Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

Übung Betriebssysteme (BS)

Tafelübung 3: C-Extended

Sören Tempel

Wintersemester 2025

94t-c-extended 2025-11-20

Einführung

Technische Universität Braunschweig

- Grundlegende Konzepte sollten aus Prog. I und Prog. II bekannt sein
- C++ (Prog. II) wurde ursprünglich als Erweiterung für C entwickelt
- Daher:
  - Grundlagen werden in dieser Veranstaltung vorausgesetzt
  - Darüberhinaus: Ersten Crashkurs zu C in letzter Tafelübung
- Dieses mal:
  - Etwas erweiterte Einführung zu C
  - Fokus auf Bibliotheksfunktionen und C-Eigenheiten

Ziel: Weitere Grundlagen vermitteln zur Bearbeitung von Zettel ≥ 2

Tafelübung 3: C-Extended

- Datentypen
- Zeiger
- Der C-Präprozessor
- Die Standard-Bibliothek
- Buildsystem: Make

Datentypen

Arrays

Technische Universität Braunschweig

- Zusammenfassung einer festen Anzahl an Daten eines Typs
- Wesentliche Eigenschaften:
  - Arrays haben eine feste Größe
  - Zugriff auf ein zufälliges Element in $\mathcal{O}(1)$

Beispiel zur Deklaration verschiedener Arrays:

```
int int_array[5];     // Eindimensionaler Array der Größe 5 von Integern
char* ptr_attry[23];  // Eindimensionaler Array der Größe 23 von Zeigern
int matrix[2][3];     // Zweidimensionaler Array
```

Arrays: Initialisierung

Beispiel:

```
int prime[4] = {2, 3, 5, 7};
char name[4] = {'B', 'o', 'b', '\0'}; // Nullbyte ('\0') zeigt Stringende an
```

- Initialisierung mit 0 bei zu wenigen Initialisierungskonstanten
- Elemente werden im Speicher hintereinander abgelegt

Spezialfälle:

```
int prime[] = {2, 3, 5, 7}; // Länge muss nicht explizit angegeben werden
char name[4] = "Bob";       // Stringliteral als syntaktischer Zucker
```

Arrays: Zugriff auf Elemente

Technische Universität Braunschweig

- Syntax ist Analog zu Java
- **Achtung:** Kein implizites *bounds checking*!
- Der Zugriffsindex muss innerhalb der Arraygrenzen liegen

Beispiele:

```
int prime[4] = {2, 3, 5, 7};
prime[0] = 11; // Zuweisung
prime[0] == 11;
prime[1] == 3;
prime[4]; // Undefined Behavior!
```

Exkurs: Undefined Behavior

Technische Universität Braunschweig

- Semantik von C ist durch einen Sprachstandard festgelegt (z.B. C11)
- Für bestimmte Edge Cases ist das Verhalten undefiniert
- Beispiele:
  - Teilen durch 0
  - Zugriff außerhalb der Arraygrenzen
  - …
- Erleichterung für Kompiler, erlaubt in bestimmten Fällen effizienteren Code
- Konsequenz: Tatsächliches Verhalten hängt von konkretem System ab
- Teilweise desaströse Konsequenzen bis hin zu kritischen Sicherheitslücken

$\Rightarrow$ Ihr solltet euren Code so schreiben dass er kein UB enthält

Typedefs

Technische Universität Braunschweig

Syntax: `typedef <Variablen-Definition>;`
Der Variablen-Name wird der neue Typname.

(1. Näherung): `typedef <Datentyp> <Neuer Typname>;`

- Ermöglicht anlegen von *Typ Aliases*
- Zweck:
  - Lesbarkeit von Code; “Sprechende Namen”
  - Portabilität

Beispiele:

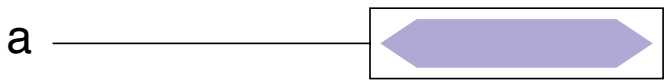
```
typedef unsigned short semester;
typedef struct student {
  char *name;
  semester fachsemester;
} student_t;
```

Zeiger

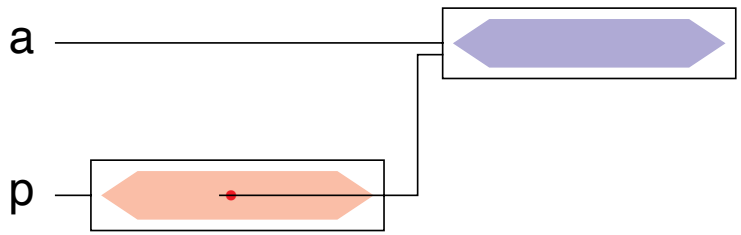
Wiederholung: Zeiger

- **Literal:** `'a'`
  Darstellung eines Wertes

- **Variable:** `char a;`
  Behälter für einen Wert



- **Zeiger-Variable:** `char *p = &a;`
  Behälter für eine Referenz
  auf eine Variable



Zeiger als Funktionsargumente

Technische Universität Braunschweig

- Parameter werden in C immer *by-value* übergeben
  - Parameterwerte werden in lokale Variablen der aufgerufenen Funktion kopiert
  - Aufgerufene Funktion kann tatsächliche Parameter des Aufrufers nicht ändern

- Das gilt auch für Zeiger (Verweise)
  - Aufgerufene Funktion erhält eine Kopie des Adressverweises
  - Mit Hilfe des *****-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden

    ⇝ **Call-by-reference**

Zeiger auf Funktionen

- Man kann Zeiger auch auf Funktionen zeigen lassen
  - Übergabe von Funktionen als Argumente an andere Funktionen
  - Name der Funktion ist Label für deren Adresse (vgl. Arrays)
  - Syntax:

```
<Rückgabetyp> (*<Variablen-Name>) (<kommagetrennte Parametertypenliste>);
```

- Beispiel:

```
void test(int i, char s, char *p_s); // Funktionsdeklaration

void (*p_zeiger) (int, char, char *); //def.: Zeiger auf Funktion
p_zeiger = &test; // Zuweisung

typedef void (*func_t)(int, char, char*);
func_t p_zeiger2 = &test;
```

- Verwendung: Aufruf wie ein normale Funktion

```
p_zeiger(23, 'a', "foobar");
p_zeiger2(23, 'a', "foobar");
```

Umwandlung von Datentypen

Implizite Umwandung

- Automatische Umwandlung von Datentypen
- Beispiel:

```
unsigned char one_byte = 0;
unsigned int four_byte = 0x1ff; //=511
one_byte = four_byte; //=0xff = 255
```

- Achtung: Ergebnis einer Operation mit `int`-Parametern ist ein `int`

```
int a = 5;
int b = 4;
float f = a / b; //=1.0
```

- Ordnung: char → short → int → long → long long → float → double
- Alle Literale (e.g., 23) sind **signed** int.

Umwandlung von Datentypen

Explizite Umwandung

- Syntax: `(Zieltyp) <Daten,Variable>`
- Beispiel:

```
unsigned char one_byte = 0;
unsigned int four_byte = 511;
one_byte = (char) four_byte //=255
//----------
int a = 5;
int b = 4;
float f = (float) a / b;   //=1.25
```

Umwandlung von Zeigern

- Auch zwischen verschiedenen Zeigern kann umgewandelt werdet
- Nützlicher Datentypen diesbezüglich: `void*` (Void-Zeiger)
  - Jeder Zeiger kann in einen Void-Zeiger umgewandelt werden
  - Void-Zeiger kann später wieder in ursprünglichen Pointer umgewandelt werden
- Obacht:
  - Pointer-Arithmetik wie bei `char*`
  - Void-Zeiger können nicht dereferenziert werden

Der C-Präprozessor

Übersicht

Technische Universität Braunschweig

- Transformiert den Source-Code vorm eigentlichen Übersetzen
- Im Prinzip eine eigene, unabhängige Sprache
- Syntax:
  - Alle Befehle an Präprozessor beginnen mit einem `#`
  - Befehle an Präprozessor werden **nicht** mit einem `;` abgeschlossen!
  - Definieren eines *Macros*: `#define`
  - Inkludieren einer anderen Datei: `#include`
  - Bedingungen: `#if`, `#ifdef`, …, `#endif`

Makros definieren

Technische Universität Braunschweig

Syntax: `#define <Name> <Ersatz>`

- Ersetzt beliebigen Begriff `<Name>` im Quellcode durch `<Ersatz>`
- Konvention: `#define` Namen werden GROSS GESCHRIEBEN
- Häufig im Einsatz als Ersatz für Konstanten

Beispiel:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Makros mit Parametern definieren

Technische Universität Braunschweig

Syntax: `#define <Name>(par1, par2, ...) <Ersatz>`

- Verhalten ähnlich einer inline-Funktion
  - Auf Klammerung achten!
  - Das Makro ist eine reine Ersetzung und bildet keinen Scope!
  - Es werden keine Typkontrollen vorgenommen

Beispiel:

```
#define ADD(a, b) \
        (a+b)
int result = 2 * ADD(10, 11); // -> 42
```

Dateien inkludieren mit #include

Technische Universität Braunschweig

Syntax:

Eigene Datei: `#include "foo.h"`

Bibliothek: `#include <stdio.h>`

- Fügt den Inhalt der angegebenen Datei ein in die aktuelle ein
- Ermöglicht Definition von Funktionsprototypen einzubinden
- Unterscheidung:
  - Suche nach Datei relativ zum aktuellen Verzeichnis
  - Suche der Datei in einem vordefinierten Systempfad

Include-Guards

Technische Universität Braunschweig

- **Problem**: Header wird einmal direkt und einmal indirekt included
  Compiler-Fehler: duplicate declaration.
- Bekanntes Pattern um dies zu umgehen

```
#ifndef __MYLIBARY_TYPES_H
#define __MYLIBARY_TYPES_H
....
#endif
```

- Moderne Alternative: `#pragma once`

Die Standard-Bibliothek

Einführung

Unterscheidung: Die C-Standardbibliothek und der POSIX Standard

1. Die C-Standardbibliothek:
   - Teil vom C Sprachstandard selbst
   - Spezifiert u.a. Funktionen für Ein-/Ausgabe, Speicherverwaltung, …
   - C11: `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`
2. Das Portable Operating System Interface (POSIX)
   - Erweitert C-Standardbibliothek um Betriebssysteme-nahe Funktionalität
   - Zum Beispiel: Funktion zur Prozessverwaltung
   - POSIX.1-2008: `https://doi.org/10.1109/IEEESTD.2018.8277153`

Hier erstmal Fokus auf ersteres, letzteres kommt später.

Fehlerbehandlung mit errno.h

Technische Universität Braunschweig

- Fast jeder Systemcall oder Bibliotheksaufruf kann fehlschlagen
- Fehlerbehandlung nötig
- Rückgabewert gibt Auskunft über Art des Fehlers
  - Fehlerfall: meist durch Rückgabewert `-1` repräsentiert
  - Fehlerursache: steht in globaler Variable **`errno(3)`** also die Art des Fehlers

Fehlerbehandlung mit errno.h

Technische Universität Braunschweig

- Fast jeder Systemcall oder Bibliotheksaufruf kann fehlschlagen
- Fehlerbehandlung nötig
- Rückgabewert gibt Auskunft über Art des Fehlers
  - Fehlerfall: meist durch Rückgabewert `-1` repräsentiert
  - Fehlerursache: steht in globaler Variable **`errno(3)`** also die Art des Fehlers
- Fehlermeldung kann als String mit **`perror(3)`** ausgegeben werden:

```c
#include <errno.h>
void perror(const char* prefix);
char *strerror(int errnum);
```

- Parameter `prefix`: zusätzliche Angabe über Auftreten des Fehlers

```c
errno = 2;
perror("cd"); // Typischerweise: Name der aufgerufenen Funktion
// cd: No such file or directory
```

Ein- und Ausgabe: stdio.h

Technische Universität Braunschweig

Funktionsumfang:

- Öffnen und Schließen von Dateien
- Lesen und Schreiben von Zeichen und Zeilen
- Formatierte Ein-/Ausgabe
- Fürs erste: Fokus auf Formatierte Ausgabe
- Mehr insb. zu Dateien folgt im Verlauf der Vorlesung

Schreiben einer ganzen Zeile:

```
int puts(const char* s);
```

- Gibt Zeichenkette `s` aus, terminiert mit einer Newline
- Rückgabewert: Gibt an ob dabei ein Fehler ausgetreten ist
- Problem: Ausgabe ist nicht parametrisiert

Formatierte Ausgabe:

```
int printf(const char* format, <Parameter>...);
```

- `format` ist eine konstante Zeichenkette mit Format-Flags
- Format-Flags werden durch formatierte Parameter `<Parameter>` ersetzt
- Format-Flags: `%d, %s, %x, ...`
- Reihenfolge der Flags mus den Parametern entsprechen

Beispiel:

```
printf("int: 32 bit");             // int: 32 bit
printf("int: %d bit", 32);         // int: 32 bit
printf("int: %d %s", 32, "bit");   // int: 32 bit
```

stdio.h: Format-Flags

Technische Universität Braunschweig

Format-Flags für formatierte Ausgabe der Argumente in `printf(3)`

- Syntax: `%` markiert Beginn eines Formatierungszeichens
- `%d`: **Dezimal ganzzahlig**
- `%x`: **Hexadezimal ganzzahlig** (Kleinbuchstaben)
- `%X`: Hexadezimal ganzzahlig (Großbuchstaben)
- `%p`: **Zeigerdarstellung** (wie Hexadezimal aber mit 0x vorweg)
- `%s`: **String** (= char array)
- `%c`: Einzelner Buchstabe (character)
- `%f`: Gleitkommazahl

Für mehr Informationen: Siehe die `printf(3)` man page.

stdlib.h

Technische Universität Braunschweig

Stellt verschiedene Hilfsfunktionen bereit, u.A.:

- Funktionen zur Konvertierung von Zeichenketten in Zahlen, z.B. `atoi()`, `strtol()`
- Abbruch-Funktionen, z.B. `exit()` oder `abort()`
- Funktionen zum Zugriff auf Umgebungsvariablen, z.B. `getenv()`
- Funktionen zur dynamischen Speicherverwaltung
  - `malloc()`
  - `free()`
  - `calloc()`
  - `realloc()`

Speicherverwaltung mit stdlib.h: `malloc` (Wdh.)

```
void* malloc(size_t size)
```

- Fordert zusammenhängende Menge an Speicher von der Größe `size` an
- Rückgabe: `void`-Zeiger auf das erste Speicherelement
  - Kann wie ein Array interpretiert werden
  - Vor Anwendung Zeiger in sinnvollen Datentyp umwandeln (*casten*)
  - Speicher wird *nicht* initialisiert
- *Achtung: Speicher wird nicht automatisch freigegeben!*

Beispiel:

```
int* array;

array = (int*) malloc(10 * sizeof(int));
```

Speicherverwaltung mit stdlib.h: `free` (Wdh.)

```
void free(void* p)
```

- Gibt dynamisch angeforderten Speicher wieder frei
- Erwartet einen Zeiger auf den freizugebenden Speicher

Beispiel:

```
int* array;

array = (int*) malloc(10, sizeof(int));
// etwas mit dem array machen
free(array);
```

Speicherverwaltung mit stdlib.h: `calloc`

Technische Universität Braunschweig

```
void* calloc(size_t nobj, size_t size)
```

- Fordert zusammenhängende Menge an Speicher von `nobj` × `size` Bytes an
- Initialisiert den Speicher mit `0`, sonst wie `malloc`

Beispiel:

```
int* array;

array = (int*) calloc(10, sizeof(int));
```

Speicherverwaltung mit stdlib.h: `realloc`

Technische Universität Braunschweig

```
void* realloc(void* p, size_t size)
```

- Vergrößert reservierten Speicher auf den `p` zeigt auf `size` Bytes
- Rückgabe:
  - Zeiger auf den neuen Speicherbereich
  - `NULL`-Zeiger, wenn die Anforderung nicht erfüllt werden kann

Beispiel:

```
data_t* data = malloc(sizeof(data_t));
data_t* more_data;
more_data = realloc(data, sizeof(data_t) * 2);
```

Speicherverwaltung mit stdlib.h: `realloc`

Technische Universität Braunschweig

```
void* realloc(void* p, size_t size)
```

- Vergrößert reservierten Speicher auf den `p` zeigt auf `size` Bytes
- Rückgabe:
  - Zeiger auf den neuen Speicherbereich
  - `NULL`-Zeiger, wenn die Anforderung nicht erfüllt werden kann

Beispiel:

```
data_t* data = malloc(sizeof(data_t));
data_t* more_data;
more_data = realloc(data, sizeof(data_t) * 2);
```

Achtung:

- Neu hinzugefügter Speicher ist *nicht* initialisiert
- Die zurückgegebene Adresse kann sich von `p` unterscheiden!

Buildsystem: Make

Warum brauchen wir ein Buildsystem?

Technische Universität Braunschweig

- Bisher: Manuelles aufrufen von `gcc(1)` zum Übersetzen von C Code
  - Kompileroptionen müssen jedes mal angegeben werden
  - Skaliert nicht wenn das Projekt aus mehreren C-Dateien besteht
- Ziel, insb. bei größeren Projekten: Unnötiges Kompilieren vermeiden
  - Daher: Nur die Dateien neu kompilieren die verändert wurden
  - Dafür brauchen wir ein **Buildsystem**
- Für C gibt es verschiedene Buildsysteme, wir benutzen `make(1)`

- **High-Level Idee**: Die `Makefile`-Sprache spezifiziert einen gerichteten, azyklischen Graphen (DAG) der Build-Abhängikeiten; von den Quelldateien zu den Build-Artefakten.

Makefile: Aufbau

Technische Universität Braunschweig

- Make, weit verbreitetes generisches Buildsystem
  - Nicht ausschließlich auf Übersetzen von C Code beschränkt
  - Daher erforderlich: Projekt-spezifische Konfiguration
- Konfiguration erfolgt über eine Datei mit dem Namen `Makefile`
- `Makefile` beschreibt *Targets* und *Regeln* wie sie gebaut werden
- Targets können *Abhängigkeiten* auf andere Targets haben

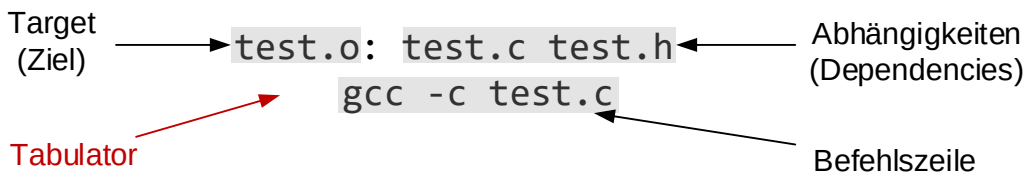
Aufbau:

```
<Target-1>: <Abhängigkeiten>
        <Shell-Befehle zum Bauen von Target-1>

<Target-2>: <Abhängigkeiten>
        <Shell-Befehle zum Bauen von Target-2>
```

Makefile: Funktionsweise

- Beispiel einer Target-Regel:



- Target: *Was?* (Hier: `test.o`)
- Abhängigkeiten: *Woraus?* (Hier: `test.c test.h`)
- Befehlszeile: *Wie?* (Hier: `GCC`-Kommando)
- Aufruf `make [<Target>]`$_{opt}$
  - Muss im Verzeichnis ausgeführt werden, in dem die Makefile liegt
  - Ohne Angabe des Targets wird das `Default`-Target ausgeführt
    - Default-Target ist das *erste* Target in der Makefile

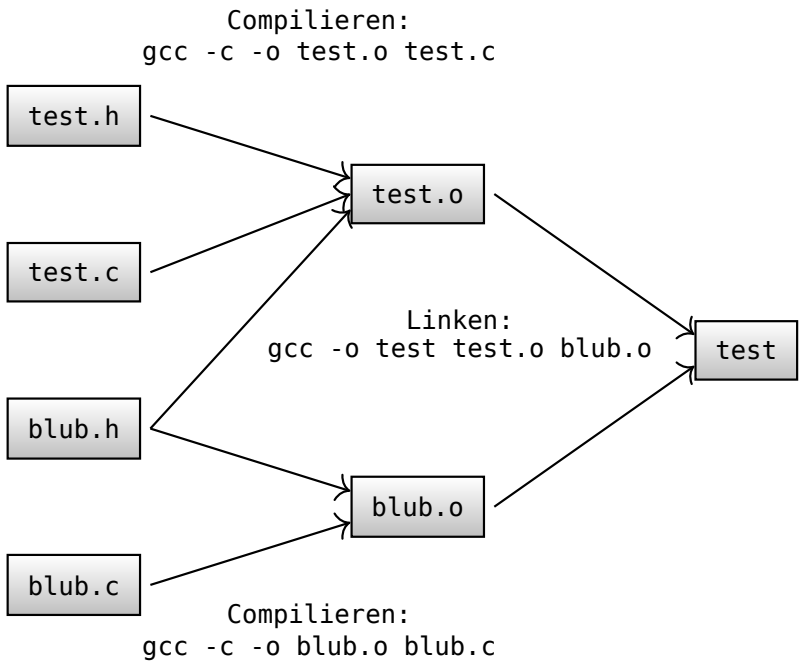
Beispiel

- Beispiel mit mehreren Regeln

```
test: test.o blub.o
        gcc –o test test.o blub.o

test.o: test.c test.h blub.h
        gcc –c test.c

blub.o: blub.c blub.h
        gcc –c blub.c
```



- Aufruf aus der Kommandozeile:

`make test` oder nur `make`

Variablen

Technische Universität Braunschweig

- Man kann Variablen definieren
  - Definition: `<NAME> := <wofür die Variable steht>`
  - Beispiel: `SOURCE := test.c blub.c`
- Verwendung von Variablen mit `$(<NAME>)` oder `${<NAME>}`

```
test: $(SOURCE)
        gcc –o test $(SOURCE)
```

- Übliche Variablen:
  - `CC` für C-Compiler-Befehl

    ⇒ Bei uns `CC := gcc`
  - `CFLAGS` für Optionen des C-Compilers

    ⇒ Bei uns: `CFLAGS := -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror`
  - `LD` – Das Linker-Kommando
  - `LDFLAGS` – Flags für den Linker
  - `CXX` – Der C++ Compiler
  - `CXXFLAGS` – Flags für den C++ Compiler

Implizite Regeln

Technische Universität Braunschweig

- Allgemeine Regel zum Umwandeln von Dateien
  - Dateien müssen anhand ihrer Endung erkennbar sein
  - % repräsentiert Dateiname ohne Dateiendung
- Beispiel für implizite Regel: `.o`-Datei aus `.c`-Datei erzeugen

```
%.o: %.c
        gcc –c $<
```

- Explizite Regeln überschreiben implizite Regeln

```
test.o: test.c
        $(CC) $(CFLAGS) –DXYZ –c $<
```

Vordefinierte Variablen

Technische Universität Braunschweig

- **$@** Name des Targets

```
test: test.o blub.o
        gcc –o $@ test.o blub.o
```

- **$*** Basisname des Targets (ohne Dateiendung)

```
test.o: test.c test.h
        gcc –o test.o -c $*.c
```

- **$<** Name der ersten Abhängigkeit

```
test.o: test.c test.h
        gcc –o test.o -c $<
```

- **$^** leerzeichengetrennte Liste aller Abhängigkeiten

```
test.o: test.c blub.o
        gcc –o test -c $^
```

⇒ Vordefinierte Variablen können nur in der Befehlszeile verwendet zu werden

Variablen: Implizite Regeln

- Allgemeine Regel zum Umwandeln von Dateien
  - Dateien müssen anhand ihrer Endung erkennbar sein
  - % repräsentiert Dateiname ohne Endung
  - Anwendbar im Target und in den Abhängigkeiten
- Beispiel für Suffix-Regel: `.o`-Datei aus `.c`-Datei erzeugen

```
%.o: %.c
        gcc –c $<
$CFLAGS -o $@
```

- Explizite Regeln überschreiben Suffix-Regeln

```
test.o: test.c
        $(CC) $(CFLAGS) –DXYZ –c $<
```

Vordefinierte implizite Regeln

Technische Universität Braunschweig

- Einige geläufige implizite Regeln sind in Make vordefiniert
  - Müssen nicht selbst spezifiziert werden
  - Funktionieren für einfache Abhängigkeiten
  - Verwenden Variablen, die such verändert werden können:
    - `CC` für Compileraufruf
    - `CFLAGS` für Compilerparameter
    - `LDFLAGS` für Linkerparameter

Beispiel: Vordefinierte implizite Regel zum Kompilieren von C-Code

```
%.o: %.c
        $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
```

Variablen: Erweiterte Regeln

- Ersetzung in existierenden Variablen
  - Beispiel: in Variable `SOURCE` werden Wörter, die auf `.c` enden, in gleichnamige Wörter, die auf `.o` enden, umbenannt und das Ergebnis wird an das Makro `OBJS` übergeben:

    ```
    OBJS = $(SOURCE:%.c=%.o)
    ```
  - oder

    ```
    OBJS = $(SOURCE:.c=.o)
    ```
  - *Achtung:* Leerzeichen werden hier als Teil des Namens interpretiert!

  ⇒ Dies ist im Allgemeinen *nicht* gewünscht
- Hinzufügen eines Elements zu einer Liste von Elementen:

```
ALLJOBS = $(OBJS) hallo.o
```

- Variablen werden verzögert aufgelößt. Es gibt unterschiedliche Zuweisungsoperatoren:

```
ALLJOBS = $(OBJS) hallo.o

ALLJOBS := $(OBJS) hallo.o
```

Pseudo-Targets und PHONY

Technische Universität Braunschweig

- Dienen nicht der Erzeugung einer gleichnamigen Datei
- High-Level Ziele, wie beispielsweise „Baue alles“.
- Konventionelle Namen: `all`, `install`, `clean`

Aufräumen mit `make clean`:

```
clean:
	rm -f $(OBJS) main
```

Installieren mit `make install`:

```
install:
	cp main /usr/local/bin/mein-program
```

- Deklaration als Abhängigkeit des Spezial-Targets `.PHONY`
  - Beispiel: `.PHONY: clean install`
  - Diese Targets werden *unabhängig von Änderungen immer gebaut*
  - Es ist egal ob es eine Datei `clean` gibt.

Vollständiges Beispiel

Technische Universität Braunschweig

```
CC = gcc
CFLAGS = -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall

SOURCE = test.c blub.c
OBJS = $(SOURCE:%.c=%.o)    # test.o blub.o
HEADER = $(SOURCE:%.c=%.h)  # test.h blub.h

test: $(OBJS)
        $(CC) –o $@ $(OBJS)

test.o: test.c $(HEADER)
        $(CC) $(CFLAGS) -ffast-math –c $<

%.o: %.c
        $(CC) $(CFLAGS) –c $<
```

Weitere Ressourcen

Technische Universität Braunschweig

- Bücher!
  - Manfred Dausman, et al. C als erste Programmiersprache. 2011
  - Brain W. Kernighan, Dennis M. Ritchie. The C Programming Language. 1988
  - Richard Resse. Understanding and Using C Pointers. 2013
  - Ben Klemens. 21st Century C. 2013
- Webseiten
  - Die C FAQ: `https://c-faq.com`
  - Interaktives C Tutorial: `https://www.learn-c.org/`
- Der C11 Sprachstandard:

  `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`