C-Kurs
Einführung

hannes.weisbach@tu-dresden.de [Programmierung]
https://auditorium.inf.tu-dresden.de/courses/2154638

Code: https://github.com/hannesweisbach/ccourse.git

Programmieren?

Problemanalyse (Sprachunabhängig)	Implementierung (Umsetzung)
Modell	Syntax
Dekomposition	Sprachkonstrukte, Features (Standard-)Bibliotheken

C?

Ausgewählte Dialekte

'69-'73: Entwickelt von Dennis Ritchie

'78: "The C Programming language" ("K&R")

'89/'90: ANSI C/ISO C ("C89")

'99: "C99"

'11: "C11"

Warum C?

speed

code portability

embedded systems

stability

intermediate language

systems programming

type punning

low overhead

operating systems

near-universal availability

efficieny

hardware access

Beispielhaftes

Vorgehen

bei der Programmierung

Code schreiben.

Kompilieren.

Testen.

Wiederhole.

Werkzeuge

Editor

(emacs, **vim**, gedit, Notepad, sublime, ed, …)

Compiler

(gcc, **clang**, cl.exe, …)

Debugger

(gdb, lldb, WinDbg, …)

Projektverwaltung

(make, **cmake**, scons, xcodebuild, vcproj, …)

Versionsverwaltung

(**git**, darcs, mercurial, subversion, …)

IDE

(kdevelop, XCode, VS, Code::Blocks, Eclipse)

C ist eine *imperative Sprache*

defines computation as statements that change a program state

(en.wikipedia.org)

Imperative Programmierung besteht aus einer Sequenz an Befehlen oder Instruktionen, die den Programmzustand (Daten) modifizieren.

Anatomie eines C Programmes

	(syntaktische) Umsetzung	Datei
Spezifikation	Deklaration	Header (.h-Datei)
Implementierung	Definition	Source (.c-Datei)

Anatomie eines C Programmes (cont'd)

Ablauf:

- Start bei Funktion main();
- Abarbeitung der Anweisungen
- Ende bei Verlassen von main();

Code?

Beispielprogramm:

```c
#include <stdio.h>
int
main()
{
        printf("Hello, world!\n");
        return 0;
}
```

```
$ gcc -Wall -Wextra -Werror -pedantic
-std=c99 -o hello hello.c
$ ./hello
```

How to get the code.

$ git clone https://github.com/hannesweisbach/ccourse.git

$ mkdir build

$ cd build

$ [CC=clang] cmake ../ccourse

$ make

$ ./introduction/hello

```
$ tree ccourse/
ccourse/
├── CMakeLists.txt
├── LICENSE
├── README.md
└── introduction
    ├── CMakeLists.txt
    ├── fizzbuzz-simple.c
    ├── hello-cmd.c
    ├── hello.c
    ├── safe-sum.c
    └── sum.c
```

Anweisungen?

"statements"

Anweisungen enden mit dem Semikolon ';'.

Nur ein Semikolon ist die leere Anweisung.

Ausdrücke gefolgt von einem Semikolon werden zu Anweisungen: `<expression>;`

Blöcke (`{...}`) enthalten Anweisungen und sind selbst eine Anweisung.

Ausdrücke?

"expressions"

arithmetische Ausdrücke (Addition, ...)

Logische/Relationale Ausdrücke (größer-als, ...)

Zuweisungen

Bitweise Ausdrücke

einige Andere (address-of, dereference, sizeof ...)

Datentypen

Primitive Typen
(int, char, double, _Complex)

```c
int count = 0;
float angle = 3.1415/6;
```

Aufzählungen
(enum)

```c
enum SUIT {SPADES, HEARTS, DIAMONDS, CLUBS};
enum SUIT suit = HEARTS;
```

Felder
(Array)

```c
char msg[] = "Hello.";
char ipv4[16];
```

Zeiger
(Pointer)

```c
int a;
int *a_ptr = &a;
```

Komponierte Typen
(struct, union)

```c
struct point {
  int x;
  int y;
};
struct point p = {3, 0};
struct point p2 = {
  .x = 5;
  .y = -2;
};
```

Funktionen

Deklaration

```
return-type funname(param-type param-name[, ...]);
int main(int argc, char* argc[]);
int printf(const char * fmt, ...);
```

Definition

```
int main()
{
     ...;
     return 0;
};
```

Aufruf

```
printf("some-string");
```

Recap

C-Kurs - Einführung

- Beispielprogramm:

```
#include <stdio.h>
int
main()
{
        printf("Hello, world!\n");
        return 0;
}
```

- expressions / statements
- Datentypen

Hello "Your Name"!

Schreibe ein Programm, welches

»Hello "<your-name>"«

ausgibt.

Arithmetik

```
  int r; int x = 4; int y = 2;

r = x + y;   // addition
  r++; ++r; // post/pre-increment (+1)
r = x - y;   // subtraction
  r--; --r; // post/pre-decrement (-1)
r = x * y;   // multiplication
r = x / y;   // division
r = x % y;   // modulo
// Op-Präzedenz: a = 3; b = c = 2;
r = a + b * c;
r = (a + b) * c;
```

main(int argc, char* argv[])

argument **c**ount

argument **v**ector (Array von strings)

```
argv[0]; // erstes Argument als string
argv[1]; // zweites Argument als string
```

Summe zweier Zahlen

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
    int a;
    int b;
    int sum = 0;

    if (argc < 3) {
        printf("Missing argument(s). Usage: %s <op1> <op2>\n", argv[0]);
        return EXIT_FAILURE;
    }

    sscanf(argv[1], "%d", &a);
    sscanf(argv[2], "%d", &b);

    //sum???

    printf("%s + %s = %d\n", argv[1], argv[2], sum);

    return EXIT_SUCCESS;
}
```

Bedingte Ausführung

```
if (<condition-expr>) {
   /* wenn condition zu nicht-0 evaluiert */
   <statement>;
} else {
   /* wenn condition zu 0 evaluiert */
   <statement>;
}
```

Arithmetische Vergleiche

gleich / ungleich	a == b a != b
kleiner / kleiner	a < b a > b
kleiner-gleich / größer-gleich	a <= b a >= b

Verknüpfung von Bedingungen

Konjunktion ("und")	&& and	`if (a < 5 && b > 7) { ...; }`
Disjunktion ("oder")	|| or	`if (a > 5 || b > 5) { ...; }`
Negation	! not	`if (!(a < 5)) { ...; }`

`and or not: #include <iso646.h>`

FizzBuzz (1)

Schreibe ein Programm, welches "Fizz" ausgibt, wenn eine eingegebene Zahl durch 3 teilbar ist, "Buzz", wenn die Zahl durch 5 teilbar ist, ansonsten die Zahl selbst.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
    int fizzbuzz;

    if (argc < 2) {
        printf("Missing argument(s). Usage: %s <op1>\n", argv[0]);
        return EXIT_FAILURE;
    }

    sscanf(argv[1], "%d", &fizzbuzz);

    /* ???? */

    return EXIT_SUCCESS;
}
```

FizzBuzz (2)

Wende FizzBuzz auf alle Zahlen von 1 bis 100 an.

Loops: Doing things more than once.

Anfang (init expr.)

Bedingung (condition expr.)

Schritt (iteration expr.)

Oder übersichtlicher:

```
for(int i = 1; i <= 100; i++) {
    fizzbuzz(i);
}
```

FizzBuzz (2)

Wende FizzBuzz auf alle Zahlen von 1 bis 100 an.

Hello, argc & argv!

Schreibe ein Programm, welches deinen Namen als Argument erhält und:

»Hallo "<your-name>"«

ausgibt.

Reading input

Schreibe ein Programm, welches die Anzahl der gegebenen Parameter und den kompletten Programmaufruf ausgibt.

Elements of style.

Sourcecode wird nur einmal geschrieben aber oft gelesen. *Lesbarkeit* ist wichtig.

Stil: Egal, hauptsache *konsistent*.

Kommentare: *Warum* und *Wie*?

Selbsdokumentierender Code: Namen!

Gute Programmierer:

- wiederholen sich nicht (DRY)
- schreiben Tests
- schreiben robusten (defensive) Code
- benutzen `assert()`s