

## LiesMich

fuer eine Distribution freier Software unter Windows  
für die Lehrveranstaltung **TB1-IN1** im **SS12**  
an der Beuth Hochschule für Technik Berlin.  
Autor: Ulrich Grude

## Inhaltsverzeichnis

<b>LiesMich</b> .....	<b>1</b>
1. Übersicht.....	1
2. Installation.....	2
3. Ein erstes Beispielprogramm mit der Windows-cmd-Shell bearbeiten.....	2
4. Ein erstes Beispielprogramm mit der bash-Shell bearbeiten.....	3
5. Besondere cmd- und bash-Shell-Fenster.....	3
6. Tabellarische Übersicht über wichtige Kommandos.....	5
7. Ein neues Projekt einrichten und mit dieser Distribution bearbeiten.....	5
8. Die a-Kommandos zum Ändern und Anpassen.....	6
9. Die Skripte setBcc32Cfg.cmd und setLccRegistry.cmd.....	6
10. Skripte zum Testen der Skripte dieser Distribution.....	7
11. Der Editor Textpad.....	7
12. Ein C-Interpreter.....	7

### 1. Übersicht

Diese Distribution umfasst folgende Compilations-Systeme (Compiler, Binder, Bibliotheken etc.):

den **Gnu C-Compiler gcc**, Version 4.5.2

(siehe <http://www.mingw.org>)

den **C-Compiler Lcc-Win32**, Version 3.8

(siehe <http://www.cs.virginia.edu/~lcc-win32/>)

den **Borland C/C++-Compiler bcc32**, Version 5.5.1

(siehe <http://edn.embarcadero.com/article/20633>)

den **Tiny-C-Compiler tcc**, Version 0.9.25

(siehe <http://bellard.org/tcc/>)

Außerdem enthält diese Distribution

- ein paar Beispielprogramme in C
- auf Windows portierte Unix-Werkzeuge (bash, find, which, ...)
- ein paar bash-Skripte zum Compilieren und Binden von C-Programmen
- ein paar cmd-Skripte für denselben Zweck

## 2. Installation

Diese Distribution besteht aus den folgenden 8 Ordnern (**fett** hervorgehoben) und 4 Dateien:

<b>BCC55</b>	Der Borland C/C++-Compiler
<b>BspC</b>	Beispiel-Programme in C
<b>C-Docs</b>	Rationale und Standard für die Sprache C
<b>FuerTextpad</b>	Dateien und Erläuterungen zum Einrichten des Editors Textpad
<b>Home</b>	Wird von der bash-Shell benutzt
<b>Lcc</b>	Der C-Compiler lcc-win32
<b>MinGW</b>	Eine MinGW-Distribution, inklusive gcc und bash etc.
<b>Tcc</b>	Der Tiny-C-Compiler tcc
LiesMich.odt	Die vorliegende Datei als Open Dokument Datei
LiesMich.pdf	Die vorliegende Datei als .pdf-Datei
startBash.cmd	cmd-Skript zum Starten einer besonderen bash-Shell
startCmd.cmd	cmd-Skript zum Starten einer besonderen cmd-Shell

Eine spezielle *Installation* ist *nicht notwendig*. Es genügt, diese Ordner und Dateien einem Windows-System in einem Installationsordner (der hier HD wie *home directory* genannt wird) zugänglich zu machen. Dabei kann HD z.B. folgendes sein:

- der oberste Ordner (das Wurzelverzeichnis) eines Speicher-Sticks (sehr empfehlenswert!)
- ein anderer Ordner auf einem Stick
- ein Ordner auf einer Festplatte

Der Benutzer muss natürlich berechtigt sein, im Installationsordner HD zu *lesen*, zu *schreiben* und (Skripte und Programme) *auszuführen*. Diese Bedingung ist meistens automatisch erfüllt.

## 3. Ein erstes Beispielprogramm mit der Windows-cmd-Shell bearbeiten

1. Starten Sie eine *besondere Windows-cmd-Shell*, indem Sie im Ordner %HD% auf die Datei `startCmd.cmd` doppelklicken. Dadurch wird ein cmd-Shell-Fenster geöffnet und der Ordner %HD%\BspC zu Ihrem aktuellen Arbeitsordner (current working directory) gemacht.

2. Navigieren Sie mit diesem Fenster (und dem Kommando `cd`) zum Ordner %HD%\BspC\Hallo01.

3. Geben Sie das Kommando `b hallo01` ein.

Daraufhin sollten die folgenden Zeilen ausgegeben werden:

```
----- 0
Eine ausfuehrbare Datei hallo01.exe wird erzeugt.
Compiler und Binder: Gnu-C-Compiler-und-Binder gcc.
----- 1
Compiliert und gebunde werden: hallo01.c
----- 2
```

Das Kommando `b` (oder mit vollem Namen: `b.cmd`) compiliert alle `.c`-Dateien im aktuelle Arbeitsordner mit dem Gnu-C-Compiler `gcc` zu Objektdateien (`.o`-Dateien) und bindet die Objektdateien zu einer `.exe`-Datei zusammen. Gibt man hinter `b` einen Namen an (z.B. `hallo01` oder `karlHeinz`) so legt man dadurch den Namen der `.exe`-Datei fest. Ausserdem "merkt sich" das System den Namen (in der Datei %HD%\BspC\AktuelleDatei.nam). Gibt man später nur `b` ein (ohne Namen dahinter), wird der zuletzt "gemerkte" Name verwendet.

Startet man das Programm `hallo01.exe`, so sollte es folgende Zeilen ausgeben:

```
*****
Hallo aus einem C-Programm namens hallo01!
*****
```

Der Kommando-Name `b` soll an "compilieren und binden" erinnern.

Das Kommando `e` (oder mit vollem Namen: `e.cmd`) entfernt (löscht) im aktuellen Arbeitsordner alle von einem Compiler oder Binder erzeugten Dateien.

Das Kommando `b` ruft den Gnu-C-Compiler-und-Binder `gcc` auf.

Das Kommando `bb` ruft den Borland C++ Compiler-und-Binder `bcc32` auf.

Das Kommando `bl` ruft den Little-C-Compiler/Binder `lcc/lcclnk` auf.

Das Kommando `bt` ruft den Tiny-C-Compiler-und-Binder `tcc` auf.

Regelmäßig *verschiedene* C-Compiler-und-Binder zu verwenden, hat mehrere Vorteile. Solange man den C-Standard (ca. 500 Seiten, siehe die Datei `C99-Standard.pdf` im Ordner `%HD%\C-Docs`) noch nicht auswendig kann und nur *einen* Compiler *A* verwendet, liegt es nahe, die Sprache *die A akzeptiert und übersetzt* und *die Sprache C* zu verwechseln. Und wenn man sich über die unverständlichen Fehlermeldungen eines Compilers ärgert, kann es beruhigend wirken zu sehen, dass die Meldungen anderer Compiler noch unverständlicher sind :-).

#### 4. Ein erstes Beispielpogramm mit der bash-Shell bearbeiten

1. Starten Sie eine *besondere bash-Shell*, indem Sie im Ordner `$HD` auf die Datei `startBash.cmd` doppelklicken. Dadurch wird ein bash-Shell-Fenster geöffnet und der Ordner `$HD/BspC` zu Ihrem aktuellen Arbeitsordner (current working directory) gemacht.

2. Navigieren Sie mit diesem Fenster (und dem Kommando `cd`) zum Ordner `$HD/BspC/Hallo01`.

3. Geben Sie das Kommando `b hallo01` ein.

Daraufhin sollten die folgenden Zeilen ausgegeben werden:

```
----- 0
Eine ausfuehrbare Datei hallo01.exe wird erzeugt.
Compiler und Binder: Gnu-C-Compiler-und-Binder gcc.
----- 1
Compiliert und gebunden werden: hallo01.c
----- 2
```

Das Kommando `b` (es heißt auch mit vollem Namen nur `b`) compiliert alle `.c`-Dateien im aktuelle Arbeitsordner mit dem Gnu-C-Compiler `gcc` zu Objektdateien (`.o`-Dateien) und bindet die Objektdateien zu einer `.exe`-Datei zusammen. Gibt man hinter `b` einen Namen an (z.B. `hallo01` oder `karlHeinz`) so legt man dadurch den Namen der `.exe`-Datei fest. Ausserdem "merkt sich" das System den Namen (in der Datei `$HD/BspC/AktuelleDatei.nam`). Gibt man später nur `b` ein (ohne Namen dahinter), wird der zuletzt "gemerkte" Name verwendet.

Startet man das Programm `hallo01.exe`, so sollte es folgende Zeilen ausgeben:

```
*****
Hallo aus einem C-Programm namens hallo01!
*****
```

Das Kommando `e` (es heißt auch mit vollem Namen nur `e`) entfernt (löscht) im aktuellen Arbeitsordner alle von einem Compiler oder Binder erzeugten Dateien.

Das Kommando `b` ruft den Gnu-C-Compiler-und-Binder `gcc` auf.

Das Kommando `bb` ruft den Borland C++ Compiler-und-Binder `bcc32` auf.

Das Kommando `bl` ruft den Little-C-Compiler/Binder `lcc/lcclnk` auf.

Das Kommando `bt` ruft den Tiny-C-Compiler-und-Binder `tcc` auf.

#### 5. Besondere cmd- und bash-Shell-Fenster

Wenn man (unter Windows, z.B. im Explorer) einen Links-Doppelklick auf die Datei `%HD%\startCmd.cmd` ausführt, öffnet sich eine *besondere cmd-Shell*. Das Besondere an diesem cmd-Shell-Fenster ist, dass man darin direkten Zugriff auf alle Werkzeuge dieser Software-Distribution hat. Dieser direkte Zugriff wird dadurch ermöglicht, dass während des Startvorgangs der Shell bestimmte Anpassungen vorgenommen werden, vor allem wird der Inhalt der Umgebungsvariablen `PATH` um ein

paar Pfadnamen erweitert. Diese Anpassungen werden durch das Skript `setEnv.cmd` ausgeführt. Wenn man die besondere cmd-Shell schließt, hat die Variable `PATH` aber wieder ihren alten Wert.

Für eine *besondere bash-Shell* (die man durch einen Links-Doppelklick auf `$HD/startBash.cmd` öffnet) gilt ganz Entsprechendes. Beim Start einer besonderen bash-Shell werden die Anpassungen durch das Skript `setEnv.bsh` ausgeführt.

In einer *besonderen (cmd- oder bash-) Shell* stehen einem eine Reihe nützlicher Kommandos zur Verfügung, die man hinter dem Promptzeichen eingeben muss. In einer cmd-Shell ist das Prompt-Zeichen ein Größerzeichen `>`, in einer bash-Shell ist es ein Dollarzeichen `$`. Im folgenden verwenden wir `$>` als Promptzeichen.

```
$> echu
```

Gibt den Inhalt der Umgebungsvariablen `PATH` *in lesbarer Form* aus (während das Kommando `echo %PATH%` bzw. `echo $PATH`, welches auch in einer normalen Shell möglich ist, den Inhalt derselben Variablen *in unlesbarer Form* ausgibt).

```
$> echu c
```

Gibt den Inhalt der Umgebungsvariablen `CLASSPATH` *in lesbarer Form* aus. Den Inhalt aller anderen Umgebungsvariablen kann man sich anzeigen lassen, indem man den Namen der Variablen als Parameter von `echu` angibt, z.B. so:

```
$> echu home
```

"echu" soll an "echo für Umgebungsvariablen" erinnern.

Mit den folgenden Kommandos kann man sich Hilfe-Informationen von den C-Compilern dieser Distribution anzeigen lassen:

```
$> gcc --help
```

```
...
```

```
$> bcc32
```

```
...
```

```
$> lcc
```

```
...
```

```
$> tcc
```

Der Compiler `bcc32` gibt bei jedem Aufruf auch seine Versions-Nr aus. Mit den folgenden Kommandos kann man die Versions-Nrn der anderen Compiler erfahren:

```
$ gcc -dumpversion
```

```
...
```

```
$ lcc -v
```

```
...
```

```
$ tcc -v
```

Die folgenden Kommandos compilieren eine Quelldatei namens `qdat.c` in eine Objektdatei und binden die Objektdatei in eine ausführbare Datei namens `adat.exe`:

```
$ gcc -o adat.exe qdat.c
```

```
...
```

```
$ bcc32 -eadat.exe qdat.c
```

```
...
```

```
$ lcc qdat.c
```

```
$ lcclnk -o adat.exe qdat.obj
```

```
...
```

```
$ tcc -o adat.exe qdat.c
```

Ein C-Compiler übersetzt jede *Quelldatei* in eine *Objektdatei* (das war schon lange vor dem Aufkommen der objektorientierten Programmierung so und hat nichts mit ihr zu tun). Standardmäßig haben Objektdateien bei den Compilern `gcc` und `tcc` die Erweiterung `.o`, bei den Compilern `bcc32` und `lcc` dagegen die Erweiterung `.obj`.

Die Kommandos `b`, `bb`, `bl`, `bt` (jeweils mit oder ohne einem Dateinamen wie z.B. `hallo01` als Parameter) wurden schon in den vorigen beiden Abschnitten besprochen.

Die Kommandos `c`, `cb`, `cl` und `bt` dienen dazu, eine einzelne `.c`-Datei (C-Quelldatei) mit einem der Compiler in eine Objektdatei zu übersetzen. Genauer:

```
$> c hallo01
```

Dieses Kommando bewirkt, dass im aktuellen Arbeitsordner eine Datei `hallo01.c` gesucht wird. Falls keine gefunden wird, erfolgt nur eine Fehlermeldung. Sonst wird die Datei `hallo01.c` mit dem Gnu-C-Compiler `gcc` in eine Objektdatei übersetzt und das System "merkt sich den Namen" `hallo01`.

Angenommen, `hallo01` ist der zuletzt gemerkte Name. Dann bewirkt das Kommando

```
$> c
```

genau das Gleich wie das vorige Kommando (`c hallo01.c`).

Das Kommando `c` ruft den Gnu-C-Compiler `gcc` auf.

Das Kommando `cb` ruft den Borland C++ Compiler `bcc32` auf.

Das Kommando `cl` ruft den Little-C-Compiler `lcc` auf.

Das Kommando `ct` ruft den Tiny-C-Compiler `tcc` auf.

Diese `c`-Kommandos sind für den Fall gedacht, dass eine C-Quelldatei vermutlich noch zahlreiche Fehler enthält, so dass sie noch mehrmals verbessert und erneut compiliert werden muss, ehe es sich lohnt, einen Binde-Versuch (z.B. mit einem der `b`-Kommandos) zu machen.

## 6. Tabellarische Übersicht über wichtige Kommandos

Mit AAO ist im Folgenden der *aktuelle Arbeitsordner* (the current working directory) gemeint.

1	<code>echu</code>	Inhalt der Variablen <code>PATH</code> wird angezeigt	
2	<code>echu c</code>	Inhalt der Variablen <code>CLASSPATH</code> wird angezeigt	
3	<code>echo home</code>	Inhalt der Variablen <code>HOME</code> wird angezeigt	
4	<code>b hallo01</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>hallo01.exe</code> erzeugt	mit <code>gcc</code>
5	<code>bb hallo01</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>hallo01.exe</code> erzeugt	mit <code>bcc32</code>
6	<code>bl hallo01</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>hallo01.exe</code> erzeugt	mit <code>lcc</code>
7	<code>bt hallo01</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>hallo01.exe</code> erzeugt	mit <code>tcc</code>
8	<code>c hallo01</code>	Die Datei <code>hallo01.c</code> wird übersetzt in <code>hallo01.o</code>	mit <code>gcc</code>
9	<code>cb hallo01</code>	Die Datei <code>hallo01.c</code> wird übersetzt in <code>hallo01.obj</code>	mit <code>bcc32</code>
10	<code>cl hallo01</code>	Die Datei <code>hallo01.c</code> wird übersetzt in <code>hallo01.obj</code>	mit <code>lcc</code>
11	<code>ct hallo01</code>	Die Datei <code>hallo01.c</code> wird übersetzt in <code>hallo01.o</code>	mit <code>tcc</code>

Angenommen, `berta17` ist der "zuletzt gemerkte" Name. Dann gilt:

12	<code>b</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>berta17.exe</code> erzeugt	mit <code>gcc</code>
13	<code>bb</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>berta17.exe</code> erzeugt	mit <code>bcc32</code>
14	<code>bl</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>berta17.exe</code> erzeugt	mit <code>lcc</code>
15	<code>bt</code>	Aus allen <code>.c</code> -Dateien im AAO wird <code>berta17.exe</code> erzeugt	mit <code>tcc</code>
16	<code>c</code>	Die Datei <code>berta17.c</code> wird übersetzt in <code>berta17.o</code>	mit <code>gcc</code>
17	<code>cb</code>	Die Datei <code>berta17.c</code> wird übersetzt in <code>berta17.obj</code>	mit <code>bcc32</code>
18	<code>cl</code>	Die Datei <code>berta17.c</code> wird übersetzt in <code>berta17.obj</code>	mit <code>lcc</code>
19	<code>ct</code>	Die Datei <code>berta17.c</code> wird übersetzt in <code>berta17.o</code>	mit <code>tcc</code>
20	<code>e</code>	Entfernt aus dem AAO alle von Compilern/Bindern erzeugten Dateien	

## 7. Ein neues Projekt einrichten und mit dieser Distribution bearbeiten

Im Ordner `%HD%\BspC` gibt es eine Reihe von Projektordnern (namens `AlleStdKopfdat`, `DefDek01`, `Hallo01`, `Hallo02`, ...). Jeder Projektordner enthält alle Quelldateien eines C-Programms. Mit einer besonderen Shell und den Werkzeugen dieser Distribution kann man daraus eine ausführbare Datei (eine `.exe`-Datei) erzeugen und starten.

Wenn man ein neues C-Programm mit den Werkzeugen dieser Distribution bearbeiten will, sollte man im Ordner `%HD%\BspC` einen weiteren Projektordner erstellen und alle Quelldateien des neuen C-Programms dort erstellen oder hineinkopieren. Dann kann man auch aus diesen neuen Quelldateien mit einer besonderen Shell und den Werkzeugen dieser Distribution eine ausführbare Datei erzeugen und starten.

## 8. Die a-Kommandos zum Ändern und Anpassen

Die bisher behandelten b-, c- und e-Kommandos sind durch Skripte realisiert, die alle im Ordner %HD%\Skripte stehen. Man kann sie nur in *besonderen* (cmd- oder bash-) *Shell-Fenstern* aufrufen, weil sie sich auf bestimmte Einstellungen (vor allem auf bestimmte Einträge in der PATH-Variablen) verlassen.

Im Ordner %HD%\BspC\Hallo03 gibt es noch weitere a- und e-Skripte, die man in einer *normalen cmd- bzw. bash-Shell* aufrufen kann. Diese Skripte bearbeiten nur die C-Quelldatei, die im selben Ordner wie die Skripte steht. Die a-Skripte "machen alles" (sie compilieren, binden und führen die fertige .exe-Datei auch aus). Die e-Skripte entfernen alle von den a-Skripten erzeugten Dateien.

Bevor die a- Skripte ihre Aufgabe erfüllen, rufen sie ihrerseits das Skript setEnv.cmd bzw. setEnv.bsh auf und verschaffen sich dadurch Zugriff auf die Werkzeuge dieser Distribution.

Ein fortgeschrittener Benutzer kann den Ordner Hallo03 an eine beliebige Stelle seines Dateisystems kopieren. Damit das Programm hello03.exe auch an diesem ganz anderen Ort compiliert, gebunden und gestartet werden kann, muss der Benutzer dann an den a-Skripten von Hand bestimmte (hoffentlich naheliegende) Änderungen und Anpassungen vornehmen.

Der Ordner %HD%\BspC\Hallo03 und die darin befindlichen a- und e-Skripte sollen also einen fortgeschrittenen Benutzer dabei unterstützen, neue Projektordner nicht "innerhalb der Distribution" (im Ordner %HD%\BspC) anzulegen, sondern an beliebigen anderen Orten seines Dateisystems.

## 9. Die Skripte setBcc32Cfg.cmd und setLccRegistry.cmd

Ein C-Compiler muss auf bestimmte Standard-Kopfdateien (.h-Dateien) zugreifen können. Die stehen typischerweise in einem Ordner namens include.

Ein Binder muss auf bestimmte Standard-Bibliotheken zugreifen können. Die stehen typischerweise in einem Ordner namens lib.

Wie dieser Zugriff im Einzelnen geregelt wurde, ist von Compiler zu Compiler verschieden.

Die Compiler gcc und tcc "wissen von allein", wo die Standard-Kopfdateien und Standard-Bibliotheken stehen. Die Compiler bcc32 und lcc erfordern dagegen "a little help from their friends".

Der Compiler/Binder bcc32 erwartet, dass in seinem Bin-Ordner eine Datei namens bcc32.cfg steht, die zwei Zeilen wie die folgenden enthält:

```
-I"S:\TB1-IN1-SWD\ceh\BCC55\Include"
-L"S:\TB1-IN1-SWD\ceh\BCC55\Lib"
```

Hinter -I muss der absolute Pfad des include-Ordners und hinter -L muss der absolute Pfad des lib-Ordners stehen.

Wenn man die vorliegende Software-Distribution in einen bestimmten Ordner kopiert (z.B. in den Ordner H: oder in den Ordner S:\TB1-IN1-SWD\ceh etc.), muss man den Inhalt der Datei bcc32.cfg entsprechend anpassen. Diese Aufgabe erledigt das Skript setBcc32Cfg.cmd, welches von den Skripten setEnv.cmd und setEnv.bsh aufgerufen wird.

Der Compiler lcc und der Binder lcclnk erwarten, dass die absoluten Pfade ihres include- und lib-Ordners (an bestimmten Stellen) in der Windows-Registry stehen.

Wenn man die vorliegende Software-Distribution in einen bestimmten Ordner kopiert (z.B. in den Ordner H: oder in den Ordner S:\TB1-IN1-SWD\ceh etc.), muss man die Inhalte dieser Registry-Einträge entsprechend anpassen. Diese Aufgabe erledigt das Skript setLccRegistry.cmd, welches ebenfalls von den Skripten setEnv.cmd und setEnv.bsh aufgerufen wird.

Die beiden Skripte setBcc32Cfg.cmd und setLccRegistry.cmd müssen Windows-Pfade mit abwärts-Schrägstrichen (z.B. S:\TB1-IN1-SWD\ceh\) manipulieren, und nicht MinGW-Pfade mit

aufwärts-Schrägstrichen (z.B. /s/TB1-IN1-SWD/ceh/). Deshalb wurden Sie nur als `.cmd`-Skripte realisiert und es gibt hier keine äquivalenten `bash`-Skripte.

Die beiden Skripte `setBcc32Cfg.cmd` und `setLccRegistry.cmd` müssten eigentlich nur *einmal* aufgerufen werden, nachdem man die vorliegende Software-Distribution in einen bestimmten Ordner kopiert hat. Tatsächlich werden sie bei jedem Start einer besonderen Shell aufgerufen, was aber auf vielen PCs "unauffällig schnell" geht.

## 10. Skripte zum Testen der Skripte dieser Distribution

Das `bash`-Skript `$HD/Skripte/testeSkripte01.bsh` dient dazu, die `.cmd`- und `.bsh`-Skripte im Ordner `$HD/BspC/Hallo03` zu testen. Bevor man es starten kann, muss man das Verzeichnis `$HD/Skripte` zum aktuellen Arbeitsordner machen (sonst beendet es sich sofort mit einer Fehlermeldung). Es erzeugt eine Protokolldatei `$HD/Skripte/testeSkripte01.txt`, die "von Hand" (per Auge?) geprüft werden muss.

Das `bash`-Skript `$HD/BspC/Test/testeSkripte02.bsh` und das `cmd`-Skript `%HD%\BspC\Test\testeSkripte02.cmd` dienen dazu, die Skripte

`b, bb, bl, bt`  
`c, cb, cl, ct`

im Ordner `$HD/Skripte` zu testen. Sie geben für jedes getestete Skript eine o.k.-Meldung oder eine Fehlermeldung aus.

Das `bash`-Skript `$HD/BspC/Test/testeSkripte03.bsh` und das `cmd`-Skript `%HD%\BspC\Test\testeSkripte03.cmd` dienen dazu zu testen, welche Fehlermeldungen die `c`-Skripte im Ordner `%HD%\Skripte` in bestimmten Fehlersituationen ausgeben. Die Fehlermeldungen müssen "von Hand" (per Auge?) geprüft werden.

## 11. Der Editor Textpad

Wie man den Editor *Textpad* zum Bearbeiten von `bash`-Skripten, `cmd`-Skripten und C-Quelldateien einrichten kann, wird in der Datei `$HD/FuerTextpad/DenTextpadEinrichten.txt` beschrieben.

## 12. Ein C-Interpreter

Elegantier als mit einem Compiler, einem Binder und einem Ausführer für ausführbare Dateien (unter Windows: für `.exe`-Dateien) kann man Quellprogramme (z.B. C-Quellprogramme) direkt von einem *Interpreter* ausführen lassen. Insbesondere während der *Entwicklung* eines Programms reicht die Geschwindigkeit eines Interpreters in aller Regel vollständig aus. Aber auch in anderen Situationen sind Interpreter heutzutage so schnell, dass man den "umständlichen Umweg" über Compiler und Binder häufig nicht mehr nötig hat.

Einen besonders guten, ausgereiften und kostenlosen C/C++-Interpreter namens `ch` (sprich: ssi eitsch) findet man unter der Adresse <http://www.softintegration.com/>. Allerdings muss man sich dort erst registrieren, bekommt dann innerhalb von ein paar Stunden per Email ein Passwort und damit kann man sich dann eine der Editionen des Interpreters herunterladen. Für StudentInnen besonders empfehlenswert ist die *ch student edition*. Die kann unter anderem C-Quellprogramme, die dem Standard C99 entsprechen, ausführen.

Um ein C-Programm, welches nur aus *einer* Quelldatei besteht, vom Interpreter `ch` ausführen zu lassen, genügt es, dem Interpreter den Pfadnamen der Quelldatei (z.B. `hallo01.c` oder `H:\BspC\Hallo02\hallo02.c`) mitzuteilen. Um ein aus *mehreren Quelldateien* bestehendes C-Programm vom `ch` ausführen zu lassen, muss man ein spezielles `ch`-Skript erstellen, in dem man dem Interpreter mitteilt, welche Quelldateien zum Programm gehören.

**Beispiel-01:** Ein `ch`-Skript für die Interpretation eines Programms `multiDat01`:

```
1 #!/bin/ch
2 /* -----
3 ch-script multiDat01.ch
4 ----- */
5 #pragma import "multiDat01A.c"
6 #pragma import "multiDat01B.c"
```

Die Zeile 1 kennzeichnet die Datei als ein `ch`-Skript. Die Zeilen 2 bis 4 sind ein Kommentar. Die Zeilen 5 und 6 enthalten die Pfadnamen von zwei C-Quelldateien. Dieses Skript und die C-Quelldateien findet man auch im Ordner `%HD%\BspC\MultiDatei01`.

Den Interpreter `ch` darf man benutzen, aber nicht weiter verbreiten. Darum wurde er nicht in diese Software-Distribution aufgenommen.