Module

Ein C-Programm geht durch mehrere Stadien um vom Quellcode zum ausführbaren Programm zu gelangen. Dies wird genutzt, um Programme aus mehreren Quelldateien zu erstellen, d. h. Programme, die Funktionen oder Variablen haben die in genau einer Datei definiert und in mehreren Dateien benutzt werden.

In jeder Datei, die auf solche Funktionalität zugreift, diese aber nicht definiert, muss eine Deklaration dieser benutzten Symbole stehen, ohne diese aber zu definieren. Dazu wird die Deklaration als `extern` markiert. Zur einfachen Verwaltung von „nach außen hin sichtbaren“ Symbolen, werden Header-Dateien verwaltet, welche diese Deklarationen enthalten und in allen Modulen, die diese verwenden, mit `#include` eingebunden werden müssen. Um zu vermeiden, dass bei mehrfachen Einbinden des gleichen Headers im Kompilationsprozess eine Variablen-, Typen- oder Konstentendefinition doppelt auftaucht, werden Header mit Include-Guards versehen:

```
1 #ifndef MODULNAME_H
2 #define MODULNAME_H
3
4 #define BUFFER_SIZE 1024
5 const int foo = 42;
6 struct bar {
7     int baz;
8 }
9
10 #endif // MODULNAME_H
```

Diese führen dazu, dass bei doppeltem Einbinden, der CPP den Inhalt jeder weiteren Einbindung ignoriert.

Ausschließlich intern-benutzte Symbole sollten mit `static` also solche markiert werden.

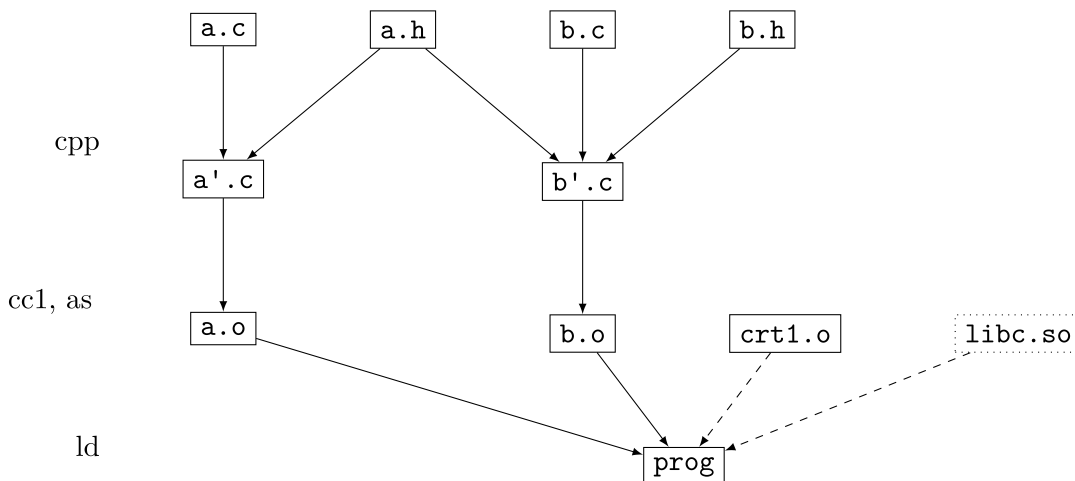

Abbildung 1: Beispiel: Programm aus mehreren Modulen

Beim Kompilationsprozess auf sog. Hosted-Umgebungen wird die C-Standardbibliothek bereitgestellt, diese enthält die entsprechenden Header, den eigentlichen Programmeintrittspunkt `_start` (in einer `crt0.o` /`crt1.o`) und die Implementationen der Funktionen (`libc.so`).

Während die Header explizit eingebunden werden müssen, aber – genauso wie alle anderen Header auch – nur bei der Kompilation eine Stütze bilden, werden die C-Runtime und die libC implizit hinzu-ge„linkt“, erstere statisch (d. h. sie ist komplett Teil des entstandenen Programmes), letztere i. d. R. dynamisch (die eigentliche Zusammenführung geschieht erst bei Ausführung des Programmes).

Makefiles

Makefile Struktur Makefiles automatisieren den Prozess der Programmerstellung, indem man Rezepte vorgibt, aus welchen Dateien (Prerequisites) wie (Rule) welche Resulte (Target) erstellt werden.

```
target1: prereq1 prereq2 ...
	rule # eingerückt mit Tab!

target2: ...
```

Hier wird *target1* mit Hilfe den Kommandozeilenbefehlen *rule* gebaut. Benötigt eine Regel vorher erstellte Dateien, wird nach einer Regel gesucht, diese zu bauen. Existieren diese Voraussetzungen nicht, werden sie mit der Regel erstellt. Existiert ein Target und auch seine Voraussetzungen, wird geprüft ob die Voraussetzungen neuer sind als das Target (also bspw. verändert wurden) und es wird ggf. das Target neu gebaut.

Aufruf von Make Make wird wiefolgt aufgerufen:

```
$ make [-einpqrst] [-f makefile]... [-k|-S] [macro=value...]
        [target_name...]
```

Wobei die Optionen `einpqrst`, `k` und `S` den Verlauf von Make modifizieren. Mit `-f makefile` kann ein spezifisches Makefile aufgerufen werden, sonst wird nach `makefile` oder `Makefile` gesucht. Normalerweise werden Makefiles mit großem „M“ geschrieben, da sie dadurch bei **ls(1)** direkt zu Beginn gelistet werden.

Es können bestimmte Variablen/Makros direkt beim Aufruf auf bestimmte Werte gesetzt werden, z. Bsp. um den C-Compiler `clang` zu forcieren:

```
$ make CC=clang
```

Das erste Target das im Makefile steht, wird gebaut. Dieses wird kanonisch „all“ genannt. Man kann ansonsten auch andere Targets explizit angeben, mit *target_name*.

0.1 Makefiles im Detail

Um das Schreiben zu erleichtern gibt es mehrere Mechanismen.

Implizite/Standard Variablen Anstatt sich festzulegen, dass man den Compiler `cc`, `gcc` oder `clang` benutzt, gibt es die Variable `CC`. Analoge Variablen gibt es auch für vieles andere:

CC C-Compiler

CFLAGS Optionen für den C-Compiler

CPPFLAGS Optionen für den C-Präprozessor, nicht C++!

CXXFLAGS Optionen für den C++-Compiler

Automatische Variablen Innerhalb einer Regel gibt es zusätzlich Variablen die für diese Regel bestimmte Werte enthalten, so z. Bsp.:

\$@ Target

\$^ Prerequisites

\$+ Prerequisites, in Reihenfolge der Listung und Doppelungen möglich

\$< Erste Prerequisite

Pattern-Regeln Man kann in Target und Prerequisites der Regeln den Platzhalter `%` benutzen, um Regeln zu bauen, die bspw. für bestimmte Dateiendungen gelten:

```
1 %.o: %.c
2     $(CC) $(CFLAGS) -o $@ $<
3 %: %.o
4     $(CC) $(LDFLAGS) -o $@ $+ $(LDLIBS)
```

Implizite Regeln Viele dieser Pattern-Regeln sind bereits eingebaut (bzw. äquivalente Suffix-Regeln, auf die wir nicht eingehen werden).

Spezielle Targets Es gibt Targets, die erstellen keine Datei. So werden häufig Targets `all` und `clean` definiert. Damit, falls zufällig eine Datei gleichen Namens existiert, die Regel trotzdem ausgeführt wird, sollten diese Targets als Prerequisite des speziellen Targets `.PHONY` gelistet werden.

Wenn Dateien als Zwischenergebnis von Regeln entstehen, dann werden sie standardmäßig gelöscht, sobald der Prozess abgeschlossen ist. Um das zu verhindern, sollten diese in `.PRECIOUS` gelistet werden.

0.2 Automatische Abhängigkeitsgeneration

Das Listen von Abhängigkeiten ist recht trivial, solange z. Bsp. aus einer C-Datei eine Objektdatei gebaut werden soll.

Jedoch möchte man eigentlich auch, dass ein Objekt neugebaut wird, ändert sich ein Header der von besagter C-Datei inkludiert wird – oder ein Header der von diesem Header wiederum eingebunden wird, etc. Anstatt aufwändig solche Listen selber zu Pflegen, unterstützen viele Compiler Make bei der Erstellung von Abhängigkeiten, hierzu müssen aber dem CPP spezielle Optionen übergeben werden.

-M Gebe für angegebene C-Datei eone Makeregel mit den Headern als Abhängigkeiten aus

-MM Selbiges, ohne Systemheader

-MD Kompiliere und schreibe nebenbei die C-Dateien in eine Datei

-MMD Selbiges, ohne Systemheader

-MF <Datei> Gibt Dateinamen an

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2 CPPFLAGS = -MMD -MF $*.d # Generate dependency files
3 LDLIBS = -lm
4 all: main
5
6 -include *.d # Include dependency files, ignore if non-existant
7
8 main: test.o
9 %.o: %.d # If dependencies have changed, recompile too
10 %.d: ;    # If a rule depends on a %.d but it doesn't exist, ignore
11 clean:
12     @$(RM) -v main *.o *.d
13 .PHONY: all clean
14 .PRECIOUS: *.o *.d
```

Codebeispiel 1: Beispiel für Makefile mit Abhängigkeitsgeneration

Modularisierung in Unterverzeichnisse

Wenn unsere Projekte weiter wachsen, ist es sinnvoll, diese in einer Ordnerstruktur abzulegen. Eine recht elegante Weise ist, ein Hauptmakefile zu haben, welches die Regeln enthält und eine Liste an zu erstellenden Dateien anzulegen, die man in jedem Unterordner erweitert.

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2 CPPFLAGS = -MMD -MF $*.d
3
4 all: main
5
6 include mod1/Makefile
7
8 main: $(OBJS) # build main from all OBJS (added in subdirs)
9 %.o: %.d # If dependencies have changed, recompile too
10 %.d: ; # if dep-file is missing (eg. on first run), don't panic
11 clean:
12     # delete object files and their deps as well as the executable
13     @$(RM) -v $(OBJS) $(OBJS:.o=.d) main
14 .PHONY: all clean
15 .PRECIOUS: *.d *.o
```

Codebeispiel 2: Beispiel für modulares Makefile: Hauptdatei

```
1 OBJS += mod1/mod1.o
2 include mod1/*.d
```

Codebeispiel 3: Beispiel für modulares Makefile: Untermodul

Versionsverwaltung mit Git

Git ist ein Versionsverwaltungssystem, das ist im Prinzip ein Protokoll der Änderung an eueren Dateien. Dies ermöglicht euch, Schnappschüsse vom aktuellen Zustand zu machen und zu diesen immer wieder zurückzukehren. Hierzu ist erstmal kein Server (GitHub, GitLab,...) notwendig! Zudem ermöglicht es unterschiedliche Entwicklungszweige parallel zu verwalten, für unterschiedliche Features oder, damit gleichzeitig mehrere Leute arbeiten können. Diese Änderungen können dann wiederum in einen Entwicklungszweig reinmigriert werden.

Aufbau Ein Git-Repository besteht aus einem „gerichteten azyklischen Graphen". Jeder Knoten in diesem Graphen ist ein *commit* und entspricht einem Schnappschuss, er zeigt auf seine(n) „Vorgänger", also die vorherige(n) Version(en). Man hat immer mindestens einen Entwicklungszweig, i. d. R. „master" genannt. Eine solche *branch* ist schlicht ein Zeiger auf den letzten Commit in diesem Zweig. Besonders wichtige Versionen kann man mit Labels versehen, anstatt dafür jedes mal eine weitere Branch zu machen, soetwas nennt man *tag*, diesen können auch Metadaten angefügt werden, z. Bsp. Release Notes (*annotated tag*). Auf den Commit, auf dem aktuell gearbeitet wird (*checked out*, zeigt der Zeiger *HEAD*.

Remotes Um Online zu arbeiten, benutzt man einen Server, welcher schlicht die gleichen Daten enthält wie ein lokales Git-Repository. Diesen Server nennt man *remote*. Möchte man die Änderungen (Commits) von der Branch **master** auf der Remote **origin** ein die eigene Branch integrieren, macht man dazu ein *fetch* und ein *merge*, oder kurz: *pull*. Analog, um Änderungen hochzuladen, ein *push*.

Wichtig ist, das Remote und Local divergieren können, also nicht eines aktueller ist als das andere. Git versucht die Änderungen so aufzulösen, dass alle Änderungen eingepflegt werden. Hierbei wird immer nur die lokale Branch modifizert, das Online-Repo sollte als „heilig" gelten und möglichst nicht bestehende Änderungen verändert, sondern nur angehängt werden! Bei Änderungen an unterschiedlichen Dateien, oder sogar der gleichen Datei aber in ganz unterschiedlichen Zeilen, kann Git i. d. R ohne Probleme Änderungen zusammenfügen. Funktioniert das nicht, muss manuell gemergt werden.

Merging Auch in anderen Fällen müssen Änderungen integriert werden, wenn zum Beispiel unterschiedliche Entwicklungszweige wieder zussamengeführt werden sollen. Explizit tut dies der Befehl *merge <branch>*.

Können die Änderungen nicht automatisch zusammengefügt werden, modifiziert Git alle von beiden Seiten veränderten Dateien, indem es beide Änderungen gleichzeitig, untereinander in die Datei einträgt, inklusive Mergetags. Es kann nun eine der beiden Varianten gewählt werden oder manuell beides zusammengeführt, die Mergetags entfehrt, committed, und dann weitergemacht werden.

Literatur

[1] wikipedia contributors. *Git*. 2018. URL: `https://en.wikipedia.org/wiki/Git`.

[2] wikipedia contributors. *Make (software)*. 2018. URL: `https://en.wikipedia.org/wiki/Make_(software)`.

[3] Free Software Foundation. *GNU make Manual*. 2008. URL: `https://www.gnu.org/software/make/manual/make.html`.

[4] Git project. *Git Reference*. 2018. URL: `https://git-scm.com/docs`.

[5] Ben Straub Scott Chacon. *Pro Git E-Book*. 2014. URL: `https://git-scm.com/book/en/v2`.