Einsatz von Makefiles

Version: 1.9(01.10.2018)
Autor: Franz Regensburger

Voraussetzungen:
- Abarbeitung der Anleitung
  10_VirtuelleMaschine-Mint183.pdf
- Rudimentäre Kenntnisse der C-Programmierung
- Rudimentäre Kenntnisse im Umgang mit dem Editor vim

Einleitung

Nach der Installation der Entwicklungsumgebung stehen uns folgende Werkzeuge zur Verfügung:

1. Editor vim
2. C-Compiler gcc

Mehr ist prinzipiell zur Programmierung in C nicht notwendig.

Mit dem Editor erstellt man den Quellcode (die .c-Datei), und mit dem Compiler erzeugt man aus dem Quellcode eine ausführbare Datei.

Allerdings muss man diesen Übersetzungsvorgang nach jeder Änderung am Quell-Code ausführen und sich dabei auch jedesmal überlegen

- mit welchen Optionen man den Compiler ausführt
- welche C-Dateien neu übersetzt werden sollen
- mit welchen Bibliotheken gebunden werden muss
- mit welchen anderen kompilierten Dateien gebunden werden muss

Bei größeren Projekten, die aus vielen Quelldateien und auch aus vielen Bibliotheken bestehen, wird dieser Ablauf aber sehr schnell unübersichtlich und unhandlich.

Aus diesem Grund gibt es Build-Systeme, die anhand einer Spezifikation aller zum Projekt gehörenden Ressourcen (C-Quellen und Bibliotheken) und ihrer gegenseitigen Abhängigkeiten automatisch alle notwendigen Verarbeitungsschritte ausführen, um das fertige Produkt (zum Beispiel eine ausführbare Datei) nach einer Änderung neu zu erzeugen.

Das klassische Build-System für die C-Programmierung ist das Make-System, das mit sogenannten Makefiles arbeitet. Die GNU-Variante des Make-Systems ist in unserer virtuellen Maschine Mint-183 enthalten.

Ein sehr empfehlenswerter moderner Abkömmling von make ist zum Beispiel das Build-System cmake (http://www.cmake.org/), welches wir aber im Rahmen der Einführungsvorlesung nicht behandeln.

Ein einfaches Makefile

Im Folgenden stellen wir ein einfaches Makefile vor, in dem für die unterschiedlichen Projekte jeweils **nur eine Zeile** angepasst werden muss, damit die ausführbare Datei des Projekts richtig erzeugt wird.

Dieses Makefile funktioniert sowohl unter Linux/Unix als auch in der MinGW-Umgebung unter Windows.

Beispiel Programm: Hello World

Gegeben sei die C-Quelldatei hello.c

```
#include<stdio.h>

void main() {
     printf("Hello World\n");
}
```

Diese wird in einer Shell mit folgenden Kommandos in eine ausführbare Datei übersetzt, wobei die ausführbare Datei in das Unterverzeichnis **bin** gelegt wird.

```
$ mkdir -p bin                  (erzeugt bei Bedarf bin)
$ gcc -g -o bin/hello  hello.c  (kompiliert hello.c nach bin)
```

Das weiter unten vorgestellte Makefile erlaubt folgende bequemere und sicherere Benutzung in der Shell.

```
$ make                          (erzeuge das Executable aus C-Datei)

$ make clean                    (Cleanup: räume auf)
```

Die gezeigte Nutzung des Kommandos make wird durch die Erstellung des folgenden Makefiles möglich, welches im Projektverzeichnis in einer Datei mit dem Namen **Makefile** liegen sollte.

Im Projektverzeichnis (hier: ~/GdP1/CodeExamples/HelloWorld) befinden sich also zu Beginn nur zwei Dateien:

```
lars@vm00-16:~/GdP1/CodeExamples/HelloWorld
$ ls

hello.c  Makefile
```

Nach Ausführung des Kommandos **make** gibt es ein neues Unterverzeichnis **bin**, in dem das neu erzeugte Executable liegt.

Die Kommandos, die das Build-System Aufgrund der Spezifikation im Makefile ausführt, werden zur Information auf der Konsole ausgegeben. Somit sieht man immer, was das Build-System macht.

```
$ make clean
rm -rf bin
```

```
$ make
mkdir bin
gcc -g -Wall hello.c -o bin/hello
```

Unter Windows heißt das im Unterverzeichnis bin erzeugte Executable **hello.exe**, unter Unix einfach nur **hello**

Inhalt der Datei :~/GdP1/CodeExamples/HelloWorld/Makefile

```
############################################################
# General purpose makefile
#
# Works for all simple C-projects where
# - binaries are compiled into sub-dir bin
# - binaries are created from a single c-source of the same name
#
# Note: multiple targets (binaries) in ./bin are supported
#

# Please add all targets in ./bin here

TARGETS += $(BIN_DIR)/hello
TARGETS += $(BIN_DIR)/helloPerCharacter

################################################
# There is no need to edit below this line
################################################

# Generate debugging symbols?
CFLAGS = -g -Wall

#### Fixed variable definitions
CC = gcc
RM_DIR = rm -rf
MKDIR = mkdir
SHELL = /bin/bash
BIN_DIR = bin

####

all: $(BIN_DIR) $(TARGETS)

#### Fixed build rules
$(BIN_DIR)/% : %.c
	$(CC) $(CFLAGS) $< -o $@

$(BIN_DIR):
	$(MKDIR) $(BIN_DIR)

.PHONY: clean
clean :
	$(RM_DIR) $(BIN_DIR)
```

Für ein anderes Projekt, welches ebenfalls nur aus einer C-Datei besteht, muss nur in der rot markierten Zeile der Name der C-Datei geändert werden.

Achtung: in Makefile haben **Tabulator-Zeichen** spezielle Semantik Falls Sie oder Ihr Editor die Tabulator-Zeichen in Lehrzeichen umwandeln, funktioniert das Makefile nicht mehr!

Durch kleine Erweiterungen am Makefile können aber auch kompliziertere Projekte mit vielen Quelldateien und Header-Dateien unterstützt werden.

Derartige kompliziertere Makefiles werden zum Beispiel im C-Praktikum verwendet (Projekt Worm)

Ausblick

Für die Entwicklung von Software, die unter vielen unterschiedlichen Plattformen (Unix, Linux, Windows,…) lauffähig sein soll, wurden in den letzten Jahren leistungsfähigere Build-Systeme entwickelt, die automatisch auf die Plattform zugeschnittene Makefiles auf der Basis von Meta-Makefiles erzeugen können.

Ein Vertreter dieser neuen Generation von Build-Systemen ist das cmake-System (http://www.cmake.org/. Es unterstützt unter anderem Cross-Plattform-Entwicklung, die automatische Berechnung von Abhängigkeiten (dependencies) und die Erzeugung von Makefiles für unterschiedliche plattformspezifische Build-Systeme.

Im Rahmen der Einführungsvorlesung wäre die Verwendung von cmake jedoch zu kompliziert. Daher beschränken wir uns hier auf die Verwendung des klassischen Werkzeugs make.

Die für die Code-Beispiele und das Praktikum benötigten Makefiles werden im Allgemeinen vorgegeben.