Kapitel 03 - Kontrollfluss

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Rückblick

Bisher: Strikte Ablauf des Programms von oben nach unten (*sequenziell*)

Jetzt: Abhängig von der Eingabe anderes Programmverhalten

```c
int main() {
    // Bei gerader Eingabe soll der Wert multipliziert werden,
    // ansonsten addiert
    int eingabe = 3;
    int ergebnis = eingabe + eingabe;
    printf("ergebnis = %d\n", ergebnis);
    return 0;
}
```

```c
int main() {
    // Bei gerader Eingabe so
    // ansonsten addiert
    int eingabe = 2;
    int ergebnis = eingabe *
    printf("ergebnis = %d\n",
    return 0;
}
```

Heute: Einfache Wiederholung von Befehlen und alternative Programmabläufe

Programmverzweigungen

- Häufig: Programmverhalten abhängig von ext. Faktoren → Eingabe
- Beispiel
  - Wenn die Eingabe gerade ist, dann multipliziere sie mit sich selbst.
  - Wenn die Eingabe ungerade ist, dann multipliziere sie mit sich selbst.
- Lösung: Verzweigung des Programms
- Schlüsselwörter: `if` und `else`

Programmverzweigungen

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int eingabe = 2, ergebnis = 0;

    if (eingabe % 2 ==
        ergebnis = eing
    } else {
        ergebnis = eing
    }
    printf("ergebnis =
    return 0;
}
```

Was genau ist `(eingabe % 2 == 0)` eigentlich?

Wahrheitswerte

Logischer Datentyp (`bool`)

- Zum Speichern von Wahrheitswerten „wahr” und „falsch” (*Boolean*)
- Wertevorrat: `true` und `false`
- Datendefinition: `bool b;`
- Zuweisung: `b = true;`

Logische Operatoren

Name	Symbol	Beispiel
UND (*AND*)	&&	b && (x < 7)
ODER (*OR*)	||	b || x > 8
Negation (*NOT*)	!	!b

(Die logischen Operatoren werden in Kapitel 5 noch einmal im Detail besprochen.)

Wahrheitswerte

Ein Ausdruck, der zu `true` oder `false` evaluieren kann, heißt *boolescher Ausdruck*

(Ursprung in der mathematischen Logik)

```
int x;
// Mehr Code (x wird verändert)
x < 7; // Evaluiere x
```

- `true` , falls der Wert von x kleiner 7
- `false` , sonst (x ist größer oder gleich 7)

Prüfung auf Gleichheit von Werten

- Bisher: $x < y \rightarrow$ „Ist x kleiner als y?"
- Problem: Das Gleichheitszeichen wird bereits verwendet

```
int i = 10;
```

- Lösung: Vergleichsoperation verwendet stattdessen zwei aufeinanderfolgende Gleichheitszeichen (==)

```
int x = 10;
x == 7; // Evaluiert zu false
```

- 🚨Vorsicht: Beliebte Quelle für subtile Fehler!

Prüfung auf Gleichheit von Werten

Welche Kombination von booleschem Ausdruck und Wert wird angenommen? cpp Run ▶

```cpp
#include <stdio.h>
int main() {
    int x = 10;
    if (x = 9) {
        printf("Evaluiert zu 'true' (x: %d)", x);
    } else {
        printf("Evaluiert zu 'false' (x: %d)", x);
    }
}
```

A: `true` und 10

B: `true` und 9

C: `false` und 10

D: `false` und 9

Prüfung auf Gleichheit von Werten

- C kennt nur `0` oder `1` (bzw. *nicht* `0`)

  → Falls Ausdruck *genau* `0`, dann `false`

  → Falls Ausdruck *ungleich* `0`, dann `true`

- Im Hintergrund implizite Umwandlung von skalarem Typ zu booleschem Ausdruck
- Historie
  - Ursprünglich gab es `true` / `false` nicht (wurde mit C++ eingeführt), stattdessen nur `0` / `1`
  - Später als Variablen eingeführt deren Werte `0` / `1` waren
  - Seit Version C23 sind `true` / `false` feste Schlüsselwörter

Beispiel - Prüfung auf Gleichheit

cpp Run ▶

```cpp
int main() {
    int x = 10;
    char c = 'c';
    if (x);
    if (0.000001);
    if (0.0);
    if (c);
    if ("Hello World");
    if ("");
}
```

Beispiel - Prüfung auf Gleichheit

cpp Run ▶

```cpp
int main() {
    int x = 10;
    char c = 'c';
    if (x); // true
    if (0.000001); // true
    if (0.0); // false (das entspricht genau 0)
    if (c); // true
    if ("Hello World"); // t
    if (""); // true (Litera
}
```

Zurück zum Kontrollfluss!

Programmverzweigungen

```c
#include <stdio.h>

int main() {
    int eingabe = 2, ergebnis = 0;

    if (eingabe % 2 == 0) {
        ergebnis = eing
    } else {
        ergebnis = eing
    }
    printf("ergebnis =
    return 0;
}
```

Wir wollen jetzt auf Teilbarkeit durch 3 prüfen!

Verkettungen von Programmverzweigungen

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int eingabe = 6, ergebnis = 0;

    if (eingabe % 2 == 0) {
        ergebnis = eingabe * eingabe;
    } else if (eingabe % 3 == 0) {
        ergebnis = eingabe - 3;
    } else {
        ergebnis = eingabe + eingabe;
    }
    printf("ergebnis = %d\n", ergebnis);
    return 0;
}
```

- Mehrere Alternativen sind mit `else if` möglich → Kette aus Optionen

Regeln zur Verkettungen

- Bedingungen **müssen** in Klammern gesetzt werden → `if (...)`
- Schließt die Verkettung mit `else` unbedingt ab

  → `else` hat daher keinen zugehörigen Ausdruck
- Der erste `if` -Ausdruck wird immer evaluiert
- Die erste Verzweigung mit wahrer Bedingungen wird genommen

  → Reihenfolge der Abfragen ist relevant!
- Ein `if` kann alleine stehenn; `else if` und `else` sind optional
- Achtung: Ein `else` bezieht sich nur auf das letzte `if`
- Der letzte Punkt ist eine häufige Fehlerquelle!

Dangling-Else

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int sum = 0;
    if (sum < 0) {
        sum += 10;
    }
    if ((sum + 10) > 100) {
        sum += 100;
    }
    else {
        printf("%d", sum);
    }
    return 0;
}
```

- Ein `else` bezieht sich nur auf das letzte `if`

Code-Blöcke

cpp Run ▶

```cpp
#include <stdio.h>
int main() {
    int sum = -1;
    if (sum < 0)
        sum += 10;
        printf("Sum wurde um 10 erhöht");
    return 0;
}
```

- 🚨 Achtung
  - Zu einem `if` -Statement gehört **nur** die erste Anweisung danach
  - Alles weitere wird **nicht** mehr konditional ausgeführt!
  - Eine beliebte Fehlerquelle: fehlende Code-Blöcke

Code-Blöcke erstellen

cpp Run ▶

```cpp
#include <stdio.h>
int main() {
    int sum = -1;
    if (sum < 0) {
        sum += 10;
        printf("Sum wurde um 10 erhöht");
    }
    return 0;
}
```

- „Code-Blöcke fassen viele Anweisung zu einer zusammen”
- Nach `if` -Anweisung wird immer noch **nur eine** Anweisung ausgeführt
- Hier: ein Block stellt eine Anweisung dar

Zwischenstand

- Mittels Verschachtelung von `if` und `else` können mehrere Bedingungen geprüft werden
- Ziel jetzt: Variable `sum` auf zehn verschiedene Werte prüfen

```c
#include <stdio.h>

int main() {
    int sum = -1;
    if (sum == 0) {
        sum += 10;
        printf("Sum wurde um 10 erhöht");
    } else if (sum == 1) {
        sum += 10;
    } else if (sum == 2) {
        sum += 20;
    } else if (sum == 3) {
        sum += 21;
```

Switch-Cases

- `if-else` gut für wenige Fälle geeignet
- Es wird aber schnell unhandlich bzw. schlecht leserlich
- Lösung: Fallunterscheidung mittels `switch` und `case`
- Möglich für konstante Werte: `case 4711`
- Die zu einem Fall zugehörige Konstante wird *Label* genannt

Switch-Cases

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int sum = 0;
    switch (sum) {
        case -1:
            printf("-1\n");
            break;
        case 0:
            printf("0\n");
            break;
        default:
            printf("Kein passender Wert\n");
            break;
    }
    return 0;
}
```

Switch-Cases

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int sum = 0;
    switch (sum) {
        case 0:
            printf("0");
        case 1:
            printf("1");
        case 2:
            printf("2");
            break;
        default:
            printf("Kein passender Wert");
            break;
```

- Beliebter Fehler: Das Vergessen der `break` -Anweisung
- … oder Absicht 😀

Fallthrough bei *Switch-Case*

- Bekanntes Beispiel

```c
#include <stdio.h>

int main() {
    int sum = -1;
    if (sum == 0) {
        sum += 10;
    } else if (sum == 1) {
        sum += 10;
    } else if (sum == 2) {
        sum += 20;
    } else if (sum == 3) {
        sum += 20;
    } else if (sum == 4) {
```

- Was können wir hier mit einer `switch`-Anweisung verbessern?

Beispiel - *Fallthrough* bei *Switch-Case*

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int sum = -1;
    switch (sum) {
        case 0:
        case 1:
            sum += 10;
            break;
        case 2:
        case 3:
            sum += 20;
            break;
        case 4:
        case 5:
```

Konditionaler Operator

- Beispiel: Eine einfache `if-else`-Verzweigung

```
int x = 3, y = 0;
if (x == 3) {
    y = 42;
} else {
    y = 0;
}
```

- Informatiker:innen sind schreibfaul 😀
- Das geht kürzer!

Konditionaler Operator

- Für ein einfaches `if-else` gibt es in C den ternären/konditionalen Operator
- Braucht als einziger Operator 3 Bestandteile (→ Namensherkunft)
- Syntax: expression1 ? expression2 : expression3
- expression1 wird ausgewertet und als `bool` betrachtet
  - Falls `expression1 == true` → wird `expression2` ausgeführt
  - Falls `expression1 == false` → wird `expression3` ausgeführt
- expression2/expression3: Beliebiger Ausdruck (Zahl/Wert, Code-Schnipsel, …)

Beispiel - Konditionaler Operator

C Run ▶

```c
#include <stdio.h>

int main() {
    bool b = true;
    int i = 0;

    b ? printf("1\n") : printf("2\n");

    i = b ? 3: 4;
    printf("%d\n", i);
    return 0;
}
```

Zwischenfazit

- Neue Fähigkeit: Abhängig vom Zustand den Programmablauf beeinflussen
  - `if-else` dient der bedingten Ausführung von Code-Blöcken
  - `switch-case` vereinfacht den Umgang mit vielen Verzweigungen
  - Beides wertvolle und häufiggenutzte Werkzeuge
- Zur Erinnerung: Problemstellungen zu Beginn der Vorlesung
  - Es ist nur genau eine Abfolge von Befehlen möglich ✅
  - Gleiche Schritte müssen jedes Mal **erneut** getippt werden
- Das können wir jetzt besser! 😀

Exkurs: Einfachster Rücksprung mit `goto`

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int sum = 0;
ziel: // Label wie bei Switch-Case
    sum += 10;
    if (sum > 4711) {
        printf("%d", sum);
        return 0; // Ende von main()
    }
    else {
        goto ziel; // Sprung zum benannten Label
    }
    return 0;
}
```

Exkurs: Einfachster Rücksprung mit `goto`

- `goto` sieht auf den ersten Blick relativ harmlos und nützlich aus
- 🚨 Achtung: Es ist das Tor zur Hölle!
- Mit großem Abstand das **schlimmste** Schlüsselwort in C
  - der Sprung erfolgt unbedingt
  - man **überall** hinspringen!
  - chaotische Programmabläufe trivial möglich
- Das Schlimmste: Es gibt (fast) immer bessere Alternativen → Schleifen

🚨 Verwendet **niemals** `goto`! 🚨

Schleifen

- Deutlich besser als `goto` → Schleifen
- Erlauben prinzipiell **beliebige** Wiederholung von Programmabschnitten
- An Schleife geknüpft: Boolescher Ausdruck
  - Bestimmt, ob Schleife (erneut) durchlaufen wird
  - Schleife läuft solange der Ausdruck `true` ist
- Es existieren mehrere Variationen in C
  - `while`
  - `do-while`
  - `for`

While-Schleife

- Besteht aus einem …
  - Schleifenkopf mit booleschem Ausdruck (*Head*) und
  - einem Schleifenrumpf (*Body*)
- Schleifenkopf enthält die Laufbedingung

```
#include &lt;stdio.h&gt;

int main() {
    int i = 0;          // Laufvariable

    while (i < 10) {    // Auswertung *vor* Betreten des Rumpfes
        ++i;            // Fortschritt
        printf("%d\n", i);
    }                   // Springt zurück zum Kopf
    return 0;
}
```

Beispiel - *While*-Schleife

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 0;           // Laufvariable

    while (i < 10) {     // Auswertung *vor* Betreten des Rumpfes
        ++i;             // Fortschritt
        printf("%d\n", i);
    }                    // Springt zurück zum Kopf
    return 0;
}
```

Abbruch einer *While*-Schleife

- Vorzeitiges Verlassen von Schleifen ist möglich mithilfe von `break`

  (→ Kennt Ihr schon von `switch-case`)

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 0;
    while (i < 10) {
        ++i;
        if (i == 3) {
            break;
        }
    }
    printf("%d\n", i);
    return 0;
}
```

While-Schleife

cpp Run ▶

```cpp
#include <stdbool.h>  // Für C-Standard < C23
#include <stdio.h>

int main() {
    int i = 0;
    while (true) {          // Potenziell unendlich
        ++i;
        if (i >= 3) {
            break;          // Stattdessen Kontrolle mit break
        }
    }
    printf("%d\n", i);
    return 0;
}
```

Do-While-Schleife

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 0;
    do {
        ++i;
    } while (i < 2);
    printf("%d\n", i);
    return 0;
}
```

- Variation der `while` -Schleife
  - Unterschied: Rumpf **vor** „Kopf”
  - Wird mind. einmal ausgeführt
  - Nennt sich *post-checked* (vs. *pre-checked*)

For-Schleife

```
int main() {
    int i = 0, k = 0;
    while (i < 10) {
        k += i;
        ++i;
    }
    return 0;
}
```

- `while` -Schleife ist gut, im Allgemeinen aber mehr Schreibaufwand
- Hier bedeutet dies:
  - Erst die Laufvariable deklarieren und initialisieren,
  - am Ende die Laufvariable erhöhen und dann
  - die Laufbedingung prüfen
- `for` -Schleife bietet etwas andere und **lesbarere** Schreibweise

- Funktional sind sie aber gleich!

For-Schleife

for(*init-clause*; *cond-expression*; *iteration-expression*) { *loop-statement* }

- *init-clause:* Variablendeklaration/ -initialisierung
  - Findet **vor** dem ersten Durchlauf statt
  - Variablen sind nur für die Dauer der Schleife gültig
- *cond-expression:* Laufbedingung
  - Boolescher Ausdruck, wird **vor** dem Betreten des Rumpfes evaluiert
  - Wird **vor** jedem Durchlauf geprüft → *pre-checked*
- *iteration-expression*: Fortsetzung
  - Wird **nach** jedem Durchlaufen des Rumpfes ausgeführt
  - Intendierter Zweck: Zählvariablenwert verändern.
- *loop-statement:* Reihe von Anweisungen - wie bisher. Darf nicht leer sein

Umwandlung *while*- zu *for*-Schleife

```
int main() {
    int i = 0, k = 0;
    while (i < 10) {
        k += i;
        ++i;
    }
    return 0;
}
```

```
int main() {
    int k = 0;
    for (int i = 0; i < 10; i++) {
        k += i;

    }
    return 0;
}
```

- Beide Schleifen verhalten sich gleich
- Schreibweise der *for*-Schleife ist kompakter

Beispiel - Gültigkeit in *for*-Schleifen

cpp Run ▶

```cpp
#include <stdio.h>
int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d ", i); // 0 1 2 3 4 5 6 7 8 9
    }
    // Laufvariable i ist ab hier nicht mehr bekannt
}
```

Beispie - Mehrere Variablen in *For*-Schleife

cpp Run ▶

```cpp
#include <stdio.h>
int main() {
    for (int i = 0, j = 2; i < 10; ++i, j += 2) {
        j -= 1;
        printf("%d %d, ", i, j);
    }
}
```

- Verwendung mehrerer Variablen ist natürlich auch möglich
- Einfügen in jeweiligen Abschnitt und mit Komma getrennt

Vergleich *While*- und *For*-Schleife

- `for` und `while` sind funktionsgleich!

```cpp
#include <stdio.h>
int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d ", i);


    }
}
```

```cpp
#include <stdio.h>
int main() {
    int i = 0;
    while (i < 10) {
        printf("%d ", i);
        ++i;
    }
}
```

Beispiele - Minimalen Schleifen

- Auch die minimalen Schleifen sind funktionsgleich

```
int main() {
    for (;;) {
        ;
    };
}
```

```
int main() {
    while(true) {
        ;
    };
}
```

```
int main() {
    do {
        ;
    } while (true);
}
```

- Einblick in das Verhalten des *Compilers*

```
int main() {
    // Wenn dies geschrieben wird...
    for (;;) { ; }
    // ... macht der Compiler das hier daraus (Ausdruck wird nicht Null)
    for (; 1;) { ; }
}
```

Zusammenfassung

- Wir können den Kontrollfluss eines Programms steuern 😎
- Verzweigungen mit `if`–`else` bestimmen, welcher Code ausgeführt wird
- `switch`–`case` als sinnvolle Alternative zu langen `if`–`else` -Ketten
- Mehrfache Ausführung von [Arbeitsschritten]{.sysgreen] mit Schleifen
  - `while` - und `do`–`while` -Schleifen
  - `for` -Schleifen zur Kapslung der relevanten Teil in einer Zeile

Ausblick

- Wir müssen dank Schleifen nicht mehr jeden Schleifendurchlauf manuell tippen 🥳
- … aber: Was ist, wenn wir die Schleife mehrfach nutzen wollen?

```
int main() {
    int sum = 0, start = 0, end = 10;
    for (int i = 0; i <= 10; ++i) {
        sum += i;
    }
    // Zusätzlich wird Summe von Intervall [10,35] benötigt
}
```

Ausblick

- Wir müssen dank Schleifen nicht mehr jeden Schleifendurchlauf manuell tippen 🎉
- … aber: Was ist, wenn wir die Schleife mehrfach nutzen wollen?

```
int main() {
    int sum = 0, start = 0, end = 10;
    for (int i = 0; i <= 10; ++i) {
        sum += i;
    }
    sum = 0;
    for (int i = 10; i <= 35; ++i) {
        sum += i;
    }
}
```

- Wir müssen die Schleife trotzdem kopieren 😕
- Wir haben das Problem teilweise gelöst:

  Für unterschiedliche Startwerte muss trotzdem jedes Mal neu getippt werden (✅)

- Bessere Wiederverwendung mittels Funktionen → Nächstes Kapitel

Kapitel 03 - Kontrollfluss