# GNU Make

Jörg Faschingbauer

# Table of Contents

FASCHINGBAUER

# Overview

FASCHINGBAUER

# Ursprung und Zielsetzung

FASCHINGBAUER

- Entwickelt (passiert?) Mitte der Siebziger in den Bell Labs
- UNIX besteht schon damals aus vielen Sourcefiles
- Immerwährende Frage: was muss ich machen, wenn ich ein File geändert habe?
- Erste Näherung einer Lösung: Shellscript (`find -newer ...` und dergleichen)
- $\rightarrow$ unwartbar!
- $\rightarrow$ Stuart Feldman: Make

# GNU Make (1)

FASCHINGBAUER

- Das Siebziger Make war aus heutiger Sicht featurefrei
- → Clones und Erweiterungen
- Microsoft `nmake`: genauso featurefrei
- BSD Make: in genauso vielen Varianten, wie es BSD-Versionen gibt
- GNU Make: heute die am weitesten verbreitete Variante
  - Richard Stallman Anfang der Achtziger
  - Sukzessive weiterentwickelt

# GNU Make (2)

FASCHINGBAUER

# GNU Make (3)

FASCHINGBAUER

# Overview

FASCHINGBAUER

# Abhängigkeiten

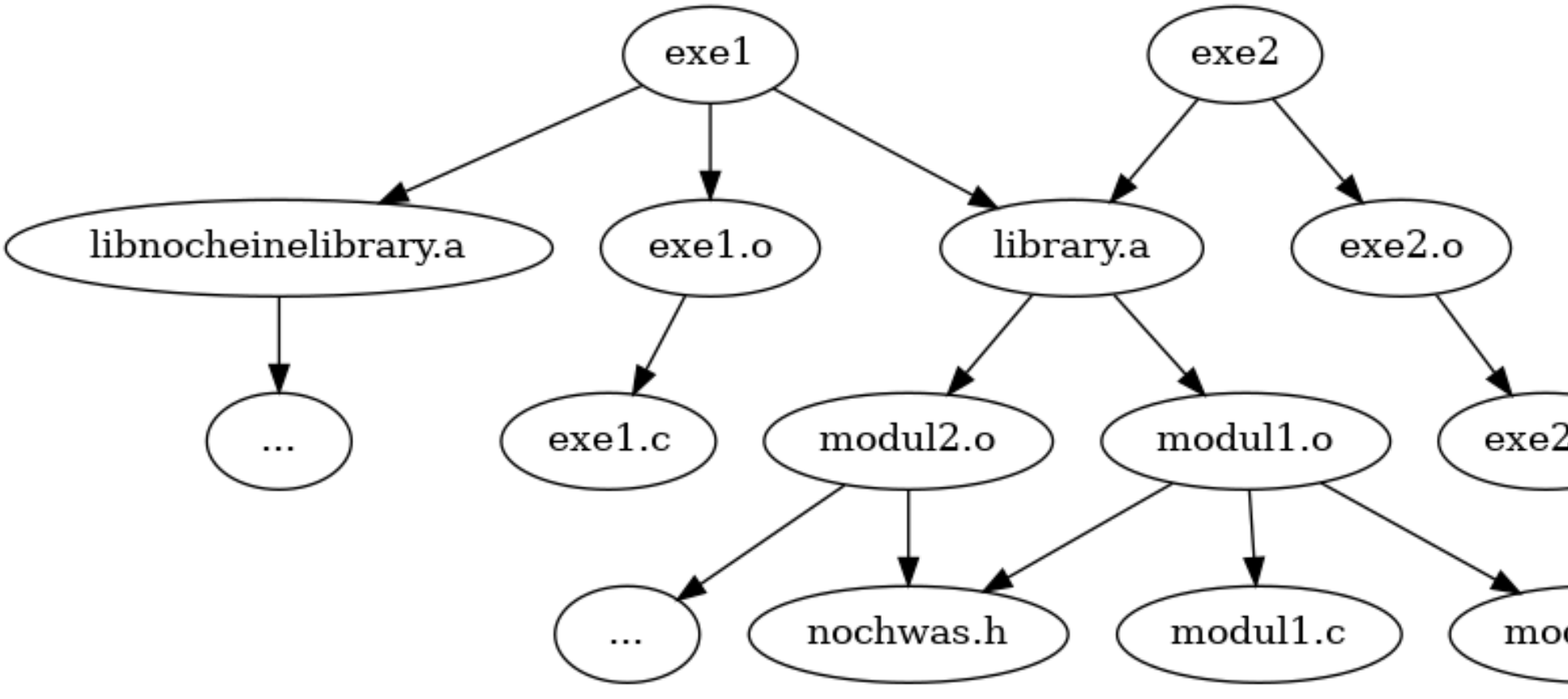
FASCHINGBAUER

- *Was muss ich machen, wenn ich ein File geändert habe?*
- Genauer: welche C Files includieren den Header, den ich gerade geändert habe?
- → diese müssen neu compiliert werden
- Object Files liegen nicht zum Spass herum
- → Libraries müssen als Folge neu archiviert werden
- → Shared Libraries müssen neu gelinkt werden
- → Executables müssen neu gelinkt werden
- ... und ... und ... und ...

→ *Abhängigkeitsgraph (Dependency Graph)*

# Dependency Graph

FASCHINGBAUER



# Die Programmiersprache *Make*

FASCHINGBAUER

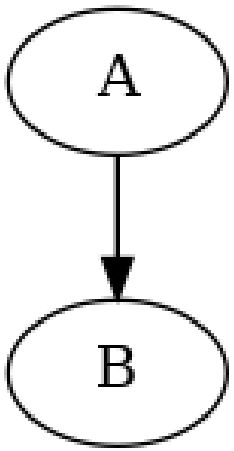
- Make baut einen Dependency Graph auf
- *"Hängt ab von"* $\Leftrightarrow$ *Timestamps*
- Spezifikation in Form von *Rules* ...
- ... in einem `Makefile`

**Challenge (und Thema des Kurses)**: Formulieren des Dependency Graph

- Eigene Programmiersprache
- $\rightarrow$ Probleme: Wartbarkeit, Struktur, ...

# Rules (1)

FASCHINGBAUER




**Einfache Rule**

```
A: B
        command
```

- A hängt ab von B
- Wenn B neuer als A ist, muss A neu gebaut werden
- A wird durch `command` neu gebaut
- Vorsicht: Einrückung durch *Tabulator*

# Rules (2)

FASCHINGBAUER

**Allgemein:**

- *Target*: Knoten, von dem Kanten ausgehen
- *Prerequisite*: Knoten, bei dem Kanten eingehen
- *Command*: Liste von Shell-Commands zum Bauen der Targets
- Mehrere Targets $\rightarrow$ implizit mehrere Rules

**Rule Syntax**

```
target ...: prerequisite ...
        command
        ...
```

# Commands

FASCHINGBAUER

- Fehler eines Commands → ganze Rule fehlerhaft
- Alle folgenden Rule fehlerhaft
- -: Ignoriert Fehler
- @: Command selbst wird nicht auf `stderr` geschrieben (nur sein Output)

## Commands

```
A: B
        @echo "Jetzt ist A dran" 1>&2
        -rm *.o
```

# Aufruf

FASCHINGBAUER

Oftgesehene Aufrufe (und Kombinationen davon) ...

- `make`: liest von `Makefile` im CWD
- `make -f MyMakefile`
- `make -C directory`: wechsle nach `directory` und mache dort
- `make mytarget`: baue nur `mytarget` und alles, was dazu nötig ist
- `make VARIABLE=value`: setze die Variable `VARIABLE` ($\rightarrow$ später)

# Feinheiten

FASCHINGBAUER

- Kein explizites Target per Commandline → erstes Target im Makefile ("Default Goal")
- Konvention: `all`
- Target ist kein File → `.PHONY`

## Realistisch ...

```
.PHONY: all
all: das-executable
das-executable: library1.a library2.a main.o
        gcc -o das-executable -lrary1 -lrary2 main.o
main.o: main.c header.h
        ...
```

# Overview

FASCHINGBAUER

# Übung: Mein Erstes Makefile

FASCHINGBAUER

Erstellen Sie ein Programm und ein dazugehöriges Makefile wie folgt. Achten Sie darauf, dass die Abhängigkeiten *lückenlos* spezifiziert sind!

- Das Programm (`main.c`) ruft nacheinander zwei Funktionen `f1()` und `f2()` auf. (Funktionalität ist beliebig und irrelevant.)
- Diese beiden Funktionen kommen aus einer statischen Library.
- Die beiden Funktionen sind in eigenen C Files (z.B. `f1.c` und `f1.c`) implementiert und werden durch dazugehörige Headerfiles exportiert.
- Sowohl `f1.c` und `f1.c` als auch `main.c` includieren die jeweiligen Headers, um zu den Funktionsdeklarationen zu gelangen.

# Overview

FASCHINGBAUER

# Rules: Duplizierter Code

FASCHINGBAUER

Uns fällt auf:

- Man will nicht für jedes Objectfile immer dieselbe Rule schreiben
- Beim Bauen der Library ist die Liste der Abhängigkeiten (Prerequisites) gleich der Members der Library

Abhilfe:

- Pattern Rules
- Implizit definierte Variablen (für den Anfang: `$@`, `$<`, `$^`)

# Pattern Rules, Variablen

FASCHINGBAUER

- Pattern instanziiert Rule für jedes `.c` File
- `$@`: Name des Targets ($\rightarrow$ `.o`)
- `$<`: Name der ersten Prerequisite ($\rightarrow$ `.c`)
- `$^`: alle Prerequisites

## Patterns, Variable

```
%.o: %.c
	gcc -c -o $@ $<
library.a: f1.o f2.o
	ar cr library.a $^
```

# Weitere Variablen

FASCHINGBAUER

- `$(MAKE)`: Name des Make Programms (für rekursive Aufrufe)
- `$(MAKEFLAGS)`: Optionen und Variablen (für rekursive Aufrufe)
- `$(CURDIR)`: Current working directory
- `$(SHELL)`: Verwendete Shell (falls man nicht `/bin/sh` will)
- u.v.a.m. $\rightarrow$ siehe Dokumentation

# Overview

FASCHINGBAUER

# Übung: Pattern Rules

FASCHINGBAUER

Schreiben Sie das Makefile von vorhin so um, dass es Pattern Rules und Variablen verwendet.

- Können Sie damit die Dependencies genauso lückenlos abdecken?
- Was fällt Ihnen auf? Welche Features benötigt man noch?

# Overview

FASCHINGBAUER

# Variablen: Zuweisung (1)

FASCHINGBAUER

Zuweisung an Variable immer toplevel → nicht Bestandteil einer Rule!

**Einfache Zuweisung**

- Bisheriger Wert wird überschrieben
- Bzw. Variable wird "definiert"

**Variablenzuweisung**

```
VARIABLE = value
all:
        echo $(VARIABLE)
```

# Variablen: Zuweisung (2)

FASCHINGBAUER

**Zuweisung, wenn noch nicht definiert**

- Variable kann von der Commandline kommen (explizit)
- → Defaultwert

**Defaultwert**

```
VARIABLE ?= defaultvalue
```

# Variablen: Expansionen sind rekursiv

FASCHINGBAUER

**Grundprinzip von GNU Make**:

- Variablen (in Make-Terminologie: "Macros") werden solange expandiert, bis nichts mehr zu expandieren ist.
- Vergleichbar mit M4
- Fast wie funktionales Programmieren

**Rekursive Zuweisung**

- $\rightarrow$ Endlosrekursion

**Rekursive Zuweisung**

```
VARIABLE = $(VARIABLE) nochwas
all:
        echo $(VARIABLE)
```

# Variablen: Noch mehr Zuweisungen

FASCHINGBAUER

**Nichtrekursive Zuweisung**

- ... quasi als Quick-Fix

**Addition**

- ... na Gottseidank!

**Nichtrekursive Zuweisung**

```
VARIABLE := $(VARIABLE) nochwas
```

**Addition**

```
VARIABLE += nochwas
```

# Variablen: Multi-Line Werte

FASCHINGBAUER

Für Programmierung interessant: Variablen enthalten Code

- → Funktionen
- → literale Zuweisung, incl. Linefeeds

## Variablenzuweisung als Funktionsdefinition

```
define object_rules
$(1).o: $(1).c:
        gcc -c -o $@ $<
endef
```

# Overview

FASCHINGBAUER

# Funktionen: Grundprinzip

FASCHINGBAUER

Einfaches Prinzip, aber verworrene Syntax

- `$(funname param1,param2,...)`
- Genau ein Space nach `funname`
- Kein Space vor und nach "," (ansonsten Teil des Parameters)
- *Expansion ergibt Text* $\rightarrow$ Wert einer Variable
- Expandiert, bis es nichts mehr zu expandieren gibt
- $\rightarrow$ "Rekursive" Expansion

# Textmanipulation

FASCHINGBAUER

Beliebige Textmanipulationen:

- `$(subst xx,bb,aaxxcc)` → aabbcc
- `$(patsubst %.c,%.o,f1.c f2.c)` → `f1.o f2.o`
- `$(strip a b c)` → `a b c`
- `$(filter %.c %.cc,f1.h f1.c f2.h f2.c f3.cc)` → `f1.c f2.c f3.cc`
- u.v.a.m. → Manual

# Filenamen

FASCHINGBAUER

Spielen mit Filenamen:

- `$(wildcard *.h *.c *.cc)` → wie Shell-Glob
- `$(basename f1.c dir/f2.c)` → `f1 dir/f2`
- `$(dir f1.c dir/f2.c)` → `./ dir`
- u.v.a.m. → Manual

# Kontrollfunktionen

FASCHINGBAUER

Das Wichtigste (fast) zum Schluss: Conditionals und Schleifen!

- `$(if CONDITION,THEN-PART[,ELSE-PART])`
- `$(or CONDITION1[,CONDITION2[,CONDITION3...]])`
- `$(and CONDITION1[,CONDITION2[,CONDITION3...]])`
- `false` $\Leftrightarrow$ leerer String
- `$(foreach VAR,LIST,TEXT)`

# Funktionen zum Selbermachen

FASCHINGBAUER

**Make ist eine Programmiersprache**

- Funktionsdefinitionen
- Funktionen sind nur Variablen
- → `call`, mit positionellen Parametern

## Funktion, Aufruf

```
define make_objects =
$(addsuffix .o,$(basename $(1)))
endef
OBJECTS = $(call make_objects,$(wildcard *.c *.cc))
$(info $(OBJECTS))
```

# Funktionen: Schlusswort

FASCHINGBAUER

**Wichtig**: Funktionen allein sind (relativ) wertlos!

- Sie expandieren zu einem String
- Den String kann man einer Variable zuweisen
- $\rightarrow$ es werden dadurch noch keine Rules dynamisch erstellt
- $\rightarrow$ `eval` (doch dazu später)

# Funktionen: Beispiele (1)

FASCHINGBAUER

## Listen von Files aus dem CWD

```
# C-like files in the current working directory
SOURCE_FILES = $(wildcard *.h *.c *.cc)

# filter out those that have to be compiled
COMPILED_FILES = $(filter %.c %.cc,$(SOURCE_FILES))

# once compiled, we have a list of object files
O = $(addsuffix .o,$(basename $(COMPILED_FILES)))

# see if we're fine
$(info $(O))
```

# Funktionen: Beispiele (2)

FASCHINGBAUER

## Ein String mit einer Rule drin

```
define object_rule =
$(basename $(1)).o: $(basename $(1)).c
        gcc -c -o $$@ $$<
endef
one_rule = $(call object_rule,f1.c)
```

*Achtung aufs Quoting → $$!*

# Overview

FASCHINGBAUER

# Eval: Inhalt von Variablen als Makefile-Inhalt

FASCHINGBAUER

**Das fehlende Glied:**

- Wir können mit Funktionen Variablen zusammenbauen
- Variablen können beliebigen Makefile-Code beinhalten
- `eval` expandiert eine Variable und fügt das Resultat *als Makefile-Fragment* ein
- Der expandierte Wert der Variable selbst ist leer → keine Seiteneffekte
- → *Make ist dynamisch*

# Beispiel

FASCHINGBAUER

## Dynamische Rules

```
define object_rule
$(1).o: $(1).c
        gcc -c -o $$@ $$<
endef
$(foreach stem,$(basename $(wildcard *.c)),\
      $(eval $(call object_rule,$(stem))))
```

# Overview

FASCHINGBAUER

# Übung: Funktionen, Variablen, Eval

FASCHINGBAUER

Ersetzen Sie die Pattern-Rules in Ihrem Makefile durch Rules, die von `eval` erzeugt werden!

# Overview

FASCHINGBAUER

# Includieren von Makefile-Fragmenten

FASCHINGBAUER

Mit `include` wird der Inhalt eines Files als Makefile-Syntax interpretiert ...

- File existiert nicht → Warnung und weiter
- Nach Ende wird versucht, alle fehlgeschlagenen Includes nachzuholen
- Versucht, das includierte File zu erzeugen → Rule muss vorhanden sein
- Erst dann Fehler

# #include-Dependencies

FASCHINGBAUER

- Händisch Dependencies auf Headerfiles zu pflegen, ist fehleranfällig (unmöglich?)
- Compiler muss Headerfiles includieren, um compilieren zu können
- Er weiss es am besten, von welchen Files ein Objectfile abhängt
- $\rightarrow$ Warum lässt man nicht *ihn* die Dependencies generieren?

# Generieren von Dependencies mit GCC

FASCHINGBAUER

## Zum Beispiel so ...

```
include f1.d
f1.d: f1.o
f1.o:
        gcc -M f1.c > f1.d
        gcc -c -o $@ $<
```

# Overview

FASCHINGBAUER

# Übung: Automatische Dependencies

FASCHINGBAUER

Ändern Sie Ihr Makefile dahingehend, dass die Rule-generierende Funktion auch die bis jetzt fehlenden Dependencies generiert!

# Overview

FASCHINGBAUER

# Schlusswort (1)

FASCHINGBAUER

**Make ist für Hartgesottene, denn:**

- Es ist innen schön (weil konsistent) und aussen hässlich
- Es ist von Natur aus schwer, Dependencies lückenlos zu formulieren
- Spätestens bei parallelem Build (`make -j`) *müssen* Dependencies lückenlos sein
- *Assembler des Software-Build*

**Aber:**

- Es ist mächtig
- Auch ausserhalb von C/C++ anwendbar

# Schlusswort (2)

FASCHINGBAUER

**Higherlevel Tools:**

- Autotools
  - "GNU Build System"
  - `http://sourceware.org/autobook/autobook/autobook_toc.html`
  - `http://nostarch.com/autotools.htm`
- CMake, `http://www.cmake.org`
- → *Beide generieren Makefiles*
- GNU Make Standard Library (GMSL)
  - Geniales Musterbeispiel für Make-Programmierung
  - http://gmsl.sourceforge.net/