



C Programmierkurs

3. Stunde: Mehr Kontrolle

Johannes Hayeß & Mirko Seibt

8. November 2018

Technische Universität Dresden

For-Schleifen (Wiederholung)

```
for (size_t i = 0; i < 5; ++i) { }
```

```
for (size_t i = 5; i; --i) { }
```

```
for (;;) { }
```

Wie mächtig sind For-Schleifen?

```
unsigned x;  
size_t z = 1;  
for (size_t y = x; y != 0; y = 0) {  
    printf("Do this!\n");  
    z = 0;  
}  
for (; z != 0; z = 0) {  
    printf("Do that!\n");  
}
```

Alternativen/Verzweigungen

Schlüsselwort: if

```
if ([Bedingung]) {  
    /* if-branch */  
}
```

Schlüsselwort: **if**

```
if ([Bedingung]) {  
    /* if-branch */  
}
```

Wenn Bedingung **erfüllt** ist, dann führe den if-Zweig aus.
Wenn Bedingung **nicht erfüllt** ist, dann überspringe den if-Zweig.

Vollständige Alternativen

Schlüsselwörter: **if**, **else**

```
if ([Bedingung]) {  
    /* if-branch */  
} else {  
    /* else-branch */  
}
```

Vollständige Alternativen

Schlüsselwörter: **if**, **else**

```
if ([Bedingung]) {  
    /* if-branch */  
} else {  
    /* else-branch */  
}
```

Wenn Bedingung **erfüllt** ist, dann führe den if-Zweig aus.

Wenn Bedingung **nicht erfüllt** ist, dann führe den else-Zweig aus.

Beispiel: Vollständige Alternativen

```
int a,b;
if(a) {
    if (b) {
        printf("a and b");
    } else {
        printf("a and not b");
    }
} else if(b) {
    printf("not a and b");
} else {
    printf("not a and not b");
}
}
```

Boolesche Logik

Der bool-Typ

In stdbool.h ist der bool-Typ definiert.
Ebenso sind true und false definiert.

```
#include<stdbool.h>

// true -> 1
// false -> 0
bool is_false = false;
bool is_true = is_false == 0;
```

Zweistellige Funktion:

```
[Operant] [Operator] [Operant]
```

Operatoren: && (logisches Und), || (logisches Oder)

Zweistellige Funktion:

[Operant] [Operator] [Operant]

Operatoren: && (logisches Und), || (logisches Oder)

Einstellige Funktion:

![Operant]

Operator: ! (Negator)

Beispiel: Boolesche Logik

```
bool a,b;  
if((a && !b) || (!a && b)) {  
    /* true */  
} else {  
    /* false */  
}
```


Beispiel: Boolesche Logik

```
bool a,b;  
if((a && !b) || (!a && b)) {  
    /* true */  
} else {  
    /* false */  
}
```

Mit &&, || und ! lassen sich alle booleschen Funktionen abbilden!

While-Schleifen

While Schleifen

Schlüsselwörter: **while**, **do**

```
while ([Bedingung]) {  
    /* while-code */  
}
```

Führe while-code aus, **solange** die Bedingung **erfüllt** ist.

While Schleifen

Schlüsselwörter: **while**, **do**

```
while ([Bedingung]) {  
    /* while-code */  
}
```

Führe while-code aus, **solange** die Bedingung **erfüllt** ist.

```
do {  
    /* while-code */  
} while ([Bedingung]);
```

while-code wird **mindestens** 1x ausgeführt.

For-Schleifen sind geeignet für Zählschleifen mit definierbarer Zählfunktion!

```
for(size_t i = 5; i; --i) { }
```

While vs. For

For-Schleifen sind geeignet für Zählschleifen mit definierbarer Zählfunktion!

```
for(size_t i = 5; i; --i) { }
```

While-Schleifen werden verwendet, wenn die Variable der Bedingung während des Schleifendurchlaufs verändert wird.

While vs. For

For-Schleifen sind geeignet für Zählschleifen mit definierbarer Zählfunktion!

```
for(size_t i = 5; i; --i) { }
```

While-Schleifen werden verwendet, wenn die Variable der Bedingung während des Schleifendurchlaufs verändert wird.

Vorsicht: Wenn Bedingungen **immer erfüllt** sind, entsteht eine Endlosschleife!

Schleifen ohne Ausführung:

```
while(0) { }
```

```
for(size_t i = 0; i; ) { }
```


Endlosschleifen:

```
while(true) { }
```

```
for(unsigned i = 1; i; i += 2) { }
```

Vorsicht: Endlosschleifen sind i.d.R. unerwünscht.

Schleifen kontrollieren

Schleifenabbruch

Schleifenunterbrechung:

Schlüsselwort: **break**

```
unsigned i = 10;
while(true) {
    if (i != 0) {
        --i;
    } else {
        break;
    }
}
```

Schleifenübersprung

Schleifenübersprung:

Schlüsselwort: **continue**

```
for(size_t i = 5; i; --i) {  
    if (i > 1) {  
        continue;  
    }  
    printf("Done!");  
}
```

Funktionen

Funktionsdefinition

Grundsätzlicher Aufbau:

```
[Rückgabe-Typ] [Name]([Typ] [Parameter], ...) {  
    /* function-code */  
    return [Wert];  
}
```

Funktionsdefinition

Grundsätzlicher Aufbau:

```
[Rückgabe-Typ] [Name]([Typ] [Parameter], ...) {  
    /* function-code */  
    return [Wert];  
}
```

Beispiel:

```
int main(void) {  
    /* function-code */  
    return EXIT_SUCCESS;  
}
```

```
int function(int parameter) {  
    return ((parameter * 3.1415) / 100);  
}  
  
int main(void) {  
    int parameter = 1337;  
    printf("%d\n", function(parameter));  
    return EXIT_SUCCESS;  
}
```


Scopes

Wichtig: Wir können nur Dinge verwenden die wir bereits kennen

Wichtig: Wir können nur Dinge verwenden die wir bereits kennen

Variablen und Funktionen müssen erst deklariert werden, bevor wir sie verwenden können!

Wichtig: Wir können nur Dinge verwenden die wir bereits kennen

Variablen und Funktionen müssen erst deklariert werden, bevor wir sie verwenden können!

Variablen sind nur innerhalb von Funktionen gültig (Kapselung).

Scopes: Unser erstes C-Programm

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* The main thing that this program does. */
5  int main(void) {
6  // Declarations
7      double A[5] = {
8          [0] = 9.0,
9          [1] = 2.9,
10         [4] = 3.E+25,
11         [3] = .00007,
12     };
13     // Doing some work
14     for (size_t i = 0; i < 5; ++i) {
15         printf("element %zu is %g,\ tits square is %g\n", i, A[i], A[i] * A[i]);
16     }
17
18     return EXIT_SUCCESS;
19 }
```

vgl. Gustedt, *Modern C*, Seite 2

Problem

```
void f(int x) {  
    g(x);  
}  
  
void g(int x) {  
    f(x);  
}
```

Zum Zeitpunkt des Aufrufs ist g() noch garnicht definiert

```
void g(int x);  
  
void f(int x) {  
    g(x);  
}  
  
void g(int x) {  
    f(x);  
}
```

```
void g(int x);  
  
void f(int x) {  
    g(x);  
}  
  
void g(int x) {  
    f(x);  
}
```

Vor allem wichtig in Bezug auf die main()-Funktion


```
void f(int x) {  
    /* function-code */  
}  
  
int g(int x) {  
    /* function-code */  
    return y;  
}  
  
int main(void) {  
    /* function-code */  
    return EXIT_SUCCESS;  
}
```

Vorsicht: Reihenfolge beachten!

```
void f(int x);  
int g(int x);  
int main(void) {  
    /* function-code */  
    return EXIT_SUCCESS;  
}  
  
void f(int x) {  
    /* function-code */  
}  
  
int g(int x) {  
    /* function-code */  
    return y;  
}
```

Ein- und Ausgabe

Ausgabe: printf

Bibliothek:

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

Ausgabe:

```
printf(  
    "[Text, Platzhalter, Steuerzeichen]",  
    [Parameter, Funktionen, Ausdrücke],  
    ...  
);
```

Vorsicht: Parameter werden der Reihe nach den Platzhaltern zugewiesen.

Eingabe: scanf

Bibliothek:

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

Eingabe:

```
[Typ] [Name] = scanf("[Typ-Platzhalter]", &[Name]);
```

Eingabe: scanf

Bibliothek:

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

Eingabe:

```
[Typ] [Name] = scanf("[Typ-Platzhalter]", &[Name]);
```

```
[Typ] [Name];  
scanf("[Typ-Platzhalter]", &[Name]);
```

Vorsicht: Traue niemals dem Benutzer!
Flasche Eingaben können womöglich nicht zugeordnet werden.

Vorsicht: Traue niemals dem Benutzer!

Flasche Eingaben können womöglich nicht zugeordnet werden.

Beispiele:

Typen passen nicht zueinander,

Leere Eingabe

Vorsicht: Traue niemals dem Benutzer!

Flasche Eingaben können womöglich nicht zugeordnet werden.

Beispiele:

Typen passen nicht zueinander,

Leere Eingabe

Überprüfe die Eingabe auf Richtigkeit!

Platzhalter beginnen immer mit '%':

Platzhalter beginnen immer mit '%':

Beispiele:

Typ	Beschreibung	Variablentypen
%d	Dezimalzahl	int, char
%c	Buchstabe, Zeichen	char, int (< 256)
%u	vorzeichenlose Dezimalzahl	unsigned int, unsigned char
%X	Hexadezimalzahl	char, int
%ld	lange Dezimalzahl	long
%f	Gleitkommazahlen	float, double
%zu	Sondertyp	size_t
%s	String	char[]

Und viele mehr ...

Steuerzeichen beginnen immer mit '\':

\n - neue Zeile

\t - Tabulator

\\ - Backslash

\\" - Anführungszeichen

%% - Prozentzeichen

und Weitere ...

Zeichen, Buchstaben

Schlüsselwort: **char**

Schlüsselwort: **char**

Können vorzeichenbehaftet sein (voreingestellt)

Speicherung: 1 Byte (8Bit) pro Zeichen

Zeichen, Buchstaben

Schlüsselwort: **char**

Können vorzeichenbehaftet sein (voreingestellt)

Speicherung: 1 Byte (8Bit) pro Zeichen

Repräsentieren genau einen Buchstaben

Zeichen werden ASCII kodiert

$A \mapsto 65$

Zeichen, Buchstaben

Schlüsselwort: **char**

Können vorzeichenbehaftet sein (voreingestellt)

Speicherung: 1 Byte (8Bit) pro Zeichen

Repräsentieren genau einen Buchstaben

Zeichen werden ASCII kodiert

$A \mapsto 65$

Beispiele:

```
char a = 'A';  
char b = 65;
```

Zeichenketten, Strings

Zeichenketten/Strings sind 0-terminierende **char**-Arrays,
d.h. im letzten Arrayselement **muss** eine 0 stehen:

Zeichenketten, Strings

Zeichenketten/Strings sind 0-terminierende **char**-Arrays, d.h. im letzten Arrayselement **muss** eine 0 stehen:

```
char string[6] = { "Hallo" };
```

```
char string[] = "Hallo";
```

Zeichenketten, Strings

Zeichenketten/Strings sind 0-terminierende **char**-Arrays, d.h. im letzten Arrayselement **muss** eine 0 stehen:

```
char string[6] = { "Hallo" };
```

```
char string[] = "Hallo";
```

```
char string[] = { 'H', 'a', 'l', 'l', 'o', 0, };
```

```
char string[6] = { 'H', 'a', 'l', 'l', 'o', };
```



Jens Gustedt. *Modern C*. Lizenz: CC BY-NC-ND 4.0. URL: <https://creativecommons.org/licenses/by-nc-nd/4.0/>.



Richard Mörbitz. *C-Kurs TU-Dresden 2017*. Inspirierte die Abschnitte bezüglich Kontrollstrukturen, char und I/O. Lizenz: CC BY 4.0. URL: <http://creativecommons.org/licenses/by/4.0/>.

Diese Präsentation ist lizenziert unter der **Creative Commons Attribution-ShareAlike 4.0 International** Lizenz.

