

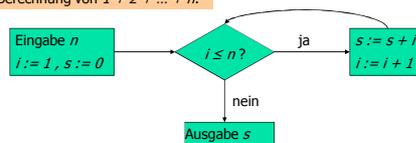
Kontrollanweisungen

Auswahlweisungen,
Iterationsanweisungen, Blöcke,
Sprunganweisungen

Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man "Verzweigungen" und "Sprünge"

Berechnung von $1 + 2 + \dots + n$.



Auswahlweisungen

realisieren "Verzweigungen"

- `if` Anweisung
- `if-else` Anweisung

`if`-Anweisung

```
if ( condition )
    statement
```

- *statement* : beliebige Anweisung (*Rumpf* der `if`-Anweisung)
- *condition* : konvertierbar nach `bool`

`if`-Anweisung

```
if ( condition )
    statement
```

Wenn *condition* Wert *true* hat, dann wird *statement* ausgeführt.

```
int a;
std::cin >> a;
if ( a % 2 == 0 )
    std::cout << "even";
```

`if-else` Anweisung

```
if ( condition )
    statement1
else
    statement2
```

- *condition* : konvertierbar nach `bool`
- *statement1* : Rumpf des `if`-Zweiges
- *statement2* : Rumpf des `else`-Zweiges

if-else Anweisung

```
if ( condition )
    statement1
else
    statement2

int a;
std::cin >> a;
if ( a % 2 == 0 )
    std::cout << "even";
else
    std::cout << "odd";
```

Layout:

← Einrückung!

← Einrückung!

Iterationsanweisungen

realisieren "Schleifen"

- for-Anweisung
- while-Anweisung
- do-Anweisung

Berechne $1 + 2 + \dots + n$

```
// Program: sum_n.cpp
// Compute the sum of the first n natural numbers.

#include <iostream>

int main()
{
    // input
    std::cout << "Compute the sum 1+...+n for n =? ";
    unsigned int n;
    std::cin >> n;

    // computation of sum_{i=1}^n i
    unsigned int s = 0;
    for (unsigned int i = 1; i <= n; ++i) s += i;

    // output
    std::cout << "1+...+" << n << " = " << s << ".\n";
    return 0;
}
```

for-Anweisung

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

- *init-statement* : Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition* : konvertierbar nach bool
- *expression* : beliebiger Ausdruck
- *statement* : beliebige Anweisung (*Rumpf* der for-Anweisung)

for-Anweisung

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

Deklarationsanweisung:

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

for-Anweisung

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

Ausdruck vom Typ `bool`:

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

for-Anweisung

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

Ausdruck vom Typ `unsigned int`:

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

for-Anweisung

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

Ausdrucksanweisung:

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

for-Anweisung: Semantik

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

- *init-statement* wird ausgeführt.
- *condition* wird ausgewertet.
 - *true* : Iteration beginnt.
 - *statement* wird ausgeführt.
 - *expression* wird ausgewertet.
 - *false* : `for`-Anweisung wird beendet.

for-Anweisung: Beispiel

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

Annahme: `n == 2`

```
s == 0  
s == 1 ← i == 1 true  
s == 3 ← i == 2 true  
                i == 3 false  
s == 3
```

Der kleine Gauss (1777-1855)

- Mathe-Lehrer wollte seine Schüler mit folgender Aufgabe beschäftigen:
"berechne die Summe der Zahlen 1 bis 100 !"
- Gauss war nach einer Minute fertig.

Der kleine Gauss (1777-1855)

- Die Lösung von Gauss:
 - gesuchte Zahl ist
 $1 + 2 + 3 + \dots + 98 + 99 + 100$
 - das ist die Hälfte von
 $1 + 2 + 3 + \dots + 98 + 99 + 100 + 100 + 99 + 98 + \dots + 3 + 2 + 1 =$
 $101 + 101 + 101 + \dots + 101 + 101 + 101$
100 mal

Der kleine Gauss (1777-1855)

- Die Lösung von Gauss:
 - gesuchte Zahl ist
 $1 + 2 + 3 + \dots + 98 + 99 + 100$
 - das ist die Hälfte von
 $1 + 2 + 3 + \dots + 98 + 99 + 100 +$
 $100 + 99 + 98 + \dots + 3 + 2 + 1 =$
10100

Antwort: 5050

for-Anweisung: Terminierung

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

Hier und meistens:

- expression* ändert einen Wert, der in *condition* vorkommt
- nach endlich vielen Iterationen hat *condition* Wert *false*: **Terminierung**

Endlosschleifen

- sind leicht zu produzieren:

```
for ( ; ; ) ;
```

↑
Leere *condition* hat Wert *true*

Endlosschleifen

- sind leicht zu produzieren:

```
for ( ; ; ) ;
```

↑
Leere *expression* hat keinen Effekt

Endlosschleifen

- sind leicht zu produzieren:

```
for ( ; ; ) ;
```

↑ ↑
Null-Anweisungen

Endlosschleifen

- sind leicht zu produzieren:

```
for ( e ; v ; e ) r ;
```

↑ ↑
Null-Anweisungen

- ...aber nicht automatisch zu erkennen.

Halteproblem

Satz (siehe VL von J. Hromkovic):

Es gibt kein C++ Programm, das für jedes C++ Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

Beispiel: Primzahltest

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, n-1\}$ ein Teiler von n ist.

```
unsigned int d;  
for (d = 2; n % d != 0; ++d);
```

Rumpf ist die Null-Anweisung!

Beispiel: Primzahltest

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, n-1\}$ ein Teiler von n ist.

```
unsigned int d;  
for (d = 2; n % d != 0; ++d);
```

Beobachtung 1:

Nach der `for`-Anweisung gilt $d \leq n$.

Beispiel: Primzahltest

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, n-1\}$ ein Teiler von n ist.

```
unsigned int d;  
for (d = 2; n % d != 0; ++d);
```

Beobachtung 2:

n ist Primzahl genau dann wenn $d = n$.

Beispiel: Primzahltest

```
// Program: prime.cpp  
// Test if a given natural number is prime.  
#include <iostream>  
  
int main ()  
{  
    // Input  
    unsigned int n;  
    std::cout << "Test if n>1 is prime for n =? ";  
    std::cin >> n;  
  
    // Computation: test possible divisors d  
    unsigned int d;  
    for (d = 2; n % d != 0; ++d);  
  
    // Output  
    if (d < n)  
        // d is a divisor of n in {2,...,n-1}  
        std::cout << n << " = " << d << " * " << n / d << ".\n";  
    else  
        // no proper divisor found  
        std::cout << n << " is prime.\n";  
    return 0;  
}
```

Blöcke

o gruppieren mehrere Anweisungen zu einer neuen Anweisung

```
{ statement1 statement2 ... statementN }
```

o Beispiele:

```
o int main() { ... } Block (Rumpf der main-Funktion)
```

Blöcke

- gruppieren mehrere Anweisungen zu einer neuen Anweisung

```
{ statement1 statement2 ... statementN }
```

- Beispiele:

- `int main(){...}` Block (Schleifenrumpf)
- ```
for (unsigned int i = 1; i <= n; ++i) {
 s += i;
 std::cout << "partial sum is " << s << "\n";
}
```

## Sichtbarkeit

Deklaration in einem Block ist ausserhalb des Blocks nicht "sichtbar".

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

## Kontrollanweisung "=" Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke:

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

## Deklarative Region...

...einer Deklaration:

Programmteil, in dem diese vorkommt:

- Block

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

## Deklarative Region...

...einer Deklaration:

Programmteil, in dem diese vorkommt:

- Block
- Kontrollanweisung

```
for (unsigned int i = 0; i < 10; ++i) s += i;
```

## Deklarative Region...

...einer Deklaration:

Programmteil, in dem diese vorkommt:

- Block
- Kontrollanweisung
- Funktionsrumpf

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

## Potentieller Gültigkeitsbereich

...einer Deklaration:  
 Programmteil, in dem diese potentiell sichtbar ist (ab Deklaration bis Ende der deklarativen Region):

```

int main()
{
 int i = 2;
}
int main()
{
 int i = 2;
 return 0;
}
for (unsigned int i = 0; i < 10; ++i) s += i;

```

## Gültigkeitsbereich...

...einer Deklaration:

- Programmteil, in dem diese sichtbar ist (d.h. benutzt werden kann)
- Meistens gleich dem potentiellen Gültigkeitsbereich...
- ...aber nicht immer!

```

#include <iostream>
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;
}

```

## Gültigkeitsbereich...

Deklaration des *gleichen Namens* im potentiellen Gültigkeitsbereich *einer* Deklaration ist erlaubt (aber nicht empfohlen).

```

#include <iostream>
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;
}

```

## Gültigkeitsbereich...

...einer Deklaration:

- Potentieller Gültigkeitsbereich...

```

#include <iostream>
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;
}

```

## Gültigkeitsbereich...

...einer Deklaration:

- Potentieller Gültigkeitsbereich *minus* potentielle Gültigkeitsbereiche von Deklarationen des gleichen Namens darin

```

#include <iostream>
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;
}

```

## 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)

## Automatische Speicherdauer

```
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
}
```

## while-Anweisungen

```
while (condition)
 statement
```

- *statement* : beliebige Anweisung (Rumpf der `while`-Anweisung)
- *condition* : konvertierbar nach `bool`

## while-Anweisungen

```
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

## while-Anweisung: Beispiel

Collatz-Folge für eine natürliche Zahl  $n$  :

- $n_0 = n$
- $n_i = \begin{cases} n_{i-1} / 2, & \text{falls } n_{i-1} \text{ gerade} \\ 3 n_{i-1} + 1, & \text{falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1$

$n = 5$ : 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

## while-Anweisung: Beispiel

Collatz-Folge für eine natürliche Zahl  $n$ :

- $n_0 = n$
- $n_i = \begin{cases} n_{i-1} / 2, & \text{falls } n_{i-1} \text{ gerade} \\ 3 n_{i-1} + 1, & \text{falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1$

Collatz-Folge wird repetitiv, sobald die Zahl 1 erscheint.

## while-Anweisung: Beispiel

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

// Iteration
while (n > 1) { // stop if 1 is reached
 if (n % 2 == 0) // n is even
 n = n / 2;
 else // n is odd
 n = 3 * n + 1;
 std::cout << n << " ";
}
```

## Die Collatz-Folge

$n = 27$ :

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`
  - *condition* bei `for`, `while` erlaubt mehr...

## do-Anweisung

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

- ist äquivalent zu

```
for (bool firsttime = true; firsttime || expression; firsttime = false)
 statement
```

## do-Anweisung: Semantik

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

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

## do-Anweisung: Beispiel

Taschenrechner: addiere Zahlenfolge (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

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

## Sprunganweisungen

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

## break-Anweisung

```
break;
```

- o umschließende Iterationsanweisung wird *sofort* beendet.
- o nützlich, um Schleife "in der Mitte" abbrechen zu können

## break-Anweisung: Beispiel

Taschenrechner: addiere Zahlenfolge (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; // irrelevant in letzter Iteration
 std::cout << "sum = " << s << "\n";
} while (a != 0);
```

## break-Anweisung: Beispiel

Taschenrechner: **unterdrücke irrelevante Addition von 0**

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

## break-Anweisung: Beispiel

Taschenrechner: **äquivalent und noch etwas einfacher:**

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

## break-Anweisung: Beispiel

Taschenrechner: **Version ohne break (wertet a stets zweimal aus und braucht zusätzlichen Block):**

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

## continue-Anweisung

**continue;**

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

## continue-Anweisung: Beispiel

Taschenrechner: **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:

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

**Nicht ganz so einfach!**

Es gilt aber:

- o alle drei Iterationsanweisungen haben die gleiche "Ausdruckskraft" (Skript)

## while kann for simulieren

Gegeben eine `for`-Anweisung

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

finde äquivalente `while`-Anweisung!

Erster Versuch:

```
init-statement
while (condition) {
 statement
 expression;
}
```

geht nicht, falls `statement` ein `continue`; enthält!

## Auswahl der "richtigen" Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- wenige Anweisungen
- wenige Zeilen Code
- einfacher Kontrollfluss
- einfache Ausdrücke

Ziele sind oft nicht gleichzeitig erreichbar.

## Auswahl der "richtigen" Iterationsanweisung: Beispiel

Ausgabe der ungeraden Zahlen in {0,...,100}:

Erster (korrekter) Versuch:

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

## Auswahl der "richtigen" Iterationsanweisung: Beispiel

Ausgabe der ungeraden Zahlen in {0,...,100}:

Weniger Anweisungen, weniger Zeilen:

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

## Auswahl der "richtigen" Iterationsanweisung: Beispiel

Ausgabe der ungeraden Zahlen in {0,...,100}:

Weniger Anweisungen,  
einfacherer Kontrollfluss:

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

Das ist hier die "richtige" Iterationsanweisung !