

# Makefiles und git

## Automatisierung des Build-Prozesses

# Der Build-Prozess

Aufbau:

1. Übersetzung der Quelldateien in Objektdateien
2. Linken der Objektdateien zu ausführbaren Dateien / Bibliotheken
3. Einbinden von externen Bibliotheken
4. (optional) automatisches Erkennen von geänderten Quelldateien bzw. nur partielles Rekompilieren

# make

- ▶ 1977 entwickelt, seit dem mehrere Implementationen
- ▶ „Kochrezept“
- ▶ In POSIX standardisiert, unter Linux setzen wir jedoch **GNU Make** ein, welches deutlich mehr Features hat
- ▶ Bei größeren Projekten lässt man die sog. **Makefiles** oft automatisch generieren.
- ▶ *Wir schreiben unsere Makefiles von Hand!*

# Regeln I: Grundlagen

Ein schlechtes Beispiel:

```
1 main: # Target
2 cc -std=c99 -Wall -Wextra -pedantic -o main main.c # Command
```

Target:

- ▶ „Name“ der Regel
- ▶ i. d. R. die Datei die durch das Command erstellt wird
- ▶ Kann mit `$ make <Target>` ausgeführt werden

## Regeln II: Variablen

Ein (etwas) besseres, äquivalentes Beispiel:

```
1 CC=gcc
2 CFLAGS=-std=c99 -Wall -Wextra -pedantic
3 main:
4 $(CC) $(CFLAGS) -o main main.c
```

Standardvariablen:

**CC** Der C-Compiler, i. d. R. `cc`, nur Überschreiben wenn nötig!

**CFLAGS** Parameter für Aufruf eines C-Compilers

**CPPFLAGS** Parameter für den C-Präprozessor

**CXXFLAGS** Parameter für Aufruf eines C++-Compilers

## Regeln III: Voraussetzungen

Ein (wieder etwas) besseres Beispiel:

```
1 CFLAGS=-std=c99 -Wall -Wextra -pedantic
2 main: main.c # Prerequisite
3 $(CC) $(CFLAGS) -o main main.c
```

Voraussetzungen:

- ▶ „Zutaten“ für die Regel
- ▶ Falls diese sich ändern, gilt das Target als veraltet und wird neu kompiliert

## Regeln IV: Automatische Variablen

Ein (noch etwas) besseres Beispiel:

```
1 CFLAGS=-std=c99 -Wall -Wextra -pedantic
2 main: main.c
3 $(CC) $(CFLAGS) -o $@ $^ # Automatic Variables
```

Automatische Variablen:

$\$@$  Name des Targets

$\$^$  Alle Prerequisites

$\$+$  Ähnlich, jedoch Mehrfachlistungen möglich und Reihenfolge wird eingehalten

$\$<$  Erste Prerequisite

## Regeln V: Pattern-Regeln & Abhängigkeiten

Ein (noch) besseres Beispiel:

```
1 CFLAGS=-std=c99 -Wall -Wextra -pedantic
2 all: main # Default
3 %.o: %.c # Pattern for object files
4 $(CC) $(CFLAGS) -c -o $@ $<
5 %: %.o # Pattern for linking executable
6 $(CC) $(LDFLAGS) -o $@ $^
7 .PHONY: all # all is a special rule (does not produce file "all")
```

Pattern & mehrere Regeln:

- ▶ Erste Regel (Standard) hat als Voraussetzung, dass main existiert
- ▶ Die Regel dafür setzt voraus, dass main.o existiert *und sich seit dem nicht geändert hat*
- ▶ Analog setzt main.o main.c voraus.

## Regeln VI: Implizite Regeln

Ein „perfektes“ (aber nicht komplettes) Beispiel:

```
1 CFLAGS=-std=c99 -Wall -Wextra -pedantic
2 all: main # Rule for main is implicit
3 .PHONY: all
```

Einige implizite Regeln:

- ▶ Objektdatei `%.o` aus `%.c`: `$(CC) $(CPPFLAGS) $(CFLAGS) -c`
- ▶ Ausführbare Datei `%` aus `%.o`: `$(CC) $(LDFLAGS) n.o $(LDLIBS)`

## Regeln VII: Kompletteres Beispiel

Ein kompletteres Beispiel:

```
1 CFLAGS=-std=c99 -Wall -Wextra -pedantic
2 LDFLAGS=-lm # Link with libm.a (maths)
3 all: main
4 main: test.o # additional object used for main, ie. main = main.o + test.o
5
6 clean: # Delete build files
7   @$(RM) -v *.o # @: Silence output, -v: list removed files
8 .PHONY: all clean
9 .PRECIOUS: *.o # Don't delete intermedita *.o files
```

Weiteres: wie gehen wir mit Headern um?

# Header I

Problem:

- ▶ Header sind eigentlich auch „Zutaten“, deren Änderung die Kompilate veralten lassen können
  - ▶ Wenn eine Quelldatei einen neuen Header inkludiert müssen wir das Makefile erneut anpassen
- ⇒ Präprozessor soll uns sagen, welche Header inkludiert wurden – `make` testet dann auf Änderungen

## Header II

Präprozessoroptionen (GNU Toolchain):

- M Gebe Make-Regeln auf der Konsole aus
- MM Selbiges, ohne Systemheader
- MD Kompiliere ganz normal, schreibe die Informationen in Datei
- MMD Selbiges, ohne Systemheader
- MF <Datei> In Kombination mit Obigem: Schreibe in diese Datei

Beispiel:

```
1 $ gcc -MM main.c
2 main.o: main.c test.h
```

**Anmerkung:** Ab hier könnte es ggf. sinnvoll sein, gcc oder clang festzuschreiben

## Header III

„Vollständiges“ Beispiel:

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

# Was ist Versionsverwaltung?

Bietet:

- ▶ Protokoll der Änderungen an den Daten
- ▶ Ermöglicht dadurch auf beliebige alte Versionen zuzugreifen
- ▶ Unterschiedliche Entwicklungszweige
- ▶ Geteilter Zugriff auf die Daten

## Umsetzung in Git

- ▶ Git ist dezentral, es benötigt *keinen* Git-Server, insbesondere keinen Host wie GitHub, GitLab, ...
- ▶ Die Protokollierung der Änderungen geschieht durch sog. **commits**
- ▶ **commits** sind identifizierbar über ihren Hash und haben *eindeutige* Eltern!
- ▶ Die Struktur ist ein „gerichteter azyklischer Graph“

# Logbeispiel



- ▶ Zeiger auf den aktuellen commit: **HEAD**
- ▶ Man alle Vorgängercommits die zum aktuellen Status der Daten führen erreichen!
- ▶ Andere Entwicklungszweige sind einfach weiterer solcher Zeiger, sog. **branches**.
- ▶ „master“ ist ein typischer Name der Hauptbranch *aber nicht vorgegeben!*

# Tags

Also für jedes Release / jeden Zustand den man „festhalten“ möchte eine neue Branch? – Nein, es gibt **tags**.

- ▶ Referenz auf einen bestimmten **commit**
- ▶ Hat einen Namen (bspw. Versionsnummer v3.14)
- ▶ Es gibt „lightweight“ Tags und „annotated“ Tags

## Tags II

### Lightweight

- ▶ Besteht rein aus Referenz und Name
- ▶ Keinerlei weitere Informationen wie Name des Autors und Zeitpunkt

### Annotated

- ▶ Speichert zusätzlich Metainformationen, bspw. auch Beschreibung
- ▶ Sinnvoll für größere Releases, bspw. um relevante Änderungen zu listen

## Remotes: Arbeiten mit Online-Repositories

Bisher: Entwickler arbeitet alleine auf seinem Rechner, Code bleibt lokal.

Möchten: Code Teilen: **remotes**

# Remotes: Arbeiten mit Online-Repositories

**Bisher:** Entwickler arbeitet alleine auf seinem Rechner, Code bleibt lokal.

**Möchten:** Code Teilen: **remotes**

Remote:

- ▶ Benötigt Server (bspw. `git.imp.fu-berlin.de`, GitHub, GitLab)
- ▶ Es können mehrere remotes eingetragen werden
- ▶ **pull/push:** Änderungen holen/veröffentlichen

# Divergente Entwicklung: Mergen

Szenario:

1. Mehrere Entwickler ändern Datei **A** in ihren lokalen Repos **I1** und **I2**

## Divergente Entwicklung: Mergen

Szenario:

1. Mehrere Entwickler ändern Datei **A** in ihren lokalen Repos **I1** und **I2**
2. Entwickler 1 veröffentlicht Änderung auf gemeinsamer Remote **r**

# Divergente Entwicklung: Mergen

Szenario:

1. Mehrere Entwickler ändern Datei **A** in ihren lokalen Repos **I1** und **I2**
2. Entwickler 1 veröffentlicht Änderung auf gemeinsamer Remote **r**
3. Entwickler 2 versucht ebenfalls zu **pushen**: Schlägt fehl, da „Geschichte“ divergiert ist / seine lokale Version nicht mehr „aktuell“ ist

## Divergente Entwicklung: Mergen

Szenario:

1. Mehrere Entwickler ändern Datei **A** in ihren lokalen Repos **I1** und **I2**
2. Entwickler 1 veröffentlicht Änderung auf gemeinsamer Remote **r**
3. Entwickler 2 versucht ebenfalls zu **pushen**: Schlägt fehl, da „Geschichte“ divergiert ist / seine lokale Version nicht mehr „aktuell“ ist
4. Muss vorher **pullen**: Die Änderungen auf dem Server werden in die lokale Kopie integriert

## Divergente Entwicklung: Mergen

Szenario:

1. Mehrere Entwickler ändern Datei **A** in ihren lokalen Repos **I1** und **I2**
2. Entwickler 1 veröffentlicht Änderung auf gemeinsamer Remote **r**
3. Entwickler 2 versucht ebenfalls zu **pushen**: Schlägt fehl, da „Geschichte“ divergiert ist / seine lokale Version nicht mehr „aktuell“ ist
4. Muss vorher **pullen**: Die Änderungen auf dem Server werden in die lokale Kopie integriert
  - ! **WICHTIG**: Die Reihenfolge der Änderungen auf der Remote sind fest und dürfen nicht mehr geändert werden

## Divergente Entwicklung: Mergen

Szenario:

1. Mehrere Entwickler ändern Datei **A** in ihren lokalen Repos **I1** und **I2**
  2. Entwickler 1 veröffentlicht Änderung auf gemeinsamer Remote **r**
  3. Entwickler 2 versucht ebenfalls zu **pushen**: Schlägt fehl, da „Geschichte“ divergiert ist / seine lokale Version nicht mehr „aktuell“ ist
  4. Muss vorher **pullen**: Die Änderungen auf dem Server werden in die lokale Kopie integriert
    - ! **WICHTIG**: Die Reihenfolge der Änderungen auf der Remote sind fest und dürfen nicht mehr geändert werden
- ⇒ Lokale Änderungen werden (automatisch) angepasst und „angehängt“

# Mergeconflicts: Überschneidende Änderungen

Problem: Gleiche Datei wurde geändert.

Original:

```
1 the quick brown
2 fox jumps ovver
3 the lazy dog
4
```

Entwickler 1:

```
1 The quick brown
2 fox jumps ovver
3 the lazy dog
4
```

Entwickler 2:

```
1 the quick brown
2 fox jumps over
3 the lazy dog
4
```

- ▶ Textdatei & Änderung in verschiedenen Zeilen
- ⇒ Git kann idR. automatisch Änderungen zusammenführen

# Referenzen I

- ▶ Free Software Foundation.  
GNU make Manual  
<https://www.gnu.org/software/make/manual/make.html>
- ▶ Wikipedia Autoren.  
Artikel der englischen Wikipedia  
[https://en.wikipedia.org/wiki/Make\\_\(software\)](https://en.wikipedia.org/wiki/Make_(software))
- ▶ Git project.  
Git Referenz  
<https://git-scm.com/docs>

## Referenzen II

- ▶ Scott Chacon, Ben Straub.  
Pro Git E-Book  
<https://git-scm.com/book/en/v2>
- ▶ Wikipedia Autoren.  
Artikel der englischen Wikipedia  
<https://en.wikipedia.org/wiki/Git>