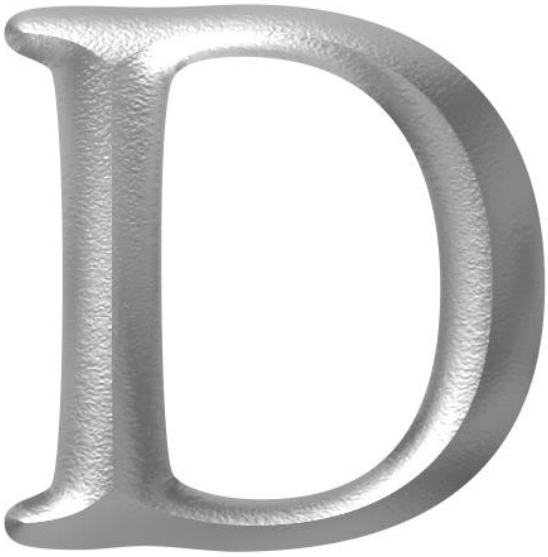
Programmiersprache



Autor:	Manfred Hansen
Mail:	m.hansen@kielnet.net
Erstelldatum:	16. Oktober 2003
Letzte Änderung:	26. September 2007

Inhaltsverzeichnis

1 Einleitung

Willkommen bei der Programmiersprache D. Dieses Buch wendet sich an Programmieranfänger als auch für Fortgeschrittene Programmierer. D ist eine leicht zu erlernende Programmiersprache, die als Nachfolger von C dienen soll, was ja auch schon der Name andeutet. Die Sprache (gdc) ist verfügbar unter Windows, Linux, Mac OS X, FreeBSD, SkyOS und Solaris.Entwickelt wird D von *Walter Bright*.
D Programme werden kompiliert und sind vergleichbar schnell wie c oder c++. Somit eignet sich D sowohl für die Systemprogrammierung als auch für die Anwenderprogrammierung. Wer die Programmiersprache C und ein bisschen Erfahrung aus der Objektorientierten Programmierung mitbringt, wird sich schnell in die neue Sprache einarbeiten können. Die Speicherfreigabe wird von einem `Garbage Collector` übernommen, somit braucht man sich um die Freigabe des Arbeitsspeichers keine Gedanken mehr zu machen. Der `Garbage Collector` kann auch ausgeschaltet werden, was bei Echtzeitanwendungen erforderlich ist, mit `delete` werden dann die Objekte löschen. Des weiteren unterstützt D eine Reihe von Features wie

- Templates
- unittest

Aus folgenden Gründen finde ich die Sprache besonders interessant zu erlernen:

- Keine Speicherreservierung und Speicherfreigabe (GC)
- Dadurch keine Speicherlecks und Buffer Overflow
- Keine Warnings vom Compiler, entweder es ist richtig oder falsch programmiert.
- Zeiger werden nur noch benötigt, um auf C Funktionen zugreifen zu müssen.
- Ausführungsgeschwindigkeit so schnell wie C oder C++.

Zu finden ist D unter [1]. Es gibt eine englischsprachige Newsgroup [3] zu D, die ich nur empfehlen kann. Des weiteren schreibe ich dieses Buch, um mich intensiver mit D auseinander zu setzten und weil es leider noch keine deutschsprachige Dokumentation gibt.
Ich werde in diesem Buch viele Programmbeispiele bringen, weil wie ich finde, das man an Beispielen, am schnellsten, eine neue Programmiersprache erlernt. Außerdem sollen die Beispiele so als Nachschlagewerk dienen.

Anregungen und Verbesserungsvorschläge bitte an m.hansen@kielnet.net mailen.

1.1 Entwicklung der Sprache

Walter Bright der Erfinder und Entwickler der Sprache hatte schon 1988 die Idee eine Nachfolgersprache für C zu entwickeln. 1999 begann er die Sprache D unter Windows zu entwickeln hierfür erstellte er den Compiler `dmd`. Am 10 Mai 2003 wurde die `dmd` Version 0.63 veröffentlicht, die auch die Linux Plattform unterstützte. Des weiteren gab es Bestrebungen ein Frontend für die GNU Compiler Collection `gcc` zu erstellen [12]. Am 02.Januar 2004 gelang es Ben Hinkle mit seinem GCC Frontend ein "Hello World" Programm zu erstellen. Ben Hinkle sein D Frontend liegt jetzt in der Version 0.21 vor, wird von Ihm aber nicht mehr weiter entwickelt. David Friedmann stellte sein D Frontend am 22. März 2004 vor. Dieses Frontend unterstützt mittlerweile alle Eigenschaften die auch der `dmd` von Walter Bright besitzt.

2 Erstes Programm

In diesem Kapitel wird gezeigt wie man die Compiler für D installiert und das erste `Hello World` Programm erstellt. Die Beispielprogramme können unter [2] runter laden.

2.1 Installation von dmd unter Linux

Die Installation ist denkbar einfach, hierzu wird der D Compiler von `http://www.digitalmars.com/d/changelog.html` herunter geladen. Um alle Beispiele im D-Buch kompilieren zu können, benötigen Sie die Version 2.x. Anschließend wird er entpackt und die Dateien `dmd` z.B. nach `/usr/local/bin` und `libphobos.a` nach `/usr/local/lib` kopiert. Sicherheitshalber noch ein `ldconfig` ausführen, damit auch die neuen Libraries gefunden werden.
Als letztes noch die `dmd.conf` nach `/etc` kopieren. Mit einem Editor `vim, kate, nano, usw.` den Pfad der Quellen von `phobos` eintragen.
Zum Beispiel:
`DFLAGS=-I/home/hansen/d/dmd/src/phobos`

2.2 Installation von dmd unter Windows

Den `D` Compiler `dmd` und den `C` Compiler von `ftp://ftp.digitalmars.com/dmd.zip` und `ftp://ftp.digitalmars.com/dmc.zip` herunter laden. Dann z.b. mit WinZip nach `C:\` entpacken. Anschließend kann man mit `C:\dmd\bin\dmd hello.d` in der Eingabeaufforderung das erste Programm kompilieren und linken.

2.3 Installation vom gdc unter Linux

Das **D** Frontend wird von `David Friedmann` entwickelt und steht unter der GPL. Zu finden unter [5]. Bleibt zu hoffen das D im gcc mit aufgenommen wird, und sich die Installation vereinfacht. Dies würde auch die Verbreitung von D sehr hilfreich sein.

Den gdc von [6] herunterladen. Dann folgende Schritte durchführen:

1. tar -xvjf dc-0.24-i686-linux-gcc-4.0.3.tar.bz2
2. cp -r ~/gdc/include/d/ /usr/local/include/
3. cp -r ~/gdc/bin/gdc /usr/local/bin
4. cp -r ~/gdc/bin/gdmd /usr/local/bin
5. cp -r ~/gdc/lib/* /usr/local/lib
6. cp -r ~/gdc/libexec/* /usr/local/libexec
7. cp -r ~/gdc/man/man1/* /usr/share/man/man1

Eine Beschreibung des Programms liegt unter ~/gdc/share/doc. Überprüfen kann man die Installation mit `man gdc`, `gdc -v` und einem kompilieren eines Beispielprogramms z.B: `gdmd hello.d`.

2.3.1 Gdc selber kompilieren

Bevor man die Installation durchführt, sollte man die Installationsleitung die auch unter [5] zu finden ist, durchlesen.

Die Installation vom gdc wird in mehreren Schritten durchgeführt. Am besten sie legen ein neues Verzeichnis z.B. `D` in Ihrem `home` Verzeichnis an. Dann sollte man sich das frontend für gcc von [5] herunterladen und z.B. mit `tar -xvzf gdc-0.25.tar.gz` entpacken. Die `GNU Compiler Collection` von [7] herunterladen. Der `gcc` wird jetzt im Verzeichnis `D` entpackt, mit `tar -xvjf gcc-4.1.2.tar.bz2`, hier wird das Frontend nach `~/D/gcc-4.1.2/gcc` kopiert und entpackt. Nun gibt es ein neues Verzeichnis **d**.
Jetzt geht es ans patchen, hierzu wechseln Sie ins Verzeichnis `~/D/gcc-4.1.2/` und führen `./gcc/d/setup-gcc.sh` aus.
Es erscheint folgende Meldung:

```
patching file Makefile.def
patching file Makefile.in
.
.
.
GDC setup complete.
```

Legen Sie nun ein neues Verzeichnis in `~/D/` an, z.B. `build` und wechseln sie in das neu erstellte Verzeichnis. Jetzt wird der gcc kompiliert, hierzu findet man eine ausführliche Anleitung unter [8]. Hier muss angegeben werden das Sie jetzt die Programmiersprache **D** mit erstellen möchten. Minimalerweise müssen Sie folgendes ausführen:
`../gcc-4.1.2/configure --prefix=/usr/local/gcc-4.1.2 --enable-languages=c,d,c++`

Mit gcc -v können Sie sich auch die Konfiguration des installierten gcc Compilers anschauen und Ihre Konfiguration ergänzen. Nun wird mit `make bootstrap` der gcc kompiliert. Im Verzeichnis `D/build/gcc` sollte sich jetzt der **gdc** befinden. Als User `root` ein `make install` ausführen um den neuen gcc zu installieren. Soweit so gut, jetzt fehlt noch die Bibliothek `libgphobos.a` Hierzu ins Verzeichnis `~/D/gcc-4.1.2/gcc/phobos` wechseln z.B. und mit `./configure --prefix=/usr/local/gcc-4.1.2` konfigurieren dann `make` und `make install` ausführen. Als letztes kann man noch die Umgebungsvariable `PATH` um `/usr/local/gcc-4.1.2/bin/` erweitern, damit der Compiler gdc finden kann. Das war es.
Hier noch mal die einzelnen Schritte

1. mkdir D im Homeverzeichnis
2. Herunterladen des Frontends von [5]
3. Herunterladen des GCC von [7]
4. cp gcc-4.1.2.tar.bz2 /D
5. cd ~/D
6. tar -xjf gcc-4.1.2.tar.bz2

7. cp gdc-0.21.tar.gz gcc-4.1.2/gcc/
8. cd gcc-4.1.2/gcc/
9. tar -xzf gdc-0.21.tar.gz
10. cd ~/D/gcc-4.0.2
11. ./gcc/d/setup-gcc.sh
12. cd ~/D
13. mkdir build
14. cd build
15. ../gcc-4.0.2/configure –enable-languages=c,d,c++ –prefix=/usr/local/gcc-4.0.2 –enable-shared
16. make bootstrap
17. make install
18. export PATH=$PATH:/usr/local/gcc-4.0.2/bin
19. ldconfig

2.4 Hello World

Traditionsgemäß ist das erste Programm was man in einer neuen Programmiersprache erstellt das `Hello World` Programm. Bitte geben Sie mit einem Editor Ihrer Wahl das erste Programm ein. Die Zeilennummern werden nicht wie z.B. unter `GWBASIC` mit eingegeben ☺.

Listing 1: hello

```
1 import std.stdio;
2 int main() {
3     writef("Hello World\n");
4     return 0;
5 }
```

Das Programm wird folgendermaßen übersetzt:
`dmd hello.d` oder mit
`gdc -o hello hello.d`
Jetzt wird ein Programm hello erstellt. Starte es einfach mit `./hello` . Falls beim Übersetzen eine Fehlermeldung der Art kann `Object.d` nicht finden versuchen sie es bitte mit der Option **-I** und direkt dahinter das Verzeichnis in dem die Quellen von phobos liegen, also z.B.:
`gdc -o hello hello.d -I/usr/local/gcc-3.4.3/include/d/`
Zeile 1 importiert das Modul `stdio` hiermit wird die Funktion `writef` dem Programm bekannt gemacht. In Zeile 2 wird die `main` Funktion aufgerufen. Die `main` Funktion darf nur **ein** mal in einem Programm vorkommen. Zu dem int vor der main Funktion und der return Anweisung kommen wir bei den Funktionen noch darauf zurück. Die `main` Funktion ist immer der Einstiegspunkt in dem Programm, von hier aus geht es also los. In Zeile 3 wird `Hello World` auf der Standardausgabe mit der Funktion `writef` ausgegeben. Das `\n` bewirkt ein Zeilenvorschub . Das \ nennt man auch Fluchtoperator oder Escapesequence.

2.5 Compiler

Beim Aufruf des dmd oder gdc Compiler wird das Programm auf die richtige Syntax überprüft, compiliert und gelinkt. Ein Preprocessor lauf wie unter C oder C++ findet nicht statt. Diese Schritte kann man auch einzeln durchlaufen: Die Datei `hello.d` soll erst mal nur kompiliert werden.
dmd -c hello.d
Jetzt befindet sich eine Datei mit der Dateiendung `.o` im Verzeichnis. Das ist die Objektdatei . Diese Objektdatei wird jetzt mit den anderen Dateien zusammen gelinkt.

```
gcc hello.o -o hello -lphobos -lpthread -lm
```

Mit **-o** wird festgelegt wie das Programm heißen soll.
Das Argument -l linkt zusätzliche Bibliotheken zu dem Programm hinzu. Hier haben wir -lphobos ,-lm ,-lpthread. Vor diesen Dateien muss man sich ein `lib` davor und ein `.so` oder `.a` hinten dran denken. Dann bekommen wir die Dateien libphobos.a, libpthread.so libm.so
Wenn man jetzt ldd für das Programm hello aufruft bekommt man ungefähr folgende Ausgabe:

```
hansen@client:~/tex/d$ ldd hello
libpthread.so.0 => /lib/tls/libpthread.so.0 (0x40028000)
libc.so.6 => /lib/tls/libc.so.6 (0x40037000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Das `hello` verwendet 3 gemeinsam genutzte Bibliotheken die `.so` Dateien auch `shared object` Dateien genannt.
Die `libc` Bibliothek enthält die Funktionen der Standard C-Bibliothek. `ld-linux.so.2` wird vom `linker` verwendet. Sie sollten niemals die Datei `libc` umbenennen oder löschen, weil dann so gut wie nichts mehr auf Ihrem System funktioniert. Da kann man dann nur noch versuchen mit einem Rettungssystem die `libc` Datei wieder herzustellen. Zusätzlich ist es möglich mit der **-L** Option weitere Verzeichnisse anzugeben, in dem der Linker nach weiteren Objektdateien sucht.
Man kann sich auch beim `gdc` den Assemblercode ausgeben, den der Compiler erzeugt, dies geschieht mit der Option **-S**.

```
gdc -S hello.d
```

Danach befindet sich eine Datei `hello.s` in deinem Verzeichnis. Beim `dmd` kann man sich den Assemblercode nicht direkt ausgeben lassen, hier kann aber mit dem Programm `obj2asm` der Code erstellt werden.

```
./obj2asm -o hello.obj -c hello.cod
```

Weitere Compiler Optionen für den GDC befinden sich in der Datei `d/lang.opt` . Die Datei wird beim Frontend mitgeliefert..

2.6 gdc

Hier möchte ich die Optionen vom gdc auflisten. Die Liste ist aber nicht vollständig.

- -I : Angabe des Verzeichnes, in dem die Quellen liegen
- -S : Erzeugt den Assemblercode
- -o : Gibt den Namen an, wie die zu Kompilierende Datei heißen soll
- -frelease : Führt weniger Prüfungen aus. Performance Steigerung, zu nutzen bei Programmfertigstellung.

2.7 gdmd

Der `gdmd` ist ein Perl Script, was die selbe Schnittstelle bietet wie der `dmd` Compiler.

- -fall-sources

2.8 strip

Mit strip kann man die ausführbare Datei `hello` noch verkleinern. Bei mir ist z.B. die `hello` Datei 184 kB groß.
`strip hello`

Nach dem strip ist die Datei nur noch 65 kB. Was fast nur noch ein drittel ist. Falls das einem immer noch zu groß ist, kann man mit **upx** die Dateigröße weiter verringern.
`upx -o hello_upx hello -9`
Jetzt ist die Datei nur noch 30 kB groß. Das ist jetzt nur noch ein sechstel von 180 kB, also eine durchaus lohnende Angelegenheit.

2.9 vim

Um Syntax Highlighting für `vim` zu aktivieren ,muss man sich die Datei d.vim von [13] herunterladen und kopiert die Datei nach
`/usr/share/vim/vim62/syntax` .
In der vimrc muss folgendes eingetragen sein:

- syntax on
- filetype plugin on

Des weiteren sollte man noch in die Datei `filetype.vim` im Verzeichnis
`/usr/share/vim/vim62`
die zwei Zeilen eintragen:

```
" D Language
au BufNewFile,BufRead *.d                       setf d
```

Die Pfade zu den `vim` Dateien können auf ihrem System unterschiedlich sein. Ab der `vim` Version 6.3 soll die Syntax Datei schon mit eingebunden sein.

2.10 libphobos.a

Hier möchte ich zeigen wie man das MySQL Module in die libphobos.a unter Linux einbindet, so das man beim kompilieren das mysql.d weglassen kann.
In `~/dmd/src/phobos/etc/c` legt man das Verzeichnis mysql an, anschließend kopiert man dort die Datei mysql.d von [14] in das Verzeichnis. Hier wird ein Object file mit `dmd -c mysql.d` erstellt. Nun muss noch die `linux.mak` Datei in phobos angepasst werden.
Zu der Variable `OBJS` fügen wir jetzt mysql.o hinzu. Jetzt noch folgendes hinzufügen:
`MYSQL_OBJS = etc/c/mysql/mysql.o`

```
libphobos.a : $(OBJS) internal/gc/dmgc.a linux.mak
  ar -r $@ $(OBJS) $(ZLIB_OBJS) $(GC_OBJS) $(RECLS_OBJS) $(MYSQL_OBJS)

### etc/c
mysql.o : etc/c/mysql/mysql.d
$(DMD) -c $(DFLAGS) etc/c/mysql/mysql.d -ofmysql.o
```

Vor dem `$(DMD)` dürfen keine Leerzeichen stehen sondern nur ein `Tabulator` Zeichen. So jetzt geht es ans kompilieren und erstellen der libphobos.a. Eventuell ist es erforderlich den Pfad von dmd in der `linux.mak` Datei anzupassen: Bei mir sieht es folgendermaßen aus `DMD=/usr/local/bin/dmd`. Führen sie `make -f linux.mak` aus bis zur Fehlermeldung:

```
ar: etc/c/recls/recls_api.o: Datei oder Verzeichnis nicht gefunden
make: *** [libphobos.a] Fehler 1
```

Jetzt in das Verzeichnis `etc/c/recls/` gehen und dort ebenfalls `make -f linux.mak` ausführen. Falls erforderlich das selbe für die `zlib` wiederholen.
Anschließend wenn alles geklappt hat befinden sich ein Haufen neuer Dateien im Verzeichnis `~/dmd/src/phobos/`. Jetzt kann die libphobos.a z.b. nach `/usr/local/lib/` kopiert werden. Mit `ar -t libphobos.a` kann man sich noch mal vergewissern ob die Datei mysql.o in der neuen Bibliothek vorhanden ist. Spaßeshalber kann man jetzt auch mal das neu erstellte Programm `unittest` ausführen.

In Ihrem mysql Programm muss die mysql.d Datei noch importiert werden, das sieht jetzt folgendermaßen aus:

```
import etc.c.mysql.mysql;
```

2.11 Windows Mobile

In diesem Abschnitt geht es darum zu zeigen, wie man unter Linux D Programme kompiliert und unter `Windows Mobile` ausführen kann. Ziel soll es sein ein `Hallo D` auf der MSDOS Kommandozeile aus zu geben.

Abbildung 1: console

2.11.1 Cross-compiling

`Cross-Compiling` bedeutet, das man auf einem Betriebssystem für ein anderes Betriebssystem und/oder andere Hardware ein Programm übersetzt. In unserem Fall werden wir auf einem x86 Prozessor unter dem Betriebssystem Linux ein Programm Compilieren für ein `ARM` Prozessor für das Betriebssystem `Windows Mobile`. Hierzu müssen wir erst mal den Compiler von [16] oder [17] herunterladen. Diesen entpacken und dort befindet sich ein Verzeichnis `arm-wince-pe`. Dieses inklusiv der Unterverzeichnisse nach `usr/local` kopieren. Die Umgebungsvariable `PATH` erweitern, z.B. so:
`PATH=$PATH:/usr/local/arm-wince-pe/bin`
Eventuell ist es noch nötig in `usr/local` folgenden Befehl auszuführen:
`chmod -R a=rwx arm-wince-pe`
Hiermit werden für alle Lese.- Schreib.- und Ausführungsrechte erteilt.

2.11.2 PocketConsole

Jetzt soll der MSDOS Eingabeprompt auf dem PDA installiert werden. Bei `WM5` ist es notwendig die `HKEY_LOCAL_MACHINE\Drivers\Console\OutputTo` auf `0` zu setzen. Als erstes sollten Sie auf Ihrem Windows Rechner ActiveSync installieren. Den kann man unter [20] herunter laden. PDA mit dem Windows Rechner verbinden. Als nächstes auf [18] gehen und `PocketConsole (ARM)` auswählen und dann auf dem Windows Rechner installieren, dann werden automatisch die Programme auf dem PDA übertragen. Beim PDA erscheint das Folgende Meldung:

Hier auf `ok` gehen und den PDA neu booten.

Jetzt muss noch `PocketCMD` installiert werden, das liegt unter [19]. Jetzt `PocketCMD (ARM)`

auswählen und wie bei der Installation von `PocketConsole` vorgehen. Anschließend sollte man unter `Programme` auf dem PDA `FreeDos` installiert sein.

2.11.3 Erstes Programm

Hierzu folgendes Programm unter Linux erstellen:

Listing 2: helloarm

```
1 void main()
2 {
3     printf("Hallo D\n");
4 }
```

Dieses Programm folgendermaßen kompilieren:
`arm-wince-pe-gdc helloarm2.d -o helloarm2.exe`
Weil die exe Datei recht groß ist kann man Sie mit:
`arm-wince-pe-strip helloarm2.exe`
noch verkleinern. Wem das noch nicht langt, kann mit `upx` noch mal die Größe der Datei verringern. Als letztes kopiert man das Programm `helloarm2.exe` auf dem PDA, z.B. mit der `SD Card` und führt das Programm wie in Abbildung 1 aus.

Stellt sich noch die Frage, warum `printf` gewählt wurde, statt `writefln`? Das liegt daran, das eine Variable Argumentenliste noch nicht unterstützt wird. Lesen Sie bitte hierzu die `Readme` Datei die beim Compiler mitgeliefert wird.

Teil I.
Einführung

3 Grundlagen

3.1 Kommentare

Kommentare sind ein wichtiges Hilfsmittel um den Quellcode zu dokumentieren. Sie sollen beschreiben was gemacht wird und nicht das wie.

Listing 3: Kommentare

```
1  int main() {
2      // Wird eine Zeile auskommentiert
3      /*  Zeile 1
4          Zeile 2 wird auskommentiert
5      */
6
7      /+
8      // printf("Hallo\n");
9      +/
10     return 0;
11 }
```

Zeile 2: Mit `//` wird eine Zeile auskommentiert.
Zeile 3 u. 5: Die `/*` und `*/` sorgen dafür das ein ganzer Bereich auskommentiert wird.
Zeile 7 u. 8: Verschachtelte Kommentare werden mit `/+` und `+/` eingeleitet.

3.2 Steuerzeichen

Die Steuerzeichen müssen in Doppelten Anführungszeichen gesetzt werden.

Steuerzeichen	Bedeutung
\a	Akustischen Signal
\b	BS (backspace) - setzt Cursor um eine Position nach links
\f	Formfeed Seitenvorschub
\n	Zeilenvorschub
\r	carrige Return Curser springt zum Anfang der Zeile
\t	Tabulatorzeichen
\"	Doppelte Anführungszeichen werden gequotet
\u1234	wchar Zeichen
\U00101234	dchar Zeichen
\v	VT (vertical tab) - Cursor springt zur nächsten vertikalen Tabulatorposition
\0101	Oktale Schreibweise
\x41	Hexadezimale Schreibweise
\0	Null Zeichen, markiert das Ende eines Strings
\’	Zeichen ’ wird ausgegeben
\"	Zeichen " wird ausgegeben
\\	Zeicen wird ausgegeben
\?	Zeichen ? wird ausgegeben

Tabelle 1: Steuerzeichen

3.3 Datentypen

Für unterschiedliche Aufgaben gibt es verschiedene Datentypen. Wenn man weiss das nur positive Zahlen benötigt werden, kann man vorzeichenlose (unsigned) Datentypen verwenden, somit kann man den passenden Datentyp wählen um Arbeitsspeicher zu sparen. Ausserdem unterscheiden Sie sich von der grösse des Speicherbedarf der benötigt wird. Es gibt Datentypen für Fliesskommazahlen Complexe Zahlen und mehrere Datentypen für Zeichen. Die Tabelle auf Seite 120 listet alle möglichen Datentypen auf.

3.3.1 char

Der Datentyp char dient zur Anzeige eines einzelnen Zeichens wie 'a'. Das char Feld wird in UTF-8 codiert und hiermit kann auch der ganze ASCII [9] Zeichensatz ausgedruckt werden.

Listing 4: char

```
1  import std.stdio;
2  int main() {
3      char zeichen = 'A';
4      // Ein Zeichen wird zugewiesen
5      printf("zeichen = %c\n",zeichen);
6      char zeichen1 = '\x41';
7      // Ein Zeichen in Hexadezimal zugewiesen
8      printf("zeichen1 = %c\n",zeichen1);
9      char zeichen2 = 0x41;
10     // Ein Zeichen in Hexadezimal zugewiesen
11     printf("zeichen2 = %c\n",zeichen2);
12     char zeichen3 = '\101';
13     // Ein Zeichen in Oktal zugewiesen
14     printf("zeichen = %c\n",zeichen3);
15     char zeichen4 = 0101;
16     // Ein Zeichen in Oktal zugewiesen
17     printf("zeichen4 = %c\n",zeichen4);
18     printf("zeichen = %d\n",cast(int)zeichen3);
19     // Ein Zeichen in Oktal zugewiesen
20     printf("max = %d \t min = %d\t\n",cast(int)zeichen3.max,cast(int)
           zeichen3.min);
21
22     char a = '1';
23     char b = '2';
24     printf("%c\n",a+b);
25     printf("%d\n",cast(uint)a+cast(uint)b);
26     return 0;
27 }
```

Ausgabe des Programms:

```
zeichen = A
zeichen1 = A
zeichen2 = A
```

```
zeichen = A
zeichen4 = A
zeichen = 65
max = 255        min = 0
c
99
```

In Zeile 24 werden die Charakter Zeichen addiert und als Ergebnis kommt `c` heraus. Wenn man in die ASCII Tabelle [9] schaut sieht man das 1 die Dezimalziffer 49 und die 2 die Dezimalziffer 50 hat. Die Summer ergibt dann 99 und das ergibt das Zeichen `c`.

3.3.2 wchar

wchar ist 16 bit gross und wird in UTF-16 codiert, respektive UCS-2. [10] Unter Linux ist wchar 32 bit gross, Analog zu `wchar_t` in C. Die 2 steht für 2 Byte. Soll der ASCII Code angezeigt werden, muss vor dem ASCII Wert 2 Nullen geschrieben werden. Z.b.:
`wchar sonderzeichen= '\u0041'; ergibt das Copyright Zeichen`
In der ASCII Tabelle [9] steht 41hex für das Zeichen `A`.

3.3.3 dchar

dchar beseht aus 32 bit und wird in UTF-32 codiert, bzw UCS 4. Hier werden 4 Byte für ein Zeichen reserviert. Hier verhält es sich ähnlich, als wenn man ASCII Zeichen ausdrucken will, nur das 6 Nullen von dem ASCII Wert geschrieben werden müssen. Zum Beispiel `dchar sonderzeichen= '\U000000A9';` ergibt das Copyright Zeichen. Nun geht die ASCII Tabelle aber nicht bis A9, also woher kommt das Zeichen? Das A9 bezieht sich hier auf dem ANSI Zeichensatz [9].
Der Quellcode kann in UTF-8 erstellt werden.
Hierzu muss aber der Client auf UTF-8 umgestellt sein. Vielleicht noch ein kleiner Tipp, wenn Sie ihren Client auf UTF-8 umstellen, und wollen sich auf ein Rechner z.B. per SSH verbinden, der kein UTF-8 unterstützt, kann man folgenden Aufruf ausführen:

```
luit -encoding 'ISO 8859-15' ssh servername
```

```
1  import std.string;
2  import std.stdio;
3
4  int main() {
5
6          bit b;
7          printf("bit\tmin: %d\tmax: %d %u\n", b.min, b.max,b.sizeof);
8
9          byte by;
10         printf("byte\tmin: %d\tmax: %d %u\n", by.min, by.max,by.sizeof);
11
12         ubyte bu;
```

```
13          printf("ubyte\tmin: %d\tmax: %d %u\n", bu.min, bu.max,bu.sizeof);
14
15          short sh;
16          printf("Speicher= %d Min = %d Max = %d\n", sh.sizeof, sh.min, sh.max);
17
18          ushort ush;
19          printf("Speicher= %d Min = %d Max = %d\n", ush.sizeof, ush.min, ush.max);
20
21          int isn;
22          printf("int: Speicher= %d Min = %ld Max = %ld\n", isn.sizeof, isn.min, isn.max);
23
24          uint uisn;
25          printf("uint: Speicher= %d Min = %u Max = %u\n", uisn.sizeof, uisn.min, uisn.max);
26
27          long lo;
28          printf("long: Speicher= %d Min = %lli Max = %lli\n", lo.sizeof, lo.min, lo.max);
29
30          ulong ulo;
31          printf("ulong: Speicher= %d Min = %lld Max = %llu\n", ulo.sizeof, ulo.min, ulo.max);
32
33          float fl;
34          printf("float: Speicher= %d Min = %lg Max = %lg\n", fl.sizeof, fl.min, fl.max);
35
36          double dl;
37          printf("double: Speicher= %d Min = %lg Max = %lg \n", dl.sizeof, dl.min, dl.max );
38
39          real r;
40          printf("real: Speicher= %d Min = %llg Max = %llg \n", r.sizeof, r.min, r.max);
41
42          ireal ir;
43          printf("ireal: Speicher= %d Min = %Lg Max = %Lg \n", ir.sizeof, ir.min, ir.max);
44
45          ifloat ifl;
46          printf("ifloat: Speicher= %d Min = %Lg Max = %Lg \n", ifl.sizeof, ifl.min, ifl.max);
47
48          idouble id;
49          printf("idouble: Speicher= %d Min = %lg Max = %lg \n", id.sizeof, id.min, id.max);
50
51          dchar d_sonderzeichen = '\U000000A2'; // Cent Zeichen
52          writefln("d_sonderzeichen = %s",d_sonderzeichen) ;
53          wchar w_sonderzeichen= '\u20ac'; // Euro Zeichen
54          writefln("w_sonderzeichen = %s",w_sonderzeichen) ;
55          wchar sonderzeichen = 'ü';
56          writefln("sonderzeichen = %s",sonderzeichen) ;
57          writefln("Laenge von dchar %d",dchar.sizeof);
58          writefln("Laenge von wchar %d\n",wchar.sizeof);
59          string  öß = "Besondere Variable";
60          writefln("öß = %s\n",öß) ;
61
62          return 0;
63  }
```

Ausgabe des Programms:

bit min: 0 max: 1 1

byte min: -128 max: 127 1
ubyte min: 0 max: 255 1
Speicher= 2 Min = -32768 Max = 32767
Speicher= 2 Min = 0 Max = 65535
int: Speicher= 4 Min = -2147483648 Max = 2147483647
uint: Speicher= 4 Min = 0 Max = 4294967295
long: Speicher= 8 Min = -9223372036854775808 Max = 9223372036854775807
ulong: Speicher= 8 Min = 0 Max = 18446744073709551615
float: Speicher= 4 Min = 1.17549e-38 Max = 3.40282e+38
double: Speicher= 8 Min = 2.22507e-308 Max = 1.79769e+308
real: Speicher= 10 Min = 3.3621e-4932 Max = 1.18973e+4932
ireal: Speicher= 10 Min = 3.3621e-4932 Max = 1.18973e+4932
ifloat: Speicher= 4 Min = 1.47256e-4932 Max = 1.07857e-4930
idouble: Speicher= 8 Min = 2.22507e-308 Max = 1.79769e+308
d_sonderzeichen = ¢
w_sonderzeichen = €
sonderzeichen = ü
Laenge von dchar 4
Laenge von wchar 2

öß = Besondere Variable

In Zeile 59 wird noch mal deutlich, das der Quellcode in UTF-8 erstellt werden kann.

3.3.4 Formatspezifizierer

Mit dem Formatspezifizierer wird das Format angegeben wie ein Zeichen, String oder Zahl formatiert werden soll. Am einfachsten erlernt man die Bedeutung der Formatspezifizierer an der `printf` Funktion, weil man gleich die Ausgabe auf dem Monitor (Standardout) betrachten kann. Wichtig sind Sie aber auch noch bei der sprintf Funktion. In `D` sollte man die `printf` Funktion nicht mehr nutzen, als ich 2003 angefangen habe das Buch zu schreiben, gab es noch keine `writef` Funktion.

Spezifiezierer	Bedeutung	Typen
`%c`	Ein einzelnes Charakter Zeichen	char
`%d`	Für decimal Zeichen	int, uint,short,uschort,bit,byte
`%e`	Expotentialschreibweise	long,ulong
`%f %F`	Fliesskommazahlen	float,ifloat,ireal,double,idouble,real
`%g %G`	Expotinatialschreibweise	long,ulong
`%lld`	Grosse Dezimahlzahlen	long
`%llu`	Grosse Vorzeichenlose Dezimalzahlen	ulong
`llx`	Denn Sinn noch nicht verstanden	long ulong
`%p`	Gibt die Adresse aus	gilt für alle
`%s`	String der null `'\0'` terminiert ist.	char arrays
`%.*s`	String der nicht null terminiert ist.	char arrays
`%u`	Vorzeichenlose Dezimahlzahlen	uint,ushort,bit
`%x %X`	Hexadezimale schreibweise	char
`%`	Prozentzeichen wir ausgegeben	

Tabelle 2: Formatspezifizierer

Hier ein paar Beispielprogramm zu den Formatspezifizieren

Listing 5: Formatspezifizierer

```
1  import std.stdio;  // wird fuer writefln() benoetigt
2  int main() {
3      float f = 23.456;
4      int i = 5;
5      string s;
6
7      printf("f = %.2f\n",f);
8      writefln("f = %1.2f\n",f);
9      printf("%03d\n",1); // wird mit 0 aufgefuellt
10     writefln("(%-10s)","Hallo"); // Links, statt rechtsbuendig
11     writefln("(%10s)","Hallo");
12     writef("tab \t"); // Tabulator Zeichen
13     writefln("tab1");
14     s = std.string.format("i in float = %.4f",cast(float) i);
15     writefln(s);
16
17     return 0;
18 }
```

Die Ausgabe des Programms:

```
f = 23.46
f = 23.46

001
(Hallo     )
(     Hallo)
tab     tab1
i in float = 5.0000
```

In Zeile 14 wird mit der `cast` Funktion der Integer Wert *i* zum *float* Datentyp umgewandelt. Mit *%.4f* wird die float Zahl mit 4 Nachkommastellen ausgegeben.

3.3.5 union und struct

Union wird der Vereinigungstyp genannt. In union kann immer nur ein Wert zur Zeit abgespeichert werden, weil immer die gleiche Speicherstelle benutzt wird. Der Vorteil von union gegenüber struct ist, das hier Speichplatz gespart wird. Der Speicherplatz für die Variablen wird im Struct hintereinander abglegt, somit summiert sich der Speicherplatzbedarf mit der Anzahl der Variablen.

Listing 6: Union

```
1  private import std.stdio;
2
3  int main() {
4
5      union U {
6          int a;
7          double b;
8          string  c;
9      }
10     static U u = { c : "Hallo" };
11     writefln("u.c = %s",u.c);
12     writefln("bytes = %d",U.sizeof);
13     return 0;
14 }
```

Ausgabe des Programms:

```
u.c = Hallo
bytes = 8
```

In Zeile 5 wird der neue Union Type U deklariert. Die Zeilen 6,7,8 sind die möglichen Variablen die der Typ zur Verfügung stellt. Die Variable c vom Union wird in Zeile 10 zugewiesen und letztendlich in Zeile 11 ausgegeben. Wichtig ist hier das die Variable `u.c` über die Punktnotation ausgegeben wird. Die Anzahl der Bytes die vom Union belegt werden wird in Zeile 12 ausgegeben. Hier sind es 8 Bytes, weil double 8 Bytes Speicherplatz belegt.

Listing 7: Struct

```
1  private import std.stdio;
2
3  int main() {
4
5      struct U {
6          int a;
7          double b;
8          string c;
9      }
10     static U u = { a : 5, b:5.0 };
11     static U v = { c : "Hallo" };
12     writefln("u.a = %d",u.a);
13     writefln("u.b = %f",u.b);
14     writefln("u.c = %s",v.c);
15     writefln("bytes = %d",U.sizeof);
16
17
18     return 0;
19 }
```

Ausgabe des Programms:

```
u.a = 5
u.b = 5.000000
u.c = Hallo
bytes = 24
%8
```

In diesem Listing wurde in Zeile 5 `union` gegen `struct` getauscht. Hier können nun mehrere Varaiblen gleichzeitig initialisiert werden wie man in Zeile 10 sieht. In Zeile 15 wird der Speicherbedarf des Struct ausgebeben.

Listing 8: Struct

```
1  import std.stdio;
2
3  struct Person {
4      string anrede;
5      int alter;
6      int hausnummer;
7      void write() {
8          writefln("Cool, eine Funktion innerhalb vom struct");
9      }
10 }
11
12 int main(string[] args) {
13     Person person;
14     person.anrede="Herr";
15     person.alter=54;
16     person.hausnummer=23;
17     writefln("Anrede: %s",person.anrede);
18     writefln("Alter: %s",person.alter);
19     writefln("Hausnummer: %s",person.hausnummer);
20     person.write();
21     return 0;
22 }
```

Ausgabe:
Anrede: Herr
Alter: 54
Hausnummer: 23

Ein struct ist eigentlich ein Relikt aus der C Programmierung, weil in C nur Zahlen, aber keine Zeichen an eine Funktion übergeben werden können, als Zahl wird in C eine Adresse übergeben. Struct dienen aber auch der Übersichtlichkeit, man sieht in dem Beispiel das alle Variablen innerhalb des struct zur Person gehören. Ein struct wird mit dem Schlüsselwort `struct` eingeleitet. In Zeile 3 wird das struct initialisiert, anschliessend kann man in die Variablen innerhalb des Structs Werte zu weisen wie in Zeile 14 bis 16. Mit der Punktnotation

können die Variablen wie in Zeile 17 bis 19 ausgelesen werden. In D ist es auch möglich innerhalb von struct Funktionen zu deklarieren, sowie es in Zeile 7 gezeigt wird. Diese write Funktion wird dann in Zeile 20 ausgeführt.
Man kann auch Struct und Union zusammen verwenden.

Listing 9: Struct mit Union

```
1  private import std.stdio;
2
3  int main() {
4
5      struct S {
6          int a;
7          double b;
8          string c;
9          union U {
10             int d;
11         };
12     }
13     static S u = { a : 5, b:5.0 };
14     static S v = { c : "Hallo" };
15     static S.U w = { d : 4};
16     writefln("u.a = %d",u.a);
17     writefln("u.b = %f",u.b);
18     writefln("u.c = %s",v.c);
19     writefln("w.d = %d",w.d);
20     writefln("bytes = %d",S.sizeof);
21
22
23     return 0;
24 }
```

Ausgabe des Programms:

```
u.a = 5
u.b = 5.000000
u.c = Hallo
w.d = 4
bytes = 24
```

Das eigentlich interessante ist hier Zeile 15, wie mit `S.U` die Variable `d` vom `Union` initialiesiert wird.

3.4 Konstanten

Konstanten werden mit dem Schlüsselwort `const` erstellt.

Listing 10: Konstanten

```
1 private import std.stdio;
2 int main() {
3     const int i = 3;
4     //i = 5;
5     const char[] fahrzeug = "Auto";
6     writefln("i = %d",i);
7     writefln("fahrzeug = %s",fahrzeug);
8     return 0;
9 }
```

Konstanten können während der Laufzeit nicht mehr verändert werden. Würde man die Zeile 4 einkommentieren bekommt man eine Fehlermeldung.

3.5 Variablen

Variablen sind Platzhalter die Arbeitspeicher reservieren. Hier unterscheidet man das initialisieren und das deklarieren von Variablen. Bei der Deklaration wird der Arbeitsspeicher für die Variable reserviert und mit einem Standardwert belegt. Für Integervariablen ist er `0` und für Floatvariablen ist er `nan`. `nan` steht für `not a number` .

Listing 11: Deklaration u. Initialisierung

```
1 import std.c.stdio;
2
3 int main() {
4     int a; //deklaration (a noch undefiniert)
5     a = 3; // Variable a wird initialiesiert
6     int b = 3; // dekleration und Initialisierung
7
8     return 0;
9 }
```

In Zeile 4 wird eine Variable deklariert und Speicher wird reserviert. Da es sich hier um den Datentyp 4 Integer handelt, werden 4 Byte im Arbeitsspeicher reserviert. Zeile 4 wird der Variablen `a` der Wert 3 zugewiesen, das geschieht mit dem = Operator , das nennt man auch initialisieren. Bei der Zeile 6 wird die Varable `b` gleichzeitig deklariert und initialisiert.

Listing 12: Auf NaN prüfen

```
1  import std.stdio;
2  import std.math;
3
4  void main() {
5      float f;
6
7      if (isnan(f)) {
8          writefln("f ist nicht definiert");
9          writefln("f: ", f);
10     } else {
11         writefln("f ist definiert");
12         writefln("f: ", f);
13     }
14 }
```

Ausgabe:
f ist nicht definiert
f: nan

Die Variable `f` wird in Zeile 5 deklariert aber nicht initialisiert. Der Standardwert ist nun `nan`. Deshalb wird in Zeile 8 `f ist nicht definiert` ausgegeben.
Um überhaupt mit der Funktion `isnan` zu prüfen, muss das Modul `std.math` eingebunden werden.

Bei Variablen, muss das erste Zeichen, ein Buchstabe sein, sonst können Sie auch aus Ziffern bestehen. Der Unterstrich _ gilt auch als Buchstabe. Es wird zwischen Gross und Kleinschreibung unterschieden. Schlüsselwörter als Variablen sind nicht zulässig. Die Variablennamen sollte man so wählen, das man schon am Namen sehen kann welchen Zweck die Variable hat. Bei Schleifenzähler nimmt man meistens die Variablen `i,j`

3.5.1 Gültigkeitsbereich von Variablen

Es gibt verschiedene Attribute die den Gültigkeitsbereich von Variablen angeben oder deren art und weise wie die Variable verwendet wird festlegen:

- auto
- static

`volatile` gilt nicht für Datentypen wie in **C**.

3.5.2 auto

Sobald vor einer lokalen Variable nichts steht ist sie automatisch **auto**. Wird innerhalb einer Funktion eine Variable definiert so ist Sie lokal. Auf diese Variable kann ausserhalb der Funktion nicht zugegriffen werden. Es kann auch ausserhalb der Funktion eine neue Variable

die den gleichen Namen hat deklariert werden, ohne das sich die Variablen im Speicherbereich in die quere kommen. Es ist wichtiger Besandteil der strukterierten Programmierung, das die Funktionen in sich abgeschlossen sind. Aus diesem Grund sollte man immer versuchen Variablen lokal zu definieren. Man stelle sich vor es werden externe Funktionen eingebunden und man weiss gar nicht welche Variablennamen schon vergeben sind, das würde also nicht gehen. Beim verlassen der Funktion oder des Anweisungsblockes werden die lokalen Variablen wieder zerstört, bzw. der Garbage Collector gibt den Arbeitsspeicher wieder frei.

3.5.3 static

Listing 13: static und auto

```
1  import std.c.stdlib;
2
3  void add() {
4      static int x = 0; // Statische Variable
5      auto int y = 0;   // auto kann auch weggelassen werden
6      x++;
7      y++;
8      printf("x = %d\n",x);
9      printf("y = %d\n",y);
10 }
11
12 int main() {
13     add();
14     add();
15     return 0;
16 }
```

Die Ausgabe des Programms:
x = 1
y = 1
x = 2
y = 1

Hier wird eine Funktion `add()` in Zeile 13 und 14 aufgerufen. In der Funktion wird in Zeile 5 die Variable x mit dem `static` attribut deklariert und initialisiert und in Zeile 6 wird eine Variable y ebenfalls deklariert und initialisiert. Man sieht an der Ausgabe das beim zweiten Aufruf der `add()` Funktion der x Wert auf 2 gesetzt wird, wohin gegen der y Wert immer noch bei 1 ist. Das liegt daran, das `static` Variablen nur beim ersten Aufruf **einmal** initialisiert werden, und nach Beendigung der Funktion ihren Wert in der Variable behält. Die Variable, mit dem Attribut `auto`, oder auch ohne dem Attribut auto, (Standardmässig sind alle Variablen auto) wird bei jedem Funktionsaufruf neu initialisiert.

Innerhalb der `main()` Funktion macht es keinen Unterschied ob man die Variablen statisch deklariert. Wenn man die main Funktion beendet wird das Programm beendet und es können keine Variablen zwischengespeichert werden.

3.5.4 extern

Das Schlüsselwort extern gilt nicht um die Speicherungsart der Variablen festzulegen. Das hat sich in **D** gegenüber **C** vereinfacht. In **C** war es üblich globale Variablen, die also ausserhalb der Funktion deklariert wurden, in externen Programmen sichtbar zu machen. Hier ein Beispiel wie es jetzt in D ist:

Listing 14: Variablenübergabe in Modulen mod_haupt.d

```
1  import std.c.stdio;
2  import mod1;
3
4  int g = 3; // globale Variable
5  int main( char[][] arg ) {
6      int f = 5;
7      funk();
8      printf("f mit Modulnamen = %d\n",mod1.f);   //mit Modulnamen
9      printf("f = %d\n",f);           // lokale Variable
10
11     return 0;
12 }
```

Listing 15: Variablen übergabe in Modulen mod1.d

```
1  import mod_haupt;
2
3  int f = 3;  // globale Variable
4  void funk() {
5      printf("g = %d\n",g);
6  }
```

Ausgabe des Programms:
g = 3
f mit Modulnamen = 3
f = 5

Übersetzt wird das Programm mit:
`dmd mod_haupt.d mod1.d`

`g` und `f` sind beides globale Variablen, da sie ausserhalb der Funktion deklariert wurden. Jedes externe Programm hat seinen eigenen Namensbereich (namespace) . Um nun auf die Variablen der Module zugreifen zu können müssen sie importiert werden wie in Zeile 1 und Zeile 2 der beiden Programme. In Zeile 6 des Programms `mod_haupt.d` wurde die Variable `f` noch mal lokal deklariert und initialisiert, deshalb ist f auch hier 5. In Zeile 8 wird die Variable `f` mit dem vorangestellten Modulnamen `mod1` ausgegeben. Der Wert ist hier 3, weil im Programm `mod1.d`, die Variable mit 3 initialisiert wurde.

3.6 Operatoren

Es gibt drei verschiedene Arten von Opertatoren:

- unär
- binär
- tenär

Der `unäre` Operator hat nur ein Argument wie z.B. der `++` und der `--` Operator. Binäre Operatoren haben 2 Operanden wie z.B. `+ - / *`. Es gibt nur ein `tenären` Operator, dem `?:`, der 3 Operanden besitzt. Desweiteren besitzen die Operatoren eine Rangfolge in der die Operatoren ausgeführt werden. Jeder kennt wohl den Satz aus der Schule `Punktrechnung kommt vor Strichrechnung`, genau das beschreibt die Rangfolge der Operatoren . Ausserdem wird die Reihenfolge unterschieden in dem ein Ausdruck ausgewertet wird. Hier kann von `links nach rechts` und von `rechts nach links` ausgewertet werden. Ausserdem besitzen unäre Operatoren mehr Vorrang als binäre Operatoren. Die Auswertung von `rechts nach links` bedeutet beim Zuwesungsoperator `=` das der rechte Teil vom `=` Zeichen ausgewertet wird und links dem Bezeichner also der Variablen zugewiesen wird.

Listing 16: Rangfolge von Operatoren

```
1
2  int main() {
3      int a;
4      int b = 3;
5      int c[4];
6      int i = 2;
7
8      a = b++ +7;
9      printf("%d\n",a);
10     printf("%d\n",b);
11
12     return 0;
13 }
```

Bei den `++` und `--` Operatoren, unterscheidet man nach dem Präfixinkrement bzw Präfixdekrement und den Postfixinkrement bzw. Postfixdekrement. Beim Präfix wird der Operator vor dem Bezeicher und beim Postfix hinter dem Bezeichner geschrieben. In Zeile 8 hat der Bezeichner *b* ein Postfixinkrement. Das Ergebnis wird in Zeile 9 und 10 Ausgeben und ist *10* und *4*. Hier wird also *b + 7 (b = 3)* addiert das Erbebnis *10* wird der Variablen *a* zugewiesen. Anschliessend wird *b* um 1 inkrementiert, das ergibt 4. Hätte die Variable *b* ein Präfixoperator *++b* wäre das Ergebnis *11*, weil vor der Auswertung des Ausdrucks b inkrementiert wird. Solche Art der Programmierung wo die Rangfolge nicht so einfach ersichtlich ist, ist kein guter Programmierstil.

Priorität	Operator	Bedeutung	Operand	Assoziativität	Seite
1	()	Funktionsaufrufe	2	links nach rechts	
1	[]	Arrayindex	2	links nach rechts	
1	.	Element Zugriff von Union, Struct und Objektattribut	2	links nach rechts	
1	++	Erhöhung nach Auswertung Postinkrement	1	links nach rechts	28
1	–	Erniedrigung nach Auswertung	1	links nach rechts	
2	++	Erhöhung vor Auswertung Präinkrement	1	rechts nach links	
2	–	Erniedrigung nach Auswertung Prädekrement	1	rechts nach links	
2	!	Logische Negation	1	rechts nach links	
2	+ -	Vorzeichen	1	rechts nach links	
2	~	Komplement	1	rechts nach links	30
3	new	Instanz erzeugen	1	rechts nach links	
3	cast	Typ Umwandlung	1	rechts nach links	
4	* / %	Multiplikation, Division Modulo	2	links nach rechts	57
5	+ -	Addition u. Subtraktion	2	links nach rechts	28
5		String Verkettung	2	links nach rechts	
6	<<	Schift Links	2	links nach rechts	31
6	>>	Schift Rechts	2	links nach rechts	31
6	>>>	Rechtsshift unsigned	2	links nach rechts	31
7	< <=	kleiner, kleiner gleich	2	links nach rechts	
7	> >=	größer, größer gleich	2	links nach rechts	
8	==	Gleichheit	2	links nach rechts	
8	!=	Ungleichheit	2	links nach rechts	
8	===	Referenz Vergleich	2	links nach rechts	
9	&	Logisches und Bitweises UND	2	links nach rechts	30
10	^	Logisches und Bitweises XOR	2	links nach rechts	30
11	—	Bitweises ODER	2	links nach rechts	30
12	&&	Logisches UND	2	links nach rechts	
13	——	Logisches ODER	2	links nach rechts	
14	?:	bedingte Auswertung (tenär)	3	rechts nach links	
15	+= -= %= ^= \= *= &= |= <<= >>= >>>= ~= ~=	arithmetische Zuweisungen arithmetische Zuweisungen kombinierte Zuweisungen Bitweise Zuweisungen Bitweise Zuweisungen Bitweise Zuweisung String Zuweisungen	2	rechts nach links	 31 31 43

Tabelle 3: Operatoren

3.7 Bitoperatoren

Listing 17: Komplement

```
1  int main() {
2      ubyte i = 254;
3
4      //128 64 32 16 8 4 2 1
5      //   1  1  1  1 1 1 1 0 wird beim Einer-Komplement zu
6      // 00000001
7
8      printf("i = %d\n",i);
9      printf("i = %d\n",~i); //Komplement
10
11     return 0;
12 }
```

Listing 18: Komplement

```
1  int main() {
2      ubyte  a = 23,b = 7,c; // einzelnes bit
3
4      c = a & b;
5      printf("& = %d\n",c); // UND
6
7      c = a | b;
8      printf("| = %d\n",c); // ODER
9
10     c = a ^ b;
11     printf("^ = %d\n",c); // Exklusiv Oder XOR
12
13     return 0;
14
15 }
```

Zeile 4 ergibt das Ergebnis 7. Hier wird eine `UND` Operation ausgeführt.

```
64 32 16 8 4 2 1
---------------
 0  0  1 0 1 1 1  = 23
 0  0  0 0 1 1 1  = 7
===============
 0  0  0 0 1 1 1  = 7
```

Nur wenn beide Bits eins sind, kommt es bei einer `UND` Verknüpfung zu 1.
Die `ODER` Verküpfung in Zeile 7 ergibt 23.

```
 0 0 1 0 1 1 1  = 23
 0 0 0 0 1 1 1  = 7
==============
 0 0 1 0 1 1 1  = 23
```

Wenn ein Bit eins ist kommt es bei einer `ODER` Verknüpfung zu 1.
Als letztes haben wir noch die Exklusiv oder `XOR` in Zeile 10, das ergibnis ist 16.

```
0 0 1 0 1 1 1  = 23
0 0 0 0 1 1 1  = 7
==============
0 0 1 0 0 0 0  = 16
```

Nur wenn einer 0 und ein Bit 1 ist kommt es bei der `XOR` Verknüpfung zu einer 1.

Die Shift Operatoren << und >> verschieben die Bits um eine stelle und füllen sie mit Nullen auf.
00000011 >> 00000001 shift rechts
00000011 << 00000110 shift links

Listing 19: Komplement

```
1  int main() {
2      ubyte a = 12;  // 00001100 (12)
3      ubyte b;
4      int c = -16;
5      int d;
6
7      b = a >> 1;
8      printf("b = %d\n",b); // 00000110 (6)
9
10     b = a << 1;
11     printf("b = %d\n",b); // 00011000 (24)
12
13     a <<= 1;
14     printf("a = %d\n",a); // 00011000 (24)
15
16     a >>= 1;
17     printf("a = %d\n",a); // 00001100 (12)
18
19     a >>>=1;
20     printf("a = %d\n",a); // 00000110 (6)
21
22     c >>>=2;
23     printf("c = %d\n",c); // 1073741820
24
25     return 0;
26
27 }
```

Der Operator >>> behandelt alle Zahlen nicht vorzeichenbehaftet.
In Zeile 22 wird -16 um 2 nach rechts verschoben. Um die Zahl -16, in Bit Darstellung zu überführen wird das Komplement gebildet und 1 dazu addiert.

00000000 00000000 00000000 00010000 = 16
11111111 11111111 11111111 11101111 = Komplement
00000000 00000000 00000000 00000001 = +1
==========
11111111 11111111 11111111 11110000 = -16
^

Vorzeichen Bit
Jetzt 2 Shift nach rechts und links wird mit **0** aufgefüllt.
00111111 11111111 11111111 11111100 = 1073741820

3.8 Schleifen

Schleifen dienen dazu um einen Anweisungsblock mehrmals hintereinander zu durchlaufen.

3.8.1 for-Schleife

Die for Schleife ist eine abweisende Schleife, weil vor dem Durchlauf des Schleifenkörpers die Bedingung geprüft wird und gegebenenfalls gar nicht durchlaufen wird. Sie wird also abgewiesen. Die Anzahl der Durchläufe des Schleifenkörpers ist feststehend.

Listing 20: for-Schleife

```
1 import std.stdio;
2
3 void main() {
4     for(int i = 0; i < 3; i++) {
5         writefln("Zahl i = %d",i);
6     }
7 }
```

Ausgabe des Programms:

```
Zahl i = 0
Zahl i = 1
Zahl i = 2
```

In Zeile 4 wird die Variable `i` deklariert und mit dem Startwert `0` initialisiert. Mit `i < 3` wird die Bedingung geprüft.Mit i++ wird die Variable um 1 hochgezählt man nennt das inkrementieren. . Zeile 5 ist der Schleifenkörper, hier wird also die i ausgegeben. Man hätte das Inkrementieren von `i` auch im Schleifenkörper machen können, wie folgendes Beispiel zeigt:

Listing 21: for-Schleife

```
1 import std.stdio;
2
3 void main() {
4     for(int i = 0; i < 3;) {
5         writefln("Zahl i = %d",i);
6         i++;
7     }
8 }
```

3.8.2 while Schleife

Die while Schleife ist auch eine Abweisende Schleife mit einer Variablen Anzahl von durchläufen.

Listing 22: while Schleife

```
1  import std.stdio;
2
3  void main() {
4      int i = 13;
5
6      while (i > 10) {
7          i--;
8          writefln("Zahl i = %d",i);
9      }
10 }
```

Ausgabe des Programms:

```
Zahl i = 12
Zahl i = 11
Zahl i = 10
```

In Zeile 4 wird die Variable `i` mit `4` initialisiert. Zeile 6 beginnt die while Schleife und endet in Zeile 9. Vor Eintritt in den Schleifenkörper wird die Bedingung ob `1 > 10` geprüft. Am Anfang ist `i=15`, also wird die Schleife durchlaufen. Mit `i--` wir `i` um eins herunter gezählt, man nennt das auch decrementieren.

3.8.3 do-while Schleife

Die `do-while` Schleife ist eine Annehmende Schleife. Hier wird auf jeden Fall mindestens einmal der Schleifenkörper durchlaufen und erst am Ende der Schleife die Bedingung geprüft. Sie wird deshalb auch `Fuß Schleife` genannt.

Listing 23: do-while Schleife

```
1  import std.stdio;
2
3  void main() {
4      int i = 0;
5
6      do {
7          i++;
8          writefln("Zahl i = %d",i);
9      } while (i > 3);
10 }
```

Ausgabe des Programms:

```
Zahl i = 1
```

Die Schleife beginnt in Zeile 6 mit dem `Ausdruck`. Die Variable `i` wird mit 0 initialisiert und in Zeile 7 um 1 inkrementiert. Zeile 8 gibt `i = 1` aus und mit Zeile 9 wird geprüft ob `i` größer ist als 3, das ist nicht der Fall und die Schleife wird nicht noch mal durchlaufen.

3.8.4 foreach Schleife

Die `foreach` Schleife ist wieder eine Abweisende Schleife, prüft also vor Eintritt in den Schleifenkörper die Bedingung.

Listing 24: foreach Schleife

```
1  import std.stdio;
2
3  void main() {
4
5      string str = "Hallo";
6
7      foreach (char c; str) {
8          writefln("Zeichen %s",c);
9      }
10 }
```

Ausgabe des Programms:

```
Zeichen H
Zeichen a
Zeichen l
Zeichen l
Zeichen o
```

Die Variable `str` ist ein Array von einzelnen Zeichen die wir mit der `foreach Schleife` (für jedes) durchlaufen, und jedes Zeichen einzeln Ausgeben. Wie man sieht braucht man sich nicht mal mehr Gedanken zu machen wie viele Zeichen das Wort `Hallo` hat. Die Schleife wird einfach solange ausgeführt bis das letzte Zeichen ausgegeben wurde.

Listing 25: foreach Schleife

```
1  import std.stdio;
2
3  void main() {
4
5      foreach (char c; "Hallo") {
6          writefln("Zeichen %s",c);
7      }
8  }
```

Man kann auch den String `Hello` direkt in die foreach Schleife setzen, so wie in Zeile 5.

3.8.5 break

Listing 26: Schleifenabbruch mit break

```
1  import std.stdio;    //writefln
2  import std.math;     // PI
3  import std.conv;     // toInt
4  import std.cstream; // readLine
5
6  void main() {
7      int r;
8      char[] x;
9      while(1) {
10         writef("Bitte geben Sie den Radius ein: ");
11         r = toInt(din.readLine());
12         if (r == 0) {
13             break;
14         }
15         writefln("Die Kreisflaeche betraegt %f",PI * r * r);
16     }
17 }
```

`while(1)` in Zeile 9 bedeutet, die Bedingung ist immer wahr, die while Schleife wird also nicht beendet. Erst wenn eine 0 als Radius eingegeben wird , wird in Zeile 12 verglichen ob `r` gleich `0` ist, wenn das der Fall ist, wird mit `break` der Schleifenkörper verlassen und führt gegebenenfalls weitere Anweisungen aus, die nach der while schleife kommen würden. In Zeile 10 wird der Satz `Bitte geben Sie den Radius ein` ausgegeben. Mit `toInt` wird der eingegebene Wert in ein Integer Wert umgewandelt. Wenn man statt eine Zahl ein Buchstabe eingibt, kommt es zu einer Fehlermeldung und das Programm wird abgebrochen. Diesen Fehler sollte man in einem „richtigen“Programm abfangen. Das `din` von Zeile 11 findet man im Modul `cstream` wieder. Dort steht, das `din, dout, derr` die Standard IO Geräte sind.

3.9 Verzweigungen

Es gibt die einfachen `if` Verzweigungen und die mehrfachen Verzweigungen mit `switch`.Ausserdem kann mit dem tenären Operator auch abhängig von der Bedingung verzweigt werden.

3.9.1 if Anweisung

Hier ist ein Beispielprogramm für eine `if`-Anweisung.

Listing 27: if-else Anweisung

```
1  import std.stdio;
2
3  int main( char[][] arg ) {
4      if ( arg.length < 2 ) {
5          writefln("Es wurde kein Buchstabe eingegeben!" );
6          return -1;
7      }
8
9      if ( arg[1] == "a" ) {
10         writefln("Buchstabe a");
11     } else if (arg[1] == "b")  {
12         writefln("Buchstabe b");
13     } else  {
14         writefln("Weder Buchstabe a oder b");
15     }
16     return 0;
17 }
```

In Zeile 4 haben wir die erste `if` Anweisung. Hier wird geprüft ob `arg.length` kleiner 2 ist. Wenn dem so ist, wird die Fehlermeldung `Es wurde kein Buchstabe eingegeben` ausgegeben. Anschliessend wird das Hauptproggamm (main Funktion) auf Grund der `return` Anweisung mit `-1` **sofort** beendet. Bei erfolgreicher Ausführung des Programms wird eine `0` an das aufzurufende Progamm zurückgegegen. Falls ein Buchstabe eingegeben wurde wird in Zeile 9 geprüft ob es ein `a` ist. Das `<` und `==` sind Vergleichsoperatoren. Wenn das erste Argument ein `a` ist, dann ist der Ausdruck innerhalb der Klammer `true`. Das bedeutet die Bedingung wird geprüft und liefert ein `boolean` Wert zurück, entweder `true` oder `false`. Innerhalb der Klammer kann aber auch eine Zahl zurückgegeben werden. Die `0` entspricht `false` und alle anderen Zahlen entsprechen `true`. Folgende Ausdrücke sind also möglich:

```
if (0) { } else {writefln("false");}
if (true) {writefln("true");}
```

In Zeile 11 wird geprüft ob der Buchstabe `b` eingegeben wurde ist das der Fall wird der Text `Buchstabe b` ausgegeben. Falls es weder der Buchstabe `a` noch `b` ist wird die `else` Anweisung in Zeile 13 abgearbeitet. Man kann also das `if else` konstrukt beliebig oft wiederholen. Für diesen Fall ist aber besser die `switch` Anweisung zu benutzen.

3.9.2 switch Anweisung

Listing 28: switch Anweisung

```
1  import std.stdio;
2
3  int main( char[][] arg ) {
4      if ( arg.length < 2 ) {
5          writefln("Es wurde nichts eingegeben!" );
6          return -1;
7      }
8
9      switch( arg[1] ) {
10         case "this":
11             writefln("Dies");
12             break;
13         case "is":
14             writefln("ist");
15             break;
16         case "a":
17             writefln("ein");
18             break;
19         case "pen":
20             writefln("Fueller");
21             break;
22         default :
23             writefln("Irgend ein String");
24             break;
25     }
26     return 0;
27 }
```

In Zeile 9 beginnt die `switch` Anweisung und prüft was für ein Wert eingegeben wurde. Wurde `is` eingegeben wird mit der `case` Anweisung in Zeile 13 fortgefahren. Hier wird dann `ist` ausgegeben. Anschliessend wird die `switch` Anweisung mit der `break` Anweisung verlassen und das Programm wird nach der `switch` Anweisung fortgesetzt. Trifft nun kein Wert für die `switch` Bedingung zu, wird die `default` Anweisung in Zeile 22 ausgeführt. Wenn man also z.B. `toll` eingibt, wird also `Irgend ein String` ausgegeben.

3.9.3 tenärer Operator

Listing 29: tenärer Operator

```
1 private import std.stdio;
2
3 void main() {
4     int i=4,j;
5
6     j=( i < 5 ) ? i+2 : i+4;
7     writefln("j = %d",j);
8 }
```

3.10 Arrays

Es gibt dynamische, statische und assoziative Arrays. In einem Array können mehrere Werte vom gleichen Typ gespeichert werden.
`int[2] a`

Das Array hat jetzt 2 Elemente in dem 2 Integer Werte gespeichert werde können. Die Anzahl der Elemente ist nur vom Hauptspeicher begrenzt.

3.10.1 statische Arrays

Die Arrays können auf verschiedene Weise deklariert werden. Es gibt einmal die Prefix Array Deklaration und die Postfix Array Deklaration. Die Prefix Array Deklaration erfolgt **vor** dem Identifier (auch Bezeichner genannt) und wird von links nach rechts gelesen.
`int[3] a;`
Hier ist *a* der Identifier.

Bei der Postfix Array Deklaration, erfolgt die Deklaration **nach** dem Identifier und wird von rechts nach links gelesen.

Die Prefix Deklaration ist vorzuziehen, da die Postfix Deklaration nur für Umsteiger von C/C++ gedacht ist. Folgendes geht nun nicht mehr: `int a[3],b;` dies muss nun in zwei Zeilen geschrieben werden. Statische Arrays können in ihrer Anzahl der Elemente nicht mehr verändert werden. Zur Zeit ist es so, das vor dem Array das Schlüsselwort static geschrieben werden muss, wenn es initialisiert wird, dies ist aber ein Bug und wird hoffentlich irgendwann beseitigt.
Das `static` hat also nichts mit statischen Arrays zu tun. Siehe hierzu:

`http://www.wikiservice.at/wiki4d/wiki.cgi?PendingPeeves`
`http://www.digitalmars.com/drn-bin/wwwnews?D/26695`

Listing 30: Array deklaration

```
1  import std.stdio;
2
3  int main() {
4
5      static char[5][2] string = ["house","car"];
6      writefln("string[0] = %s",string[0]);
7      writefln("string[1] = %s\n",string[1]);
8
9      static int[3] a = [0:2,4,5];  // statische Array (geht noch nicht)
10     writefln("a[0] = %d",a[0]);
11     writefln("a[1] = %d",a[1]);
12     writefln("a[2] = %d",a[2]);
13     writefln("a laenge = %d\n",a.length);
14
15     enum color {red,green,blue};
16     static int[3] b = [color.red:3,7,9];
17     writefln("b[0] = %d",b[0]);
18     writefln("b[1] = %d",b[1]);
19     writefln("b[2] = %d",b[2]);
20
21     int[2] iarray;
22     iarray[0] = 3;
23     iarray[1] = 4;
24     writefln("iarray = %d",iarray[0]);
25     writefln("iarray = %d",iarray[1]);
26
27     return 0;
28 }
```

```
static char[5][2] string = ["house","car"];
```

Die Variable string in Zeile 5 ist ein 2 Dimensionales Array vom Datentyp char. In der ersten Dimension wird die Anzahl der einzelnen Zeichen, hier 5 und in der zweiten Dimension die Anzahl der Elemente, hier 2 gespeichert. `house` hat genau 5 Zeichen. string[0] ist dann house und string[1] = car

```
static int[3] a = [0:2, 4,5];
```

Hier bekommt die Variable a[0] den Wert 2, a[1] = 4 und a[2] = 5
Das selbe gilt auch für strings:

Mit `enum` wird ein neuer Datentyp deklariert. `color.red` hat den Wert **3** und wird in Zeile 16 zugewiesen.
`int[2] iarray;`
Dieses Array kann jetzt zwei Intergerwerte aufnehmen. Hier wird also nicht bei 0 begonnen zu zählen, sondern erst bei 1. Bei der Zuweisung des Arrays wird aber wieder bei 0 angefangen.
`iarray[0] = 3;`

```
iarray[1] = 4;
```

3.10.2 3-dimensionales statisches Array

Listing 31: 3 dimensionales Array

```
1  import std.stdio;
2  int main() {
3          //static int  h[2][3][4] = [[[1,2,3,4],
4      //                              [5,6,7,8],
5      //                              [9,10,11,12]
6      //                               ],
7      //                             [ [13,14,15,16],
8      //                               [17,18,19,20],
9      //                   [21,22,23,24]] ];
10         static int[4][3][2] h = [ [[1,2,3,4],
11                                    [5,6,7,8],
12                       [9,10,11,12]
13                      ],
14                                      [[13,14,15,16],
15                       [17,18,19,20],
16                       [21,22,23,24]
17                      ]
18                     ];
19     writefln("h = %d",h[0].length);
20         for (int i=0; i <h.length; i++) {
21                 for (int j=0; j <h[0].length; j++) {
22                         for (int k=0; k <h[0][0].length; k++) {
23                                 writefln("h = %d",h[i][j][k]);
24                         }
25                 }
26         }
27
28         return 0;
29 }
```

In Zeile 3 oder in Zeile 10 wird ein 3 dimensionales Array initialisiert.
Über die `for` (Zeile 20 bis 26) Schleifen wird dann jedes einzelne Element vom Array ausgegeben. In diesem Array konnen 2 * 3 * 4 = 24 Elemente gespeichert werden. Man sieht hier die Struktur das 4 Elemente in ein Array, dann 3 Arrays mit jeweils 4 Elementen und wiederum diese 2 Arrays in der Variable h gespeichert wird. Mit `h.length` wird die Anzahl der Elemente des Array festgestellt. In diesem Beispiel ist es 2. Für `h[0].length` ist es 3.

3.10.3 dynamische Arrays

Dynamische Arrays enthalten die Länge und ein pointer zum Garbage Collector. Die Strings brauchen nun nicht mehr mit `\0` terminiert werden, da ja nun die Länge der Strings bekannt ist. Siehe hierzu auch Kapitel 11.

Desweiteren sind dynamische Arrays Zeiger auf die eigentlichen Array Daten.

Listing 32: Array deklaration

```
1  import std.c.stdio;
2
3  int main() {
4
5      string[] string = ["house","car"];
6      string[0] ~="boot";
7      string  ~= "bike";
8      printf("string[0] = %.*s\n",string[0]);
9      printf("string[1] = %.*s\n\n",string[1]);
10     printf("string[2] = %.*s\n",string[2]);
11     int[] a;
12     a ~= 1;
13     printf("a[0] = %d\n\n",a[0]);
14
15
16     //static int[] b = [0:2, 4,5];
17     static int[] b = [1,2,6];  // dynamisches Array
18     printf("b[0] = %d\n",b[0]);
19     printf("b[1] = %d\n",b[1]);
20     printf("b[2] = %d\n",b[2]);
21     printf("b laenge = %d\n\n",b.length);
22
23     int[] c;
24     c.length=4;
25     printf("c laenge = %d\n",c.length);
26
27     return 0;
28 }
```

In Zeile 5 wird der Variable `string` die Werte `house` und `car` zugewiesen. In Zeile 6 wird `boot` mit dem Operator ~ (tilde) wird `house` und `boot` zusammen geführt oder konkateniert. Zeile 7 weisst jetzt der Variablen `string` ein zusätzliches Element hinzu. Hier wird noch mal deutlich, das man ohne vorher festgelegt zu haben wie viele Elemente die Variable `string` hat,einfach ein zusätzliches Element also dynamisch hinzu fügen kann. Zeile 12 ist im Prinzip das selbe wie in Zeile 7 nur das der Datentyp `int` ist. In Zeile 21 wird noch mal die Anzahl der Elemente ausgegeben, dies geschieht mit der Funktion `length`. Doch mit `length` kann man bei dynamischen Arrays die Anzahl der Elemente festlegen, dies geschieht in Zeile 24. Das geht mit statischen Arrays natürlich nicht.

Da das neue vergrössern oder realozieren des Arrays mit dem ~ Operator eine ziemlich aufwendige operation ist, kann man die Anzahl der Elemente mit `length` vorher festlegen.

Listing 33: Array deklaration

```
1  import std.c.stdio;
2
3  int main() {
4
5      string[] string;
6      string.length=2;
7      string[0] = "house";
8      string[1] = "car";
9      string[0] ~="boot";
10     string ~= "bike";
11     printf("string[0] = %.*s\n",string[0]);
12     printf("string[1] = %.*s\n",string[1]);
13     printf("string[2] = %.*s\n",string[2]);
14
15     return 0;
16 }
```

Man kann jetzt die einzelnen Werte einfach mit dem = Operator zuweisen. Dies geht aber nur bei den ersten beiden Elementen, beim dritten Element in Zeile 10 geht das nicht mehr, weil ja nur 2 Elemente mit `length` festgelegt wurden.

3.10.4 3-dimensionales dynamisches Array

Listing 34: 3 dimensionales Array

```
1  int main() {
2
3      static int[][][] h = [ [[1,2,3,4],
4                              [5,6,7,8],
5                              [9,10,11,12]],
6                              [[13,14,15,16],
7                              [17,18,19,20],
8                              [21,22,23,24]]
9                              ];
10
11     foreach(int[][] i;h[][][]) {
12         foreach(int[] j;i[][]) {
13             foreach(int k;j[]) {
14                 printf("h = %d\n",k);
15             }
16         }
17     }
18
19     return 0;
20 }
```

3.10.5 Matrix

Listing 35: Dynamische Matrix

```
1  int main() {
2      int[][] matrix;
3      int z=0,rechteck=5;
4
5      matrix.length=rechteck;
6      for (int i=0; i < rechteck; i++){
7          matrix[i].length=rechteck;
8          for (int j=0; j < rechteck; j++){
9              if (i == j) {
10                 matrix[i][j] = 1;
11             } else {
12                 matrix[i][j] = 0;
13             }
14
15         }
16     }
17     matrix[3-1][2-1] = 9;  // Zeile 3 u. Spalte 2 wird eine 9 gesetzt.
18     //Ausgabe
19     foreach(int[] j;matrix[][]) {
20         foreach(int i;j[]) {
21             if (z == rechteck -1 ) {
22                 printf("%d \n",i);
23                 z = 0;
24             } else {
25                 printf("%d ",i);
26                 z++;
27             }
28         }
29     }
30     return 0;
31 }
```

Ausgabe des Programms:

```
1 0 0 0 0
0 1 0 0 0
0 9 1 0 0
0 0 0 1 0
0 0 0 0 1
```

3.10.6 assoziative Arrays

Bei assoziativen Arrays werden die Werte zusammen mit einmem eindeutigem `key` gespeichert und auch über den key wieder gelöscht oder gesucht. In anderen Programmiersprachen werden Sie auch `hash` genannt.

Listing 36: Assoziative Arrays

```
1  import std.string;
2
3  int main() {
4      string[string] hash;
5      hash["en"] = "Hello, world!";
6      hash["es"] = "Hola, mundo!";
7      hash["de"] = "Hallo Welt!";
8
9      foreach ( string key; hash.keys ){
10         if (hash[key] == "Hello, world!") {
11             printf("%.*s\n",hash[key]);
12         }
13     }
14     return 0;
15 }
```

In Zeile 4 wird das assoziative Array mit dem Identifier `hash` deklariert. Die Elemente die gespeichert werden können sind vom Typ char und der key ist ebenfalls vom Typ char. Bei den Zeilen 5,6 und 7 wird das assoziative Array mit Werten initialiesiert. Die keys sind hier `en,es` und `de`. Über die `foreach` Schleife in Zeile 9 werden die einzelnen keys durchlaufen und in Zeile 10, wird mit der `if` Anweisung überprüft, ob es den Wert `Hello, world!` gibt. Wenn ja wird er nochmal in Zeile 11 `Hello, world!` ausgegeben.

Listing 37: Assoziative Array Funktionen

```
1  int main() {
2
3      int[string] hash;
4      hash["en"] = 5;
5      hash["es"] = 3;
6      hash["de"] = 7;
7      hash["pl"] = 1;
8
9      foreach ( string key; hash.keys.sort ){
10         printf("%d\n",hash[key]);
11     }
12
13     hash.remove("en");
14
15     if ( "en" in hash ) {
16         printf("Ja\n");
17     } else {
18         printf("Nein\n");
19     }
20
21     return 0;
22 }
```

In Zeile 3 wir ein assoziatives Array deklariert, wobei jetzt die Elemente vom Datentyp int und der key vom Datentyp char ist. Die keys werden in Zeile 9 mit dem Befehl `sort` alphabetisch sortiert. Mit `remove` in Zeile 13 wird der key aus dem assoziativen Array gelöscht. Hier wird nur der key und nicht der Wert aus dem Array gelöscht. Das Schlüsselwort `in` in Zeile 15 prüft ob es den key `en` noch gibt und liefert den Datentyp boolean als Ergebnis zurück.

3.10.7 rechteckige Arrays

Mehrdimensionale `rechteckige` Arrays kommen aus der numerischen Programmierung, wie zum Beispiel von der Programiersprache `Fortran`. Die Syntax ist die selbe wie beim erstellen von Mehrdimensionalen Arrays. Dynamische rechteckige Arrays gibt es in D nicht. Das Casting von dynamischen zu statischen Arrays ist leider noch nicht möglich.

Listing 38: Array deklaration

```
1  import std.c.stdio;
2
3  int main() {
4
5      string[] string;
6      string.length=2;
7      string[0] = "house";
8      string[1] = "car";
9      string[0] ~="boot";
10     string ~= "bike";
11     printf("string[0] = %.*s\n",string[0]);
12     printf("string[1] = %.*s\n",string[1]);
13     printf("string[2] = %.*s\n",string[2]);
14
15     return 0;
16 }
```

3.11 Funktionen

3.11.1 Einfache Funktion

Listing 39: Einfache Funktion

```
1  import std.stdio;
2
3  void p(string x) {
4      writefln(x);
5  }
6
7  int main (string[] args){
8      string n = "Text in Variable n";
9      n.p();
10     p("Hallo");
11     return 0;
12 }
```

Hier heisst die Funktion (Zeile 3) `p` und es wird ein `string` übergeben. In Zeile 8 wird der Variablen `n` ein Text zugewiesen. Bei Zeile 9 wird die Funktion `p` aufgerufen und der Wert von `n` wird ausgegeben. Diese Art des Aufrufs ist eigentlich nicht üblich, scheint aber zu gehen. Das ganze sieht eher aus, als ob eine Methode aufgerufen wird. Normalerweise wird eine Funktion aufgerufen wie in Zeile 10. Der Parameter ist hier der String `Hallo`.

3.11.2 Variable Argumentenliste

Bei einer Variablen Argumentenliste können beliebig viele Argumente an die Funktion übergeben werden, ohne vorher zu wissen wie viele Argumente übergeben werden sollen.

Listing 40: Variable Argumentenliste

```
1  import std.c.stdarg;
2
3  int foo(string x, ...) {
4
5      va_list ap;
6      va_start!(typeof(x))(ap, x);
7      printf("&x = %p, ap = %p\n", &x, ap);
8      printf("%.*s\n",x);
9
10     int i;
11     i = va_arg!(typeof(i))(ap);
12     printf("i = %d\n", i);
13
14     long l;
15     l = va_arg!(typeof(l))(ap);
16     printf("l = %lld\n", l);
17
18     uint k;
19     k = va_arg!(typeof(k))(ap);
20     printf("k = %u\n", k);
21
22     va_end(ap);
23
24     return i + l + k;
25 }
26
27 int main() {
28     int j;
29
30     //j = foo("hello", 3, 23L, 4);
31     j = foo("hello", 3);
32     printf("j = %d\n", j);
33     //assert( j == 30);
34
35     return 0;
36 }
```

In Zeile 3 wird die Funktion mit beliebig vielen Argumenten aufgerufen, das wird deutlich an den 3 Punkten **...** .
Zeile 5 liefert einen Zeiger auf **ap** zurück der von den Funktionen (Templates) `va_list,va_start` und `va_arg` benötigt wird. Mit `va_start` wird ap die Adresse des ersten optionalen Argument, hier `x` initialisiert. Mit `va_arg` wird das erste Argument aus der Liste von `ap` übernommen. Die Variable **ap** wird in Zeile 22 mit `va_end` wieder zurück gesetzt. Nach dem Quelltext in stdarg.d (phobos) passiert bei dieser Funktion noch nichts.

Ausgabe des Programms:

```
&x = 0xbffff7f8, ap = 0xbffff800
hello
i = 3
l = 23
k = 4
j = 30
```

3.11.3 Funktion mit inout

Listing 41: Funktion mit inout

```
1  void foo(inout int x) {
2      printf("x = %d\n",x);
3      x +=5;
4  }
5
6  int main () {
7      int a = 2;
8      foo(a);
9      printf("a = %d\n",a);
10     return 0;
11 }
```

Die Ausgabe des Programms ist:
x = 2
a = 7

Mit dem Parameter `inout` in Zeile 1 wird das Argument x entgegengenommen und auch wieder zurückgegeben. Deshalb ist a = 7. Wenn man den Parameter `inout` wegglässt, also nicht vor dem `int` wie in diesem Beispiel, wird der Funktionsparameter per default auf `in` gesetzt.

3.11.4 Funktion mit out

Listing 42: Funktion mit out

```
1  void foo(out int x) {
2      printf("x = %d\n",x);
3      x +=5;
4  }
5
6  int main () {
7      int a = 2;
8      foo(a);
9      printf("a = %d\n",a);
10     return 0;
11 }
```

Die Ausgabe des Programms:

x = 0
a = 5
In Zeile 8 wird die Funktion `foo` mit dem Parameter a = 2 aufgerufen. Dann wird in Zeile 2 das x ausgegeben. Hier wäre der Wert Normalerweise 2, aber aufgrund des `out` Parameters in Zeile 1 wird der Wert nicht entgegengenommen und x bleibt deshalb 2. Anschliessend wird zum `x`, 5 addiert und in Zeile 9 ausgegeben.

- in: ist der Default Parameter. Nimmt Parameter entgegen und gibt nichts zurück
- out: Nimmt kein Parameter entgegen und gibt ein Wert zurück
- inout: Nimmt Parameter entgegen und gibt ein Wert zurück

Bei `inout` und `out` wird nur die Referenz übergeben. `out` wird bei `int` mit **0** initialiesiert. Man nennt das auch `call by reference`. Es wird hier also nicht der Wert direkt an die Funktion übergeben, sondern nur eine Refernez. Die Variable wird also nicht kopiert, sondern nur eine Referenz übergeben. In dieser Referenz steht dann drin wo der eigentliche Wert zu finden ist. Dann gibt es noch `call by value`. Hier wird der Wert kopiert und der Funktion zur Verfügung gestellt. Die Funktion bekommt also eine Kopie.

Noch mal ein Beispiel wo bei der `foreach` Schleife der Parameter `inout` verwendet wird.

Listing 43: foreach mit inout

```
1  int main() {
2
3      //size_t = a;
4
5      static int [5] b = [1,2,3,4,5];
6
7      foreach (inout int a;b) {
8          a +=5;
9      }
10
11     foreach ( int c; b) {
12         printf("c = %d\n",c);
13     }
14
15     return 0;
16 }
```

Die Werte vom Array `b` werden in Zeile 7 mit der `foreach` Schleife durchlaufen und in Zeile 8 mit 5 addiert. Gleichzeitig wird auch jedes Element im Array um 5 erhöht. Anschliessend wir in Zeile 11,12 und 13 das Array `b` ausgegeben.

Ausgabe des Programms:
c = 6
c = 7
c = 8

c = 9
c = 10

3.11.5 Geschachtelte Funktionen

3.12 Daten Konvertieren

3.13 unittest

Unittest dienen der Qualitätskontrolle und sollen Programmfehler vermeiden. Es bietet eine einfache Möglichkeit Komponenten automatisiert zu testen. Diese Tests können dann beliebig wiederholt werden. Es werden an Funktionen oder Methoden Parameter übergeben und deren Ergebnis werden mit Testdaten verglichen. Welche Methoden und Funktionen getestet werden sollen muss der Entwickler entscheiden, besonders Triviale Methoden wie z.B. Get und Setter Methoden brauchen sicherlich nicht getestet werden. Auch bei Änderungen am Quellcode kann es manchmal zu unerwünschten Nebeneffekten kommen, die durch Unittest aufgedeckt werden können. Aber auch bei Änderungen an Backendsystem wie z.B. Datenbanken, Webservices, LDAP Betriebsystem kann eine Überprüfung mit Unittest sinnvoll sein. Ausserdem stellt sich die Frage ob man nicht zuerst die Test schreiben kann und dann den eigentliche Programm. Solche vorgehenweise wird Testgetriebene (Test-Driven Development) Entwicklung genannt. Als weitere Vorgehnsweise ist noch das Top-Down-Verfahren, zu nennen. Hier wird sehr abstrakt vorgegangen und die `main` Funktion enthält nur weninge Zeilen Programmcode (zumindest sollte Sie das). Man beutzt in der main-Funktion schon weitere Funktionen oder Methoden/Klassen, die erst noch implimentiert werden müssen.

Listing 44: Unittest

```
1 import std.stdio;
2
3 unittest {
4     writefln("Good mornig Vietnam");
5 }
6
7 void main() {
8     writefln("Main Funktion");
9 }
```

Ausgabe:
Good mornig Vietnam
Main Funktion

Kompiliert wird das Programm mit dem Schalter **unittest**.
`dmd -unittest unittest.d`
Beim ausführen des Programms wird zuerst `Good morning Vietnam` ausgegeben, anschliessend `Main Funktion`. Es werden also zuerst die Unit-Tests ausgeführt und dann erst die `main`-Funktion.

3.13.1 assert

Asserts sind ein wichtiger Bestandteil von **Design by Contracts** (DBC). Es dient dazu innerhalb eines Programms zu prüfen ob der Variableninhalt einem erwartenden Wert entspricht.

Listing 45: Assert

```
1 import std.stdio;
2
3 int main() {
4     assert((4-3) == 1);
5     writefln("Hallo");
6     return 0;
7 }
```

In Zeile 4 wird der Ausdruck `(4-3) == 2` ausgewertet. In diesem Fall ist das Ergebnis falsch und es wird eine Exception geworfen.
`Error: AssertError Failure assert1.d(4)`
Die Abarbeitung des Programms wird beendet. Der Ergebnistyp des assert Ausdrucks ist `void`.

Listing 46: Assert

```
1  import std.stdio;
2  class Dif {
3      int sub(int x,int y) {
4          return x - y;
5      }
6
7      unittest {
8          Dif x = new Dif;
9          assert(x.sub(4,3) == 1);
10         writefln("Hallo Manfred");
11
12     }
13 }
14
15 int main() {
16     return 0;
17 }
18
19 //dmd -unittest assert.d
```

In Zeile 2 wird die Klasse `Dif` erstellt, sie enthält die Methode `sub`. Die Methode nimmt 2 integer Werte entgegen und subtrahiert sie in Zeile 4 und gibt das Ergebnis mit der return Anweisung zurück. Unittest in Zeile 7 wird nur aufgerufen, wenn mit der Option `-unittest` kompiliert wurde. In Zeile 8 wird ein neues Objekt `x` von der Klasse `Dif` erstellt. Das assert in Zeile 9 ruft die Methode `sub` auf und übergibt die Parameter `4` und `3` und bekommt als Ergebnis eine `1` zurück. Das Ergebnis wird nun mit `== 1` verglichen. Da `1 = 1` ist, wird die Abarbeitung der `unittest` Funktion fortgesetzt.

3.13.2 Testgetriebene Programmierung

Hier ein Beispiel wir man an eine testgetriebene Programmierung herangehen könnte.

Listing 47: TTD

```
1  import std.stdio;
2  import std.cpuid;
3  import std.string;
4  import std.regexp;
5
6  void main() {
7  }
8
9  unittest {
10     assert("Pentium"==getCPU());
11 }
12
13 char[] getCPU() {
14     return "Pentium";
15 }
16
17 /*
18 char[] getCPU() {
19     int i = 0;
20     char[][] lines = std.string.split(std.cpuid.toString(),"\n");
21     foreach(line;lines) {
22         if(i <= 0) {
23             i = std.regexp.find(line, "Pentium");
24         }
25     }
26     if (i > 0) {
27         return "Pentium";
28     } else {
29         return "CPU Unbekannt!";
30     }
31
32 }
33 */
```

Als erstes erstellt man eine `main`-Funktion, damit sich das Programm überhaupt übersetzten läßt. Anschließend die `Unittests`. Hier soll festgestellt werden ob der Prozessor ein Pentium ist. Als letztes würde ich dann die Funktion `getCPU` in Zeile 13 erstellen. Nun kann man die Tests durchführen, die ja jetzt noch erfolgreich durchlaufen werden, da die Funktion `getCPU` immer `Pentium` zuück gibt. Als letztes implimentiert man die richtige Funktion, Zeilen 18 bis 32. Die Funktion in Zeile 13 könnte man auch als `Mock-Objekt` bezeichnen. Das sind Dummy Methoden/Funktionen/Klassen die meistens fest einprogrammierte Werte zurück liefern. In unserem Fall eben `Pentium`. Sie finden auch Verwendung um Schnittstellen zu simulieren.

3.13.3 debug

Der Compiler unterstützt den Schalter `-debug`. Damit kann in Abhängigkeit vom Debug-Wert Programmcode ausgeführt werden.

Listing 48: debug

```
1  import std.stdio;
2
3  void main() {
4      writefln("hier");
5  }
6
7  unittest {
8      debug(CPU) {
9          writefln(getCPU());
10         assert("Pentium"==getCPU());
11     }
12     debug(ARM) {
13         assert("arm"==getCPU());
14     }
15 }
16
17 char[] getCPU() {
18     return "Pentium";
19 }
```

Kompiliert man das Programm mit `dmd -debug=CPU -unittest debug.d`, wird nur Zeile 13 **nicht** ausgeführt. Bei `-debug=ARM` werden Zeile 8 und 9 nicht ausgeführt. Somit ist eine weitere Steuerung der unittest möglich.

3.14 Design by Contract (DBC)

Mit Contract ist hier Vertrag gemeint, der zwischen dem Kunden und Anbieter erstellt wird. Als Anbieter muss man sich eine Klasse vorstellen, die Dienste, also Methoden zur verfügung stellt. Der Kunde ist der jenige, der die Klassen beutzt. In dem Vertrag werden Bedingungen festgehalten, die nicht verletzt werden dürfen. Es gibt

- Vorbedingungen
- Bedingunden während der Lebenszeit von Objekten (Invarianten)
- Nachbedingungen

Die Vorbedingungen werden mit dem Schlüsselwort, `in` die Nachbedingungen mit `out` und die Bedingungen die immer erfüllt sein müssen, werden mit `invariant` geprüft. Die Invarianten gehören zu einer Klasse, Sie werden deshalb auch an weitere Klassen weiter vererbt. Um die hier gezeigten Beispiele nachvollziehen zu können, sollten Sie das Kapitel `Objektorientierte Programmierung` durchgearbeitet haben.

3.14.1 invariant

Eine Invariante beschreibt Bedingungen, die beim Erstellen und während der Lebenszeit des Objekts immer erfüllt sein müssen. Diese Bedingungen werden jedes mal geprüft, wenn auf eine öffentliche Methode ausgeführt wird, oder bei Beendigung des Konstruktors. Auf private Methoden wird die Invariante nicht geprüft.

Listing 49: Invariant

```
1  import std.stdio;
2
3  class GeradeZahl {
4      public int gzahl;
5
6      invariant() {    // ! dmd 2.0 () zugekommen
7          assert((gzahl % 2) == 0);
8      }
9
10     this(int zahl) {
11         this.gzahl=zahl;
12     }
13 }
14
15 void main() {
16     GeradeZahl geradeZahl = new GeradeZahl(7);
17 }
```

Ausgabe:
Error: AssertError Failure invariant(7)

In Zeile 16 wir das Objekt `geradeZahl` mit dem Übergabeparameter für den Konstruktor erzeugt. Zeile 11 wird die Variable `gzahl` dem Wert `zahl` zugewiesen. Als Ausgabe bekommen wir ein `AssertError`, weil 9 natürlich keine gerade Zahl ist, denn 9 modulo 2 in Zeile 7 liefert Rest 1 und nicht 0.

Hier noch ein weiteres Beispiel, was die Verwendung von `in` und `out` zeigt.

Listing 50: Ofen

```
1  import std.stdio;
2  class Ofen {
3      char[] ofen;
4      int anzahlHolzstuecke,i;
5      this(char[] ofen) {
6          writefln("Konstruktor");
7          this.ofen=ofen;
8      }
9      invariant {
10         // nur Holzofen sollen erstellt werden
11         writefln("%d: Ofen= %s",++i, ofen);
12         assert(ofen == "Holzofen");
13     }
14     void setBrennstoff(char[] brennstoff,int anzahlHolzstuecke) in {
15         // Es soll nur Holz verbrannt werden
16         assert(brennstoff == "Holz");
17         writefln("assert setBrennstoff");
18     }
19     out {writefln("out setBrennstoff");}
20     body {
21         writefln("Methode setBrennstoff");
22         this.anzahlHolzstuecke= anzahlHolzstuecke;
23     }
24
25     int getVorlauftemperatur() out (result) {
26         // Vorlauftemperatur muss immer unter 100 Grad sein.
27         assert(result < 100);
28         writefln("assert getVorlauftemperatur");
29     } body {
30         // Vorlauftemperatur soll sich pro Holzstueck um 30 Grad
               erhoehen
31         //this.ofen="Gastherme";
32         writefln("Methode getVorlauftemperatur");
33         this.geblaese();
34         this.ofen="Holzofen";
35         return anzahlHolzstuecke * 30;
36     }
37     private void geblaese() {
38         writefln("Mathode geblaese");
39         this.ofen="Gastherme";
40         writefln("Ofen: %s", this.ofen);
41     }
42 }
43 void main() {
44     int anzahlHolzstuecke=3;
45     Ofen ofen = new Ofen("Holzofen");
46     ofen.setBrennstoff("Holz",anzahlHolzstuecke);
47     writefln("Vorlauftemperatur: %dGrad", ofen.getVorlauftemperatur());
48 }
```

Ausagbe:

```
Konstruktor
1: Ofen= Holzofen
assert setBrennstoff
2: Ofen= Holzofen
Methode setBrennstoff
3: Ofen= Holzofen
out setBrennstoff
4: Ofen= Holzofen
Methode getVorlauftemperatur
Mathode geblaese
Ofen: Gastherme
5: Ofen= Holzofen
assert getVorlauftemperatur
Vorlauftemperatur: 90Grad
```

Leider musste ich das Programm ein wenig quetschen, damit es auf eine Seite paßt, laß dich nicht von der Größe des Programms abschrecken, das meiste sind `writefln` anweisungen. Dieses Programm erstellt ein Holzofen, der nur Holz verfeuern kann und bei dem die Vorlauftemperatur nicht größer als 100 Grad werden darf. Gehen wir mal der Reihe nach die Ausgabe durch. Bei erstellen wir der Konstruktor aufgerufen und anschliessend gleich die Invariante `1: Ofen= Holzofen`. Als zweites rufen wir den Methode `setBrennstoff` in Zeile 46 auf. In dem `in` Block wird ebenfalls die Invariante `2: Ofen= Holzofen` aufgerufen. Dann geht es zum `body`-Block der Mehtode und wieder wird die Invariante `3: Ofen= Holzofen` überprüft. Nun geht es zum `out`-Block und die Invariante gibt `4: Ofen= Holzofen` aus. Bei der Methode `getVorlauftemperatur` wiederholt sich das Spiel. Jetzt wird aber in der Methode `getVorlauftemperatur` die Methode `geblaese` (Zeile 33) aufgerufen, die mit dem Attribut private belegt ist. Bei privaten Methoden wird die Invariante nicht überprüft, deshalb wir auch `Ofen: Gastherme` ausgegeben. In Zeile 34 muss die Variable `ofen` wieder auf `Holzofen` gesetzt werden, weil der `out`-Block noch abgearbeitet wird, und damit die Invariante überprüft wird. Ganz zum Schluß wird in Zeile 47 die `Vorlauftemperatur: 90Grad` ausgegeben.

Zusammenfassend kann man sagen das die Invariante beim `in,out` und `body` Block aufgerufen wird. Bei privaten Methoden wird die Invariante nicht überprüft. Mit dem `in`-Block findet eine Vorbetrachtung statt, also ob die übergebenen Parameter an die Methode einen gültigen Wert besitzen. In unserem Beispiel muß `brennstoff` immer Holz sein. Der `out`-Block überprüft den Rückgabewert, der in unserem Beispiel nicht groeßer als 100 Grad sein darf (Zeile 27). Das entspricht der Nachbetrachtung. Die Variable `result` wird nur gesetzt, wenn die Funktion oder Methode ein Rückgabewert besitzt. Wenn man das Programm mit der Compiler Option `-release` übersetzt, findet keine Überprüfung der Invariante statt. Das wirkt sich natürlich positiv auf die Geschwindigkeit aus.

3.15 Heap und Stack

Hier möchte ich noch auf die unterschiede vom `heap` und `stack` hinweisen, zumal das für das Verständnis von Rückgabewerte von Funktionen notwendig ist. Zum Schluss bringe ich hierzu noch ein Beispiel

Als erstes müssen wir mal klären was der Begriff `stack` bedeutet. Es gibt einmal ein `stack` der vom Prozessor benutzt wird. Hier werden Rücksprungadressen von Funktionen gespeichert, damit der Prozessor weiss zu welcher Adresse er springen muss, wenn eine Funktion beendet wurde. Es können dort auch lokale Variablen von laufenden Funktionen gespeichert werden.
Dann gibt es noch den `stack` der eine Datenstruktur beschreibt die nach dem `LIFO` (last in first out) Prizip arbeitet. Das heisst der Wert der als letztes dort gespeichert wird, wird auch als erstes ausgelesen. Man nennt das auch `Kellerspeicher` . Im Prinzip kann man sich vorstellen, das man z.B. 5 Kisten Bier übereinander gestapelt hat, um an die unterste Kiste heranzukommen, müssen erst die obersten 4 Kisten heruntergenommen (ausgelesen) werden, bevor man an die letzte Kiste dran kommt.

Die Werte im `Stack` können sehr effizient vom Prozessor ausgelesen werden, sie liegen immer an einer festen Position relativ zum `Stack-Pointer` (Adressregister). Die Werte werden mit `push` und `pop` in Assembler hinzugefügt, bzw. entfernt.

Der `heap` dient zur dynamischen Speicherverwaltung. Hier werden zusammenhängende Speicherbereiche reserviert (allokiert) bzw. wieder frei gegeben. Im `heap` reservierte Objekte können über Zeiger/References zugegriffen werden. Dises Referenzen sind auf dem `stack` gespeichert. Die Speicherverwaltung im `heap` ist wesentlich flexibler als im `stack`.

Nun fragt man sich warum man das als Programmierer wissen muss?
Erstens ist der Zugriff auf Variablen oder Referenzen schneller als auf dem `heap`. Zum anderen werden nach dem löschen einer Funktion die Variablen oder Referenzen auf dem `stack` gelöscht. Hierzu eine Auflistung was auf dem `stack` gespeichert wird:

- lokale Variablen
- Rücksprungsaddressen von Funktionen
- Argumente der Funktion
- primitive Typen
- struct
- union
- statische Arrays

Was auf dem `heap` gespeichert wird:

- Arrays und Objekte die mit new erstellt werden
- setzen der .length für ein dynamisches Array
- konkatinieren von Arrays mit ~= oder ~
- Aufruf von .dup auf einem Array

In `D` wird also eine Referenz im `stack` gespeichert, die auf das Objekt oder das Array zeigt. Das heisst wenn die Referenz im `stack` gelöscht wird, wird der `Garbage Collector` das Objekt auf dem `heap` löschen. Besteht noch eine weitere Referenz auf dem Stack wird das Objekt auf dem `heap` nicht gelöscht. Mit `delete` kann der Speicherbereich ebenfalls auf dem `heap` gelöscht werden.

Listing 51: heap stack

```
1  import std.stdio;
2
3  void main() {
4
5      int [] result = returnAnArray2();
6      writefln(result);
7      int b = retrunAnInt();
8      writefln(b);
9  }
10
11 int[] returnAnArray() {
12     //int[10] result;   // nicht ok
13     int[] result = new int[10];
14     result[] = 42;
15     return result;
16 }
17
18 int[] returnAnArray1() {
19     int[] result;
20     result.length = 10;
21     result[] = 42;
22     return result;
23 }
24
25 int[] returnAnArray2() {
26         int[10] result;
27         result[] = 42;
28         return result.dup;
29 }
30
31 // Geht, weil int ist "return by value" und nicht by reference
32 int retrunAnInt() {
33     int result = 69;
34     return result;
35 }
```

Wird nun z.B. die Funktion `returnAnArray` aufgerufen wird eine neue Schicht (frame) auf dem `stack` angelegt, dies geschieht völlig automatisch. Bei jedem Aufruf der Funktion wird ein neuer `frame` angelegt. In dem `frame` werden die lokalen Variablen gepeichert, hier die Variable `result` aus Zeile 12, die auskommentiert ist. Wird nun die Funktion verlassen, wird der `frame` vom `stack` wieder gelöscht und damit auch unsere Variable `return`. Deshalb muss die Variable auf dem `heap` gepeichert werden, sowie es in den Zeilen 13,20,26 und 33 durchgeführt wurde.

Integer Variablen werden als Wert zurückgegeben (return by value), das heisst die Integer Variable hätten wir auf dem `stack` anlegen können, Sie wird bei Ausführung der `return` Anweisung kopiert worden. Was auch noch bemerkenswert ist, das ein struct auf dem `stack` gespeichert wird.

4 Weitere Programmierung

4.1 pragma

Mit `pragma` können dem Compiler Hinweise gegeben werden, z.B. zur Optimierung. Es gibt vordefinierte Pragmas, z.B. `msg` um ausgaben während des kompilieren zu erstellen.

Listing 52: pragma

```
1 pragma(msg, "Hallo Compiler");
```

Ausgabe:
Hallo Compiler

Ein weiteres schönes Beispiel finden Sie unter [15]. Es wird mit `dmd -c -o- beer.d` kompiliert.

4.2 align

Listing 53: align

```
1  private import std.stdio;
2
3  align(4) struct NodeLink {
4      char c;
5      long node;
6  }
7  int main() {
8
9      struct NodeLink1 {
10         align(1):
11         char c;
12         long node;
13     }
14
15     struct NodeLink2 {
16         align(4):
17         char c;
18         long node;
19     }
20
21     struct NodeLink3 {
22         char c;
23         long node;
24     }
25
26     writefln("NodeLink:%d", NodeLink.sizeof);
27     writefln("NodeLink1:%d", NodeLink1.sizeof);
28     writefln("NodeLink2:%d", NodeLink2.sizeof);
29     writefln("NodeLink3:%d", NodeLink3.sizeof);
30     return 0;
31 }
```

Ausgabe:

```
NodeLink:12
NodeLink1:9
NodeLink:12
NodeLink3:16
```

Teil II.
Standard Bibliothek phobos

Die Standard Bibliothek `phobos` wird mit dem Compiler zusammen ausgeliefert. Die Bibliothek besteht aus einzelnen Modulen . Diese Module sind zu einer Bibliothek zusammen gefasst. Unter Linux ist das die `libphobos.a` und unter Windows heisst Sie `phobos.lib`. In diesen Modulen kann man sehen wie die Klassen, Methoden und Funktionen aufzurufen sind. Besonders interessant ist immer der Bereich `unittest` . Hier befinden sich Beispiele wie die Funktionen oder Methoden aufzurufen sind. Hier würde ich immer als erstes reinschauen um Information von Funktionen zu finden.

5 Ein und Ausgabe

Es gibt 2 Möglichkeiten Dateien zu bearbeiten, einmal mit dem Modul `stream` und mit dem Modul `File` .

5.1 Dateien mit Stream bearbeiten

Als erstes betrachten wir das Modul stream zum lesen und schreiben von Dateien.

5.1.1 Datei Zeilenweise lesen

Der Vorteil beim Zeilenweisen einlesen ist, das man den Dateiinhalt nicht in den Arbeitsspeicher lädt, sondern jede Zeile kann einzeln abgearbeitet werden. Solche Arbeitsweise sollte immer bevorzugt werden, es sei den man weiss im vorraus wie gross die Dateien sind. Ich werde hierzu auch ein Beispiel zeigen in der der Dateiinhalt in den Arbeitsspeicher geladen wird. Das verarbeiten des Dateiinhalts der direkt im Arbeitsspeicher gelesen wurde ist sicherlich schneller. Unter samples im `dmd,gdc` wird in dem Programm `wc` solch eine Verarbeitung durchgeführt.

Listing 54: Zeilenweise einlesen

```
1  import std.stream;
2
3  int main() {
4      //File file = new File("testdatei.txt");
5      File file = new File;
6      file.open("testdatei.txt",FileMode.In);
7      while (!file.eof()) {
8          printf("%.*s\n",file.readLine());
9      }
10     file.close();
11     return 0;
12 }
```

In Zeile 5 wird ein neues Objekt `file` von der Klasse `File` erstellt und in Zeile 6 wird die Methode `open` aufgerufen um das File zu öffnen. Die Datei wird mit `eof` in Zeile 7 solange gelesen, bis es auf das End of file Zeichen trifft. Mit `readLine` wird eine Zeile eingelesen und über die printf Funktion

ausgegeben. Als letztes wird in Zeile 10 mit close das File Handle geschlossen. Zeile 4 zeigt ein Beispiel wie man ein neues Obkekt erstellt und gleichzeitig die Datei `testdatei.txt` öffnet.

5.1.2 In Datei schreiben

Listing 55: Datei schreiben

```
1  import std.stream;
2
3  int main() {
4      File file = new File;
5      file.create("testdatei.txt",FileMode.Out);
6
7      file.writeLine("Zeile 1");
8      file.writeLine("Zeile 2");
9      file.writeString("Hello, world!");
10     file.writeString("Hello, world1!");
11
12     printf("Lesbar %d\n",file.readable);
13     printf("Lesbar %d\n",file.writeable);
14     if (file.writeable) {
15         printf("Datei ist schreibbar\n");
16     }
17     file.close();
18     return 0;
19 }
```

In Zeile 5 wird die Methode `create` aufgerufen, das die Datei `testdatei.txt` erstellt. Mit `writeLine` wird eine ganze Zeile in die Datei rein geschrieben, also auch mit einem Zeilenvorschub `'\n'`. Zeile 9 u. 10 wird ein String in die Datei geschreiben.

5.1.3 Anhängendes Schreiben

Dieses Programm erstellt eine neue Datei und schreibt zwei Zeilen in die Datei. Ist die Datei schon vorhanden wird sie **nicht** neu erstellt und es werden zwei Zeilen an die vorhandenen Zeilen angefügt.

Listing 56: Anhängendes schreiben

```
1  import std.stream;
2  import std.stdio;
3
4  int main(char[][] argv) {
5      File file = new File;
6      try {
7          file.open("testdatei.txt",FileMode.Out);
8      } catch (OpenException e) {
9          file.create("testdatei.txt",FileMode.Out);
10     }
11
12     file.seekEnd(0);
13     file.writeLine("Zeile 1");
14     file.writeLine("Zeile 2");
15     file.close();
16
17     return 0;
18 }
```

Zeile 7 öffnet eine vorhandene Datei, gibt es Sie nicht, wird eine Exception geworfen und die Anweisung in Zeile 8 wird ausgeführt, hier wird dann eine neue Datei erstellt. Mit der Funktion `seekEnd(0)` wird zur letzten Zeile in der Datei gesprungen, damit sie nicht überschrieben wird.

5.1.4 Datei mit Fehlerbehandlung

Ob eine Datei erfolgreich zum lesen oder schreiben geöffnet werden kann sollte man **immer** überprüfen. Hierzu gibt es im Allgemeinen zwei Möglichkeiten, die Funktion die man aufgerufen hat gibt einen Rückgabewert zurück den man dann auswerten muss. Hierzu werde ich ein Beispiel bei den MySQL Funktionen zeigen. Die zweite Art der Fehlerbehandlung ist, das man überprüft ob eine Exception geworfen wurde.
Als quasi dritte Möglichkeit eine Exception abzufangen, sehen sie im Beispiel auf Seite 125.

Listing 57: Datei schreiben mit Fehlerbehandlung

```
1  import std.stream;
2
3  int main() {
4      File file = new File;
5      try {
6          file.open("testdatei.txt",FileMode.Out);
7          file.writeLine("Zeile 1");
8          printf("Zeile wird bei einer Exception nicht
9              ausgefuehrt\n");
10         if (file.writeable) {
11             printf("Datei ist schreibbar\n");
12         }
13
14     } catch (OpenException e) {
15         printf("%.*s\n",e.toString());
16     } finally {
17         printf("finally ausfuehren\n");
18         file.close();
19     }
20
21     return 0;
22 }
```

Die zu überprüfende Methode muss in einem `try catch` Block eingeschlossen sein. Wenn die Methode `open` eine Exception wirft, weil die Datei z.B. nicht schreibbar ist, wird sie so zusagen mit catch eingefangen. Hier sollten Sie mal die Datei `testdate.txt` zum Beispiel mal die Rechte mit chmod auf 444 setzen. Jetzt wird beim Aufruf des Programms eine Exception geworfen.
`file 'testdatei.txt' not found`
die in Zeile 15 ausgegeben wird. Eine Exception wird mit throw geworfen. Die Meldung kommt aus dem Modul std.stream. Das Schlüsselwort `finally` sorgt dafür das die Anweisungen innerhalb des Blocks in jedem Fall ausgeführt werden. Hier wird der Filehandle mit `close` geschlossen.

5.1.5 Datei in Puffer einlesen

Listing 58: Datei mit Puffer einlesen

```
1  import std.stream;
2  import std.stdio;
3  import std.math;
4
5  int main() {
6      BufferedFile file = new BufferedFile;
7      file.open("testdatei.txt",FileMode.In);
8      while (!file.eof()) {
9          //printf("%.*s\n",file.readLine());
10         writefln("%s",file.readLine());
11     }
12     file.close();
13     return 0;
14 }
```

Hier wird in Zeile 5 mit einem Puffer, dem `BufferedFile` gearbeitet. Dieser Puffer sorgt dafür das nicht jedes Byte einzeln gelesen werden muss, somit verringert sich die Anzahl der Zugriffe auf externe Datenträger. Dies kann die Lesegeschwindigkeit erhöhen. Nach meiner persönlichen Erfahrung sollte man den *BufferedFile* immer verwenden, wenn man Dateien von der Festplatte einlesen will.

5.2 Dateien mit File bearbeiten

Zum bearbeiten von Dateien wird das Modul `std.file` benötigt. Mit dem Modul `std.file` werden die Daten nicht Zeilenweise eingelesen und verarbeitet, wie bei `std.stream`, sondern in ein Puffer eingelesen und anschliessend verarbeitet. Der vorteil liegt vermutlich in der schnelleren Verarbeitung der Daten. Der Nachteil, die Datei die man einliesst wird direkt in den Arbeitsspeicher eingelesen. Wenn die Datei grösser ist als der Arbeitsspeicher bekommt man hier Probleme. Unter Linux kann es sein, das der Prozess gekillt wird (kann man mit dmesg sehen) oder es wird erst versucht einiges im `Swap` bereich auszulagern.

5.2.1 Datei einlesen und ausgeben

Listing 59: Datei mit std.file.read einlesen

```
1  import std.file;
2  import std.stdio;
3
4  int main() {
5      void[] buf;
6      // Datei in ein Buffer laden
7      buf = read("testdatei.txt");
8      //writef("%s",cast(char[])buf);
9      // Einzelne Zeichen ausgeben
10     foreach (char wert; cast(char[])buf ) {
11         writef("%s",wert);
12     }
13
14     return 0;
15 }
```

In Zeile 7 wird die Datei in den Puffer `buf` eingelesen. Die Variable `buf` ist vom Typ `void[]`, deshalb muss in diesem Beispiel für die Ausgabe in Zeile 10 der Type `void[]` nach `char[]` gekastet werden. Das Besondere an dem `void` Typ ist, das dort beliebige Zeichen gespeichert werden können, also kann man damit auch binär Dateien eingelesen werden.

Listing 60: Datei mit std.string.splitlines ausgeben

```
1  import std.stdio,std.file;
2
3  int main() {
4
5      string[] lines;
6      string Text;
7      string File = "testdatei.txt";
8      if(!std.file.exists(File)) {
9          throw new Exception(std.string.format("File'%s'notfound.",File)
               );
10     } else {
11         Text = cast(string)std.file.read(File);
12         lines = std.string.splitlines(Text);
13     }
14     foreach (string line; lines) {
15         writefln(line);
16     }
17     return 0;
18 }
```

Der Vorteil ist, wenn man die Zeilen mit `splitlines`, wie in Zeile 12 aufteilt, ist es unabhängig vom Zeilenende. Er werden berücksichtigt:

- \n = Unix
- \r\n = Windows
- \r = Mac

Listing 61: Datei mit std.file.write schreiben

```
1  import std.file;
2  import std.stdio;
3
4  int main() {
5      string str1 = "Zeile 1\n", str2 = "Zeile 2 \n";
6      const(void)[] buf;
7
8      buf ~= str1; buf ~= str2;
9      write("test_neu.txt",buf);
10
11     return 0;
12 }
```

5.2.2 Datei und Verzeichnis Operationen

```
1  import std.file,std.stdio;
2
3  int main() {
4
5          void[] buf = cast(char[])"Zeile 3\n";
6
7          append("testdatei.txt",buf);
8
9          std.file.copy("testdatei.txt","testdatei.back");
10
11         std.file.rename("testdatei.txt","testdatei1.txt");
12
13         std.file.remove("testdatei1.txt");
14
15         ulong Size = getSize("testdatei.back");
16         writefln("Size = %d",Size);
17
18         if ( exists("testdatei.txt") ) {
19                 writefln("testdatei.back ist da");
20         } else {
21                 writefln("testdatei.back ist nicht da");
22         }
23
24         writefln("Datei Attribute /dev/xdb3 %o ",cast(int)getAttributes("/dev/hda"));
25         writefln("Datei Attribute /dev/video56 %o ",cast(int)getAttributes("/dev/ttys1"));
26         writefln("Datei Attribute hello %o ",cast(int)getAttributes("hello"));
27         writefln("Datei Attribute /sys %o ",cast(int)getAttributes("/sys"));
28         writefln("Datei Attribute /tmp %o ",cast(int)getAttributes("/tmp"));
29         writefln("Datei Attribute /hello.d %o ",cast(int)getAttributes("d_buch.tex"));
30         writefln("Datei Attribute /tmp/.gdm_socket %o ",cast(int)getAttributes("/tmp/.gdm_socket"));
31         writefln("Datei Attribute /dev/cdrom %o ",cast(int)getAttributes("/dev/cdrom"));
```

```
32
33          if ( isfile("d_buch.tex") ) {
34                  writefln("d_buch.tex ist eine Datei");
35          } else {
36                  writefln("d_buch.tex ist ein Verzeichnis");
37          }
38
39          if ( isdir("/tmp") ) {
40                  writefln("/tmp ist eine Datei");
41          } else {
42                  writefln("/tmp ist ein Verzeichnis");
43          }
44
45          std.file.copy("testdatei.back","testdatei.txt");
46
47          chdir("/tmp");
48          mkdir("/tmp/dir1");
49          rmdir("dir1"); // Erst in das Verzeichnis wechseln
50
51          writefln("Hole aktuelles Verzeichnis %s",getcwd());
52
53          foreach (string directory; listdir("/")) {
54                  writefln("directory = %s",directory);
55          }
56
57          return 0;
58  }
```

- Zeile 7: Mit append wir eine Zeile an einer Datei angehängt.
- Zeile 9: Kopiert eine Datei.
- Zeile 11: Datei wird umbenannt.
- Zeile 13: Datei `testdatei1.txt` wird gelöscht.
- Zeile 15: Datei größe wird ermittelt.
- Zeile 18: Prüft ob es die Datei `testdatei.txt` gibt.
- Zeile 24 bis 31: Datei Attribute werden ausgegeben. Man beachte das **%o**.
- Zeile 33: Wird geprüft ob `d_buch.d` eine Datei ist.
- Zeile 39: Wird geprüft ob `/tmp` ein Verzeichnis ist.
- Zeile 45: Kopiert eine Datei.
- Zeile 47: Wechselt ins Verzeichnis \`tmp`.
- Zeile 48: Erstellt ein Verzeichnis.
- Zeile 49: Löscht das Verzeichnis
- Zeile 51: Zeigt das aktuelle Verzeichnis an.
- Zeile 53 bis 55: Gibt alle Verzeichnisse vom root Verzeichnis aus.

6 Strings

Strings sind ein wichtiger Bestandteil jeder Programmiersprache. Strings in D werden als `struct` gespeichert. Dieses `struct` enthält den String und die Länge des Strings. Das ist also ein wesentlicher Unterschied zu `C/C++`. In `D` stellen sich die Strings aber als Array (char[]) dar.

6.1 String Manipulation

6.1.1 Substring

In dem `struct` werden die Strings als einzelne Zeichen, als `char` abgelegt. Auf diese `char` Elemente kann nun direkt über das Array angesprochen werden. Da Strings eine Aneinanderreihung von char Elementen sind, kann man die char Elemente auch einzeln aus einem Array ansprechen.

Listing 62: Substrings

```
1  import std.stdio;
2  import std.string;
3
4  void main() {
5      string str = "hallo";
6      writefln("slice %s", str[2..4]);
7      string r = replaceSlice(str, str[2..4], "UBUNTU");
8      //char[] r = replaceSlice(str, "ll", "br");
9      writefln("r = %s",r);
10
11     for (int i = 0; i < r.length; i++)
12         writefln("str %s",r[i]);
13
14 }
```

Ausgabe:

```
slice ll
r = haUBUNTUo
str h
str a
str U
str B
str U
str N
str T
str U
str o
```

In Zeile 4 wird ein dynamischen Array angelegt und dem String `Hello` zugewiesen. Mit dem slice Operator `..` werden die beiden Zeichen `ll` aus dem array `str` von char angezeigt. Mit der Zeile 7 werden die beiden Zeichen `ll` durch `UBUNTU` ersetzt. Dies geschieht mit der `replaceString` Funktion, die in `std.string` enthalten ist. Leider funktioniert die Funktion nur mir char Datentypen, nicht mit wchar oder dchar. In den Zeile 11 und 12 wird das array r durchlaufen und die einzelnen Zeichen werden ausgegeben.

6.1.2 Strings splitten

Listing 63: Strings splitten

```
1  import std.string;
2  import std.stdio;
3
4  int main () {
5      string  s = "peter,paul,jerry";
6      string[] words;
7
8      words = split(s, ",");
9      //printf("%.*s\n",words[1]);
10
11     foreach(string str; words) {
12         writefln(str);
13     }
14
15     return 0;
16 }
```

Hier wird mit der *split* (Zeile 8) Funktion der String *s* in Zeile 5 nach dem **,** auf gesplittet. Die einzelnen Namen werden in dem 2 Dimensionalem Array *words* gespeichert. Anschließend in Zeile 11,12 und 13 ausgegeben.

6.2 Reguläre Ausdrücke

7 Sonstige Funktionen

7.1 Random

Listing 64: Random

```
1  //import std.c.linux.linux;    // dmd
2  import std.c.unix.unix;       // gdc
3  import std.stdio;
4  import std.random;
5  import std.date;
6
7  void main ()
8  {
9
10   int anzahl,begin,end,zufallzahl,lg;
11   int[int] hashi;
12
13   anzahl=7;
14   begin=1;
15   end=49;
16
17   rand_seed(time(null),time(null));
18
19
20   while (lg < anzahl) {
21     zufallzahl=1+cast(int)(20*rand()/9999999+1.0); //legt groesse der
         zufallzahl fest
22     if(zufallzahl>=begin && zufallzahl<=end) {
23       hashi[zufallzahl]=zufallzahl;
24       lg=hashi.length;
25     }
26   }
27
28
29   writef("\nHeute ist LOTTO !!\n");
30   writef("\nTageziehung vom %s  <<<< 7 von 1 bis 49 >>>>\n",
       toDateString(getUTCtime()));
31   writef("=============================================\n")
       ;
32   foreach(int key; hashi.keys.sort) {
33     writef("[ %d ] ", hashi[key]);
34   }
35   writef("\n=============================================\n
       ");
36
37
38 }
```

Mit der Random Funktion werden Zufallszahlen generiert. *time(null)* liefert die Sekunden von `1.1.1970 00:00:00` Uhr. Dies ist der start Wert für die *rand_seed* Funktion in Zeile 15, diese wird für die *rand* Funktion benötigt.

Teil III.
Objektorientierte Programmierung

Die Objektorientierte Programmierung wurde eingeführt um Programmteile wieder verwenden zu können. Es soll die Entwicklungszeit verkürzen. Dadurch das Programmteile immer wieder verwendet werden erreichen sie Reife und Stabilität. Unsere Umgebung besteht auch aus lauter Objekten (Auto,Haus usw.) und an dieses Prinzip soll sich die Objektprogrammierung anlehnen. Unterstützt wird die einfache Vererbung, Interfaces und Polymorphismus. Die Objekte einer Klasser werden in `D` als Referenzen instanziiert.

8 Objekt und Klassen

Objekte werden aus einer Klasse erstellt. Man kann sich die Klasse erst mal wie ein Unterprogramm vorstellen mit zusätzlichen Eigenschaften. Manche beschreiben eine Klasse, als Bauplan, wie ein Objekt erstellt werden soll und was es alles können soll. Das was ein Objekt können soll, nennt man die Methoden und deren Eigenschaften sind die Attribute. Mit unter ist es auch sinnvoll sich ein Objekt wie eine Variable vorzustellen die man mit return zurückgeben kann. Der Typ der Variable wird durch die Klasse bestimmt. Als Beispiel stellen wir uns mal vor, wir wollen Häuser verkaufen. Der Kunde soll erst mal nur die Mölichkeit bekommen ein Standard Haus zu erwerben, später werden wir unsere Angebotpalette erweitern.

Listing 65: Haus

```
1 class Haus {
2 }
3
4 int main( char[][] arg ) {
5     Haus StandardHaus = new Haus;
6
7     return 0;
8 }
```

Wenn man das Programm ausführt, passiert erstmal noch nicht viel, doch in Zeile 1 wird mit dem Schlüsselwort `class` die Klassendefinition eingeleitet. Die Klassendefinition ist sozusagen unser Bauplan, der im Moment noch leer ist. In der Zeile 5 wird die Variable `StandardHaus` deklariert und initialisiert. Nun haben wir ein Objekt, das StandardHaus heisst, erstellt. Beim erstellen des Objekts wird Arbeitsspeicher belegt, bzw. allokiert, falls das fehl schlägt, wird eine `OutOfMemoryException` geworfen.

8.1 Methoden

Listing 66: Haus1

```
1  import std.stdio;
2  class Haus {
3      void klingeln() {
4              writefln("Ding Dong");
5      }
6  }
7
8  int main( char[][] arg ) {
9      Haus standardHaus = new Haus;
10     standardHaus.klingeln();
11     return 0;
12 }
```

Jetzt hat das Haus eine Türklingel bekommen. In Zeile 3 wird die Methode `klingeln()` definiert, dies kann man sich erstmal wie ein Unterprogramm vorstellen. Über der Punktnotation in Zeile 10 wir die Methode `klingeln()` aufgerufen. Man ruft also vom Objekt `standardHaus` die Methode `klingeln()` auf.

An einer Methode können Variablen übergeben werden und die Methode kann auch eine Variable zurück liefern, hat also ein Rückgabewert.

Listing 67: Haus2

```
1  import std.stdio;
2
3  class Haus {
4      string klingeln(int anzahl) {
5          for (int i = 0; i < anzahl;i++) {
6              writefln("Ding Dong");
7          }
8          if (anzahl < 1) {
9              return("Tuer bleibt zu!");
10         } else {
11             return("Tuer wird geoeffnet!");
12         }
13     }
14 }
15
16 int main( string[] arg ) {
17     Haus standardHaus = new Haus;
18     string tuer = standardHaus.klingeln(2);
19     writefln(tuer);
20     return 0;
21 }
```

Hier wird nun in Zeile 18 der Integer Wert 2 als Parameter an die Methode `klingeln()` übergeben. In Zeile 9 und 11 wird mit `return` ein String zurückgegeben und der Variable `tuer` in Zeile 18 zugewiesen.
Die Ausgabe des Programms ist:

```
Ding Dong
Ding Dong
Tuer wird geoeffnet!
```

Nun verhält sich die Methode wie eine Funktion oder einem Unterprogramm.
Es speilt auch keine Rolle ob die `main` Funktion vor oder hinter der Klasse steht. Die Anzahl der Methoden innerhalb einer Klasse ist beliebig.

8.2 Methoden überladen

Wenn es mehrere Methoden mit dem gleichen Namen gibt, kann der Compiler anhand der Parameter feststellen, welche Methode aufgerufen werden soll, sowas nennt man überladen von Methoden.

Listing 68: Haus5

```
1  import std.stdio;
2  class Haus {
3      void klingeln() {
4              writefln("Ding Dong");
5      }
6
7      void klingeln(int anzahl) {
8          writefln("Es klingelt %d mal",anzahl);
9      }
10 }
11
12 int main( char[][] arg ) {
13     Haus standardHaus = new Haus;
14     standardHaus.klingeln(2);
15     standardHaus.klingeln();
16     return 0;
17 }
```

Ausgabe:

```
Es klingelt 2 mal
Ding Dong
```

Hier gibt es 2 Methoden `klingeln()`, die eine wird in Zeile 14 mit eine Integer Parameter aufgerufen und die zweite Methode, in Zeile 15, wird ohne Parameter aufgerufen.

8.3 Variablen

Der Kunde soll die Möglichkeit bekommen verschiedene Dachpfannen und Verblender aussuchen zu können. Aus diesem Grunde werden sogenannte Instanzvariablen eingeführt.

Listing 69: Haus3

```
1  import std.stdio;
2  class Haus {
3      public string pfannenfarbe;
4      public string verbleander = "Gelb";
5      void farben() {
6          writefln("Farbe der Dachpfanne ist %s",this.pfannenfarbe);
7          writefln("Farbe des Verblaenders ist %s",verbleander);
8      }
9  }
10
11 int main( string[] arg ) {
12     Haus standardHaus = new Haus;
13     standardHaus.pfannenfarbe = "Rot";
14     standardHaus.farben();
15     standardHaus.verbleander = "Rot";
16     standardHaus.farben();
17     return 0;
18 }
```

Ausgabe:

```
Farbe der Dachpfanne ist Rot
Farbe des Verblaenders ist Gelb
Farbe der Dachpfanne ist Rot
Farbe des Verblaenders ist Rot
```

In Zeile 3 wird die Variable `pfannenfarbe`, man nennt sie auch Instanzvariable , deklariert, und in Zeile 4 ebenfalls, nur das die Instanzvariable `verbleander` auch initialisiert wird. Zeile 13 wird jetzt zu dem Objekt `standardHaus` der Instanzvariablen `pfannenfarbe` dem Wert `Rot` zugewiesen. Anschliessend wird wieder die Methode `farben()` aufgerufen. Die Zeile 15 überschreibt die Verblender Farbe Gelb mit Rot. Die Methode `farben()` kann auf die Instanzvariablen der eigenen Klasse zugreifen, was eigentlich nur über die Punktnotation gehen soll. Um das zu erreichen wurde das Schlüsselwort `this` eingeführt, wie man in Zeile 6 sieht. Also `this.pfannenfarbe`. Bei der Instanzvariable in Zeile 7 hab ich das `this` nicht verwendet, weil der Compiler automatisch ein `this` vor der Instanzvariable interpretiert. Hierzu im nächsten Beispiel mehr. Das `public` vor dem `char[]` in Zeile 3 wird `Attribut` genannt. Dem Attributen werde ich einem extra Kapitel widmen.

8.3.1 this

Hier möchte ich noch mal näher auf das Schlüsselwort `this` anhand eines Beispiels eingehen.

Listing 70: Haus4

```
1  import std.stdio;
2  class Haus {
3      public string pfannenfarbe;
4      public string verbleander = "Gelb";
5      void farben() {
6          string verbleander ="Gruen";
7          string pfannenfarbe ="Gruen";
8          writefln("Farbe der Dachpfanne ist %s",this.pfannenfarbe);
9          writefln("Farbe des Verblaenders ist %s",verbleander);
10     }
11 }
12
13 int main( string[] arg ) {
14     Haus standardHaus = new Haus;
15     standardHaus.pfannenfarbe = "Rot";
16     standardHaus.farben();
17     return 0;
18 }
```

Ausgabe:

```
Farbe der Dachpfanne ist Rot
Farbe des Verblaenders ist Gruen
```

Normalerweise würde man denken, das die `pfannenfarbe` ebenfalls grün ist, weil ja in Zeile 7, der Variable `pfannenfarbe` grün zugewiesen wurde. Hier handelt es sich aber um eine lokale Variable und nicht um eine Instanzvariable, und da vor dem `pfannenfarbe` in Zeile 8 `this` steht, wird die Instanzvariable ausgegeben. In Zeile 9 wird die lokale Variable `verblaender` ausgegeben. Innerhalb einer Methode kann man keine Instanzvariable deklarieren, sobald man ein Attribut vor der Variablen schreibt, gibt es eine Fehlermeldung.

8.4 Objekte als Typen

Ich hatte ja schon eingang erwähnt, das man sich Objekte wie Variablen vorstellen kann. Der Typ der Variable wird durch die Klasse bestimmt. Wenn nun ein Objekt erstellt wird, hat es den Typ der Klasse. Die Objekte können als Referenz an weiter Funktionen oder Methoden übergeben werden. Hierzu ein Beispiel.

Listing 71: Value Objekt

```
1  import std.stdio;
2
3  class Person {
4      private string vorname;
5      private int alter;
6
7      public void setVorname(string vorname) {
8          this.vorname = vorname;
9      }
10     public string getVorname() {
11         return this.vorname;
12     }
13
14     public void setAlter(int alter) {
15         this.alter = alter;
16     }
17     public int getAlter() {
18         return this.alter;
19     }
20 }
21
22 Person stammdaten(Person person) {
23     writefln(person.getVorname());
24     writefln(person.getAlter());
25     person.setVorname("Ursula");
26     return person;
27 }
28
29 void main() {
30     Person person = new Person;
31     person.setAlter(23);
32     person.setVorname("Urs");
33     // Objekt person an eine Funktion uebergeben
34     stammdaten(person);
35     writefln(person.getVorname());
36 }
```

Ausgabe:
Urs
23
Ursula

Hier haben wir eine Klasse `Person`. Sie besteht aus `set` und `get` Methoden (Setter und Getter Methoden) , in der Vorname und Alter abgespeichert werden können. Objekte die dazu dienen nur Werte (Values) abzuspeichern, nennt man `Value-Objekte`. In Zeile 30 wird das Objekt `person` vom Typ `Person` erstellt. Die Werte werden in Zeile 30 und 31 gesetzt. Nun wird in Zeile 34 das Objekt als Referenz an die Funktion `stammdaten` übergeben. In der Funktion `stammdaten` setze ich den Name von Urs auf Ursula um (Zeile 25). Anschließend gibt die Funktion das Objekt `person`

wieder an die `main` Funktion zurück und Ursula wird ausgegeben.

8.5 Konstruktor

Der Konstruktor wird mit `this()` eingeleitet und hat kein Rückgabewert wie es bei einer Methode der Fall ist. Konstruktoren werden beim initialisieren des Objekts ausgeführt und sie werden nicht vererbt.

Listing 72: Haus6

```
1  import std.stdio;
2  class Haus {
3      this() {
4          writefln("Dach ist undicht");
5      }
6  }
7
8  int main( char[][] arg ) {
9      Haus standardHaus = new Haus();
10     return 0;
11 }
```

Ausgabe:

```
Dach ist undicht
```

In Zeile 9 wird das Objekt `standardHaus` erzeugt und mit dem erzeugen wird auch der Kostruktor in Zeile 3 aufgerufen. Der Konstruktor hat hier keine Parameter.

Listing 73: Haus8

```
1  import std.stdio;
2  private import std.c.time;
3  class Haus {
4
5      public string farbe;
6
7      this(string farbe) {
8          writefln("Dach ist undicht");
9          this.farbe=farbe;
10         farbe = "Gelb";
11         writefln("Farbe = %s",this.farbe);
12         writefln("Farbe = %s",farbe);
13         this(2);
14     }
15
16     this(int anzGauben) {
17         writefln("Dach hat %d Gauben. ",anzGauben);
18         //this("Lila");
19     }
20 }
21
22 int main( char[][] arg ) {
23     Haus comfortHaus = new Haus("Braun");
24     Haus standardHaus = new Haus(1);
25     return 0;
26 }
```

Ausgabe:

```
Dach ist undicht
Farbe = Braun
Farbe = Gelb
Dach hat 2 Gauben.
Dach hat 1 Gauben.
```

Hier rufen wir in Zeile 23 den Konstruktor mit einem Parameter auf. Anhand des Parameter erkennt der Compiler welchen Konstruktor er aufrufen muss. In Zeile 24 wird als Parameter ein Integer Wert übergeben, deshalb wird hier der Konstruktor in Zeile 16 aufgerufen. Mit `this(2)` in Zeile 13 rufen wir ebenfalls der Konstruktor in Zeile 16 aufgerufen. Man nennt das `Verkettung von Kunstruktoren`. Würde man noch die Zeile 18 einkommentieren, dann wird der Konstruktor in Zeile 7 aufgerufen. Der wiederum ruft in Zeile 13 den Konstruktor in Zeile 16 auf, und das Spiel beginnt von vorne, also eine Endlosschleife. In den Zeilen 9 bis 10 hab ich noch mal ein Beispiel gebracht um den Unterschied zwischen lokaler und globaler Variablen zu zeigen. Die globale Variable `farbe` wird in Zeile 5 deklariert.

8.5.1 Static Konstruktor

Statische Konstruktoren werden mit dem Attribut `static` eingeleitet. Normalerweise wird beim erzeugen jedes Objektes der Konstruktor aufgerufen. Bei `static` wird der Konstruktor aber nur **einmal** aufgerufen.

Listing 74: Statischer Konstruktor

```
1  import std.stdio;
2
3  class A {
4      public static int OBJEKT=0;
5      static this() {
6          OBJEKT++;
7          writefln("Object = %d",OBJEKT);
8      }
9  }
10
11 int main( char[][] arg ) {
12     A a1 = new A;
13     A a2 = new A;
14     A a3 = new A;
15     writefln("Es gibt %d Objekt.",A.OBJEKT);
16     return 0;
17 }
```

Ausgabe:

```
Object = 1
Es gibt 1 Objekt.
```

Bei dem Programm werden in Zeile 12,13 und 14 3 Objekte erzeugt, bei denen jedesmal der Konstruktor aufgerufen wird, der die Klassenvariable `OBJEKT` um 1 hochzählt. Da nun aber der Konstruktor statisch ist (Zeile 5), wird er aber nur einmal aufgerufen, deshalb bekommen wir in der Ausgabe des Programms nur ein Objekt angezeigt, obwohl wir 3 Objekte erzeugt haben. Siehe auch Kapitel 8.10.4

8.6 Destruktor

Die Destruktor Methode wird aufgerufen, wenn das Objekt gelöscht wird. Der Garbage Collektor gibt dann den Arbeitsspeicher frei, welches vom Objekt belegt wurde.

Listing 75: Haus7

```
1  import std.stdio;
2  private import std.c.time;
3  class Haus {
4      this() {
5          writefln("Dach ist undicht");
6      }
7
8      ~this() {
9          writefln("Dach ist wieder dicht");
10     }
11 }
12
13 int main( char[][] arg ) {
14     Haus standardHaus = new Haus;
15     sleep(3);
16     delete standardHaus;
17     sleep(3);
18     return 0;
19 }
```

Ausgabe:

```
Dach ist undicht
Dach ist wieder dicht
```

Aufgrund der `sleep` Funktion wird erst 3 Sekunden später das Objekt `standardHaus` mit `delete` gelöscht. Und beim löschen wird der Destruktor aufgerufen. Sicherlich praktisch um offene Datenbankverbindungen zu schliessen, bevor das Objeke gelöscht wird.

8.6.1 Object auf null prüfen

Hier möchte ich kurz zeigen wie man ein Objekt mit dem Wert `null` erzeugt und wie man es richtig prüft.

Listing 76: objectNull

```
1  private import std.stdio;
2
3  class Foo {};
4
5  int main(char [][] args) {
6      Foo foo = new Foo;
7      delete(foo);
8      if ( foo is null ) {
9          writefln("foo ist null!");
10     }
11     return 0;
12 }
```

Mit `delete` in Zeile 7 wird das Objekt gelöscht, bzw. auf `null` gesetzt. Anschliessend wird in Zeile 8 geprüft ob es `null` ist. Folgenderweise sollte man **nicht** prüfen:

`if ( foo == null ) {`
denn hier kommt es zu einem Speicherzugriffsfehler, wenn `foo` null ist.

8.7 Vererbung von Klassen

In `D` können Klassen einfach, nicht mehrfach, vererbt werden. Hier ein einfaches Beispiel:

Listing 77: Haus9

```
1  private import std.stdio;
2
3  class Haus {
4      private string pfannenfarbe = "schwarz";
5      void klingeln() {
6          writefln("Ding Dong");
7          writefln("Pfannenfarbe = %s",this.pfannenfarbe);
8      }
9  }
10
11 class Villa : Haus {
12     void brennen() {
13         writefln("Der Kamin ist an!");
14     }
15 }
16
17 class Schloss : Villa {
18     public string pfannenfarbe="Rot";
19     void brennen() {
20         writefln("Der Kamin brennt im Schloss.");
21         writefln("Pfannenfarbe = %s",this.pfannenfarbe);
22     }
23 }
24
25 int main( string[] args ) {
26     Villa villa = new Villa();
27     villa.klingeln();
28     Schloss schloss = new Schloss();
29     schloss.klingeln();
30     schloss.brennen();
31     writefln(schloss.pfannenfarbe);
32     return 0;
33 }
```

Ausgabe:

```
Ding Dong
Pfannenfarbe = schwarz
Ding Dong
Pfannenfarbe = schwarz
Der Kamin brennt im Schloss.
Pfannenfarbe = Rot
Rot
```

In dem Programm haben wir drei Klassen Haus,Villa und Schloss. Die Klasse Villa erbt die Eigenschaften von `Haus`, dies geschieht in Zeile 11 mit dem `:` Operator. Die Klasse `Haus` wird auch Vaterklasse genannt. `Schloss` erbt die Eigenschaften von `Villa`. Die Vererbung kann beliebig fortgesetzt werden. Das erste `Ding Dong` in der Programmausgabe wird von der Methode `klingeln()` vom Objekt `villa` ausgegeben und anschliessend die Pfannenfarbe. Nun wird in Zeile 28 das Objekt schloss erstellt und die Methode `klingeln()` aufgerufen, es wird wieder `Ding Dong` ausgegeben und wieder die Pfannenfarbe. Es wird in Zeile 30 die Methode `brennen()` aufgerufen. Da in Zeile 18 die `pfannenfarbe` mit `Rot` initialisiert wird, wird in der Ausgabe des Programms die Pfannenfarbe Rot ausdgegeben. Als letztes wird in Zeile 30 noch mal die Farbe Rot ausgegeben.

8.8 Alles Super

Mit dem Schlüsselwort `super` erhält man Zugriff auf die überschriebenen Methoden der Vaterklasse oder auch Superklasse genannt.

Listing 78: Haus10

```
1  private import std.stdio;
2
3  class Villa {
4      void brennen() {
5          writefln("Der Kamin brennt in der Villa!");
6      }
7  }
8
9  class Schloss : Villa {
10     void brennen() {
11         writefln("Der Kamin brennt im Schloss!");
12     }
13     void altesbrennen() {
14         super.brennen();
15     }
16 }
17
18 int main( char[][] args ) {
19     Schloss schloss = new Schloss();
20     schloss.brennen();
21     schloss.altesbrennen();
22     return 0;
23 }
```

Ausgabe:

```
Der Kamin brennt im Schloss!
Der Kamin brennt in der Villa!
```

Über den Methodenaufruf in Zeile 21 wird mit hilfe von `super` in Zeile 14 die Methode `brennen()` von der Klasse `Villa` ausgeführt.

8.8.1 Super Konstruktoren

Listing 79: Haus11

```
1  private import std.stdio;
2
3  class Haus {
4      this(int hausnummer) {
5          writefln("Die Villa hat die Hausnummer %d.", hausnummer);
6      }
7  }
8
9  class Villa : Haus {
10
11     this() {
12         super(4);
13     }
14 }
15
16 int main( char[][] args ) {
17     Villa villa = new Villa();
18     return 0;
19 }
```

Ausgabe:

```
Die Villa hat die Hausnummer 4.
```

Da Konstruktoren nicht vererbt werden, können wir den Konstruktor der Klasse `Haus` vom Objekt `villa` nur über das Schlüsselwort `super` ausführen, so wie es in Zeile 12 passiert.

8.9 abstract

Von einer abstrakten Klasse können keine Objekte instanziiert (erzeugt) werden. Die Klasse dient als Signautur wie abgeleitete Klassen erstellt werden müssen.

Listing 80: formen

```
1  import std.stdio;
2  private import std.c.time;
3  public abstract class Form {
4      protected int breite, hoehe;
5      protected double radius;
6      protected abstract double berechneFlaeche();
7  }
8
9  public class Rechteck : Form {
10     public double berechneFlaeche() {
11         return this.breite * this.hoehe;
12     }
13 }
14
15 public class Kreis : Form {
16     public double berechneFlaeche() {
17         return 3.14 * this.radius * this.radius;
18     }
19 }
20
21 int main( char[][] arg ) {
22     Form kreis = new Kreis();
23     kreis.radius=2.0;
24     writefln("Kreisflaeche = %f",kreis.berechneFlaeche());
25
26     return 0;
27 }
```

Ausgabe:

```
Kreisflaeche = 12.560000
```

Die Klasse Form wird mit dem Schlüsselwort `abstract` definiert. Das bedeutet von dieser Klasse kann kein Objekt erstellt werden, sondern von Ihr kann nur abgeleitet werden. Die Klassen `Kreis` und `Rechteck` erben die Eigenschaften der Klasse Form. In Zeile 6 wird die Methode `berechneFlaeche()` mit dem Attribut `abstract` definiert. Wenn eine Klasse eine abstrakte Methode oder Variable enthält, muss die ganze Klasse selbst mit `abstract` definiert werden. Das Objekt `kreis` wird in Zeile 22 erzeugt.

8.9.1 Interfaces

In Interfaces werden Methoden fesgelegt, die von der geerbten Klasse implementiert werden müssen. Mit Interfaces ist es möglich Mehrfachvererbungen durchzuführen, das heisst die erbende Klasse muss alle Methoden der beiden, oder beliebig vielen, implementieren.

Listing 81: interface

```
1  import std.stdio;
2
3  interface Fisch {
4      private static final int i =1;
5      string kiemen();
6  }
7
8  interface Tier {
9      string beine();
10 }
11
12 class Amphibie : Fisch,Tier {
13     string kiemen() {
14         return "Unterwasser atmen.";
15     }
16     string beine() {
17         return "Auf dem trockenen laufen.";
18     }
19 }
20
21 int main( string[] arg ) {
22     Amphibie lurch = new Amphibie;
23     writefln(lurch.kiemen());
24     writefln(lurch.beine());
25     writefln("Zahl: %d",lurch.i);
26
27     return 0;
28 }
```

Ausgabe:

```
Unterwasser atmen.
Auf dem trockenen laufen.
Zahl: 1
```

Hier haben wir jetzt das Interface `Fisch` und `Tier`. Die Klasse `Amphibie` muss jetzt die beiden Methoden `kiemen()` und `beine()` implementieren. In Zeile 4 gibt es noch eine Konstante **i**, die braucht in der Klasse `Amphibie` nicht implementiert werden. Die erbenden Interfaces werden wie in Zeile 12 durch Komma getrennt an die Klasse `Amphibie` vererbt. Man hätte es auch so programmieren können.

Listing 82: interface1

```
1  import std.stdio;
2
3  interface Fisch {
4      private static final int i =1;
5      string kiemen();
6  }
7
8  interface Tier: Fisch {
9      string beine();
10 }
11
12 class Amphibie : Tier {
13     string kiemen() {
14         return "Unterwasser atmen.";
15     }
16     string beine() {
17         return "Auf dem trockenen laufen.";
18     }
19 }
20
21 int main( string[] arg ) {
22     Amphibie lurch = new Amphibie;
23     writefln(lurch.kiemen());
24     writefln(lurch.beine());
25     writefln("Zahl: %d",lurch.i);
26
27     return 0;
28 }
```

Hier ist jetzt nur Zeile 8 interessant, weil das Interface Tier vom Interface Fisch erbt.

Ein nettes Programm was ich im Internet gefunden habe, möchte ich nicht vorenthalten.

Listing 83: interface3

```
1  import std.stdio;
2
3  interface Animal {
4      string how_you_cry();
5  }
6
7  class Dog : Animal {string how_you_cry() {return "bark"; } }
8  class Cat : Animal {string how_you_cry() {return "mew"; } }
9  class Horse : Animal {string how_you_cry() {return "neigh"; } }
10 class Cow : Animal {string how_you_cry() {return "moo"; } }
11 class Mouse : Animal {string how_you_cry() {return "squeak"; } }
12
13 int main( string[] arg) {
14     Animal[] a;
15     a ~= new Dog;
16     a ~= new Cat;
17     a ~= new Horse;
18     a ~= new Cow;
19     a ~= new Mouse;
20     for(int i=0; i != a.length;++i) {
21         writefln( a[i].how_you_cry() );
22     }
23     return 0;
24 }
```

Ausgabe:

```
bark
mew
neigh
moo
squeak
```

Was hier neu ist, das in Zeile 14 ein Array von Objekten deklariert wird, diese wird dann in Zeile 15 bis 19 gefüllt. Diese einzelnen Objekte werden dann in der `for` Schleife durchlaufen und die Methode `how_you_cry` ausgeführt.

8.10 Protection Attribute

Ich möchte hier die Unterschiede von `private` , `protected` und `public` heraus arbeiten.

Methoden und Variablen die mit dem Attribut `public` versehen sind, können von anderen Klassen und Methoden aufgerufen werden. Sie sind also öffentlich. Bei `private` sind die Methoden und Variablen nur innerhalb der Klasse aufrufbar oder innerhalb eines Modules. Beim Attribut `protected` können die abgeleiteten Klassen auf die Varaiblen und Methoden der Vaterklasse zugreifen, also in der ganzen Vererbungshierachie. Wenn kein Attribut bei den Variablen oder Methode angegeben wird, ist sie automatisch `protected`.

Listing 84: Person

```
1  module Person;
2  private import std.stdio;
3
4  class Person {
5      protected string vorname= "Toni";
6      protected string getVorname() {
7          return this.vorname;
8      }
9
10 }
```

Listing 85: Hauptprogramm

```
1  import Person;
2  import Angestellter;
3  import std.stdio;
4
5  int main (char[][] arg ) {
6      Angestellter angestellt = new Angestellter();
7      angestellt.getVorname_person();
8      Person person = new Person();
9      //writefln("Vorname: %s",person.getVorname());
10     return 0;
11 }
```

Listing 86: Angestellter

```
1  module Angestellter;
2  private import std.stdio;
3  import Person;
4  class Angestellter : Person {
5      public void getVorname_person() {
6          writefln("Vorname: ",this.vorname);
7          writefln("Vorname: %s",this.getVorname());
8      }
9  }
```

Ausgabe:

```
Vorname: Toni
Vorname: Toni
```

Übersetzt werden die 3 Programme mit:
`dmd PersonHaupt.d Angestellter.d Person.d`
Würde man die Variable `vorname` auf `private` setzen könnte man Sie nur noch innerhalb der Klasse bzw. des modules `Person` aufrufen. Das selbe gilt auch für die Methode. Nur wenn man in der Klasse `Person` die Variable `vorname` auf `public` setzen würde, kann man Zeile 9 im Hauptprogramm einkommentieren, ohne eine Fehlermeldung beim kompilieren zu bekommen.

8.10.1 package

Mit dem Attribut `package` können Klassen auf Methoden und Variablen untereinander zugreifen, wenn Sie im selben Package liegen.

Listing 87: PersonPackage

```
1  import pack.Person;
2  import pack.Angestellter;
3  import pack.Arbeiter;
4  import std.stdio;
5  //dmd Person_haupt.d pack/Angestellter.d pack/Person.d pack/Arbeiter.d
6
7  int main (char[][] arg ) {
8      Angestellter angestellt = new Angestellter();
9      angestellt.getVorname_person();
10     //Person person = new Person();
11     //writefln("Vorname: %s",person.getVorname());
12     //writefln("Vorname: %s",person.vorname);
13     Arbeiter arbeiter = new Arbeiter();
14     writefln("Vorname_arbeiter: %s",arbeiter.getVorname_arbeiter());
15
16     return 0;
17 }
```

Listing 88: Person

```
1  module pack.Person;
2  private import std.stdio;
3
4  class Person {
5      package string vorname= "Toni";
6      protected string getVorname() {
7          return this.vorname;
8      }
9
10 }
```

Listing 89: Arbeiter

```
1  module pack.Arbeiter;
2  import pack.Person;
3  private import std.stdio;
4
5  class Arbeiter {
6      string getVorname_arbeiter() {
7          Person person = new Person();
8          return person.vorname;
9      }
10
11 }
```

Listing 90: Angestellter

```
1  module pack.Angestellter;
2  private import std.stdio;
3  import pack.Person;
4  class Angestellter : Person {
5      public void getVorname_person() {
6          writefln("Vorname_angestellter: ",this.vorname);
7          writefln("Vorname_angestellter: %s",this.getVorname());
8      }
9  }
```

Ausgabe:

```
Vorname_angestellter: Toni
Vorname_angestellter: Toni
Vorname_arbeiter: Toni
```

Übersetzen lässt sich das Programm `PersonPackage` mit
`dmd PersonPackage.d pack/Angestellter.d pack/Person.d pack/Arbeiter.d`.
Das Wort `pack` ist der package Name. Ein Package spiegelt immer ein Unterverzeichnis wieder. Deshalb müssen die 3 Module `Angestellter.d,Person.d` und `Arbeiter.d` in dem Unterverzeichnis pack liegen. Also alle die im selben Package `pack` sind, können auf die Varibale `vorname` in Zeile 5 von der Klasse `Person` zugreifen. Deshalb kann auch von der Klasse `Arbeiter` auf die Variable `vorname` von der Klasse `Person` zugegrifen werden. Von der Klasse `Angestellter` kann auch auf die Varibale `vorname` zugegriffen werden, weil zum einen die Klasse `Angestellter` von `Person` abgeleitet wurde und weil sie im selben Package liegt. Würde man Zeile 10 und 11 oder 12 einkommentieren, wird vom Kompiler ein Fehler angezeigt: `PersonPackage.d(11): class pack.Person.Person member getVorname is not accessible` , Das Hauptprogramm gehört nicht zum Package `pack`, deshalb der Fehler.

8.10.2 const

Mit `const` werden Variablen als Konstant definiert, sie können nicht mehr verändert werden. In Java würde man das mit final machen.

Listing 91: const

```
1  import std.stdio;
2
3  class Wasser {
4      const char[] wasser="H2O";
5  }
6
7  int main( char[][] arg ) {
8      Wasser wasser = new Wasser;
9      writefln("Wasser: %s",wasser.wasser);
10     //wasser.wasser="H2O2";
11     return 0;
12 }
```

Ausgabe:

```
Wasser: H2O
```

Würde man die Zeile 10 einkommentieren, kommt es zu folgender Fehlermeldung:
`const.d(4): wasser.wasser is not an lvalue`

8.10.3 final

In `D` gibt es keine `final` Klassen wie in `Java`, das heisst man kann auch von diesen Klassen erben. Es werden aber alle Methoden als `final` deklariert, die sich innerhalb der Klasse befinden. Diese Methoden können dann ncht mehr überschrieben werden.

Listing 92: Final Methoden

```
1  import std.stdio;
2
3  /* final */ class A {
4      /* final */ void getName() {
5          writefln("Klasse A");
6      }
7  }
8
9  class B : A {
10     void getName() {
11         writefln("Klasse B");
12     }
13 }
14
15 int main( char[][] arg ) {
16     B b = new B;
17     b.getName();
18     return 0;
19 }
```

Ausgabe:

```
Klasse B
```

Hier hab ich die `final` in Zeile 3 und 4 auskommentiert, weil es sonst zu einem Fehler kommt.

8.10.4 static

Variablen, die mit dem Atributt `static` aufgerufen werden, nennt man `Klassenvariablen` , und bei Methoden nennt man es `Klassenmethoden` . Sie können unabhängig von einem Objekt aufgerufen werden. Dadurch das man Klassenvariablen in allen Objekten aufrufen kann, wirken sie wie globale Variablen.

Listing 93: Klassenmethoden/Klassenvariablen

```
1  import std.stdio;
2
3  class A {
4      public static int OBJEKT=0;
5
6      public static void klassenMethode() {
7          writefln("Mich kann man ohne Objekt aufrufen!");
8      }
9
10     this() {
11         OBJEKT++;
12         writefln("Object = %d",OBJEKT);
13         //A.klassenMethode();
14     }
15 }
16
17 int main( char[][] arg ) {
18     A.klassenMethode();
19     A a1 = new A;
20     A a2 = new A;
21     A a3 = new A;
22     writefln("Es gibt %d Objekte.",A.OBJEKT);
23     return 0;
24 }
```

Ausgabe:

```
Mich kann man ohne Objekt aufrufen!
i = 1
i = 2
i = 3
Es gibt 3 Objekte.
```

Das Programm zählt die Anzahl der Objekte die von der Klasse `A` erstellt werden. Dies geschiet mit der Klassenvariable `OBJEKT`, sie wird mit dem Konstruktor beim erzeugen eines Objektes um eins hochgezählt. Da Klassenvariablen sich wie globale Variablen verhalten, sollte man sie gross schreiben. Die Klassenvariablen bzw. Klassenmethoden werden mit der Punktnotation über die **Klasse** aufgerufen, bei den Instanzvariablen bzw. Instanzmethode wird mit der Punktnotation über das Objekt aufgerufen. Für statische Konstruktoren siehe bitte Kapitel 8.5.1.

8.10.5 override

Das `override` Attribut zeigt, das die Methode von der Vaterklasse überschrieben werden muss. Wenn man eine Methode mit dem Attribut `override` versieht, muss es die Methode schon in der Vaterklasse geben. Wenn nun in der Vaterklasse die Methode entfernt wird, oder die Parameter ändern sich, tritt sofort ein Fehler beim kompilieren auf.

Listing 94: override

```
1  import std.stdio;
2
3  class A {
4      void getName() {}
5  }
6
7  class B : A {
8      override void getName() {
9          writefln("Klasse B");
10     }
11
12 }
13
14 int main( char[][] arg ) {
15     B b = new B;
16     b.getName();
17     return 0;
18 }
```

Ausgabe:

```
Klasse B
```

In Zeile 8 wird mit dem Attribut `override` die Methode `getName` versehen. Wenn man die Methode in Zeile 4 entfernen würde, gibit es einen Fehler beim kompilieren.

Teil IV.
Fortgeschrittene Programmierung

9 Fortgeschrittene Programmierung

Bei der Fortgeschrittenen Programmierung wird es um Tamplates, Boxing, Interfaces Assembler usw. gehen.

9.1 Boxing

Beim Boxing können beliebige Datentypen in einer Box gespeichert werden und erst beim auslesen der Box muss der Typ mit anegegeben werden.

Listing 95: Box

```
1  private import std.boxer;
2  private import std.stdio;
3
4  int main() {
5
6      Box z = box("foobar");
7      char[] w = unbox!(char[])(z);
8      writefln("w = %s",w);
9
10     Box x = box(4);
11     float f;
12     try f = unbox!(float)(x);
13     catch (UnboxException error) {
14         writefln("Geht nicht\n");
15     }
16     writefln("float = %f",f);
17
18     char c;
19     try c = unbox!(char)(x);
20     catch (UnboxException error) {
21         writefln("Geht nicht");
22     }
23
24     return 0;
25 }
```

Das Programm muss mit `-release` übersetzt werden, sonst kommt es beim linken zu einer Fehlermeldung. In Zeile 6 wird die Box deklariert und mit dem Wert `foobar` initialiesiert. Zeile 7 wird der Wert mit `unbox!` wieder ausgelesen und der Datentyp muß mit angegeben werden, in diesem Fall `char[]`. Falls beim unboxing ein Wert nicht in den entsprechenden Datentyp umgewandelt werden kann wird eine Exception geworfen, wie z.B. in Zeile 13 und 20. Zum Beispiel wenn man vom Datentyp `float` nach `int` wechselt. Aber auch die `4` lässt sich in Zeile 20 nicht in ein `char` umwandeln, weil char mit UTF-8 kodiert wird. Würde man in Zeile 18 und 19 char durch ubyte

oder byte ersetzen, dann wird keine Exception mehr geworfen.

9.1.1 Box mit Hash

Listing 96: Hash Box

```
1  private import std.boxer;
2  private import std.stdio;
3
4  void main() {
5      char[] s = "key";
6      int[Box] array;
7      array[box(s)] = 42;
8      assert(box(s) in array );
9      char[] str= unbox!(char[])(box(s));
10     int i= (array[box(s)]);
11     writefln("str = %s", str);
12     writefln("i = %d",i);
13 }
```

9.2 shared Libraries unter Linux

Shared Libraries werden zur Laufzeit dynamisch zu dem ausführbaren Programm hinzugeladen. Im gegensatz zur statischen Libraries, die beim Linken zum Programm hinzu kopiert werden. Der Vorteil ist, das Programme die shared Libraries benutzen kleiner sind, weil ja die Funktionen nicht direkt mit hinzu gelinkt werden. Ausserdem können im begrenzten Umfang shared Libraries ausgetauscht werden, ohne das Programme neu übersetzt werden müssen. Das werde ich auch an einem Beispiel zeigen. Ausserdem können shared Libraries von anderen Programmen ebenfalls benutzt werden.

Das Programm `app` ist das ausführbare Programm, es enthält auch die `main` Funktion.

Listing 97: app

```
1  import std.stdio;
2  import libsquare;
3
4  void main() {
5      writefln("4x4 = %d",squareIt(4));
6  }
```

`libsquare` wird die shared Library.

Listing 98: libsquare

```
1  module libsquare;
2
3  int squareIt(int i) {
4      return i*i;
5  }
```

Kompilieren mit dem `dmd`:
`dmd -c -H libsquare.d`

```
gcc -shared -o libsquare.so libsquare.o
dmd app.d -L-lsquare
```

Kompilieren mit dem `gdc`:

```
gdc -c -fintfc libsquare.d
gcc -shared -o libsquare.so libsquare.o
gdc app.d -o app -lsquare
```

Die `libsquare.so` nach `/usr/local/lib` kopieren.

Wenn Sie jetzt ein `ldd app` ausführen, bekommen Sie folgende Ausgabe:

```
linux-gate.so.1 =>  (0xffffe000)
libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7f1d000)
libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7ef7000)
libsquare.so => /usr/local/lib/libsquare.so (0xb7ef4000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7dc0000)
/lib/ld-linux.so.2 (0xb7f43000)
```

Die `libsquare.so` wird nun von `app` benötigt. Mit dem Flag `-H` bei `dmd` oder `-fintfc` für den `gdc` werden Header Dateien erzeugt mit der Dateiendung `di`. Die werden benötigt, wenn man die `libsquare.d` nicht zur Verfügung hat, z.B. dann, wenn `app` in einem anderem Verzeichnis als `libsquare.d` liegt.

Nun soll nur die `libsquare.d` Datei geändert werden. Verändern Sie Zeile 4 in `return i*i*i` und erstellen die shared Library neu. Führen sie jetzt das Programm `app` aus, erhalten Sie als Ergebnis: `4x4 = 64` statt `4x4 = 16`, ohne das Programm `app` neu kompilieren zu müssen.

9.3 delegate

Delegates bestehen aus 2 Teilen zum einen sind es typisierte Funktionszeiger und zum anderen eine Referenz auf Methoden, die zur einer Klasse gehören. Sie werden also ähnlich wie Funktionszeiger (*) wie in `C` verwendet. Ausserdem kann man mit delegates auf stack Variablen zugreifen, also auf Variablen die sich lokal innerhalb einer Funktion befinden.

9.3.1 delegate mit Funktionen

Listing 99: delegate

```
1  import std.stdio;
2
3  int delegate() dg;
4
5  void main() {
6      int a = 7;
7      int foo() {
8          return a + 3;
9      }
10
11     dg = &foo;
12     writefln(&dg);
13     writefln("a = %d",a);
14     int i = dg();
15     writefln("i = %d",i);
16 }
```

Ausgabe:

```
a = 7
i = 10
```

In Zeile 3 wird ein delegate `dg` deklariert. Das `int` gibt den Typ an, der zurückgegeben wird. Nun wird in Zeile 10 `dg` initialisiert. Zeile 12 lass ich noch mal die Adresse vom `dg` ausgeben. Als letztes wird in Zeile 14 die Funktion `foo` über delegate aufgerufen und ausgeführt, und der Rückgabewerti, wird der Variablen `i` zugewiesen.

Listing 100: delegate1

```
1  import std.stdio;
2
3  int delegate(int) dg;  // deklaration einses delegates(funktions Zeiger
       ) dg
4
5  void main() {
6      //int a = 7;
7      int foo(int a) {
8          return a + 3;
9      }
10
11     dg = &foo;    // Initialiesierung dg zur Funktion foo
12     int i = dg(8); // Funktion foo wird aufgerufen und i wird auf 10
           gesetzt
13     writefln("i = %d",i);
14 }
```

Ausgabe:

```
i = 11
```

Dieses Beispiel zeigt wie man einen Parameter mit delegate übergibt, desahlb wird in Zeile 3 `delegate(int)` mit übergeben. In Zeile 12 wird das dann als Parameter der Wert 8 übergeben.

Listing 101: delegate2

```
1  import std.stdio;
2
3  class Abc {
4      string s;
5      void print() {
6          writefln(s);
7      }
8  }
9
10 int main( string[] arg ) {
11     Abc a = new Abc;
12     void delegate() ev = &a.print;
13
14     a.s = "aeiou";
15     ev();
16     return 0;
17 }
```

Ausgabe:

```
aeiou
```

Hier wird statt einer Funktion die Methode `print` von der Klasse `ABC` aufgerufen. Die Deklaration und Initialisierung vom delegate geschieht hier in Zeile 12. In Zeile 14 wird der Instanzvariablen `s` die Vokale zugewiesen. Anschliessend wird in Zeile 15 die Methode `print` ausgeführt.

9.3.2 delegate mit Methoden

Ein delegate Beispiel für Methoden.

Listing 102: Delegate mit Methoden

```
1  import std.stdio;
2
3  class Quad {
4      int quad(int i) {
5          return i*i;
6      }
7  }
8
9  void main() {
10     int delegate(int) dg;
11     Quad q = new Quad;
12     dg = &q.quad;
13     writefln("dg: ", dg(3));
14
15 }
```

Ausgabe:

```
dg: 9
```

Hier wird in Zeile 11 das Objekt `q` erstellt und in Zeile 12 wird dann die Adresse der Methode `quad` übergeben. In Zeile 13 wird mit dem delegate und dem Parameter 3 die Methode `quad` aufgerufen.

9.3.3 delegate mit struct

Listing 103: Delegate mit Strukt

```
1  import std.stdio;
2
3  struct Foo {
4      int bar(int i) {
5          writefln("i: ",i);
6          return i*i;
7      }
8  }
9
10 int foo(int delegate(int) dg) {
11     return dg(3) + 1;
12 }
13
14 void main() {
15     Foo f;
16     int k = foo(&f.bar);
17     writefln("k: ",k);
18 }
```

So jetzt wird es ein etwas komplizierter, es wird in Zeile 16 die Funktion `foo` aufgerufen mit der Adresse vom struct `Foo` und der Funktion `bar`. Nun wird in Zeile 11 `delegate` mit Parameter 3

aufgerufen, das wiederum ruft die Funktion `bar` in `Foo` auf und gibt als Wert 9 zurück. Anschliessend wird dann +1 in Zeile 11 dazu addiert und letztendlich in Zeile 16 der Wert 10 der Variablen `k` zugewiesen.

9.3.4 delegate mit statement

Listing 104: Delegate mit Anweisungen

```
1  import std.stdio;
2
3  void main() {
4      writefln("Wert: ",test());
5  }
6
7  double test()
8  {   double d = 7.6;
9      float f = 2.3;
10
11     void schleife(int k, int j, void delegate() statement)
12     {
13         for (int i = k; i < j; i++)
14         {
15             statement();
16         writefln("f: ",f);
17         writefln("d: ",d);
18         }
19     }
20
21     schleife(5, 7, { d += 1;writefln("Ausgabe");f += 3; } );
22     //schleife(3, 10,  { f += 3; } );
23
24     return d + f;
25 }
```

Ausagbe des Programms:

```
Ausgabe
f: 5.3
d: 8.6
Ausgabe
f: 8.3
d: 9.6
Wert: 17.9
```

In Zeile 21 wird die Funktion `Schleife` mit den Parametern vom Typ `int` und `delegate` aufgerufen. Das `delegate` ist vom Rückgabetyp `void`. Der Witz ist hier aber, das nicht irgendwelche Werte oder Objekte an die Funktion Schleife übergeben wird, sonden mehrere Anweisungen. Das Programm läuft folgendermassen ab, start in Zeile 3, wie immer mit der `main` Funktion, anschliessend wird in Zeile 4 die Funktion `test` aufgerufen. Von dort geht es zur Zeile 7, die Variablen `d,f` werden deklariert und initialisiert, nun wird in Zeile 21 die Funktion `schleife` einmal aufgerufen und es geht zur Zeile 11. Die Variablen `k` hat jetzt den Wert 5 und `j` hat den Wert 7.

Deshalb wird die `for`-Schleife in Zeile 13 auch nur 2-mal durchlaufen. Nun werden die Anweisungen `d += 1;writefln("Ausgabe");f += 3;` ausgeführt. Damit werden die Variablen `f` mit 1 und `d` mit 3 aufaddiert.

9.3.5 Anonyme delegate

Listing 105: Anonyme delegates

```
1  import std.stdio;
2
3  void main()
4  {
5      auto dg = delegate (int delegate( inout int) dgp)
6      {
7          static int j=0;
8          int res=0;
9          while( j < 5){
10             j++;
11             res= dgp(j);
12             if(res) break;
13         }
14         return res;
15     };
16
17     foreach (int i; dg)
18     {
19         writefln(i);
20     }
21 }
```

Ausagbe des Programms:

```
1
2
3
4
5
```

Leider muss ich gestehen, das ich das Programm nicht vollständig verstanden habe. Falls jemand mehr weiss dann mit bitte eine Mail senden.

In Zeile 5 wird ein anonymes delegate erzeugt das wiederum als Argument ein delegate besitzt. Der Programm ablauf ist folgender:
Es wird nur **einmal** in der `foreach`-Schleife das anonyme delegate `dg` aufgerufen. Es läuft dann in die `while`-Schleife, `j` wird um 1 inkrementiert. Dann wird in Zeile 11 delegate dgp ausgeführt mit dem Argument `j` und wird dann in Zeile 19 ausgegeben. Warum er da hinspringt ist mir noch ein Rätsel. Wenn Sie noch mal `dgp(j)` einfügen wird der Rumpf der `foreach`-Schleife noch mal durchlaufen. Das ganze procedere wird 5-mal durchgeführt. Erst dann wird das anonyme delegate in Zeile 14 mit `return` verlassen. Der Typ vom anonymen delegate `dg` wird aufgrund des Typs vom `return`-Wertes bestimmt. Deshalb steht in Zeile 5 auch ein `auto` vor dem delegate.

10 Template

Mit Templates (Schablonen) lassen sich Funktionen unabhängig vom Datentyp erstellen. So ist es möglich die Funktionen unabhängig vom Typ immer wieder zu verwenden. Erst bei der Instanzierung muss der Datentyp festgelegt werden. Als Parameter können dem Template Typen, Werte oder Symbole übergeben werden. Die Werte müssen vom Typ sein, die Bestandteil der Sprache D sind , z.B. Fließkommazahlen, Flie kommakonstanten kommakonstaneten, null oder Strings. Ein Template hat seinen eigenen Namensbereich, das heißt die Variablennamen können frei gewählt werden. Innerhalb des Templates können nicht nur Funktionen erstellt werden, sondern auch Structs, Enums, Variablen, Typen und weitere Templates.

10.1 Template Funktionen

Hier werden innerhalb des Template Funktionen erstellt, die dann typunabh ngig aufgerufen werden können.

Listing 106: Quadrierung

```
1  import std.stdio;
2
3  template TQuad(T) {
4      void quad(inout T a) {
5          a = a * a;
6      }
7  }
8
9  void main() {
10     double a= 4.5;
11     TQuad!(double).quad(a);
12     writefln("a = ",a);
13 }
```

Ausgabe:
a = 20.25

In Zeile 3 wird das Template `TQuad` mit dem Schlüsselwort `template` deklariert. Hier wird der Parameter `T` übergeben. Man sieht das dieser Parameter noch gar keinem Typ (int, char usw.) zugewiesen wurde. Das geschieht erst beim initialisieren in Zeile 11. Mit Zeile 4 wird die Template-Funktion deklariert. `void` bedeutet das es keinen Rückgabewert gibt, der Typ des Parameters `a` ist auch noch unbekannt und wird später mit dem Template Parameter bestimmt. Mann kann sich das auch als Platzhalter für den anzugebenden Typ vorstellen. Das `inout` wird hier quasi als Zeiger interpretiert, es wird hier also keine Kopie der Variable erstellt, sondern nur die Referenz übergeben. In Zeile 11 wird das Template initialisiert. `TQuad` ist der Templatename und der Typ wird hier mit `double` festgelegt. Die Template Funktion `quad` wird hier mit dem Parameter `a` aufgerufen.

Listing 107: Quadrierung

```
1  import std.stdio;
2
3  template TQuad(T) {
4      T quad(T a) {
5          a = a * a;
6          return a;
7      }
8  }
9
10 void main() {
11     double a= 4.5;
12     a = TQuad!(double).quad(a);
13     writefln("a = ",a);
14 }
```

Bei diesem Beispiel gibt die Template-Funktion ein Wert zurück. Deshalb wird in Zeile 4 ein Rückgabewert des Template Parameters `T` zurückgegeben. Außerdem wird hier vor dem Parameter kein `inout` vorangestellt. Man könnte das zwar tun, aber sinnvoller wäre es hier nur `in` vor dem Parameter zu schreiben, um noch mal anzudeuten, das nur ein Parameter entgegengenommen wird. In Zeile 12 wird dann der Rückgabewert der Variablen `a` zugewiesen.
Ausgabe:
a = 20.25

Listing 108: swap

```
1  import std.stdio;
2
3  template TSwap(T) {
4      void swap( inout T a, inout T b) {
5          T tmp = a;
6          a = b;
7          b = tmp;
8      }
9  }
10
11 void main() {
12     int a= 4,b=5;
13     double c= 5.5,d=6.7;
14     alias TSwap!(int).swap swap_int;
15     alias TSwap!(double).swap swap_double;
16     swap_int(a,b);
17     writefln("b = %d",b);
18     swap_double(c,d);
19     writefln("c = %f",c);
20 }
```

Ausgabe:
b = 4
c = 6.700000

Hier noch ein letztes Beispiel bei der der swap Funktion 2 Parameter übergeben werden. Mit `alias` wird dem Ausdruck `TSwap!(int).swap` einfach ein neuen verkürzten Name gegeben, und man kann die Funktion wie in Zeile 16 verwenden.

10.1.1 Template Parameter

Listing 109: swap2

```
1  import std.stdio;
2
3  template TSwap(T,int max) {
4      void swap( inout T a, inout T b) {
5          T tmp = a;
6          a = b;
7          b = tmp;
8          writefln("max = %d",max);
9      }
10 }
11
12 void main() {
13     int a= 4;
14     int b= 5;
15     alias TSwap!(int,3).swap swap;
16     swap(a,b);
17     writefln("b = %d",b);
18 }
```

Man kann auch dem Template direkt ein Wert mit übergeben, in unserem Beispiel ist das `max`. Der Wert 3 für max wird in Zeile 15 mit übergeben.

Ausgabe:
max = 3
b = 4

10.1.2 Template mit 2 Parameter

Listing 110: See

```
1  import std.stdio;
2
3  template TSee(T,U) {
4      void see( T a, U b) {
5          writefln("a = ",a);
6          writefln("b = ",b);
7      }
8  }
9
10 void main() {
11     int a= 4;
12     string str="Hallo";
13     alias TSee!(int,string).see see;
14     see(3,"Weil Einfach einfach Einfach ist!");
15 }
```

Ausgabe: a = 3
b = Weil Einfach einfach Einfach ist!
Hier haben wir nun 2 Template Parameter in Zeile 11. Nun müssen auch 2 Typen mit übergeben werden, wie es in Zeile 13 geschieht.

10.1.3 Template Struct

Listing 111: Struct Template

```
1  import std.stdio;
2
3  struct foo(T) {
4      T x;
5      void getsize() {
6          writefln("%d", x.sizeof);
7      }
8  }
9
10 void main(char[][] args) {
11
12     foo!(int) y;
13     y.getsize();
14
15     int i;
16     writefln("%d",i.sizeof);
17
18     foo!(creal) r;
19     r.getsize();
20     writefln("%d", r.sizeof);
21 }
```

Ausgabe:
4
4
24
24

Hier wird das Struct Template (Zeile 3) nicht mit dem Schlüsselwort `template`, sondern mit `struct` eingeleitet. Als Platzhalter für den Typ wird wieder `T` verwendet. Die Variable `x` bekommt den Typ vom Platzhalter zugewiesen, der ja erst bei der Initialisierung festgelegt wird. In Zeile 5 wird die Funktion `getsize` wie auch im Programm 8 deklariert. Diese Funktion gibt die Größe in Byte der Variablen `x` aus. Dies geschieht wiederum mit der Funktion `sizeof`. In Zeile 12 und 18 wird das Struct Template mit `int` und `creal` initialisiert. Die Funktionen `getsize` werden jeweils in Zeile 13 und 14 mit der Punktnotation aufgerufen. Man sieht hier das die Variable `x` einmal vom Typ `int` ist und nur 4 Byte groß und einmal vom Type `creal` ist und 24 Byte groß ist.

10.1.4 Template Klassen

Listing 112: Klassen Template

```
1  import std.stdio;
2
3  class Foo(T) {
4      public T x = 0.5;
5  }
6
7  void main() {
8      Foo!(double) thing = new Foo!(double);
9      writefln(thing.x);
10     writefln("%s",thing.classinfo.name);
11 }
```

Ausgabe:
0.5
Thing

Tja, was es in D alles gibt ist schon erstaunlich. Hier kann mal also der Klasse `Foo` ein Template Parameter mit übergeben.

Listing 113: Template

```
1  template SortTemplate(Type)
2  {
3    void insertionSort(Type array[], bit delegate(Type a,Type b) lessThan
       )
4    {
5      for (int i=1; i < array.length; i++)
6      {
7        Type index = array[i];
8        int j = i;
9        while ((j > 0) && lessThan(index,array[j-1]))
10       {
```

```
11          array[j] = array[j-1];
12          j = j - 1;
13        }
14        array[j] = index;
15      }
16    }
17  }
18
19  class Person
20  {
21    this(string n, int a)
22    {
23      name = n;
24      age = a;
25    }
26
27    string name;
28    int    age;
29  }
30
31  alias SortTemplate!(Person).insertionSort PersonSort;
32  //SortTemplate!(Person)
33
34  void PrintByName(Person[] people)
35  {
36    PersonSort(people,
37          delegate bit(Person a, Person b) { return a.name < b.name; } );
38
39    printf("\nPeople sorted by name\n==========\n");
40    for(int i = 0; i < people.length; ++i)
41      printf("%.*s,%d\n",people[i].name,people[i].age);
42  }
43
44  void PrintByAge(Person[] people)
45  {
46    PersonSort(people,
47          delegate bit(Person a, Person b) { return a.age < b.age; } );
48
49    printf("\nPeople by sorted by age\n==========\n");
50    for(int i = 0; i < people.length; ++i)
51      printf("%.*s,%d\n",people[i].name,people[i].age);
52  }
53
54  void main(char[][] argv)
55  {
56    Person people[];
57
58    people ~= new Person("Zeb",27);
59    people ~= new Person("Betty",22);
```

```
60      people ~= new Person("Casy",15);
61      people ~= new Person("Mike",50);
62      people ~= new Person("Stephan",42);
63      people ~= new Person("Linda",12);
64      people ~= new Person("Freda",33);
65      people ~= new Person("Rudy",45);
66      people ~= new Person("Holly",18);
67
68      PrintByName(people);
69      PrintByAge(people);
70  }
```

```
People sorted by name
=====================
Betty,22
Casy,15
Freda,33
Holly,18
Linda,12
Mike,50
Rudy,45
Stephan,42
Zeb,27

People sorted by age
====================
Linda,12
Casy,15
Holly,18
Betty,22
Zeb,27
Freda,33
Stephan,42
Rudy,45
Mike,50
```

10.2 mixin

Mit `mixin` werden vom Template die Deklaration praktisch an der Stelle eingefügt bei das Schlüsselwort `mixin` auftritt. Die Templatedeklaration wird praktisch in dem Bereich rein gemixt. Die mit `mixin` eingefügte Templatedeklaration hat ihren eigenen Namensbereich, bzw. Variablen mit dem gleichen Namen werden überschrieben. Dies wird an dem unteren Beispiel deutlich.

Listing 114: Mixin

```
1  private import std.stdio;
2
3  int x = 6;
4  template Foo() {
5      int x = 4;
6  }
7
8  void main() {
9      {
10         int x = 5;
11         {
12             mixin Foo;
13             writefln("x = %d",x);
14         }
15         writefln("x = %d",x);
16     }
17     writefln("x = %d",x);
18 }
```

Ausgabe:

```
x = 4;
x = 5;
x = 6;
```

In diesem Beispiel haben wir drei Namensbereiche. Der erste beginnt mit der Klammer in Zeile 8 und endet in Zeile 18, der zweite geht von Zeile 9 und endet in Zeile 16, der letzte Namensbereich geht von Zeile 11 bis Zeile 14. In Zeile 12 wird jetzt die Templatedeklaration mit `mixin Foo` eingefügt. Also wird praktisch `int x = 4` in Zeile 12 hineinkopiert. Es wird dann auch `x = 4` ausgegeben. Mit Zeile 15 wird `x = 5` ausgegeben, weil in Zeile 10 die Zuweisung für `x` stattfindet. Als letzten wird `x = 6` ausgegeben. Das ist die globale Variable, die in Zeile 3 initialisiert wurde. Am besten man spielt mal mit dem Programm ein bisschen rum, und kommentiert Zeile 12 aus, oder die Klammern in den Zeilen 11 und 14.

Listing 115: Mixin1

```
1  import std.stdio;
2
3  int y = 3;
4
5  template Foo() {
6      int abc() {
7          return y;
8      }
9  }
10
11 void main() {
12     int y = 8;
13     mixin Foo;
14     writefln("abc(): ", abc());
15     writefln("Foo!().abc(): ", Foo!().abc());
16 }
```

Ausgabe:

```
abc(): 8
Foo!().abc(): 3
```

Hier wird die Funktion `abc()` vom Template `Foo` in Zeile 14 aufgerufen. Dies wurde erst möglich, weil das TemplateMixin in Zeile 13 eingebunden wurde. Da die globale Variable `y = 3` in Zeile 3, wird in Zeile 12 überschrieben, deshalb wird zuerst 8 ausgegeben. Ruft man die Templatefunktion direkt auf, wie in Zeile 15 wird wieder die globale Variable `y = 3` ausgegeben.

Man kann beim TemplateMixin auch ein Parameter übergeben. Das Programm `mixin` 114 in der abgeänderten Form.

Listing 116: Mixin mit Parameter

```
1  private import std.stdio;
2
3  int x = 6;
4  template Foo(T) {
5      T x = 4;
6  }
7
8  void main() {
9      {
10         int x = 5;
11         {
12             mixin Foo!(int);
13             writefln("x = %d",x);
14         }
15         writefln("x = %d",x);
16     }
17     writefln("x = %d",x);
18 }
```

Ausgabe:

```
x = 4;
x = 5;
x = 6;
```

Nur die Zeilen 4,5 und 12 haben sich gegenüber 114 geändert. Es wurde ein Parameter `T` hinzu gefügt. Beim TemplateMixin wird dann der Integraltyp `int` mit übergeben.

10.2.1 Mixin bei Klassen

Ein TemplateMixin kann in Modulen, Klassen, Structs und Unions verwendet werden.
Hier ein Beispiel für Klassen.

Listing 117: Mixin mit Klassen

```
1  import std.stdio;
2
3  class Bar {
4      void func() {
5          writefln("Foo.func()");
6      }
7  }
8
9  class Code : Bar {
10
11     void func() {
12         writefln("Code.func()");
13     }
14     void func1() {
15         writefln("Code.func1()");
16     }
17 }
18
19 void main() {
20     Bar b = new Bar();
21     b.func();
22
23     b = new Code();
24     b.func();
25     (cast(Code)b).func1();
26 }
```

Ausgabe:

```
Foo.func()
Code.func()
Code.func1()
```

In der Klasse `Bar` wird die Funktion `func()` vom Template `Foo` eingebunden. Die Klasse `Code` erbt alle Methoden von der Klasse `Bar`. Also auch die Methode `func()`. Nun ist `b` eine Objektvariable vom Typ Bar. Die Objektvariable wird als Referenz gespeichert. Nun wird in Zeile 23 eine neue Instanz der Klasse `Code` erstellt, die auf `b` zeigt. Da die Objektvariable `b` auf `Bar` referenziert, kennt nun die Instanz die Methode `func1()` nicht. Um jetzt trotzdem die Methode `func1()` aufrufen zu können, muss die Referenz auf ein Objekt von `Code` zeigen, deshalb muss wie in Zeile 25 gecastet werden.

10.3 IFTI

`IFTI` steht für „Implicit Function Template Instantiation“. Hier wird das Schlüsselwort template nicht mehr benötigt und der Aufruf von “templateName!(int).funktionsName(parameter) „entfällt hier. Das ganze läßt sich nun wesentlich eleganter und intuitiver schreiben.

10.3.1 IFTI Template

Listing 118: IFTI Template

```
1  import std.stdio : writefln,writef;
2
3  T func(T)(T val) {
4      return val;
5  }
6
7
8  void main() {
9      string str = "Hali ";
10     writef(func(str));
11     writefln(func("hallo"[]));
12 }
```

Ausgabe:
Hali hallo

In Zeile 1 werden nur die beiden Funktionen `writefln` und `writef` importiert. Wichtig ist hier noch Zeile 11, weil `("hallo"[])` mit den rechteckigen Klammern übergeben werden muss. Dies ist notwendig, weil der Compiler sonst den Parameter als statisches Array annimmt. Mit den Klammern wird es als dynamisches Array übergeben. In D können keine statischen Typen übergeben werden, nur Integral Typen, wie int,void,char usw.
Nun kann man die Funktion `func` in Zeile 10 wie eine ganz „normale“Funktion aufrufen, was ich persönlich ziemlich genial finde.

10.3.2 IFTI Template mit 2 Parameter

Listing 119: IFTI Template

```
1  import std.stdio : writefln;
2
3  T func(T,T1)(T val,T1 val1) {
4      writefln(val1);
5      return val;
6  }
7
8
9  void main() {
10     string str = "Hallo ";
11     string ret = func(str,4.35);
12     writefln(ret);
13 }
```

Ausgabe:
4.35 Hallo
In Zeile 11 wird das Template `func` nun mit 2 Parameter aufgerufen, und die Typen werden vom Compiler selbst ermittelt.

11 printf Funktion

Der printf Befehl ist in object.d definiert als:
`extern (C) int printf(char *, ...)`
extern (C) bedeutet das es sich um eine externe C Funktion handelt.
Die C Funktion erwartet das jeder String mit `\0` (null Byte Zeichen) terminiert ist. Mit char * erwartet die printf Funktion die Anfangsadresse des Strings. Die **...** bedeuten, das man beliebig viele Argumente der Funktion printf übergeben kann.Z.B.:

```
printf("%.*s %.*s","Argument 1","Argument 2")
```

Die Anzahl, wie viele Argumente übergeben werden, muss der Funktion mitgeteilt werden, dies geschieht bei printf, in dem die Anzahl der Konvertierungsspezifizierer im Formatstring ausgelesen wird, also hier 2 mal `%.*s` .
Bei der Anweisung:

Mit
`char[] hello = "Hello"`
wird ein konstanter String der Variable hello zugewiesen und `\0` wird an das Stringende automatisch angehängt. In D wird die Länge des Strings mit gespeichert, so das die Strings nicht null terminiert sein müssen. Mit `%.s` wird D mittgeteilt solange auszugeben wie die Länge des Strings ist, **oder** bis das `\0` Zeichen erscheint. In C wird eben nur so lange ausgegeben bis das `\0` Zeichen kommt. C kennt die Länge des Strings nicht.

Die printf Funktion erwartet aber ein Zeiger vom Typ char. Der Compiler führt automatisch ein cast durch. printf(hello) könnte man auch so schreiben:
printf((char*)hello);
Oder auch folgendes:
`printf(&hello[0]);`
Das & Zeichen ist der Addressoperator , hiermit wird noch mal deutlich das die erste Anfangsadresse des Strings der Funktion printf übergeben wird. Wenn man nicht bei 0 sondern beim zweiten Element die Adresse übergibt wird halt das H von *Hello* nicht mit ausgegeben. `printf(&hello[1]);`

`char[]` ist ein dynamisches Array. Das bedeutet das man nicht vorher wissen muss wie viele Zeichen man in das Array abspeichern möchte. Ich kann sowohl 1 oder beliebig viele Abspeichern (abhängig vom Arbeitsspeicher vom Rechner).
Bei statischen Array ist das anders hier wird die Anzahl der Elemente die in ein Array gespeichert werden soll vorher festgelegt. `char he[6];` Hier können also 5 Zeichen und das `\0` Zeichen gespeichert werden.

Datentypen	Speicherbedarf in Bit	Min	Max	Kommentar
void				Hat kein Typ
bit	1	0	1	
byte	8	-128	+128	
ubyte	8	0	255	Vorzeichenlos
short	16	-32768	32767	
ushort	16	0	65535	
int	32	-2147483648	2147483647	
uint	32	0	4294967295	
long	64	-9223372036854775808	9223372036854775807	
ulong	64	0	18446744073709551615	
cent	128			Reserviert
ucent	128			Reserviert
float	32	1.17549e-38	3.40282e+38	Fließkommazahlen
double	64	2.22507e-308f	1.79769e+308f	Fließkommazahl `%lf o. %lgf`
real	80	3.3621e-4932	1.18973e+4932	80 bit für Intel CPU's
ireal	80	3.3621e-4932	1.18973e+4932	
ifloat	32	1.47256e-4932	1.18973e+4932	
idouble	64	2.22507e-308	1.79769e+308	
cfloat	64	2.84809e-306	1.40445e+306	
cdouble				
creal	24			
char	8	0	255 (0xFF)	Vorzeichenlos 8 bit UTF-8
wchar	16	0	65535 (0xFFFF)	Vorzeichenlos 16 bit UTF-16
dchar	32	0	1114111 (0x0000FFFF)	Vorzeichenlos 32 bit UTF-32

Tabelle 4: Datentypen

12 Exception sichere Programmierung

In diesem Kapitel werden 3 verschiedene Möglichkeiten vorgestellt um Exception sichere Programmierung durchzuführen. Anfangen werden wir mit RAII (resource acquisition is initialization) *Ressourcenbelegung ist Initialisierung*. Anschließend werden wir uns das *try-catch-finally* Konzept vornehmen und als letztes die Lösung in D mit `scope` ansehen.

12.1 RAII

Mit RAII (Ressourcenbelegung ist Initialisierung) wird gesteuert wie Ressourcen belegt und wieder frei gegeben werden. Die Lebensdauer einer Resource ist an ein Objekt gebunden, sobald der Namensbereich des Objekts verlassen wird, wird der Destruktor aufgerufen und gibt die Resource frei. Das freigeben der Ressource geschieht mit einem Destruktor. Da es ein Destruktor in Java nicht gibt, ist es in der Programmiersprache auch nicht möglich das Konzept von RAII durch zu führen.

Am Beispiel eines Programms, das eine Datei öffnet, soll mit RAII sichergestellt werden, das Sie korrekt geschlossen wird, auch dann wenn ein Fehler beim Lesen auftritt. In der Praxis ist das z.B. besonders wichtig, wenn Datenbank-Verbindungen geschlossen werden müssen.

Listing 120: RAII

```
1  import std.stdio;
2  import std.c.process;
3  import std.c.time;
4  import std.stream;
5
6  class Datei
7  {
8      File file;
9      void openDatei()
10     {
11         writefln("Datei wird zum lesen geoeffnet.");
12         file = new File;
13         file.open("testdatei.txt",FileMode.In);
14         while (!file.eof()) {
15             writefln("%s",file.readLine());
16             sleep(1);
17             //file.close();
18         }
19     }
20
21     ~this()      // Destruktor
22     {
23         writefln("Datei wird geschlossen.");
24         file.close();
25     }
26 }
27
28 void main()
29 {
30     auto Datei datei = new Datei();
31     datei.openDatei;
32 }
```

Wichtig in diesem Programm sind die Zeilen 17 und die Zeile 30. Wenn Sie Zeile 17 einkommentieren, wird während des Lesens die Datei geschlossen und es kommt zu einer Fehlermeldung und der Destruktor wird aufgerufen. Das heißt durch diesen Fehler verlassen wir den Namensbereich des Objekts `datei`. Hier die Ausgabe des Programms mit einkommentierter Zeile 17:

```
Datei wird zum lesen geöffnet.
Zeile 3
Datei wird geschlossen.
Error: Stream is not readable
```

In Zeile 30 steht das Schlüsselwort `auto` vor der Deklaration des Objekts `datei`. Wenn Sie das nicht voranstellen bekommen Sie folgende Ausgabe:

```
Datei wird zum lesen geöffnet.
Zeile 3
Error: Stream is not readable
```

Und wie man sieht wird der Destruktor nicht mehr aufgerufen und damit wird auch unsere Datei nicht korrekt geschlossen. Dieses `auto` stellt also sicher,das der Destruktor aufgerufen wird, sobald die `auto reference` außerhalb des Bereichs geht.

12.2 try-catch-finally

Beim `try-catch-finally` bzw. `try-finally` wird **immer** nach der Abarbeitung des `try` Blocks, der `finally` Block ausgeführt. Hier das Beispiel für das öffnen und schließen einer Datei. Den catch Block hätte man auch weglassen können.

Listing 121: try-catch-finally

```
1  import std.stdio;
2  import std.c.process;
3  import std.c.time;
4  import std.stream;
5
6  void main()
7  {
8      File file = new File;
9      try
10     {
11         file.open("testdatei.txt",FileMode.In);
12         while (!file.eof()) {
13             sleep(1);
14             writefln("%s",file.readLine());
15             file.close();
16         }
17     }
18     catch (Exception e)
19     //catch (OpenException e)
20     {
21         writefln("%s",e.toString());
22     }
23         finally
24     {
25         writefln("finally");
26         file.close();
27     }
28 }
```

Ausgabe des Programms:

```
Zeile 3
Stream is not readable
finally
```

Durch das schließen der Datei in Zeile 15 wird eine Exception geworfen, die mit der `catch` Anweisung abgefangen wird. Die erstellt die Meldung `Stream is not readable` in Zeile 21. Zum Schluss wird der finally Block ausgeführt und das File wird geschlossen unabhängig davon ob eine Exception geworfen wurde oder nicht.

12.3 scope

Jetzt kommen wir zur dritten und wirklich neuen Möglichkeit Resource frei zu geben. Doch bevor wir auf das Beispiel mit der Datei kommen, müssen wir erst mal die unterschiedlichen scope Eigenschaften näher beleuchten.

12.3.1 scope(exit)

`scope(exit)` wird ausgeführt, wenn der Bereich also scope verlassen wird. Das heißt bei korrektem und inkorrektem verlassen des Bereichs wird scope ausgeführt.

Listing 122: scope(exit)

```
1  import std.stdio;
2
3  void main() {
4      abc();
5  }
6
7  void abc() {
8      //throw new Exception(std.string.format("paff"));
9      scope(exit) writefln("5");
10 }
```

Ausgabe:

```
5
```

Wird nun Zeile 8 einkommentiert, wird die 5 nicht mehr ausgegeben. Wenn das gewünscht ist, dann sollte `scope` vor dem `throw` stehen.

12.3.2 scope(failure)

`scope(failure)` wird nur ausgeführt, wenn ein Fehler auftritt, oder eine Exception geworfen wird.

Listing 123: scope(failure)

```
1  import std.stdio;
2
3  void main() {
4      abc();
5  }
6
7  void abc() {
8      scope(failure) writefln("5");
9      throw new Exception(std.string.format("paff"));
10 }
```

Ausgabe:

```
5
```

Hier wird die 5 ausgegeben, weil eine Exception geworfen wurde. Kommentiert man Zeile 9 aus, wird auch keine 5 mehr ausgegeben.

12.3.3 scope(success)

`scope(succses)` wird nur ausgeführt, wenn der Bereich korrekt verlassen wird.

Listing 124: scope(success)

```
1  import std.stdio;
2
3  void main() {
4      abc();
5  }
6
7  void abc() {
8      scope(success) writefln("5");
9      //throw new Exception(std.string.format("paff"));
10     writefln("Hallo");
11 }
```

Ausgabe:

```
5
Hallo
```

Wird nun Zeile 9 einkommentiert, wird `5` nicht mehr ausgegeben.

12.3.4 Dateibeispiel

Jetzt kommen wir zu dem Datei Beispiel:

Listing 125: scope

```
1  import std.stdio;
2  import std.c.process;
3  import std.c.time;
4  import std.stream;
5
6  void main()
7  {
8      auto file = new BufferedFile("testdatei.txt", FileMode.In );
9      scope(failure) {writefln("SCOPE"); file.close();}
10     while (!file.eof()) {
11         sleep(1);
12         writefln("%s",file.readLine());
13         file.close();
14     }
15     file.close();
16 }
```

Ausgabe:

```
Zeile 3
SCOPE
Error: Stream is not readable
```

Beim ersten Durchlauf der `while`-Schleife, wird die erste Zeile der `testdatei.txt` gelesen und ausgegeben. Anschließend wird die Datei geschlossen, deshalb kommt es beim zweiten Durchlauf zur Exception. Um die Exception aufzufangen muß `scope(failure)` **vor** der `while`-Schleife geschrieben werden.

Listing 126: scope2

```
1  import std.stdio;
2  import std.c.process;
3  import std.c.time;
4  import std.stream;
5
6  void main()
7  {
8      File file = new File;
9      file.open("testdatei.txt",FileMode.In);
10     scope(exit) {writefln("SCOPE"); file.close();}
11     while (!file.eof()) {
12         sleep(1);
13         writefln("%s",file.readLine());
14         //file.close();
15     }
16 }
```

Ausgabe:

```
Zeile 3
Zeile 3
SCOPE
```

Nach dem durchlaufen der `while` Schleife wird exit ausgeführt und die Datei wird korrekt geschlossen.

12.3.5 scope rekursiv

Listing 127: scope4

```
1  import std.stdio;
2
3  void main() {
4      scope(exit) {
5          writefln("Block wird ausgefuehrt");
6      }
7      scope(exit) writefln("1");
8  }
```

Ausgabe:

```
1
Block wird ausgeführt
```

Wenn mehrere `scope` Anweisungen auftreten, werden sie rückwärts ausgeführt. Deshalb wird zuerst 1 und dann der Block ausgeführt.

Teil V.
GUI Programmierung

`GUI` steht für `Graphical User Interface` also für grafische Benutzeroberflächen. Schätzungsweise gibt es ein halbes Dutzend GUI's für D, hier die wichtigsten

- DWT
- DUI
- Harmonia
- DFL
- wxd

Zurzeit läuft nur `wxd` unter Windows, Unix/Linux und Mac OS X. Für `DFL` gibt es den GUI-Designer `Entice` zum erstellen der GUI's, es läuft aber nur unter Windows. Die GUI Programmierung läuft Event gesteuert ab, das heisst das ganze Programm läuft in einer „Haupt-Schleife "(Main Loop) und wartet auf ein Ereignis (Event), z.b. wenn mit der Maus auf ein Button gedrückt wird. Dann wird zur entsprechenden Methode gesprungen und ausgeführt. Anschliessend befindet man „sich "wieder in der Haupt-Schleife. Das Programm wird also nicht von oben nach unten abgearbeitet, oder springt von Methode zu Methode, sondern die Event's bestimmen den Programmablauf.

13 wxd

wxd baut auf die wxWidgets auf. wxWidgets wurde von Anfang an Plattformunabhängig erstellt. Deshalb läuft es auch unter so vielen verschiedenen Betriebsystemen.

13.1 Installation von wxd

Für die Installation hält man sich am besten an die Anweisungen auf der Homepage, die man unter [21] findet. Ich möchte hier trotzdem noch mal erläutern wie ich wxd unter Linux installiert habe. Als erstes sollte man genau drauf achten, welche wxWidget Version auf der Homepage empfohlen wird. Zur Zeit ist es `wxWidgets-2.6.3`. Ich habe also `wxGTK-2.6.3.tar.bz2` runter geladen und entpackt. Anschliessend `./configure --disable-shared`, `make` und `make install` ausführen. Den letzten Befehl unter `root` oder bei `Ubuntu` mit `sudo` ausführen. Die ganzen kompilierten `wxWidget` Dateien befinden sich dann unter `/usr/local/lib`.
Nun geht es daran wxd zu entpacken und mit make zu installieren. Da ich zu dem Zeitpunkt den GDC Compiler verwendet habe, musst ich noch in der Datei `GNUmakefile` den Comiler auf GDC ändern. Anschliessend `make -f GNUmakefile` ausführen. Möchte man noch die Beispieldateien, die in wxd unter Samples liegen auch kompilieren, dann bitte noch `make test` ausführen. Bei mir klappte das leider nicht auf anhieb, da die `libstd++` nicht gefunden wurde. Unter `usrlib` hab ich dann folgenden Link angelegt: `sudo ln -s libstdc++.so.6.0.8 libstdc++.so`. Nicht schön aber selten ☺. Nun liegen in `wxd` unter `bin` die Beispieldateien.

13.1.1 Erstes Programm

Um das erste Programm zu erstellen ist es am einfachsten unter `wxd/Samples` das `Hello` Verzeichnis zu kopieren und um zu benennen. In `GNUmakefile` oder `Makefile` die Variable `NAME=Hello` in `NAME=First` ändern. Nun erstellen wir das erste Programm.

Listing 128: First Wx

```
1  import wx.wx;
2  import std.stdio;
3
4  public class HelloWorldApp : wxApp
5  {
6      public override bool OnInit()
7      {
8              wxFrame frame = new wxFrame(null, wxID_ANY, "Hello
                   wxWidgets World");
9              frame.Show(true);
10             TopWindow = frame;
11             return true;
12     }
13 }
14
15 int main()
16 {
17     HelloWorldApp app = new HelloWorldApp();
18     app.Run();
19     return 0;
20 }
```

Abbildung 2: Erstes Widget

Kompiliert wird das Programm mit `make -f GNUmakefile` oder `make -f Makefile` je nach dem ob man den gdc oder dmd Compiler verwendet. In Zeile 1 wird das Modul `wx` importiert, es liegt in `wxd/wx/wx.d`. Hier sollten Sie mal nach `wxID_ANY` greppen, es wird in der Datei `Defs.d` auf `-1` gesetzt. Dieser Wert wird in Zeile 8 benötigt. Die Widgets haben alle eine eigene Nummer oder ID. Mit dem Wert `-1` wird irgendeine beliebige Nummer vergeben. Aber ich möchte noch auf die

Klasse `HelloWorldApp` zurückkommen, denn Sie erbt die Eigenschaften von `wxApp`, sie wird nur einmal instanziert. Kommen wir jetzt zur Zeile 6, Hier wird die Methode `OnInit` mit dem Attribute `override` aufgerufen. Diese Methode ist der Einstiegspunkt in unsere Applikation, sozusagen die `main` Funktion. Diese Methode darf es nur einmal geben. Nun zur Zeile 8, hier wird eine Instanz der Klasse wxFrame erzeugt. Das `null` gibt die Vaterklasse an, da wir hier keine Vaterklasse haben, wird dem Konstruktor `null` übergeben. `Hello wxWidget World` wird dem Frame die Überschrift übergeben. Mit `frame.Show(true)` wird das Frame angezeigt. Mit Zeile 10 wird das Frame direkt auf dem Desktop gezeichnet. In Zeile 17 wird die Klasse `HelloWorldApp` instanziert. Intressant ist Zeile 18, hier wird mit der Methode `Run()` die Ereignisschleife (Event Loop) gestartet, und läuft so lange bis das `frame` kein `true` zurück liefert, also im unserem Beispiel für immer.

13.1.2 StatusBar

Listing 129: First1 Wx

```
1  import wx.wx;
2
3  public class HelloWorldApp : wxApp
4  {
5      public override bool OnInit()
6      {
7          wxFrame frame = new wxFrame(null, wxID_ANY, "Hello wxWidgets
               World");
8          frame.CreateStatusBar();
9          frame.StatusText = "Status Bar";
10         frame.Show(true);
11         TopWindow = frame;
12         return true;
13     }
14 }
15
16 int main()
17 {
18     HelloWorldApp app = new HelloWorldApp();
19     app.Run();
20     return 0;
21 }
```

Hier wurde das Frame um die StatusBar mit den Zeilen 8 und 9 ergänzt.

13.1.3 Menu

Abbildung 3: Frame mit StatusBar

Literaturverzeichnis

[1] http://www.digitalmars.com/d/

[2] http://www.steinmole.de/d/beispiele.tgz

[3] http://www.digitalmars.com/drn-bin/wwwnews?newsgroups=*

[4] http://www.sprungmarke.net

[5] http://home.earthlink.net/~dvdfrdmn/d/

[6] http://dgcc.sourceforge.net/

[7] http://gcc.gnu.org

[8] http://gcc.gnu.org/install/

[9] http://www.torsten-horn.de/techdocs/ascii.htm

[10] http://www.cl.cam.ac.uk/ mgk25/unicode.html

[11] http://www.allegro-c.de/unicode/

[12] http://www.opend.org

[13] http://www.vim.org/scripts/script.php?script_id=379

[14] http://www.steinmole.de/d

[15] http://www.99-bottles-of-beer.net/language-d-1212.html

[16] http://www.bigupload.com/d=F902F09C

[17] http://www.steinmole.de/d/arm-wince-pe-gdc.tar.bz2

[18] http://www.symbolictools.de/public/pocketconsole/download.htm

[19] http://www.symbolictools.de/public/pocketconsole/applications/PocketCMD/index.htm

[20] http://www.microsoft.com/downloads/details.aspx?
FamilyID=b5a8b224-6604-42fa-8777-faa1f70e99e5&displaylang=debitem

[21] http://wxd.sourceforge.net

Listings

Sachregister