Kommandos zur Unterstützung der Programmierung in C

Als *Kommandos* werden hier alle Befehle bezeichnet, die man in einem *Kommandoeingabe-Fenster* eingeben kann. Unter Linux ist z. B. ein sogenanntes *Konsolen-Fenster*, in dem eine `bash`-Shell läuft, ein Kommandoeingabe-Fenster. Unter Windows ist z. B. ein *DOS-Fenster* (manchmal auch als *DOS-Eingabeaufforderung* bezeichnet) oder ein *Cygwin-bash-Fenster* ein Kommandoeingabe-Fenster.

Typische Beispiele für Kommandos unter Linux:

`cd` change directory, wechsle das aktuelle Arbeitsverzeichnis
`ls` list, zeige Informationen über Dateien an
`rm` remove, lösche Dateien
`gcc` gnu compiler collection, compiliere und/oder binde Programmteile

Typische Beispiele für Kommandos unter Windows:

`cd` change directory, wechsle das aktuelle Arbeitsverzeichnis
`dir` directory, zeige Informationen über Dateien an
`del` delete, lösche Dateien
`gcc` gnu compiler collection, compiliere und/oder binde Programmteile

In diesem Papier sollen einige besonders häufig benötigte Kommandos kurz erläutert werden.

1. C-Programme compilieren und binden mit gcc (Gnu Compiler Collection)

Ein C-Programm, welches nur aus einer Quelldatei namens `hallo01.c` besteht:

```
1 // Datei hallo01.c
2 #include <stdio.h>
3
4 int main() {
5    printf("****************************************\n");
6    printf("Hallo aus einem C-Programm namens hallo01!\n");
7    printf("****************************************\n");
8    return 0; // Meldung an das Betriebssystem: Alles o.k.
9 } // main
```

Dieses C-Programm compilieren und binden mit dem gcc:

```
10 > gcc hallo01.c
```

Das Ergebnis ist eine ausführbare Datei namens `a.exe` (unter Windows) bzw. `a.out` (unter Linux).

Das C-Programm compilieren und binden und dem Ergebnis einen selbstgewählten Namen geben:

```
11 > gcc -o hallo01.exe hallo01.c
```

Das Ergebnis ist eine ausführbare Datei namens `hallo01.exe` (unter Windows und Linux). Mit der Option `-o` legt man den Namen des **o**`utputs` fest.

Anmerkung: Auch unter Linux ist es sinnvoll, ausführbaren Dateien Namen zu geben, die mit `.exe` enden. Damit erleichtert man sich das Erkennen und Bearbeiten aller ausführbaren Dateien in einem Verzeichnis (z. B. kann man dann mit dem Befehl `rm *.exe` alle ausführbaren Dateien im aktuellen Verzeichnis löschen).

Anmerkung: Den Namen der ausführbaren Datei (den man nach `-o` angibt) kann man völlig frei und unabhängig vom Namen der Quelldatei(en) wählen, z. B. so:

```
12 > gcc -o KarHeinz.exe hallo01.c
```

Häufig ist es günstig, das Kommando zum Compilieren und Binden eines Programms nicht direkt einzugeben, sondern es in ein *Shell-Skript* zu schreiben und diesem Shell-Skript einen besonders kurzen, leicht zu merkenden und leicht einzugebenden Namen zu geben, z. B. den Namen `xx`. Es folgt ein

bash-Skript zum Compilieren und Binden des Programms hallo01. Dieses Skript ruft die erzeugte ausführbare Datei hallo01.exe auch gleich noch auf:

```
13 #! /bin/bash
14 # Datei xx zum Binden und Compilieren des Programms hallo01
15 gcc -o hallo01.exe hallo01.c
16 hallo01.exe
```

Unter Linux ist eine solche Skript-Datei nicht automatisch ausführbar. Wir müssen sie mit dem chmod-Kommando ausführbar machen, z.B. so:

```
17 > chmod a+x xx
```

Dieses Kommando bewirkt, dass alle Benutzer (a wie all) die Ausführungsberechtigung (x wie execute) für die Datei namens xx bekommen.

Ein Programm, welches aus *mehreren* Quelldateien quellDat01.c, quellDat02.c und quellDat03.c besteht, kann man ganz entsprechend compilieren und binden, etwa so:

```
18 > gcc -o meinProg.exe quellDat01.c quellDat02.c quellDat03.c
```

Auch in diesem Beispiel haben wir der ausführbaren Datei einen Namen gegeben (meinProg.exe), der nichts mit den Namen der Quelldateien zu tun hat. Allgemein gilt: Genau eine der Quelldateien muss ein Unterprogramm namens main enthalten. Häufig nennt man die ausführbare Datei entsprechend *der* Quelldatei, die das main-Unterprogramm enthält.

2. Die Umgebungsvariable PATH ansehen

In einem bash-Fenster kann man sich den Wert der Umgebungsvariablen PATH durch Eingabe des folgenden Kommandos anzeigenn lassen:

```
1 > echo $PATH
```

Falls das selbstgeschriebene Skript namens echu ("Echo für Umgebungsvariablen") zur Verfügung steht, ist das folgende Kommando leichter einzugeben und produziert eine leichter lesbare Ausgabe:

```
2 > echu path
```

Die PATH-Variable enthält (unter Linux und Windows) eine Folge von *Pfadnamen* von Verzeichnissen (Pfadnamen von einzelnen Dateien sind nicht erlaubt). Unter Linux sind die Pfadnamen durch *Doppelpunkte* : voneinander getrennt, unter Windows durch *Semikolons* ;. In einem Cygwin-bash-Fenster unter Windows gelten ganz ähnliche Regeln wie unter Linux (in der PATH-Variablen stehen also Doppelpunkte :, keine Semikolons ;).

Beispiel unter Linux:

```
3 > echo $PATH
4 > /bin:/user/bin:/home/s123456/bsp01
```

In diesem Beispiel enthält die PATH-Variable 3 Pfadnamen. Wenn der Benutzer versucht, in einem bash-Fenster mit dem Kommando

```
5 > otto
```

ein Programm namens otto zu starten, so wird in den entsprechenden 3 Verzeichnissen (und sonst nirgends!) nach einer ausführbaren Datei namens otto gesucht. Wenn keine gefunden wird, gibt die bash-Shell eine Fehlermeldung aus (bash: otto: command not found). Sonst wird die (ausführbare) Datei otto ausgeführt.

Beispiel unter Windows:

```
6 > echo %PATH%
7 > C:\Windows;C:\Windows\system32;D:\meine\CehBeispiele
```

Auch in diesem Beispiel enthält die PATH-Variable 3 Pfadnamen. Wenn der Benutzer versucht, in einem DOS-Fenster mit dem Kommando

```
8 > otto
```

ein Programm namens `otto` zu starten, so wird in den entsprechenden 3 Verzeichnissen (und sonst nirgends!) nach einer ausführbaren Datei namens `otto.exe` oder `otto.bat` oder `otto.cmd` gesucht (ähnlich wie unter Linux).

3. Die Verzeichnisnamen Punkt . und Punkt Punkt ..

Der sehr kurze Name `.` (Punkt) ist eine Abkürzung für den *absoluten Pfadnamen des aktuellen Arbeitsverzeichnisses*. Mit dem Kommando `cd` (change directory) verändert man also unter anderem die Bedeutung dieses Namens.

Der Verzeichnisname Punkt `.` bezeichnet *immer* das aktuelle Arbeitsverzeichnis, egal wie oft wir es z. B. durch `cd`-Kommandos wechseln. Der Verzeichnisname Punkt `.` ist also "abstrakter", und damit "mächtiger" als jeder konkrete Verzeichnisname (siehe dazu auch den folgenden Abschnitt).

Der auch nicht sehr lange Name `..` ist eine Abkürzung für den absoluten Pfadnamen des Mutterverzeichnisses ("eins höher") des aktuellen Arbeitsverzeichnisses.

4. Die Umgebungsvariable PATH verändern

Angenommen, wir arbeiten unter Linux und im aktuellen Arbeitsverzeichnis steht eine ausführbare Datei namens `otto`, das aktuellen Arbeitsverzeichnis steht aber nicht in der PATH-Variablen. Um die Datei `otto` trotzdem zu starten, muss man ihren absoluten Pfadnamen angeben, etwa so:

```
 9 > /home/s123456/bsp01/bin/otto
10 > ./otto
```

In diesem Beispiel soll `/home/s123456/bsp01/bin` das aktuelle Arbeitsverezichnis sein, welches man auch mit dem kürzeren Namen Punkt `.` bezeichnen kann. Somit bewirken die beiden Kommandos in Zeile 9 und 10 dasselbe (und man verwendet praktisch immer nur das kürzere).

Unter Linux hat jeder Benutzer ein Heimatverzeichnis. Immer wenn das Shell-Programm `bash` gestartet wird, sucht es zuerst im Heimatverzeichnis des Benutzers nach einer Datei namens `.bashrc` (wie "bash resources") und führt die Kommandos darin aus (bzw. macht nichts, falls es die Datei namens `.bashrc` nicht findet). Indem man die Datei `.bashrc` mit einem Editor verändert und geeignete Kommandos hineinschreibt, kann man z.B. die Werte von Umgebungsvariablen wie PATH oder CLASSPATH verändern.

Angenommen, wir wollen häufig ausführbare Dateien starten, die im aktuellen Arbeitsverzeichnis liegen, das aktuelle Arbeitsverzeichnis steht aber (noch) nicht in der PATH-Variablen. Dann können wir es in die PATH-Variable schreiben lassen, indem wir das folgende Kommando in die Datei `.bashrc` einfügen:

```
11 > PATH=.:$PATH
```

Dieses Kommando weist der PATH-Variablen einen neuen Wert zu, der aus der Zeichenkette `.:` gefolgt vonm alten Inhalt der PATH-Variablen (bzeichnet durch $PATH) besteht. Das Kommando in Zeile 11 trägt also den Verzeichnisnamen Punkt `.` in die PATH-Variable ein, wobei der alte Inhalt der Variable erhalten bleibt.

Das folgende Kommando trägt ebenfalls den Verzeichnisnamen Punkt `.` in die PATH-Variable ein:

```
12 > PATH=.
```

Allerdings zerstört dieses Kommando den alten Inhalt der Variablen vollständig und nach seiner Ausführung kann man nur noch ausführbare Dateien wie gewohnt starten, die im aktuellen Arbeitsverzeichnis liegen. Das ist eine sehr unangenehme Situation, weil viele wichtige und nützliche Program-

me nicht im aktuellen Arbeitsverzeichnis, sondern in anderen Verzeichnissen liegen. Um sie zu starten, muss man jetzt nicht mehr nur ihren *Namen* (z. B. `gcc`) angeben, sondern ihren *absoluten Pfadnamen* (z. B. `/usr/bin/gcc`). Das Kommando in Zeile 12 sollte man also in aller Regel besser *nicht* direkt eingeben oder in eine Skriptdatei wie `.bashrc` eintragen.

Unter Windows-XP kann man Umgebungsvariablen verändern, indem man folgende Knöpfe/Menüpunkte anklickt/auswählt: *Start*, *Systemsteuerung*, *System*, *Erweitert*, *Umgebungsvariablen*. Unter anderen Versionen von Windows muss man ein bisschen anders vorgehen.

5. Mehrere Quelldateien compilieren und zu einem Programm zusammen binden

Ein komplizierteres C-Programm besteht in aller Regel nicht nur aus *einer* Quelldatei, sondern aus *mehreren* Quelldateien `qd01.c`, `qd02.c`, `qd03.c`, ... Um all diese Quelldateien zu compilieren und dann zu einer ausführbaren Datei zusammenzubinden, kann man etwa so vorgehen:

1. Man legt alle zusammengehörigen Quelldateien in ein Verzeichnis V.
2. Man legt keine anderen C-Quelldateien in das Verzeichnis V.
3. Man öffnet ein `bash`-Fenster und macht das Verzeichnis V zum aktuellen Arbeitsverzeichnis.

Das Kommando

```
1 > gcc *.c
```

macht aus allen C-Quelldatei im aktuellen Arbeitsverzeichnis eine ausführbare Datei namens `a.out` (wenn keine der Quelldateien formale Fehler enthält). Das Kommando

```
2 > gcc -o otto.exe *.c
```

leistet ganz Ähnliches, nennt seine Ausgabe (engl. output, die ausführbare Datei) aber nicht `a.out` sondern `otto.exe`.