



Projekte verwalten mit make

Der große Macher



Keine bunten Bildchen - aber dafür effizient und leistungsfähig. Das Tool **make** ist auch im Zeitalter grafischer Oberflächen **nicht außer Mode**. Unter Unix ist es **Standard**, unter Windows häufig im Einsatz.

OLIVER MÜLLER

Wer Projekte verwalten muss, die nur aus einer Quelltextdatei bestehen, benötigt kein Programm, um den Build-Prozess zu steuern. Software-Projekte haben aber die unangenehme Eigenheit sehr schnell sehr komplex und umfangreich zu werden.

QUICK INDEX

- ▶ **Was macht make?**
make ist für die effiziente Steuerung von Kompilierung und Linken verantwortlich.
- ▶ **Beispielhaft**
Auf der Heft-CD finden Sie ein kleines Beispielprojekt.
- ▶ **make im Einsatz**
In der Befehlszeile von make sind mehrere Optionen möglich.
- ▶ **Sag mir, was zu tun ist**
Im Makefile wird festgesetzt was make machen muss.
- ▶ **Makrokosmos**
Über Makros können Makefiles einfacher gestaltet werden.
- ▶ **Implizierungen**
Mit impliziten Regeln können Sie die Makefiles weiter vereinfachen.
- ▶ **Pseudoziel**
Ein Ziel muss nicht immer eine Datei erzeugen.

Besser früher als später sollte man sich über die Verwaltung der Projekte Gedanken machen. Ein Weg sind integrierte Entwicklungsumgebungen (IDEs). Allerdings gibt es Momente, in denen IDEs nicht mehr einsetzbar sind. Gerade unter Unix/Linux und im Open-Source-Bereich sind IDEs eher die Ausnahme als die Regel. Das Tool **make** ist eine alt bewährte Möglichkeit, um Projekte zu verwalten.

■ Was macht make?

Wenn Sie sich ein Projekt ansehen, das aus mehreren Quelltextdateien besteht, erfolgt ein Build im wesentlichen in zwei Schritten:

1. Jede Quelltextdatei zu Objektcode kompilieren.
2. Die einzelnen Objektcodes und die benötigten Libraries zum ausführbaren Programm linkern.

Wurde bereits ein Build komplett durchgeführt, wird die Software getestet. Wird danach eine Änderung in einem Quelltext vorgenommen, ist es nicht immer nötig alle Quelltexte neu zu kompilieren. Einige haben

sich gar nicht geändert. Eine Neukompilierung dieser Module würde wieder den gleichen Objektcode erzeugen. Warum das also durchexerzieren?

Es genügt in der Regel nur die Quelltexte neu zu kompilieren, die wirklich geändert wurden. Die anderen Objektcodes der Quelltexte, die sich nicht geändert haben, werden einfach unverändert beim Linken hinzugefügt. Das spart Ressourcen, Zeit und Nerven.

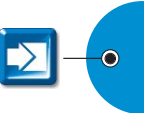
Die erste Aufgabe von **make** ist festzustellen, welche Quelltexte geändert wurden. Diese geänderten Module kompiliert **make** dann einfach erneut.

Zwischen den einzelnen Dateien eines Projekt können auch Abhängigkeiten bestehen. Eine erste Form der Abhängigkeit wurde bereits indirekt erwähnt. Wenn sich ein Quelltext ändert und dieser neu kompiliert wird, ändert sich zwangsläufig der Objektcode. Sobald sich ein Objektcode verändert, muss das ausführbare Programm neu gelinkt werden. Dieses Programm basiert auf dem Objektcode und damit indirekt auf dem geänderten Quelltext. Es ist von diesen Dateien abhängig.

Aber auch andere Formen der Abhängigkeit treten in Projekten auf. Stellen Sie sich vor, dass Sie eine Header-

VORDEFINIERTER MAKROS

Makro	Funktion
\$*	Ziel mit Pfad ohne Erweiterung.
\$@	Ziel mit Pfad mit Erweiterung.
\$<	Dateiname mit Pfad einer in Bezug auf das Ziel veralteten Datei (nur in impliziten Regeln verwendbar).
**	Liste der abhängigen Dateien.
?	Dateinamen mit Pfad der in Bezug auf das Ziel veralteten Dateien.
\$\$@	Zielname der gerade ausgewerteten Datei. (Nur in nmake.)



Datei von C++ oder eine import-Datei von Java verändern. Alle Quelltexte, die diese Dateien einbinden, müssen dann neu kompiliert werden - schließlich haben sich Deklarationen und/oder Definitionen geändert.

Wenn Sie sich jetzt ein kleines Projekt mit 20 Quelltextdateien vorstellen, werden Sie verstehen, dass dieser Vorgang per Hand nicht mehr zu erledigen ist. Trauen Sie sich wirklich zu, immer zu wissen, welche Dateien Sie neu kompilieren müssen, wenn ein Quelltext geändert wird? Ganz davon abgesehen, dass Sie sich als Tippakrobat beim Zirkus bewerben könnten, wenn Sie nur mal einen Monat ein solches Projekt durch Eingeben von einzelnen Compiler- und Linker-Aufrufen auf dem Kommando-prompt verwalten.

Mit make werden Ihnen diese Verwaltungsaufgaben abgenommen. In einem *Makefile* legen Sie fest, welche Dateien in welcher Art von anderen abhängen. Außerdem geben Sie in dieser Datei an, mit welchen Kommandos (Compiler-, Linker-Aufruf etc.) die Module verarbeitet werden.

■ Beispielhaft

Auf der Heft-CD finden Sie im Verzeichnis Beispiele / make einen Tarball für Linux und ein ZIP-Archiv für Windows. Darin enthalten ist ein kleines Beispielprojekt. Wir liefern Ihnen das berühmte "Hallo Welt!"-Programm in C++. Dieses obligatorische Beispiel wollen wir Ihnen in diesem Heft nicht als Beispiel zur Programmierung aufhalsen, für ein make-Beispiel eignet es sich jedoch recht gut.

Bei dem Beispiel kommt es ohnehin nur auf das Makefile an. Das Programm ist für diesen Beitrag nebensächlich. Wir benötigen lediglich etwas, was sich kompilieren und linken lässt. "Hallo Welt!" dürfte dabei keine Schwierigkeiten machen, egal welches Betriebssystem Sie einsetzen.

Das Projekt besteht aus einer C++-Klasse *HelloWorld*, die in *hellow.h* deklariert und in *hellow.cpp* implementiert wird. Die *main()*-Funktion liegt in der Datei *main.cpp*, welche die Header-Datei *hellow.h* einbindet. Die beiden C++-Quelltexte werden zu Objektcodes kompiliert und diese wiederum zum ausführbaren Programm gelinkt.

■ make im Einsatz

Das Makefile, mit dem make arbeitet, nennt sich üblicherweise entweder

Makefile, oder *makefile*. Es ist aber möglich die Informationen in einer Datei mit beliebigen Namen zu speichern. Diese Dateien sollten die Dateieindung *.mak* tragen.

Diese Dateien enthalten für die einzelnen Knoten eines Projektes je einen Eintrag, wie dieser Knoten erzeugt wird und von welchen anderen Knoten (bzw. Dateien) er abhängt. *Knoten* sind hierbei alle Dateien eines Projekts, wie Quelltexte, Objektdateien und ausführbare Programme. Diese Knoten hängen teilweise von einander ab. Ein ausführbares Programm setzt voraus, dass die Objektcodes, die gelinkt werden sollen, zuvor bereits kompiliert worden sind.

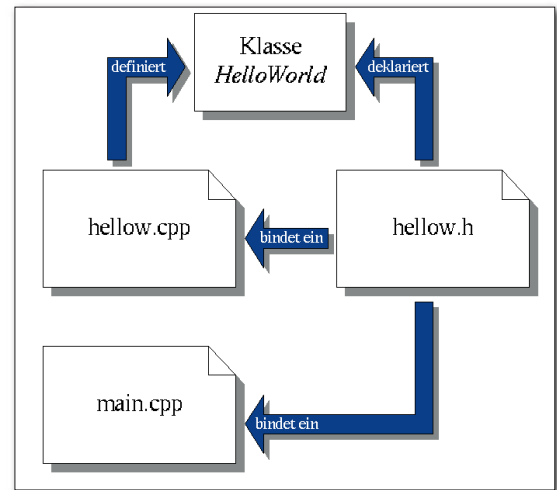
Um einen Build (=kompilieren und linken) zu starten, rufen Sie auf dem Kommando-prompt make unter Linux durch

```
make
```

bzw. unter Windows mit Visual C++ durch

```
nmake
```

auf. make sucht in diesem Fall nach der Datei Makefile bzw. *makefile* im aktuellen Arbeitsverzeichnis und verwendet die darin angegebenen Informationen.



DIE ABHÄNGIGKEITEN zwischen den Quelltexten im Beispielprojekt.

Als Knoten, der erzeugt werden soll, setzt make den ersten entsprechenden Eintrag im Makefile fest.

Wollen Sie ein anderes Makefile angeben, zum Beispiel *foo.mak*, so führen Sie es durch die Option *-f* bzw. */f* bei *nmake* an:

```
make -f foo.mak
```

bzw.

```
nmake /f foo.mak
```

In der Kommandozeile von make können Sie auch einen Wunsch-Knoten angeben, den Sie erzeugen wollen. Soll nicht der erste im Makefile aufgeführte Knoten kompiliert oder gelinkt werden, sondern der zweite, so können Sie beim

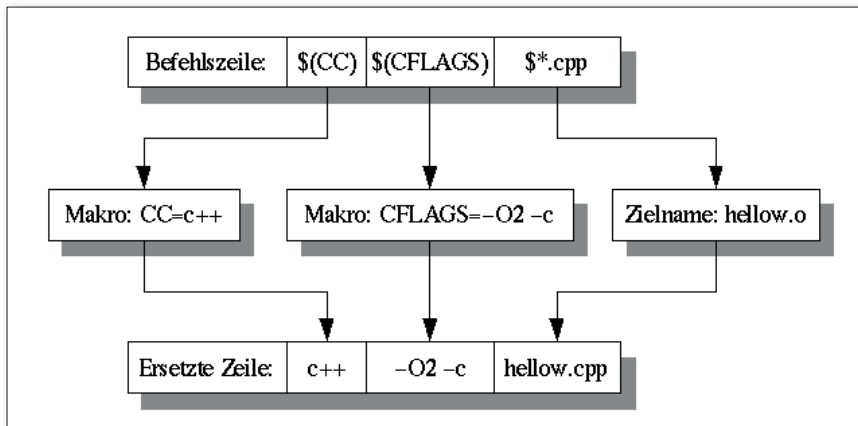
VERWENDETE KOMMANDOZEILENTOOLS

Befehl	System	Funktion
c++ -c -O2	Linux	GNU-C++-Compiler: Kompilierung zu Objektcode.
cl /c /G3 /GX	Windows	Visual-C++-Compiler: Kompilierung zu Objektcode.
c++ -o BIN	Linux	GNU-C++-Compiler: Linken zu Programmdatei BIN.
cl	Windows	Visual-C++-Compiler: Linken zu Programmdatei.

WEITERARBEITEN TROTZ FEHLER

In manchen Fällen kann es sinnvoll sein bei Fehlern weiterzuarbeiten. Angenommen Sie wollen ein umfangreiches Projekt mit Hunderten von Objektcodes über die Nacht kompilieren und am nächsten Morgen die Fehler ausmerzen. In diesem Fall wäre es schlecht, wenn beim Auftreten eines Fehlers beim Kompilieren des dritten Objektcodes make bereits abbricht. Hunderte von eventuell fehlerfrei kompilierbaren Quelltexten würden dann gar nicht kompiliert und kostbare

Zeit verschwendet. Geben Sie in diesem Fall einfach die Option *-k* bzw. */k* (*nmake*) beim Aufruf von make an. make versucht dann so viele Knoten wie möglich fertigzustellen, auch wenn ein Fehler auftritt. Das Linken wird wohl nicht stattfinden, aber Sie haben immerhin schon eine ganze Reihe von vorcompilierten Objektcodes. Beim Ausmerzen der Fehler können Sie sich viel Zeit sparen, da nur noch die fehlerhaften Quelltexte (neu-) kompiliert werden müssen.



SO FUNKTIONIEREN MAKROS und erleichtern dieTipparbeit.

Aufruf von make den anderen Knoten zur Bearbeitung angeben.

```
make Knoten2
```

bzw.

```
nmake Knoten2
```

erzeugt den zweiten Knoten.

make arbeitet normalerweise entweder bis der betreffende Knoten fehlerfrei erzeugt wurde, oder ein Fehler auftritt. Kann ein Quelltext nicht kompiliert werden, weil er einen Tippfehler enthält, dann hält make nach dem Auftreten dieses Fehlers an. Es wird nicht versucht dann noch zu linkern oder anderes anzustellen, da dies in der Regel nicht sinnvoll ist. Was soll gelinkt werden, wenn ein Objektcode nicht erstellt werden konnte?

Sag mir, was zu tun ist

Ein Makefile enthält Einträge für jeden Knoten. Jeder Eintrag ist dabei wie folgt aufgebaut:

```
Ziel : Abhängigkeiten
      Kommandos
```

Das Ziel benennt den Knoten und gibt die Datei an, die erzeugt werden soll. Eine Zielangabe darf niemals mit einem Leer- oder Tabulatorzeichen beginnen. Die Abhängigkeiten ist eine Auflistung von Dateien oder anderen Knoten, von denen das Ziel abhängt. Die Angabe Kommandos wird dabei durch mindestens ein Leerzeichen oder einen Tabulator eingeleitet. Sie kann aus einer Reihe von Zeilen bestehen oder in manchen Fällen komplett fehlen.

Angenommen Sie wollen das Beispielsprogramm *hellow* (Linux) bzw. *hellow.exe* (Windows) erzeugen. Zunächst kompilieren Sie die beiden Objektdateien aus den .cpp-Sourcen.

Unter Linux würden Sie hierfür die Kommandos

```
c++ -c -O2 hellow.cpp
c++ -c -O2 main.cpp
```

und unter Windows

```
cl /c /G3 /GX hellow.cpp
cl /c /G3 /GX main.cpp
```

verwenden. Sie erhalten daraufhin die Objektdateien *hellow.o* und *main.o* (Linux) bzw. *hellow.obj* und *main.obj* (Windows).

Im nächsten Schritt linken Sie die Objektcodes zu einem ausführbaren Programm (*hellow* bzw. *hellow.exe*) linken. Hierzu würden Sie die Kommandos

```
c++ hellow.o main.o -o hellow
```

bzw.

```
cl hellow.obj main.obj
```

verwenden.

Es existieren drei zu erzeugende Knoten:

- der Objektcode *hellow.o* bzw. *hellow.obj*,
- der Objektcode *main.o* bzw. *main.obj* und
- das Executable *hellow* bzw. *hellow.exe*.

In einem einfachen (nicht optimal genutzten) Makefile, ergeben sich drei Einträge. Dieses Makefile finden Sie als Listing 1 bzw. Listing 2. Sie sehen als Ziel die zu erzeugenden Knoten. Als Abhängigkeiten sind die Dateien angegeben, die benötigt werden, um das Ziel zu erzeugen.

Makrokosmos

Die Regeln in make können Sie durch Makros sehr flexibel gestalten. Wie Sie sehen, tauchen schon bei diesem kleinen Projekt Befehle mehrfach auf. Die Kompileraufrufe und die entsprechenden Optionen sind mehrfach angeben. Wenn Sie eine Option (zum Beispiel die Optimierungsstufe) ändern möchten, müssten Sie diese Änderung in jedem Kompileraufruf vornehmen. Stellen Sie sich ein Projekt vor, das aus 1000 Quellcodes besteht. Hier könnten Sie sich wieder als Spezialist für Tippakrobatik bewerben.

Über Makros können Sie sich diese Arbeit erheblich erleichtern und effizienter gestalten. Sie definieren einmal ein Makro und wenden diese dann mehrfach an. Anstatt eine Befehlszeile direkt einzutippen, binden Sie das Makro ein.

Makros erzeugen Sie mit make nach dem Schema

```
Makroname=Wert
```

Makronamen werden zumeist in Großbuchstaben geschrieben, damit Sie

LISTING 1

```
hellow: hellow.o main.o
        c++ hellow.o main.o -o hellow

hellow.o: hellow.cpp hellow.h
        c++ -O2 -c hellow.cpp

main.o: main.cpp hellow.h
        c++ -O2 -c main.cpp
```

LISTING 2

```
hellow: hellow.obj main.obj
        cl hellow.obj main.obj

hellow.obj: hellow.cpp hellow.h
        cl /G3 /GX /c hellow.cpp

main.obj: main.cpp hellow.h
        cl /G3 /GX /c main.cpp
```

LISTING 3

```
CC = c++
LD = c++
CFLAGS = -O2 -c
LDLFLAGS =
OBJ = hellow.o main.o

hellow: $(OBJ)
        $(LD) $(LDLFLAGS) $(OBJ) -o $@

hellow.o: hellow.cpp hellow.h
        $(CC) $(CFLAGS) $*.cpp

main.o: main.cpp hellow.h
        $(CC) $(CFLAGS) $*.cpp
```



Kommandos unterschieden werden können. Der Wert eines Makros ist eine Zeichenkette, die in den Einträgen anstelle des Makroaufrufs eingesetzt wird. Listing 3 bzw. 4 zeigen bereits die Kommandos, wie sie als Makros realisiert werden können. Der C++-Compiler wird als Makro CC festgelegt:

```
CC=c++
```

bzw.

```
CC=cl
```

Die Compilerflags werden nach dem gleichen Schema als CFLAGS definiert. Makros werden in die Regeln durch

```
$(Makroname)
```

LISTING 4

```
CC = cl
LD = cl
CFLAGS = /G3 /c /GX
LDFLAGS =
OBJ = hellow.obj main.obj

hellow.exe: $(OBJ)
    $(LD) $(LDFLAGS) $(OBJ)

hellow.obj: hellow.cpp hellow.h
    $(CC) $(CFLAGS) *.cpp

main.obj: main.cpp hellow.h
    $(CC) $(CFLAGS) *.cpp
```

LISTING 5

```
CC = c++
LD = c++
CFLAGS = -O2 -c
LDFLAGS =
OBJ = hellow.o main.o

.cpp.o:
    $(CC) $(CFLAGS) *.cpp

hellow: $(OBJ)
    $(LD) $(LDFLAGS) $(OBJ) -o $@

hellow.o: hellow.cpp hellow.h

main.o: main.cpp hellow.h
```

LISTING 6

```
CC = cl
LD = cl
CFLAGS = /G3 /c /GX
LDFLAGS =
OBJ = hellow.obj main.obj

.cpp.obj:
    $(CC) $(CFLAGS) *.cpp

hellow.exe: $(OBJ)
    $(LD) $(LDFLAGS) $(OBJ)

hellow.obj: hellow.cpp hellow.h

main.obj: main.cpp hellow.h
```

eingebunden. Der Compileraufruf zum Erstellen der Objektcodes ändert sich damit zu

```
$(CC) $(CFLAGS) *.cpp
```

Anstatt die .cpp-Datei direkt anzugeben, wird noch ein vordefiniertes make-Makro verwendet. \$* wird durch den Namen des Ziels ohne Erweiterung ersetzt. Heißt das Ziel hellow.o bzw. hellow.obj wird \$* von make zu hellow ersetzt. Indem jetzt .cpp angehängt wird, entsteht der Eintrag hellow.cpp, der Name des C++-Quelltextes.



Weitere vordefinierte Makros finden Sie im Kasten "Vordefinierte Makros".

Bei der Auswertung des Eintrags durch make wird der Ausdruck \$(CC) durch c++ bzw. cl ersetzt. Aus \$(CFLAGS) werden die Compilerflags, und \$* wird zum Zielnamen ohne Erweiterung. Damit entsteht zum Beispiel für hellow.o bzw. hellow.obj wieder das Kommando

```
c++ -c -O2 hellow.cpp
```

bzw.

```
cl /G3 /c /GX hellow.cpp
```

■ Implizierungen

Bislang haben Sie nur explizite Regeln bei den Einträgen kennen gelernt. Jeder Eintrag enthält eine explizite Angabe des Kommandos, das zum Erzeugen des Ziels verwendet werden soll.

Nun werden diese Kommandos immer wieder in gleicher Form übernommen. Bei jedem C++-Quelltext, der zu Objektcode kompiliert wird, wurde bis jetzt das gleiche Kommando immer und immer wieder angegeben. Diesen Aufwand können Sie sich durch implizite Regeln sparen. Sie teilen make mit, was zu tun ist, wenn eine Datei mit der Endung .cpp in ein Ziel .o oder .obj überführt werden soll. Sie schreiben hierfür

```
.cpp.o:
    $(CC) $(CFLAGS) *.cpp
```

bzw.

```
.cpp.obj:
    $(CC) $(CFLAGS) *.cpp
```



.cpp.o bzw. .cpp.obj sagt genau aus, was mit .cpp-Dateien gemacht werden soll, die in ein Ziel .o oder .obj führen sollen.

Bei den Einträgen lassen Sie jetzt die Kommandos weg. Damit ergeben sich die Listings 5 und 6. Bei einer Regel wie

```
main.o: main.cpp hellow.h
```

weiss make jetzt, dass die implizite Regel .cpp.o anzuwenden ist. Die erste Abhängigkeit für das Ziel main.o (=eine .o-Datei als Ziel) ist die Datei main.cpp (=eine .cpp-Datei als Quelle). Eine .cpp-Datei, die in ein .o-Ziel zu überführen ist.

■ Pseudoziel

Ein Ziel muss nicht immer eine Datei erzeugen. Die häufig verwendeten Aufräumziele in Makefile sind Beispiele für solche "Pseudoziele". Ein Eintrag

```
clean:
    rm -f *.o
```

bzw.

```
clean:
    del *.obj
```

löscht alle Objektdateien, wenn als Ziel in der Kommandozeile clean angegeben wird. Wie Sie sehen hat dieses Ziel keine Abhängigkeiten und erzeugt keine Datei (im Gegenteil: es löscht welche).

Pseudoziele können auch von anderen Zielen abhängen. Angenommen, Sie erzeugen zwei ausführbare Dateien server und client bzw. in Windows server.exe und client.exe. Für jede dieser Executables haben Sie einen eigenen Eintrag. Wenn Sie als erstes Ziel folgendes angeben:

```
all: server client
```

bzw.

```
all: server.exe client.exe
```

werden automatisch beide Dateien erstellt, wenn Sie make aufrufen.

