

§3 KONTROLLSTRUKTUREN – ALLGEMEINES

Leitideen: Die Syntax der Kontrollstrukturen in C++ soll möglichst wenige reservierte Worte verwenden, daher werden Bedingungen geklammert und mehrere abhängige Anweisungen gegebenenfalls zu einer Blockanweisung zusammengefasst.

Die Laufanweisung (`for`) kann auf die einfachste Wiederholungsanweisung (`while`) zurückgeführt werden. Wegen ihrer einprägsamen Syntax ist sie beliebter als die elementarere Anweisung.

Sprünge (`goto`) sollten im allgemeinen vermieden werden, spezielle Sprunganweisungen (`break`, `continue`) sind aber nützlich, um Wiederholungsanweisungen zu beenden oder unmittelbar fortzusetzen.

Der Auswahlanweisung (`switch`) ist ihre potentielle Implementierung mittels Sprunganweisungen deutlich anzusehen.

§3 KONTROLLSTRUKTUREN - THEMENÜBERSICHT

- if-Anweisung – Ergänzungen
- while-Anweisung – Ergänzungen
- do-while-Anweisung – Anmerkungen und Beispiel (ggT)
- for-Anweisung – Anmerkungen, Kommaausdruck, Beispiel (Binomialkoeffizient)
- Sprunganweisungen (goto, break, continue, return)
- switch-Anweisung – typische Verwendung

if-Anweisung - Ergänzungen

Unterschied zu anderen Programmiersprachen:

- ▶ Die Bedingung braucht kein logischer Ausdruck zu sein, zulässig ist auch ein arithmetischer Ausdruck [oder ein Zeigerausdruck], weil dieser automatisch in `bool` umgewandelt wird.
- ▶ Kein `then`, statt dessen Klammerung des Bedingungs-
ausdrucks.
- ▶ Die Pascalregel „Kein Strichpunkt vor `else`“ gilt *nicht*!
In Pascal trennen Strichpunkte Anweisungen.
In C++ schließen Strichpunkte bestimmte Anweisungen
ab, allerdings nicht die meisten Kontrollstrukturen.

Bsp. `if (a>b) x=a; else x=b; okay`
 `if (a>b) {x=a;} else {x=b;} okay`
 `if (a>b) {x=a;}; else {x=b;}; falsch`

if-Anweisung - Ergänzungen II

Problem: Syntaktische Zweideutigkeit bei

```
if (B1) if (B2) A1 else A2
```

if (B1)	<i>oder</i>	if (B1)	?
if (B2)		if (B2) A1	
A1		else	
else		A2	
A2			

- ▶ *Regel:* Der else-Zweig wird der letzten vorangehenden if-Anweisung ohne else zugeordnet.
- ▶ Die Einrückungen im linken Beispiel entsprechen der syntaktischen Bedeutung.
- ▶ Einrückungen haben selbstverständlich *keinen* Einfluss auf syntaktische Struktur des Programms . Sie dienen allein der Erhöhung der Verständlichkeit eines Programms für einen menschlichen Leser, und sind daher *sehr* zu empfehlen.

if-Anweisung - Ergänzungen III

Empfehlung: Vermeidung potentieller Fehlinterpretationen durch geeignete Klammerung der abhängigen Anweisungen

if (B1)		if (B1)
{ if (B2)		{ if (B2) A1 }
A1	<i>bzw.</i>	else
else		A2
A2		
}		

- Die geschweiften Klammern links könnten weggelassen werden, ohne die Bedeutung zu ändern.

if-Anweisung - Ergänzungen IV

else-if-Kaskaden

- ▶ Seien $B1, \dots, BN$ Bedingungen, die *nacheinander* überprüft werden sollen, bis eine davon erfüllt ist. Dann soll jeweils die entsprechende der Anweisungen $A1, \dots, AN$ ausgeführt werden.
- ▶ Optional: Falls *keine* davon zutrifft, soll die Anweisung A ausgeführt werden.

```
if (B1)
    A1
else if (B2)
    A2
    :
else if (BN)
    AN
// Optional
else
    A
```

- ▶ Bei *else-if-Kaskaden* handelt es sich — trotz der Schreibweise — um geschachtelte *if-else*-Anweisungen, *nicht* wie in anderen Programmiersprachen um eine eigenständige Kontrollstruktur.
- ▶ *Empfehlung:* Zur Vermeidung von Fehlinterpretationen $A1, \dots, AN$ mit geschweiften Klammern umgeben.

while-Anweisung - Ergänzungen

Unterschied zu anderen Programmiersprachen:

- ▶ Die Bedingung braucht kein logischer Ausdruck zu sein, zulässig ist auch ein arithmetischer Ausdruck [oder ein Zeigerausdruck], weil dieser automatisch in `bool` umgewandelt wird.
- ▶ Kein `do`, statt dessen Klammerung des Bedingungs-
ausdrucks.

Iterationsverfahren:

- ▶ Geg.: $I \subset \mathbb{R}$, $\phi : I \rightarrow I$, Startwert $x_0 \in I$,
 $x_{i+1} = \phi(x_i)$ ($i \in \mathbb{N}_0$).
- ▶ Häufig: Abbruch, falls x_i und x_{i+1} „nahe“ beieinander liegen.
- ▶ Für die Durchführung reichen 2 Variablen (Umspeicherungstechnik).

while-Anweisung - Ergänzungen II

Bsp.: $x_0 = 0$, $x_{i+1} = \sqrt{a + x_i}$ ($i \in \mathbb{N}_0$) ($a \geq 0$ fest).

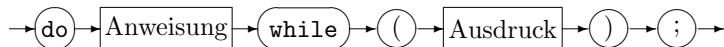
Abbruchkrit.: $|x_{i+1} - x_i| \leq \varepsilon$ oder $i > itmax$,

d.h. Wdhl. solange wie $|x_{i+1} - x_i| > \varepsilon$ und $i \leq itmax$.

Programmfragment:

```
i=0; xa=0; xn=sqrt(a);  
// xa:  $x_0$    xn:  $x_1$   
while (abs(xn-xa)>eps && i<=itmax) {  
    // xa:  $x_i$    xn:  $x_{i+1}$   
    // Richtig fuer  $i = 0$   
    xa=xn;  
    // xa:  $x_{i+1}$   
    xn=sqrt(a+xa);  
    // xn:  $x_{i+2}$   
    ++i;  
    // xa:  $x_i$    xn:  $x_{i+1}$   
}
```


do-while-Anweisung - Anmerkungen



- ▶ Unterschied zur while-Anweisung: Überprüfung der Bedingung erst *nach* Ausführung der abhängigen Anweisung.
- ▶ Daher: Abhängige Anweisung wird mindestens einmal ausgeführt.
- ▶ do-while-Anweisung endet mit einem Strichpunkt!
- ▶ Eher seltene Verwendung der do-while-Anweisung.

Caveat

- ▶ `while(...); {...}`
Strichpunkt falsch!
- ▶ `do {...} while(...);`
Strichpunkt notwendig!

Euklidischer Algorithmus – ggT(a,b)

Verfahren

$$\left. \begin{array}{l} a_0 = a, \quad b_0 = b, \quad (a \in \mathbb{N}_0, b \in \mathbb{N}) \\ a_i = b_i q_i + r_i \text{ mit } 0 \leq r_i < b_i \\ a_{i+1} = b_i, \quad b_{i+1} = r_i \end{array} \right\} (i = 0, 1, 2, \dots, l \text{ bis } r_l = 0)$$

Es gilt $\text{ggT}(a, b) = a_{l+1}$ wegen

$$\begin{aligned} \text{ggT}(a_i, b_i) &= \text{ggT}(b_i, r_i) = \text{ggT}(a_{i+1}, b_{i+1}), \text{ also} \\ \text{ggT}(a, b) &= \text{ggT}(a_0, b_0) = \dots = \text{ggT}(a_{l+1}, \underbrace{b_{l+1}}_0) = a_{l+1} \end{aligned}$$

Programmfragment

```
do {  
    r = a%b;  
    a = b; b = r;  
} while (r!=0) ;  
cout << "ggT(a,b) = " << a << endl;
```

for-Anweisung - Anmerkungen

- Die for-Anweisung in C++ ist viel allgemeiner als die Lauffanweisungen in anderen Programmiersprachen.

for (Ausdruck1; Ausdruck2; Ausdruck3) Anweisung
init *condition* *update*
ist im wesentlichen äquivalent zu

```
Ausdruck1;  
while (Ausdruck2) {  
    Anweisung  
    Ausdruck3;  
}
```

Bsp.: Berechnung von a^n ($n \in \mathbb{N}$)

```
pot=1; for (i=1; i<=n; ++i) pot*=a;  
 $\hat{=}$  pot=1; i=1;  
while (i<=n) {  
    pot*=a;  
    ++i;  
}
```

for-Anweisung - Anmerkungen II

- ▶ Im vorigen Beispiel würde man allerdings eher schreiben:
`for (i=0; i<n; ++i)` statt `for (i=1; i<=n; ++i)`
- ▶ Grund: Indizes eines Vektors mit n Komponenten laufen von $0, \dots, n-1$.

Bsp.: `vector<double> a(n);`
// Definiert Vektor a mit Komponenten
// $a[0], \dots, a[n-1]$
`for (i=0; i<n; ++i) a[i]=sqrt(i*i+1);`
// $a_i = \sqrt{i^2 + 1}$ ($i = 0, \dots, N-1$)

- ▶ An Stelle von `Ausdruck1`; kann auch *eine* einfache Vereinbarung stehen. Ihr Gültigkeitsbereich erstreckt sich *nur* bis zum Ende der for-Anweisung

Bsp.: `pot=1; for (int i=0; i<n; ++i) pot*=a;`

- ▶ Vergleichbare Syntaxerweiterungen existieren für die Bedingungen der if-Anweisung und der while-Anweisung, erscheinen aber wenig nützlich.

Komma-Ausdruck

Wirkung: Zusammenfassung mehrerer Ausdrücke zu einem einzigen Ausdruck.

- ▶ Ermöglicht die Ausführung mehrerer Seiteneffekte in einem Ausdruck, z.B. im Initialisierungs- oder Updateteil der for-Anweisung.
- ▶ Wert des Kommaausdrucks ist der Wert des *rechten* Operanden.
- ▶ Die Auswertung erfolgt von links nach rechts (d.h. der linke Operand wird vor dem rechten Operanden ausgewertet)
- ▶ Der Kommaoperator ist linksassoziativ.
- ▶ Durch Komma getrennte Funktionsargumente oder Variablendeklarationen bilden *keine* Kommaausdrücke.

Bsp.: `y = (x=3, x+2); // x=3; y=5;`

Caveat

- ▶ `pi=3,14159; // pi=3 !!`
[Zuweisungsoperator bindet stärker als der Kommaoperator]

Binomialkoeffizient

$$\binom{n}{k} = \frac{n(n-1) \cdots (n-k+1)}{1 \cdot 2 \cdots k} \quad (k, n \in \mathbb{N})$$

Schrittweise Berechnung

$$\binom{n}{k} = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdots \frac{n-k+1}{k}$$

Durchführung in ganzzahliger Arithmetik möglich, denn:

$$\underbrace{\frac{n(n-1) \cdots (n-i+1)}{1 \cdot 2 \cdots i}}_{\binom{n}{i}} \cdot \frac{n-i}{i+1} = \binom{n}{i+1},$$

weil Binomialkoeffizienten ganzzahlig sind

Programmfragment

```
int n,k,b,i; cin >> n >> k;
for (i=1,b=1; i<=k; ++i) b=b*n/i;
```

Sprunganweisungen - Ergänzungen

goto-Anweisung

- ▶ Sprünge sind nur innerhalb einer Funktion möglich.
- ▶ Vereinbarung von Sprungmarken *nicht* erforderlich; Voranstellen von z.B. `marke:` vor eine Anweisung definiert die Sprungmarke `marke`.
- ▶ Sprungmarken sind syntaktisch Namen.
- ▶ Sprungmarken bilden einen eigenen Namensraum.
- ▶ `goto` sparsam verwenden (Übersichtlichkeit!)
- ▶ Stattdessen: Wiederholungsanweisungen, bedingte Anweisungen und Auswahlanweisungen vorziehen.
- ▶ Herausspringen aus bzw. unmittelbares Fortsetzen von Wiederholungsanweisungen vorzugsweise mit `break`- bzw. `continue`-Anweisung.
- ▶ Beispiel für eine sinnvolle `goto`-Anweisung auf Infobl. 6/3.
- ▶ Java: kein `goto`!

Sprunganweisungen - Ergänzungen II

break-Anweisung

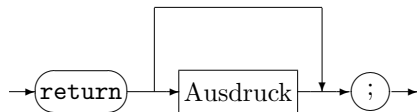
- ▶ Nur innerhalb von Wiederholungs- und Auswahlanweisungen möglich.
- ▶ Heraussprung erfolgt aus der am engsten umschließenden Wiederholungsanweisung (bzw. Auswahlanweisung).
- ▶ Beispiel für typische Verwendung auf Infobl. 6/3.
- ▶ Herausspringen aus *mehreren* geschachtelten Wiederholungsanweisungen (bzw. Auswahlanweisungen) ist mit `break` *nicht* möglich.

continue-Anweisung

- ▶ Nur innerhalb von Wiederholungsanweisungen möglich.
- ▶ Setzt die engste umschließende Wiederholungsanweisung unmittelbar fort. (Wirkung siehe Infobl. 6/4.)
- ▶ Wird erheblich seltener als die break-Anweisung verwendet, ist dennoch gelegentlich nützlich.

Sprunganweisungen - Ergänzungen III

return-Anweisung



- ▶ Bewirkt Beendigung der aufgerufenen Funktion und Rücksprung in die aufrufende Funktion.
- ▶ Der Wert des Ausdruck nach `return` wird Funktionswert. Fehlt genau dann, wenn Funktion Ergebnistyp `void` hat.
- ▶ In `main` bewirkt `return` die Beendigung des Hauptprogr. An das aufrufende Programm (z.B. Kommandointerpreter, Shell) wird der Wert des `return`-Ausdrucks übergeben. (Konvention: 0 – Erfolg, > 0 – Fehlerfall)
- ▶ In `main`: `return n` ähnliche Wirkung wie `exit(n)`. Daher findet man oft: `return(n)` statt `return n` (Anmerkung: `#include <cstdlib>` bei Verwendung von `exit` erforderlich)

switch-Anweisung - typische Verwendung

```
switch (Ausdruck) {  
    case Konst1: Anweisungen1 break;  
    case Konst2: Anweisungen2 break;  
                :  
    default:    Anweisungen break;  
}
```

- ▶ `case Konst` wirkt als Sprungmarke, zu der verzweigt wird, wenn der Ausdruck den Wert `Konst` annimmt. Bei einem vergessenen `break` wird deshalb die nächste Anweisung ausgeführt.
- ▶ Das letzte `break` ist nicht notwendig, aber üblich, weil dann problemlos weitere `case`-Anweisungen angefügt werden können.
- ▶ Mehrfache `case`-Marken sind möglich (genauso wie mehrfache Sprungmarken)
- ▶ Der Ausdruck muss ganzzahlig sein (Zeichenketten sind nicht möglich).