make

Das Programm „make“ wird benutzt, um das Kompilieren eines Projektes zu vereinfachen. Um vom Quellcode zu einem ausführbaren Programm zu kommen, sind (vereinfacht gesagt) grundsätzlich zwei Schritte nötig:

1. Kompilieren + Assemblieren

   Der Quellcode wird zuerst zu Assemblercode kompiliert, dieser wird im Anschluss zu einer Objekt-Datei („.o“-Datei) assembliert. Das passiert für jede einzelne Quellcode-Datei des Projektes.

2. Linken

   Ein Projekt besteht häufig aus vielen unterschiedlichen Quellcode-Dateien. Nachdem im vorherigen Schritt alle in Objekt-Dateien übersetzt wurden, werden die einzelnen Objekt-Dateien zu einer einzigen ausführbaren Datei zusammen „gelinkt“.

Bei einer simplen Kompilierung wie z. B. durch „clang project.c“ wird die Aufteilung in die beiden Schritte vor dem Anwender versteckt und vom Compiler automatisch durchgeführt. Besteht das Projekt jedoch nicht aus einer sondern z. B. aus 1000 Quellcode-Dateien, würde ein Aufruf wie oben folgendes Problem darstellen: Bei jedem Kompiliervorgang würde man den Compiler dazu anweisen, alle 1000 Quellcode-Dateien neu zu kompilieren und assemblieren und anschließend zu linken. Hat man jedoch nur Änderungen an einer oder wenigen der Quellcode-Dateien durchgeführt, die man jetzt in seine ausführbare Datei übernehmen möchte, würden die übrigen Quellcode-Dateien trotzdem neu kompiliert, obwohl sie unverändert sind. Bei größeren Projekten wird das Kompilieren damit schnell sehr ineffizient und dauert jedes Mal sehr lange.

An dieser Stelle kommt „make“ zum Einsatz. make nimmt ein sogenanntes „Makefile“ als Eingabe, in welchem verschiedene Regeln definiert sind, die angeben wie ein Projekt gebaut werden soll. In diesem Makefile können Abhängigkeiten angegeben werden, von denen die Verarbeitung einzelner Datei abhängen. Für diese Abhängigkeiten prüft make dann, ob sie seit dem letzten Kompiliervorgang verändert wurden oder nicht und kompiliert diese nur erneut, falls es Änderungen gab.

Für den oben beschriebenen Problem-Fall heißt das:

1. Der Kompiliervorgang wird in die beiden oben beschrieben Schritte aufgespalten
2. Für das Kompilieren + Assemblieren der einzelnen Quellcode-Dateien sind die Quellcode-Dateien selbst die Abhängigkeiten und die dadurch erzeugten Objekt-Dateien die sogenannten „Targets“. Das heißt: eine Quellcode-Datei wird nur dann zu einer Objekt-Datei kompiliert, wenn die Quellcode-Datei seit dem letzten Kompilier-Vorgang verändert wurde.
3. Für das Linken sind alle Objekt-Dateien Abhängigkeiten. Der Link-Vorgang wird also nur ausgeführt, wenn eine dieser Objekt-Dateien seit dem letzten Aufruf verändert wurde (sprich sie wurde zwischendurch neu kompiliert).

Durch diese Verwendung von make werden also nur die Quellcode-Dateien erneut kompiliert, die seit dem letzten Mal auch tatsächlich verändert wurden, was den gesamten Vorgang effizienter macht.

Makefiles

In einem Makefile können die Regeln zum Bauen des Projektes definiert werden. Eine Regel hat dabei folgenden Syntax:

Target: Abhängigkeit
<Tab> Befehl
<Tab> Noch ein Befehl

Um aus einer Quellcode-Datei eine Objekt-Datei zu erzeugen, kann die entsprechende Regel so aussehen:

input.o: input.c

clang –c –o input.o input.c

1. Die Flag „-c“ sagt dem Compiler, dass er nur die Compile- und Assemble-Schritte ausführen soll, also, dass er eine Objekt-Datei erzeugen soll.
2. Die Objekt-Datei „input.o“ wird nur erzeugt, falls „input.c“ seit dem letzten Aufruf von make verändert wurde, da „input.c“ als Abhängigkeit für das Target „input.o“ angegeben ist.

Weiterhin können in einem Makefile Variablen definiert werden, die bei Verwendung einfach durch den ihnen zugewiesen Wert ersetzt werden:

VARIABLE=input

$(VARIABLE).o: $(VARIABLE).c

clang –c –o $(VARIABLE).o $(VARIABLE).c

1. Die Regel ist äquivalent zur vorherigen, da jedes Vorkommen von „$(VARIABLE)“ durch „input“ ersetzt wird.

Eine zweite sehr nützliche Funktionalität von make ist es, dass es das Kompilieren für den Anwender sehr vereinfachen kann. Deshalb muss nicht für jede einzelne Quellcode-Datei eine eigene Regel angegeben werden, sondern es können Platzhalter verwendet werden:

„%“: Pattern-Symbol, das benutzt werden kann, um ein Pattern für Targets und Abhängigkeiten anzugeben. Eine Regel die dieses Symbol in Targets und Abhängigkeiten enthält wird Pattern-Rule genannt, da diese Regel auf jedes Target angewendet wird, für das eine Datei, deren Name durch das von der Abhängigkeit definierte

Pattern gegeben ist, existiert. Dabei wird „%“ durch den Dateinamen ersetzt (siehe folgendes Beispiel).

„$@“: Makro für den Namen des Targets

„$<“: Makro für den Namen der ersten Abhängigkeit

Mit diesen Platzhaltern kann die obige Regel stark vereinfacht werden:

%.o: %.c

clang –c –o $@ $<

1. Durch die Verwendung von „%“ wird diese Regel zu einer Pattern-Regel. Das heißt, zuerst wird nach Zeichenfolgen gesucht, auf die das Target-Pattern „%.o“ zutrifft. Wird eine solche Zeichenfolge gefunden, z. B. „input.o“, wird überprüft, ob im aktuellen Verzeichnis eine Datei existiert, auf die das Abhängigkeits-Pattern zutrifft, also „input.c“. Existiert die Datei „input.c“ wird die Regel für das Target „input.o“ ausgeführt. Da diese Regel auf jedes passende Pattern ausgeführt wird, können somit mehrere Quellcode-Dateien mit einer einzigen Regel kompiliert werden.
   Hinweis: Damit das Pattern „%.o“ auch auf „input.o“ zutrifft muss die Zeichenfolge „input.o“, wie oben gesagt, im Makefile gefunden werden. Der String „input.o“ sollte deshalb zuvor im Makefile angegeben werden, z. B. als Teil eines Variablen-Wertes. Wird „input.o“ nicht im Makefile gefunden, wird die Regel auch nicht für dieses Target ausgeführt, sprich die Objekt-datei wird nicht erstellt, egal ob eine entsprechende Quellcode-Datei existiert oder nicht.
2. Für jedes gefundene Target auf das die Regel angewendet wird, kann jetzt „$@“ verwendet werden, um den Namen das Target, also z. B. „input.o“, zu benutzen. Das gleiche gilt für „$<“, wodurch die erste Abhängigkeit, in diesem Beispiel also „input.c“ gewählt wird und als Wert benutzt werden kann.

Nachdem mit der vorherigen Regel bereits sehr einfach viele Quellcode-Datei effizient kompiliert werden können, müssen diese im Anschluss noch zur Ausführbaren Datei gelinkt werden:

ausführbare_datei: input.o beispiel.o

clang –o $@ input.o beispiel.o

1. Eigentlich gibt es hier nichts neues: Das Target heißt „ausführbare_datei“, clang wird also angewiesen die Objekt-Dateien „input.o“ und „beispiel.o“ zu „ausführbare_datei“ zusammen zu linken. Da „input.o“ und „beispiel.o“ Abhängigkeiten sind, wird dies nur gemacht wenn mindestens eine der beiden Dateien auch verändert wurde.
2. **Es könnte sich hier lohnen „input.o beispiel.o“ in eine Variable auszulagern und stattdessen die Variable zu verwenden.**

Wenn man für sein Projekt die passenden Regeln zusammengestellt hat, schreibt man diese einfach in eine Textdatei, nennt sie „Makefile“ und kann durch einen simplen Aufruf von „make“ (im gleichen Ordner in dem sich das Makefile befindet) das gesamte Projekt einfach und effizient kompilieren lassen.