III.

Teil III: Praktische Methoden und Werkzeuge der Softwareentwicklung

17

Abläufe automatisieren mit make

Dieses Kapitel behandelt die folgenden Themen:

- Einführung in make
- Variablen und Muster
- Universelles Makefile für einfache Projekte

Bei der Programmentwicklung gibt es immer wiederkehrende Aufgaben. Darunter sind die Compilation und das Testen eines Programms besonders häufig. Andere häufige Aufgaben sind das Herstellen einer aktuellen Dokumentation, das Sichern von Dateien, Bauen einer Zusammenstellung[1] von Programm und Dokumentation zur Verteilung oder das Laden von Programmen auf einen Web-Server. Auch das Löschen nicht mehr benötigter Dateien gehört dazu. Diese Aufgaben in Einzelschritten per Hand durchzuführen, ist mühsam und fehleranfällig, weswegen schon früh dafür Werkzeuge ersonnen wurden. Der eigentliche Ablauf der Übersetzung und Erzeugung des lauffähigen Programms, Build-Prozess genannt, findet bei einer Entwicklungsumgebung (IDE) im Hintergrund statt. Programme wie *make*, die über die Kommandozeile aufgerufen werden können, erlauben das Bauen (englisch *build*) eines Programms auch ohne IDE.

make wird mit Regeln gesteuert, die sich in einer Textdatei, allgemein Makefile genannt, befinden. Im C/C++-Bereich gab und gibt es das Programm *make*, das auch für Win-

[1] Distribution genannt.

dows und Mac OS erhältlich ist. In der Java-Programmierung setzte sich das ebenso mächtige Tool *ant* durch, das von einer XML-Datei gesteuert wird wie auch MSBuild von Microsoft. Ein Makefile hat einen speziellen syntaktischen Aufbau. Es gibt mehrere »Dialekte«, die sich allerdings ähneln. Von den vielen make-Programmen ist GNU-make das Bekannteste [make]. Es bietet nicht nur einen großen Funktionsumfang, sondern ist im Gegensatz zu manch anderen Programmen auch noch portabel. Aus diesem Grund beschränke ich mich im Folgenden auf GNU-make, dessen Grundlagen und sinnvoller Einsatz kurz beschrieben werden. Aus Platzgründen werden nur die Funktionsweise, die wichtigsten Eigenschaften und konkrete Beispiele für den täglichen Gebrauch beleuchtet. Wer mehr wissen möchte, sei auf die angegebene Quelle [make] verwiesen und auf das Kapitel 23, das fortgeschrittene Techniken auf der Basis von make zum Inhalt hat. Unter Linux gibt es eine kurze Hilfe mit den Befehlen `man make` oder `info make`.

Makefiles sind Dateien, die Abläufe von Dienstprogrammen des Betriebssystems steuern. Dabei wird auf Datum und Uhrzeit der beteiligten Dateien geachtet, sodass überflüssige Operationen vermieden werden, zum Beispiel die Compilation einer längst übersetzten Datei, die nicht geändert wurde. Bei Änderungen werden also nur die davon betroffenen Dateien neu übersetzt – ein großer Vorteil! Dieser Mechanismus wird auch von Entwicklungsumgebungen genutzt, von denen manche direkt *make* benutzen und auch ein Makefile erzeugen, das man sich ansehen kann.

17.1 Quellen

Linux

Auf Linux-Systemen sind in aller Regel nicht nur der GNU-C++-Compiler, sondern auch *make* und andere Hilfsprogramme enthalten.

Windows

Es gibt für Windows Umgebungen, die eine Linux-Umgebung emulieren. Für die Beispiele dieses Buchs wird MinGW zusammen mit MSYS (*http://www.mingw.org/*) verwendet. MinGW stellt das Programm *make* und einen C++-Compiler zur Verfügung. Die Abkürzung MinGW steht für »Minimalist GNU for Windows«. Diese Umgebungen enthalten die wichtigsten Open Source-Werkzeuge zur Entwicklung von Software. Die IDE Code::Blocks benötigt die Installation von MinGW. Für die Funktionsweise einiger unten vorgestellter Skripte wird die Installation von MinGW und MSYS entsprechend der Anleitung auf der DVD vorausgesetzt. Damit steht Ihnen *make* mit all seinen Möglichkeiten zur Verfügung.

17.2 Wirkungsweise

Ein Makefile hat eine spezielle Struktur, in der die Begriffe *Ziel* (englisch *target*), *Abhängigkeit* (englisch *dependency*) und *Aktion* eine besondere Rolle spielen. Der syntaktische Aufbau ist wie folgt:

```
# Kommentar
Ziel1: Abhaengigkeiten1
→|Aktion1

Ziel2: Abhaengigkeiten2
→|Aktion2
# ... und so weiter
```

Tipp

Das durch →| symbolisierte Tabulatorzeichen kann bei manchen make-Programmen durch Leerzeichen ersetzt werden, nicht aber bei GNU-make!

Das Ziel kann (muss aber nicht) eine Datei sein, die erzeugt werden soll. Unter Abhängigkeiten werden die Dateien aufgelistet, von denen das Ziel abhängt, und die Aktion definiert, wie das Ziel erreicht wird. Alles, was in der Zeile nach dem →|-Zeichen folgt, wird einer Instanz des Kommandointerpreters des Betriebssystems übergeben. Dabei ist jede →|-Zeile unabhängig von den anderen!

Die nach dem Ziel notierte Abhängigkeit bezieht sich auf Datum und Uhrzeit der Dateien: Nur wenn eine der Dateien in der Liste *neuer* ist als das Ziel, wird die Aktion ausgeführt. Typische Aufrufe:

`make -f makedatei.mak ziel`	Ziel `ziel` der Datei *makedatei.mak* erzeugen.
`make -f makedatei.mak`	Das erste Ziel in *makedatei.mak* erzeugen.
`make ziel`	Ziel `ziel` erzeugen, die Steuerdatei heißt *GNUmakefile*, *makefile* oder *Makefile*.
`make`	Das erste Ziel erzeugen, die Steuerdatei heißt *GNUmakefile*, *makefile* oder *Makefile*.

Der Mechanismus sei am Beispiel der Klasse `Rational` von Seite 164 ff. gezeigt. Es gibt die Dateien *main.cpp*, *rational.h* und *rational.cpp*. Daraus soll die Datei *projekt.exe* als ausführbare Datei erzeugt werden. Wie Sie aus Kapitel 3 wissen (siehe Abbildung 3.9 auf Seite 124), müssen erst die Objektdateien *main.o* und *rational.o* erzeugt werden, die dann zum Endergebnis *projekt.exe* zusammengelinkt werden. Daraus ergibt sich die folgende Make-Datei, sie sei *m1.mak* genannt:

Listing 17.1: Einfache Make-Datei

```
# cppbuch/k17/m1.mak
# Regel 1
projekt.exe: rational.o main.o
→|g++ -o projekt.exe rational.o main.o
```

```
# Regel 2
rational.o: rational.cpp rational.h
→g++ -c rational.cpp

# Regel 3
main.o: main.cpp rational.h
→g++ -c main.cpp

# Regel 4
clean:
→rm -f rational.o main.o
```

make prüft zuerst, ob das erste Ziel *projekt.exe* (wenn kein anderes angegeben wird), überhaupt vorhanden ist. Wenn nicht, sucht *make* nach einer Regel zur Erzeugung dieses Ziels und findet Regel 1 mit der entsprechenden Aktion. Falls jedoch das Ziel existiert, werden Datum und Uhrzeit der Dateien rechts vom Doppelpunkt : geprüft, von denen das Ziel abhängt. Falls sie neuer als das Ziel sind, wird die Aktion ebenfalls ausgeführt, d.h. *projekt.exe* neu erzeugt.

Nun kann es aber sein, dass *main.o* oder *rational.o* noch nicht existieren. Dann sucht *make*, bevor die Aktion aus Regel 1 ausgeführt wird, nach einer Regel zur Erzeugung dieser Dateien und findet sie in den Regeln 2 und 3, deren Aktionen dann ausgeführt werden. Ein Aufruf `make -f m1.mak` würde die Datei *projekt.exe* erzeugen, verbunden mit der Bildschirmausgabe

```
g++ -c rational.cpp
g++ -c main.cpp
g++ -o projekt.exe rational.o main.o
```

Dem Parameter `-f` folgt der Dateiname des Makefiles. Wenn man ihn weglässt, sucht *make* nach den Dateien *makefile* oder *Makefile*. Die Wiederholung des Aufrufs würde nur die Meldung `make: "projekt.exe"ist bereits aktualisiert.` ergeben – schließlich hat sich ja auch nichts geändert. Falls nun zum Beispiel die Datei *rational.cpp* geändert wird, würden nur die Schritte

```
g++ -c rational.cpp
g++ -o projekt.exe rational.o main.o
```

ausgeführt; die erneute Übersetzung von *main.cpp* ist nicht erforderlich. Gerade bei großen, aus sehr vielen Dateien bestehenden Programmen erspart dieser Mechanismus viel Compilationszeit. *make* wählt sich stets das erste Ziel aus, es sei denn, ein anderes wird angegeben. So löscht der Aufruf

```
make -f m1.mak clean
```

die Objektdateien, die man abschließend nicht mehr benötigt und die bei Bedarf stets neu erzeugt werden können. Die Option `-f` unterdrückt Fehlermeldungen bei nicht existierenden Dateien. Mit dem Ziel *clean* ist eine Besonderheit verbunden: Es ist keine Datei, und es gibt auch keine Aktion, die so eine Datei erzeugt. *make* merkt das und legt die Datei nicht an. Man kann *make* die Prüfung ersparen, indem man die Zeile `.PHONY: clean` einfügt.

17.3 Variablen und Muster

Bei Projekten mit mehreren oder vielen Dateien ist es lästig, für jede cpp-Datei eine eigene Regel zu schreiben. Das haben die *make*-Entwickler erkannt: Die Regeln 2 und 3 oben können entfallen; sie werden von *make* durch *implizite Regeln* ergänzt. Dabei wird auch die Variable `CXXFLAGS` berücksichtigt. Das ist jedoch nicht ausreichend, wenn Sie dem Compiler bestimmte Informationen mittels weiterer Variablen mitgeben wollen.

Variablen haben den Vorteil, dass man die Definition ändern kann, ohne den Rest eines Makefiles bearbeiten zu müssen. *Muster* haben den Vorteil, nicht für jede Datei eine Regel schreiben zu müssen. Das Makefile *m2.mak* zeigt ein Beispiel, das nachfolgend erläutert wird:

Listing 17.2: Make-Datei mit Variablen und Mustern

```
# cppbuch/k17/m2.mak
.PHONY: clean

# Definition der Variablen
CXX := g++
CXXFLAGS := -c -g -Wall
objs := rational.o main.o

# Regel 1
projekt.exe: $(objs)
→$(CXX) -o $@ $^

# Regel 2
%.o: %.cpp rational.h
→$(CXX) $(CXXFLAGS) $<

clean:
→rm -f $(objs)
```

`CXX` ist eine Variable, die den Compiler bestimmt. Sie wird nachfolgend mit `$(CXX)` angesprochen. Damit kann nur durch Ändern dieser Variablen ein anderer Compiler benutzt werden. Der Name ist natürlich (relativ) frei wählbar, aber `CXX` ist üblich und bei GNU-make die Voreinstellung für den C++-Compiler. `CXXFLAGS` beschreibt an den Compiler zu übergebende Flags. Hier sollen außer der reinen Compilation (-c) Debug-Informationen erzeugt (-g) und alle Warnungen ausgegeben (-Wall) werden. `objs` gibt die zu erzeugenden Objektdateien an. Regel 1 zeigt die Verwendung der Dateien. Dabei sehen Sie auch merkwürdige Zeichen am Ende der Regeln 1 und 2 – dies sind von *make* bereitgestellte automatische Variablen mit den folgenden Bedeutungen:

- `$<` gibt die erste Abhängigkeit an.
- `$^` gibt alle Abhängigkeiten an.
- `$@` gibt das Ziel an.

Es gibt noch mehr mit dem $-Zeichen beginnende automatische Variablen, siehe [make]. Man kann sich die Werte leicht ausgeben lassen, indem zum Beispiel bei Regel 1 die Zeilen

```
	@echo 1 $^
	@echo 2 $@
	@echo 3 $<
```

eingefügt werden. Diese drei Zeilen bewirken nach Löschen aller Objektdateien und dem Aufruf `make -f m2.mak` die Ausgabe

```
1 rational.o main.o
2 projekt.exe
3 rational.o
```

Regel 2 zeigt die Verwendung eines Musters, mit dem die implizite Regel, cpp-Dateien mit `g++ -c` zu übersetzen, neu definiert wird. Die Zeile

```
%.o: %.cpp rational.h
```

bedeutet, dass jede Datei mit der Endung .o von einer ansonsten gleichnamigen Datei mit der Endung .cpp abhängt sowie von der Datei *rational.h*. An dieser Stelle %.h zu schreiben, wäre falsch, weil es in diesem Beispiel keine Datei *main.h* gibt und zur Erzeugung von *main.o* die implizite Regel zur Geltung käme. Hier gibt es einen kleinen Schönheitsfehler zu sehen: Typischerweise sind die cpp-Dateien in einem Projekt von verschiedenen Header-Dateien abhängig. Wie dieses Problem gelöst wird, sehen Sie in Abschnitt 23.1. Die Zeile

```
	$(CXX) $(CXXFLAGS) $<
```

bedeutet, dass die jeweilige cpp-Datei übersetzt wird. Diese Datei ist die erste Datei, von der das Ziel `%.o` abhängt, deswegen `$<` (siehe oben).

`SHELL` ist eine eingebaute Variable,[2] die mit */bin/sh* für den Kommandointerpreter voreingestellt ist, aber geändert werden kann.

17.4 Universelles Makefile für einfache Projekte

Im obigen Makefile mussten noch die Objektdateien angegeben werden. Schöner wäre ein Makefile, dass sich selbst die Informationen zusammensucht. Dies sei in *m3.mak* gezeigt. Dabei sei angenommen, dass sich alle Dateien zu diesem Projekt (aber keine anderen *.cpp-Dateien!) im aktuellen Verzeichnis befinden. Es darf in den cpp-Dateien exakt eine `main()`-Funktion geben.

Listing 17.3: Universelles Makefile für einfache Projekte

```
# cppbuch/k17/m3.mak
.PHONY: all clean
CXX := g++
CXXFLAGS := -c -g -Wall
```

[2] `make -p`, in einem Verzeichnis *ohne Makefile* aufgerufen, zeigt alle eingebauten Variablen und mehr.

```
LDFLAGS := -g
HEADERS := $(wildcard *.h)
objs := $(patsubst %.cpp,%.o,$(wildcard *.cpp))

all: projekt.exe

projekt.exe: $(objs)
	$(CXX) $(LDFLAGS) -o $@ $^

%.o: %.cpp $(HEADERS) # (noch) nicht optimal, siehe unten
	$(CXX) $(CXXFLAGS) $<

clean:
	rm -f $(objs)
```

Diese Datei unterscheidet sich von der vorherigen Datei *m2.mak* in folgenden Punkten:

- Es gibt eine Variable `LDFLAGS` für Informationen an den Linker.
- Es gibt eine Variable `HEADERS`, die mithilfe der Funktion `wildcard` gebildet wird, und die die Namen aller .h-Dateien des Verzeichnisses enthält.
- Die Variable `objs` wird mit Hilfe der Funktion `patsubst` gebildet. `patsubst` steht für »pattern substitution« und hat drei Parameter. `$(patsubst gesucht, ersatz, quelle)` bewirkt, dass jedes Auftreten von `gesucht` in `quelle` durch `ersatz` ersetzt wird. `objs` wird also gebildet, indem in der Liste der Namen aller cpp-Dateien die Endung .cpp durch .o ersetzt wird.
- Es gibt ein neues Ziel `all`, das nur von `projekt.exe`, der zu erzeugenden ausführbaren Datei, abhängt. Dies ist eigentlich nicht notwendig, folgt aber der Konvention, dass jedes Makefile ein Ziel `all` haben sollte, das die Übersetzung anstößt. Der Name `all` ist nicht ganz zutreffend, weil es andere Ziele geben kann, die gar nicht über `all` aktiviert werden. Zu dieser Gruppe gehören alle Ziele, die nicht so oft aufgerufen werden müssen, wie etwa das von oben bekannte Ziel `clean`. Es gibt weitere Ziele mit konventionellen Namen, die im obigen Makefile nicht enthalten sind. Eine Auswahl:
  - `install`: Das lauffähige Programm auf dem Computer (oder einem Web-Server) installieren.
  - `docs`: Die zugehörige Dokumentation erzeugen.
  - `check`: Tests des Programms laufen lassen.
  - `dist`: Eine Distribution erzeugen.
- Die zu erzeugenden Dateien hängen von der Variable `HEADERS` ab; es wird also alles neu übersetzt, wenn sich nur eine Header-Datei geändert hat – nicht optimal, aber bei kleinen Projekten tolerierbar. Im Abschnitt 23.1 sehen Sie, wie die tatsächlichen Abhängigkeiten automatisch ermittelt werden können.

Mit *m3.mak* liegt ein universelles Makefile vor, das Sie für einfache Projekte einsetzen können. Falls Sie die Grundlagen vertiefen möchten oder Tipps für spezielle Problemstellungen suchen, empfehle ich Ihnen einen Blick in das Kapitel 23.