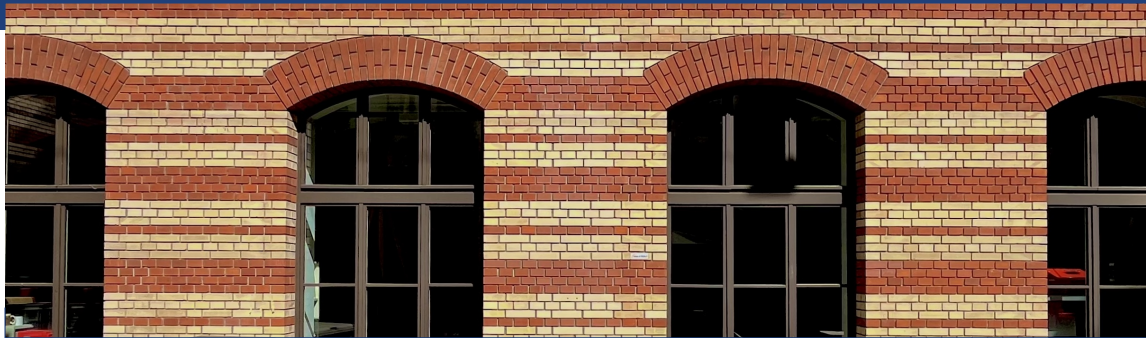




Informatik

Schleifen, Taylor-Reihen, Sichtbarkeitsbereiche (Scopes)

Adel Gavranović

ETH Zürich — 2026

Überblick

Schleifen
Taylor-Reihen berechnen
Sichtbarkeitsbereiche (Scopes)



adelgavranovic.ch

 Webpage

 Mail

1. Follow-up

Follow-up aus letzter Übungsstunde

- Boolesche Ausdrücke minimieren (Übungsstunde 2): nochmals kurz Schritt für Schritt
- Python Tutor richtig kennenlernen
- Wie gross sind `int`, `long` und `double` auf verschiedenen Systemen?
- Erlaubte Hilfsmittel an der Prüfung: 4 A4-Blätter (beidseitig), handschriftlich oder gedruckt

Boolesche Ausdrücke minimieren

Minimiert die folgenden Ausdrücke, wobei `a` und `b` vom Typ `bool` sind und `n` vom Typ `int` ist:

1. `b == true && !(n < 0) && (n != 0) && (n != 6) && (n < 6)`
2. `a && !(a == true) || (b == false)`
3. `!(a != false && !(b == false) == false)`

💡 Wie beim Auswerten: zuerst Klammern setzen (Präzedenz!), dann Stück für Stück vereinfachen.

Lösung 1

`b == true && !(n < 0) && (n != 0) && (n != 6) && (n < 6)`

`b && !(n < 0) && (n != 0) && (n != 6) && (n < 6)` $b == \text{true} \rightarrow b$

`b && n >= 0 && n != 0 && n != 6 && n < 6` $!(n < 0) \rightarrow n >= 0$

`b && n > 0 && n != 6 && n < 6` $n >= 0 \text{ \&\& } n != 0 \rightarrow n > 0$

`b && n > 0 && n < 6` $n < 6 \text{ impliziert schon } n != 6$

Lösung 2

```
a && !(a == true) || (b == false)
```

```
(a && !(a == true)) || (b == false)
```

```
(a && !a) || !b
```

```
false || !b
```

```
!b
```

&& bindet stärker als ||

$a == \text{true} \rightarrow a$, $b == \text{false} \rightarrow !b$

$a \ \&\& \ !a$ ist immer false

Lösung 3

`!(a != false && !(b == false) == false)`

`!((a != false) && (!(b == false)) == false))`

“!” > “==” > “&&”

`!(a && (!!b == false))`

$a \neq \text{false} \rightarrow a, \quad b == \text{false} \rightarrow !b$

`!(a && (b == false))`

$!!b \rightarrow b$

`!(a && !b)`

$b == \text{false} \rightarrow !b$

`!a || b`

De Morgan: $!(x \&\& y) \rightarrow !x \ || \ !y$

Python Tutor

Trotz des Namens: Python Tutor kann auch C++! 

1. Code einfügen
2. Auf “Visualize Execution” klicken
3. Mit **Next** > und < **Prev** Schritt für Schritt vor- und zurückgehen

Man sieht in jedem Schritt:

- welche Zeile gerade ausgeführt wurde und welche als nächstes kommt
- alle Variablen, die gerade existieren, mit ihren aktuellen Werten
- die bisherige Ausgabe

💡 Eingaben am besten durch feste Werte ersetzen, z.B. `int n = 3;` statt `std::cin >> n;`

Wie gross sind `int`, `long` und `double`?

Der C++-Standard schreibt nur *Mindestgrössen* vor, z.B. `int` $\geq$ 16 Bit, `long` $\geq$ 32 Bit, `long long` $\geq$ 64 Bit.

In der Praxis, auf heutigen 64-Bit-Systemen:

	Linux, macOS	Windows
<code>int</code>	32 Bit	32 Bit
<code>long</code>	64 Bit	32 Bit
<code>long long</code>	64 Bit	64 Bit
<code>float</code>	32 Bit	32 Bit
<code>double</code>	64 Bit	64 Bit

💡 Selbst nachschauen: `sizeof(long)` gibt die Grösse in Bytes (1 Byte = 8 Bit). Braucht ihr sicher 64 Bit, nehmt `long long`.

2. Feedback zu **code** expert

Allgemeines bezüglich **code expert**

- Fragen zu den Übungsaufgaben der letzten Woche?
- Bei einer Aufgabe steckengeblieben? Kommt ins Study Center (jeden Mittwoch, Infos auf der Study-Center-Webseite)!

Fragen bezüglich **code expert** eurerseits?

3. Lernziele

Ziele

- ☐ Komplexe Ausdrücke mit arithmetischen und logischen Operatoren auswerten können
- ☐ Mathematische Summen in C++ umsetzen können
- ☐ `for`-, `while`- und `do-while`-Schleifen tracen können
- ☐ Eine Schleife durch eine andere Art von Schleife ausdrücken können

4. Zusammenfassung

Getting on the same page

- Was wurde diese Woche behandelt?

5. Schleifen

Schleifenkorrektheit

Kann man beim Ausführen dieses Programms einen Unterschied zwischen den Ausgaben der drei Schleifen feststellen? Wenn ja, wie?

Nehmt an, dass `n` eine Variable vom Typ `int` ist, deren Wert eingegeben wird.

Löst die Aufgabe alleine (10 Minuten).

```
int n;
std::cin >> n;
int i;
// loop 1
for (i = 1; i <= n; ++i) {
    std::cout << i << "\n";
}
// loop 2
i = 0;
while (i < n) {
    std::cout << ++i << "\n";
}
// loop 3
i = 1;
do {
    std::cout << i++ << "\n";
} while (i <= n);
```

Schleifenkorrektheit – Lösung

Es gibt die folgenden Unterschiede:

- Anders als die Schleifen 1 und 2 gibt Schleife 3 für die Eingabe `n == 0` eine `1` aus, weil der Rumpf einer `do`-Schleife immer einmal ausgeführt wird, bevor die Bedingung geprüft wird. (Dasselbe gilt für alle `n < 0`.)
- Ist `n` die grösstmögliche positive Ganzzahl, zeigen die Schleifen 1 und 3 undefiniertes Verhalten, weil `++i` bzw. `i++` den Wert von `i` über den grössten `int`-Wert hinaus erhöht, bevor die Bedingung `i <= n` die Schleife stoppen kann.

💡 Probiert es aus: Lasst das Programm mit `int n = 0`; statt `std::cin >> n`; in Python Tutor laufen! [Python Tutor](#)

Genauer hingeschaut: `n == INT_MAX`

- **Schleife 1:** Für `i == INT_MAX` ist `i <= n` noch **true**: Ausgabe, dann `++i` $\Rightarrow$ **Overflow!**
- **Schleife 2:** `++i` macht aus `INT_MAX - 1` den Wert `INT_MAX`: Ausgabe, dann ist `i < n` **false** $\Rightarrow$ **fertig**
- **Schleife 3:** `i++` liefert `INT_MAX` für die Ausgabe, erhöht `i` aber gleich mit, *bevor* `i <= n` geprüft wird $\Rightarrow$ **Overflow!**

Ein Overflow bei **signed int** ist undefiniertes Verhalten (wie letzte Woche). Oft wird `i` dabei negativ, `i <= n` bleibt **true** und die Schleife läuft endlos – garantiert ist aber gar nichts.

Fragen/Unklarheiten?

Schleifen ineinander umformen

Formt die folgenden Schleifen in äquivalente Schleifen um. (Ein paar Minuten alleine.)

1. for $\rightarrow$ while

```
for (int i = 0; i < n; ++i) {  
    // BODY  
}
```

2. while $\rightarrow$ for

```
while (condition) {  
    // BODY  
}
```

3. do-while $\rightarrow$ for

```
do {  
    // BODY  
} while (condition);
```

for $\rightarrow$ while

Aufgabe

```
for (int i = 0; i < n; ++i) {  
    // BODY  
}
```

Mögliche Lösung

```
{  
    int i = 0;  
    while (i < n) {  
        // BODY  
        ++i;  
    }  
}
```

Warum der zusätzliche Block {...} aussen herum? Er beschränkt den Scope von `i`, genau wie bei der `for`-Schleife. (Mehr zu `Scopes` später.)

while $\rightarrow$ for

Aufgabe

```
while (condition) {  
    // BODY  
}
```

Mögliche Lösung

```
for (; condition; ) {  
    // BODY  
}
```

💡 init und expression dürfen leer sein!

do-while $\rightarrow$ for

Aufgabe

```
do {  
    // BODY  
} while (condition);
```

Mögliche Lösung

```
{  
    // BODY  
}  
for (; condition; ) {  
    // BODY  
}
```

Der Rumpf einer `do`-Schleife läuft immer mindestens einmal, also führen wir ihn einmal vor der Schleife aus.

Die Klammern geben der ersten Kopie einen eigenen Scope, wie beim Rumpf der `do`-Schleife. (Klammern um alles würden nicht reichen: `condition` sähe sonst die Variablen der ersten Kopie.)

do-while → for: ohne doppelten Rumpf

Den Rumpf zweimal hinzuschreiben ist unschön. Es geht auch so:

```
for (bool first = true; first || condition; first = false) {  
    // BODY  
}
```

- Im ersten Durchlauf ist `first` `true`: dank *Kurzschlussauswertung* wird `condition` gar nicht ausgewertet, der Rumpf läuft also sicher einmal.
- Danach ist `first` `false`: die Schleife läuft genau so lange, wie `condition` `true` ist.

Für Aufmerksame: `continue`

Unsere Umformungen gehen von einem “einfachen” Rumpf aus. Mit `continue` kann es schiefgehen:

```
for (int i = 0; i < n; ++i) {  
    if (i == 0) continue;  
}
```

```
int i = 0;  
while (i < n) {  
    if (i == 0) continue;  
    ++i;  
}
```

In der `for`-Schleife wird `++i` nach `continue` trotzdem ausgeführt, in der `while`-Schleife nicht $\Rightarrow$ Endlosschleife (falls `0 < n`)!

Gut aufgepasst! Für diese Übung ignorieren wir solche Spezialfälle aber.

Fragen/Unklarheiten?

6. Taylor-Reihen berechnen

Taylor-Reihen

Wir schauen uns nun an, wie man komplexe mathematische Reihen berechnet, am Beispiel der Taylor-Reihen.

Taylor-Reihe

...ist eine unendliche Summe von Termen, die eine Funktion mithilfe ihrer Ableitungen an einer bestimmten Stelle approximiert.

Heute approximieren wir damit die Sinusfunktion.

Aufgabe

Aufgabe “Taylor Series”

Schreibt ein Programm, das $\sin x$ mit einer Genauigkeit von 0.000001 berechnet. Dafür könnt ihr die Maclaurin-Reihe verwenden:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

Ausgeschrieben:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Aufwärmen: Von der Reihe zur Schleife

Mathematische Summen und Produkte lassen sich direkt in Schleifen übersetzen:

$$\sum_{i=1}^n f(i)$$

$$\prod_{i=1}^n f(i)$$

```
double sum = 0;
for (int i = 1; i <= n; ++i) {
    sum += f(i);
}
```

```
double prod = 1;
for (int i = 1; i <= n; ++i) {
    prod *= f(i);
}
```

Warum startet die Summe bei 0, das Produkt aber bei 1?
Das sind die neutralen Elemente: $x + 0 = x$ und $x \cdot 1 = x$.

Aufwärmen: Fakultät

Betrachten wir die Formel

$$n!$$

Frage 1

Wie können wir das als Reihe darstellen?

$$n! = \prod_{k=1}^n k = 1 \cdot 2 \cdot \dots \cdot n$$

Nun entwickeln wir ein Programm, das $n!$ berechnet, wobei n eine Eingabe ist.

Das brauchen wir nachher für die Taylor-Reihe!

Aufwärmen: Fakultät

Frage 2.1

Welche Variablen verändern sich während der Berechnung, und welche Typen haben sie?

```
int n;           // input
int k;           // loop variable
int result;      // result
```

Aufwärmen: Fakultät

Frage 2.2

Wie initialisieren wir `k` und `result`, und warum?

`k = 1`: das Produkt beginnt bei $k = 1$.

`result = 1`: neutrales Element der Multiplikation (und $0! = 1$).

```
#include <iostream>
int main() {
    int n;
    int k = 1;
    int result = 1;

    std::cin >> n;

    return 0;
}
```

Aufwärmen: Fakultät

Frage 2.3

Wie schreiben wir die Schleife?
Was ist die Abbruchbedingung
und wie entwickelt sich `result`?

Solange `k <= n`: `result` mit `k`
multiplizieren.

💡 $12! = 479\,001\,600$ passt noch in
einen `int`, $13! = 6\,227\,020\,800$
nicht mehr!

```
#include <iostream>
int main() {
    int n;
    int k = 1;
    int result = 1;

    std::cin >> n;
    for (; k <= n; k++) {
        result = result * k;
    }
    std::cout << result << "\n";
    return 0;
}
```

Fragen/Unklarheiten?

Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 4: Exercise Session”, öffnet “Taylor Series”
- Überlegt euch alleine, wie ihr die Aufgabe mit Stift und Papier lösen würdet (10 Minuten)
- Bildet Zweiergruppen mit euren Sitznachbar:innen, diskutiert eure Ideen und schreibt zusammen das Programm, inklusive `fac(n)` (15 Minuten)
- Danach diktiert ihr mir eure Lösung, und wir schauen sie gemeinsam an (10 Minuten)

Teilaufgaben

1. `fac(x)`: $x!$ berechnen (genau unser Aufwärmprogramm!)
2. `sin(x)`: in jedem Schritt Zähler und Nenner berechnen, mit `fac(x)` für den Nenner und `std::pow(x, y) =  $x^y$` für den Zähler
3. `sin2(x)`: schneller machen, indem ihr Zähler und Nenner des $(n - 1)$ -ten Terms für den n -ten Term wiederverwendet. Ohne `fac` und `std::pow`!

💡 Zuerst Teilaufgaben 1 und 2, Teilaufgabe 3 nur, falls ihr noch Zeit habt.

Testet zuerst mit kleinen Werten wie $x = 0.5$ oder $x = 1$ (warum, sehen wir später).

Tipps

Aus der Aufgabenstellung

1. Welche Schleife passt hier am besten?
2. Löst die Aufgabe zuerst, ohne an die Genauigkeit zu denken. Überlegt dann: Welche Bedingung garantiert die geforderte Genauigkeit? Wann hört man auf, Terme zu addieren?

Zusätzlich

- Startet bei $n = 1$. (Warum? Dazu gleich mehr.)
- Zu Teilaufgabe 3: Betrachtet Zähler und Nenner separat. Was muss man mit dem $(n - 1)$ -ten Zähler bzw. Nenner machen, um den n -ten zu erhalten?

Mögliche Lösung zu *Taylor Series* – fac & sin (1/2)

```
#include <iostream>
#include <cmath>

int fac(int n) {
    int k = 1;
    int result = 1;

    for (; k <= n; k++) {
        result = result * k;
    }

    return result;
}
```

```
double sin(double x) {
    double numerator = x;
    double denominator = 0;

    double sum = x;
    double term;
    double term_abs;
    unsigned int n = 1;

    // ...
}
```

fac ist genau unser
Aufwärmprogramm.

Mögliche Lösung zu *Taylor Series* – fac & sin (2/2)

```
do {
    numerator = std::pow(-1.0, n) * std::pow(x, 2 * n + 1);
    denominator = fac(2 * n + 1);
    term = numerator / denominator;
    sum += term;
    if (term < 0) {
        term_abs = -term;
    } else {
        term_abs = term;
    }
    ++n;
} while (term_abs > 0.000001);

return sum;
}
```

Genauer hingeschaut

- Warum eine `do-while`-Schleife?

Wir müssen zuerst einen Term berechnen, bevor wir prüfen können, ob er klein genug ist.

- Wann hören wir auf?

Sobald der Betrag des letzten Terms höchstens 0.000001 ist. Da die Terme abwechselnd positiv und negativ sind und (am Schluss) immer kleiner werden, ist der Fehler dann höchstens so gross wie der nächste Term.

Vom $(n - 1)$ -ten zum n -ten Term

Der n -te Term ist

$$t_n = \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Zähler $(-1)^n x^{2n+1} = \underbrace{(-1)^{n-1} x^{2n-1}}_{\text{Zähler von } t_{n-1}} \cdot (-x^2)$

Nenner $(2n+1)! = \underbrace{(2n-1)!}_{\text{Nenner von } t_{n-1}} \cdot (2n) \cdot (2n+1)$

Wir müssen also in jedem Schritt nur noch zwei Multiplikationen machen!

Mögliche Lösung zu *Taylor Series* – sin2

```
double sin2(double x) {  
    double numerator = x;  
    double denominator = 1;  
  
    double sum = x;  
    double term;  
    double term_abs;  
    unsigned int n = 1;  
  
    // ...  
}
```

```
do {  
    numerator *= -(x * x);  
    denominator *= (2 * n) * (2 * n + 1);  
    term = numerator / denominator;  
    sum += term;  
    if (term < 0) {  
        term_abs = -term;  
    } else {  
        term_abs = term;  
    }  
    ++n;  
} while (term_abs > 0.000001);  
  
return sum;  
}
```

Warum startet n bei 1?

- `sum` startet schon mit dem Term für $n = 0$: $t_0 = \frac{x}{1!} = x$. Die Schleife muss also nur noch die Terme ab $n = 1$ berechnen.
- In `sin2` baut jeder Term auf dem vorherigen auf, darum starten dort auch `numerator = x` und `denominator = 1! = 1` mit den Werten für $n = 0$.

Und wenn wir bei $n = 0$ starten?

- `sin` zählt x doppelt.
- `sin2` multipliziert den Nenner mit $(2 \cdot 0) \cdot (2 \cdot 0 + 1) = 0 \Rightarrow$ Division durch 0!
(Bei `double` stürzt das nicht ab: man erhält `inf` oder `nan`, `sin2` liefert `nan` oder hängt.)

Warum ist `sin2` besser?

- **Effizienter:** `sin` berechnet $(2n + 1)!$ und x^{2n+1} in jedem Schritt von vorne, `sin2` braucht pro Schritt nur zwei Multiplikationen.
- **Kein Overflow:** `fac` rechnet mit `int`, aber $13!$ passt nicht mehr hinein. `sin` braucht $13!$ schon ab $|x| \approx 1.4$ und liefert z.B. für $x = 2$ Unsinn oder terminiert gar nicht. `sin2` rechnet alles mit `double`.

Darum vorhin der Tipp mit kleinen x : `main` ruft zuerst `sin` auf. Bleibt das hängen, erscheint auch die Ausgabe von `sin2` nie.

Fragen/Unklarheiten?

7. Sichtbarkeitsbereiche (Scopes)

Scopes

Frage

Wer weiss noch, was Scopes sind und warum wir sie brauchen?

Antwort

Scopes definieren die Codeabschnitte, in denen eine Variable existiert. Der Scope einer Variable beginnt bei ihrer Definition und endet am Ende des Blocks, in dem sie definiert wurde. Zum Beispiel:

```
if (x < 7) {  
    int a = 8;           // <-- a's scope BEGINS here  
    std::cout << a;     // fine, prints 8  
}                        // <-- a's scope ENDS here  
std::cout << a;         // compiler error: a does not exist
```

Warum ist das wichtig?

- Endet der Scope einer Variable, wird sie freigegeben: sie “stirbt”.
- In zwei Wochen lernt ihr *Referenzen* kennen. Um sie richtig zu verwenden, ist es sehr wichtig zu wissen, wie lange jede Variable lebt.

Fehler beheben?

Ein möglicher Weg, den
Kompilierfehler zu beheben, wäre:

```
int a = 2;  
if (x < 7) {  
    int a = 8;  
    std::cout << a;  
}  
std::cout << a;
```

Frage

Was gibt dieses Programm aus,
wenn `x == 2`?

Antwort

Es gibt 82 aus.

Warum? Wir tracen das Programm
Schritt für Schritt, mit der Notation
aus dem Program Tracing Guide.
(Diese Notation brauchen wir später
oft, für Funktionsaufrufe und
Referenzen.)



Program Tracing Guide

Scopes tracen

```
int a = 2;  
if (x < 7) {  
    int a = 8;  
    std::cout << a;  
}  
std::cout << a;
```

a (aussen)

2

Ausgabe

Scopes tracen

```
int a = 2;  
if (x < 7) {  
    int a = 8;  
    std::cout << a;  
}  
std::cout << a;
```

a (aussen)

Ausgabe

2

$2 < 7 \rightarrow \text{true}$

Scopes tracen

```
int a = 2;  
if (x < 7) {  
    int a = 8;  
    std::cout << a;  
}  
std::cout << a;
```

a (aussen)

2

a (innen)

8

Ausgabe

Neues a im inneren Scope: es verdeckt das äussere a

Scopes tracen

```
int a = 2;  
if (x < 7) {  
    int a = 8;  
    std::cout << a;  
}  
std::cout << a;
```

a (aussen)

2

a (innen)

8

Ausgabe

8

a bezeichnet hier das innerste sichtbare a

Scopes tracen

```
int a = 2;  
if (x < 7) {  
    int a = 8;  
    std::cout << a;  
}  
std::cout << a;
```

a (aussen)

2

Ausgabe

8

Das innere a existiert nicht mehr: sein Scope ist zu Ende

Scopes tracen

```
int a = 2;  
if (x < 7) {  
    int a = 8;  
    std::cout << a;  
}  
std::cout << a;
```

a (aussen)

2

Ausgabe

82

Ausgabe: 82

Scopes in Schleifen

Frage

Was ist der Scope von `sum`, `i` und `a` im folgenden Beispiel?

```
int sum = 0;
for (int i = 0; i < 5; ++i) {
    int a;
    std::cin >> a;
    sum += a;
}
```

Antwort

`sum` (Mindestens) das gesamte Snippet

`i` Die gesamte `for`-Schleife

`a` Eine Schleifeniteration. Anders gesagt: Am Anfang des Schleifenrumpfs hat `a` *nicht* garantiert den Wert, den es am Ende des Rumpfs in der vorherigen Iteration hatte.

Fragen/Unklarheiten?

8. Outro

Bis nächste Woche!