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:  undeklarierter Name
    return 0;
}
```

Sichtbarkeit

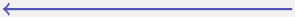
Deklaration in einem Block ist ausserhalb des Blocks nicht „sichtbar”.

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

main block

block

„Blickrichtung”



Kontrollanweisung definiert 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:  undeklarierter Name
    return 0;
}
```

Kontrollanweisung definiert 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:  undeklarierter Name
  return 0;
}
```

block

Potenzieller Gültigkeitsbereich

Im Block

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

Im Funktionsrumpf

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

In Kontrollanweisung

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

Potenzieller Gültigkeitsbereich

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

```
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;
}
```

Potenzieller Gültigkeitsbereich

```
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;
}
```

Wirklicher Gültigkeitsbereich

```
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;
}
```

Lokale Variablen

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```

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

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```

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

`while` Anweisung

```
while ( condition )
  statement
```

`while` Anweisung

```
while ( condition )
  statement
```

ist äquivalent zu

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

`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;
```

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

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

Die Collatz-Folge

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

n=5: 5

Die Collatz-Folge

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

n=5: 5, 16

Die Collatz-Folge

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

n=5: 5, 16, 8

Die Collatz-Folge

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

n=5: 5, 16, 8, 4

Die Collatz-Folge

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

n=5: 5, 16, 8, 4, 2

Die Collatz-Folge

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

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

Die Collatz-Folge

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

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

Die Collatz-Folge

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

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

Die Collatz-Folge

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

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

Die Collatz-Folge

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

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

Die Collatz-Folge in C++

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

// Iteration
while (n > 1)       // stop when 1 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 in C++

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

// Iteration
while (n > 1)       // stop when 1 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 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
```

`do` Anweisung

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

`do` Anweisung

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

ist äquivalent zu

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

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);
```

Sprunganweisungen

- `break;`
- `continue;`

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 `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";
}
```

Kontrollfluss

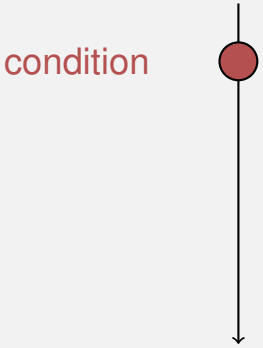
Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...

Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

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

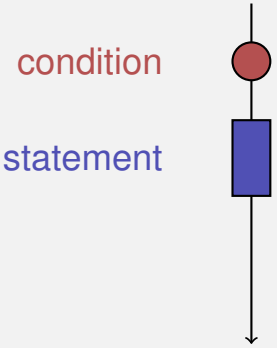


`if` (*condition*)
statement

Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

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

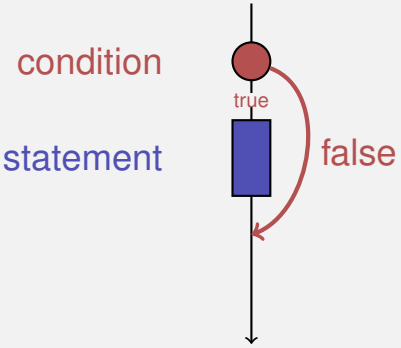


`if` (*condition*)
statement

Kontrollfluss

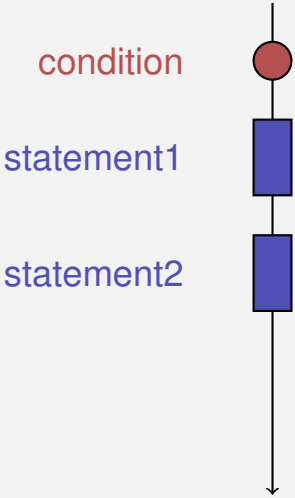
Reihenfolge der (wiederholten) Ausführung von Anweisungen

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



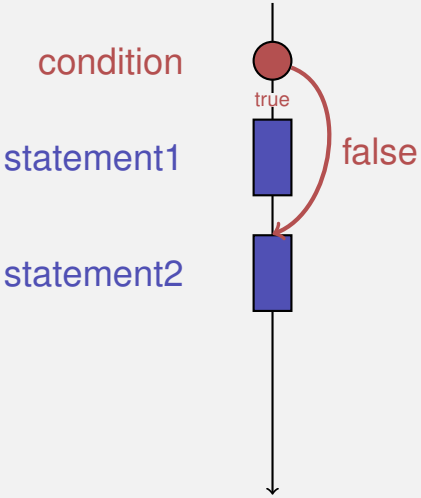
`if` (*condition*)
statement

Kontrollfluss `if else`



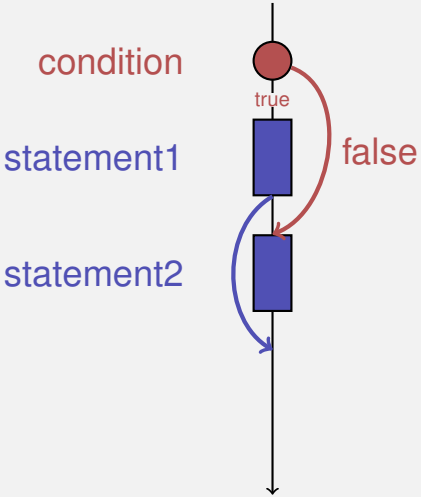

```
if ( condition )
    statement1
else
    statement2
```

Kontrollfluss `if else`




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

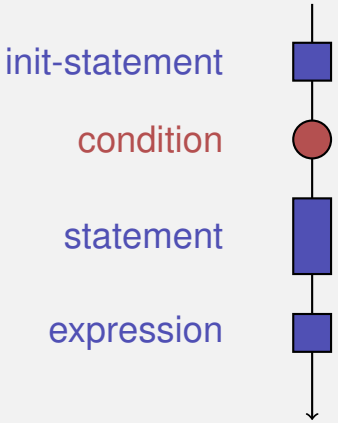
Kontrollfluss `if else`




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

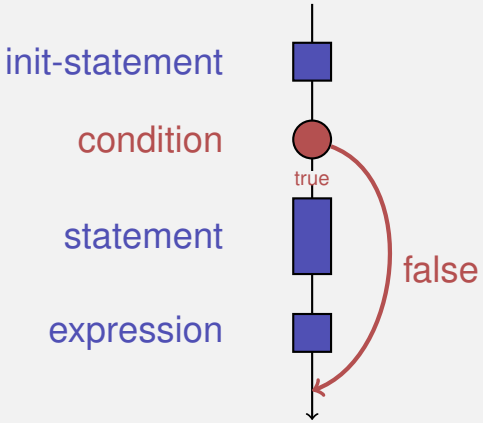
Kontrollfluss `for`

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



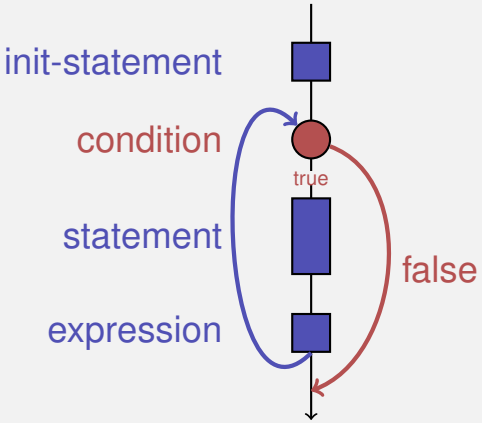
Kontrollfluss `for`

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

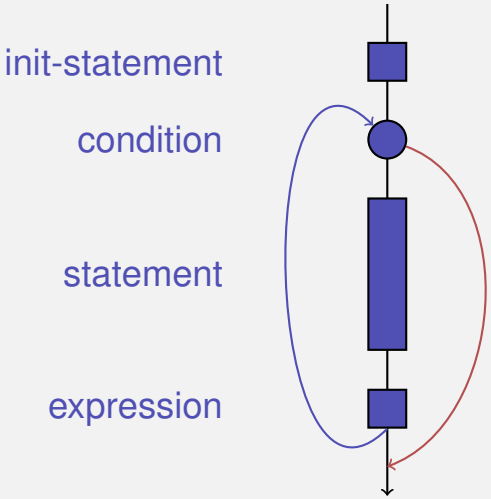


Kontrollfluss `for`

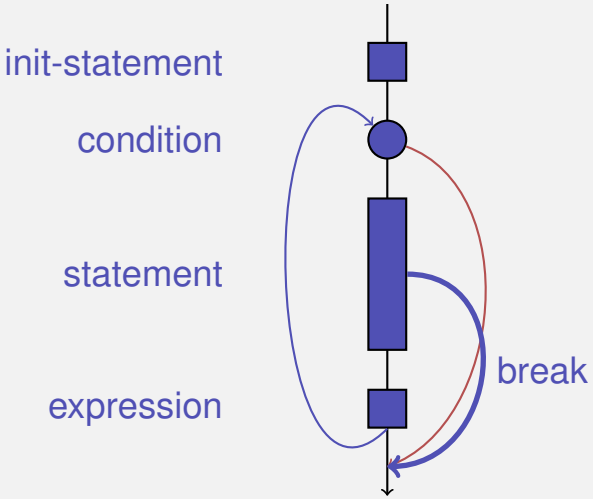
`for` (*init statement* *condition* `;` *expression*)
statement



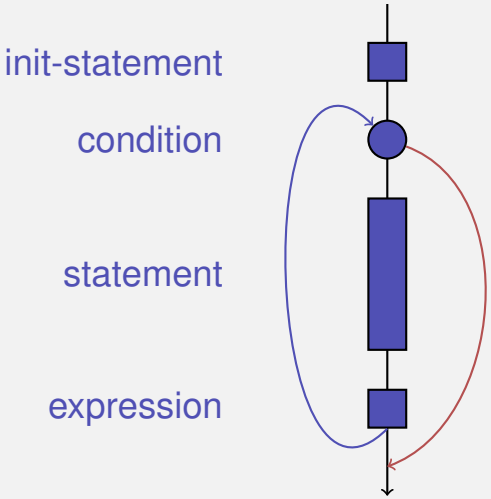
Kontrollfluss `break` und `continue` in for



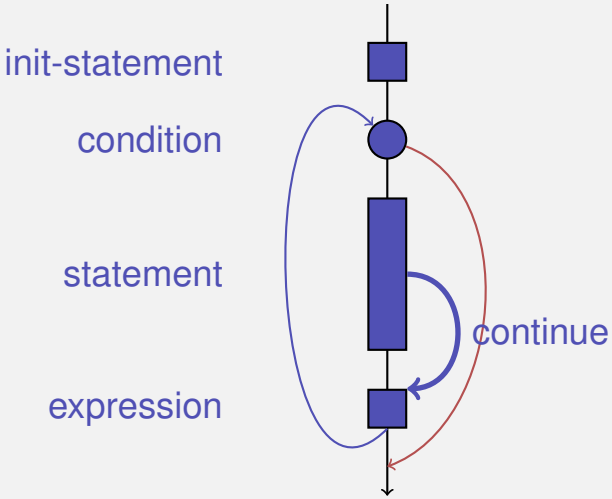
Kontrollfluss `break` und `continue` in for



Kontrollfluss `break` und `continue` in for



Kontrollfluss `break` und `continue` in for



Kontrollfluss: Die guten alten Zeiten?

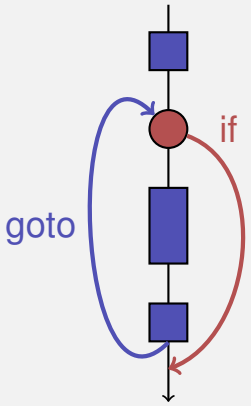
Beobachtung

Wir brauchen eigentlich nur `ifs` und Sprünge an beliebige Stellen im Programm (`goto`).

Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur `ifs` und Sprünge an beliebige Stellen im Programm (`goto`).



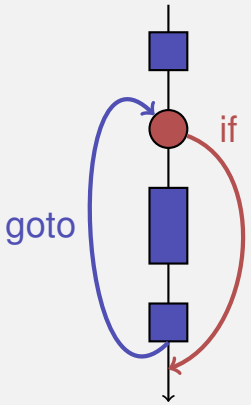
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur `ifs` und Sprünge an beliebige Stellen im Programm (`goto`).

Modelle:

- Maschinensprache



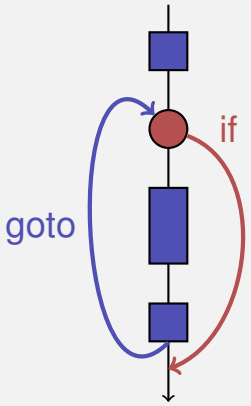
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur `ifs` und Sprünge an beliebige Stellen im Programm (`goto`).

Modelle:

- Maschinensprache
- Assembler (“höhere” Maschinensprache)



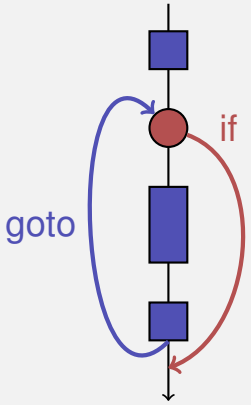
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur `ifs` und Sprünge an beliebige Stellen im Programm (`goto`).

Modelle:

- Maschinensprache
- Assembler (“höhere” Maschinensprache)
- BASIC, die erste Programmiersprache für ein allgemeines Publikum (1964)



BASIC und die Home-Computer...

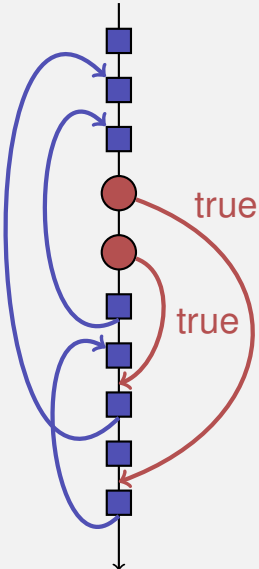
...ermöglichten einer ganzen Generation von Jugendlichen das Programmieren.

Home-Computer Commodore C64 (1982)

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  N=N+1
80  GOTO 20
100 PRINT N
110 GOTO 70
```



Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

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

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 $\{0, \ldots, 100\}$

Erster (korrekter) Versuch:

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

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

Weniger Anweisungen, *weniger* Zeilen:

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

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

Weniger Anweisungen, *einfacherer* Kontrollfluss:

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

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

Weniger Anweisungen, *einfacherer* Kontrollfluss:

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

Das ist hier die “richtige” Iterationsanweisung!

Die `switch`-Anweisung

`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

```
int Note;
...
switch (Note) {
    case 6:
        std::cout << "super!";
        break;
    case 5:
        std::cout << "cool!";
        break;
    case 4:
        std::cout << "ok.";
        break;
    default:
        std::cout << "hmm...";
}
```

Die `switch`-Anweisung

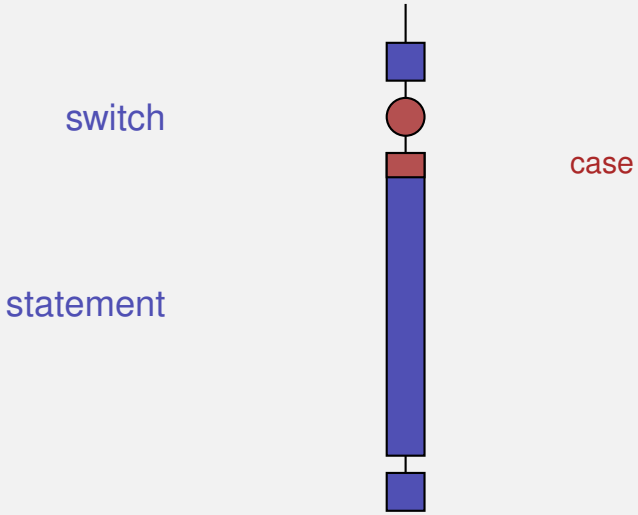
`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

- *condition*: Ausdruck, konvertierbar in einen integralem Typ
- *statement* : beliebige Anweisung, in welcher **`case`** und **`default`**-Marken erlaubt sind, **`break`** hat eine spezielle Bedeutung.

Kontrollfluss `switch`



Kontrollfluss `switch`

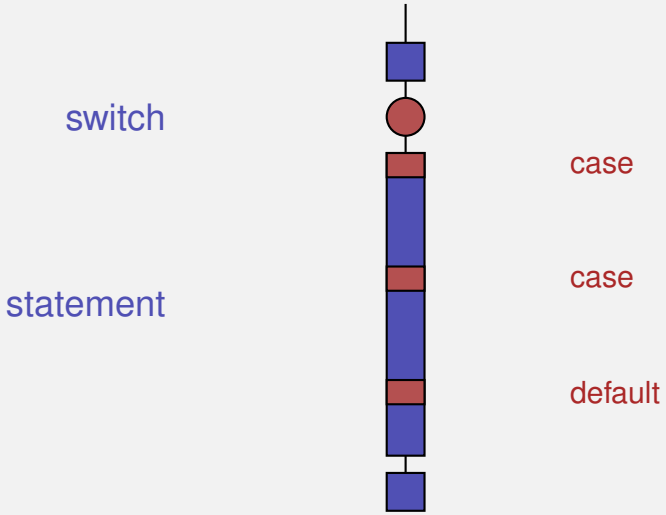
switch

statement

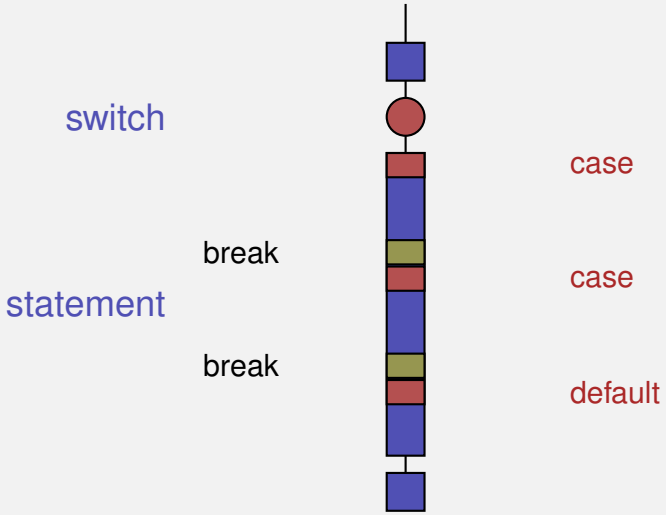
case

case

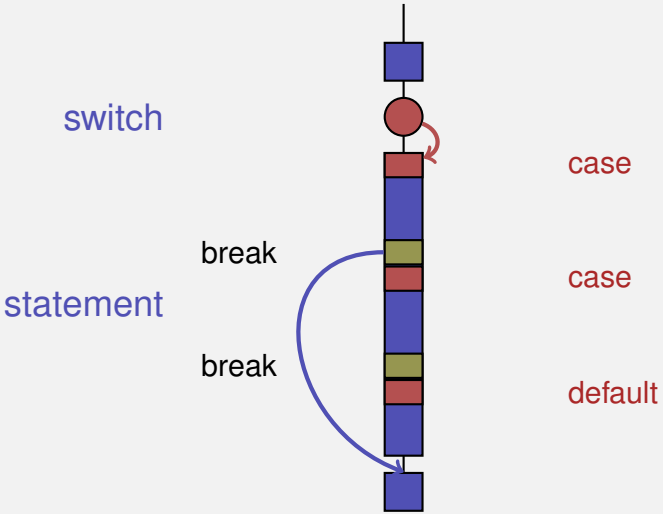
Kontrollfluss `switch`



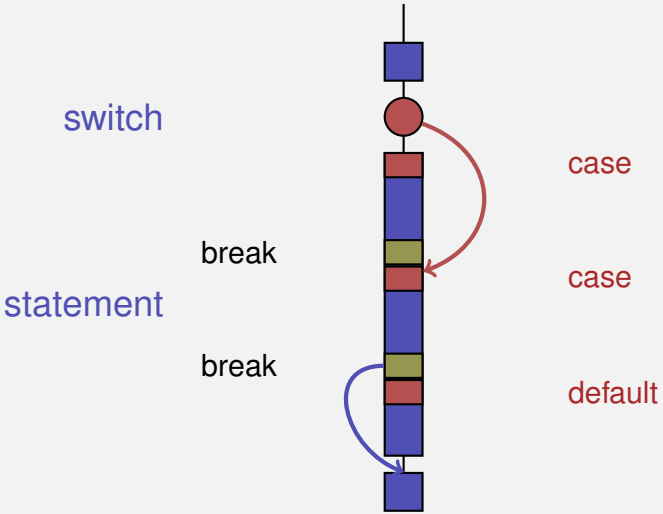
Kontrollfluss `switch`



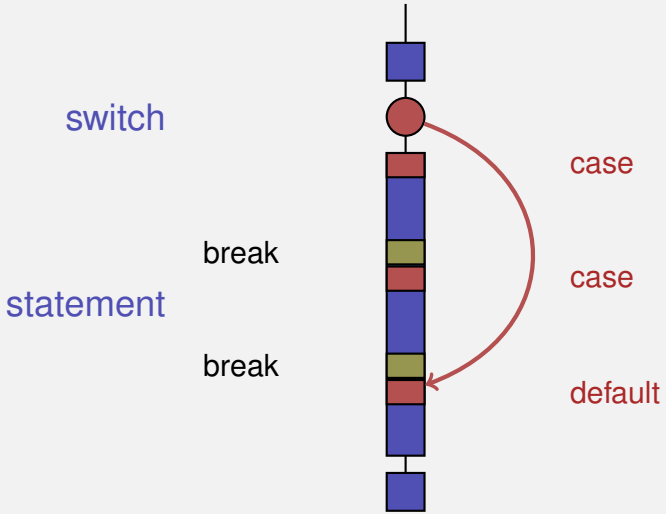
Kontrollfluss `switch`



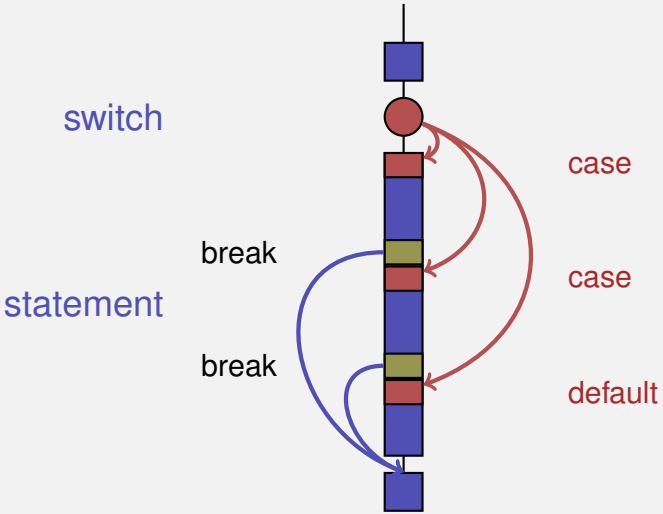
Kontrollfluss `switch`



Kontrollfluss `switch`



Kontrollfluss `switch`



6. Fliesskommazahlen I

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

„Richtig Rechnen”

```
// 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";
```

28 degrees Celsius are 82 degrees Fahrenheit.

„Richtig Rechnen”

```
// 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";
```

28 degrees Celsius are 82 degrees Fahrenheit.

richtig wäre 82.4

„Richtig Rechnen”

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
float celsius; // Fliesskommazahlentyp
std::cin >> celsius;

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

28 degrees Celsius are 82.4 degrees Fahrenheit.

Fixkommazahlen

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

Fixkommazahlen

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

```
82.4 = 0000082.400
```

Fixkommazahlen

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

```
82.4 = 0000082.400
```

Nachteile

- Wertebereich wird *noch* kleiner als bei ganzen Zahlen.

Fixkommazahlen

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

```
0.0824 = 0000000.082
```
← dritte Stelle abgeschnitten

Nachteile

- Repräsentierbarkeit hängt von der Stelle des Kommas ab.

Fliesskommazahlen

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

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

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

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

Fliesskommazahlen

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

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

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

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

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 *Signifikand* $\times 10^{\textit{Exponent}}$

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

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

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

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

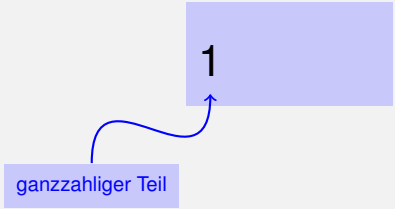
Arithmetische Operatoren

Wie bei `int`, aber ...

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

Literale

unterscheiden sich von Ganzzahlliteralen

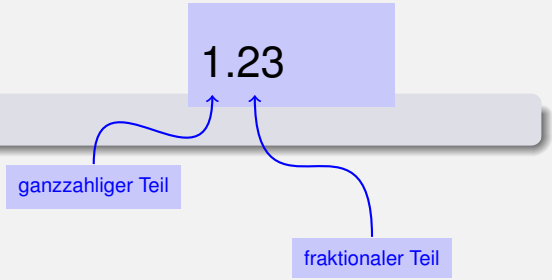


Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1



Literale

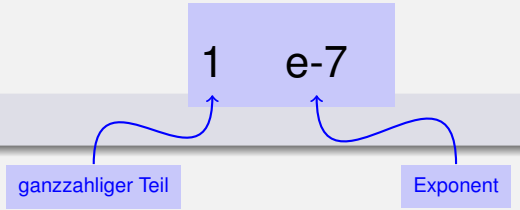
unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

  `1.0` : Typ `double`, Wert 1

- oder Exponent.

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

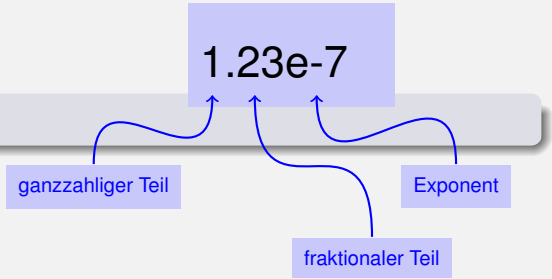


Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1



- und / oder Exponent.

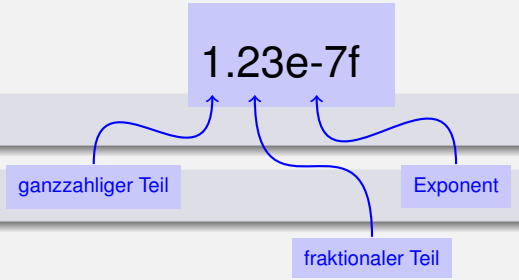
`1e3` : Typ `double`, Wert 1000

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

Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma
- und / oder Exponent.



`1.0` : Typ `double`, Wert 1

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

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

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

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

Rechnen mit `float`: Beispiel

Approximation der Euler-Zahl

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

mittels der ersten 10 Terme.

Rechnen mit `float`: Eulersche Zahl

```
// 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";
}
```

Rechnen mit `float`: Eulersche Zahl

```
// 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";
}
```

Rechnen mit `float`: Eulersche Zahl

```
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.

Gemischte Ausdrücke, Konversion

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

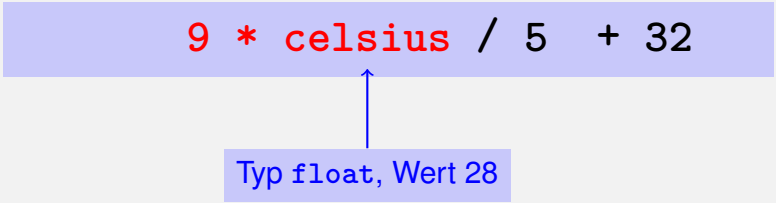
Gemischte Ausdrücke, Konversion

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

```
9 * celsius / 5  + 32
```

Gemischte Ausdrücke, Konversion

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



Gemischte Ausdrücke, Konversion

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

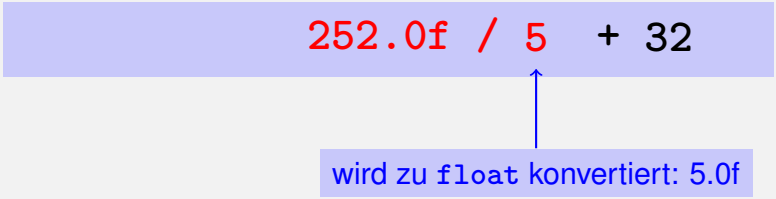
```
9 * 28.0f / 5  + 32
```

Gemischte Ausdrücke, Konversion

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

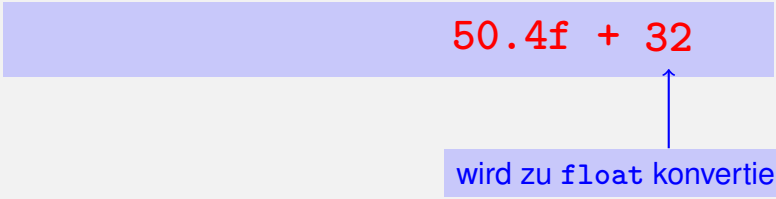
Gemischte Ausdrücke, Konversion

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



Gemischte Ausdrücke, Konversion

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



Gemischte Ausdrücke, Konversion

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

```
                                    82.4f
```

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...

Wertebereich

Ganzzahlige Typen:

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

Wertebereich

Fliesskommatypen:

- Über- und Unterlauf selten, aber ...

Wertebereich

Fliesskommatypen:

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

Löcher im Wertebereich

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

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

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

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

Löcher im Wertebereich

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

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

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

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

Eingabe 1.5

Eingabe 1.0

Eingabe 0.5

Löcher im Wertebereich

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

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

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

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

Eingabe 1.5

Eingabe 1.0

Eingabe 0.5

Ausgabe 0

Löcher im Wertebereich

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

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

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

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

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Löcher im Wertebereich

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

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

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

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

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.23517e-8

Löcher im Wertebereich

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

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

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

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

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.23517e-8

Was ist denn hier los?