

6. 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: undeklariierter Name  
  return 0;  
}
```

main block

block

„Blickrichtung“

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
```

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)

do Anweisung

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

do Anweisung

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

ist äquivalent zu

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

break und continue in der Praxis

- Vorteil: Können verschachtelte `if-else`-Blöcke (oder komplexe Disjunktionen) vermeiden

break und continue in der Praxis

- Vorteil: Können verschachtelte `if-else`-Blöcke (oder komplexe Disjunktionen) vermeiden
- Aber führen zu mehr Sprüngen (vor- und rückwärts) und somit zu potentiell komplexerem Kontrollfluss

break und continue in der Praxis

- Vorteil: Können verschachtelte `if-else`-Blöcke (oder komplexe Disjunktionen) vermeiden
- Aber führen zu mehr Sprüngen (vor- und rückwärts) und somit zu potentiell komplexerem Kontrollfluss
- Ihr Einsatz ist daher umstritten und sollte mit Vorsicht geschehen

Kontrollfluss for

*for (init statement condition ; expression)
statement*

init-statement

condition

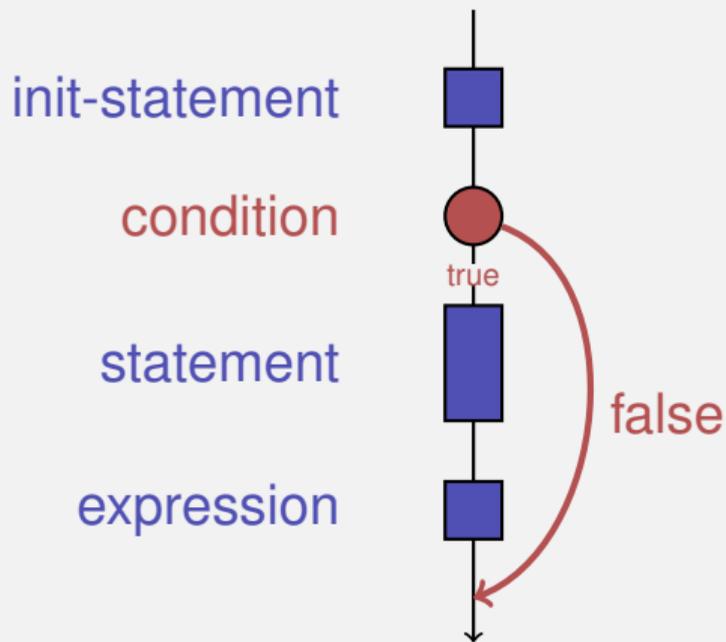
statement

expression



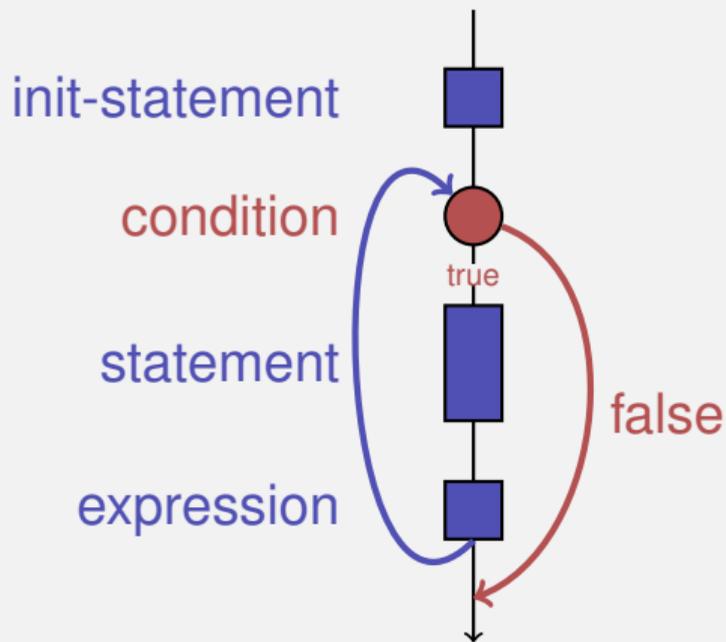
Kontrollfluss for

`for (init statement condition ; expression)
 statement`

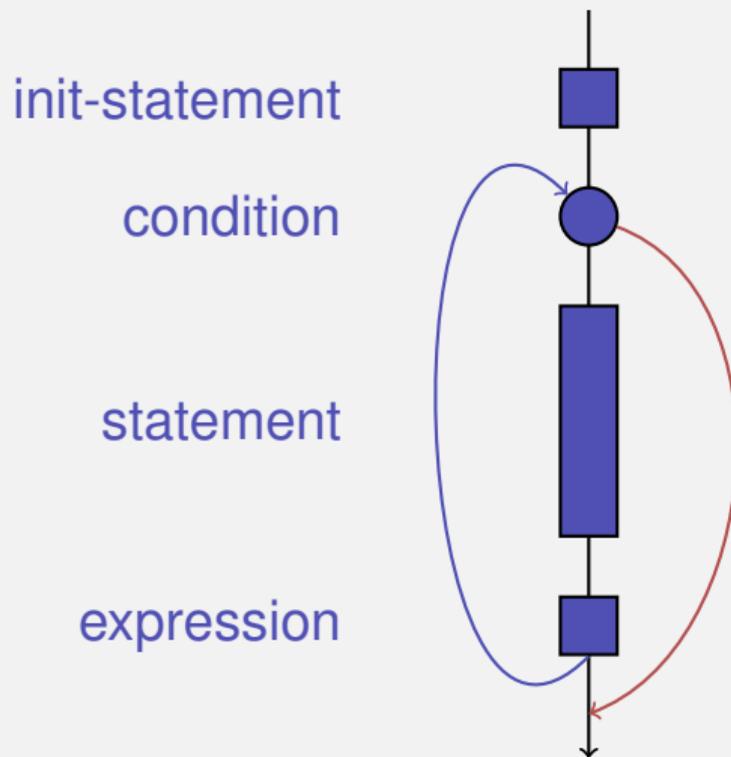


Kontrollfluss for

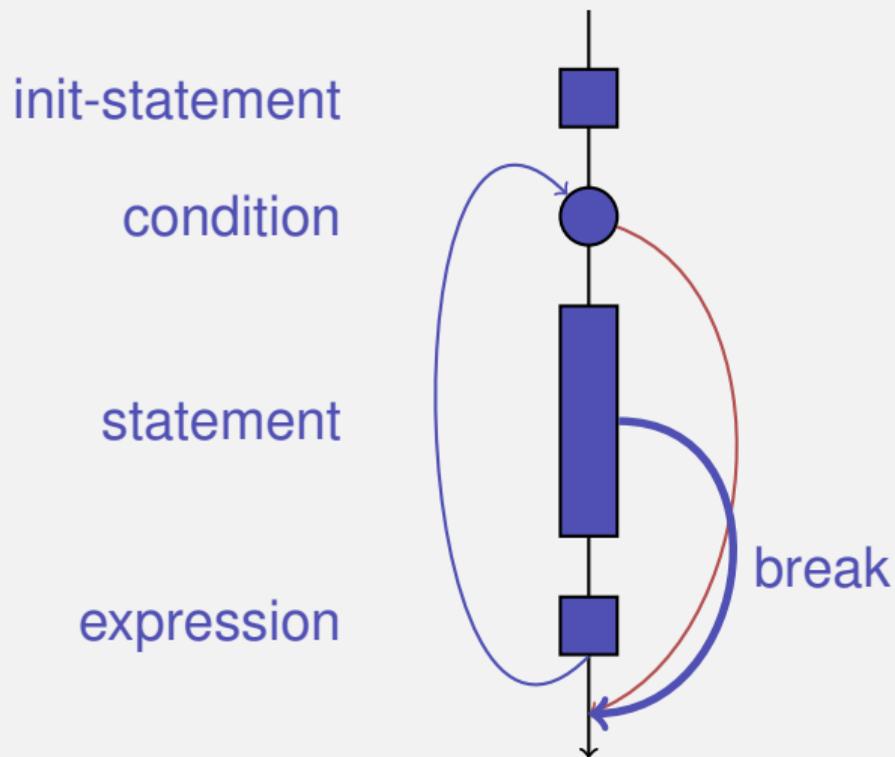
`for (init statement condition ; expression)
 statement`



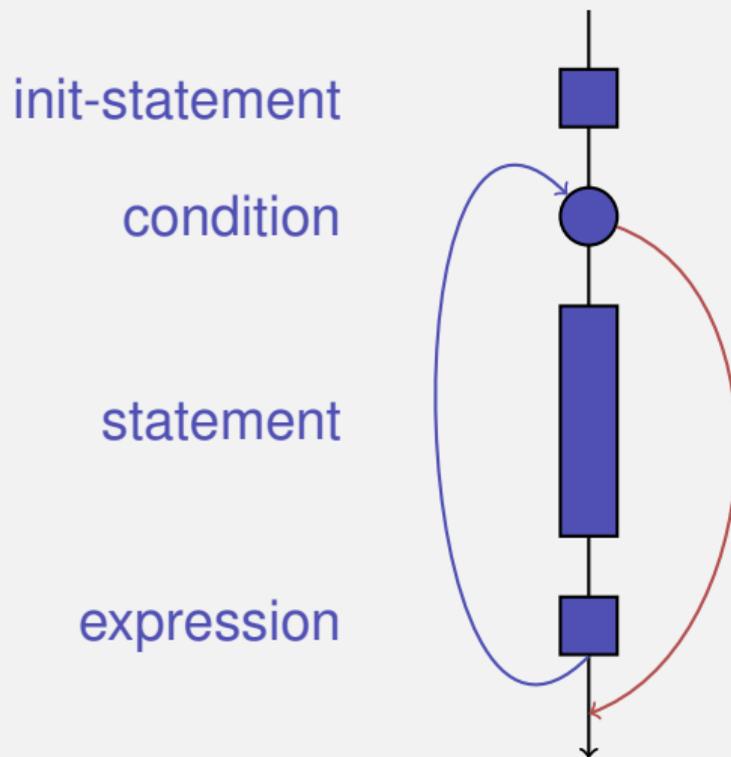
Kontrollfluss break und continue in for



Kontrollfluss `break` und `continue` in `for`



Kontrollfluss break und continue in for



Kontrollfluss: Die guten alten Zeiten?

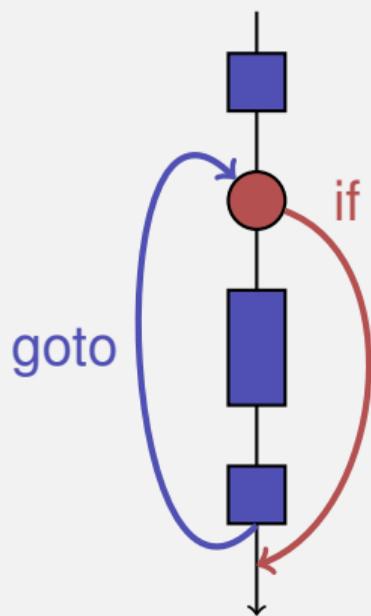
Beobachtung

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`).



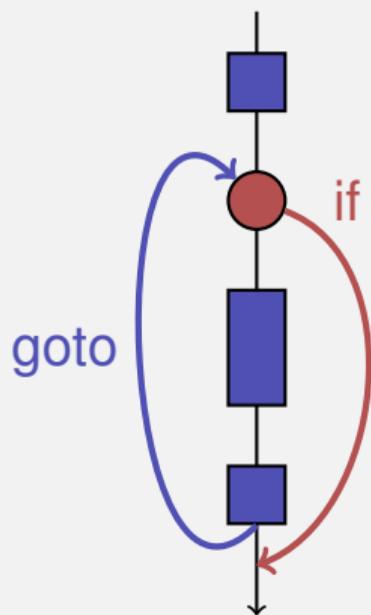
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Sprachen, die darauf basieren:

- Maschinensprache



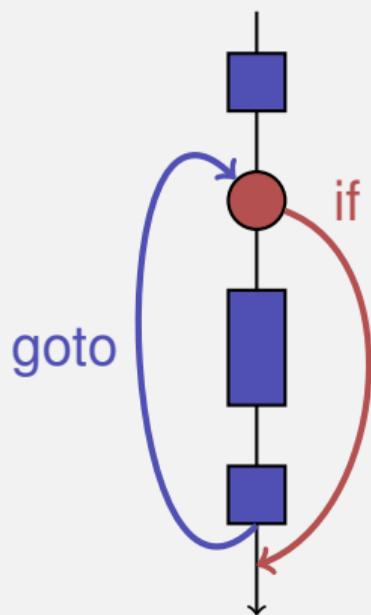
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Sprachen, die darauf basieren:

- Maschinensprache
- Assembler („höhere“ Maschinensprache)



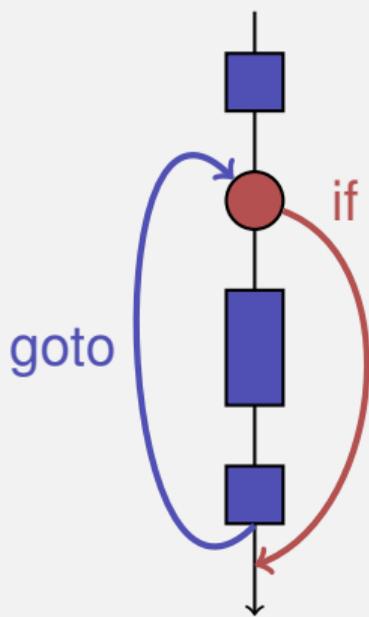
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Sprachen, die darauf basieren:

- 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 von ??????????
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 $\{0, \dots, 100\}$

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";
```

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

Notenausgabe

1. Funktionale Anforderung:

6 → "Excellent ... You passed!"

5,4 → "You passed!"

3 → "Close, but ... You failed!"

2,1 → "You failed!"

sonst → "Error!"

Notenausgabe

1. Funktionale Anforderung:

6 → "Excellent ... You passed!"

5,4 → "You passed!"

3 → "Close, but ... You failed!"

2,1 → "You failed!"

sonst → "Error!"

2. Ausserdem: Text- und Codeduplikation vermeiden

Notenausgabe mit `if`-Anweisungen

```
int grade;
...
if (grade == 6) std::cout << "Excellent ... ";
if (4 <= grade && grade <= 6) {
    std::cout << "You passed!";
} else if (1 <= grade && grade < 4) {
    if (grade == 3) std::cout << "Close, but ... ";
    std::cout << "You failed!";
} else std::cout << "Error!";
```

Notenausgabe mit `if`-Anweisungen

```
int grade;
...
if (grade == 6) std::cout << "Excellent ... ";
if (4 <= grade && grade <= 6) {
    std::cout << "You passed!";
} else if (1 <= grade && grade < 4) {
    if (grade == 3) std::cout << "Close, but ... ";
    std::cout << "You failed!";
} else std::cout << "Error!";
```

Nachteil: Kontrollfluss – und somit Programmverhalten – nicht gerade offensichtlich

Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Notenausgabe mit `switch`-Anweisung

```
switch (grade) { ← Springe zu passendem case
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Durchfallen



Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break; ← Durchfallen  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Verlasse switch

Notenausgabe mit switch-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Durchfallen



Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break; ← Durchfallen  
        break; ← Verlasse switch  
    default: std::cout << "Error!";  
}
```

Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!"; ← In allen anderen Fällen  
}
```

Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Vorteil: Kontrollfluss klar erkennbar

Die `switch`-Anweisung

```
switch (condition)  
    statement
```

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

Die `switch`-Anweisung

```
switch (condition)  
    statement
```

- *condition*: Ausdruck, konvertierbar in einen integralen Typ
- *statement*: beliebige Anweisung, in welcher `case` und `default`-Marken erlaubt sind, `break` hat eine spezielle Bedeutung.
- Benutzung des Durchfallens in der Praxis umstritten, Einsatz gut abwägen (entsprechende Compilerwarnung kann aktiviert werden)

7. 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; // Enable fractional numbers
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

- Beobachtung: Unterschiedlich „effiziente“ Darstellungen einer Zahl, z.B.

$$\begin{aligned} 0.0824 &= 0.00824 \cdot 10^1 &= 0.824 \cdot 10^{-1} \\ &= 8.24 \cdot 10^{-2} &= 824 \cdot 10^{-4} \end{aligned}$$

Anzahl *signifikanter Stellen* bleibt konstant

Fliesskommazahlen

- Beobachtung: Unterschiedlich „effiziente“ Darstellungen einer Zahl, z.B.

$$\begin{aligned}0.0824 &= 0.00824 \cdot 10^1 &= 0.824 \cdot 10^{-1} \\ &= 8.24 \cdot 10^{-2} &= 824 \cdot 10^{-4}\end{aligned}$$

Anzahl *signifikanter Stellen* bleibt konstant

- Fließkommarepräsentation daher:
 - Feste Anzahl signifikanter Stellen (z.B. 10),
 - Plus Position des Kommas mittels Exponenten
 - 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

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:
 - `float`: ca. 7 Stellen, Exponent bis ± 38
 - `double`: ca. 15 Stellen, Exponent bis ± 308

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:
 - `float`: ca. 7 Stellen, Exponent bis ± 38
 - `double`: ca. 15 Stellen, Exponent bis ± 308
- sind auf den meisten Rechnern sehr schnell (Hardwareunterstützung)

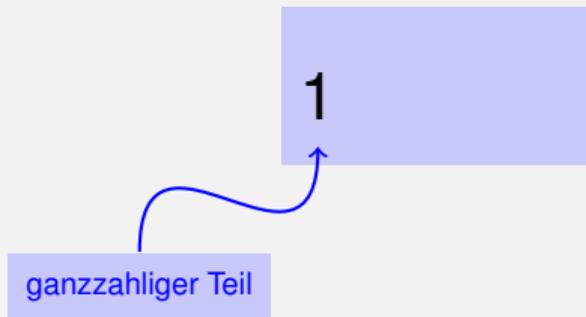
Arithmetische Operatoren

Wie bei `int`, aber ...

- Divisionsoperator `/` modelliert „echte“ (reelle, nicht ganzzahlige) Division
- Kein Modulo-Operator, d.h. kein `%`

Literale

unterscheiden sich von Ganzzahlliteralen



Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

■ Dezimalkomma

1.0 : Typ `double`, Wert 1

1.23

ganzzahliger Teil

fraktionaler Teil

Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1

ganzzahliger Teil

Exponent

- oder Exponent.

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

1 e-7

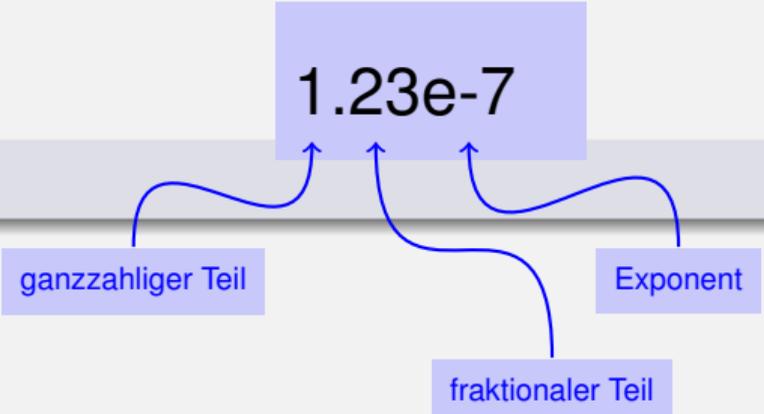
Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1

`1.23e-7`



ganzzahliger Teil

Exponent

fraktionaler Teil

- 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

1.0 : Typ `double`, Wert 1

1.27f : Typ `float`, Wert 1.27

■ 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

ganzzahliger Teil

Exponent

fraktionaler Teil

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

```
std::cout << "Approximating the Euler number... \n";

// values for i-th iteration, initialized for i = 0
float t = 1.0f; // term 1/i!
float e = 1.0f; // i-th approximation of e

// iteration 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.

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.

9 * celsius / 5 + 32

↑
Typ float, Wert 28

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.

9 * 28.0f / 5 + 32

wird zu float konvertiert: 9.0f

Gemischte Ausdrücke, Konversion

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

`252.0f / 5 + 32`

wird zu `float` konvertiert: `5.0f`

Gemischte Ausdrücke, Konversion

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

50.4f + 32

wird zu float konvertiert: 32.0f

Gemischte Ausdrücke, Konversion

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

82.4f

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;
```

Eingabe 1.5

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

Eingabe 1.0

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

Eingabe 0.5

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

Löcher im Wertebereich

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

Eingabe 1.5

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

Eingabe 1.0

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

Eingabe 0.5

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

Ausgabe 0

Löcher im Wertebereich

```
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

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

Löcher im Wertebereich

```
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

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

Ausgabe 2.23517e-8

Löcher im Wertebereich

```
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

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

Ausgabe 2.23517e-8

Was ist denn hier los?

Wertebereich

Ganzzahlige Typen:

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

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“.