in der Semantik dieser Methoden. Diese werden einerseits zur Suche eines Synchronisationspartners verwendet, andererseits aber auch um einen Prozess als einen möglichen Partner anzubieten. Grundlage dieser Semantik ist die Synchronisationsklasse `WaitQ`. Diese dient zur Synchronisation zweier Prozesse, wobei einer der zwei nach erfolgter Synchronisation “aufgeweckt” wird, um eine Leistung zu erbringen. Die zwei Prozesse unterscheiden sich damit in ihren Rollen; der nach erfolgter Synchronisierung aktive Prozess wird als *master* bezeichnet, der inaktive als *slave*. Prozesse, welche die *master*-Rolle einnehmen wollen, suchen sich ihre *slaves* aus und können dafür eine Gewichtsfunktion verwenden. Falls ein *master* sich zur Synchronisation anmeldet, so wird die Liste vorhandener *slave*-Prozesse nach dem Prozess mit dem maximalen Ergebniswert der Gewichtsfunktion für die Synchronisation ausgewählt. Hierbei erfolgt der Aufruf der Gewichtsfunktion durch `(master ->* Gewichtsfunktion)(a),`

2. Verwendung der neuen Features in ODEMX

wobei `master` konstant ist und die wartenden *slave*-Prozesse `a` iteriert wird.

Wenn hingegen ein *slave*-Prozess sich für die Synchronisation anmeldet, so wird entsprechend ein *master*-Prozess mit maximalem Gewicht gewählt und der Aufruf der Gewichtsfunktion erfolgt über `(a ->* Gewichtsfunktion)(slave)`, wobei `slave` konstant ist und `a` die wartenden *master*-Prozesse sind.

Hieran zeigt sich, dass bei der aktuellen Implementation bei *master*-Prozessen der `this`-Zeiger und damit das erste Argument der Gewichtsfunktion konstant gebunden werden muss, während im Falle eines *slave*-Prozesses das zweite Argument gebunden wird. Somit müssten allgemeine Gewichtsfunktionen beide der folgenden Signaturen erfüllen:

```
// Auswahl von slave-Prozessen
bool (*master_weight)(odemx::base::Process*);

// Auswahl von master-Prozessen
bool (odemx::base::Process::*slave_weight)();
```

Damit ist keine konsistente beide Fälle vereinende Implementation möglich. Eine mögliche Lösung wäre die Aufteilung der Schnittstelle in zwei getrennte Typen, die die obigen Signaturen nutzen.

2.3. Process::wait

ODEMX bietet eine Synchronisationsfunktion `Process::wait` für Prozesse, um auf Zustandsänderungen von Ableitungen der Klasse `IMemory` zu warten.⁴ So kann beispielsweise auf das Eintreffen eines Objekts am Ausgang eines Nachrichtenpuffers gewartet werden.

Es gibt zwei Überladungen der Methode `Process::wait`, die den Aufruf mit bis zu 6 oder mit beliebig vielen Objekten ermöglichen:

```
/* Aufruf mit bis zu 6 Argumenten */
IMemory* wait( IMemory* m0, IMemory* m1 = 0,
               IMemory* m2 = 0, IMemory* m3 = 0,
               IMemory* m4 = 0, IMemory* m5 = 0 );

/* Warten auf beliebig viele Objekte */
IMemory* wait( IMemoryVector* memvec );
```

`Process::wait` liefert einen Zeiger auf das Objekt zurück, welches durch eine Zustandsänderung zur Aktivierung des wartenden Prozesses geführt hat. Dies ist also einer der übergebenen Zeiger

⁴Die Implementation dieser Methode wird in [KFA07a, S. 43] ausführlich dargestellt und daher hier ausgelassen.

oder, falls der Prozess während des Wartens durch Aufruf der Methode `Process::interrupt` unterbrochen wurde, ein Nullzeiger.

Die aktuelle Implementation dieser zwei Funktionen bringen eine Unannehmlichkeit für den Nutzer mit sich, die im folgenden Abschnitt erörtert werden soll. Wie sich darauf zeigen wird, kann dieses stilistische Problem unter Verwendung neuer Sprachmerkmale gelöst werden.

2.3.1. Vereinfachung durch C++11

`Process::wait` ist umständlich zu verwenden: So können bis zu 6 `IMemory` Objekte direkt als Argumente an die Methode übergeben werden, aber keine 7. Will man mehr als 6 Argumente verwenden, so muss ein Vektor von `IMemory` Objekten aufgebaut und der Methode übergeben werden.

C++11 bietet hierfür zwei Möglichkeiten zur stilistischen Verbesserung an:

1. Durch *braced initialization* kann ein Vektor direkt durch `{ }` initialisiert werden.
2. Mit *variadic templates* kann eine Funktion mit beliebig vielen Argumenten implementiert werden.

Die Ansätze unterscheiden sich in ihrer syntaktischen Verwendung. So verlangt der erste Ansatz den Aufruf der Methode Vektor, der auch ein *rvalue* sein könnte:

```
p.wait({m1,m2,m3,m4});
```

Der zweite Ansatz hingegen ermöglicht den direkten Aufruf der Methode mit beliebig vielen Argumenten und damit eine an die C-Funktion `printf` ähnelnde Verwendung:

```
p.wait(m1,m2,m3,m4,m5,m6,m7,m8);
```

Beide Ansätze haben Vor- und Nachteile: *Variadic templates* ermöglichen beliebig viele Argumente ohne Wissen des Nutzers über *braced initialization*, sind aber aufwändiger zu implementieren. *Braced initialization* erfordert hingegen keine Neuimplementation in der Bibliothek, da die Standardbibliothek mit dem neuen Standard bereits die Möglichkeit der Initialisierung eines Vektors durch `{ }` mitbringt, aber Nutzer müssen dies wissen.

2.3.2. Implementation in ODEmx

Die umgesetzte Implementation in ODEmx vereint beide Ansätze. `Process::wait` wird weiterhin in zwei Überladungen angeboten, wobei ein *variadic template* die Methode mit Standardargumenten ersetzt. Diese neue Methode leitet ihre Argumente verpackt als Vektor an

2. Verwendung der neuen Features in ODEMX

`IMemory* wait(const IMemoryVector* memvec);` weiter und muss somit nicht als rekursive Funktion⁵ implementiert werden:

```
template< class ... Types >
    typename std::enable_if< odemx::type_traits::all_inheriting< IMemory, Types...
        >::value, IMemory* >::type
wait( Types* ... m ) { return wait( IMemoryVector({ m... }) ); }

IMemory* wait( const IMemoryVector& memvec );
```

Wie an dem Quelltextausschnitt zu sehen, wird das function parameter pack entpackt, um einen temporären Vektor zu erstellen, der an `IMemory* wait(const IMemoryVector& memvec);` übergeben wird. Die Signatur der Methode mit einem Vektor als Parameter wurde um das Schlüsselwort `const` erweitert, um auch den Aufruf der Funktion mit einem *rvalue* zu ermöglichen.

Die Implementation der Methode mit beliebig vielen Argumenten ist aufgrund der Verwendung von `std::enable_if` kompliziert. `std::enable_if` ist nötig, um Typsicherheit garantieren zu können. Die komplizierte Darstellung des Rückgabetyps der Methode sorgt dafür, dass eine Instanziierung der Methode nur dann möglich ist, wenn alle Argumente der Funktion Zeiger auf Objekte vom Typ `IMemory` sind. Dies verhindert den Aufruf mit falschen Argumenten und erlaubt die Ausschrift einer Fehlermeldung.⁶

So meldet G++ 4.7 bei Aufruf der Methode mit `p.wait(&timer1, &timer2, timer3)` für drei Objekte `timer{1,2,3}`:

```
template argument deduction/substitution failed:
  mismatched types 'Types*' and 'odemx::synchronization::Timer'
```

Dies bedeutet, dass eines der Argumente einen Typ `odemx::synchronization::Timer` hat, und nicht mit dem geforderten Zeigertyp übereinstimmt.

2.3.3. Performanz

Variadic templates haben einen geringen Einfluss auf die für die Programmausführung benötigte Zeit, da diese bereits zur Compilezeit ausgewertet und instanziiert werden. In unseren Tests hat sich herausgestellt, dass auch die Compilierung einfacher Programme nicht unter der Verwendung von Funktionen mit beliebiger Anzahl von Argumenten leidet. Tabelle 2.2 stellt die benötigte Zeit zur Compilierung des nachfolgenden Programms dar.

⁵Vergleiche dazu die typsichere `printf`-Funktion in Abschnitt 1.8.1.

⁶Das verwendete *type trait* `odemx::type_traits::all_inheriting< T, Xs... >` ist nicht in der Standardbibliothek enthalten und findet sich im Anhang.


```

int main() {
    odemx::base::Simulation& sim = odemx::getDefaultSimulation();
    odemx::synchronization::Timer t1(sim, "Timer");

    TestProcess p(sim);
    p.wait(&t1); // #1
}

```

Listing 2.6: Performanzmessung

Wie leicht zu sehen, unterscheidet sich die benötigte Zeit nur minimal und kann durchaus im Bereich von Messfehlern liegen.

Um ein realistischeres Bild auf die Laufzeitauswirkung der neuen Implementation zu gewinnen, wurde ein weiterer Versuch aufgebaut, bei dem die benötigte Zeit zur Compilation eines Programms der folgenden Form gemessen wurde:

```

int main() {
    odemx::base::Simulation& sim = odemx::getDefaultSimulation();
    odemx::synchronization::Timer t1(sim, "Timer");
    odemx::synchronization::Timer t2(sim, "Timer");
    :
    odemx::synchronization::Timer tN(sim, "Timer");

    TestProcess p(sim);
    p.wait(&t1);
    p.wait(&t1, &t2);
    :
    p.wait(&t1, &t2, ..., &tN);
}

```

Listing 2.7: Performanzmessungen erfolgten unter erwendung eines Skripts, das kompilierbare Quelltexte dieser Form generierte.

Dies soll die mehrfache Instanziierung des Templates simulieren und damit ein realistischeres Bild über die Auswirkung auf die Compilezeit erbringen. In obigem Template wird die Funktion `Process::wait()` $1, \dots, N$ mal instantiiert. Die Ergebnisse dieses Tests werden mit einem Testprogramm verglichen, dass das Template nur ein einziges Mal mit $1, \dots, N$ Argumenten instantiiert. Dies ist in Abbildung 2.3 dargestellt.

Es zeigt sich, dass die Instantiierung für kleine $n = 1, \dots, N$ konstanten Zeitverlust bringt und somit für eine realistische Menge an Instanziierungen des Templates kein enormer Geschwindigkeitsverlust zu erwarten ist. Zusammen mit der Erkenntnis, dass auch im Vergleich zur vorherigen Implementation offenbar nur wenig mehr Zeit für die Instanziierung verbraucht wird, ergibt sich ein positives Bild der Neuimplementation. Der durch sie erbrachte Mehrwert an Nutzbarkeit schlägt sich nicht negativ in der Performanz nieder.

2. Verwendung der neuen Features in ODEMx

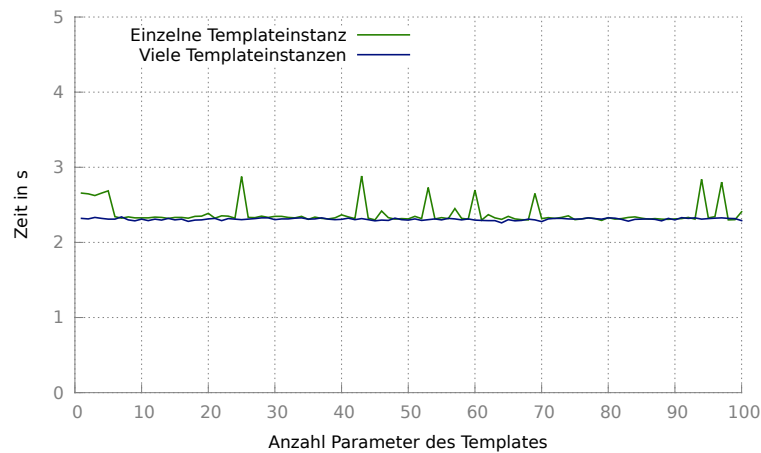


Abbildung 2.3.: Diese Abbildung bildet die in 2.3.3 beschriebenen Versuche zur Messung des Zeitverlusts bei der Compilation durch Mehrfachinstantiation einer Templatefunktion. Es zeigt sich, dass die verwendete Version des Compilers GCC (GCC 4.7) praktisch einen konstanten Zeitwaufrwand hierbei aufweist. Die Schwankungen bei der Instantiierung einer einzelnen Instanz unter Verwendung einer variablen Anzahl von Argumenten erklären sich aus äußeren Umständen der Testumgebung.

int			Teststructure		
Typ	Alte Impl.	Neue Impl.	Typ	Alte Impl.	Neue Impl.
a	0.0063	0.0062	a	0.0063	0.0063
b	0.0061	0.0062	b	0.0064	0.0062
c	0.0122	0.0210	c	0.0124	0.0123
d	0.0122	0.0210	d	0.0280	0.0119

int*		
Typ	Alte Impl.	Neue Impl.
a	0.0012	0.0006
b	0.0005	0.0002
c	0.0218	0.0290
d	0.0123	0.0288

Tabelle 2.1.: Diese Tabelle stellt das Ergebnis den in Anhang D.4 abgebildeten Benchmark dar. Es zeigt sich, dass die veränderte Implementation unter Verwendung einer neuen generischen Hüllklasse der vorhergehenden Implementation in Bezug auf die Geschwindigkeit nicht nachsteht. Sowohl bei Verwendung von einfachen Typen, Zeigern, aber auch komplizierteren Objekten benötigen beide Implementation vergleichbar lange zur Durchführung des Benchmarks.

(a) Alte Implementation		(b) Neue Implementation	
Index	Zeit in s	Index	Zeit in s
Durchschnitt	2.605	Durchschnitt	2.633
Standardabweichung	0.036	Standardabweichung	0.082
Median	2.603	Median	2.610

Tabelle 2.2.: Diese Tabellen stellen die benötigte Zeit zur Compilation eines einfachen Programms mit der alten und neuen Implementation von `Process::wait()` gegenüber. Es zeigt sich, dass die neue Implementation nur geringfügig langsamer als die alte compiliert wird, wobei dieser Zeitunterschied auch durch äußere Umstände erzeugt worden sein kann.

3. Fazit und Ausblick

Die durch C++11 erbrachten Neuerungen bringen viele Erweiterungen mit sich, die bestehende Bibliotheken vereinfachen und zugänglicher machen können. ODEMX kann an verschiedenen Stellen durch die neuen Sprachmerkmale profitieren und damit ohne spürbaren Performanzverlust an Mehrwert gewinnen. Inwiefern sich die Gesamtperformanz der Bibliothek unter C++11 verändert konnte im Rahmen dieser Arbeit leider nicht festgestellt werden. Hierfür fehlt ein Werkzeug zur kontinuierlichen Performanzmessung der Bibliothek. Wir stellen dies als sinnvolle Erweiterung durch eine zukünftige Arbeit in Abschnitt 3.1.3 vor.

Viele der Änderungen innerhalb der Bibliothek sind stilistischer Natur und betreffen Nutzer der Bibliothek nicht unmittelbar. Hierzu gehört die konsequente Verwendung von Typableitung durch die Schlüsselwörter `auto` und `decltype`, die den Wartungsaufwand verringern. Damit dienen die verschiedenen neuen Sprachmerkmale mehr der Konsolidierung bestehenden Quelltexts und weniger der Einführung neuer Werkzeuge.

Aber auch Neuerungen konnten sinnvoll unter Verwendung von C++11 implementiert werden. So ermöglichen *variadic templates* und *rvalue Referenzen* eine natürlichere und generalisierte Verwendung der Bibliothek. An welchen Stellen diese Sprachmerkmale noch eingesetzt werden kann sich aber nur durch die Praxis zeigen und muss über Rückmeldung von Nutzern erfolgen. Hierzu gehören unter anderem auch die neuen, auf Mehrkernarchitekturen ausgerichteten, Merkmale für nebenläufige Programmierung. Diese haben großes Potential, um aufwändige Prozesse zu beschleunigen, aber die explizite Serialisierung von Ereignissimulationen stellt eine große Hürde für die Verwendung solcher Konzepte dar.

3.1. Ausblick

Im Folgenden werden mehrere Problemfelder angegeben, die in einer späteren Arbeit aufgegriffen werden können, um ODEMX weiter zu verbessern. Diese haben nicht alle unmittelbar etwas mit dem neuen Sprachstandard zu tun, wurden aber im Laufe der Arbeit als Probleme identifiziert. Ihre Lösung hätte den Rahmen der Arbeit gesprengt, weshalb nachfolgende Ar-

3. Fazit und Ausblick

beiten diese aufgreifen sollten. Die jeweiligen Vorschläge werden in kurzen Unterabschnitten präsentiert und es wird, wo sinnvoll, die Grundlage von Lösungen gelegt.

3.1.1. Weitere neue Sprachmerkmale und Änderungen der Standardbibliothek

Wie in Abschnitt 1.10 angesprochen behandelte diese Arbeit nicht alle neuen Sprachmerkmale und ging auch nur bedingt auf die Erweiterungen der Standardbibliothek ein. Eine Übersicht über die ausgelassenen Sprachmerkmale findet sich in Anhang A. Dieser Abschnitt des Anhangs listet nicht die Änderungen der Standardbibliothek auf, da diese den Rahmen der Ausarbeitung sprengen würden. Informationen zu den Neuerungen in der Standardbibliothek finden sich zum Beispiel in der FAQ von Stroustrup zu C++11 [Str11a]. Sowohl die neuen Sprachmerkmale wie beispielsweise native Unterstützung von Multithreading als auch Änderungen der Standardbibliothek wie der neu eingeführte generische Container `std::array` könnten für ODEMX interessant sein.

3.1.2. Vereinigung BinT/ResT

Die Synchronisationsklassen `BinT<T>` und `ResT<T>` haben große Ähnlichkeiten. Beides sind Container, die eine Menge von Objekten enthalten, die durch Prozesse entnommen werden. Prozesse warten auf diese Objekte und werden aktiviert, wenn neue Objekte aufgenommen wurden. Der Unterschied zwischen den beiden Klassen liegt in der Anzahl der Objekte, die sich in den Instanzen befinden können: In Instanzen der Klasse `BinT<T>` können sich beliebig viele Objekte befinden, während Instanzen der Klasse `ResT<T>` eine obere Schranke für die Anzahl der enthaltenen Objekte haben. Das Verhalten der Synchronisationsklassen ist in großen Teilen identisch und unterscheidet sich in der Beschränktheit der Anzahl. Aktuell teilen sich die Klassen aber keine gemeinsame Implementationsbasis; daher müssen zwei Klassen mit fast identischem Verhalten gewartet werden. Diese Klassen könnten sinnvoll zusammengeführt werden.

Mögliche Ansätze Mehrere Ansätze stehen zur Auswahl um die gemeinsame Funktionalität nicht mehrfach zu implementieren:

- Einarbung einer gemeinsamen Basisklasse
- Implementation beider Klassen als Template unter Verwendung einer Policy-Klasse¹

¹Hierunter versteht man einen Templateparameter, der das Verhalten oder Eigenschaften der parametrisierten Klasse festlegt. In dem Fall von `BinT/ResT` würde die *Policy-Klasse* festlegen, ob eine obere Schranke für die Anzahl der Objekte existiert oder nicht.

- Modellierung ähnlich des Port-Konzepts in ODEMx: Assoziation mit einer Implementationsklasse, die für den Nutzer nicht sichtbar ist.

Die Änderungen sind räumlich lokal und können implementiert werden, ohne bestehenden Code brechen zu müssen, da die Schnittstellen der Klassen unverändert bleiben können.

3.1.3. Performanzmessung der Bibliothek

ODEMx bringt kein Rahmenwerk zur Erfassung von Mikro- und Makrobenchmarks mit. Um im weiteren Verlauf der Entwicklung die Auswirkung von Änderungen auf die Performanz konsistent erfassen zu können, sollte ODEMx neben einer Testumgebung auch kontinuierlichen Performanzmessungen unterworfen werden.

Mögliche Implementation Die Implementation könnte ähnlich dem Ansatz der Testbibliothek *UnitTest++*² erfolgen: Für jede Klasse wird eine Menge von Benchmarks definiert und automatisch der Reihe nach ausgeführt. Diese Mikrobenchmarks auf Funktionsebene ermöglichen einen genauen Einblick in Performanzänderungen auf niedriger Ebene und ermöglichen die Messung von Änderungen der Klasse selbst. Zusätzlich sind auch Tests auf einer höheren Ebene erforderlich, um die Auswirkungen auf größere Simulationen bewerten zu können. Hierfür könnten größere Beispiele wie die in [FA96] eingesetzten Simulationen herangezogen und automatisiert bei Änderungen neu gemessen werden.

3.1.4. Simulationszeit als Template Parameter für Simulationen

ODEMx bietet sowohl zeitkontinuierliche als auch zeitdiskrete Simulation an. Die Simulation zeitkontinuierlicher Prozesse erfordert die Verwendung eines Fließkommatyps als Grundtyp der Zeit, bei Simulation zeitdiskreter Prozesse reicht ein Typ, mit diskreter Schrittweite. Hieraus resultiert, dass ODEMx in zwei unterschiedlichen Formen kompiliert werden muss, falls ein Nutzer mit den unterschiedlichen Zeittypen arbeiten will. Dies ist für Nutzer der Bibliothek ein Problem, die unter Umständen zwei unterschiedliche Versionen der Bibliothek nutzen müssen und nur durch die Fehlermeldungen des Compilers über die falsche Ausprägung der Bibliothek informiert werden können.

Der Zeittyp `SimTime` ist ein `typedef`, der von einer zur Compilezeit festgelegten Makrodefinition abhängt. Bei späterer Verwendung der Bibliothek muss darauf geachtet werden, dass (a) die richtige Bibliothek verwendet wird, da die Funktionssignaturen von dem zur Compilezeit festgelegten Typ abhängen (b) Templateparameter, die zum Zeittyp passen müssen, entsprechend richtig gewählt werden. Dies ist unhandlich und sollte in einer späteren Arbeit gelöst werden.

²<http://unittest-cpp.sourceforge.net/>, besucht am 19.10.2011.

3. Fazit und Ausblick

Eine mögliche Implementation Eine mögliche Lösung wäre die Bindung des Zeittyps an die Simulation: Es erscheint logisch, dass die Simulationszeit eine Eigenschaft der Simulation ist und somit auch deren Grundtyp zur Simulation gehört. Durch Einführung eines Typs `template < class SimTime > Simulation< SimTime >` könnten das obige Problem gelöst und eine stärkere logische Verbindung zwischen Simulation und entsprechendem Zeittyp hergestellt werden. Dieser Ansatz ist jedoch nicht kostenlos: Sehr viele andere Klassen nutzen den Typ `SimTime` und müssten damit auch zu Templateklassen werden, was wiederum die Komplexität der Bibliothek erhöht.

Problematic Compilezeit Neue Templateklassen erhöhen aber nicht nur die Komplexität, sondern bringen längere Compilezeiten mit sich. Es ist zu beachten, dass eine Implementation zwingend auch den Mehrbedarf an Compilezeit beachten muss. Problematisch hierbei ist, dass sich die Compilezeit nicht nur für die Bibliothek erhöht, sondern auch von Nutzerprogrammen, da Templates grundsätzlich bei der Compilierung vorliegen müssen und ein Nutzer bei Verwendung der Templates somit immer die komplette Definition nutzen muss. Eine Verbesserung der Situation könnten *extern templates* bringen, die in C++11 eingeführt wurden. Diese sollen hier aber nicht weiter erörtert werden.

3.1.5. Ersetzen von Aufzählungstypen in ODEmx

ODEmx verwendet eine Vielzahl von Aufzählungstypen wie zum Beispiel `Process::State`. Viele dieser Typen könnten durch Polymorphie ersetzt werden und damit stärker auf Sprachmittel von C++ zurückgreifen. Beispiele hierfür sind die Enumerationen `Sched::Type` oder `ReportTable::Column::Type`. Hierbei muss evaluiert werden, welche Aufzählungstypen sinnvoll ersetzt werden können und ob die Performanzauswirkungen diese Änderungen rechtfertigen. Um dies bewerten zu können, sollte aber wie in Abschnitt 3.1.3 beschrieben kontinuierlich Performanzmessungen durchgeführt werden.

3.2. Fazit

C++11 bringt eine Vielzahl an Änderungen. Viele neue Sprachmerkmale erleichtern die Arbeit für Entwickler, andere verbessern die Erlernbarkeit der Sprache. Die Erweiterung der Standardbibliothek implementiert eine große Zahl von nützlichen Funktionen und Typen, die bisher nur in Bibliotheken wie Boost³ verfügbar waren. Dies schließt verschiedene Abstraktionsmittel, wie zum Beispiel dem polymorphen Funktionshülltyp, aber auch eine neue *type traits* zur

³<http://www.boost.org/>, bezogen am 19.10.2011.

Bestimmung von Typmerkmalen, ein. C++11 erscheint nach der langen Entwicklungszeit als guter Zwischenschritt hin zum nächsten Standard, der die in diesem Standard ausgesparten Aspekte aufgreifen soll. Hierzu gehört unter anderem die Erweiterung um *concepts*, welche zur Definition der Eigenschaften und Anforderungen an Templateparametern dienen sollen.

Wie in dieser Arbeit dargelegt, profitiert ODEMX an verschiedenen Stellen durch die Änderungen des neuen Standards. Einige Änderungen des Standards erfordern nur geringe Eingriffe in die Bibliothek und bringen automatisch Verbesserungen mit sich. Hierzu gehört die neue *move semantic* und *rvalue* Referenzen. Andere neue Sprachmerkmale, beispielsweise die *braced initialization*, erfordern viele kleine Änderungen, die sich aber aufgrund der besseren Lesbarkeit und klarerer Syntax für spätere Weiterentwicklungen der Bibliothek auszahlen können.

A. Tabellarische Übersicht der ausgelassenen Neuerungen

Dieser Anhang enthält mit Tabelle A.1 eine Übersicht der ausgelassenen Neuerungen aus C++11. Falls das entsprechende Sprachmerkmal bereits in GCC integriert wurde, so wird dies unter Verweis auf die entsprechende Version angegeben.

Es ist zu beachten, dass die Tabelle nicht alle neuen Sprachmerkmale und Erweiterungen aus C++11 umfasst, sondern nur jene, die mit Absicht in dieser Arbeit ausgelassen wurden. Einige der Änderungen sind nicht weitreichend, werden aber der Vollständigkeit hier genannt, um den interessierten Leser auf diese hinzuweisen. Eine ausführliche Übersicht der neuen Sprachmittel kann unter [Str11a] gefunden werden. [GCC11] enthält eine Übersicht des Implementationsstands in GCC 4.7. Der Stand entspricht dem Stand des Compilers am 20. März 2011 und wird sich voraussichtlich zur Veröffentlichung des Compilers nicht geändert haben.

A. Tabellarische Übersicht der ausgelassenen Neuerungen

Name	Beschreibung	Ab GCC Version
constexpr	Verallgemeinerung von <code>const</code> , um Konstanten zur Compilezeit nutzen zu können.	4.6
extern templates	Vermeide mehrfaches Instantiieren von Templates	4.3
Delegation von Konstruktoren	C++98 erlaubt nicht den gegenseitigen Aufruf von Konstruktoren innerhalb eines Objekts. Dies wird mit C++11 behoben und ermöglicht es, einfache Basiskonstruktoren zu schreiben die von anderen Konstruktoren genutzt werden können.	4.7
Explizites Überschreiben virtueller Funktionen	C++11 führt Schlüsselwörter ein, um Methoden explizit zu überschreiben und einen Fehler zu erzeugen, falls die Methode keine Methode der Basisklasse überschreibt.	4.7
String Literale	C++11 erlaubt Programmierern, neue Stringlitterale zu implementieren. Damit können zum Beispiel reguläre Ausdrücke direkt mit den Sprachmitteln von C++ dargestellt werden: <code>R"/[abc]/"</code> .	4.7
Speichermodell	Um Multithreading zu unterstützen, muss die Sprache ein Speichermodell implementieren.	—
Attribute	C++11 spezifiziert verallgemeinerte Attribute, die compilerübergreifend Eigenschaften spezifizieren sollen.	—
Weiterleiten von Exceptions	C++11 erweitert die Sprache um Mittel zur Weiterleitung von Exceptions zwischen verschiedenen Threads.	4.4
Template Argumente	C++11 erlaubt nun auch die Verwendung lokaler oder unbenannter Typen als Template Argumente	4.5
Initialisierung außerhalb des Konstruktors	C++11 erlaubt dem Nutzer, Instanzvariablen innerhalb der Klasse durch direkte Zuweisung zu initialisieren.	4.7

Abbildung A.1.: Eine Auswahl ausgelassener Sprachänderungen

B. Ausgewählte Grammatikregeln

In diesem Kapitel des Anhangs geben wir ausgewählte Grammatikregeln wieder, auf die wir im Laufe der Ausarbeitung verwiesen haben.

B.1. Braced initialization

Die *braced-init-list* wird zu Beginn von [ISO11, §8.5] beschrieben.

Grammatik

```
initializer:  
  brace-or-equal-initializer  
  ( expression-list )  
  
brace-or-equal-initializer:  
  = initializer-clause  
  braced-init-list  
  
initializer-clause:  
  assignment-expression  
  braced-init-list  
  
initializer-list:  
  initializer-clause ...opt  
  
initializer-list , initializer-clause ...opt  
  
braced-init-list:  
  { initializer-list ,opt }  
  {}
```


C. Hilfsprogramme und -algorithmen

Dieses Kapitel enthält Algorithmen und Codeausschnitte, auf die in der Arbeit verwiesen wurde, deren Umfang aber den Fließtext stören würde.

C.1. Type Traits

Diese Klasse ermöglicht es bei der Kompilation zu prüfen, ob alle Templateparameter Subklassen einer gegebenen Klasse sind.

```
namespace odemx {
namespace type_traits {
template <typename ...Args> struct all_inheriting : std::integral_constant< bool,
    false > {};

template <typename S, typename T, typename ...Args> struct all_inheriting<S, T,
    Args...> :
    std::integral_constant< bool, std::is_base_of<typename std::decay<S>::type,
        typename std::decay<T>::type>::value && all_inheriting<S,
        Args...>::value >
    {};

template <typename S, typename T> struct all_inheriting<S, T> :
    std::integral_constant< bool, std::is_base_of<typename std::decay<S>::type,
        typename std::decay<T>::type>::value >
    {};

template <typename T> struct all_inheriting<T>: std::integral_constant< bool,
    true > {};
}
}
```

src/all_inheriting.cpp

Ein Beispiel findet sich in der Implementation der Methode `Process::wait` in Abschnitt 2.3.2. Dort wird die Instanziierung einer Templatefunktion verhindert, falls nicht alle Argumente der Funktion Zeiger auf Subklassen der Klasse `IMemory` sind.

D. Programme zur Messung von Laufzeitverhalten

Im Rahmen der Studienarbeit wurden zwei einfache Klassen implementiert, die zur Generierung von Performanzmessungen genutzt werden können. Beide Klasse verwenden einfache Methoden zur Messung der Ausführungszeit einer übergebenen Funktion. Wir demonstrieren dies anhand eines einfachen Beispiels:

```
#include <PortableBenchmark.h>

void foo (const int N) {
    // Fuehre etwas aus
}

// generiere den naechsten Wert, mit dem foo() gerufen wird
void generator(const int N) {
    return N * 2;
}

int main () {

    PortableBenchmark benchmarker;

    const int wiederholungen = 10;
    const int von = 1, bis = 1000;
    benchmarker.bench(wiederholungen, &foo, von, bis, &generator);
}
```

Dieses Programm führt die Funktion `foo` mehrfach für verschiedene Eingaben aus und gibt für jeden durch `generator` generierten Wert den Mittelwert der Ausführungszeit aus. Nachfolgend werden die zwei Klassen vorgestellt, wobei die erste vorgestellte Klasse C++98-kompatibel ist und die zweite C++11 erfordert.

D.1. Portable Benchmark

Diese Klasse wurde zur Messung von einfachen Funktionen implementiert und ist C++98 kompatibel.

D. Programme zur Messung von Laufzeitverhalten

```
#include <iostream>
#include <vector>
#include <algorithm>

#include <sys/time.h>

class PortableBenchmark {
public:
    // fuehre einen Test N mal durch, wobei die Testfunktion fuer alle
    // Werte
    // initial..end aufgerufen wird. Es erfolgt die Ausgabe der
    // Durchschnittswerte
    // fuer jedes i in zwischen initial und end das durch die generator
    // Funktion
    // generiert wird
    template< typename T, typename F >
    void bench( int N, F f, T initial, T end, T (*generator)( T ) ) {
        for( T generator_value = initial; generator_value < end; generator_value =
            generator( generator_value ) ) {
            std::vector< double > times( N, double() );
            for( unsigned int i = 0; i < times.size(); i++ ) {
                times[i] = seconds();
                try {
                    f( generator_value );
                } catch( ... ) {
                    std::cerr << "An unhandled error occurred. Exiting." << std::endl;
                    exit(-1);
                }
                times[i] = seconds() - times[i];
            }
            std::cout << generator_value << ": " << sorted_mean( times ) << std::endl;
        }
    }

    /// Wie oben, fuehrt aber nach jedem Lauf von f destroy(i) aus
    template< typename T, typename F >
    void bench( int N, F f, T initial, T end, T (*generator)( T ), void(*destroy)( T
        ) ) {
        for( T generator_value = initial; generator_value < end; generator_value =
            generator( generator_value ) ) {
            std::vector< double > times( N, double() );
            for( unsigned int i = 0; i < times.size(); i++ ) {
                times[i] = seconds();
                try {
                    f( generator_value );
                } catch( ... ) {
                    std::cerr << "An unhandled error occurred. Exiting." << std::endl;
                    exit(-1);
                }
                times[i] = seconds() - times[i];
                destroy( i );
            }
            std::cout << generator_value << ": " << sorted_mean( times ) << std::endl;
        }
    }
}
```

```

// Wie die vorherige, fuehrt aber noch eine zusaetzliche Funktion gen
// vor jedem
// Durchlauf von f aus
template< typename T, typename F >
void bench( int N, F f, T initial, T end, T (*generator)( T ), void (*gen)( T ),
           void(*destroy)( T ) ) {
    for( T generator_value = initial; generator_value < end; generator_value =
        generator( generator_value ) ) {
        std::vector< double > times( N, double() );
        for( unsigned int i = 0; i < times.size(); i++ ) {
            gen( i );
            times[i] = seconds();
            try {
                f( generator_value );
            } catch( ... ) {
                std::cerr << "An unhandled error occured. Exiting." << std::endl;
                exit(-1);
            }
            times[i] = seconds() - times[i];
            destroy( i );
        }
        std::cout << generator_value << ": " << sorted_mean( times ) << std::endl;
    }
}

private:
static double seconds() {
    struct timeval tp;

    gettimeofday(&tp, NULL);
    return ( (double) tp.tv_sec + (double) tp.tv_usec * 1.e-6 );
}

template< typename T >
static T sorted_mean( std::vector< T >& v ) {
    std::sort( v.begin(), v.end() );
    T acc = T();
    for( int i = 0; i < v.size(); ++i ) acc += v[i];
    return acc / static_cast< T >( v.size() );
}
};

```

src/PortableBenchmark.h

D.2. Flexiblere Benchmarkklasse

Diese Klasse nutzt `std::function< T >` und muss damit mit einem neueren, C++11 fähigen, Compiler compiliert werden-

```

#include <iostream>
#include <functional>
#include <array>
#include <vector>
#include <algorithm>

```

D. Programme zur Messung von Laufzeitverhalten

```
#include <sys/time.h>

class Benchmark {

public:
    // fuehre einen Test N mal durch, wobei die Testfunktion fuer alle
    // Werte
    // 0..N-1 aufgerufen wird. Es erfolgt die Ausgabe der
    // Durchschnittswerte
    // fuer jedes i = 0, ..., N-1
    void bench( int N, std::function< void () > f ) {
        /// run f N times an messure time for each run
        std::vector< double > times( N, double {} );

        for( unsigned int i = 0; i < times.size(); i++ ) {
            times[i] = seconds();
            try {
                f();
            } catch( ... ) {
                std::cerr << "An unhandled error occured. Exiting." << std::endl;
                exit(-1);
            }
            times[i] = seconds() - times[i];
        }

        std::cout << sorted_mean( times ) << std::endl;
    }

    // Wie oben, aber die Argumente fur f werden durch einen gnerator
    // erzeugt.
    // Der generator kann ein lambda-Ausdruck sein, um seinen Zustand zu
    // behalten
    // (closure)
    template< typename T >
    void bench( int N, std::function< void( T ) > f, std::function< T() > generator
    ) {
        /// run f N times an messure time for each run
        std::vector< double > times( N, double {} );

        for( unsigned int i = 0; i < times.size(); i++ ) {
            times[i] = seconds();
            try {
                f( generator() );
            } catch( ... ) {
                std::cerr << "An unhandled error occured. Exiting." << std::endl;
                exit(-1);
            }
            times[i] = seconds() - times[i];
        }
        std::cout << "mean: " << sorted_mean( times ) << std::endl;
    }

    // wie oben, aber mit Laufindex i=initial,...,end, wobei die Werte durch
    // eine Generatorfunktion generator(N) erzeugt werden
    template< typename T >
    void bench( int N, std::function< void( T ) > f, T initial, T end,
    std::function< T( T ) > generator ) {
```

```

    for( T generator_value = initial; generator_value < end; generator_value =
        generator( generator_value ) ) {
        std::vector< double > times( N, double {} );
        for( unsigned int i = 0; i < times.size(); i++ ) {
            times[i] = seconds();
            try {
                f( generator_value );
            } catch( ... ) {
                std::cerr << "An unhandled error occurred. Exiting." << std::endl;
                exit(-1);
            }
            times[i] = seconds() - times[i];
        }
        std::cout << generator_value << ": " << sorted_mean( times ) << std::endl;
    }
}

private:
    static double seconds() {
        struct timeval tp;

        gettimeofday(&tp, NULL);
        return ( (double) tp.tv_sec + (double) tp.tv_usec * 1.e-6 );
    }

    template< typename T >
    static T sorted_mean( std::vector< T >& v ) {
        std::sort( v.begin(), v.end() );
        T acc = T {};
        // TODO should be using std::accumulate here
    for( T t : v ) acc += t;
        return acc / static_cast< T >( v.size() );
    }
};

#endif // __BENCHMARK_H

```

src/Benchmark.h

D.3. Unterschied move gegen copy semantic bei swap

In diesem Teil des Anhangs bilden wir zwei Benchmarks ab, die der Quantifizierung von Laufzeitänderungen durch die neue *move semantic* dienen.

D.3.1. Einfacher Benchmark

Dieser Benchmark versucht die Laufzeitunterschiede zwischen copy und *move semantic* anhand der Vertauschung einfacher Typen zu ermitteln.

```

/**
 * @file swap_performance.cpp
 * @brief How does swap performance change using c++0x?
 * @author Magnus Mueller

```

D. Programme zur Messung von Laufzeitverhalten

```
* @date 2012-03-13
*
*/

#include <PortableBenchmark.h>
#include <iostream>
#include <vector>

struct SimpleAggregate {
    int a;
    double b;
    char c;

    SimpleAggregate() :
        a()
        , b()
        , c()
    {}
};

struct CustomCopyCTor {
    int a;
    double b;
    char c;

    CustomCopyCTor() :
        a()
        , b()
        , c()
    {}

    CustomCopyCTor( const CustomCopyCTor & c ) :
        a(c.a+1)
        , b(c.b+1)
        , c(c.c+1)
    {}
};

template< class T , int Elements >
struct WrapTestRun {

    std::vector< T > a;
    std::vector< T > b;

    WrapTestRun() :
        a(Elements)
        , b(Elements)
    {}

    void operator()( const long long n ) {
        for(int i = 0; i < n; ++i) {
            std::swap(a[i], b[i]);
        }
    }
};
```

D.3. Unterschied move gegen copy semantic bei swap

```
long long generator( const long long n )
{
    return n * 2;
}

int main()
{
    Benchmark bench;
    {
        WrapTestRun< int, 1000000 > swapInts;
        std::cout << "Swap integers.." << std::endl;
        bench.bench<long long>( 100, swapInts, 1, 512000, generator );
    }
    {
        WrapTestRun< double, 1000000 > swapDoubles;
        std::cout << "Swap doubles.." << std::endl;
        bench.bench<long long>( 100, swapDoubles, 1, 512000, generator );
    }
    {
        WrapTestRun< SimpleAggregate, 10000000 > swapAggregates;
        std::cout << "Swap Aggregates.." << std::endl;
        bench.bench<long long>( 100, swapAggregates, 1, 512000, generator );
    }
    {
        WrapTestRun< CustomCopyCtor, 10000000 > swapCustomCopyCtor;
        std::cout << "Swap Custom Ctor.." << std::endl;
        bench.bench<long long>( 100, swapCustomCopyCtor, 1, 512000, generator );
    }
    {
        WrapTestRun< std::vector< int >, 10000000 > swapNestedVectors;
        std::cout << "Swap Nested Vectors.." << std::endl;
        bench.bench<long long>( 100, swapNestedVectors, 1, 512000, generator );
    }
    return 0;
}
```

src/swap_performance.cpp

D.3.2. Hinnant's Benchmark

Dieser Benchmark ist eine adaptierte Version des in [Hin06] vorgestellten Programms zur Messung von Laufzeitunterschieden bei Verwendung von move anstatt copy semantic. Der Benchmark füllt hierzu einen Vektor mit Mengen von Zufallszahlen und sortiert sowie rotiert diesen Vektor. Gemessen wird die Zeit zur Erstellung, Sortierung, Rotation und Destruktion des Vektors sowie die Gesamtlaufzeit für den Benchmark. Der Benchmark wird für verschiedene Anzahlen von Elementen jeweils 11 mal wiederholt und der Median der jeweiligen Messungen ausgegeben.

```
// From Howard E. Hinnant,
// http://cpp-next.com/archive/2010/10/howards-stl-move-semantics-benchmark/
#include <vector>
#include <iostream>
```

D. Programme zur Messung von Laufzeitverhalten

```
#include <time.h>
#include <set>
#include <algorithm>

unsigned long N;

extern bool some_test;

std::set<int>
get_set(int)
{
    std::set<int> s;
    for (int i = 0; i < N; ++i)
        while (!s.insert(std::rand()).second)
            ;
    if (some_test)
        return s;
    return std::set<int>();
}

std::vector<std::set<int> >
generate()
{
    std::vector<std::set<int> > v;
    for (int i = 0; i < N; ++i)
        v.push_back(get_set(i));
    if (some_test)
        return v;
    return std::vector<std::set<int> >();
}

float time_it(std::vector< std::vector< float > >& values)
{
    clock_t t1, t2, t3, t4;
    clock_t t0 = clock();
    {
        std::vector<std::set<int> > v = generate();
        t1 = clock();

        // construction
        values[0].push_back((float)((t1 - t0)/(double)CLOCKS_PER_SEC));

        // sort
        std::sort(v.begin(), v.end());
        t2 = clock();
        values[1].push_back((float)((t2 - t1)/(double)CLOCKS_PER_SEC));

        // rotate
        std::rotate(v.begin(), v.begin() + v.size()/2, v.end());
        t3 = clock();
        values[2].push_back((float)((t3 - t2)/(double)CLOCKS_PER_SEC));
    }
    t4 = clock();
    // destruction
    values[3].push_back((float)((t4 - t3)/(double)CLOCKS_PER_SEC));
    return (float)((t4-t0)/(double)CLOCKS_PER_SEC);
}
```



```

int main()
{
    std::cout << "#Elements Construction Sort Rotate Destruction Total"
               << std::endl;
    for (N = 250; N <= 3000; N += 250) {
        std::vector< std::vector< float > > values(5);
        std::cout << N << ' ';
        for (int j = 0; j < 11; ++j) {
            float t = time_it(values);
            values[4].push_back(t);
        }
        for(int i = 0; i < 5; i++) {
            std::sort(values[i].begin(), values[i].end());
            std::cout << values[i][5] << " ";
        }

        std::cout << std::endl;
    }
}

bool some_test = true;

```

src/howard.cpp

D.4. Performanz der veränderten Portimplementation

Um die neue Portimplementation analysieren zu können, wurde eine Reihe von Tests implementiert, die verschiedene Aspekte der Nutzung beleuchten. Da die Nutzer hauptsächlich mit der Ein- und Ausgabe von Ports interagieren, müssen diese besonders schnell implementiert sein. Daher liegt der Fokus diesen Benchmarks auf der Analyse der hieran beteiligten Komponenten. Wir messen mit dem Benchmark, wieviel Zeit beim Einfügen und Entnehmen von Elementen in den Kommunikationspuffer verloren geht und ob sich dies durch die Neuimplementation verschlechtert hat. Hierzu werden vier Tests durchgeführt und jeweils 100-mal wiederholt:

1. Füge 10000 Elemente in den Port ein und messe die dafür benötigte Zeit.
2. Entnehme 10000 Elemente aus dem Port und messe die dafür benötigte Zeit.
3. Füge ein Element ein und entnehme dieses direkt wieder; führe dies 10000 mal aus und messe die dafür benötigte Zeit.
4. Wie der vorherige Versuch, aber füge 1000 Elemente ein und nehme dann 1000 wieder raus. Dies wird 10 mal wiederholt.

Um nicht Fehlschlüssen durch die Verwendung einfacher Datentypen zu erliegen verwenden wir im Test drei verschiedene Datentypen, mit denen der Port parametrisiert wird: **int**, **double***

D. Programme zur Messung von Laufzeitverhalten

und `Teststructure`. `Teststructure` ist eine einfache Struktur wie weiter unten abgebildet und soll anhand eines größeren Datentyps demonstrieren, dass auch mit einem größeren Datendurchsatz kein Performanzverlust erzeugt wird. Die Liste der Typen enthält einen Zeigertyp um zu überprüfen, ob die neue Implementation bei Verwendung von Nichtzeigertypen schlechter abschneidet als bei Verwendung von Zeigertypen. Es stellte sich heraus, dass dies nicht der Fall ist, wie Tabelle 2.1 im Abschnitt zur Neuimplementation darstellt.

```
/**
 * @file port_old_implementation.cpp
 * @brief Benchmark the old port implementation. port.cpp doesn't compile
 *         with old revisions.
 * @author Magnus Mueller
 * @date 2011-07-27
 */

#include <odemx/odemx.h>
#include <Benchmark.h>
#include <Teststructure.h>

namespace os = odemx::synchronization;

constexpr size_t elements = 10000;

int main()
{
    /* benchmark value container */
    {
        std::cout << "==== int" << std::endl;
        odemx::base::Simulation& sim = odemx::getDefaultSimulation();
        Benchmark bench {};
        auto portHead = os::PortHeadT< int >::create(sim, "Benchmark Port",
            os::PortMode::ZERO_MODE, 1000000); // ZERO_MODE to avoid
            warns
        auto portTail = portHead->getTail();
        decltype( portHead->get() ) tmp; // temporary to avoid compiler to
            delete the symbol

        /* benchmark put */
        std::cout << "==== input ==== " << std::endl;
        bench.bench(100, [&portTail]() {
            for(int i = 0; i < elements; ++i)
                portTail->put( i );
        });

        /* benchmark get */
        std::cout << "==== output ==== " << std::endl;
        bench.bench(100, [&portHead, &tmp]() {
            for(int i = 0; i < elements; ++i)
                tmp = portHead->get();
        });

        /* combined - push one in, get one out */

        std::cout << "==== combined(1) ==== " << std::endl;
    }
}
```

D.4. Performanz der veränderten Portimplementation

```
bench.bench(100, [&portHead, &portTail, &tmp]() {
    for(int i = 0; i < elements; ++i) {
        portTail->put( i );
        tmp = portHead->get();
    }
});

/* combined - push 1000 in, get 1000 out */

std::cout << "==== combined(2) ==== " << std::endl;
bench.bench(100, [&portHead, &portTail, &tmp]() {
    for(int j = 0; j < 10; ++j) {
        for(int i = 0; i < 1000; ++i)
            portTail->put( i );
        for(int i = 0; i < 1000; ++i)
            tmp = portHead->get();
    }
});

}
{
    std::cout << "==== Teststructure" << std::endl;
    odemx::base::Simulation& sim = odemx::getDefaultSimulation();
    Benchmark bench {};
    auto portHead = os::PortHeadT< Teststructure >::create(sim, "Benchmark Port",
        os::PortMode::ZERO_MODE, 1000000); // ZERO_MODE to avoid
        warns
    auto portTail = portHead->getTail();
    decltype( portHead->get() ) tmp; // temporary to avoid compiler to
        delete the symbol

    /* benchmark put */
    std::cout << "==== input ==== " << std::endl;
    bench.bench(100, [&portTail]() {
        for(int i = 0; i < elements; ++i)
            portTail->put( i );
    });

    /* benchmark get */
    std::cout << "==== output ==== " << std::endl;
    bench.bench(100, [&portHead, &tmp]() {
        for(int i = 0; i < elements; ++i)
            tmp = portHead->get();
    });

    /* combined - push one in, get one out */

    std::cout << "==== combined(1) ==== " << std::endl;

    bench.bench(100, [&portHead, &portTail, &tmp]() {
        for(int i = 0; i < elements; ++i) {
            portTail->put( i );
            tmp = portHead->get();
        }
    });
}
```

D. Programme zur Messung von Laufzeitverhalten

```
/* combined - push 1000 in, get 1000 out */

std::cout << "==== combined(2) ==== " << std::endl;
bench.bench(100, [&portHead, &portTail, &tmp]() {
    for(int j = 0; j < 10; ++j) {
        for(int i = 0; i < 1000; ++i)
            portTail->put( i );
        for(int i = 0; i < 1000; ++i)
            tmp = portHead->get();
    }
});

}
// benchmark pointer container
{
    std::cout << "==== int*" << std::endl;
    odemx::base::Simulation& sim = odemx::getDefaultSimulation();
    Benchmark bench {};

    auto portHead = os::PortHeadT< int* >::create(sim, "Benchmark Port",
        os::PortMode::ZERO_MODE, elements); // ZERO_MODE to avoid
        warns
    auto portTail = portHead->getTail();
    decltype( portHead->get() ) tmp; // temporary to avoid compiler to
        delete the symbol

    /* benchmark put */
    std::cout << "==== input ==== " << std::endl;
    bench.bench(100, [&portTail]() {
        for(int i = 0; i < elements; ++i)
            portTail->put( new int { i } );
    });

    /* benchmark get */
    std::cout << "==== output ==== " << std::endl;
    bench.bench(100, [&portHead, &tmp]() {
        for(int i = 0; i < elements; ++i)
            tmp = portHead->get();
    });

    /* combined - push one in, get one out */

    std::cout << "==== combined(1) ==== " << std::endl;

    bench.bench(100, [&portHead, &portTail, &tmp]() {
        for(int i = 0; i < elements; ++i) {
            portTail->put( new int { i } );
            tmp = portHead->get();
        }
    });

    /* combined - push 1000 in, get 1000 out */
    std::cout << "==== combined(2) ==== " << std::endl;
    bench.bench(100, [&portHead, &portTail, &tmp]() {
        for(int j = 0; j < 10; ++j) {
            for(int i = 0; i < 1000; ++i)
                portTail->put( new int { i } );
        }
    });
}
```

D.4. Performanz der veränderten Portimplementation

```
        for(int i = 0; i < 1000; ++i)
            tmp = portHead->get();
    }
});
}

/* benchmark scaling the number of elements in the port */
{
    odemx::base::Simulation& sim = odemx::getDefaultSimulation();
    Benchmark bench {};
    std::cout << "==== Scaling Input/Output ==== " << std::endl;
    {
        auto benchmark_function = [&sim]( int n ) -> void {
            auto portTail = os::PortTailT< int >::create(sim, "Benchmark Port",
                os::PortMode::ZERO_MODE, elements); // ZERO_MODE to avoid warnings
            auto portHead = portTail->getHead();
            decltype( portHead->get() ) tmp;

            for( int i = 0; i < n; ++i ) {
                portTail->put( i );
                tmp = portHead->get();
            }
        };

        auto generator = []( int n ) -> int { return n * 10; };

        bench.bench< int >( 10, benchmark_function, 10, elements, generator );
    }
    std::cout << "==== Scaling buffer input/output " << std::endl;
    {
        volatile int access_element;
        auto benchmark_function = [&access_element, &sim]( int n ) -> void {
            auto portTail = os::PortTailT< int >::create(sim, "Benchmark Port",
                os::PortMode::ZERO_MODE, elements); // ZERO_MODE to avoid warnings
            typename std::remove_const< typename std::remove_reference<
                decltype( portTail->getBuffer() ) >::type >::type buffer;

            for( int i = 0; i < n; ++i ) {
                buffer.push_back( i );
                access_element = buffer.front();
                buffer.pop_front();
            }
        };

        auto generator = []( int n ) -> int { return n * 10; };

        bench.bench< int >( 10, benchmark_function, 10, elements, generator );

        std::cout << "Access happened: " << access_element << std::endl;
    }
}
}
```

src/port_old_implementation.cpp

D. Programme zur Messung von Laufzeitverhalten

```
/**
 * @file Teststructure.h
 * @brief A simple aggregate
 * @author Magnus Mueller
 * @date 2012-03-22
 */

#pragma once
#ifndef _TESTSTRUCTURE_H
#define _TESTSTRUCTURE_H

struct Teststructure {
    int a;
    int b;
    double c;
    double e;
    char f;

    Teststructure() {}
    Teststructure(int) {}
};

#endif
```

src/Teststructure.h

Literaturverzeichnis

- [Abr09] ABRAHAM, Dave: *Move It With Rvalue References*. <http://cpp-next.com/archive/2009/09/move-it-with-rvalue-references/>. Version: 2009
- [Ale01] ALEXANDRESCU, Andrei: *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001
- [BV11] BV, TIOBE S.: *TIOBE Programming Community Index for August 2011*. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Version: 2011
- [CDDA09] COLVIN, Greg ; DAWES, Beman ; DIMOV, Peter ; ADLER, Darin: *Smart Pointers*. http://www.boost.org/doc/libs/1_47_0/libs/smart_ptr/smart_ptr.htm. Version: 2009
- [Con63] CONWAY, Melvin E.: Design of a separable transition-diagram compiler. In: *Commun. ACM* 6 (1963), July, 396–408. <http://dx.doi.org/http://doi.acm.org/10.1145/366663.366704>. – DOI <http://doi.acm.org/10.1145/366663.366704>. – ISSN 0001–0782
- [EMSS07] E. MILLER, David ; SUTTER, Herb ; STROUSTRUP, Bjarne: *Strongly Typed Enums (revision 3)*. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2347.pdf>. Version: 2007
- [FA96] FISCHER, Joachim ; AHRENS, Klaus: *Objektorientierte Prozeßsimulation in C++*. Addison-Wesley, 1996
- [GCC11] GCC, project: *C++0x Support in GCC*. <http://gcc.gnu.org/projects/cxx0x.html>. Version: 6 2011
- [Ger03] GERSTENBERGER, Ralf: *Neue Lösungen für die Realisierung von C++-Bibliotheken zur Prozesssimulation*, Humboldt-Universität zu Berlin, Diplomarbeit, 2003

- [Gre06] GREGOR, Douglas: A Brief Introduction to Variadic Templates. (2006). <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2087.pdf>
- [HDA02] HINNANT, Howard E. ; DIMOV, Peter ; ABRAHAMS, Dave: The Forwarding Problem: Arguments. (2002), 9. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1385.htm>
- [Hin05] HINNANT, Howard E.: *Rvalue Reference Recommendations for Chapter 20*. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1856.html>. Version: 2005
- [Hin06] HINNANT, Howard E.: *Howard's STL / Move Semantics Benchmark*. <http://cpp-next.com/archive/2010/10/howards-stl-move-semantics-benchmark/>. Version: 5 2006
- [HSK08] HINNANT, Howard E. ; STROUSTRUP, Bjarne ; KOZICKI, Bronek: *A Brief Introduction to Rvalue References*. <http://www.artima.com/cppsource/rvalue.html>. Version: 3 2008
- [ISO03] ISO/IEC: *ISO/IEC 14882:2003(E): Programming Languages - C++*. ISO/IEC, 2003
- [ISO05] ISO/IEC: *Draft Technical Report on C++ Library Extensions*. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf>. Version: 2005
- [ISO11] ISO/IEC: *Working Draft, Standard for Programming Language C++*. ISO/IEC, 2011
- [KFA07a] KLUTH, Ronald ; FISCHER, Joachim ; AHRENS, Klaus: Ereignisorientierte Computersimulation mit ODEMx. (2007)
- [KFA07b] KLUTH, Ronald ; FISCHER, Joachim ; AHRENS, Klaus: *Ereignisorientierte Computersimulation mit ODEMx*. Version: November 01 2007. <http://edoc.hu-berlin.de/series/informatik-berichte/218/PDF/218.pdf>
- [SS03] SUTTER, Herb ; STROUSTRUP, Bjarne: A name for the null pointer: nullptr. (2003), 5. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1488.pdf>

- [Str95] STROUSTRUP, Bjarne: Why C++ is not just an object-oriented programming language. In: *SIGPLAN OOPS Mess.* 6 (1995), October, 1–13. <http://dx.doi.org/http://doi.acm.org/10.1145/260111.260207>. – DOI <http://doi.acm.org/10.1145/260111.260207>. – ISSN 1055–6400
- [Str96] STROUSTRUP, Bjarne: History of programming languages—II. Version: 1996. <http://dx.doi.org/http://doi.acm.org/10.1145/234286.1057836>. New York, NY, USA : ACM, 1996. – DOI <http://doi.acm.org/10.1145/234286.1057836>. – ISBN 0–201–89502–1, Kapitel A history of C++: 1979–1991, 699–769
- [Str00] STROUSTRUP, Bjarne: *The C++ Programming Language*. Addison-Wesley, 2000
- [Str09] STROUSTRUP, Bjarne: What is C++0x? In: *CVu* 21 (2009), November, Nr. 5, S. 21. – ISSN 1354–3164
- [Str11a] STROUSTRUP, Bjarne: *C++0x FAQ*. <http://www2.research.att.com/~bs/C++0xFAQ.html>. Version: 6 2011
- [Str11b] STROUSTRUP, Bjarne: *What were the aims of the C++0x effort?* <http://www2.research.att.com/~bs/C++0xFAQ.html#aims>. Version: 6 2011