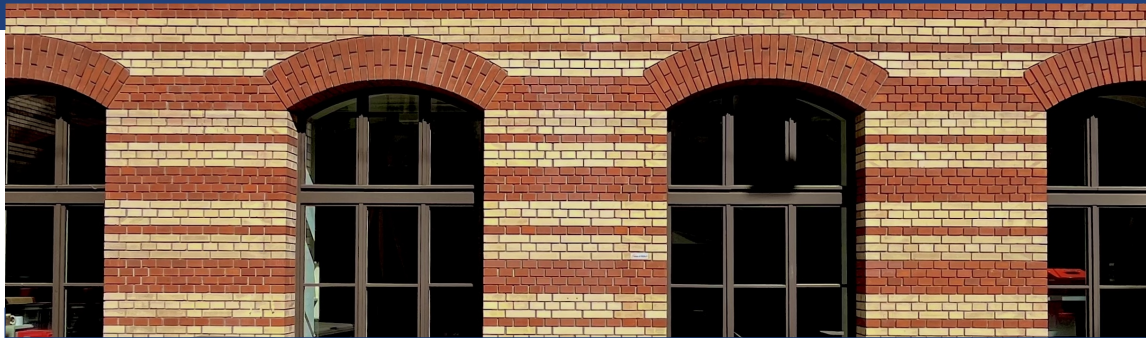




Informatik

for-Schleifen, Debugging, Fließkommazahlen

Adel Gavranović

ETH Zürich — 2026

Überblick

Sichtbarkeitsbereiche (Scopes)

for-Schleifen

Debugging

Repetition: Ausdrücke

Fliesskommazahlen

Leitlinien für Fliesskommazahlen

Fliesskommazahlen vergleichen

Extra: Debugging auf **code expert**

Extra: Auswahlanweisungen



adelgavranovic.ch

 Webpage

 Mail

1. Follow-up

Follow-up aus letzter Übungsstunde

Follow-up aus letzter Übungsstunde

- Quiz auf Seite 29 von letzter Übungsstunde

Follow-up aus letzter Übungsstunde

- Quiz auf Seite 29 von letzter Übungsstunde
- Beispiel (und Probleme) zu `int < unsigned int`

Follow-up aus letzter Übungsstunde

- Quiz auf Seite 29 von letzter Übungsstunde
- Beispiel (und Probleme) zu `int < unsigned int`
- Die Gruppen bei **code expert** sind nun gefixt

Follow-up aus letzter Übungsstunde

- Quiz auf Seite 29 von letzter Übungsstunde
- Beispiel (und Probleme) zu `int < unsigned int`
- Die Gruppen bei **code expert** sind nun gefixt
- Habe ich etwas vergessen?

2. Feedback zu **code** expert

Allgemeines bezüglich **code expert**

Allgemeines bezüglich **code expert**

- Läuft alles?

Allgemeines bezüglich **code expert**

- Läuft alles?
- **code expert** auch gut?

Allgemeines bezüglich **code expert**

- Läuft alles?
- **code expert** auch gut?
- Fragen zu den Übungsaufgaben der letzten Woche?

Fragen bezüglich **code expert** eurerseits?

3. Lernziele

Ziele

Ziele

- ☐ Programme mit `for`-Schleifen nachvollziehen (*tracen*) und debuggen können
- ☐ Einfache Programme mit `for`-Schleifen schreiben können
- ☐ Einige Leitlinien für den Umgang mit Fließkommazahlen kennen

4. Zusammenfassung

Getting on the same page

Getting on the same page

- Was wurde diese Woche behandelt?

5. Sichtbarkeitsbereiche (Scopes)

Kurzeinführung zu Scopes

Kurzeinführung zu Scopes

- Praktisch immer, wenn ihr {geschweifte Klammern} verwendet, erstellt ihr einen *Scope*

Kurzeinführung zu Scopes

- Praktisch immer, wenn ihr {geschweifte Klammern} verwendet, erstellt ihr einen *Scope*
- Stellt euch einen Scope als “Welt in einer Welt” vor

Kurzeinführung zu Scopes

- Praktisch immer, wenn ihr {geschweifte Klammern} verwendet, erstellt ihr einen *Scope*
- Stellt euch einen Scope als “Welt in einer Welt” vor
- Variablen von aussen sind im inneren Scope sichtbar, aber Variablen aus dem inneren Scope “fliessen nicht hinaus”

Kurzeinführung zu Scopes

- Praktisch immer, wenn ihr {geschweifte Klammern} verwendet, erstellt ihr einen *Scope*
- Stellt euch einen Scope als “Welt in einer Welt” vor
- Variablen von aussen sind im inneren Scope sichtbar, aber Variablen aus dem inneren Scope “fliessen nicht hinaus”
- Sobald das Programm bei der schliessenden Klammer } ankommt, “sterben” alle Variablen, die in diesem Scope deklariert wurden

Mehr zu Scopes

Scopes werden im C++-Leitfaden auf der Kurswebseite erklärt. Dort findet ihr auch einen C++-Style-Guide, der euch hilft, sauberen Code zu schreiben.

 C++-Leitfaden

Fragen dazu könnt ihr mir natürlich auch direkt stellen.

Fragen/Unklarheiten?

6. for-Schleifen

Struktur einer for-Schleife

```
for (init; condition; expression) {  
    statement1;  
    statement2;  
    // ...  
}
```

Struktur einer for-Schleife

```
for (init; condition; expression) {  
    statement1;  
    statement2;  
    // ...  
}
```

Ablauf

1. init wird genau einmal ausgeführt

Struktur einer for-Schleife

```
for (init; condition; expression) {  
    statement1;  
    statement2;  
    // ...  
}
```

Ablauf

1. `init` wird genau einmal ausgeführt
2. `condition` wird ausgewertet: ist sie `false`, ist die Schleife fertig

Struktur einer for-Schleife

```
for (init; condition; expression) {  
    statement1;  
    statement2;  
    // ...  
}
```

Ablauf

1. `init` wird genau einmal ausgeführt
2. `condition` wird ausgewertet: ist sie `false`, ist die Schleife fertig
3. Die Statements im Rumpf werden ausgeführt

Struktur einer for-Schleife

```
for (init; condition; expression) {  
    statement1;  
    statement2;  
    // ...  
}
```

Ablauf

1. `init` wird genau einmal ausgeführt
2. `condition` wird ausgewertet: ist sie `false`, ist die Schleife fertig
3. Die Statements im Rumpf werden ausgeführt
4. `expression` wird ausgeführt – also *nach* den Statements! – und es geht weiter bei 2.

Beispiel

```
int sum = 0;
for (int i = 1; i <= 3; ++i) {
    sum += i;
}
std::cout << sum << "\n";
```

Was gibt dieses Programm aus? Wir verfolgen es Schritt für Schritt, mit der Notation aus dem Program Tracing Guide.

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

0

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

0

i:

1

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

0

i:

1

1 <= 3 → **true**

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

1

i:

1

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

1

i:

2

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

1

i:

2

$2 \leq 3 \rightarrow \text{true}$

Beispiel for-Schleife

```
int sum = 0;
for (int i = 1; i <= 3; ++i) {
    sum += i;
}
std::cout << sum << "\n";
```

sum:

3

i:

2

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

3

i:

3

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

3

i:

3

$3 \leq 3 \rightarrow \text{true}$

Beispiel for-Schleife

```
int sum = 0;
for (int i = 1; i <= 3; ++i) {
    sum += i;
}
std::cout << sum << "\n";
```

sum:

6

i:

3

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

6

i:

4

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

6

i:

4

4 <= 3 → **false**

Beispiel for-Schleife

```
int sum = 0;  
for (int i = 1; i <= 3; ++i) {  
    sum += i;  
}  
std::cout << sum << "\n";
```

sum:

6

i existiert nicht mehr: sein Scope war die Schleife

Beispiel for-Schleife

```
int sum = 0;
for (int i = 1; i <= 3; ++i) {
    sum += i;
}
std::cout << sum << "\n";
```

sum:

6

Ausgabe: 6

Fragen/Unklarheiten?

Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 3: Exercise Session”, öffnet “Strange Sum”

Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 3: Exercise Session”, öffnet “Strange Sum”
- Überlegt euch alleine, wie ihr die Aufgabe mit Stift und Papier lösen würdet (5 Minuten)

Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 3: Exercise Session”, öffnet “Strange Sum”
- Überlegt euch alleine, wie ihr die Aufgabe mit Stift und Papier lösen würdet (5 Minuten)
- Bildet Zweiergruppen mit euren Sitznachbar:innen, diskutiert eure Ideen und schreibt zusammen das Programm (ca. 7 Minuten)

Aufgabe

Aufgabe “Strange Sum”

Schreibt ein Programm, das eine Zahl $n > 0$ einliest und die Summe aller positiven Zahlen bis n ausgibt, die ungerade, aber nicht durch 5 teilbar sind.

Zum Beispiel soll für $n = 10$ die Ausgabe 20 sein, denn $1 + 3 + 7 + 9 = 20$.

Ein- und Ausgabe sind in der Vorlage auf **code expert** schon vorhanden: ihr müsst nur noch die Berechnung schreiben.

Mögliche Lösung zu *Strange Sum*

Mögliche Lösung zu *Strange Sum*

```
// input
int strangesum = 0;
int n;
std::cin >> n;
// computation
for (int i = 1; i <= n; i++) {
    if (i % 2 == 1) {
        if (i % 5 != 0) {
            strangesum += i;
        }
    }
}
// output
std::cout << strangesum << "\n";
```

Kompaktere Lösung zu *Strange Sum*

Die beiden **if**-Anweisungen lassen sich zu einer zusammenfassen:

```
for (int i = 1; i <= n; i++) {  
    if (i % 2 == 1 && i % 5 != 0) {  
        strangesum += i;  
    }  
}
```

Noch kompaktere Lösung zu *Strange Sum*

Wenn wir nur die ungeraden Zahlen durchlaufen, fällt die erste Bedingung ganz weg:

```
for (int i = 1; i <= n; i += 2) {  
    if (i % 5 != 0) {  
        strangesum += i;  
    }  
}
```

Noch kompaktere Lösung zu *Strange Sum*

Wenn wir nur die ungeraden Zahlen durchlaufen, fällt die erste Bedingung ganz weg:

```
for (int i = 1; i <= n; i += 2) {  
    if (i % 5 != 0) {  
        strangesum += i;  
    }  
}
```

💡 Die *expression* muss nicht immer *i++* sein!

Fragen/Unklarheiten?

Aufgabe

Aufgabe “Largest Power”

Schreibt ein Programm, das eine positive natürliche Zahl n einliest und die grösste Zweierpotenz p (also $1, 2, 4, 8, \dots$) ausgibt, die kleiner oder gleich n ist.

Zum Beispiel soll für $n = 20$ die Ausgabe 16 sein.

Aufgabe

Aufgabe “Largest Power”

Schreibt ein Programm, das eine positive natürliche Zahl n einliest und die grösste Zweierpotenz p (also $1, 2, 4, 8, \dots$) ausgibt, die kleiner oder gleich n ist.

Zum Beispiel soll für $n = 20$ die Ausgabe 16 sein.

Gleiches Vorgehen wie vorhin: Öffnet “Largest Power” auf **code expert**, überlegt 5 Minuten alleine und schreibt dann ca. 7 Minuten zu zweit das Programm.

Mögliche Lösung zu *Largest Power*

Mögliche Lösung zu *Largest Power*

```
#include <iostream>

int main() {
    int n;
    std::cin >> n;

    int power = 1;
    for (; power <= n / 2; power *= 2);

    std::cout << power << std::endl;

    return 0;
}
```

Genauer hingeschaut

```
int power = 1;  
for (; power <= n / 2; power *= 2);
```

Genauer hingeschaut

```
int power = 1;  
for (; power <= n / 2; power *= 2);
```

- Warum ist `init` leer?

Genauer hingeschaut

```
int power = 1;  
for (; power <= n / 2; power *= 2);
```

- Warum ist `init` leer?

`power` wird *nach* der Schleife noch gebraucht, muss also vor der Schleife deklariert werden (Scope!)

Genauer hingeschaut

```
int power = 1;  
for (; power <= n / 2; power *= 2);
```

- Warum ist `init` leer?

`power` wird *nach* der Schleife noch gebraucht, muss also vor der Schleife deklariert werden (Scope!)

- Warum folgt nach der Klammer nur ein `;`?

Genauer hingeschaut

```
int power = 1;  
for (; power <= n / 2; power *= 2);
```

- Warum ist `init` leer?

`power` wird *nach* der Schleife noch gebraucht, muss also vor der Schleife deklariert werden (Scope!)

- Warum folgt nach der Klammer nur ein `;`?

Der Rumpf ist leer, die ganze Arbeit passiert in der `expression`. Leicht zu übersehen!

Genauer hingeschaut

```
int power = 1;  
for (; power <= n / 2; power *= 2);
```

- Warum ist `init` leer?

`power` wird *nach* der Schleife noch gebraucht, muss also vor der Schleife deklariert werden (Scope!)

- Warum folgt nach der Klammer nur ein `;`?

Der Rumpf ist leer, die ganze Arbeit passiert in der `expression`. Leicht zu übersehen!

- Warum `power <= n / 2` statt `power * 2 <= n`?

Genauer hingeschaut

```
int power = 1;  
for (; power <= n / 2; power *= 2);
```

- Warum ist `init` leer?

`power` wird *nach* der Schleife noch gebraucht, muss also vor der Schleife deklariert werden (Scope!)

- Warum folgt nach der Klammer nur ein `;`?

Der Rumpf ist leer, die ganze Arbeit passiert in der `expression`. Leicht zu übersehen!

- Warum `power <= n / 2` statt `power * 2 <= n`?

Für ganze Zahlen gleichwertig, aber `power * 2` kann für grosse `n` *überlaufen*. Mehr dazu gleich!

Fragen/Unklarheiten?

7. Debugging

Debugging

Debugging

...ist der Prozess, Fehler ("Bugs") in Programmen zu finden und zu beheben.

Debugging

Debugging

...ist der Prozess, Fehler (“Bugs”) in Programmen zu finden und zu beheben.

Wir schauen uns das an einem Programm an, das 6^{12} Sterne ausgeben soll.
Wir führen es lokal aus und *nicht* auf **code expert**: Endlosschleifen dort bremsen den Server für alle.

Debugging – Live Demo

```
int main() {  
    const int n = 6;  
    // Compute n^12  
    int prod = 1;  
    for (int i = 1; 1 <= i < 13; ++i) {  
        prod *= n;  
    }  
    // Output stars  
    for (int i = 1; i < prod; ++i) {  
        std::cout << "*";  
    }  
    std::cout << "\n";  
    return 0;  
}
```

Debugging – Live Demo

Frage

Das Programm terminiert nicht. Wie finden wir heraus, wo es feststeckt?

Debugging – Live Demo

Frage

Das Programm terminiert nicht. Wie finden wir heraus, wo es feststeckt?

Antwort

An interessanten Stellen im Code etwas ausgeben lassen: vor der ersten Schleife, zwischen den Schleifen und nach der zweiten Schleife, z.B.

```
std::cout << "Reached 1" << std::endl;
```

Debugging – Live Demo

Frage

Das Programm terminiert nicht. Wie finden wir heraus, wo es feststeckt?

Antwort

An interessanten Stellen im Code etwas ausgeben lassen: vor der ersten Schleife, zwischen den Schleifen und nach der zweiten Schleife, z.B.

```
std::cout << "Reached 1" << std::endl;
```

Nur "Reached 1" wird ausgegeben $\Rightarrow$ die erste Schleife terminiert nicht.

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Antwort

Die `condition` ist falsch! Werten wir sie aus:

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Antwort

Die `condition` ist falsch! Werten wir sie aus:

```
1 <= i < 13
```

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Antwort

Die `condition` ist falsch! Werten wir sie aus:

```
1 <= i < 13
```

```
(1 <= i) < 13
```

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Antwort

Die `condition` ist falsch! Werten wir sie aus:

```
1 <= i < 13
```

```
(1 <= i) < 13
```

```
true < 13    (oder false < 13)
```

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Antwort

Die `condition` ist falsch! Werten wir sie aus:

```
1 <= i < 13
```

```
(1 <= i) < 13
```

```
true < 13    (oder false < 13)
```

```
1 < 13    (oder 0 < 13)
```

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Antwort

Die `condition` ist falsch! Werten wir sie aus:

```
1 <= i < 13
```

```
(1 <= i) < 13
```

```
true < 13    (oder false < 13)
```

```
1 < 13    (oder 0 < 13)
```

```
true    immer, egal welchen Wert i hat!
```

Debugging – Live Demo

Frage

Warum terminiert die erste Schleife nicht?

Antwort

Die `condition` ist falsch! Werten wir sie aus:

```
1 <= i < 13
```

```
(1 <= i) < 13
```

```
true < 13   (oder false < 13)
```

```
1 < 13   (oder 0 < 13)
```

```
true   immer, egal welchen Wert i hat!
```

Korrekt wäre `1 <= i && i < 13`.

Debugging – Live Demo

Frage

Jetzt terminiert das Programm, gibt aber trotzdem keine Sterne aus. Wie finden wir den Grund?

Debugging – Live Demo

Frage

Jetzt terminiert das Programm, gibt aber trotzdem keine Sterne aus. Wie finden wir den Grund?

Antwort

Den Wert von `prod` nach der ersten Schleife ausgeben lassen: eine grosse *negative* Zahl!

Debugging – Live Demo

Frage

Jetzt terminiert das Programm, gibt aber trotzdem keine Sterne aus. Wie finden wir den Grund?

Antwort

Den Wert von `prod` nach der ersten Schleife ausgeben lassen: eine grosse *negative* Zahl!

Frage

Wie finden wir heraus, warum `prod` negativ wird?

Debugging – Live Demo

Frage

Jetzt terminiert das Programm, gibt aber trotzdem keine Sterne aus. Wie finden wir den Grund?

Antwort

Den Wert von `prod` nach der ersten Schleife ausgeben lassen: eine grosse *negative* Zahl!

Frage

Wie finden wir heraus, warum `prod` negativ wird?

Antwort

Den Wert von `prod` in *jeder* Iteration ausgeben lassen: Potenzen von 6, bis es plötzlich “kippt”.

Was ist passiert? Overflow!

Was ist passiert? Overflow!

- Ein `int` hat 32 Bits, also 2^{32} mögliche Bitmuster. Damit kann er die Werte von -2^{31} bis $2^{31} - 1$ darstellen.

Was ist passiert? Overflow!

- Ein `int` hat 32 Bits, also 2^{32} mögliche Bitmuster. Damit kann er die Werte von -2^{31} bis $2^{31} - 1$ darstellen.

$$6^{11} = 362\,797\,056 \quad \text{passt}$$

$$2^{31} - 1 = 2\,147\,483\,647 \quad (\text{grösster } \text{int})$$

$$6^{12} = 2\,176\,782\,336 \quad \text{zu gross!}$$

Was ist passiert? Overflow!

- Ein `int` hat 32 Bits, also 2^{32} mögliche Bitmuster. Damit kann er die Werte von -2^{31} bis $2^{31} - 1$ darstellen.

$$6^{11} = 362\,797\,056 \quad \text{passt}$$

$$2^{31} - 1 = 2\,147\,483\,647 \quad (\text{grösster } \text{int})$$

$$6^{12} = 2\,176\,782\,336 \quad \text{zu gross!}$$

- Ein Overflow bei `signed int` ist *undefiniertes Verhalten*. Hier wird `prod` negativ, also läuft die zweite Schleife kein einziges Mal.

Was ist passiert? Overflow!

- Ein `int` hat 32 Bits, also 2^{32} mögliche Bitmuster. Damit kann er die Werte von -2^{31} bis $2^{31} - 1$ darstellen.

$$6^{11} = 362\,797\,056 \quad \text{passt}$$

$$2^{31} - 1 = 2\,147\,483\,647 \quad (\text{grösster } \text{int})$$

$$6^{12} = 2\,176\,782\,336 \quad \text{zu gross!}$$

- Ein Overflow bei `signed int` ist *undefiniertes Verhalten*. Hier wird `prod` negativ, also läuft die zweite Schleife kein einziges Mal.

💡 Timeout auf `code expert`? Meistens ist es eine Endlosschleife: prüft eure Bedingungen!

Fragen/Unklarheiten?

8. Repetition: Ausdrücke

Übung

Wertet die folgenden Ausdrücke von Hand aus und notiert alle Zwischenschritte:

a) $5 < 4 < 1$

b) `true > false`

a)

a)

$$5 < 4 < 1$$

a)

$$5 < 4 < 1$$

$$(5 < 4) < 1$$

a)

5 < 4 < 1

(5 < 4) < 1

false < 1

a)

5 < 4 < 1

(5 < 4) < 1

false < 1

0 < 1

a)

```
5 < 4 < 1  
(5 < 4) < 1  
false < 1  
0 < 1  
true
```

a)

```
5 < 4 < 1  
(5 < 4) < 1  
false < 1  
0 < 1  
true
```

b)

```
true > false
```

a)

```
5 < 4 < 1  
(5 < 4) < 1  
false < 1  
0 < 1  
true
```

b)

```
true > false  
true > 0
```

a)

```
5 < 4 < 1  
(5 < 4) < 1  
false < 1  
0 < 1  
true
```

b)

```
true > false  
true > 0  
1 > 0
```

a)

```
5 < 4 < 1  
(5 < 4) < 1  
false < 1  
0 < 1  
true
```

b)

```
true > false  
true > 0  
1 > 0  
true
```

a)

```
5 < 4 < 1
(5 < 4) < 1
false < 1
0 < 1
true
```

b)

```
true > false
true > 0
1 > 0
true
```

💡 Genau wie beim Debugging: $a < b < c$ bedeutet in C++ *nicht* dasselbe wie in der Mathematik. Schreibt stattdessen $a < b \ \&\& \ b < c$.

Fragen/Unklarheiten?

9. Fließkommazahlen

9. Fließkommazahlen

9.1. Leitlinien für Fließkommazahlen

Leitlinie 1

Leitlinie 1

“Vermeide es, zwei Fließkommazahlen auf Gleichheit zu prüfen, nachdem unterschiedliche Berechnungen mit ihnen durchgeführt wurden.”

Leitlinie 1

“Vermeide es, zwei Fließkommazahlen auf Gleichheit zu prüfen, nachdem unterschiedliche Berechnungen mit ihnen durchgeführt wurden.”

Beispiel

```
float a = 2.0f;  
if (std::sqrt(2.0f) * std::sqrt(2.0f) == a) {  
    std::cout << "no output\n";  
}
```

Leitlinie 1

“Vermeide es, zwei Fließkommazahlen auf Gleichheit zu prüfen, nachdem unterschiedliche Berechnungen mit ihnen durchgeführt wurden.”

Beispiel

```
float a = 2.0f;  
if (std::sqrt(2.0f) * std::sqrt(2.0f) == a) {  
    std::cout << "no output\n";  
}
```

Die Bedingung ist `false`! Das Problem: `std::sqrt(2.0f)` ist *nicht* exakt darstellbar.

Leitlinie 2

Leitlinie 2

“Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!”

Leitlinie 2

“Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!”

Beispiel

```
float a = 67108864.0f + 0.1f;  
if (a > 67108864.0f) {  
    std::cout << "This does not output ...\n";  
}
```

Leitlinie 2

“Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!”

Beispiel

```
float a = 67108864.0f + 0.1f;  
if (a > 67108864.0f) {  
    std::cout << "This does not output ...\n";  
}
```

Die Bedingung ist `false`! Das Problem: der Signifikand ist zu kurz.

Leitlinie 2 – Rundung

Leitlinie 2 – Rundung

$$\begin{array}{r} 67108864 = 6.7108864 \cdot 10^7 \\ + \quad 0.1 = 0.00000001 \cdot 10^7 \\ \hline 67108864.1 = 6.71088641 \cdot 10^7 \end{array}$$

Leitlinie 2 – Rundung

$$\begin{array}{r} 67108864 = 6.7108864 \cdot 10^7 \\ + \quad 0.1 = 0.00000001 \cdot 10^7 \\ \hline 67108864.1 = 6.71088641 \cdot 10^7 \\ \approx 67108864 = 6.7108864 \cdot 10^7 \end{array}$$

Leitlinie 2 – Rundung

$$\begin{array}{r} 67108864 = 6.7108864 \cdot 10^7 \\ + \quad 0.1 = 0.00000001 \cdot 10^7 \\ \hline 67108864.1 = 6.71088641 \cdot 10^7 \\ \approx 67108864 = 6.7108864 \cdot 10^7 \end{array} \quad (\text{Rundung!})$$

Leitlinie 2 – Rundung

$$\begin{array}{r} 67108864 = 6.7108864 \cdot 10^7 \\ + \quad 0.1 = 0.00000001 \cdot 10^7 \\ \hline 67108864.1 = 6.71088641 \cdot 10^7 \\ \approx 67108864 = 6.7108864 \cdot 10^7 \end{array} \quad (\text{Rundung!})$$

Für die hinterste Stelle ist im Signifikanden kein Platz mehr, also geht die 0.1 beim Runden verloren.

9. Fliesskommazahlen

9.2. Fliesskommazahlen vergleichen

Das Vergleichsproblem

Das Vergleichsproblem

- Gegeben: fp1 und fp2 vom Typ `float` oder `double`

Das Vergleichsproblem

- Gegeben: `fp1` und `fp2` vom Typ `float` oder `double`
- Leitlinie: “Teste zwei Fließkommazahlen nicht auf Gleichheit, wenn mindestens eine davon vorher gerundet wurde.”

Das Vergleichsproblem

- Gegeben: `fp1` und `fp2` vom Typ `float` oder `double`
- Leitlinie: “Teste zwei Fließkommazahlen nicht auf Gleichheit, wenn mindestens eine davon vorher gerundet wurde.”
- Also sollten wir `fp1 == fp2` vermeiden.

Das Vergleichsproblem

- Gegeben: `fp1` und `fp2` vom Typ `float` oder `double`
- Leitlinie: “Teste zwei Fließkommazahlen nicht auf Gleichheit, wenn mindestens eine davon vorher gerundet wurde.”
- Also sollten wir `fp1 == fp2` vermeiden.

Wie können wir stattdessen vergleichen?

Idee: Kleine Unterschiede erlauben

Vergleich mit Toleranz

Gegeben eine Toleranz $c > 0$: $\text{fp1} = \text{fp2} \iff |\text{fp1} - \text{fp2}| < c$

($|\dots|$ ist der Betrag. In C++ gibt es dafür keine senkrechten Striche.)

Idee: Kleine Unterschiede erlauben

Vergleich mit Toleranz

Gegeben eine Toleranz $c > 0$: $\text{fp1} = \text{fp2} \iff |\text{fp1} - \text{fp2}| < c$

($|\dots|$ ist der Betrag. In C++ gibt es dafür keine senkrechten Striche.)

Beispiele ($c = 0.001$)

■ $|10.0 - 12.0| = 2.0 > c \Rightarrow$ nicht "gleich"

Idee: Kleine Unterschiede erlauben

Vergleich mit Toleranz

Gegeben eine Toleranz $c > 0$: $\text{fp1} = \text{fp2} \iff |\text{fp1} - \text{fp2}| < c$

($|\dots|$ ist der Betrag. In C++ gibt es dafür keine senkrechten Striche.)

Beispiele ($c = 0.001$)

- $|10.0 - 12.0| = 2.0 > c \Rightarrow$ nicht "gleich"
- $|10.0 - 10.000013| = 0.000013 < c \Rightarrow$ "gleich"

Schreibt die folgende Funktion:

```
// POST: returns true if and only if  $|x - y| < \text{tol}$   
bool equals(double x, double y, double tol) {  
    // ...  
}
```

Mögliche Lösung

```
// POST: returns true if and only if  $|x - y| < \text{tol}$ 
bool equals(double x, double y, double tol) {
    double diff = x - y;
    if (diff < 0) {
        diff *= -1; // absolute value
    }
    return diff < tol;
}
```

Mögliche Lösung

```
// POST: returns true if and only if  $|x - y| < \text{tol}$ 
bool equals(double x, double y, double tol) {
    double diff = x - y;
    if (diff < 0) {
        diff *= -1; // absolute value
    }
    return diff < tol;
}
```

Absolute Differenzen mit einer Toleranz zu vergleichen ist eine gute erste Idee! (Aber: z.B. Probleme, wenn die Zahlen sehr gross sind.)

Wichtig

Wichtig

Diese Folien sollen euch ein Gefühl dafür geben, warum “Löcher” im Wertebereich entstehen. Wie Fliesskommazahlen genau dargestellt werden, lernt ihr in ein paar Wochen in der Vorlesung.

Fragen/Unklarheiten?

10. Extra: Debugging auf **code** expert

Debugging auf **code expert**

Aufgabe

Öffnet das Beispiel “Debugging in CodeExpert” auf **code expert**.

Beschreibung

Findet die grösste (N) und die kleinste negative (n) Zweierpotenz, die als **int** darstellbar ist. Formal:

$$\begin{aligned} N &= \max(\{2^i : i \in \mathbb{N} \wedge 2^i \text{ ist als int darstellbar}\}) \\ n &= \min(\{-2^i : i \in \mathbb{N} \wedge -2^i \text{ ist als int darstellbar}\}) \end{aligned}$$

Debugging auf **code expert** (1/2)

```
#include <iostream>
#include <climits>
int main() {
    int start = 2;
    // Finding the largest power of 2 representable by int
    while (true) {
        if (start * 2 > INT_MAX) {
            std::cout << "Stopping..." << std::endl;
            break;
        }
        start *= 2;
    }

    std::cout << "Largest power of two representable by int: "
              << start << std::endl;
```

Debugging auf **code expert** (2/2)

```
// Finding the smallest negative power of 2 representable by int
start = -2;
while (true) {
    if (start * 2 < INT_MIN)
        std::cout << "Stopping..." << std::endl;
        break;

    start *= 2;
}

std::cout << "Smallest negative power of two representable by int: "
          << start << std::endl;

return 0;
}
```

Debugging auf **code expert**

Frage

Wie soll das Programm funktionieren?

Debugging auf **code expert**

Frage

Wie soll das Programm funktionieren?

Antwort

Zweierpotenzen berechnen, bis wir den grössten **int**-Wert erreichen.

Debugging auf **code expert**

Frage

Wie soll das Programm funktionieren?

Antwort

Zweierpotenzen berechnen, bis wir den grössten **int**-Wert erreichen.
Beim Ausführen ist das Resultat aber falsch!

Debugging auf **code expert**

Frage

Wie soll das Programm funktionieren?

Antwort

Zweierpotenzen berechnen, bis wir den grössten **int**-Wert erreichen.
Beim Ausführen ist das Resultat aber falsch!

Frage

Es gibt einen Fehler im Code, bei dem der Compiler warnt. Wer sieht ihn?

Debugging auf **code expert**

Frage

Wie soll das Programm funktionieren?

Antwort

Zweierpotenzen berechnen, bis wir den grössten **int**-Wert erreichen.
Beim Ausführen ist das Resultat aber falsch!

Frage

Es gibt einen Fehler im Code, bei dem der Compiler warnt. Wer sieht ihn?

Antwort

Beim zweiten **if** fehlen die {}. Das **break** gehört also *nicht* zum **if**, obwohl die Einrückung das suggeriert: die zweite Schleife bricht immer sofort ab.

Debugging auf **code expert**

Frage

Auch mit Klammern stimmt die Ausgabe noch nicht – wir vermuten eine Endlosschleife. Was ist noch falsch?

Debugging auf **code expert**

Frage

Auch mit Klammern stimmt die Ausgabe noch nicht – wir vermuten eine Endlosschleife. Was ist noch falsch?

Antwort

Der Ausdruck `start * 2` führt zu einem *Overflow*, was bei `signed int` undefiniertes Verhalten ist. Hier führt es zu einer Endlosschleife!

Debugging auf **code expert**

Frage

Auch mit Klammern stimmt die Ausgabe noch nicht – wir vermuten eine Endlosschleife. Was ist noch falsch?

Antwort

Der Ausdruck `start * 2` führt zu einem *Overflow*, was bei `signed int` undefiniertes Verhalten ist. Hier führt es zu einer Endlosschleife!

Frage

Wie können wir den Code korrigieren?

Debugging auf **code expert**

Frage

Auch mit Klammern stimmt die Ausgabe noch nicht – wir vermuten eine Endlosschleife. Was ist noch falsch?

Antwort

Der Ausdruck `start * 2` führt zu einem *Overflow*, was bei **signed int** undefiniertes Verhalten ist. Hier führt es zu einer Endlosschleife!

Frage

Wie können wir den Code korrigieren?

Antwort

Umformen, sodass nichts mehr überlaufen kann: `start > INT_MAX / 2`
bzw. `start < INT_MIN / 2` (wie bei *Largest Power!*)

Debugging auf **code expert** – Lösung (1/2)

```
#include <iostream>
#include <climits>
int main() {
    int start = 2;
    // Finding the largest power of 2 representable by int
    while (true) {
        if (start > INT_MAX / 2) {
            std::cout << "Stopping..." << std::endl;
            break;
        }
        start *= 2;
    }

    std::cout << "Largest power of two representable by int: "
              << start << std::endl;
```

Debugging auf **code expert** – Lösung (2/2)

```
// Finding the smallest negative power of 2 representable by int
start = -2;
while (true) {
    if (start < INT_MIN / 2) {
        std::cout << "Stopping..." << std::endl;
        break;
    }
    start *= 2;
}

std::cout << "Smallest negative power of two representable by int: "
          << start << std::endl;

return 0;
}
```

Fragen/Unklarheiten?

11. Extra: Auswahlanweisungen

Aufgabe

Aufgabe “Even Numbers”

Schreibt ein Programm, das eine ganze Zahl n (als `int`) einliest. Ist $n \geq 0$, gibt es die grösste gerade Zahl $m \leq n$ aus. Ist $n < 0$, gibt es 0 aus.

Beispiele $2 \rightarrow 2$, $15 \rightarrow 14$, $43 \rightarrow 42$

Aufgabe

Aufgabe “Even Numbers”

Schreibt ein Programm, das eine ganze Zahl n (als `int`) einliest. Ist $n \geq 0$, gibt es die grösste gerade Zahl $m \leq n$ aus. Ist $n < 0$, gibt es 0 aus.

Beispiele $2 \rightarrow 2$, $15 \rightarrow 14$, $43 \rightarrow 42$

Tipp: Spielt zuerst mit ein paar konkreten Zahlen herum, löst dann die Aufgabe für positive Zahlen und erweitert sie erst am Schluss auf negative Zahlen.

Mögliche Lösung zu *Even Numbers*

Mögliche Lösung zu *Even Numbers*

```
int n;  
std::cin >> n;  
if (n < 0) {  
    n = 0;  
}  
if (n % 2 == 1) {  
    n = n - 1;  
}  
std::cout << n << std::endl;
```

Mögliche Lösung zu *Even Numbers*

```
int n;  
std::cin >> n;  
if (n < 0) {  
    n = 0;  
}  
if (n % 2 == 1) {  
    n = n - 1;  
}  
std::cout << n << std::endl;
```

💡 Warum die negativen Zahlen zuerst behandeln? In C++ ist z.B. $-3 \% 2 == -1$, nicht 1!

Fragen/Unklarheiten?

12. Outro

Bis nächste Woche!