

5. Kontrollanweisungen II

Sichtbarkeit, Lokale Variablen, While-Anweisung,
Do-Anweisung, Sprunganweisungen

Sichtbarkeit

Deklaration in einem Block ist ausserhalb des Blocks nicht „sichtbar“.

```
int main ()  
{  
  {  
    int i = 2;  
  }  
  std::cout << i; // Fehler: undeklariertes Name  
  return 0;  
}
```

„Blickrichtung“



Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke

```
int main()
{
    block | for (unsigned int i = 0; i < 10; ++i)
           |     s += i;
           |     std::cout << i; // Fehler: undeklariertes Name
           |     return 0;
}
```

Gültigkeitsbereich einer Deklaration

Potenzieller Gültigkeitsbereich: Ab Deklaration bis Ende des Programmteils, der die Deklaration enthält.

Im Block

```
{  
    int i = 2;  
    ...  
}
```

scope

Im Funktionsrumpf

```
int main()  
{  
    int i = 2;  
    ...  
    return 0;  
}
```

scope

In Kontrollanweisung

```
for ( int i = 0; i < 10; ++i ) { s += i; ... }
```

scope

Gültigkeitsbereich einer Deklaration

Wirklicher Gültigkeitsbereich einer Deklaration = Potenzieller Gültigkeitsbereich minus darin enthaltene potenzielle Gültigkeitsbereiche von Deklarationen des gleichen Namens

```
int main()
{
  int i = 2;
  for (int i = 0; i < 5; ++i)
    // outputs 0,1,2,3,4
    std::cout << i;
  // outputs 2
  std::cout << i;
  return 0;
}
```

Gültigkeit von i

i₂ in for

in main

Automatische Speicherdauer

Lokale Variablen (Deklaration in Block)

- werden bei jedem Erreichen ihrer Deklaration neu „angelegt“, d.h.
 - Speicher / Adresse wird zugewiesen
 - evtl. Initialisierung wird ausgeführt
- werden am Ende ihrer deklarativen Region „abgebaut“ (Speicher wird freigegeben, Adresse wird ungültig)

Lokale Variablen

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs 6, 7, 8, 9, 10
        int k = 2;
        std::cout << --k; // outputs 1, 1, 1, 1, 1
    }
}
```

Lokale Variablen (Deklaration in einem Block)
haben *automatische Speicherdauer*.

while Anweisung

```
while ( condition )  
    statement
```

- *statement*: beliebige Anweisung, Rumpf der **while** Anweisung .
- *condition*: konvertierbar nach **bool**.

while Anweisung

```
while ( condition )  
    statement
```

ist äquivalent zu

```
for ( ; condition ; )  
    statement
```

while-Anweisung: Semantik

```
while ( condition )  
    statement
```

- *condition* wird ausgewertet
 - **true**: Iteration beginnt
statement wird ausgeführt
 - **false**: **while**-Anweisung wird beendet.
- 

while-Anweisung: Warum?

- Bei `for`-Anweisung ist oft expression allein für den Fortschritt zuständig („Zählschleife“)

```
for (unsigned int i = 1; i <= n; ++i)
    s += i;
```

- Falls der Fortschritt nicht so einfach ist, kann `while` besser lesbar sein.

Beispiel: Die Collatz-Folge

$(n \in \mathbb{N})$

■ $n_0 = n$

■ $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, ... (Repetition bei 1)

Die Collatz-Folge in C++

```
// Program: collatz.cpp
// Compute the Collatz sequence of a number n.

#include <iostream>

int main()
{
    // Input
    std::cout << "Compute the Collatz sequence for n =? ";
    unsigned int n;
    std::cin >> n;

    // Iteration
    while (n > 1) {
        if (n % 2 == 0)
            n = n / 2;
        else
            n = 3 * n + 1;
        std::cout << n << " ";
    }
    std::cout << "\n";
    return 0;
}
```

Die Collatz-Folge in C++

n = 27:

82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107,
322, 161, 484, 242, 121, 364, 182, 91, 274, 137,
412, 206, 103, 310, 155, 466, 233, 700, 350, 175,
526, 263, 790, 395, 1186, 593, 1780, 890, 445,
1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566,
283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719,
2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644,
1822, 911, 2734, 1367, 4102, 2051, 6154, 3077,
9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300,
650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23,
70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4,
2, 1

Die Collatz-Folge

Erscheint die 1 für jedes n ?

- Man vermutet es, aber niemand kann es beweisen!
- Falls nicht, so ist die **while**-Anweisung zur Berechnung der Collatz-Folge für einige n theoretisch eine Endlosschleife!

do Anweisung

```
do  
    statement  
while ( expression );
```

- *statement*: beliebige Anweisung, Rumpf der **do** Anweisung .
- *expression*: konvertierbar nach **bool**.

do Anweisung

```
do  
    statement  
while ( expression );
```

ist äquivalent zu

```
statement ;  
while ( expression )  
    statement;
```

do-Anweisung: Semantik

```
do  
  statement  
while ( expression );
```

- Iteration beginnt ←
 - *statement* wird ausgeführt.
 - *expression* wird ausgewertet
 - **true**: Iteration beginnt
 - **false**: do-Anweisung wird beendet.
-

do-Anweisung: Beispiel Taschenrechner

Summiere ganze Zahlen (bei 0 ist Schluss):

```
int a;          // next input value
int s = 0;     // sum of values so far
do {
    std::cout << "next number =? ";
    std::cin >> a;
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0);
```

Zusammenfassung

- Auswahl (bedingte *Verzweigungen*)
 - `if`: und `if-else`-Anweisung
- Iteration (bedingte *Sprünge*)
 - `for`-Anweisung
 - `while`-Anweisung
 - `do`-Anweisung
- Blöcke und Gültigkeit von Deklarationen

Sprunganweisungen

- `break`
- `continue`

break-Anweisung

```
break;
```

- umschliessende Iterationsanweisung wird sofort beendet.
- nützlich, um Schleife „in der Mitte“ abbrechen zu können⁵

⁵und unverzichtbar bei switch-Anweisungen.

Taschenrechner mit `break`

Summiere ganze Zahlen (bei 0 ist Schluss):

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    // irrelevant in letzter Iteration:
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0);
```

Taschenrechner mit `break`

Unterdrücke irrelevante Addition von 0:

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0)
```

Taschenrechner mit `break`

Äquivalent und noch etwas einfacher:

```
int a;
int s = 0;
for (;;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    std::cout << "sum = " << s << "\n";
}
```

Taschenrechner mit `break`

Version ohne `break` wertet `a` stets zweimal aus und benötigt zusätzlichen Block.

```
int a = 1;
int s = 0;
for (; a != 0;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a != 0) {
        s += a;
        std::cout << "sum = " << s << "\n";
    }
}
```

continue-Anweisung

```
continue;
```

- Kontrolle überspringt den Rest des Rumpfes der umschliessenden Iterationsanweisung
- Iterationsanweisung wird aber *nicht* abgebrochen

Taschenrechner mit `continue`

Ignoriere alle negativen Eingaben:

```
for (;;)
{
    std::cout << "next number =? ";
    std::cin >> a;
    if (a < 0) continue; // springe zu }
    if (a == 0) break;
    s += a;
    std::cout << "sum = " << s << "\n";
}
```

Äquivalenz von Iterationsanweisungen

Wir haben gesehen:

- **while** und **do** können mit Hilfe von **for** simuliert werden

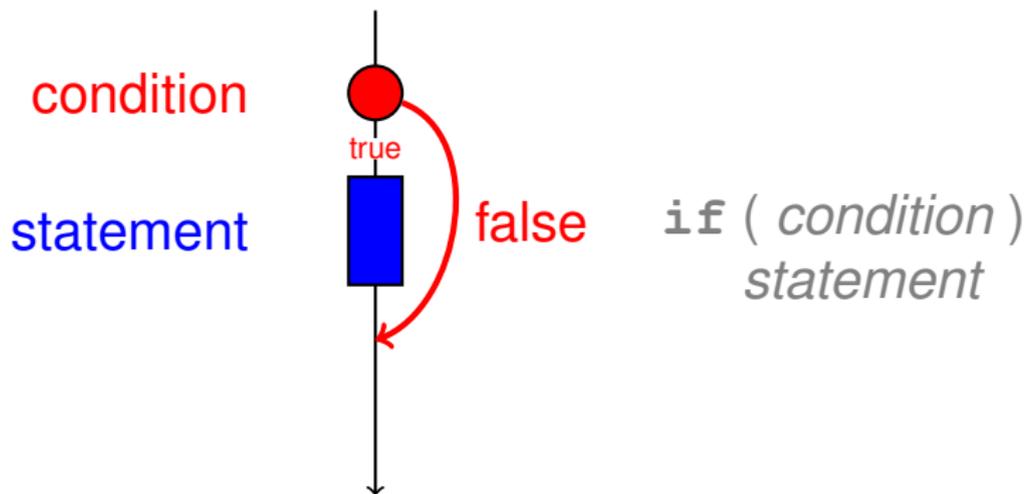
Es gilt aber sogar: Nicht ganz so einfach falls ein `continue` im Spiel ist!

- Alle drei Iterationsanweisungen haben die gleiche „Ausdruckskraft“ (Skript).

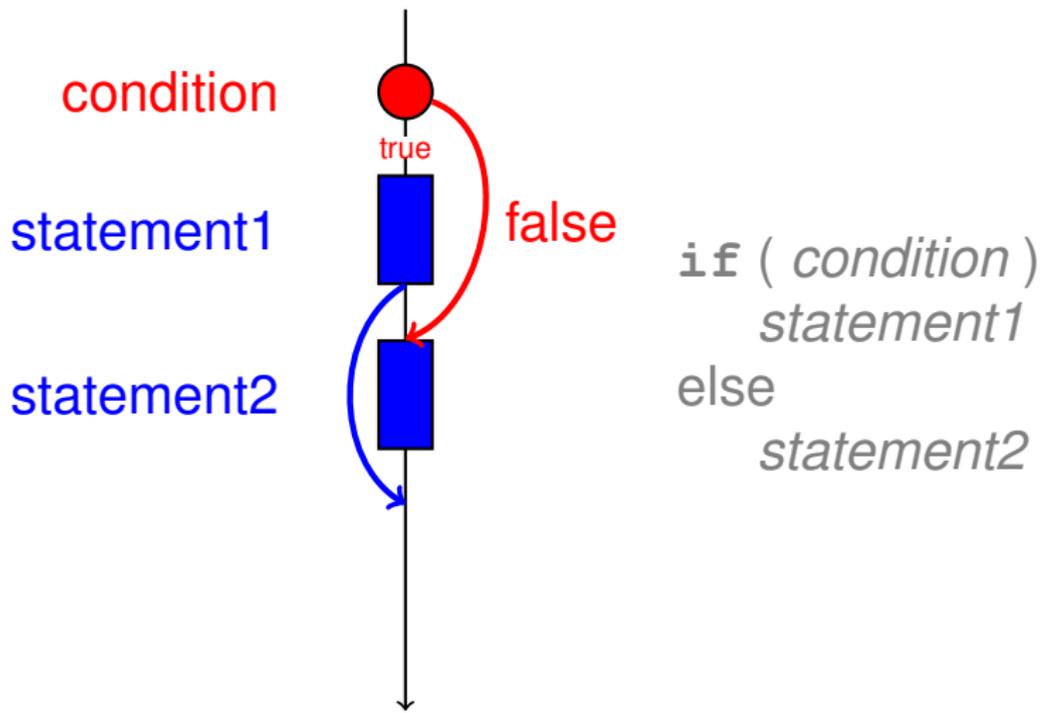
Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...
- ... ausser in Auswahl- und Kontrollanweisungen

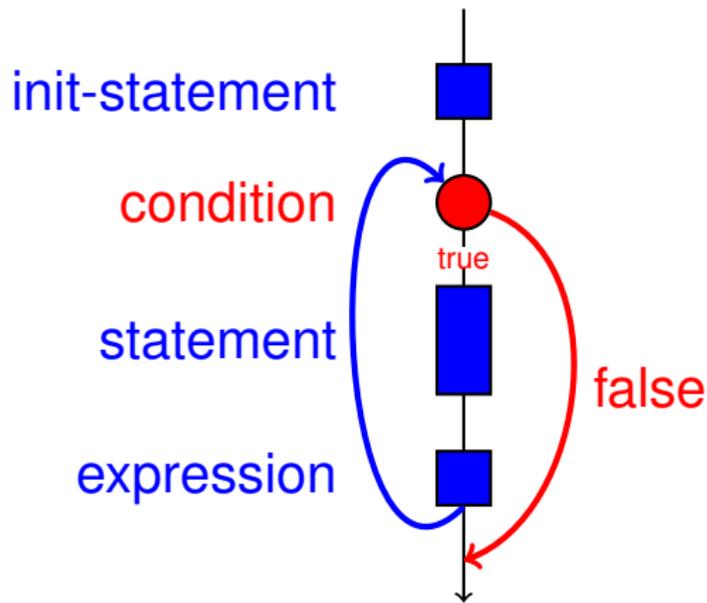


Kontrollfluss `if else`



Kontrollfluss `for`

`for` (*init statement* *condition* ; *expression*)
statement



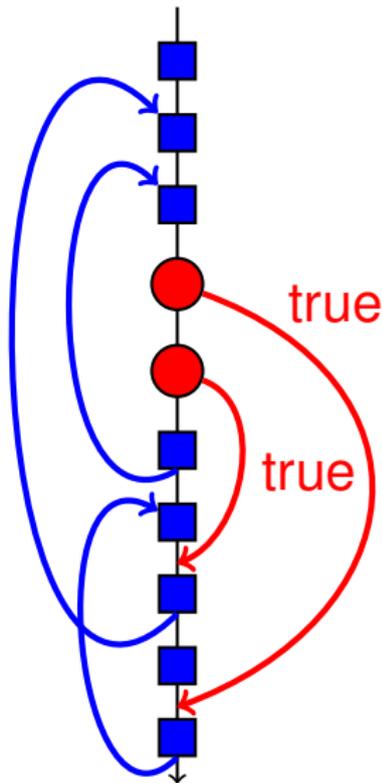
Spaghetti-Code mit goto

Ausgabe aller Primzahlen mit der Programmiersprache BASIC

```
10 N=2
20 D=1
30 D=D+1
40 IF N=D GOTO 100
50 IF N/D = INT(N/D) GOTO 70
60 GOTO 30
70 M=N+1
80 GOTO 20
100 PRINT N
110 GOTO 70
```



Computer C64



Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss
- Einfache Ausdrücke

Ziele sind oft nicht gleichzeitig erreichbar.

Ungerade Zahlen in $\{0, \dots, 100\}$

Erster (korrekter) Versuch:

```
for (unsigned int i = 0; i < 100; ++i)
{
    if (i % 2 == 0)
        continue;
    std::cout << i << "\n";
}
```

Ungerade Zahlen in $\{0, \dots, 100\}$

Weniger Anweisungen, *weniger* Zeilen:

```
for (unsigned int i = 0; i < 100; ++i)
{
    if (i % 2 != 0)
        std::cout << i << "\n";
}
```

Ungerade Zahlen in $\{0, \dots, 100\}$

Weniger Anweisungen, *einfacherer* Kontrollfluss:

```
for (unsigned int i = 1; i < 100; i += 2)
    std::cout << i << "\n";
```

Das ist hier die “richtige” Iterationsanweisung!

Sprunganweisungen

- realisieren unbedingte Sprünge.
- sind wie **while** und **do** praktisch, aber nicht unverzichtbar
- sollten vorsichtig eingesetzt werden: nur dort wo sie den Kontrollfluss *vereinfachen*, statt ihn *komplizierter* zu machen

6. Fließkommazahlen I

Typen `float` und `double`; Gemischte Ausdrücke
und Konversionen; Lücken im Wertebereich;

„Richtig Rechnen“

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```

Fixkommazahlen

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

0.0824 = 0000000.082 ← dritte Stelle abgeschnitten

Nachteile

- Wertebereich wird *noch* kleiner als bei ganzen Zahlen.
- Repräsentierbarkeit hängt von der Stelle des Kommas ab.

Fließkommazahlen

- feste Anzahl signifikante Stellen (z.B. 10)
- plus Position des Kommas

$$82.4 = 824 \cdot 10^{-1}$$

$$0.0824 = 824 \cdot 10^{-4}$$

- Zahl ist $\text{Signifikand} \times 10^{\text{Exponent}}$

Typen `float` und `double`

- sind die fundamentalen C++ Typen für Fließkommazahlen
- approximieren den Körper der reellen Zahlen $(\mathbb{R}, +, \times)$ in der Mathematik
- haben grossen Wertebereich, ausreichend für viele Anwendungen (`double` hat mehr Stellen als `float`)
- sind auf vielen Rechnern sehr schnell

Arithmetische Operatoren

Wie bei `int`, aber ...

- Divisionsoperator `/` modelliert „echte“ (reelle, nicht ganzzahlige) Division
- Keine Modulo-Operatoren `%` oder `%=`

Literale

Unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1

ganzzahliger Teil

Exponent

`1.27f` : Typ `float`, Wert 1.27

fraktionaler Teil

- und / oder Exponent.

`1e3` : Typ `double`, Wert 1000

`1.23e-7` : Typ `double`, Wert $1.23 \cdot 10^{-7}$

`1.23e-7f` : Typ `float`, Wert $1.23 \cdot 10^{-7}$

`1.23e-7f`

Rechnen mit float: Beispiel

Approximation der Euler-Zahl

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} \approx 2.71828 \dots$$

mittels der ersten 10 Terme.

Rechnen mit `float`: Eulersche Zahl

```
// Program: euler.cpp
// Approximate the Euler number e.

#include <iostream>

int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f;    // 1/i!
    float e = 1.0f;    // i-th approximation of e

    std::cout << "Approximating the Euler number...\n";
    // steps 1, ..., n
    for (unsigned int i = 1; i < 10; ++i) {
        t /= i;    // 1/(i-1)! -> 1/i!
        e += t;
        std::cout << "Value after term " << i << ": " << e << "\n";
    }

    return 0;
}
```

Rechnen mit float: Beispiel

Ausgabe:

```
Approximating the Euler number...  
Value after term 1: 2  
Value after term 2: 2.5  
Value after term 3: 2.66667  
Value after term 4: 2.70833  
Value after term 5: 2.71667  
Value after term 6: 2.71806  
Value after term 7: 2.71825  
Value after term 8: 2.71828  
Value after term 9: 2.71828
```

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

`9 * celsius / 5 + 32`

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...
- Wertebereich ist zusammenhängend (keine „Löcher“): \mathbb{Z} ist „diskret“.

Fließkommatypen:

- Über- und Unterlauf selten, aber ...
- es gibt Löcher: \mathbb{R} ist „kontinuierlich“.

Löcher im Wertebereich

```
// Input
```

```
float n1;  
std::cout << "First number      =? ";  
std::cin >> n1;
```

Eingabe 1.1

```
float n2;  
std::cout << "Second number     =? ";  
std::cin >> n2;
```

Eingabe 1.0

```
float d;  
std::cout << "Their difference =? ";  
std::cin >> d;
```

Eingabe 0.1

```
// Computation and output
```

```
std::cout << "Computed difference - input difference = "  
          << n1 - n2 - d << "\n";
```

Ausgabe 2.23517e-8

Was ist hier los?