Übungen zu Betriebssysteme

Ü4 – C-Module & Präprozessor

Sommersemester 2023

Henriette Hofmeier, Manuel Vögele, Benedict Herzog, Timo Hönig

Bochum Operating Systems and System Software Group (BOSS)

Faculty of Computer Science

RUHR UNIVERSITÄT BOCHUM

RUB

Agenda

3.1 Dokumentation: Manual-Pages

3.2 C-Module

3.3 make

3.4 Der C-Präprozessor

Dokumentation aus 1. Hand: Manual-Pages

- Aufgeteilt in verschiedene *Sections*
  - 1 Kommandos
  - 2 Systemaufrufe
  - 3 Bibliotheksfunktionen
  - 3p POSIX-Beschreibungen der Bibliotheksfunktionen
  - 5 Dateiformate (spez. Datenstrukturen etc.)
  - 7 verschiedenes (z. B. Terminaltreiber, IP)

Dokumentation aus 1. Hand: Manual-Pages

- Aufgeteilt in verschiedene *Sections*
  - 1 Kommandos
  - 2 Systemaufrufe
  - 3 Bibliotheksfunktionen
  - 3p POSIX-Beschreibungen der Bibliotheksfunktionen
  - 5 Dateiformate (spez. Datenstrukturen etc.)
  - 7 verschiedenes (z. B. Terminaltreiber, IP)

- Angabe normalerweise mit *Section*: `printf(3)`
- Aufruf unter Linux:

```
$ # man [section] begriff
$ man 3 printf
```

Dokumentation aus 1. Hand: Manual-Pages

- Aufgeteilt in verschiedene *Sections*
  - 1 Kommandos
  - 2 Systemaufrufe
  - 3 Bibliotheksfunktionen
  - 3p POSIX-Beschreibungen der Bibliotheksfunktionen
  - 5 Dateiformate (spez. Datenstrukturen etc.)
  - 7 verschiedenes (z. B. Terminaltreiber, IP)

- Angabe normalerweise mit *Section*: `printf(3)`
- Aufruf unter Linux:

```
$ # man [section] begriff
$ man 3 printf
```

- Suche nach *Sections*: `man -f begriff`
- Suche nach Manual-Pages zu einem Stichwort:

```
user@host:~$ man -k stichwort
```

- **Achtung:** Manual-Pages unter Mac OS oft abweichend von Linux
  ⇒ VM ist Referenzsystem!

Module

- neben Funktionen elementarer Baustein für die Modularisierung
- Kapselung von zusammengehörenden
  - Funktionen und deren Implementierung
  - (Typ-)definitionen Konstanten, Makros
  - globalen Daten

  $\rightarrow$ Implementierungsdetails werden verborgen

  $\rightarrow$ andere Module sehen nur die Modulschnittstelle
- `static`: globale Variable/Funktion „von Aussen“ nicht zugreifbar
- **Achtung:** unterschiedliche Bedeutung von `static` für lokale und globale Variablen
  - `static` bei lokaler Variable: modifiziert Lebensdauer
  - `static` bei globaler Variable: modifiziert Sichtbarkeit

Module

module.c

```c
// globale Variable
int global_var = 42;

// modulweite globale Variable
static int module_var = 42;

// globale Funktion
void global_func(void) {
  //...
}

// modulweite Funktion
static void module_func(void) {
  // lokale Variable (Lebensdauer: Funktion)
  int a = 42;
  a++;

  // lokale Variable (Lebensdauer: Programm)
  static int b = 42;
  b++;
}
```

- `global_var` und `global_func()` global sichtbar
  → Namen programmweit eindeutig!
- `module_var` und `module_func()` nur innerhalb des Moduls sichtbar
  → Namen wiederverwendbar!
- a und b nur in `module_func()` sichtbar
- Lebensdauer von a: Funktion
  → Wert von a immer 43 nach `module_func()`
- Lebensdauer von b: Programm
  → Wert von b mit jedem Aufruf von `module_func()` inkrementiert

Schnittstellenbeschreibung: Header

- Zusammenfassen aller global sichtbaren Teile in Header-Dateien (`*.h`)
  - Funktionsdeklaration (nicht -definition!)
  - (Typ-)definitionen, Konstanten, Makros
- Implementierung der Schnittstelle in `*.c`-Dateien
- Andere Module inkludieren Header-Datei
  $\rightarrow$ Headerdatei beschreibt Modulschnittstelle

Module

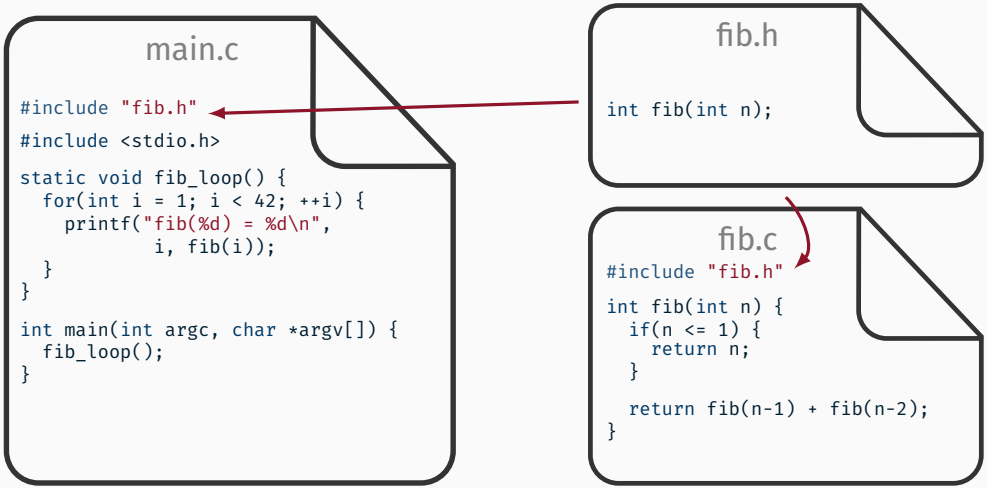
main.c

```c
#include <stdio.h>

static void fib_loop() {
  for(int i = 1; i < 42; ++i) {
    printf("fib(%d) = %d\n",
           i, fib(i));
  }
}

int main(int argc, char *argv[]) {
  fib_loop();
}
```

fib.c

```c
int fib(int n) {
  if(n <= 1) {
    return n;
  }

  return fib(n-1) + fib(n-2);
}
```

Module

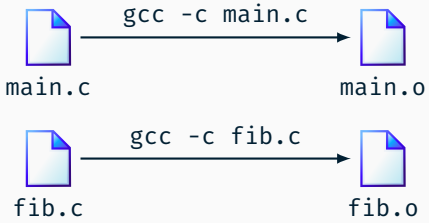

main.c
#include "fib.h"
#include <stdio.h>
static void fib_loop() {
for(int i = 1; i < 42; ++i) {
printf("fib(%d) = %d\n",
i, fib(i));
}
}
int main(int argc, char *argv[]) {
fib_loop();
}
fib.h
int fib(int n);
fib.c
#include "fib.h"
int fib(int n) {
if(n <= 1) {
return n;
}
return fib(n-1) + fib(n-2);
}

Module, Übersetzen & Linken

main.c

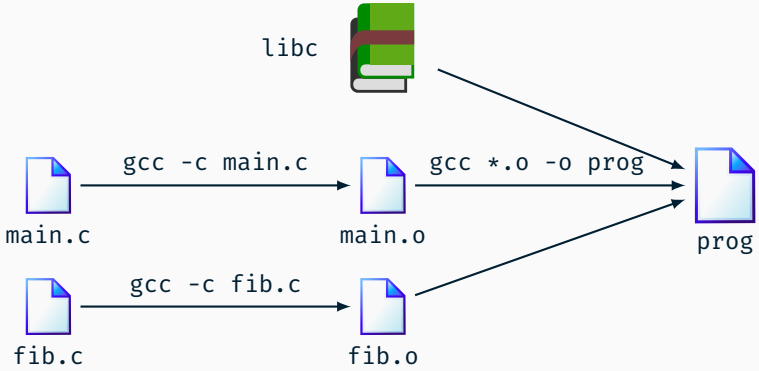
fib.c

Module, Übersetzen & Linken



- Schritt 1: Übersetzen der `.c`-Dateien zu Objektdateien
  - Ausführen des C-Präprozessors (Auflösen von `#includes`)
  - Übersetzen des Quellcodes in Binärcode (`-c` Flag des `gcc`)
  - Funktionsaufrufe in andere Module bleiben unaufgelöst

Module, Übersetzen & Linken



- Schritt 2: Linken der Objektdateien (und Bibliotheken) zum Programm
  - Zusammenfassen der Objektdateien zu einer einzelnen Binärdatei
  - Auflösen von Funktionsaufrufe in andere Module
    → Fehlermeldung falls Funktion nicht gefunden wurde

Agenda

Make – Warum?

# .c-Dateien	
lilo/sieve	1
vim 8.1	136
OpenSSH 7.9p1	269
Linux 4.19.1	$>$ **26000**

Make – Warum?

# .c-Dateien	
lilo/sieve	1
vim 8.1	136
OpenSSH 7.9p1	269
Linux 4.19.1	> **26000**

✗ von Hand übersetzen: zu aufwändig

Make – Warum?

# .c-Dateien	
lilo/sieve	1
vim 8.1	136
OpenSSH 7.9p1	269
Linux 4.19.1	> **26000**

- ✗ von Hand übersetzen: zu aufwändig
- ✗ Dauer bei wiederholtem Übersetzen

Make – Warum?

# .c-Dateien	
lilo/sieve	1
vim 8.1	136
OpenSSH 7.9p1	269
Linux 4.19.1	$>$ **26000**

✗ von Hand übersetzen: zu aufwändig

✗ Dauer bei wiederholtem Übersetzen

→ Automatisiertes Übersetzen **modifizierter Dateien**

Make – Teil 1

- Grundsätzlich: Erzeugung von Dateien aus anderen Dateien
  - für uns interessant: Erzeugung einer .o-Datei aus einer .c-Datei

- Falls Quelle(n) sich ändert wird der Befehl neu ausgeführt
- Änderung auf Basis der Modifikationszeit

Funktionsweise

- Regeldatei mit dem Namen `Makefile`

```
test.o: test.c test.h
	gcc -c -o test.o test.c
```

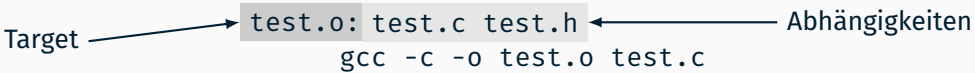
Funktionsweise

- Regeldatei mit dem Namen `Makefile`

  - Target (was wird erzeugt?)
    - Name der zu erstellenden Datei

Funktionsweise

- Regeldatei mit dem Namen `Makefile`



  - Target (was wird erzeugt?)
    - Name der zu erstellenden Datei
  - Abhängigkeiten (woraus?)
    - Namen aller Eingabedateien (direkt oder indirekt)
    - Können selbst Targets sein

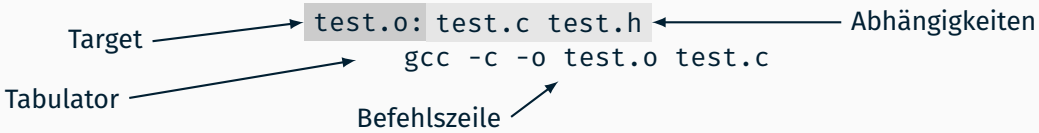
Funktionsweise

- Regeldatei mit dem Namen `Makefile`

  - Target (was wird erzeugt?)
    - Name der zu erstellenden Datei
  - Abhängigkeiten (woraus?)
    - Namen aller Eingabedateien (direkt oder indirekt)
    - Können selbst Targets sein
  - Befehlszeilen (wie?)
    - Erzeugt aus den Abhängigkeiten das Target

Funktionsweise

- Regeldatei mit dem Namen `Makefile`



  - Target (was wird erzeugt?)
    - Name der zu erstellenden Datei
  - Abhängigkeiten (woraus?)
    - Namen aller Eingabedateien (direkt oder indirekt)
    - Können selbst Targets sein
  - Befehlszeilen (wie?)
    - Erzeugt aus den Abhängigkeiten das Target

- zu erstellendes Target bei `make`-Aufruf angeben: `make test.o`
  - Falls nötig baut `make` die angegebene Datei neu
  - Davor werden rekursiv alle veralteten Abhängigkeiten aktualisiert
  - Ohne Target-Angabe bearbeitet `make` das erste Target im Makefile

Makros

- In einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

Makros

- In einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit `$(NAME)` oder `${NAME}`

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

Makros

- In einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit `$(NAME)` oder `${NAME}`

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

- Erzeugung neuer Makros durch Konkatenation

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

Makros

- In einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit `$(NAME)` oder `${NAME}`

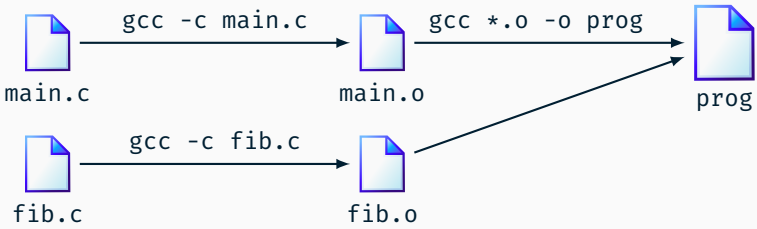
```
test: $(SOURCE)
  gcc -o test $(SOURCE)
```

- Erzeugung neuer Makros durch Konkatenation

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

- Gängige Makros:
  - `CC`: C-Compiler-Befehl
  - `CFLAGS`: Optionen für den C-Compiler

Schrittweises Übersetzen



- Rechner beim Erzeugen von ausführbaren Dateien „entlasten”
  - Zwischenprodukte verwenden und somit Übersetzungszeit sparen

```
CC = gcc
CFLAGS = -Wall -Werror -pedantic -std=c11
CFLAGS = $(CFLAGS) -D_XOPEN_SOURCE=700

prog: main.o fib.o
    $(CC) $(CFLAGS) -o prog main.o fib.o

fib.o: fib.c fib.h
    $(CC) $(CFLAGS) -c fib.c

main.o: main.c fib.h
    $(CC) $(CFLAGS) -c main.c
```

Pseudo-Targets

- Dienen nicht der Erzeugung einer gleichnamigen Datei
  - so deklarierte Targets werden immer gebaut
  - Deklaration als Abhängigkeit des Spezial-Targets `.PHONY` nötig

Pseudo-Targets

- Dienen nicht der Erzeugung einer gleichnamigen Datei
  - so deklarierte Targets werden immer gebaut
  - Deklaration als Abhängigkeit des Spezial-Targets **`.PHONY`** nötig
- Beispiel: Erzeugen einer ausführbaren Datei mit `make all`

```
.PHONY: all clean

all: lilo

clean:
    rm -f lilo

lilo: lilo.o #...
    # build lilo
```

  - Konventionen
    - **`all`** ist immer erstes Target im Makefile und baut die komplette Anwendung
    - **`clean`** löscht alle durch `make` erzeugte Dateien
    - **Hinweis:** bei Aufruf von `rm` den Parameter `-f` verwenden
      ⇒ kein Abbruch bei nicht existierenden Dateien

Der C-Präprozessor

- C-Präprozessor bearbeitet C-Code **vor** dem Compiler
- C-Präprozessor führt Texttransformationen aus
  $\rightarrow$ der Präprozessor hat kein Verständnis von C!
- Automatische Transformationen
  - Kommentare werden entfernt
  - Zeilen die mit \ enden werden zusammengefügt
  - ...

Der C-Präprozessor

- Steuerbare Transformationen (durch den Programmierer)

`#include <file>` Fügt Dateiinhalt an die aktuelle Stelle ein

Der C-Präprozessor

- Steuerbare Transformationen (durch den Programmierer)

`#include <file>`	Fügt Dateiinhalt an die aktuelle Stelle ein
`#define <makro> <replace>`	Definiert ein Makro. Ab dieser Stelle wird jedes Vorkommen von `<makro>` durch `<replace>` ersetzt. `<replace>` kann auch leer sein.
`#undef <makro>`	

Der C-Präprozessor

- Steuerbare Transformationen (durch den Programmierer)

`#include <file>`	Fügt Dateiinhalt an die aktuelle Stelle ein
`#define <makro> <replace>` `#undef <makro>`	Definiert ein Makro. Ab dieser Stelle wird jedes Vorkommen von `<makro>` durch `<replace>` ersetzt. `<replace>` kann auch leer sein.
`#if <cond.>` `#elif`,`#else`,`#endif`	Löscht/Behält den umklammerten Text abhängig von der Bedingung

Der C-Präprozessor

- Steuerbare Transformationen (durch den Programmierer)

`#include <file>`	Fügt Dateiinhalt an die aktuelle Stelle ein
`#define <makro> <replace>` `#undef <makro>`	Definiert ein Makro. Ab dieser Stelle wird jedes Vorkommen von `<makro>` durch `<replace>` ersetzt. `<replace>` kann auch leer sein.
`#if <cond.>` `#elif`,`#else`,`#endif`	Löscht/Behält den umklammerten Text abhängig von der Bedingung
`#ifdef <makro>` `#ifndef <makro>`	Löscht/Behält den umklammerten Text abhängig davon ob `<makro>` definiert wurde oder nicht

Der C-Präprozessor – Beispiele

```c
// kopiert Inhalte von stdio.h hierher
// z.B. Deklaration von printf()
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
}
```

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

// PI wird ab jetzt überall durch 3.1415 ersetzt
#define PI 3.1415

int main(void) {
  printf("%f\n", PI);
}
```

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

// falls PI bereits definiert wurde, lösche
// Definition. Ansonsten definiere PI so
// dass es immer durch 3 ersetzt wird
#ifdef PI
  #undef PI
  #define PI 3
#else
  #define PI 3
#endif

int main(void) {
  printf("%d\n", PI);
}
```

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) a + b
#define MUL(a, b) a * b

// shift nach links entspricht Multiplikation mit 2
// POW2(3) --> 2^3
#define POW2(a) 1 << a

int main(void) {
  printf("%d\n", ADD(3,5)); // --> 8
  printf("%d\n", MUL(3,5)); // --> 15
  printf("%d\n", POW2(3)); //  --> 8
}
```

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) a + b
#define MUL(a, b) a * b

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) a + b
#define MUL(a, b) a * b

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

- `MUL(ADD(2,3), 2)`

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) a + b
#define MUL(a, b) a * b

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

- MUL(ADD(2,3), 2)
- MUL(2+3, 2)

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) a + b
#define MUL(a, b) a * b

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

- MUL(ADD(2,3), 2)
- MUL(2+3, 2)
- 2+3*2 *// --> 8 statt 10*

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

- `MUL(ADD(2,3), 2)`

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

- MUL(ADD(2,3), 2)
- MUL((2+3), 2)

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 2)); // --> ?
}
```

- `MUL(ADD(2,3), 2)`
- `MUL((2+3), 2)`
- `((2+3)*2)` *`// --> 10`*

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- MUL(ADD(2,3), 3-1)

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- MUL(ADD(2,3), 3-1)
- MUL((2+3), 3-1)

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) (a + b)
#define MUL(a, b) (a * b)

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- `MUL(ADD(2,3), 3-1)`
- `MUL((2+3), 3-1)`
- `((2+3)*3-1)` *`// --> 14 statt 10`*

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) ((a) + (b))
#define MUL(a, b) ((a) * (b))

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- `MUL(ADD(2,3), 3-1)`

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) ((a) + (b))
#define MUL(a, b) ((a) * (b))

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- `MUL(ADD(2,3), 3-1)`
- `MUL(((2)+(3)), 3-1)`

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) ((a) + (b))
#define MUL(a, b) ((a) * (b))

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- `MUL(ADD(2,3), 3-1)`
- `MUL(((2)+(3)), 3-1)`
- `((((2)+(3))) * (3-1))`

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) ((a) + (b))
#define MUL(a, b) ((a) * (b))

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- `MUL(ADD(2,3), 3-1)`
- `MUL(((2)+(3)), 3-1)`
- `((((2)+(3))) * (3-1))`
- `(2+3) * (3-1)` *`// --> 10`*

Der C-Präprozessor – Beispiele

```c
#include <stdio.h>

#define ADD(a, b) ((a) + (b))
#define MUL(a, b) ((a) * (b))

int main(void) {
  printf("%d\n", MUL(ADD(2,3), 3-1)); // --> ?
}
```

- `MUL(ADD(2,3), 3-1)`
- `MUL(((2)+(3)), 3-1)`
- `((((2)+(3))) * (3-1))`
- `(2+3) * (3-1) // --> 10`

→ Macros sind potenziell fehleranfällig

→ nur für einfache Konstantendefinition benutzen (z.B. `#define SIZE 100`)

→ für alles andere (ggf. `inline`) Funktionen