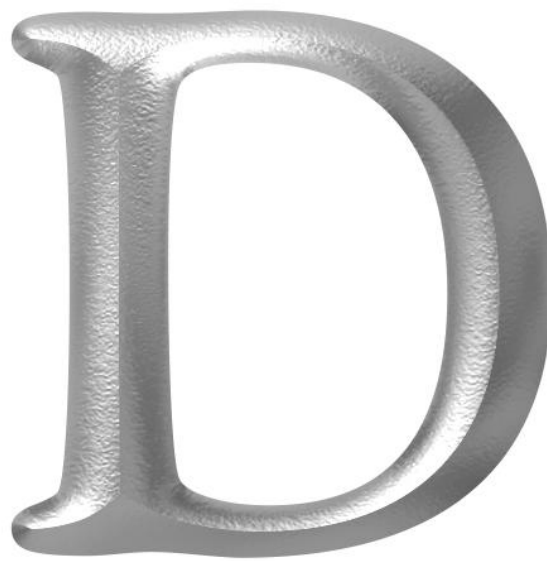


Programmiersprache



Autor: Manfred Hansen
Mail: m.hansen@kielnet.net
Erstelldatum: 16. Oktober 2003
Letzte Änderung: 11. Januar 2007

Copyright © Manfred Hansen 2007

Inhaltsverzeichnis

Inhaltsverzeichnis	2
1. Einleitung	6
1.1. Entwicklung der Sprache	6
2. Erstes Programm	7
2.1. Installation von dmd unter Linux	7
2.2. Installation von dmd unter Windows	7
2.3. Installation vom gdc unter Linux	7
2.3.1. Gdc selber kompilieren	8
2.4. Hello World	10
2.5. Compiler	10
2.6. gdc	11
2.7. gdmmd	11
2.8. strip	11
2.9. vim	12
2.10. libphobos.a	12
2.11. Windows Mobile	13
2.11.1. Crosscompiling	14
2.11.2. PocketConsole	14
2.11.3. Erstes Programm	15
I. Einführung	16
3. Grundlagen	16
3.1. Kommentare	16
3.2. Steuerzeichen	17
3.3. Datentypen	18
3.3.1. char	18
3.3.2. wchar	19
3.3.3. dchar	19
3.3.4. Formatspezifizierer	22
3.3.5. union und struct	23
3.4. Konstanten	27
3.5. Variablen	27
3.5.1. Gültigkeitsbereich von Variablen	28
3.5.2. auto	28
3.5.3. static	29
3.5.4. extern	30
3.6. Operatoren	31
3.7. Bitoperatoren	33
3.8. Schleifen	36
3.8.1. for-Schleife	36

3.8.2.	while Schleife	37
3.8.3.	do-while Schleife	37
3.8.4.	foreach Schleife	38
3.8.5.	break	39
3.9.	Verzweigungen	40
3.9.1.	if Anweisung	40
3.9.2.	switch Anweisung	41
3.9.3.	tenärer Operator	42
3.10.	Arrays	43
3.10.1.	statische Arrays	43
3.10.2.	3-dimensionales statisches Array	45
3.10.3.	dynamische Arrays	45
3.10.4.	3-dimensionales dynamisches Array	47
3.10.5.	Matrix	48
3.10.6.	assoziative Arrays	49
3.10.7.	rechteckige Arrays	50
3.11.	Funktionen	51
3.11.1.	Einfache Funktion	51
3.11.2.	Variable Argumentenliste	51
3.11.3.	Funktion mit inout	53
3.11.4.	Funktion mit out	53
3.11.5.	Geschachtelte Funktionen	55
3.12.	Daten Konvertieren	55
3.13.	unittest	56
3.13.1.	assert	56
4.	Weitere Programmierung	58
4.1.	pragma	58
4.2.	align	59
II.	Standard Bibliothek phobos	60
5.	Ein und Ausgabe	60
5.1.	Dateien mit Stream bearbeiten	60
5.1.1.	Datei Zeilenweise lesen	60
5.1.2.	In Datei schreiben	61
5.1.3.	Anhängendes Schreiben	61
5.1.4.	Datei mit Fehlerbehandlung	62
5.1.5.	Datei in Puffer einlesen	64
5.2.	Dateien mit File bearbeiten	64
5.2.1.	Datei einlesen und ausgeben	65
5.2.2.	Datei und Verzeichnis Operationen	66

6. Strings	68
6.1. String manipulation	68
6.1.1. Substring	68
6.1.2. Strings splitten	69
6.2. Reguläre Ausdrücke	69
7. Sonstige Funktionen	70
7.1. Random	70
 III. Objektorientierte Programmierung	 72
8. Objekt und Klassen	72
8.1. Methoden	73
8.2. Methoden überladen	74
8.3. Variablen	74
8.3.1. this	75
8.4. Konstruktor	76
8.4.1. Static Konstruktor	78
8.5. Destruktor	78
8.5.1. Object auf null prüfen	79
8.6. Vererbung von Klassen	80
8.7. Alles Super	81
8.7.1. Super Konstruktoren	82
8.8. abstract	82
8.8.1. Interfaces	83
8.9. Protection Attribute	86
8.9.1. package	88
8.9.2. const	89
8.9.3. final	90
8.9.4. static	90
8.9.5. override	91
 IV. Fortgeschrittene Programmierung	 93
9. Boxing	93
9.1. Box mit Hash	94
10. Sonstige Funktionen	94
10.1. shared Libraries unter Linux	94
10.2. delegate	95
10.2.1. delegate mit Funktionen	96
10.2.2. delegate mit Methoden	98
10.2.3. delegate mit struct	98

11. Template	100
11.1. Template Funktionen	100
11.1.1. Template Parameter	102
11.1.2. Template mit 2 Parameter	103
11.1.3. Template Struct	103
11.1.4. Template Klassen	104
11.2. mixin	106
11.2.1. Mixin bei Klassen	109
12.printf Funktion	111
13.Exception sichere Programmierung	113
13.1. RAII	113
13.2. try-catch-finally	115
13.3. scope	116
13.3.1. scope(exit)	116
13.3.2. scope(failure)	116
13.3.3. scope(success)	117
13.3.4. Dateibeispiel	117
13.3.5. scope rekursiv	118
V. GUI Programmierung	119
14.wxd	119
14.1. Installation von wxd	119
14.1.1. Erstes Programm	120
14.1.2. StatusBar	121
14.1.3. Menu	121
Literaturverzeichnis	123
Sachregister	127
Sachregister	127

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 Speicherreservierung wird von einem **Garbage Collector** übernommen, so dass keine **Buffer Overflows** mehr auftreten können. Der **Garbage Collector** kann auch ausgeschaltet werden, was bei Echtzeitanwendungen erforderlich ist. Desweiteren unterstützt D eine Reihe von Features wie Templates, unittests.

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. Desweiteren schreibe ich dieses Buch, um mich intensiver mit D auseinanderzusetzen und weil es leider noch keine deutschsprachige Dokumentation gibt.

Ich werde in diesem Buch viele Programmbeispiele bringen, weil wie ich finde, dass 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. Desweiteren 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 geladen werden.

2.1 Installation von dmd unter Linux

Die Installation ist denkbar einfach, hierzu wird der D Compiler von <ftp://ftp.digitalmars.com/dmd.zip> herunter geladen.

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 neue Library gefunden wird.

Als letztes noch die `dmd.conf` nach `/etc` kopieren. Mit einem Editor `vim`, `kate`, `nano`, usw. den Pfad der Sourcen 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 linkern.

2.3 Installation vom gdc unter Linux

Das **D** Frontend wird von David Friedman 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 -xvf dc-0.21-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 Installationsanleitung 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-0.21.tar.gz` entpacken. Den GNU Compiler Collection von [7] herunterladen. Der gcc wird jetzt im Verzeichnis D entpackt, mit `tar -xvjf gcc-4.0.2.tar.bz2`, hier wird das Frontend nach `~/D/gcc-4.0.2/gcc` kopiert und entpackt. Nun gibt es ein neues Verzeichnis d.

Jetzt geht es ans patchen, hierzu wechseln Sie ins Verzeichnis `~/D/gcc-4.0.2/` 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-4.0.2/configure --prefix=/usr/local/gcc-4.0.2 --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-4.0.2/gcc/phobos` wechseln z.B. und mit `./configure --prefix=/usr/local/gcc-4.0.2` konfigurieren dann `make` und `make install` ausführen. Als letztes kann man noch die Umgebungsvariable `PATH` um `/usr/local/gcc-4.0.2/bin/` erweitern, damit der Compiler gdc finden kann. Das wars. 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-4.0.2.tar.bz2 /D`
5. `cd ~/D`
6. `tar -xjf gcc-4.0.2.tar.bz2`
7. `cp gdc-0.21.tar.gz gcc-4.0.2/gcc/`

8. `cd gcc-4.0.2/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 `sources` 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, kompiliert und gelinkt. Ein Preprocessor läuft 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 Dateierdung `.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 hessen 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

Desweiteren 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 Objectfile 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 $(D_FLAGS) 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 Crosscompiling

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
```

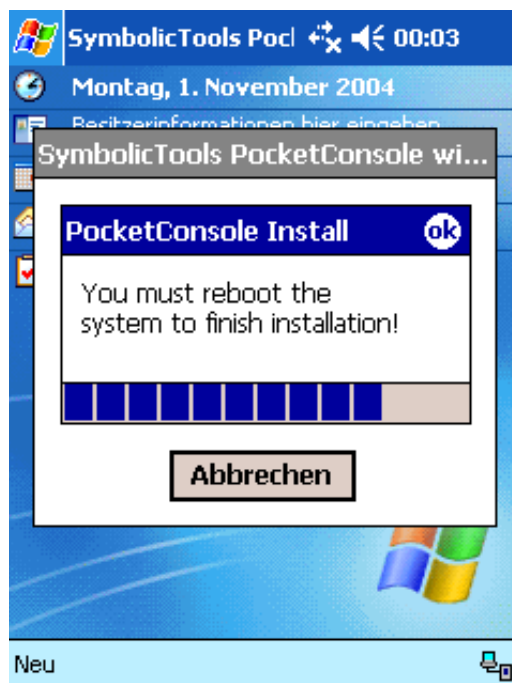
Eventuell ist es noch nötig in `usr/local` folgenden Befehl auszuführen:

```
chmod -R a=rwx arm-wince-pe
```

Hiermit werden für alle Lese.- Schreib.- und Ausführungsrechte erteilt.

2.11.2 PocketConsole

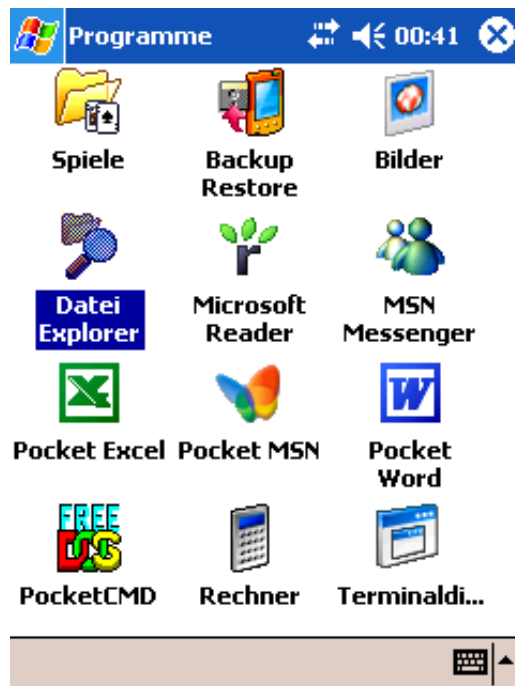
Jetzt soll der MSDOS Eingabeprompt auf dem PDA installiert werden. Bei WM5 ist es notwendig die `HKEY_LOCAL_MACHINE\Drivers\Console\OutputTo` auf 0 zu setzen. Als erstes sollten Sie auf Ihrem Windows Rechner ActiveSync installieren. Den kann man unter [20] herunterladen. PDA mit dem Windows Rechner verbinden. 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 `writeln`? Das liegt daran, dass 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	carriage Return Cursor 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
\\	Zeichen \ 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 Komplexe Zahlen und mehrere Datentypen für Zeichen. Die Tabelle auf Seite 112 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;

```

main

```

13     printf("ubyte\tdown: %d\tdown: %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     char[] 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 `wprintf` Funktion.

Spezifizierer	Bedeutung	Typen
<code>%c</code>	Ein einzelnes Charakter Zeichen	char
<code>%d</code>	Für decimal Zeichen	int, uint, short, ushort, bit, byte
<code>%e</code>	Exponentialschreibweise	long, ulong
<code>%f %F</code>	Fliesskommazahlen	float, ifloat, ireal, double, idouble, real
<code>%g %G</code>	Exponentialschreibweise	long, ulong
<code>%lld</code>	Grosse Dezimalzahlen	long
<code>%llu</code>	Grosse Vorzeichenlose Dezimalzahlen	ulong
<code>llx</code>	Denn Sinn noch nicht verstanden	long ulong
<code>%p</code>	Gibt die Adresse aus	gilt für alle
<code>%s</code>	String der null <code>'\0'</code> terminiert ist.	char arrays
<code>%. *s</code>	String der nicht null terminiert ist.	char arrays
<code>%u</code>	Vorzeichenlose Dezimalzahlen	uint, ushort, bit
<code>%x %X</code>	Hexadezimale schreibweise	char
<code>%</code>	Prozentzeichen wird 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     char[] 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 Speicherplatz gespart wird. Der Speicherplatz für die Variablen wird im Struct hintereinander abgelegt, 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         char [] 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         char [] 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 Variablen gleichzeitig initialisiert werden wie man in Zeile 10 sieht. In Zeile 15 wird der Speicherbedarf des Struct ausgegeben.

Listing 8: Struct

```
1 import std.stdio;
2
3 struct Person {
4     char[] anrede;
5     int alter;
6     int hausnummer;
7     void write() {
8         writeln("Cool, eine Funktion innerhalb vom struct");
9     }
10 }
11
12 int main(char [][] args) {
13     Person person;
14     person.anrede="Herr";
15     person.alter=54;
16     person.hausnummer=23;
17     writeln("Anrede: %s", person.anrede);
18     writeln("Alter: %s", person.alter);
19     writeln("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 13 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         char[] 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` initialisiert 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     writeln("i = %d", i);
7     writeln("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 Arbeitsspeicher reservieren. Hier unterscheidet man das initialisieren und das deklarieren von Variablen. Bei der Deklaration wird der Arbeitsspeicher für die Variable reserviert. Der Wert der Variablen ist jetzt noch unbekannt, er kann 0 sein, kann aber auch einen anderen Wert besitzen.

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 initialisiert
6     int b = 3; // deklaration 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 `int` handelt, werden 4 Byte im Arbeitsspeicher reserviert. Zeile 5 wird der Variablen `a` der Wert 3 zugewiesen, das geschieht mit dem `=` Operator, das nennt man auch initialisieren. Bei der Zeile 6 wird die Variable `b` gleichzeitig deklariert und initialisiert.

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 Variablenamen 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 Bestandteil 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 12: static und auto

```
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 a1 = new A;
19     A a2 = new A;
20     A a3 = new A;
21     A.klassenMethode();
22     writefln(" Es gibt %d Objekte.",A.OBJEKT);
23     //b.getName();
24     return 0;
25 }
```

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 13: Variablen übergabe 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 14: 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 Operatoren:

- 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 Zuweisungsoperator **=** das der rechte Teil vom **=** Zeichen ausgewertet wird und links dem Bezeichner also der Variablen zugewiesen wird.

Listing 15: 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 Bezeichner und beim Postfix hinter dem Bezeichner geschrieben. In Zeile 8 hat der Bezeichner *b* ein Postfixinkrement. Das Ergebnis wird in Zeile 9 und 10 Ausgegeben und ist 10 und 4. Hier wird also $b + 7$ ($b = 3$) addiert das Ergebnis 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	31
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	33
3	new	Instanz erzeugen	1	rechts nach links	
3	cast	Typ Umwandlung	1	rechts nach links	
4	* / %	Multiplikation, Division Modulo	2	links nach rechts	
5	+ -	Addition u. Subtraktion	2	links nach rechts	31
5		String Verkettung	2	links nach rechts	
6	<<	Schift Links	2	links nach rechts	34
6	>>	Schift Rechts	2	links nach rechts	34
6	>>>	Rechtsschift unsigned	2	links nach rechts	34
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	33
10	^	Logisches und Bitweises XOR	2	links nach rechts	33
11	—	Bitweises ODER	2	links nach rechts	33
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	34 34 46

Tabelle 3: Operatoren

3.7 Bitoperatoren

Listing 16: 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 17: 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  1  0  1  1  1  = 7
```

Nur wenn beide Bits eins sind, kommt es bei einer UND Verknüpfung zu 1. Die ODER Verknüpfung 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 ergebnis 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 18: 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 19: 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 20: 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 21: while Schleife

```
1 import std.stdio;
2
3 void main() {
4     int i = 13;
5
6     while (i > 10) {
7         i--;
8         writeln("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 13 initialisiert. Zeile 6 beginnt die while Schleife und endet in Zeile 9. Vor Eintritt in den Schleifenkörper wird die Bedingung ob `i > 10` geprüft. Am Anfang ist `i=13`, also wird die Schleife durchlaufen. Mit `i--` wird `i` um eins heruntergezählt, man nennt das auch dekrementieren.

3.8.3 do-while Schleife

Die do-while Schleife ist eine Annahmende 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 22: do-while Schleife

```
1 import std.stdio;
2
3 void main() {
4     int i = 0;
5
6     do {
7         i++;
8         writeln("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 23: foreach Schleife

```
1 import std.stdio;
2
3 void main() {
4
5     char[] 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 gedanke zu machen wie viele Zeichen das Wort **Hallo** hat. Die Schleife wird einfach solange ausgeführt bis das letzte Zeichen ausgegeben wurde.

Listing 24: 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 25: Schleifenabbruch mit break

```
1 import std.stdio;    // writeln
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         writeln("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 ternären Operator auch abhängig von der Bedingung verzweigt werden.

3.9.1 if Anweisung

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

Listing 26: if-else Anweisung

```
1 import std.stdio;
2
3 int main( char[][] arg ) {
4     if ( arg.length < 2 ) {
5         writeln("Es wurde kein Buchstabe eingegeben!");
6         return -1;
7     }
8
9     if ( arg[1] == "a" ) {
10        writeln("Buchstabe a");
11    } else if ( arg[1] == "b" ) {
12        writeln("Buchstabe b");
13    } else {
14        writeln("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 Hauptprogramm (main Funktion) auf Grund der `return` Anweisung mit `-1` sofort beendet. Bei erfolgreicher Ausführung des Programms wird eine 0 an das aufrufende Programm zurückgegeben. 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 {writeln("false");}
if (true) {writeln("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 27: 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 28: 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 werden 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, dass 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 29: Array deklaration

```

1  import std.stdio;
2
3  int main() {
4
5      static char[5][2] string = ["house","car"];
6      writeln("string[0] = %s",string[0]);
7      writeln("string[1] = %s\n",string[1]);
8
9      static int[3] a = [0:2,4,5]; // statische Array (geht noch nicht)
10     writeln("a[0] = %d",a[0]);
11     writeln("a[1] = %d",a[1]);
12     writeln("a[2] = %d",a[2]);
13     writeln("a laenge = %d\n",a.length);
14
15     enum color {red,green,blue};
16     static int[3] b = [color.red:3,7,9];
17     writeln("b[0] = %d",b[0]);
18     writeln("b[1] = %d",b[1]);
19     writeln("b[2] = %d",b[2]);
20
21     int[2] iarray;
22     iarray[0] = 3;
23     iarray[1] = 4;
24     writeln("iarray = %d",iarray[0]);
25     writeln("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 30: 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     writeln("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                 writeln("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 können $2 * 3 * 4 = 24$ 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 12.

Desweiteren sind dynamische Arrays Zeiger auf die eigentlichen Array Daten.

Listing 31: Array deklaration

```

1  import std.c.stdio;
2
3  int main() {
4
5      static char [][] 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 weist 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 32: Array deklaration

```
1  import std.c.stdio;
2
3  int main() {
4
5      char [][] 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 33: 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 34: 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 einem eindeutigen **key** gespeichert und auch über den key wieder gelöscht oder gesucht. In anderen Programmiersprachen werden Sie auch **hash** genannt.

Listing 35: Assoziative Arrays

```
1 import std.string;
2
3 int main() {
4     char [[ char []] hash;
5     hash["en"] = "Hello , world!";
6     hash["es"] = "Hola , mundo!";
7     hash["de"] = "Hallo Welt!";
8
9     foreach ( char [] 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 initialisiert. 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 36: Assoziative Array Funktionen

```
1 int main() {
2
3     int[char []] hash;
4     hash["en"] = 5;
5     hash["es"] = 3;
6     hash["de"] = 7;
7     hash["pl"] = 1;
8
9     foreach ( char [] 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 wird 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 Programmiersprache **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 37: Array deklaration

```
1 import std.c.stdio;
2
3 int main() {
4
5     char [][] 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 38: Einfache Funktion

```
1 import std.stdio;
2
3 void p(char [] x) {
4     writeln(x);
5 }
6
7 int main (char [][] args){
8     char [] 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 Argument vom Typ `char` 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 39: Variable Argumentenliste

```
1  import std.c.stdarg;
2
3  int foo(char [] 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 40: 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` weglässt, also nicht vor dem `int` wie in diesem Beispiel, wird der Funktionsparameter per default auf `in` gesetzt.

3.11.4 Funktion mit out

Listing 41: 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` initialisiert. Man nennt das auch **call by reference**. Es wird hier also nicht der Wert direkt an die Funktion übergeben, sondern nur eine Referenz. 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 42: 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 wird 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 bieten eine einfache Möglichkeit der Qualitätskontrolle.

Listing 43: Unittest

```
1 import std.stdio;
2
3 unittest {
4     writeln("Good mornig Vietnam");
5 }
6
7 int main() {
8     return 0;
9 }
```

Kompiliert wird das Programm mit dem Schalter **unittest**.

dmd -unittest unittest.d

Beim ausführen des Programms wird jetzt **Good morning Vietnam** ausgegeben. Ohne dem Schalter **unittest** wird nichts ausgegeben.

3.13.1 assert

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

Listing 44: Assert

```
1 import std.stdio;
2
3 int main() {
4     assert((4-3) == 1);
5     writeln("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: AssertionError Failure assert1.d(4)

Die Abarbeitung des Programms wird beendet. Der Ergebnistyp des assert Ausdrucks ist **void**.

Listing 45: 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.

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 Compilierens zu erstellen.

Listing 46: `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 47: 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     writeln("NodeLink:%d", NodeLink.sizeof);
27     writeln("NodeLink1:%d", NodeLink1.sizeof);
28     writeln("NodeLink2:%d", NodeLink2.sizeof);
29     writeln("NodeLink3:%d", NodeLink3.sizeof);
30 //The NodeLink.sizeof will be 5. Of course, if the layout of the struct
    doesn't
31 //have to be strict, it*d be recommended to use the default alignment
    for
32 //performance reasons; but if this is being used to paint a data
33 //structure over an arbitrary chunk of memory, you'd need the align(1).
34     return 0;
35 }
```

Ausgabe:

```
NodeLink:12
NodeLink1:9
NodeLink2:12
NodeLink3:16
```

Teil II.

Standard Bibliothek phobos

Die Standard Bibliothek **phobos** ist bei dem Compiler mit dabei. 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 48: 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 Objekt erstellt und gleichzeitig die Datei `testdatei.txt` öffnet.

5.1.2 In Datei schreiben

Listing 49: 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 geschrieben.

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 50: 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.

Listing 51: 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.writable) {
11            printf("Datei ist schreibbar\n");
12        }
13    } catch (OpenException e) {
14        printf("%.s\n", e.toString());
15    } finally {
16        printf("finally ausfuehren\n");
17        file.close();
18    }
19
20    return 0;
21 }
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 sozusagen mit **catch** eingefangen. Hier sollten Sie mal die Datei **testdatei.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 dass 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 52: Datei mit Puffer einlesen

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

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 einliest 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 53: 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 54: Datei mit `std.string.splitlines` ausgeben

```
1 import std.stdio, std.file;
2
3 int main() {
4
5     char[][] lines;
6     char[] Text;
7     char[] File = "testdatei.txt";
8     if (!std.file.exists(File)) {
9         throw new Exception(std.string.format("File '%s' notfound.", File)
10        );
11     } else {
12         Text = cast(char[]) std.file.read(File);
13         lines = std.string.splitlines(Text);
14     }
15     foreach (char[] line; lines) {
16         writefln(line);
17     }
18     return 0;
19 }
```

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 55: Datei mit `std.file.write` schreiben

```

1 import std.file;
2 import std.stdio;
3
4 int main() {
5     char[] str1 = "Zeile 1\n", str2 = "Zeile 2 \n";
6     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() {                                     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     writeln("Size = %d", Size);
17
18     if (exists("testdatei.txt")) {
19         writeln("testdatei.back ist da");
20     } else {
21         writeln("testdatei.back ist nicht da");
22     }
23
24     writeln("Datei Attribute /dev/xdb3 %o ", cast(int) getAttributes("/dev/hda"));
25     writeln("Datei Attribute /dev/video56 %o ", cast(int) getAttributes("/dev/tty1"));
26     writeln("Datei Attribute hello %o ", cast(int) getAttributes("hello"));
27     writeln("Datei Attribute /sys %o ", cast(int) getAttributes("/sys"));
28     writeln("Datei Attribute /tmp %o ", cast(int) getAttributes("/tmp"));
29     writeln("Datei Attribute /hello.d %o ", cast(int) getAttributes("d_buch.tex"));
30     writeln("Datei Attribute /tmp/.gdm_socket %o ", cast(int) getAttributes("/tmp/.gdm_socket"));
31     writeln("Datei Attribute /dev/cdrom %o ", cast(int) getAttributes("/dev/cdrom"));

```

```
32
33     if ( isfile("d_buch.tex") ) {
34         writefn("d_buch.tex ist eine Datei");
35     } else {
36         writefn("d_buch.tex ist ein Verzeichnis");
37     }
38
39     if ( isdir("/tmp") ) {
40         writefn("/tmp ist eine Datei");
41     } else {
42         writefn("/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     writefn("Hole aktuelles Verzeichnis %s",getcwd());
52
53     foreach (char[] directory; listdir("/")) {
54         writefn("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 56: Substrings

```
1 import std.stdio;
2 import std.string;
3
4 void main() {
5     char[] str = "hallo";
6     writeln("slice %s", str[2..4]);
7     char[] r = replaceSlice(str, str[2..4], "UBUNTU");
8     //char[] r = replaceSlice(str, "ll", "br");
9     writeln("r = %s", r);
10
11     for (int i = 0; i < r.length; i++)
12         writeln("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 mit `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 57: Strings splitten

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

Hier wird mit der *split* (Zeile 8) Funktion der String *s* in Zeile 5 nach dem *,* aufgesplittet. 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 58: 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.

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. 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 59: 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 60: 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 wird 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 61: Haus2

```
1 import std.stdio;
2
3 class Haus {
4     char[] 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( char[][] arg ) {
17     Haus standardHaus = new Haus;
18     char[] 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 `klingeln()` in Zeile 17 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 spielt 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 62: 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 63: Haus3

```
1 import std.stdio;
2 class Haus {
3     public char[] pfannenfarbe;
4     public char[] 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( char[][] 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 dass 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` mit einem Beispiel eingehen.

Listing 64: Haus4

```
1 import std::stdio;
2 class Haus {
3     public char[] pfannenfarbe;
4     public char[] verbleander = "Gelb";
5     void farben() {
6         char[] verbleander ="Gruen";
7         char[] 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( char[][] 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 `verbleander` ausgegeben. Innerhalb einer Methode kann man keine Instanzvariable deklarieren, sobald man ein Attribut vor der Variablen schreibt, gibt es eine Fehlermeldung.

8.4 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 65: 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 Konstruktor in Zeile 3 aufgerufen. Der Konstruktor hat hier keine Parameter.

Listing 66: Haus8

```
1 import std.stdio;
2 private import std.c.time;
3 class Haus {
4
5     public char[] farbe;
6
7     this(char[] farbe) {
8         writeln("Dach ist undicht");
9         this.farbe=farbe;
10        farbe = "Gelb";
11        writeln("Farbe = %s",this.farbe);
12        writeln("Farbe = %s",farbe);
13        this(2);
14    }
15
16    this(int anzGauben) {
17        writeln("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.

8.4.1 Static Konstruktor

Statische Konstrukturen 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 67: Statischer Konstruktor

```
1 import std.stdio;
2
3 class A {
4     public static int OBJEKT=0;
5     static this() {
6         OBJEKT++;
7         writeln("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     writeln("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 [8.9.4](#)

8.5 Destruktor

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

Listing 68: Haus7

```

1  import std.stdio;
2  private import std.c.time;
3  class Haus {
4      this() {
5          writeln("Dach ist undicht");
6      }
7
8      ~this() {
9          writeln("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 Objekte gelöscht wird.

8.5.1 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 69: 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          writeln("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.
```

8.6 Vererbung von Klassen

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

Listing 70: Haus9

```
1 private import std.stdio;  
2  
3 class Haus {  
4     private char[] 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 char[] pfannenfarbe="Rot";  
19     void brennen() {  
20         writefln("Der Kamin brennt im Schloss.");  
21         writefln("Pfannenfarbe = %s", this.pfannenfarbe);  
22     }  
23 }  
24  
25 int main( char[][] 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 ausgegeben. Als letztes wird in Zeile 30 noch mal die Farbe Rot ausgegeben.

8.7 Alles Super

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

Listing 71: Haus10

```
1 private import std.stdio;
2
3 class Villa {
4     void brennen() {
5         writeln("Der Kamin brennt in der Villa!");
6     }
7 }
8
9 class Schloss : Villa {
10     void brennen() {
11         writeln("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.

8.7.1 Super Konstruktoren

Listing 72: 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.

8.8 abstract

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

Listing 73: 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     writeln(" 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.

8.8.1 Interfaces

In Interfaces werden Methoden festgelegt, 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 74: interface

```
1  import std::stdio;
2
3  interface Fisch {
4      private static final int i =1;
5      char[] kiemen();
6  }
7
8  interface Tier {
9      char[] beine();
10 }
11
12 class Amphibie : Fisch, Tier {
13     char[] kiemen() {
14         return "Unterwasser atmen.";
15     }
16     char[] beine() {
17         return "Auf dem trockenen laufen.";
18     }
19 }
20
21 int main( char[][] 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 erben Interfaces werden wie in Zeile 12 durch Komma getrennt an die Klasse **Amphibie** vererbt. Man hätte es auch so programmieren können.

Listing 75: interface1

```
1  import std.stdio;
2
3  interface Fisch {
4      private static final int i =1;
5      char[] kiemen();
6  }
7
8  interface Tier: Fisch {
9      char[] beine();
10 }
11
12 class Amphibie : Tier {
13     char[] kiemen() {
14         return "Unterwasser atmen.";
15     }
16     char[] beine() {
17         return "Auf dem trockenen laufen.";
18     }
19 }
20
21 int main( char[][] arg ) {
22     Amphibie lurch = new Amphibie;
23     writeln(lurch.kiemen());
24     writeln(lurch.beine());
25     writeln("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 76: interface3

```
1 import std.stdio;
2
3 interface Animal {
4     char[] how_you_cry();
5 }
6
7 class Dog : Animal {char[] how_you_cry() {return "bark"; } }
8 class Cat : Animal {char[] how_you_cry() {return "mew"; } }
9 class Horse : Animal {char[] how_you_cry() {return "neigh"; } }
10 class Cow : Animal {char[] how_you_cry() {return "moo"; } }
11 class Mouse : Animal {char[] how_you_cry() {return "squeak"; } }
12
13 int main( char[][] 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         writeln( 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.

8.9 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 Variablen und Methoden der Vaterklasse zugreifen, also in der ganzen Vererbungshierarchie. Wenn kein Attribut bei den Variablen oder Methode angegeben wird, ist sie automatisch `protected`.

Listing 77: Person

```

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

```

Listing 78: 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 79: 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.

8.9.1 package

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

Listing 80: 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    //writeln("Vorname: %s", person.getVorname());
12    //writeln("Vorname: %s", person.vorname);
13    Arbeiter arbeiter = new Arbeiter();
14    writeln("Vorname-arbeiter: %s", arbeiter.getVorname-arbeiter());
15
16    return 0;
17 }
```

Listing 81: Person

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

Listing 82: Arbeiter

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


Listing 83: Angestellter

```

1 module pack.Angestellter;
2 private import std.stdio;
3 import pack.Person;
4 class Angestellter : Person {
5     public void getVorname-person() {
6         writeln(" Vorname_angestellter: ",this.vorname);
7         writeln(" 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 Variable `vorname` in Zeile 5 von der Klasse `Person` zugreifen. Deshalb kann auch von der Klasse `Arbeiter` auf die Variable `vorname` von der Klasse `Person` zugegriffen werden. Von der Klasse `Angestellter` kann auch auf die Variable `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 Compiler 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.

8.9.2 const

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

Listing 84: 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     writeln(" 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`

8.9.3 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 nicht mehr überschrieben werden.

Listing 85: Final Methoden

```
1 import std.stdio;
2
3 /* final */ class A {
4     /* final */ void getName() {
5         writeln(" Klasse A");
6     }
7 }
8
9 class B : A {
10     void getName() {
11         writeln(" 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.

8.9.4 static

Variablen, die mit dem Attribut **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 86: Klassenmethoden/Klassenvariablen

```
1 import std.stdio;
2
3 class A {
4     public static int OBJEKT=0;
5
6     public static void klassenMethode() {
7         writeln("Mich kann man ohne Objekt aufrufen!");
8     }
9
10    this() {
11        OBJEKT++;
12        writeln("Object = %d",OBJEKT);
13        //A.klassenMethode();
14    }
15 }
16
17 int main( char [][] arg ) {
18     A a1 = new A;
19     A a2 = new A;
20     A a3 = new A;
21     A.klassenMethode();
22     writeln("Es gibt %d Objekte.",A.OBJEKT);
23     //b.getName();
24     return 0;
25 }
```

Ausgabe:

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

Das Programm zählt die Anzahl der Objekte die von der Klasse A erstellt werden. Dies geschieht 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 [8.4.1](#).

8.9.5 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 87: 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, gäbe es einen Fehler beim kompilieren.

Teil IV.

Fortgeschrittene Programmierung

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

9 Boxing

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

Listing 88: 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     writeln("w = %s",w);
9
10    Box x = box(4);
11    float f;
12    try f = unbox!(float)(x);
13    catch (UnboxingException error) {
14        writeln("Geht nicht\n");
15    }
16    writeln("float = %f",f);
17
18    char c;
19    try c = unbox!(char)(x);
20    catch (UnboxingException error) {
21        writeln("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` initialisiert. 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 Box mit Hash

Listing 89: 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 }
```

10 Sonstige Funktionen

10.1 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, dass 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 dass 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 90: 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 91: 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.

10.2 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.

10.2.1 delegate mit Funktionen

Listing 92: 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     writeln(&dg);
13     writeln("a = %d",a);
14     int i = dg();
15     writeln("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 von `dg` ausgeben. Als letztes wird in Zeile 14 die Funktion `foo` über `delegate` aufgerufen und ausgeführt, und der Rückgabewert wird der Variablen `i` zugewiesen.

Listing 93: delegate1

```
1 import std.stdio;
2
3 int delegate(int) dg; // deklaration eines delegates(funktions Zeiger)
4
5 void main() {
6     //int a = 7;
7     int foo(int a) {
8         return a + 3;
9     }
10
11     dg = &foo; // Initialisierung dg zur Funktion foo
12     int i = dg(8); // Funktion foo wird aufgerufen und i wird auf 10
13     //        gesetzt
14     writeln("i = %d",i);
15 }
```

Ausgabe:


```
i = 11
```

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

Listing 94: `delegate2`

```
1 import std.stdio;
2
3 class Abc {
4     char[] s;
5     void print() {
6         writeln(s);
7     }
8 }
9
10 int main( char[][] 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.

10.2.2 delegate mit Methoden

Ein delegate Beispiel für Methoden.

Listing 95: delegate3

```
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     writeln("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.

10.2.3 delegate mit struct

Listing 96: delegate4

```
1 import std.stdio;
2
3 struct Foo {
4     int bar(int i) {
5         writeln("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     writeln("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.

11 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, Fließkommakonstanten, 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.

11.1 Template Funktionen

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

Listing 97: 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     writeln("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. Man kann sich das auch als Platzhalter für den anzugebenen 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 98: 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 99: 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.

11.1.1 Template Parameter

Listing 100: 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          writeln("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     writeln("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

11.1.2 Template mit 2 Parameter

Listing 101: See

```

1  import std.stdio;
2
3  template TSee(T,U) {
4      void see( T a, U b) {
5          writeln("a = ",a);
6          writeln("b = ",b);
7      }
8  }
9
10 void main() {
11     int a= 4;
12     char[] str="Hallo";
13     alias TSee!(int,char[]).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.

11.1.3 Template Struct

Listing 102: Struct Template

```

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

11.1.4 Template Klassen

Listing 103: 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     writeln(thing.x);
10    writeln("%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 104: Template

```
1 template SortTemplate(Type)
2 {
3     void insertionSort(Type array[], bit delegate(Type a, Type b) lessThan
4         )
5     {
6         for (int i=1; i < array.length; i++)
7         {
8             Type index = array[i];
9             int j = i;
10            while ((j > 0) && lessThan(index, array[j-1]))
11            {
```



```

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(char[] n, int a)
22     {
23         name = n;
24         age = a;
25     }
26
27     char[] 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

11.2 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 105: 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             writeln("x = %d",x);
14         }
15         writeln("x = %d",x);
16     }
17     writeln("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 106: 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 wir 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 105` in der abgeänderten Form.

Listing 107: 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             writeln("x = %d",x);
14         }
15         writeln("x = %d",x);
16     }
17     writeln("x = %d",x);
18 }
```

Ausgabe:

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

Nur die Zeilen 4,5 und 12 haben sich gegenüber 105 geändert. Es wurde ein Parameter T hinzu gefügt. Beim TemplateMixin wird dann der Integraltyp `int` mit übergeben.

11.2.1 Mixin bei Klassen

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

Listing 108: Mixin mit Klassen

```
1  import std.stdio;
2
3  class Bar {
4      void func() {
5          writeln("Foo.func()");
6      }
7  }
8
9  class Code : Bar {
10
11      void func() {
12          writeln("Code.func()");
13      }
14      void func1() {
15          writeln("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.

12 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, wieviele 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 mitgeteilt 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				
bit	1	0	1	Hat kein Typ
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	Flieskommazahlen
double	64	2.22507e-308f	1.79769e+308f	Flieskommazahl %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

13 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.

13.1 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 Destructor aufgerufen und gibt die Resource frei. Das freigeben der Resources geschieht mit einem Destructor. Da es ein Destructor in Java nicht gibt, ist es in der Programmiersprache auch nicht möglich das Konzept von RAII durch zu führen.

Als Beispiel hab ich mal ein Programm erstellt, das eine Datei öffnet und es soll durch RAII sichergestellt werden, das die Datei auch wieder geschlossen werden.

Listing 109: 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 geoeffnet.
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 geoeffnet.
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, dass der Destruktor aufgerufen wird, sobald die `auto reference` ausserhalb des Bereichs geht.

13.2 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 110: 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.

13.3 scope

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

13.3.1 scope(exit)

`scope(exit)` wird ausgeführt, wenn der Bereich also scope verlassen wird. Das heisst bei korrekten und inkorrekten verlassen des Bereichs wird scope ausgeführt.

Listing 111: `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) writeln("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.

13.3.2 scope(failure)

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

Listing 112: `scope(failure)`

```
1 import std.stdio;
2
3 void main() {
4     abc();
5 }
6
7 void abc() {
8     scope(failure) writeln("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.

13.3.3 scope(success)

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

Listing 113: `scope(success)`

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

Ausgabe:

```
5
Hallo
```

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

13.3.4 Dateibeispiel

Jetzt kommen wir zu dem Datei Beispiel:

Listing 114: `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     File file = new File;
9     file.open("testdatei.txt", FileMode.In);
10    while (!file.eof()) {
11        sleep(1);
12        scope(failure) { writeln("SCOPE"); file.close(); }
13        writeln("%s", file.readLine());
14        file.close();
15    }
16    file.close();
17    writeln("Hier hin");
18 }
```

Ausgabe:

```
Zeile 3
```

SCOPE**Error: Stream is not readable**

Da beim zweiten mal lesen die Exception geworfen wird, hab ich das **scope** vor dem **readLine** gesetzt. Hier wird dann mit **scope(failure)** die Datei noch mal geschlossen, auch wenn es keinen Sinn macht. Aber im normaler Weise schliesst man die Datei auch nicht während des Lesens.

Listing 115: 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.

13.3.5 scope rekursiv

Listing 116: 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 ausgefuehrt
```

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 **G**raphical **U**ser **I**nterface 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.

14 wxd

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

14.1 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 **usrlocallib**

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 antrieb, da die **libstdc++** 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.

14.1.1 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 117: 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
9             wxWidgets World");
10        frame.Show(true);
11        TopWindow = frame;
12        return true;
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 instanziiert. 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` instanziiert. Interessant 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.

14.1.2 StatusBar

Listing 118: 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
8              World");
9          frame.CreateStatusBar();
10         frame.StatusText = "Status Bar";
11         frame.Show(true);
12         TopWindow = frame;
13         return true;
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.

14.1.3 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=de>
- [21] <http://wxd.sourceforge.net>

Listings

1.	hello	10
2.	helloarm	15
3.	Kommentare	16
4.	char	18
5.	Formatspezifizierer	23
6.	Union	24
7.	Struct	24
8.	Struct	25
9.	Struct mit Union	26
10.	Konstanten	27
11.	Deklaration u. Initialisierung	27
12.	static und auto	29
13.	Variablen übergabe in Modulen mod_haupt.d	30
14.	Variablen übergabe in Modulen mod1.d	30
15.	Rangfolge von Operatoren	31
16.	Komplement	33
17.	Komplement	33
18.	Komplement	34
19.	for-Schleife	36
20.	for-Schleife	36
21.	while Schleife	37
22.	do-while Schleife	37
23.	foreach Schleife	38
24.	foreach Schleife	38
25.	Schleifenabbruch mit break	39
26.	if-else Anweisung	40
27.	switch Anweisung	41
28.	tenärer Operator	42
29.	Array deklaration	44
30.	3 dimensionales Array	45
31.	Array deklaration	46
32.	Array deklaration	47
33.	3 dimensionales Array	47
34.	Dynamische Matrix	48
35.	Assoziative Arrays	49
36.	Assoziative Array Funktionen	50
37.	Array deklaration	51
38.	Einfache Funktion	51
39.	Variable Argumentenliste	52
40.	Funktion mit inout	53
41.	Funktion mit out	53
42.	foreach mit inout	54
43.	Unittest	56
44.	Assert	56
45.	Assert	57
46.	pragma	58
47.	align	59

48.	Zeilenweise einlesen	60
49.	Datei schreiben	61
50.	Anhängendes schreiben	62
51.	Datei schreiben mit Fehlerbehandlung	63
52.	Datei mit Puffer einlesen	64
53.	Datei mit std.file.read einlesen	65
54.	Datei mit std.string.splitlines ausgeben	65
55.	Datei mit std.file.write schreiben	66
56.	Substrings	68
57.	Strings splitten	69
58.	Random	70
59.	Haus	72
60.	Haus1	73
61.	Haus2	73
62.	Haus5	74
63.	Haus3	75
64.	Haus4	76
65.	Haus6	76
66.	Haus8	77
67.	Statischer Konstruktor	78
68.	Haus7	79
69.	objectNull	79
70.	Haus9	80
71.	Haus10	81
72.	Haus11	82
73.	formen	83
74.	interface	84
75.	interface1	85
76.	interface3	86
77.	Person	87
78.	Hauptprogramm	87
79.	Angestellter	87
80.	PersonPackage	88
81.	Person	88
82.	Arbeiter	88
83.	Angestellter	89
84.	const	89
85.	Final Methoden	90
86.	Klassenmethoden/Klassenvariablen	91
87.	override	92
88.	Box	93
89.	Hash Box	94
90.	app	94
91.	libsquare	94
92.	delegate	96
93.	delegate1	96
94.	delegate2	97
95.	delegate3	98
96.	delegate4	98

97. Quadrierung	100
98. Quadrierung	101
99. swap	101
100. swap2	102
101. See	103
102. Struct Template	103
103. Klassen Template	104
104. Template	104
105. Mixin	107
106. Mixin1	108
107. Mixin mit Parameter	109
108. Mixin mit Klassen	110
109. RAII	114
110. try-catch-finally	115
111. scope(exit)	116
112. scope(failure)	116
113. scope(success)	117
114. scope	117
115. scope2	118
116. scope4	118
117. First Wx	120
118. First1 Wx	121

Sachregister

Zeichen	Cross-Compiling	14
~	cstream	39
ii	D	
&	Datentypen	18
A	dchar	19
abstract	Deklaration	27
abweisende schleife	Dekrement	37
ActiveSync	delegate	95
Addressoperator	delete	79
&	derr	39
align	din	39
Anfangsadresse	dmc	7
Annehmende	dout	39
append	E	
ar	enum	44
ARM	Escapesequence	10
Array	Event Loop	121
assoziative Arrays	exception	62
dynamisches Array	extern	30
rechteckige Arrays	F	
statischs Array	File	60
ASCII	Filehandle	63
Assembler	final	84, 90
Assoziativität	finally	63
Attribut	Fluchtoperator	10
Attribute	for	36
ausdruck	foreach	38, 49
auto	Formatspezifizierer	22
B	FreeBSD	6
Bezeichner	Fuß Schleife	37
Biblioheken	Funktionen	51
binär	G	
Bitoperatoren	Garbage Collector	6
boolean	gcc	8
box	gdc	8, 11
break	gdmd	11
C	getSize	66
call by reference	GNU Compiler Collection	6
call by value	GUI	119
cast	H	
catch	hash	49, 50
char	I	
chmod	Identifier	43
const		
copy		

if else	40	Operator	27
import	30	operatoren	31
in	53	Operatoren	31
Initialisierung	27	out	53
Inkrement	36	OutOfMemoryException	72
inout	53	override	91
Instanz	110		
Instanzmethode	91	P	
Instanzvariable	75, 91	package	88
interface	83	Postfix Array Deklaration	43
		Postfixdekrement	31
K		Postfixinkrement	31
key	50	Präfixdekrement	31
Klassenmethoden	90	Präfixinkrement	31
Klassenvariablen	90	pragma	58
Kommentare	16	Prefix Array Deklaration	43
Kommentare		private	86
Verschachtelte Kommentare	16	protected	86
Komplement	32	public	86
konkateniert	46	Punktnotation	73
Konstruktor	76		
Konvertierungsspezifizierer	111	R	
		RAII	113
L		Random	70
length	46	Referenz	100
libphobos.a	12, 60	remove	50
linker	11	rename	66
Logische Negation	32	replaceString	68
M		S	
Mac OS X	6	scope	116
main	10, 19, 66	scope(exit)	116
Menu	121	scope(failure)	116
Methoden	72	scope(success)	117
mixin	106	shared Libraries	94
module	60, 87	SkyOS	6
modulo	32	slice	68
MSDOS	13	Solaris	6
MySQL	12	sort	50
		split	69
N		splitlines	66
namespace	30	sprintf	22
null	79	Standardout	22
NULL	17	static	84, 90
		static Konstruktor	78
O		StatusBar	121
Objekt	72	Steuerzeichen	17
Objektdatei	10	stream	60
shared object	11	strip	11
ODER	33	struct	23
Operanden	31		

sudo	119
super	81
switch	41

T

Templates	100
tenär	31, 42
this	75
throw	63
toInt	39
try	63
try-catch-finally	113
typeof	52

U

Ubuntu	119
unär	31
unbox	93
UnboxException	93
UND	33
union	23
unittest	13, 60
upx	12

V

va_arg	51
Variable	27
Deklaration	27
Initialisierung	27
Variablen	
auto	28
static	28
Vererbung	80
Vergleichsoperator	40
Verkettung von Kunstrukturen	77

W

wchar	19
Windows Mobile	13
wrtf	10
wxd	119
wxFrame	121

X

XOR	34
-----------	----

Z

Zeilenvorschub	10
----------------------	----