Compilieren und Linken von C- und C++-Dateien

Lehrbeauftragter / Gastdozent an der Beuth Hochschule für Technik-Berlin

Internet: public.beuth-hochschule.de/~kempfer
www.kempfer.de
E-Mail: gerald@kempfer.de

Stand: 19. Februar 2011

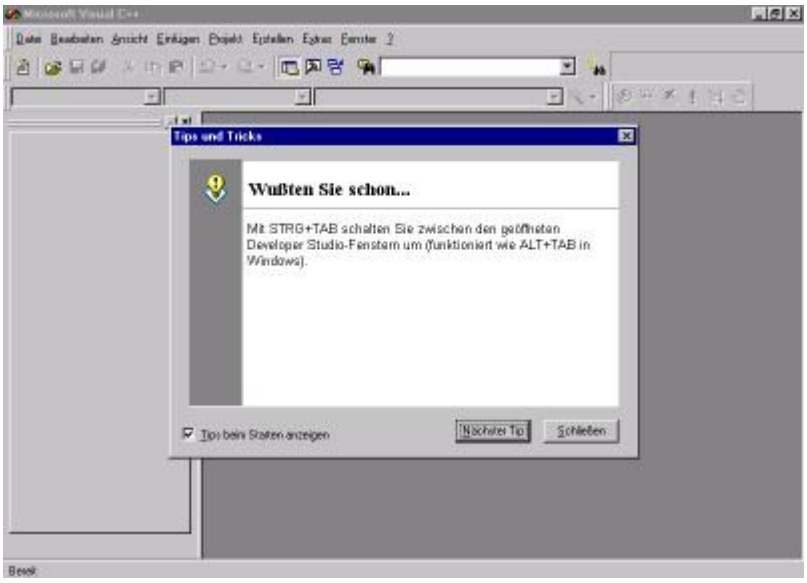
Inhaltsverzeichnis

1. MS Visual C++ (MS Windows)

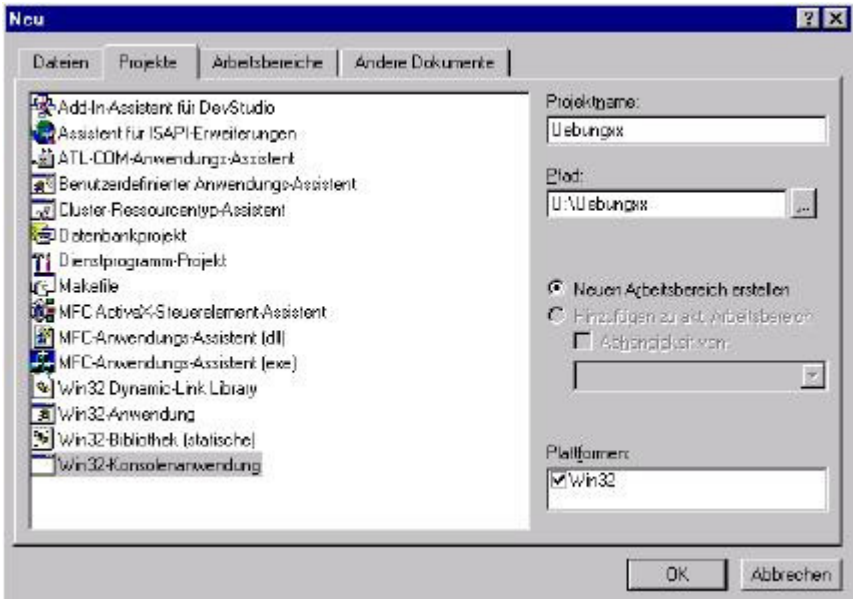
Mit der folgenden Anleitung legen Sie in MS Visual C++ 6.0 ein Projekt und eine Quellcode- bzw. eine Headerdatei an.

1.1. Anlegen eines leeres Projektes

Starten Sie - sofern noch nicht geschehen - das Programm MS Visual C++. Sollte ein Fenster mit dem Titel "Tips und Tricks" erscheinen, bestätigen Sie dieses durch Anklicken der Schaltfläche *Schließen*.

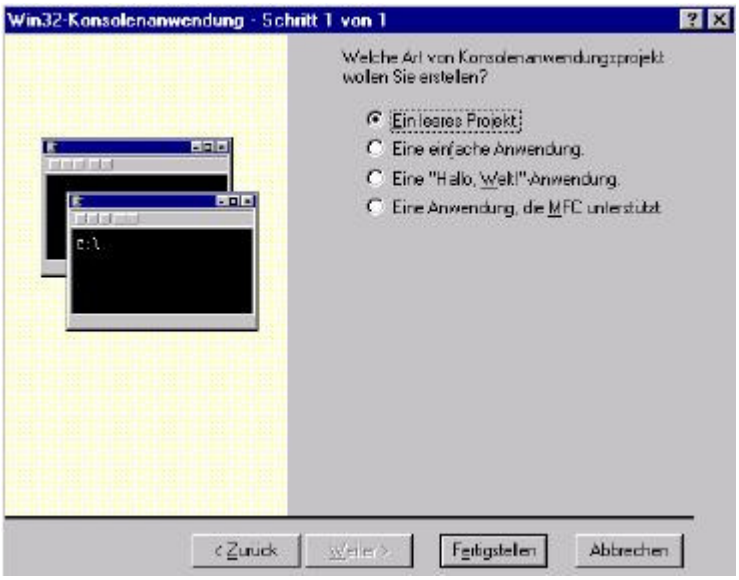


Wählen Sie den Menübefehl *Datei/Neu*. Das sich öffnenden Dialogfenster beinhaltet vier Register. Standardmäßig ist das zweite Register mit Namen *Projekte* zu sehen. Wenn nicht, wählen Sie das Register *Projekte*, in dem Sie auf den Reiter des Registers klicken.

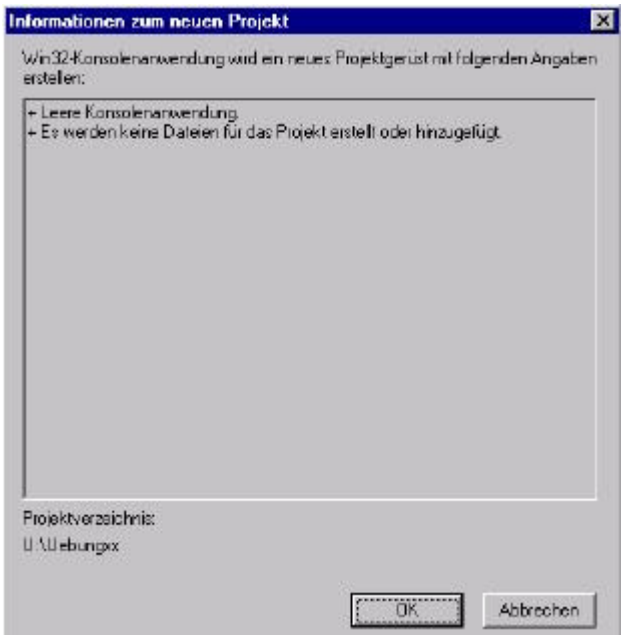


Im Register *Projekte* klicken Sie in der Auswahlliste auf der linken Seite den Punkt *Win32-Konsolenanwendung*. Geben Sie dann auf der rechten Seite im Feld *Projektname* einen Projektnamen ein, z.B. *Uebung01*. Darunter im Feld *Pfad* stellen Sie das Verzeichnis ein, in dem Ihr Projekt gespeichert werden soll. Voreingestellt ist im allgemeinen das Laufwerk C:. **Ändern Sie unbedingt das Laufwerk auf Ihr eigenes Laufwerk!** Die restlichen Einstellungen ändern Sie nicht. Klicken Sie dann auf die Schaltfläche *OK*.

Im nächsten Fenster wählen Sie die Option *Ein leeres Projekt* (sollte bereits voreingestellt sein) und klicken anschließend auf die Schaltfläche *Fertigstellen*.



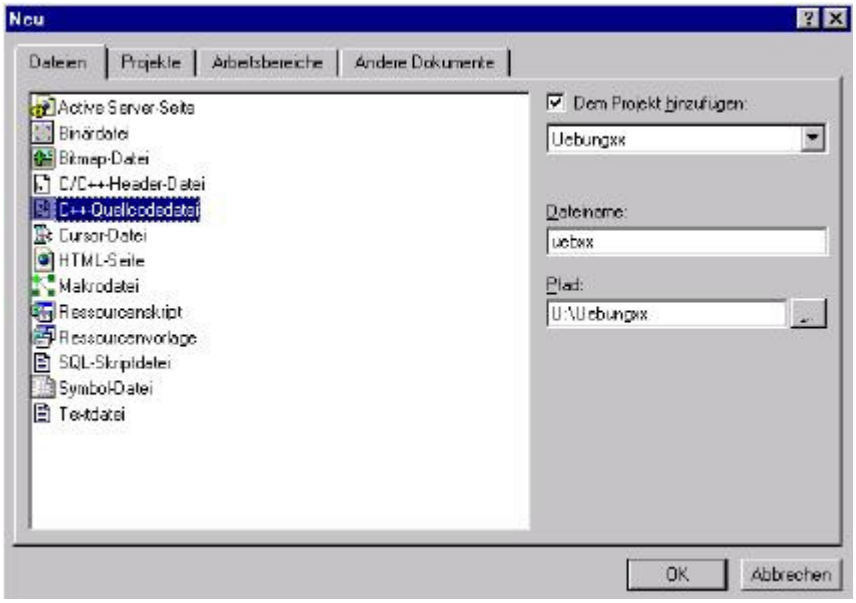
Es öffnet sich ein weiteres Fenster, in dem noch einmal alle Einstellungen angezeigt werden. Schließen Sie dieses Fenster durch Anklicken der Schaltfläche *OK*.



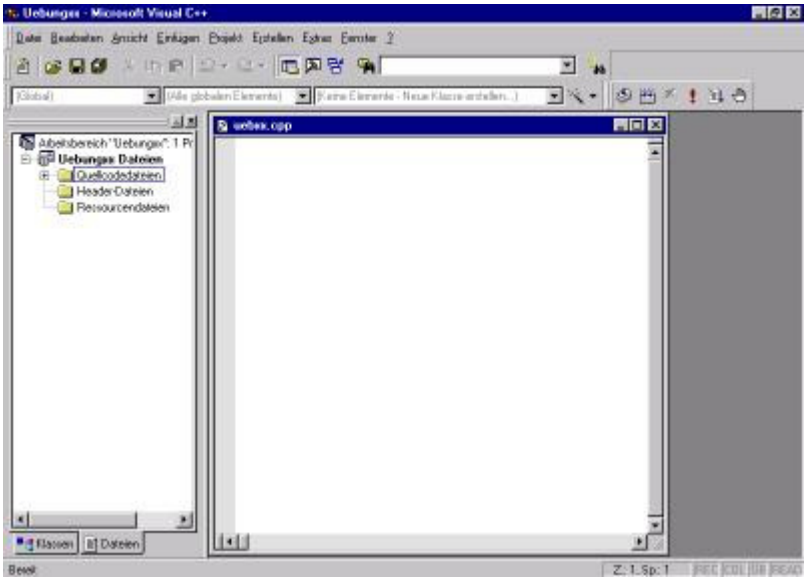
Nun haben Sie ein leeres Projekt angelegt. In diesem Projekt werden nun alle Quellcode- und Headerdateien abgelegt und verwaltet.

1.2. Anlegen einer Quellcode- oder Headerdatei

Um eine Quellcode- oder eine Headerdatei in dem geöffneten Projekt zu erzeugen, wählen Sie wieder den Menübefehl *Datei/Neu*. Diesmal wird das Register *Dateien* angezeigt (wenn nicht, wählen Sie das Register *Dateien*). Wählen Sie in der linken Auswahlliste entweder *C++-Quellcodedatei* oder *Headerdatei*. Überprüfen Sie, dass das Häckchen der Option *Dem Projekt hinzufügen* zu sehen ist und dass darunter der Projektname steht. Ferner muss der Pfad mit dem des Projektes übereinstimmen. Schließlich muss noch ein beliebiger Dateiname in dem gleichnamigen Feld eingetragen werden. Klicken Sie dann auf die Schaltfläche *OK*.



Nun ist die Quellcode- bzw. Headerdatei erzeugt. Sollen weitere Quellcode- oder Headerdateien verwendet werden, muss der vorige Schritt wiederholt werden.



1.3. Compilieren, Linken und Starten eines Programmes

In der Symbolleiste sind mehrere Schaltflächen, mit denen direkt das Compilieren (1. Symbol von links) und das Linken (2. Symbol von links) aufgerufen sowie das fertig compilierte Programm gestartet (4. Symbol von links) werden kann. Mit dem 6. Symbol von links lassen sich Haltepunkte im Programm setzen und wieder entfernen. Um das Programm nur von Haltepunkt zu Haltepunkt laufen zu lassen, wird das 5. Symbol verwendet.

2. GNU C-Compiler (gcc) und GNU C++-Compiler (g++)

Die C-/C++-Dateien sowie die Headerdateien sind reine Textdateien und lassen sich mit jedem beliebigen Texteditor bearbeiten (z.B. KWrite, Kate, KEdit, vi, ...). C-Dateien sollten die Dateiendung `.c` und C++-Dateien die Dateiendung `.cpp` erhalten, Headerdateien erhalten in C und in C++ die Dateiendung `.h`.

Außer dem Texteditor werden alle Befehle in einem Konsolenfenster (sowas ähnliches wie ein DOS-Fenster) eingegeben. Zum Kompilieren einer C-/C++-Datei wird folgender Befehl verwendet (vorausgesetzt wird, dass der GNU C-Compiler installiert ist):

`gcc -c` *`Dateiname`* für C-Dateien und
`g++ -c` *`Dateiname`* für C++-Dateien

Beispiele:
```
gcc -c test.c
g++ -c CPPTest.cpp
```

Hinweis: Unter Linux wird – im Gegensatz zu Windows – sehr genau zwischen Groß- und Kleinschreibung unterschieden!

Die Option `-c` gibt dabei an, dass die Datei (Dateinamen müssen immer mit der Dateiendung angegeben werden!) nur kompiliert aber nicht gelinkt wird. Es wird eine Objektdatei erzeugt (Dateiendung `.o`; unabhängig, ob C- oder C++-Programm).

Um aus den Objektdateien ein ausführbares Programm zu erzeugen, müssen die Objektdateien gelinkt werden:

`gcc` *`Objektdatei`* für C-Dateien und
`g++` *`Objektdatei`* für C++-Dateien

Beispiele:
```
gcc test.o
g++ CPPTest.o
```

Das erzeugte Programm ist die Datei `a.out`. Um selber festzulegen, wie das erzeugte Programm heißen soll, muss der Befehl noch um eine Option erweitert werden:

`gcc -o` *`Programmname Objektdatei`* für C-Dateien und
`g++ -o` *`Programmname Objektdatei`* für C++-Dateien

Beispiele:
```
gcc -o test test.o
g++ -o CPPTest CPPTest.o
```

Hinweis: Unter Linux müssen ausführbare Programme nicht unbedingt die Dateiendung `.exe` oder `.com` haben. Hier wurden also zwei Programme namens `test` und `CPPTest` (ohne irgendeine Dateiendung!) erzeugt.

Beide Schritte – das Kompilieren und das Linken – lassen sich auch zu einem Schritt zusammenfassen. Dafür wird beim letzten Befehl nicht die Objektdatei, sondern die C- bzw. C++-Datei angegeben.

Beispiele:
```
gcc -o test test.c
g++ -o CPPTest CPPTest.cpp
```

Um jetzt das Programm zu starten, muss einfach nur der Programmname in der Konsole eingegeben werden. Das Programm kann nicht über einen grafischen Dateimanager (z.B. Konqueror) gestartet werden.

Beispiele:
```
test
CPPTest
```

Manchmal reicht es aber nicht aus, nur den Programmnamen einzugeben. Dann ist unter Linux der Suchpfad nicht auf das entsprechende Verzeichnis eingestellt. Trotzdem lässt sich das Programm in der Konsole starten; es muss nur ein `./` vor dem Programmnamen eingefügt werden.

Beispiele:

```
./test
./CPPTest
```

3. Make-Dateien

Make-Dateien sind Textdateien, die – erst einmal ganz allgemein gesehen – eine Reihe von Konsolen-Befehlen beinhaltet. Dabei können Abhängigkeiten definiert werden, so dass diese Konsolen-Befehle in einer bestimmten Reihenfolge und nur in bestimmten Abhängigkeiten aufgerufen werden. Bestehen die Abhängigkeiten aus Dateien, wird geprüft, ob die entsprechende Datei existiert und wenn ja, ob Datum und Uhrzeit älter ist als das der Vergleichsdatei. Somit lassen sich überflüssige Operationen vermeiden. Z.B. braucht eine Quelltextdatei nicht compiliert werden, wenn die Quelltextdatei älter als die Objektdatei ist (weil dann hat sich seit dem letzten Compilieren nichts mehr verändert).

Gespeichert wird eine Make-Datei im allgemeinen unter dem Dateinamen `makefile`, es kann aber auch jeder andere Dateiname verwendet werden. Dann müssen Sie aber beim Aufruf des Make-Programms auch die Option `-f` verwenden und dahinter den Namen der Make-Datei angeben. Das Make-Programm wird wie folgt aufgerufen:

`make [-f` *`Dateiname_der_Make-Datei`*`]`

Eine Make-Datei kann im wesentlichen vier verschiedene Elemente enthalten; jedes Element kann mehrmals verwendet werden.

- Definitionen von Variablen und Funktionen (z.B. `CC = gcc -Wall`)
- Kommentare (z.B. `# Dies ist ein Kommentar!`)
- Includes (z.B. `-include Makefile.local`)
- Regeln (z.B. `%.o: %.c; -CC $<`)

3.1. Explizite Regeln

Eine Regel ist ganz allgemein wie folgt aufgebaut. Dabei sind die Elemente in eckigen Klammern optional.

```
Target [weitere Targets]:[:] [Vorbedingungen] [; Kommandos]
[<Tab> Kommandos]
[<Tab> Kommandos]
```

Wird ein Kommando angegeben, wird die Regel eine explizite Regel genannt; Regeln ohne Kommandos werden dagegen implizite Regeln genannt.

Wichtig ist, dass vor den Kommandos ab der zweiten Zeile ein Tabulator (`<Tab>`) verwendet wird. Dieser Tabulator darf vom Editor nicht in Leerzeichen umgewandelt werden!

Beispiel:

```
hallo:
   @echo Hallo Welt!
# Der Klammeraffe @ vor dem echo verhindert
# die Ausgabe des auszuführenden Befehls.

diskfree:
   df -h /
```

In diesem Beispiel werden zwei Regeln definiert und ordnet ihnen (explizit) jeweils ein Kommando zu. Diese Kommandos können nun gezielt aufgerufen werden:

```
$ make hallo
Hallo Welt!
$ make diskfree
df -h /
Dateisystem       Größe Benut  Verf Ben% Eingehängt auf
/dev/sda1           36G  9,3G   25G  28% /
```

Wird beim Aufruf kein Target angegeben, wird die erste Regel ausgeführt.

```
$ make
Hallo Welt!
```

Es können beim Aufruf auch mehrere Targets nacheinander angegeben werden. Diese werden dann in der angegebenen Reihenfolge ausgeführt.

```
$ make diskfree hallo
df -h /
Dateisystem       Größe Benut  Verf Ben% Eingehängt auf
/dev/sda1           36G  9,3G   25G  28% /
Hallo Welt!
```

3.2. Abhängigkeiten

Beim Compilieren und Linken gibt es Abhängigkeiten. So kann die Objektdatei `hallo.o` erst dann zum Programm `hallo.exe` gelinkt werden, wenn geprüft wurde, ob diese Objektdatei auch vorhanden ist und sie aktueller ist als die dazugehörige Quelltextdatei `hallo.c`. D.h. das Linken der Objektdatei erfordert das Compilieren der Quelltextdatei. In der Make-Datei wird diese Abhängigkeit wie folgt definiert:

```
hallo.o: hallo.c
   gcc -c hallo.c
```

Als Targetname wird die Objektdatei eingesetzt. Diese hängt von der Quelltextdatei ab. Ist die Quelltextdatei neuer als die Objektdatei, wird das darunterstehende Kommando ausgeführt, d.h. die Quelltextdatei wird compiliert und damit eine neue Version der Objektdatei erstellt. Wird diese Regel erneut aufgerufen, ohne dass die Quelltextdatei geändert wurde, ist die Objektdatei neuer als die Quelltextdatei und das Kommando wird nicht ausgeführt.

Nun wird eine ähnliche Abhängigkeit zwischen Objektdatei und Programmdatei erstellt:

```
hallo.exe: hallo.o
   gcc -o hallo.exe hallo.o
```

Als Targetname wird nun die Programmdatei eingesetzt. Diese hängt von der Objektdatei ab. Für die Objektdatei gibt es aber auch eine Regel (Abhängigkeit zur Quelltextdatei; siehe oben). Diese Regel wird zuerst abgearbeitet. Danach wird die aktuelle Regel weiter abgearbeitet: Ist nun die Objektdatei neuer als die Programmdatei, wird die Programmdatei neu erstellt (d.h. die Objektdatei wird gelinkt). Zusammen sieht die Make-Datei folgendermaßen aus. Dabei wird die Regel für die Programmdatei an erster Stelle gesetzt, damit die Abhängigkeiten auch dann funktionieren, wenn beim Aufruf von `make` kein Target angegeben wird.

```
hallo.exe: hallo.o
   gcc -o hallo.exe hallo.o

hallo.o: hallo.c
   gcc -c hallo.c
```

Im Kommando kann der Targetname über die Variable `$@` und die erste Vorbedingung über die Variable `$<` eingesetzt werden. Ferner kann der Compileraufruf über eine Variable geschehen. Sollte mal ein anderer Compiler zu Einsatz kommen, braucht der Compiler-Aufruf nur an einer Stelle geändert werden. Daraus ergibt sich folgende Make-Datei:

```
CC = gcc -Wall

hallo.exe: hallo.o
   $(CC) -o $@ $<

hallo.o: hallo.c
   $(CC) -c $<
```

Die Option `-Wall` beim Aufruf des Compilers bedeutet, dass alle Warnungen angezeigt werden. Bei existierender Quelltextdatei `hallo.c` (keine Fehler vorausgesetzt) wird beim Aufruf von `make` folgendes auf dem Bildschirm ausgegeben:

```
gcc -Wall -c hallo.c
gcc -Wall -o hallo.exe hallo.o
```

Ist ein Programm in mehrere Module unterteilt, bedeutet dies für die Make-Datei, dass die Programmdatei von mehreren Objektdateien abhängt. Die oben eingeführte Variable `$<` steht aber nur für die erste Vorbedingung, d.h. für die erste Objektdatei. Daher muss jetzt die Variable `$^` verwendet werden, in der alle Vorbedingungen enthalten sind. Da das Compilieren zusätzlich von Headerdateien abhängen kann, müssen diese dann in den Abhängigkeiten mit erwähnt werden.

makefile_3_1

```
01 CC = gcc -Wall
02
03 hallo.exe: main.o teil1.o teil2.o
04     $(CC) -o $@ $^
05
06 main.o: main.c teil1.h teil2.h
07     $(CC) -c $<
08
09 teil1.o: teil1.c
10     $(CC) -c $<
11
12 teil2.o: teil2.c
13     $(CC) -c $<
```

Es macht Sinn, noch weitere Targets zu definieren, die aber keinen Dateien entsprechen. Dabei haben sich drei Targets eingebürgert: `all`, `clean` und `run`. Mit `clean` werden die Programmdatei sowie alle Objektdateien gelöscht. Damit wird erzwungen, dass beim nächsten Aufruf von `make` alle Quelltextdateien neu compiliert und diese dann neu gelinkt werden. Mit `run` kann die fertiggestellte Programmdatei gestartet werden. Dies ist vor allem interessant, wenn der Aufruf des Programms komplexer ist (z.B. mit mehreren Parametern und in einem anderen Terminalfenster). Mit `all` werden nacheinander die Targets `clean`, der Name der Programmdatei und `run` aufgerufen. Damit wird alles neu compiliert und gelinkt und gleich das fertige Programm gestartet. Die Make-Datei sieht dann wie folgt aus:

makefile_3_2

```
01 CC = gcc -Wall
02
03 hallo.exe: main.o teil1.o teil2.o
04     $(CC) -o $@ $^
05
06 main.o: main.c teil1.h teil2.h
07     $(CC) -c $<
08
09 teil1.o: teil1.c
10     $(CC) -c $<
11
12 teil2.o: teil2.c
13     $(CC) -c $<
14
15 all: clean hallo.exe run
16
17 clean:
18     -rm *.o hallo.exe -f
19
```

```
20 run:
21     ./hallo.exe
```

3.3. *Implizite Regeln*

Die Regeln `main.o`, `teil1.o` und `teil2.o` im letzten Beispiel führen die selben Kommandos aus und unterscheiden sich nur in den Vorbedingungen. Diese Regeln können in zwei Teile aufgeteilt werden: In implizite Regeln, mit denen die Abhängigkeiten erklärt werden, und eine explizite Regel, bei der die Kommandos definiert sind. Daraus ergibt sich folgende Make-Datei:

```
01 CC = gcc -Wall
02
03 hallo.exe: main.o teil1.o teil2.o
04     $(CC) -o $@ $^
05
06 # explizite Regel definiert das Kommando
07 main.o teil1.o teil2.o:
08     $(CC) -c $<
09
10 # implizite Regeln definieren die Abhaengigkeiten
11 main.o:    main.c teil1.h teil2.h
12 teil1.o:   teil1.c
13 teil2.o:   teil2.c
14
15 all: clean hallo.exe run
16
17 clean:
18     -rm *.o hallo.exe -f
19
20 run:
21     ./hallo.exe
```

Die Liste der Objektdateien kann in einer Variablen gespeichert werden. Dadurch kann die Make-Datei problemlos um weitere Objektdateien erweitert werden. Dazu muss die Variable mit den Objektdateien erweitert werden und für die neue Objektdatei eine weitere implizite Regel eingegeben werden.

```
01 CC = gcc -Wall
02 Objektdateien = main.o teil1.o teil2.o
03 Programm = hallo.exe
04
05 $(Programm): $(Objektdateien)
06     $(CC) -o $@ $^
07
08 # explizite Regel definiert das Kommando
09 $(Objektdateien):
10     $(CC) -c $<
11
12 # implizite Regeln definieren die Abhängigkeiten
13 main.o:    main.c teil1.h teil2.h
14 teil1.o:   teil1.c
15 teil2.o:   teil2.c
16
17 all: clean $(Programm) run
18
19 clean:
20     -rm $(Objektdateien) $(Programm) -f
21
```

```
22 run:
23     ./$(Programm)
```