

Die Programmiersprache D

1. November 2007

Inhaltsverzeichnis

Abkürzungsverzeichnis	III
Tabellenverzeichnis	IV
1 Einführung	5
1.1 Informationen und Geschichte zu D	5
1.2 D-Compiler	5
2 Grundlagen der Sprache	7
2.1 Kommentare	7
2.2 Datentypen	8
2.3 Steuerzeichen	10
2.4 Spezielle Token	11
2.5 Statements	11
2.5.1 foreach und foreach_reverse	11
2.5.2 with	12
2.6 Deklaration von Variablen	13
2.7 Speicherklassen von Variablen	13
2.8 Arrays	14
2.8.1 Statische Arrays	14
2.8.2 Dynamische Arrays	15
2.8.3 Arrays und Pointer	16
2.8.4 Assoziative Arrays	16
2.8.5 Spezielle Array-Operationen	17
2.9 Strings	18
2.10 Module und Pakete	20
2.10.1 Auflösung von Namensräumen	21
2.10.2 Statische Imports	22
2.10.3 Imports umbenennen	23
2.10.4 Selektive Imports	23
2.11 Funktionen	23
3 Benutzerdefinierte Typen	26
3.1 typedef und alias	26
3.2 enum	26
3.3 Strukturen	27
3.4 Klassen	28
3.5 Interfaces	28
3.6 Implizite und Explizite Konvertierung	29
4 Fortgeschrittene OOP-Techniken	31
4.1 Funktionszeiger und Delegates	31
4.2 Properties	32
4.3 Operatoren überladen	33
5 Templates	35
5.1 Rekursive Templates	36
5.2 Template Mixins	37
6 Boxing	39
7 Fehlerbehandlung	40
7.1 Exceptions	40

7.2 scope	41
8 Garbage Collection	43
9 Die Standardbibliothek Phobos	45
10 Schlussbemerkung	46
Anhang	47
Literaturverzeichnis	50

Abkürzungsverzeichnis

GC Garbage Collector oder Garbage Collection

DMD Digital Mars D Compiler

GNU GNU is not Unix

GCC GNU Compiler Collection

GDC GNU D Compiler

OOP Objektorientierte Programmierung

STL Standard Template Library

IDE Integrated Development Environment

Tabellenverzeichnis

Tabelle 1: Ganzzahlige Datentypen	8
Tabelle 2: Fließkomma-Datentypen	8
Tabelle 3: Datentypen zur Darstellung von Zeichen	9
Tabelle 4: Weitere Datentypen	9
Tabelle 5: Steuerzeichen	10
Tabelle 6: Spezielle Token	12
Tabelle 7: Properties von statischen Arrays	15
Tabelle 8: Properties von dynamischen Arrays	16
Tabelle 9: Properties von assoziativen Arrays	18
Tabelle 10: Properties von enums	29
Tabelle 11: Properties von allen Typen	34

1 Einführung

1.1 Informationen und Geschichte von D

D ist eine Systemprogrammiersprache, die im Dezember 1999 von Walter Bright erfunden und am 3. Januar 2007 in der Version 1.0 freigegeben wurde. D lehnt sich von der Syntax stark an C(++) an und übernimmt auch einige Konzepte der Sprache (z.B. Templates, OOP und strukturierte Programmierung). Anderes hat man hingegen verworfen (z.B. Mehrfachvererbung, umfangreiche implizite Typkonvertierung). Ideen für D hat man auch in den Sprachen Java und C# gefunden, wobei Delegates, Template Mixins und Garbage Collection die auffälligsten Beispiele sind.

Mit D ist sowohl eine sehr moderne Programmierweise mit OOP, Design Patterns oder Metaprogrammierung als auch eine sehr systemnahe mittels Binärkompatibilität zur C-ABI und Inline Assembler möglich.

Da D noch nicht lange auf dem Markt ist, gibt es wenig Informationen zu D. Aus diesem Grund baut diese Seminararbeit auf der offiziellen D Language Specification auf, die im Moment die exaktesten Informationen zu D liefern kann.

1.2 D-Compiler

Es existieren derzeit zwei Compiler für D: Der offizielle Digital Mars D-Compiler (DMD) und das Frontend GNU D Compiler (GDC) für die GNU Compiler Collection (GCC). Während der DMD-Compiler nur für Windows und GNU/Linux zur Verfügung steht, existiert der freie GDC praktisch für alle Plattformen, auf denen auch eine Portierung des GCC existiert. Das umfasst z.B. *BSD, Solaris und Mac OS X.

Im Unterschied zu der Mehrheit der Compiler für andere Programmiersprachen wirft der DMD-Compiler keine Warnungen aus, sondern zeigt nur Fehler an, d.h. Warnungen werden zu Fehlern gemacht. Der Verzicht auf Warnungen resultiert daher, dass viele C-Compiler zum Beispiel sehr tolerant sind, wenn es darum geht, Pointer zu casten. In vielen Fällen erzeugt so ein Code eine Warnung, die aber vom Programmierer ignoriert wird oder aufgrund einer zu niedrigen Warnstufe einfach nicht ausgegeben wird. Da ein mit Warnungen behafteter Code aber sehr schnell zum Problem werden kann (undefiniertes Verhalten, Probleme bei Portierungen), hat man sich beim

DMD-Compiler darauf festgelegt, den Programmierer mit dem Weglassen von Warnungen zur Beseitigung der heiklen Codestelle zu zwingen.

D-Code wird vom Compiler direkt in Maschinencode übersetzt, um eine möglichst hohe Ausführungsgeschwindigkeit zu erzielen.

2 Grundlagen der Sprache

Anweisungen werden, wie in C(++), mit einem Semikolon abgeschlossen (ein einzelnes Semikolon stellt auch die kürzeste Anweisung dar), Groß- und Kleinschreibung ist relevant und Variablennamen dürfen nicht mit Zahlen beginnen. Schlüsselwörter sind reserviert und dürfen nicht für Variablennamen herangezogen werden. Die Syntax wurde größtenteils von C(++), übernommen, um potentielle Nutzer nicht noch mit einer neuen Syntax zu belasten, sondern den Aufwand für einen Umstieg von C(++), Java oder C# möglichst gering zu halten.

Jedes D-Programm braucht einen Einstiegspunkt. Und wie viele andere Sprachen auch, dient in D die so genannte main-Funktion diesem Zweck. Diese Funktion wird, nachdem statische Variablen usw. allokiert wurden, beim Programmstart aufgerufen und darf nur ein einziges mal in einem Programm existieren.

Es gibt folgende Möglichkeiten, die main-Funktion zu deklarieren:

```
void main(char[][] args) { }  
void main() { }  
int main(char[][] args) { return 0; }  
int main() { return 0; }
```

Die char-Matrix 'args' enthält die an das Programm übergebenen Kommandozeilenparameter und ist optional.

Auch in D finden sich die u.a. aus C(++), bekannten Pointer, es lassen sich aber alle Situationen, wie z.B. Polymorphie oder dynamische Speicherverwaltung, ohne Pointer lösen (siehe Abschnitt über Klassen für Anwendungsbeispiele). D verwendet für Typen wie dynamische Arrays oder Klasseninstanzen Referenzen, die denen aus Java sehr ähnlich sind. Sie können *null* sein und auch auf ein anderes Objekt zeigen, als das Ursprüngliche. Im Inneren sind diese Referenzen natürlich immer noch Pointer, jedoch kann man auf sie keine Pointer-Arithmetik anwenden, welche bei C(++), eine Risikoquelle war.

2.1 Kommentare

Es gibt drei verschiedene Schreibweisen für Kommentare:

1. /* Kommentar */
2. /+ Kommentar +/
3. // Kommentar

Die ersten beiden Formen, so genannte Blockkommentare, können sich über mehrere Zeilen erstrecken und in Anweisungen eingebettet werden, während die letzte Form immer nur bis zum Zeilenende auskommentiert. Die zweite Form darf sogar verschachtelt werden, allerdings fördert dies natürlich nicht die Lesbarkeit des Quellcodes.

Da die Inhalte von Kommentaren und Strings vom Parser nicht näher analysiert werden, läutet ein `/*` innerhalb eines Strings keinen Kommentar ein, so wie doppelte Hochkommata innerhalb eines Kommentars nicht das Schließen des Kommentars:

```
char x[] = "Dies ist /*ein*/ String";    //Wird als Dies ist /*ein*/ String
                                        //ausgegeben.
int a = /* "Kommentar" */ 7; //a bekommt den Wert sieben zugewiesen
```

2.2 Datentypen

Viele Datentypen, die man von C++ kennt, findet man auch in D. Es gibt jedoch auch einige neue Datentypen, z.B. komplexe Zahlen. Steht bei einem Typ `signed`, so heißt das, dass er vorzeichenbehaftet ist, also sowohl positive als auch negative Zahlen darstellen kann, während `unsigned` Datentypen nur positive Werte darstellen können.

Es existieren als ganzzahlige Datentypen:

Schlüsselwort	Bezeichnung	Initialwert	Suffix
byte	signed 8 bits	0	-
ubyte	unsigned 8 bits	0	-
short	signed 16 bits	0	-
ushort	unsigned 16 bits	0	-
int	signed 32 bits	0	-
uint	unsigned 32 bits	0	U, u
long	signed 64 bits	0L	L
ulong	unsigned 64 bits	0L	UL, uL
cent	signed 128 bits (für die Zukunft reserviert, noch nicht benutzbar)	0	-
ucent	unsigned 128 bits (für die Zukunft reserviert, noch nicht benutzbar)	0	-

Vgl. D Language Specification, S. 31

Tabelle 1: Ganzzahlige Datentypen

als Fließkomma-Datentypen:

Schlüsselwort	Bezeichnung	Initialwert	Suffix
float	Fließkommazahl mit 32 bit	float.nan	F, f
double	Fließkommazahl mit 64 bit	double.nan	-
real	Größte darstellbare Fließkommazahl (auf Intel werden 80 bits verwendet)	real.nan	L
ifloat	Imaginäres float	float.nan * 1.0i	fi
idouble	Imaginäres double	double.nan * 1.0i	i
ireal	Imaginäres real	real.nan * 1.0i	Li
cfloat	Komplexe Zahl aus zwei floats	float.nan + float.nan * 1.0i	-
cdouble	Komplexe Zahl aus zwei double	double.nan + double.nan * 1.0i	-
creal	Komplexe Zahl aus zwei real	real.nan + real.nan * 1.0i	-

Vgl. D Language Specification, S. 31

Tabelle 2: Fließkomma-Datentypen

Datentypen zur Darstellung von Zeichen:

Schlüsselwort	Bezeichnung	Initialwert	Suffix
char	unsigned 8 bit UTF-8	0xFF	c
dchar	unsigned 16 bit UTF-16	0xFFFF	d
wchar	unsigned 32 bit UTF-32	0x0000FFFF	w

Vgl. D Language Specification, S. 31

Tabelle 3: Datentypen zur Darstellung von Zeichen

Weitere Datentypen:

Schlüsselwort	Bezeichnung	Initialwert
void	typlos	kein Initialwert
bool	bool'scher Wert	false

Vgl. D Language Specification, S. 31

Tabelle 4: Weitere Datentypen

Alle elementaren Datentypen haben, im Gegensatz zu C++, einen definierten Initialwert, der über

das Property `datatype.init` abgefragt werden kann (zum Beispiel `float.init`). Es ist aber möglich, die Default-Initialisierung zu unterbinden, indem man der Variablen bei der Deklaration umgehend den Wert `void` zuweist. Ein Zugriff auf diese Variable resultiert in undefiniertem Verhalten, bis man ihr einen Wert zugewiesen hat. Sinnvoll ist dieses Vorgehen, wenn man an kritischen Stellen Laufzeitverbesserungen erreichen möchte.

2.3 Steuerzeichen

Steuerzeichen, die auch als Escape-Sequenzen bezeichnet werden, werden unter anderem dazu benutzt, die Ausgabe in der Konsole zu manipulieren oder spezielle Zeichen (z.B. `\`) auszugeben.

Folgende Steuerzeichen stehen in D zur Verfügung:

Steuerzeichen	Funktion
<code>\'</code>	Ausgabe des Zeichens <code>'</code>
<code>\</code>	Ausgabe des Zeichens <code>\</code>
<code>\?</code>	Ausgabe des Zeichens <code>?</code>
<code>\\</code>	Ausgabe des Zeichens <code>\</code>
<code>\a</code>	Ausgabe eines akustischen Signals (meistens durch den PC-Speaker)
<code>\b</code>	Bewegt den Cursor um eine Position nach links (Backspace)
<code>\f</code>	Führt einen Seitenumbruch durch (Formfeed)
<code>\n</code>	Zeilenumbruch (Newline)
<code>\r</code>	Cursor an den Anfang der Zeile setzen (Carriage Return)
<code>\t</code>	Ausgabe eines Tabulators
<code>\v</code>	Ausgabe eines vertikalen Tabulators (Vertical Tabulator)
<code>\x</code>	Ausgabe einer hexadezimalen Zahl
<code>\u</code>	Ausgabe einer vierstelligen hexadezimalen Sequenz: <code>\uABCD</code>
<code>\U</code>	Ausgabe einer achtstelligen hexadezimalen Sequenz: <code>\U00100010</code>
<code>\0</code>	Ausgabe einer oktalen Zahl
<code>\& var</code>	<code>var</code> kann hier z.B. den Wert <code>quot</code> haben, d.h. man kann hier die Steuerzeichen verwenden, die auch in HTML existieren

Vgl. D Language Specification, S. 7f

Tabelle 5: Steuerzeichen

2.4 Spezielle Token

D bietet einige spezielle Zeichen an, um beispielsweise Informationen über Zeile oder Datei herauszufinden, um diese Daten dann z.B. bei einem Fehler in ein Log-File abzuspeichern:

Token	Funktion
__FILE__	Der Dateiname der aktuellen Quellcodedatei
__LINE__	Die Zeilennummer der aktuellen Anweisung
__DATE__	Das Datum zum Zeitpunkt der Kompilierung im Format mmmm dd yy
__TIME__	Der Zeitpunkt, zu dem die Datei kompiliert wurde
__TIMESTAMP__	Der Zeitpunkt und das Datum der Kompilierung im Format www mmmm dd hh:mm:ss yyyy

Vgl. D Language Specification, S. 16

Tabelle 6: Spezielle Token

Diese Token werden allesamt bei der Kompilierung durch die entsprechenden Werte ersetzt, d.h. im Kompilat steht dann nicht mehr __LINE__ sondern z.B. 147.

2.5 Statements

Selbstverständlich bietet D neben einer Reihe von konditionalen und iterativen Anweisungen auch Statements wie beispielsweise *return* an, die aus anderen Sprachen bekannt sind und gleich funktionieren. Die *if*-, *while*- und *do while* Statements sind identisch zu denen aus C(++) und werden deshalb nicht näher besprochen. Switch wurde erweitert, so dass es nun auch Strings auswerten kann.

2.5.1 foreach und foreach_reverse

Die Statements *foreach* bzw. *foreach_reverse* sind iterative Anweisungen, die man bereits aus vielen Sprachen kennt, C++ z.B. besitzt in der STL eine *std::for_each* Funktion. Mit *foreach* kann man über Arrays, assoziative Arrays, Strukturen, Klassen oder Delegates iterieren, während *foreach_reverse* in umgekehrter Reihenfolge iteriert.¹

¹ Vgl. D Language Specification (2007), S. 69

Ein einfaches foreach-Beispiel:

```
import std.stdio;

int main() {
    char [] x = "Hallo";
    foreach (char c; x)
        writef(c, " ");

    writefln("");

    foreach (int index, char c; "Welt")
        writefln("Index: ", index, " Wert: ", c);
    return 0;
}
```

Die Ausgabe ist:

```
H a l l o
Index: 0 Wert: W
Index: 1 Wert: e
Index: 2 Wert: l
Index: 3 Wert: t
```

Um über Klassen zu iterieren, müssen diese die Methoden *opApply* bzw. *opReverseApply* implementieren. Ein Beispiel hierzu findet sich in der D Language Specification auf Seite 71ff.

2.5.2 with

Das *with*-Statement ist bereits aus JavaScript bekannt und dient dazu, häufigen Zugriff auf ein Objekt zu vereinfachen, indem es den Sichtbereich auf das Objekt legt und somit die Angabe des Objekts beim Zugriff auf Attribute oder Methoden überflüssig macht:

```
import std.stdio;

struct Foo {
    int x;
    void bar() { writefln("bar: ", x); }
};

int main() {
    Foo f;
    with (f) {
        x = 7;
        bar();
    }

    return 0;
}
```

2.6 Deklaration von Variablen

In der Deklaration von Variablen unterscheidet sich D von C++ in einigen Punkten. Sobald in D mehrere Variablen mit einer Anweisung deklariert werden, müssen sie denselben Typ besitzen, daher ist folgender Code unzulässig:

```
int a, *b;
```

Dieser Code kompiliert z.B. in beiden Sprachen, liefert aber unterschiedliche Ergebnisse:

```
int* a, b, c;
```

In C++ wird ein Pointer auf *int* 'a' und zwei normale *int*-Variablen 'b' und 'c' deklariert. In D hingegen deklariert man drei Pointer auf *int*, da der Typ ja für alle drei Variablen gleich sein muss.

Die Deklaration von Variablen bietet weitere Möglichkeiten:

```
int x = 12_34_56;           //gleich wie: int x = 123456;
int x = 0b0110             //Interpretiert 0110 als binäre Zahl
int x = 0xFF;              //0x leitet eine hexadezimale Zahl ein
int x = 0666;              //0 leitet eine oktale Zahl
float f = 12_34.567        //gleich wie: float f = 1234.567;
float f = 123._4e-5        //gleich wie: float f = 123.4e-5
```

2.7 Speicherklassen von Variablen

In D existieren auch *static* und *const*, die die gleiche Funktion wie in C(++) erfüllen. Das Schlüsselwort *auto* erhält eine neue Bedeutung.

Die Speicherklasse *auto* ermöglicht es, dass die explizite Angabe des Typs bei der Deklaration und der gleichzeitigen Definition einer Variablen entfallen kann:

```
auto x = 45;
auto y = 2.5f;
auto str = "Hallo Welt";
```

'x' erhält hier den Typ *int*, 'y' den Typ *float* und 'str' *char[10]*. Dieses Sprachmittel ist zwar sehr simpel, aber dennoch mächtig und vorteilhaft, da man niemals Datentypen anpassen muss, wenn sich der Initialwert einer Variablen ändert (der ja durchaus auch aus einem Funktionsaufruf stammen kann).

Diese Bequemlichkeit hat allerdings auch seine Schattenseiten. Man betrachte folgendes

Beispiel:

```
auto a = 7;
auto b = 3;

auto c = a/b;
```

'c' hat hier den Typ *int* und den Wert 2. Hätte man 'a' z.B. den Wert 7.3f zugewiesen, wäre 'a' vom Typ *float* und bei der Division wäre 'b' ebenso in *float* konvertiert worden, was dann ein 'c' vom Typ *float* mit dem Wert 2.43333 ergeben hätte. In diesem Fall hängt also sowohl der Typ als auch der Wert von 'c' von einem .3f nach der 7 ab.

Es ist daher empfehlenswert, den Einsatz von *auto* genau abzuwägen, um kein Programm zu schreiben, dessen Verhalten von kleinsten Details und Flüchtigkeitsfehlern abhängt.

2.8 Arrays

D stellt drei Typen von eingebauten Arrays zur Verfügung: Die normalen (statischen) Arrays, dynamische Arrays und so genannte assoziative Arrays. Die D Language Specification listet auf S. 82 auch Pointer auf Arrays (z.B. *int *p*) als eigenen Arraytyp auf, jedoch ist es so, dass *int *p* ein Zeiger auf ein einzelnes *int* ist, wobei dieses *int* auch das erste Element eines Arrays sein kann. Nur ist deshalb der Zeiger noch kein eigener Arraytyp. An dieser Stelle hat sich, aus C-Kompatibilitätsgründen, eine Inkonsistenz in der D Language Specification aufgetan.

D bietet weiterhin so genanntes Array Bounds-Checking an. Dies sorgt dafür, dass die Arraygrenzen von Arrays nicht über bzw. unterschritten werden. D versucht solche Fehler bereits zur Kompilierzeit aufzuzeigen, allerdings ist dies nicht immer möglich (z.B. wenn der Index dynamisch zur Laufzeit generiert wird) und daher führt ein unentdeckter Zugriff außerhalb der Arraysgrenzen dann zu einem Absturz des Programms. Die Fehlermeldung zeigt einem immerhin die Datei und die Zeile an, in der der Fehler auftrat.

2.8.1 Statische Arrays

Statische Arrays dürfen in D maximal 16 MB groß sein, allerdings empfiehlt es sich bei solchen Größen, ein dynamisches Array zu verwenden, um den Stack nicht unnötig zu belasten.

Bei der Deklaration von (mehrdimensionalen) Arrays hat man zwei Möglichkeiten: Zum Einen

der neue D-Stil, der von rechts nach links gelesen wird, und zum Anderen die bekannte C-Syntax, welche von links nach rechts gelesen wird.

Neue D-Syntax:

```
int[5] x = [1,2,3,4,5];           //x ist ein Array mit fünf Elementen (und wird
                                //gleich initialisiert)
int[2][3] y;                     //y ist ein Array von drei Arrays mit jeweils zwei ints
int[2]*[4] z;                    //z ist ein Array von vier Zeigern auf Arrays mit zwei ints
```

Alte C-Syntax:

```
int x[5] = {1,2,3,4,5};          //x ist ein Array mit fünf Elementen
int y[2][3];                    //y ist ein Array von zwei Arrays mit jeweils drei ints
int (*z[4])[2];                 //z ist ein Array von vier Zeigern auf Arrays mit zwei ints
```

Folgende Properties hat ein statisches Array in D:

Property	Beschreibung
.sizeof	Gibt die Größe des Arrays zurück (Anzahl Elemente * Größe eines Elements in Byte)
.length	Gibt die Anzahl der Elemente im Array zurück (read-only)
.ptr	Gibt einen Pointer auf das erste Element zurück
.dup	Kopiert das Array in ein neues, dynamisches Array und gibt dieses zurück
.reverse	Invertiert die Reihenfolge der Elemente und gibt das Array zurück (In-place Algorithmus)
.sort	Sortiert das Array und gibt das Array zurück (In-Place Algorithmus)

Vgl. D Language Specification, S. 86f

Tabelle 7: Properties von statischen Arrays

Properties werden in Abschnitt 4.2 näher erläutert.

2.8.2 Dynamische Arrays

Die dynamischen Arrays sind in D etwas mächtiger als ihre C++ Pendants, denn sie merken sich ihre Länge in der Property *.length*. Des Weiteren besteht ein dynamisches Array intern aus einem Pointer auf die Daten.

Dynamische Arrays werden folgendermaßen allokiert:

```
int []x = new int[20];
```

Diesen Speicher braucht man nicht mehr explizit freizugeben, da D über einen Garbage Collector verfügt (siehe Abschnitt 8).

Dynamische Arrays verfügen über gleichen Properties wie statische Arrays, allerdings haben die folgenden beiden Properties eine andere Bedeutung:

Property	Beschreibung
.sizeof	Liefert die Größe der Referenz auf das Array zurück. Bei 32-Bit Systemen ist dieser Wert immer 8 (4 Byte für die Länge und 4 Byte für den Pointer auf das Array)
.length	Liefert die Anzahl der Elemente im Array. Wird .length per Zuweisung verändert, wird das Array neu allokiert und die alten Daten werden, soweit möglich, in das neue Array kopiert (In-Place Algorithmus)

Vgl. D Language Specification, S. 87

Tabelle 8: Properties von dynamischen Arrays

2.8.3 Arrays und Pointer

Man kann auch in D Pointer-Arithmetik auf ein Array anwenden, wie man es aus C(++) gewohnt ist:

```
import std.stdio;

int main() {
    int [10] x;
    for (int i=0;i<x.length;++i)
        x[i] = getId();

    int *p = &x[0];
    while ( p <= &x[x.length-1] )
        writefln(*p++);
    return 0;
}
```

Es ist allerdings nicht möglich die Zeile `int *p = &x[0];` durch `int *p = x;` zu ersetzen, da man eine klare Grenze zwischen Arrays und Pointer ziehen wollte, die in C(++) nicht existiert. In C(++) geht das, da der Name eines Arrays wie ein Zeiger auf das erste Element des Arrays behandelt werden kann.

Des Weiteren sind Properties wie `.length` oder `.dup` nicht auf Pointer auf Arrays anwendbar, da der Pointer nicht über die notwendigen Informationen verfügt.

2.8.4 Assoziative Arrays

Assoziative Arrays bilden jeweils ein Schlüssel-Wert Paar ab und sind aus C++ unter der STL-Klasse `std::map<T, V>` bekannt. Der Typ des Schlüssels darf keine Funktion oder `void` sein,

ansonsten sind alle Typen erlaubt. Der Schlüssel des assoziativen Arrays muss zudem eindeutig sein, da sonst eine korrekte Zuordnung nicht gewährleistet werden kann.

Die komplette Funktionalität von assoziativen Arrays wurde in D direkt in die Sprache eingebaut:

```
import std.stdio;

int main() {
    char [][] zahlen = ["eins", "zwei", "drei", "vier", "fuenf"];

    char [][] x;          //Das assoziative Array

    auto key = 0;
    foreach (char [] iter; zahlen)
        x[++key] = iter;

    x.remove(5);          //Wir entfernen das Paar mit dem Schlüssel 5

    char []* p = 1 in x;  //Verprobung, ob sich der Schlüssel 1 in
                          //x befindet
    if ( p != null)      //Falls p null ist, wurde der Schlüssel nicht
                          //gefunden

    writefln("Wert zu Schluessel 1: ", *p);

    writefln("\n", x);    //Ausgabe: [1:eins,2:zwei,3:drei,4:vier]
    return 0;
}
```

Ein assoziatives Array hat zudem folgende, zusätzliche Properties:

Property	Beschreibung
.keys	Gibt ein dynamisches Array mit den Schlüsseln zurück
.values	Gibt ein dynamisches Array mit den Werten zurück
.rehash	Baut das Array frisch auf, sodass Suchanfragen schneller beantwortet werden können und gibt eine Referenz auf das neue Array zurück

Vgl. D Language Specification, S. 95

Tabelle 9: Properties von assoziativen Arrays

Als Schlüssel für ein assoziatives Array kann auch ein *struct*, *union* oder *class* verwendet werden.

Allerdings muss für diese Typen dann eine eigene hash-Funktion (zur eindeutigen Unterscheidbarkeit) und eine Vergleichsfunktion geschrieben werden. Die D Language Specification stellt ab Seite 93ff Implementierungen dieser Funktionen zur Verfügung.

2.8.5 Spezielle Array-Operationen

Da Arrays ein häufig gebrauchtes Mittel in der Programmierung sind, wurde ihnen noch weitere

Funktionalitäten verliehen, die bisher nur aus Skriptsprachen wie Python bekannt waren. Nun ist es in D möglich, einfach Subarrays zu bilden (Slicing), ganze Array-Bereiche auf ein mal zu kopieren (Array copying), Arrays aneinanderzufügen (Concatenation, funktioniert nur bei dynamischen Arrays) und sogar Funktionen als Properties für Arrays aufzurufen (Function Properties)!

Nachstehendes Codelisting soll alle Funktionalitäten in ihrer Anwendung aufzeigen:

```
import std.stdio;

void power_2(int []arr) {
    foreach (int x; arr)
        writeln(x,"*",x," = ",x*x);
}

int main() {
    int [4] foo = [0,1,2,3];
    int [10] bar = [0,0,0,0,4,5,6,7,8,9];

    //Subarrays bilden:
    writeln(bar[6..8]);    //Ausgabe: [6,7]

    //Gruppenweise setzen:
    bar[0..4] = -1;
    writeln(bar[0..4]);    //Ausgabe: [-1,-1,-1,-1]

    //Kopieren:
    bar[0..4] = foo[0 .. length];    //length ist hier foo.length
    writeln(bar);    //Ausgabe: [0,1,2,3,4,5,6,7,8,9]

    //Aneinanderfügen:
    int [] x = [12,11,10];
    x ~= bar.reverse;    //Äquivalent zu: x = x ~ bar.reverse;
    writeln(x);    //Ausgabe: [12,11,10,9,8,7,6,5,4,3,2,1,0]

    //Funktionen als Properties
    bar.power_2();    //Äquivalent zu power_2(bar);

    return 0;
}
```

Man sieht, in D sind Arrays fast das, was `std::vector<T>` in C++ ist, nur dass die Arrays leichter zu handhaben und direkt in die Sprache eingebaut sind.

2.9 Strings

In D sind Strings, wie in C(++), Arrays von *chars* (bzw. *dchars* oder *wchars*), jedoch ist keine Terminierung mit `\0` mehr notwendig, außer man möchte das String-Array an eine C-Funktion

übergeben.

Es existieren drei verschiedene Typen von Strings in D:

1. Ein normaler String: `char x[] = Hallo Welt ;`
2. Ein WYSIWYG-String: `char x[] = r Hallo\n Welt ;`
3. Ein Hex-String: `char x[] = x 0A ;`

Ein String wird immer von doppelten Hochkommata eingeschlossen, nur WYSIWYG-Strings können auch mit ``Hallo\n Welt`` angegeben werden.

WYSIWYG (What You See Is What You Get) Strings haben die Eigenschaft, in ihnen eingebettete Steuerzeichen bei der Ausgabe zu ignorieren, d.h. würde man obigen String ausgeben, würde man nicht etwa einen Zeilenumbruch, sondern ein `'\n'` lesen. Besonders interessant sind WYSIWYG-Strings bei der Angabe von Windows-Pfadnamen:

```
char []path = r"C:\WINNT\SYSTEM32\calc.exe"
```

Normalerweise würde ein `\` eine Escape-Sequenz einleiten, aber durch das Benutzen eines WYSIWYG-Strings wird dies vermieden.

Hex-Strings werden hauptsächlich für die Low-Level Programmierung benutzt. Bei Hex-Strings werden Leerstellen sowie Zeilenumbrüche ignoriert, so dass die Formatierung einfacher fällt.

Außerdem müssen die angegebenen Hex-Zeichen ein Vielfaches von zwei sein.

Da Strings dynamische Arrays sind, stehen auch alle Array-Operationen für Sie zur Verfügung:

```
import std.stdio;

int main() {
    char [] x = "Hallo";
    char [] y = "Welt";
    char [] z = "Geld";

    //Zusammenfügen:
    writefln(x ~ " " ~ y); //Ausgabe: Hallo Welt

    //Zuweisen:
    y = z;
    writefln(x ~ " " ~ y); //Ausgabe: Hallo Geld

    y = "hallo";

    //Lexikalischer Vergleich:
    if (x < y)
        writefln(x, " ist kleiner als ", y);
    else if (x == y)
```

```
        writefln(x, " ist gleich wie ", y);
    else
        writefln(y, " ist kleiner als ", x);

    //Anhängen:
    x ~= " schönes ";
    x ~= z;

    writefln(x);      //Ausgabe: Hallo schönes Geld
    return 0;
}
```

Obwohl vorerst auf eine eigene String Klasse verzichtet wurde, lässt sich mit den *char*-Arrays recht komfortabel arbeiten.

2.10 Module und Pakete

Sobald man umfangreiche Programme schreibt, wird man den Quellcode auf mehrere Dateien verteilen wollen, um zum Einen die Übersicht zu behalten und zum Anderen ein gewisses Abstraktionslevel einzuführen. In D gibt es zu diesem Zweck Module.

Jede Quellcodedatei ist automatisch ein Modul, das standardmäßig den Namen der Quellcodedatei erhält. Um Namenskonflikte mit Inhalten aus anderen Modulen zu vermeiden, stellen Module einen Namensraum für ihre Inhalte zur Verfügung.

Module haben folgende Eigenschaften:

1. Es gibt nur eine, statisch allokierte, Instanz des Moduls
2. Module können nicht vererben
3. Pro Datei gibt es nur ein einziges Modul
4. Module können importiert werden, um auf die Inhalte zuzugreifen
5. Module werden immer global kompiliert, d.h. sie stehen im gesamten Programm zur Verfügung, wenn man sie einbindet
6. Alle Klassen innerhalb eines Moduls haben Zugriff auf die privaten Elemente der anderen Klassen im selben Modul

Man kann Module in Hierarchien zu so genannten Packages gruppieren. Durch Module entfällt auch die Notwendigkeit, die Deklaration und Definition in *.cpp und *.hpp Dateien aufzuteilen, wie dies bei C(++) der Fall ist. Zusätzlich verringert sich die Kompilierzeit, da Abhängigkeiten leichter aufzulösen sind als mit dem Header-Konzept.

Da Module automatisch für jede Quellcodedatei erstellt werden, muss man sie nicht speziell deklarieren. Möchte man ihnen aber einen anderen Namen geben, ist eine Deklaration erforderlich:

```
module math;

int add(int a, int b) { return a+b; }
//...
```

Hier bekommt das Modul den Namen 'math'. Möchte man das Modul zusätzlich in ein Package einbinden, muss man dies ebenfalls angeben:

```
module mypackage.math;

int add(int a, int b) { return a+b; }
//...
```

Hier wird also das Modul 'math' im Package 'mypackage' deklariert. Es ist empfehlenswert, sowohl Modul als auch Packagenamen klein zu schreiben, da eine direkte Verbindung zu einer Datei besteht und manche Dateisysteme nicht case-sensitive sind. Schreibt man die Namen immer klein, werden mögliche Schwierigkeiten beim Wechsel des Dateisystems bereits im Voraus beseitigt. Möchte man ein Modul nun in seinen Quellcode einbinden, steht einem dabei die import-Anweisung in den folgenden, grundlegenden Varianten zu Verfügung:

```
import std.stdio;
import std.math, std.string;
```

Die erste importiert das Modul 'stdio' aus dem Package 'std' und die zweite Anweisung importiert 'math' und 'string' aus dem 'std'-Package. Die Reihenfolge der imports ist für die Verwendung der Modulinhalte ohne Bedeutung, d.h. sollte eine Funktion aus 'stdio' auf einer anderen Funktion aus 'math' basieren, stellt obiger Code kein Problem dar, obwohl 'stdio' vor 'math' importiert wurde.

2.10.1 Auflösung von Namensräumen

Wenn man in D eine Funktion aufruft, wird immer zuerst versucht, die Funktion im aktuellen Namensraum zu finden. Gelingt dies nicht, wird versucht, die Funktion über die imports aufzulösen. Wird die Funktion gefunden, ist alles in Ordnung. Wird die Funktion aber in mehreren imports gefunden, wird ein Error ausgelöst.

Hierzu folgendes Beispiel:

```
//foo enthält Funktionen sendMessage(), receiveMessage() und parseMessage()
import foo;
//bar enthält ebenso die Funktion sendMessage(), aber andere Implementierung
import bar;

void receiveMessage() { /* */ }

int main() {
    receiveMessage(); //Ruft das lokale receiveMessage auf, da es vor den
                    //imports gefunden wird
    parseMessage(); //Ruft foo.parseMessage auf
    sendMessage(); //Führt zu einem Fehler, da die Funktion in beiden
                    //imports existiert
    return 0;
}
```

Der Sichtbereich von imports ist standardmäßig private, d.h. dass ein import (z.B. 'std.stdio') nur in der aktuellen Quellcodedatei (bzw. Modul) verfügbar ist. Wird dieses Modul dann wieder importiert, ist der import std.stdio nicht mehr zugänglich. Um dies zu umgehen, kann der import-Anweisung ein public vorangestellt werden:

```
public import std.stdio;
```

2.10.2 Statische Imports

Die Problematik des letzten Abschnitts, auf einen Funktionsaufruf mehrere Funktionen aus verschiedenen Module zu finden, wird mit einer hohen Anzahl an imports stark steigen und zu einigen Problemen führen. Dies kann man jedoch mit einer vollständig qualifizierenden Angabe von Package und Modulname verhindern. Und um den Entwickler zu dieser Angabe zu zwingen, können Module statisch importiert werden:

```
static import std.math;

int main() {
    int x = std.math.abs(-5);
    return 0;
}
```

Ein direkter Aufruf einer Funktion aus 'std.math' würde zu einem `Undefinierte Funktion` -Fehler führen.

2.10.3 Imports umbenennen

Sollte man seine Module in Packages gliedern, kann dies schnell zu sehr langen

Funktionsaufrufen führen, möchte man eine vollständig qualifizierenden Aufruf durchführen.

Es ist aber möglich, die Imports umzubenennen, um weniger schreiben zu müssen:

```
import str = std.string;

int main() {
    char [] x = "Hallo Welt";
    int len1 = str.strlen(x);           //Ruft std.string.strlen auf
    int len2 = std.string.strlen(x);   //Fehler: std undefiniert
    int len3 = strlen(x);              //Fehler, strlen undefiniert
    return 0;
}
```

2.10.4 Selektive Imports

Gelegentlich kann es vorkommen, dass man nur ein oder zwei Funktionen aus einem Modul

braucht und deshalb nicht den aktuellen Namensraum mit vielen Funktionen verseuchen

möchte, die man nicht braucht. D bietet zu diesem Zweck so genannte selektive Imports an, die folgendermaßen funktionieren:

```
import std.string : strlen;
import io = std.stdio : printf;

int main() {
    char *x = "Hallo Welt";
    int len = strlen(x);
    io.printf("Laenge des Strings: %d\n", len);
    return 0;
}
```

Der erste import importiert nur die Funktion *strlen* in den aktuellen Namensraum, während der zweite Import die Funktion *printf* importiert und dabei das Modul 'stdio' umbenennet.

Ein Aufruf einer anderen Funktion aus den Modulen führt nun zu einem Undefinierte

Funktion -Fehler und ein vollständig qualifizierender Aufruf bzw. das Weglassen von *io* bei einem Aufruf von *printf* resultiert ebenfalls in einem Fehler.

2.11 Funktionen

Auch in D existieren Funktionen als Mittel der strukturierten Programmierung. Sie besitzen die gleichen Fähigkeiten wie in C++ (z.B. Überladen, Default-Parameter), bringen aber auch einige

Neuerungen mit. Das `inline`-Schlüsselwort von C++ ist weggefallen, da der Compiler selbst entscheidet, ob es sinnvoll ist, die Funktion `inline` zu implementieren.

Um bei einer Parameterliste anzugeben, welche Parameter als Eingabe- und welche als Ausgabewerte fungieren, wurden die Schlüsselwörter `in`, `out`, `inout` und `lazy` eingeführt. Der Default-Wert eines Parameters ist `in`. Das bedeutet, in den Parameter kann nichts zurückgeschrieben werden. Bei `out` kann jedoch nichts entgegengenommen, sondern nur ausgegeben werden. Um beides, also Ein- und Ausgabe von Daten zu ermöglichen, existiert `inout`. Das, selten verwendete, Attribut `lazy` bewirkt, dass der übergebene Parameter erst beim ersten Aufruf ausgewertet wird.

Ein Beispiel zu Funktionen:

```
import std.stdio;

//a und b sind per default in
int sub(int a, int b) { return a-b; }

//a und b werden in c addiert, welches als Ausgabeparameter benutzt wird
void add(int a, int b, out int c) { c = a+b; }

//a wird als Ein- und Ausgabeparameter benutzt
void add(inout int a, int b) { a += b; }

int main() {
    writefln(sub(5,1));

    int erg;
    add(2, 9, erg);
    writefln(erg);

    erg=3;
    add(erg, 4);
    writefln(erg);

    return 0;
}
```

Ausgabe:

```
4
11
7
```

Wäre die 'sub'-Funktion ebenfalls 'add' genannt worden, hätte es einen Konflikt mit der dritten 'add'-Funktion gegeben. Die Schlüsselwörter `in`, `out` usw. beeinflussen also nicht die Signatur von Funktionen.

Eine weitere Neuerung ist die Möglichkeit, bei Funktionen Vor- und Nachbedingungen anzugeben, damit sich das Programm immer in einem konsistenten Zustand befindet. Dies ist auch unter dem Begriff Contract-Programming bekannt. Mit der Vorbedingung prüft man die Parameter und mit der Nachbedingung wird sichergestellt, dass die Funktion korrekt gearbeitet hat.

Beispiel zu Contract Programming:

```
import std.stdio;

struct Auto {
    //...
}

bool motor_anlassen(Auto a) {
    in {
        assert( a.kupplung().gedrueckt() == true );
    }
    out {
        assert( a.motor().laeuft() == true );
    }
    body {
        return a.motor().starten();
    }
}

int main() {
    Auto a;
    motor_anlassen(a);
    return 0;
}
```

Nun wird bei jedem Aufruf der *motor_anlassen*-Funktion auf Einhaltung der Bedingungen geachtet und bei Verletzung mit einem Laufzeitfehler reagiert.

3 Benutzerdefinierte Typen

In diesen Bereich fallen alle Typen, die nicht D anbietet, sondern vom Benutzer selbst definiert werden, also Klassen, Strukturen, *typedef* oder auch *alias*. Neu ist nur *alias*, aber *typedef* und Klassen unterscheiden sich im Vergleich zu C(++). Die Möglichkeiten von *enum* wurden erweitert, während *union* gleich wie in C(++) funktioniert und deshalb nicht näher besprochen wird.

3.1 *typedef* und *alias*

Das aus C++ bekannte Schlüsselwort *typedef* findet sich auch in D wieder, allerdings mit einer anderen Funktion. Bei diesem Code:

```
typedef int myint;
```

wird der Typ *myint* eingeführt, der sich auch semantisch von *int* unterscheidet. D.h. diese Typen sind, zumindest ohne einen cast, nicht austauschbar. *typedefs* haben jeweils einen Basistyp; In dem Fall ist der Basistyp *int*.

Die Funktion, die *typedef* in C++ hat, wird in D von dem Schlüsselwort *alias* wiedergegeben. Bei *alias* sind die Typen semantisch identisch.

```
alias string.strlen mylen;
```

'mylen' ist somit nur ein anderer Name für die Funktion *string.strlen*, d.h. beim Aufruf von 'mylen' wird in Wirklichkeit auch *string.strlen* aufgerufen.

3.2 *enum*

D unterstützt natürlich auch den Datentyp Enum, der für Aufzählungen benutzt wird. Im Gegensatz zu anderen Sprachen unterstützt D aber auch die Vererbung von enums:

```
enum WorkingWeek { Monday, Tuesday, Wednesday, Thursday, Friday };  
enum Week : WorkingWeek { Saturday, Sunday };
```

'WorkingWeek' ist hier der Basistyp und 'Week' der abgeleitete Typ. Gibt man keinen Basistyp an, wird dafür *int* benutzt. Auch die Angabe des Namens kann entfallen, um ein anonymes *enum* zu erzeugen, welches z.B. innerhalb einer Struktur gebraucht wird.

Enums haben folgende Properties:

Property	Beschreibung
.init	Der Wert des ersten Elements
.max	Der größte Wert im Enum
.min	Der kleinste Wert im Enum
.sizeof	Größe des für das Enum benutzen Speichers, auf 32-Bit Systemen typischerweise 4

Vgl. D Language Specification, S. 118

Tabelle 10: Properties von enums

3.3 Strukturen

Strukturen in D dienen dem Zweck, Daten zu aggregieren bzw. Low-Level Geräte anzusprechen. Sie unterstützen keine Vererbung und sind außerdem so genannte `Value Types`, während Klassen `Reference Types` sind. Dies bedeutet, man arbeitet bei Strukturen direkt auf der Instanz, während man bei Klassen mit einer Referenz auf die Instanz arbeitet (wie in Java).

Beispiel zu Strukturen:

```
import std.stdio;

//Struktur X
struct X {
    static double foo;      //statische Member-Variable
    uint bar;              //normale Member-Variable

    //statische Member-Funktion
    static void func1() { writefln("foo: ", foo); }

    //normale Member-Funktion
    void func2() { writefln("bar: ", bar); }
} //Kein ; mehr nötig

int main() {
    X.foo = -1.0;
    X.func1();

    X x;

    x.bar = 5;
    x.func2();

    return 0;
}
```

Die Ausgabe von diesem Programm ist:

```
foo: -1  
bar: 5
```

3.4 Klassen

Klassen sind in D das Mittel, um objektorientiert zu programmieren. Sie werden immer per Referenz instantiiert und haben immer eine Superklasse. Wie in Java gibt es auch in D eine Wurzelklasse, von der abgeleitet wird, wenn der Programmierer keine Superklasse angibt. Diese Klasse heißt *Object*. Auf Mehrfachvererbung wurde verzichtet, anstatt dessen gibt es Interfaces (siehe den folgenden Abschnitt).

Methoden und Attribute können entweder *private*, *protected* oder *public* sein, wobei sie standardmäßig *public* sind. Klassen, die sich im selben Modul befinden, können untereinander auf die privaten Attribute zugreifen. Weiter können Klassen so genannte Invariants und Unit Tests haben. Invariants sind Bedingungen, die immer wahr sein müssen und Unit Tests erlauben es, innerhalb einer Klasse Tests durchzuführen, um die Funktionsfähigkeit zu gewährleisten. Sowohl Invariants als auch Unit Tests werden direkt in den Quellcode eingebaut.

Nicht-statische, nicht-private und nicht-Template-Methoden sind standardmäßig *virtual* und werden, sofern die Methode nicht überschrieben wurde, nicht-virtual gemacht, da der Compiler über umfangreiche Informationen zur Klassenhierarchie verfügt und so im Schnitt sehr viele direkte Methodenaufrufe anstatt Zugriffe auf die vtable erzeugen kann.²

Das Beispiel zu Klassen befindet sich im Anhang, da es sehr umfangreich ist.

3.5 Interfaces

Da es in D keine Mehrfachvererbung gibt, hat man sich als Ausgleich für Interfaces entschieden. Interfaces stellen eine Sammlung von Methoden dar, die eine ableitende Klasse implementieren muss.³ Interfaces können voneinander, aber nicht von Klassen, abgeleitet werden. Es ist nicht möglich, sie zu instantiiieren und ein Interface darf nur die Methoden-Deklarationen beinhalten, nicht aber die Definitionen. Eine ableitende Klasse muss zwingend alle Methoden des Interfaces

² D Language Specification (2007), S. 119

³ D Language Specification (2007), S. 114

implementieren.

Beispiel zu Interfaces:

```
import std.stdio;

interface A { void foo(); }

interface B { void bar(); }

class C : A, B {
    void foo() { writefln("foo"); }
    void bar() { writefln("bar"); }
}

int main() {
    auto c = new C;
    c.foo();    //Ausgabe: foo
    c.bar();    //Ausgabe: bar
    return 0;
}
```

3.6 Implizite und Explizite Konvertierung

Implizite Konvertierungen finden bei einer Zuweisung des des abgeleiteten Typs an den Basistyp statt. Hierbei handelt es sich immer um benutzerdefinierte Typen wie *enum* oder *class*:

```
import std.stdio;

class A {
    /* ... */
}

class B : A {
    /* ... */
}

int main() {
    A a = new B();    //implizite Konvertierung ausreichend

    B b = cast(B) new A(); //explizite Konvertierung notwendig

    return 0;
}
```

Vieles andere (z.B. *float* nach *int*) muss mit einer expliziten Konvertierung, einem so genannten *cast* oder *Typcast*, umgewandelt werden. In C++ wäre z.B. folgender Code legal:

```
const float PI = 3.1416;
int x = PI;    //PI wird implizit in den Typ int umgewandelt, x hat den Wert 3
```

In D hingegen muss gecastet werden:

```
const auto PI = 3.1416;  
int x = cast(int)PI; //Auch hier bekommt x den Wert drei zugewiesen,  
allerdings mittels eines Casts
```

Die genauen Regeln für die Konvertierung finden sich auf Seite 33 der D Language Specification.

Der Zwang zu weitreichender expliziter Konvertierung ist ein weiterer Ansatz, um den Programmierer zu einem sauberen Programmierstil zu führen, indem er auch wirklich die Datentypen benutzt, die z.B. von einer Funktion erwartet werden. Denn so bequem eine umfassende implizite Konvertierung, wie die von C(++) auch ist, sie birgt folgende Gefahren:

1. Informationen können bei casts verloren gehen wie bei float nach int
2. Problematische Umwandlungen in kleinere Datentypen wie int nach short
3. Der Compiler ruft mit einer Konvertierung eine Funktion auf, die er gar nicht aufrufen soll
4. Es wird durch Umwandlungen Code generiert, den man gar nicht schreiben wollte

Alle diese Gefahren wurden in D durch die Einschränkung von impliziten Umwandlungen gebannt und das einzige Opfer ist etwas weniger Komfort beim Programmieren. Dennoch trägt diese Änderung entscheidend zur Sicherheit und Berechenbarkeit von Programmen bei.

4 Fortgeschrittene Techniken der OOP

4.1 Funktionszeiger und Delegates

Funktionszeiger stellt D in zwei Formen zur Verfügung. Zum Einen in der gewohnten C(++)-Syntax und zum Anderen mit dem neuen Schlüsselwort *function*. Um auf Methoden zeigen zu können, wurden *delegates* eingeführt, die allerdings nicht mit statischen Methoden oder normalen Funktionen zusammenarbeiten. In solchen Fällen muss auf normale Funktionszeiger zurückgegriffen werden.

Beispiel zu Delegates:

```
import std.stdio;

int add(int a, int b) {
    return a+b;
}

class Foo {
    void bar(float f) { writefln("bar ", f); }
}

int main() {
    //Neue D-Syntax
    int function(int, int) fp1 = &add;
    writefln( fp1(2,3) );

    //Alte C-Syntax
    int (*fp2)(int, int) = &add;
    writefln( fp2(8,-2) );

    //Delegate:
    void delegate(float) d;

    Foo f = new Foo;
    d = &f.bar;
    d(3.1416f);

    return 0;
}
```

Der Vorteil von *function* und *delegate* gegenüber den klassischen Funktionszeigern ist die konsistente Syntax und der Verzicht auf sichtbar verwendete Pointer.

4.2 Properties

In den vorangegangenen Abschnitten wurden bereits mehrfach Properties erwähnt, aber noch nicht erklärt. Properties sind im Grunde genommen Eigenschaften von Objekten, aber auch nativen Datentypen. Einige Properties kann man schreiben, andere nur lesen. Bei Strukturen und Klassen kann man Properties verwenden, um den Zugriff auf die Attribute der Klasse zu vereinfachen.

Folgende Properties besitzt jeder Typ:

Properties	Beschreibung
.init	Initialwert
.sizeof	Größe in Bytes
.alignof	Größe die angibt, mit wie vielen Bytes der Typ im Speicher ausgerichtet wird und dem Compiler erlaubt, Strukturen und Klassen exakt zu allokiieren
.mangleof	Darstellung der Compiler-intern benutzten Bezeichnung des Typs

Vgl. D Language Specification, S. 35

Tabelle 11: Properties von allen Typen

Integrale und Fließkommatypen haben noch weitere Properties, die man auf Seite 35ff der D Language Specification findet.

Im Zusammenhang mit Klassen oder Strukturen können Properties getter bzw. setter Methoden ablösen:

```
import std.stdio;

class Foo {
    int rw() { return read_write; }
    void rw(int i) { read_write = i; }

    void w(float f) { write_only = f; }

    char r() { return read_only; }

    private {
        int read_write;
        float write_only;
        char read_only = 'A';
    }
}

int main() {
    //Einige Properties von eingebauten Datentypen
    writeln("Größe von int: ", int.sizeof);
    writeln("NAN von float: ", float.nan);
}
```

```
writefln("Initialwert von byte: ", byte.init);

typedef byte mybyte = 7;
writefln("Initialwert von mybyte: ", mybyte.init);

writefln("Maximalwert von int: ", int.max);

//Properties der Klasse
auto f = new Foo;
f.rw = 245;
writefln("Wert von Foo.read_write: ", f.rw);

f.w = 3.1416;

//Fehler, da write_only nur geschrieben werden kann.
//writefln("Value of Foo.write_only: ", f.w);

//Fehler, da read_only nur gelesen werden kann
//f.r = 'x';
writefln("Wert von Foo.read_only: ", f.r);

return 0;
}
```

Betrachtet man die Properties der Klasse 'Foo' näher, fällt einem auf, dass es eigentlich ganz normale Methoden mit einem Argument bzw. Rückgabewert sind. Das bedeutet, dass `f.rw = 245;` semantisch identisch zu `f.rw(245);` ist und man somit bei einem Aufruf von einer Methoden mit einem Parameter/Rückgabewert nicht unterscheiden kann, ob es als Property oder als normale Methode bestimmt war.

4.3 Operatoren überladen

Operatoren werden in D überladen, indem man eine Methode mit einer speziellen Signatur für eine Struktur oder Klasse schreibt, z.B. `opPostInc` für den `++` - Operator. Der Compiler interpretiert dann einen `++` - Aufruf als Aufruf von `opPostInc`, wobei es auch möglich ist, direkt `opPostInc` als Methode aufzurufen. Einige binäre Operatoren kann man mit zwei Methoden überladen, z.B. den Additionsoperator. Hier kann man `opAdd` und `opAdd_r` überladen. Die `opAdd` Methode wird aufgerufen, wenn unser Objekt der linke Operand ist. Ist es der rechte Operand, wird `opAdd_r` aufgerufen (auf Seite 136 ff der D Language Specification findet man die genauen Aufrufregeln für binäre Operatoren):

```
import std.stdio;

struct Foo {
    int i;

    int opAdd(int i_) {
        return i + i_;
    }
}

struct Bar {
    int i;

    int opAdd_r(Foo f) {
        return i + f.i;
    }
}

int main() {
    Foo f;
    f.i = 3;

    Bar b;
    b.i = 5;

    //Aufruf von Foo.opAdd(int)
    writeln(f+7);

    //Aufruf von Bar.opAdd(Foo)
    writeln(b+f);

    return 0;
}
```

Nicht überladbar sind die Operatoren `&&`, `||`, `..`, `!`, `?:`. In C++ ist z.B. eine Überladung von `&&` möglich, allerdings wird auch dort stärkstens davon abgeraten.⁴

Die Methodensignaturen aller überladbaren Operatoren finden sich in der D Language Specification ab Seite 134ff.

4 Vgl. Meyers, Scott (1999), S.49ff

5 Templates

Neben der OOP ist die Template Metaprogrammierung in letzter Zeit stärker in den Fokus der C++ Entwickler gelangt (hauptsächlich durch die Boost-Bibliothek, die sehr stark auf Metaprogrammierung aufbaut). Das benötigte Sprachmittel, die Templates, bietet auch D an. In ihrer grundlegenden Form funktionieren sie wie die Templates aus C++, nur die Syntax hat sich geändert. Eine Änderung der Syntax war sinnvoll, da die `<>` Syntax aus C++ problematisch ist, wenn man zum Beispiel `std::list<std::pair<int, float>> l;` schreiben möchte. Die abschließenden `>>` werden vom Compiler als Shift-Operator gelesen, was natürlich einen Fehler produziert. In D würde man `std::list!(std::pair!(int, float)) l;` schreiben.

Grundsätzlich kann ein Template Klassen, Strukturen, Typen, Enums, Variablen, Funktionen und andere Templates enthalten.⁵ Die Syntax der Templates muss immer korrekt sein, egal ob sie jemals instantiiert werden. Eine semantische Analyse wird erst bei der Instantiierung durchgeführt.

D-Templates unterstützen die Spezialisierung von Parametern, sowohl auf Typen als auch auf Werte. Es wird dabei immer das am genauesten spezialisierte Template aufgerufen.

Beispiel zu Funktions- und Klassentemplates sowie Spezialisierungen:

```
import std.stdio;

//Einfache Template-Funktion
void swap(T)(inout T a, inout T b) {
    T c = a;
    a = b;
    b = c;
}

//Spezialisierung für Pointer, damit die Werte und nicht
//Adressen getauscht werden
void swap(T : T*)(T *a, T *b) {
    T c = *a;
    *a = *b;
    *b = c;
}

//Generische Pair-Klasse. Der zweite Template-Parameter
//ist per default gleich wie der erste
class Pair(T, U=T) {
    T first;
```

⁵ Vgl. D Language Specification (2007), S. 142

```
    U second;

    this(T first_, U second_) {
        first = first_;
        second = second_;
    }

    ~this() { }
}

int main() {
    int a = 3;
    int b = 5;

    swap!(int)(a, b);          //Aufruf der normalen Template-Funktion

    int *x = &a;
    int *y = &b;

    swap!(int*)(x, y);        //Aufruf der spezialisierten Funktion

    writeln("a: ", a, " b: ", b);    //Ausgabe: a: 3 b: 5

    alias Pair!(float, char) FCPair;

    FCPair p = new FCPair(3.1416f, 'c');
    writeln(p.first, " ", p.second); //Ausgabe: 3.1416 c

    return 0;
}
```

5.1 Rekursive Templates

D unterstützt auch fortgeschrittene Techniken wie Expression-Templates, indem es rekursive Templates als Sprachmittel zur Verfügung stellt. Der Ansatz von Expression-Templates ist, den Compiler zur Kompilierzeit Berechnungen durchführen zu lassen, anstatt sie zur Laufzeit zu berechnen. Dadurch erhöht sich zwar die Kompilierzeit, aber zur Laufzeit muss der Wert nur noch weiterverarbeitet werden.

Im folgenden Beispiel berechnen wir Fibonacci-Zahlen mit Hilfe von Expression Templates:

```
import std.stdio;

template fibonacci(ulong N) {
    enum { fibonacci = fibonacci!(N-1) + fibonacci!(N-2) };
}

//Spezialisierung für den Fall, dass N = 1 ist.
```

```
template fibonacci(ulong N : 1) {
    enum { fibonacci = 1 };
}

//Spezialisierung für den Fall, dass N = 0 ist.
template fibonacci(ulong N : 0) {
    enum { fibonacci = 0 };
}

int main() {
    writefln(fibonacci!(40));

    return 0;
}
```

Es wird der Wert 102334155 ausgegeben.

5.2 *Template Mixins*

Der Begriff Template Mixins bezeichnet in D eine Technik, um den Inhalt einer Template-Deklaration an einem beliebigen Platz einzufügen. Das bedeutet, man kann sich nach dem Baukastenprinzip eine Klasse aus vorhandenen Template-Deklarationen zusammensetzen, ohne Code duplizieren oder über Hilfsfunktionen aufrufen zu müssen:

```
import std.stdio;

//Filtert Namen bei Gleichheit heraus
U filterName(T, U)(T t1, T t2) { return t1 == t2; }

//Filtert alle Minderjaehrigen heraus
U filterAge(T, U)(T t) { return t >= 18; }

//Eine simple Personenklasse
class Person {
public {
    this(char [] name_, uint age_) {
        name = name_;
        age = age_;
    }

    char [] name;
    uint age;
}
}

//Eine spezielle Filterklasse
class Filter {
    mixin filterName!(char [], bool);
    mixin filterAge!(uint, bool);
}
```

```
int main() {
    Person [] sourceArray;
    sourceArray ~= new Person("Mueller", 17);
    sourceArray ~= new Person("Maier", 19);
    sourceArray ~= new Person("Schmidt", 22);
    sourceArray ~= new Person("Meier", 20);

    Person [] targetArray;
    Filter myFilter = new Filter();

    foreach (Person p; sourceArray) {
        //Wir filtern alle minderjaehrigen und Leute mit
        //dem Namen Maier heraus
        if(!myFilter.filterName(p.name,"Maier") &&
            myFilter.filterAge(p.age))
        {
            targetArray ~= p;
        }
    }

    foreach (Person p; targetArray)
        writefln("Name: ", p.name, "\tAge: ", p.age);

    return 0;
}
```

In diesem Beispiel wurde aus vorhandenen Filter-Funktionen ein benutzerdefinierter Filter erstellt, der dann auf ein Array angewendet wird.

6 Boxing

Das Boxing ist bereits aus anderen Sprachen wie Java oder C# bekannt. In Java steht es z.B. für die problemlose Zuweisung von *int* zu der Klasse *Integer* und umgekehrt. Prinzipiell handelt es sich um implizite Umwandlungen, damit man primitive Datentypen (*int*, *float* usw.) als Objekte behandeln kann. Das Konzept des Boxings ist in D jedoch ein ganz anderes, als das in Java verwendete. Es stellt dem Programmierer einen Container (die *Box*) zur Verfügung, in der beliebige Datentypen ohne Angabe des Typs abgelegt werden dürfen. Erst beim Abrufen der Daten aus der *Box* muss der entsprechende Datentyp angegeben werden. Die *Box* stellt damit quasi einen variablen Typ dar.

Ob der Inhalt der *Box* in den gewünschten Datentyp konvertiert werden kann, kann man mit der Methode *Box.unboxable(TypeInfo t)* herausfinden, welche *true* oder *false* zurückgibt.

```
import std.boxer;
import std.stdio;

int main() {
    Box myBox = box(7.7f);
    auto f = unbox!(float)(myBox);
    writefln(f);      //Ausgabe: 7.7

    myBox = box("Hallo Welt");
    auto x = unbox!(char[])(myBox);
    writefln(x);      //Ausgabe: Hallo Welt

    return 0;
}
```

Am Aufruf der *unbox*-Funktion sieht man sehr schön, dass *Boxing* mittels *Templates* realisiert wurde, um die benötigte *Generizität* zu erreichen.

7 Fehlerbehandlung

D bietet mehrere Möglichkeiten an, Fehler zu behandeln. Die erste Möglichkeit wären if-Abfragen, die allerdings nur in gewissem Maße sinnvoll einsetzbar sind. Die nächste Möglichkeit sind Exceptions, die man schon aus anderen objektorientierten Sprachen kennt. Als letzte Möglichkeit bietet D ein neues Sprachmittel an, die so genannten scopes.

7.1 Exceptions

Mit den modernen Sprachmitteln, die D zur Verfügung steht, ergeben sich auch hohe Anforderungen an das Behandeln von Fehlern und Ausnahmefällen, die über die Möglichkeiten einer if-Abfrage hinausgehen. Um diesen Ansprüchen gerecht zu werden, gibt es in D Exceptions. Von der Superklasse *Error* sind zum Einen spezialisierte Klassen wie *ThreadError* und zum Anderen *Exception* abgeleitet. Von *Exception* wiederum sind spezielle Exceptions wie *FileException* oder *UnboxException* abgeleitet worden.

Die Klasse *Error* bietet die Methode *toString()* an, um Fehler in einem lesbaren Format anzuzeigen.

Tritt eine Exception auf, die nicht per *try-catch* abgefangen wird, ruft der Compiler den Standard Fehlerhandler auf, der die Fehlermeldung ausgibt, den Stack abbaut und das Programm dann beendet. Wird die betroffene Stelle allerdings von einer *try-catch* Klausel umgeben, hat man als Programmier noch eine Chance darauf zu reagieren, um zum Beispiel das Programm geordnet zu beenden.

```
import std.stdio;

float div(int a, int b) {
    if (b == 0)
        throw new Exception("Nenner ist null!");
    else
        return cast(float) a / b;
}

int main() {
    try {
        writefln("5/0=", div(5,0));
    } catch (Exception e) {
        writefln("Fehler aufgetreten: ", e.toString());
    }
    return 0;
}
```

7.2 *scope*

Die neue Form der Fehlerbehandlung sind scopes. Sie geben dem Programmierer die Möglichkeit, auf ein bestimmtes Ereignis innerhalb eines Codeblocks zu reagieren. Dieses Ereignis kann entweder ein aufgetretener Fehler bzw. eine Exception (failure), das erfolgreiche Abarbeiten (success) oder das Verlassen des Blocks (exit) sein. Bei der Deklaration eines scopes muss man zum Einen angeben, auf was reagiert werden soll (failure, success oder exit) und wie reagiert werden soll (z.B. Abbruch, Ausgabe einer Meldung oder eine Korrektur der Daten). Sofern exit angegeben ist, wird dies immer zuerst aufgerufen und erst dann success oder failure.

Um die Anwendung von scopes zu demonstrieren, werden wir ein ähnliches Beispiel wie im Abschnitt über Boxing verwenden:

```
import std.boxer;
import std.stdio;

int main() {

    Box myBox;

    { //Erster Codeblock
        float f = 0.0f;

        //Wir definieren einen scope für den Erfolgsfall,
        //bei erfolgreichem unboxing geben wir den Wert aus.
        scope(success) writefln(f);

        myBox = box(7.7f);
        f = unbox!(float)(myBox);
    }

    { //Zweiter Codeblock

        //Dieser scope behandelt einen möglichen Konvertierungsfehler
        //innerhalb dieses Blocks.
        scope(failure) writefln("Fehler bei der Konvertierung");

        myBox = box("Hallo Welt");

        //Es wird absichtlich falsch konvertiert, um eine
        //UnboxException auszulösen
        auto x = unbox!(int)(myBox);
        writefln(x);
    }

    return 0;
}
```

Während der erste Aufruf von *unbox* noch erfolgreich ist, bricht das Programm erwartungsgemäß beim zweiten Aufruf von *unbox* mit einer Exception und der Ausgabe von Fehler bei der Konvertierung ab.

8 Garbage Collection

Eine sehr weitreichende Neuerung gegenüber C(++) ist Garbage Collection. Man allokiert den benötigten Speicher mittels *new* und der Garbage Collector (GC) gibt ihn dann wieder frei. Bei Klassen ruft der GC zusätzlich noch den Destruktor auf, damit z.B. offene Streams sauber geschlossen werden können.

Durch einen GC fällt das Schreiben von den Destrukturen weg, die nur allokierten Speicher freigeben. Diese Destrukturen muss man auch nicht mehr testen oder debuggen, woraus kleinere und einfacher zu wartende Programme resultieren.

Ein GC macht es viel schwieriger, ein Programm zu schreiben, das aufgrund von Speicherlecks abstürzt, da er sich regelmäßig um das Freigeben des allokierten Speichers kümmert. Des Weiteren wird versucht, den Heap so auszurichten, dass möglichst wenig Paging beim Zugriff darauf nötig ist, was sich positiv auf die Performance auswirkt.

Allerdings bringt ein GC nicht nur Vorteile mit sich, sondern hat auch Nachteile. Ein Nachteil ist, dass man nicht exakt vorhersagen kann, wann und wie lange ein Säuberungslauf des GC dauert. Während bei der expliziten Deallokation der Speicher sofort freigegeben wird und zur Verfügung steht, kann es vorkommen, dass der GC erst ein wenig wartet, um genug Speicher in einem Durchgang freigeben zu können.

Der GC gerät zudem an seine Grenzen, wenn man Zeiger in nicht-Zeiger Variablen speichert, da dies zu undefiniertem Verhalten führt.

Zuletzt muss jedes Programm Zugriff auf den GC haben, das heißt die Implementation des GC muss mit dem Programm ausgeliefert werden.

Bei einem Vergleich von einem C und D Hello World Programm fällt dies jedoch kaum in's Gewicht:

main.c:

```
#include <stdio.h>

int main() {
    puts("Hello World");
    return 0;
}
```

main.d:

```
import std.c.stdio;

int main() {
    puts("Hello World");
    return 0;
}
```

Die Programme wurden mit `gcc -o main main.c` und `dmd main.d`, also ohne jede Optimierungen, kompiliert. Das C-Programm ist 6.5 KB und das D-Programm 261.2 KB groß. Der Unterschied mag groß wirken, aber die Größe des GC ist konstant, während ein großes C-Programm durch die manuelle Speicherverwaltung sehr viel zusätzlichen Code mitschleppen könnte. Bei größeren Programmen dürfte die Differenz daher eher gering ausfallen.

Würde man den GC nicht statisch einbinden, sondern durch eine dynamische Bibliothek realisieren, würde die Differenz weiter schrumpfen.

9 Die Standardbibliothek Phobos

Die Standardbibliothek von D trägt den Namen Phobos und stellt grundlegende Funktionen bereit. Sie bietet neben der kompletten C Standardbibliothek (befindet sich im Paket *std.c*) viele neue Module, die das Programmieren erleichtern und die Notwendigkeit für zusätzliche Bibliotheken von Drittherstellern mindern.

Das Augenmerk beim Design von Phobos liegt auf plattformunabhängigen und möglichst einfachen Schnittstellen, voneinander unabhängigen Klassen sowie Verwendung von Contract-Programming und Exceptions, wo es sinnvoll ist.

Neben den üblichen Modulen wie Ein- und Ausgabe sowie mathematischen Funktionen haben auch Dateioperationen, Sockets, Reguläre Ausdrücke, Funktoren, MD5-Funktionen, Threads und Funktionen für die Kompression (*std.zip* und *std.zlib*) Aufnahme in Phobos gefunden. Eine genaue Beschreibung von jedem der einzelnen Module würde den Inhalt der Arbeit bei weitem sprengen. Aus diesem Grund wird auf die Online-Dokumentation der Standardbibliothek verwiesen, die man hier findet: <http://www.digitalmars.com/d/phobos/phobos.html>

Was Phobos im Vergleich zu C++ noch fehlt sind Containerklassen (Listen, Mengen) und Algorithmen, wie sie die Standard Template Library (STL) in C++ bieten. Es gibt jedoch mehrere (inoffizielle) Projekte, die versuchen, die STL für D umzusetzen. Eins dieser Projekte nennt sich D Template Library (DTL).⁶ Unklar ist, wann und ob die DTL in Phobos aufgenommen wird.

⁶ <http://www.prowiki.org/wiki4d/wiki.cgi?Phobos/DTemplateLibrary>

10 Schlussbemerkung

Die wichtigsten Eigenschaften von D wurden nun besprochen und auf dieser Basis ist es möglich, ein Fazit zu ziehen.

D hat viele positive Eigenschaften. Es hat im Vergleich zu C++ Neuerungen wie Module, Properties, Contract-Programming, Scopes und den GC im Repertoire. Bekannte Sprachmittel wie Arrays oder Templates wurden im Vergleich zu C++ stark aufgewertet.

Die Nachteile liegen hauptsächlich in der Infrastruktur von D: Die Standardbibliothek Phobos ist noch nicht umfangreich genug und es gibt im Moment nur zwei Compiler für D. Es fehlen auch namhafte IDEs, Profiler und Debugger. Des Weiteren ist D noch ganz am Anfang, also gibt es auch kaum Bibliotheken für D von Drittherstellern. Es gibt zwar C-Binärkompatibilität, aber was D bräuchte wäre etwas vergleichbares zur C++ Boost-Bibliothek. Abgesehen davon sind viele Features nur gegenüber C++ neu, C# und Java bieten einige Sprachmittel ebenso an (Properties, Delegates oder Module).

Ob D erfolgreich wird, hängt in erster Linie von der Industrie ab. Wenn diese anfängt, D einzusetzen, werden die Hersteller von IDEs, Bibliotheken usw. nachziehen. Allerdings bin ich skeptisch, ob das wirklich passieren wird, da man mit C eine Sprache für Mikrocontroller und mit C++/Java/C# Sprachen für Anwendungsprogramme hat, die im Moment den Markt beherrschen. Natürlich kann man nicht wissen, wie sich D entwickelt, aber meiner Meinung nach wird es ein Hobbyprojekt bleiben, da es gegenüber den etablierten Sprachen nicht genug Neuigkeiten bietet, die einen Umstieg rechtfertigen würden.

Anhang

Listing zum Abschnitt über Klassen:

```
import std.stdio;

class Mitarbeiter {
    private {
        //Die Attribute
        const uint id;
        char [] vorname, nachname;
        ushort alter;
        char geschlecht; //'m' oder 'w'
    }

    protected {
        //statische Methode, um ids zu generieren
        static uint getNextId() {
            static uint id;
            return id++;
        }
    }

    //Methoden können durch 'final' nicht überschrieben werden
    final void setVorname(char [] vname) { vorname = vname; }
    final void setNachname(char [] nname) { nachname = nname; }
    final void setAlter(uint alter_) { alter = alter_; }
    final void setGeschlecht(char geschlecht_) {
        geschlecht = geschlecht_;
    }

    final uint getId() { return id; }
    final char [] getVorname() { return vorname; }
    final char [] getNachname() { return nachname; }
    final ushort getAlter() { return alter; }
    final char getGeschlecht() { return geschlecht; }

    void printToShell() {
        writef(
            "\n-----\n",
            "Name: ", nachname,
            "\nVorname: ", vorname,
            "\nId: ", id,
            "\nAlter: ", alter,
            "\nGeschlecht: ", geschlecht,
            "\n");
    }

    //Konstruktor
    this(char [] vname, char [] nname, uint alter_, char geschlecht_='m') {
        id = Mitarbeiter.getNextId();
        vorname = vname;
        nachname = nname;
        alter = alter_;
        geschlecht = geschlecht_;
    }
}
```

```
    }

    //Destruktor
    ~this() { }

    //Zusicherungen an die Klasse
    invariant {
        assert(geschlecht == 'w' || geschlecht == 'm');
    }
} //Kein ; mehr nötig

//Abgeleitete Klasse
class Angestellter : Mitarbeiter {
private {
    char [] abteilung;
}

    final void setAbteilung(char [] abteilung_) {
        abteilung = abteilung_;
    }

    final char [] getAbteilung() { return abteilung; }

    //Überschreiben der Methode
    void printToShell() {
        super.printToShell();
        writef("Abteilung: ", abteilung, "\n");
    }

    this( char [] vname, char[] nname, uint alter_,
        char geschlecht_='m', char [] abteilung_"A01")
    {
        //Mitarbeiter initialisieren
        super(vname, nname, alter_, geschlecht_);
        abteilung = abteilung_;
    }
    ~this() {}

    //Unit Tests
    unittest {
        Mitarbeiter m = new Angestellter("Franz", "Meier", 40);
        assert(m.getGeschlecht() == 'm');
        assert(m.getAbteilung() == "A01");
    }
}

int main() {
    auto m = new Angestellter("Hans", "Müller", 33, 'm', "A15");
    m.setNachname("Schmidt");
    m.printToShell();

    return 0;
}
```

Die invariant-Klausel in der Mitarbeiter Klasse soll sicherstellen, dass nur 'm' oder 'w' als Werte für das Attribut `geschlecht` akzeptiert werden. Bei der Übergabe eines anderen Wertes bricht das Programm ab. Mit der `unittest`-Klausel der Klasse `Angestellter` wird getestet, ob sie korrekt arbeitet, indem man einen Beispiel-Angestellten anlegt und überprüft, ob die Werte korrekt in der Klassenhierarchie durchgereicht und gesetzt wurden. Sollte der `unittest` fehlschlagen, wird ebenfalls sofort abgebrochen.

Literaturverzeichnis

1. **Hansen, Manfred (2007)**: Programmiersprache D,
http://www.steinmole.de/d/d_buch.pdf (abgerufen am 01.03.2007)
2. **Meyers, Scott (1999)**: Mehr Effektiv C++ programmieren, München (Addison-Wesley)
3. **o.V. (2007)**: D Language Specification, <http://www.prowiki.org/wiki4d/wiki.cgi?LanguageSpecification> (abgerufen am 05.03.2007)