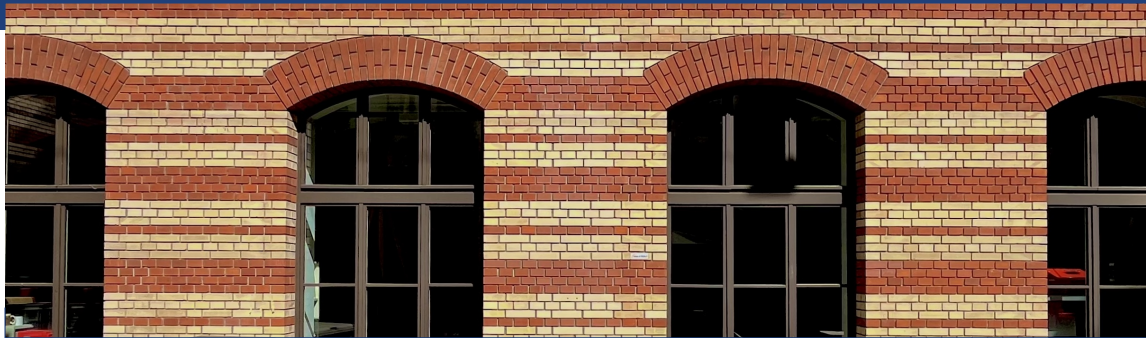




# Informatik

Funktionen, PRE und POST, Stepwise Refinement, Sauberer Code

Adel Gavranović

ETH Zürich — 2026

# Überblick

Neu auf **code expert**

Sauberer Code

Taylor-Reihe Schritt für Schritt

`assert`

Funktionen

Headers und Namespaces

Stepwise Refinement

Extra: Alte Theorieprüfungsfragen



adelgavranovic.ch

 Webpage

 Mail

# 1. Follow-up

---

# Follow-up aus letzter Übungsstunde

- Taylor-Reihe: nochmals Schritt für Schritt, von der Formel zum Code
- Sauberer Code: richtig einrücken und gute Namen wählen (mit Live-Demo)
- **code expert**: Antworten schön formatiert in Markdown schreiben (Live-Demo)

## 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)!  Study Center

Fragen bezüglich **code expert** eurerseits?

### 3. Lernziele

---

# Ziele

- ☐ Vor- und Nachbedingungen (PRE und POST) für einfache Funktionen schreiben können
- ☐ Eine Aufgabe mit Stepwise Refinement lösen können
- ☐ `assert` gezielt zum Debuggen einsetzen können
- ☐ Programme mit Funktionsaufrufen tracen können
- ☐ Lesbaren Code schreiben: sauber eingerückt und mit guten Namen

## 4. Zusammenfassung

---

# Getting on the same page

- Was wurde diese Woche behandelt?

## 5. Neu auf **code** expert

---

# Versteckte Testfälle

- Ab nächster Woche haben *einige* **code expert**-Übungen versteckte Testfälle
- Klappen sie, sehen sie aus wie normale Testfälle. Schlagen sie fehl, wird die erwartete Ausgabe *nicht* angezeigt
- So könnt ihr die Lösung nicht gegen die erwartete Ausgabe “hard-coden” (das ist sowieso nicht erlaubt)
- Nach der Abgabefrist werden die Testfälle durch *neue* ersetzt und neu bewertet, genau wie an der Prüfung
- Gekennzeichnet sind diese Übungen mit “[hidden tests]”

💡 Überlegt euch selbst Testfälle, besonders Randfälle (0, negative Zahlen, sehr grosse Zahlen ... )!

# Persistente Eingabe

- Neue, optionale Funktion in den “[hidden tests]”-Übungen
- Ihr schreibt eure Eingabe einmal in die Datei `input.txt`. Sie bleibt über mehrere Durchläufe erhalten und lässt sich im Editor bearbeiten
- Statt der Eingabe tippt ihr im Terminal nur `f` (ohne Leerzeichen) und Enter: das Programm liest dann `input.txt`
- Hat keinen Einfluss auf eure XP

Eselsbrücke: `f` wie *file*.

# Kleine Demonstration

Code Example “Rounding Numbers” auf **code expert**

1. `frac >= 0.5` durch `frac > 0.5` ersetzen: normale *und* versteckte Tests schlagen fehl
2. `num >= 0` durch `true` ersetzen: *nur* versteckte Tests schlagen fehl
3. Persistente Eingabe: `input.txt` füllen, ausführen, `f` eingeben

💡 Alle Tests grün heisst nicht, dass die Lösung stimmt. An der Prüfung werden die Tests ersetzt!

# Antworten in Markdown

Textantworten auf **code expert** sind Markdown. Ein paar Zeichen genügen für schöne Antworten:

## Ihr schreibt

```
Type `int`, value `1`, because:
```

```
- `9 * a + b` is `46`  
- `46 % 5` is **1**
```

```
```cpp  
auto c = (9 * a + b) % a;  
```
```

## Man sieht

Type int, value 1, because:

■ 9 \* a + b is 46

■ 46 % 5 is **1**

```
auto c = (9 * a + b) % a;
```

# Markdown: das Wichtigste

| Ihr schreibt                | Wofür                                        |
|-----------------------------|----------------------------------------------|
| <code>`x + 1`</code>        | Code im Text: Variablen, Ausdrücke, Ausgaben |
| <code>```cpp ... ```</code> | Codeblock über mehrere Zeilen                |
| <code>**wichtig**</code>    | <b>fett</b>                                  |
| <code>\$x^{2n+1}\$</code>   | Formel im Text: $x^{2n+1}$                   |
| <code>- Punkt</code>        | Aufzählung                                   |
| Leerzeile                   | Neuer Absatz                                 |

💡 Schaut euch vor der Abgabe die Vorschau an. Sieht sie gut aus, ist es richtig.

Live-Demo auf **code expert**

# Fragen/Unklarheiten?

## 6. Sauberer Code

---

# Warum sauberer Code?

- Code wird viel öfter gelesen als geschrieben: von euch in zwei Wochen, von euren Teampartner:innen und von mir beim Korrigieren
- Sauber eingerückter Code zeigt seine Struktur auf einen Blick, und Fehler fallen schneller auf
- Stil gehört zur Bewertung: Ist der Code schwer lesbar, kann es TA-Punkte kosten

Alles Weitere steht im Style Guide des Kurses: [Style Guide](#)

# Was ist hier schlecht?

```
#include<iostream>
int f(int x){int s=0;
while(x>0){s+=x%10;x/=10;}
    return s;}
int main(){
int n;std::cin>>n;
    int a=f(n);
        if(a%3==0)std::cout<<n<<" is divisible by 3\n";
else
std::cout<<n<<" is not divisible by 3\n";
return 0;}
```

Das Programm kompiliert und ist korrekt.

**Aufgabe** Findet mindestens vier Dinge, die das Lesen schwer machen.

(2 Minuten mit eurer Sitznachbarin oder eurem Sitznachbarn)

# Besser

```
#include <iostream>

// PRE:  n >= 0
// POST: returns the sum of the
//       decimal digits of n
int digit_sum(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

```
int main() {
    int number;
    std::cin >> number;

    // divisible by 3 iff
    // the digit sum is
    if (digit_sum(number) % 3 == 0) {
        std::cout << number
                   << " is divisible by 3\n";
    } else {
        std::cout << number
                   << " is not divisible by 3\n";
    }

    return 0;
}
```

Gleiche Ausgabe wie vorher: Stil ändert nichts am Verhalten, nur an der Lesbarkeit.

# Regeln: Einrücken und Leerzeichen

- Jeder Block { ... } wird eine Stufe tiefer eingerückt, immer gleich viel (z.B. 4 Leerzeichen)
- Die schliessende } steht auf der Höhe der Zeile, die den Block öffnet
- Leerzeichen um Operatoren (`a + b`, `x == 0`) und nach Schlüsselwörtern (`if (`, `while (`)
- Eine Anweisung pro Zeile, Leerzeilen zwischen Funktionen und zwischen Eingabe, Berechnung und Ausgabe

💡 Setzt { } auch bei einzeiligen `if` und `for`. erinnert euch an “Debugging in CodeExpert” in Woche 3!

# Regeln: Namen und Kommentare

- Namen sagen, was eine Variable enthält: `sum`, `divisor` statt `s`, `x2`
- Funktionen heißen nach dem, was sie tun: `digit_sum`, `count_perfect_numbers`. Für `bool`: eine Frage wie `is_perfect`
- Kurze Namen wie `i`, `n` nur für Schleifenzähler und wo die Mathematik sie vorgibt
- Einheitlich bleiben, im Kurs meist `snake_case`
- Über jede Funktion PRE und POST. Kommentare erklären, *warum* etwas so ist, nicht *was* dasteht

Fragen/Unklarheiten?

## 7. Taylor-Reihe Schritt für Schritt

---

# Rückblick: Aufgabe “Taylor Series”

## Aufgabe “Taylor Series”

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

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Heute bauen wir `sin(x)` Stück für Stück auf. Jeder Teil der Formel bekommt eine Farbe, und der Code, der ihn berechnet, dieselbe.

`sin2` lassen wir weg.

## Schritt 6: wann hören wir auf?

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

bis  $|t_n| \leq 0.000001$

Ist der Betrag eines Terms höchstens 0.000001, sind wir genau genug.

`std::abs` aus `<cmath>` gibt den Betrag.

```
double sin(double x) {  
    double sum = 0;  
    double term;  
    unsigned int n = 0;  
    do {  
        double numerator = std::pow(-1.0, n)  
                           * std::pow(x, 2 * n + 1);  
        double denominator = fac(2 * n + 1);  
        term = numerator / denominator;  
        sum += term;  
        ++n;  
    } while (std::abs(term) > 0.000001);  
    return sum;  
}
```

# Formel und Code auf einen Blick

| Formel                | Code                                      |                      |
|-----------------------|-------------------------------------------|----------------------|
| $\sum_{n=0}^{\infty}$ | <code>sum = 0; do ... ++n;</code>         | Schleife ab $n = 0$  |
| $(-1)^n$              | <code>std::pow(-1.0, n)</code>            | Vorzeichen           |
| $x^{2n+1}$            | <code>std::pow(x, 2 * n + 1)</code>       | Potenz               |
| $(2n + 1)!$           | <code>fac(2 * n + 1)</code>               | Nenner               |
| $+ t_n$               | <code>sum += term;</code>                 | Term dazuzählen      |
| $ t_n  \leq 0.000001$ | <code>std::abs(term) &gt; 0.000001</code> | Abbruch (umgekehrt!) |

Die Schleife läuft, *solange* die Bedingung gilt. Darum steht im Code das Gegenteil des Abbruchkriteriums.

# Genauer hingeschaut

- Auf **code expert** berechnet die Musterlösung `term_abs` mit **if/else**. Das macht genau dasselbe wie `std::abs(term)`.
- `numerator` und `denominator` braucht es nur im Rumpf, also deklarieren wir sie auch dort.
- Die Musterlösung nimmt den Term für  $n = 0$  (also  $x$ ) vorweg: `sum = x` und `n = 1`. Gleiches Resultat, ein Durchlauf weniger.
- Warum reicht  $|t_n| \leq 0.000001$ ? Die Terme wechseln das Vorzeichen und werden kleiner, also ist der Fehler höchstens so gross wie der nächste Term.

# Ausprobieren

| $x$ | unser <code>sin(x)</code> | richtig  |
|-----|---------------------------|----------|
| 0.5 | 0.479426                  | 0.479426 |
| 1   | 0.841471                  | 0.841471 |
| 1.5 | 0.997495                  | 0.997495 |
| 2   | hängt                     | 0.909297 |
| 3   | hängt                     | 0.141120 |

Was ist bei  $x = 2$  los?

## Was ist bei $x = 2$ los?

| $n$ | <code>fac(2 * n + 1)</code> | richtig wäre      | <code>term</code> |
|-----|-----------------------------|-------------------|-------------------|
| 5   | 39 916 800                  | 39 916 800        | -0.0000513        |
| 6   | 1 932 053 504               | 6 227 020 800     | 0.0000042         |
| 7   | 2 004 310 016               | 1 307 674 368 000 | -0.0000163        |
| ⋮   |                             |                   |                   |
| 17  | 0                           |                   | $-\infty$         |

- $13! = 6\,227\,020\,800$  passt nicht mehr in einen `int` (max. 2 147 483 647): Overflow!
- Die Terme werden wieder grösser, und ab  $n = 17$  ist `term`  $= \pm\infty$ , also nie kleiner als 0.000001  $\Rightarrow$  Endlosschleife

# Was fac verlangt

```
// PRE:  0 <= n <= 12 (13! does not fit into an int)
// POST: returns n!
int fac(int n) {
    int result = 1;
    for (int k = 1; k <= n; ++k) {
        result *= k;
    }
    return result;
}
```

Und sin erbt das: ab  $|x| \approx 1.4$  braucht es schon 13!.

Genau dafür gibt es PRE und POST. Dazu gleich mehr!

# Fragen/Unklarheiten?

## 8. assert

---

# Demo: “Debugging with Assert”

Code Example “Debugging with Assert” auf [code expert](#)

Mit Eingabe 0 stürzt das Programm ab: Exit-Code -8, aber wo?

**Frage** Bei einem so kleinen Programm sieht man den Fehler sofort. Aber wie bekommen wir in einem grossen Programm eine Fehlermeldung mit der genauen Stelle?

**Antwort** Mit `assert(divisor != 0);` prüfen wir unsere Annahme. Schlägt sie fehl, gibt es Exit-Code -6 und eine Meldung mit Datei und Zeile.

# assert

```
#include <iostream>
#include <cassert>

int main() {
    int a;
    int b;
    std::cin >> a >> b;
    assert(b != 0); // no division by 0
    std::cout << a / b << "\n";
    return 0;
}
```

- `#include <cassert>` nicht vergessen
- Ist die Bedingung `false`, bricht das Programm *sofort* ab, mit Datei und Zeile in Meldung
- Für die fertige Version lassen sich alle `asserts` auf einmal ausschalten

# Wofür sind asserts nützlich?

- In langen Programmen, um den Überblick zu behalten
- Um falsche (Benutzer-)Eingaben sofort zu erkennen, statt später undefiniertes Verhalten zu bekommen
- Als zusätzliche Dokumentation, wenn mehrere Leute am selben Code arbeiten
- Um eine PRE zu prüfen, z.B. in fac: `assert(0 <= n && n <= 12);`

# Fragen/Unklarheiten?

## 9. Funktionen

---

# PRE und POST

```
// PRE:  what the function expects from the caller
// POST: what the function guarantees afterwards
int your_function(int a, int b) {
    // ...
}
```

- **PRE:** muss beim Aufruf gelten, meist eine Bedingung an die Argumente
- **POST:** gilt nach dem Aufruf (Rückgabewert, Effekte), *falls* PRE gegolten hat
- Wird PRE verletzt, verspricht die Funktion nichts mehr (siehe `fac(13)`)

Sie müssen nicht sehr detailliert sein, aber vollständig: Was wird erwartet, was kommt heraus?

# PRE und POST I

**Findet PRE und POST für diese Funktion** (2–3 Minuten)

```
// PRE: ???  
// POST: ???  
double f(double i, double j, double k) {  
    if (i > j) {  
        if (i > k) return i;  
        else return k;  
    } else {  
        if (j > k) return j;  
        else return k;  
    }  
}
```

# PRE und POST I – Lösung

```
// PRE:  (not needed)
// POST: return value is the maximum of i, j and k
double f(double i, double j, double k) {
    if (i > j) {
        if (i > k) return i;
        else return k;
    } else {
        if (j > k) return j;
        else return k;
    }
}
```

Mit diesem Namen bräuchte es fast keinen Kommentar mehr:  
`max_of_three!`

# PRE und POST II

**Findet PRE und POST für diese Funktion** (2–3 Minuten)

```
// PRE:  ???  
// POST:  ???  
double g(int i, int j) {  
    double r = 0.0;  
    for (int k = i; k <= j; ++k) {  
        r += 1.0 / k;  
    }  
    return r;  
}
```

# PRE und POST II – Lösung

```
// PRE: 0 not contained in {i, ..., j} and i <= j and j < INT_MAX
// POST: return value is the sum 1/i + 1/(i+1) + ... + 1/j
double g(int i, int j) {
    double r = 0.0;
    for (int k = i; k <= j; ++k) {
        r += 1.0 / k;
    }
    return r;
}
```

- Warum  $j < \text{INT\_MAX}$ ? Sonst läuft  $++k$  über (wie letzte Woche!)
- Warum  $1.0 / k$ ?  $1 / k$  wäre Ganzzahldivision, also 0 für  $k > 1$

# Fragen/Unklarheiten?

# Was gibt dieses Programm aus?

```
#include <iostream>

int f(int i) {
    return i * i;
}

int g(int i) {
    return i * f(i) * f(f(i));
}
```

```
void h(int i) {
    std::cout << g(i) << "\n";
}

int main() {
    int i;
    std::cin >> i;
    h(i);
    return 0;
}
```

Over- und Underflows dürft ihr vernachlässigen.

# Was gibt dieses Programm aus? – Lösung

$h(i)$  gibt  $g(i)$  aus. Wir setzen Schritt für Schritt ein:

$$\begin{aligned}g(i) &= i * f(i) * f(f(i)) \\&= i * (i*i) * f(f(i)) \\&= i * (i*i) * f(i*i) \\&= i * (i*i) * ((i*i)*(i*i)) \\&= i^7\end{aligned}$$

💡 Beim Tracen von Funktionsaufrufen bekommt jeder Aufruf seine eigene Tabelle. [Program Tracing Guide](#)

# Fehlersuche

```
# include <iostream>

double f (double x) {
    return g(2.0 * x);
}

bool g (double x) {
    return x % 2.0 == 0;
}
```

```
void h () {
    std::cout << result;
}

int main () {
    double result = f(3.0);
    h();

    return 0;
}
```

**Findet drei Fehler in diesem Programm.**

# Fehlersuche – Lösung

1. `f` ruft `g` auf, bevor `g` deklariert ist: der Scope von `g` beginnt erst später.

Lösung: Vorwärtsdeklaration `bool g(double x);` vor `f`

2. Den Modulo-Operator `%` gibt es nicht für `double`.

Lösung: z.B. `std::fmod(x, 2.0)` aus `<cmath>`

3. `h` “sieht” `result` nicht: die Variable gehört zum Scope von `main`.

Lösung: als Argument übergeben, `void h(double result)` und `h(result);`

💡 Beim Tracen dürft ihr nur Variablen aus der *aktuellen* Tabelle lesen. Fehlt eine, habt ihr einen Bug gefunden.

# Anzahl Teiler (falls Zeit bleibt)

Schreibt eine Funktion `number_of_divisors`, die ein `int n` nimmt und die Anzahl Teiler von  $n$  zurückgibt (inklusive 1 und  $n$ ).

```
// PRE:  1 <= n < INT_MAX
// POST: returns number of divisors of n (incl. 1 and n)
int number_of_divisors(int n) {
    // your code
}
```

Beispiel: 6 hat 4 Teiler, nämlich 1, 2, 3, 6

# Anzahl Teiler – Lösung

```
// PRE:  1 <= n < INT_MAX
// POST: returns number of divisors of n (incl. 1 and n)
int number_of_divisors(int n) {
    assert(n > 0);
    int counter = 0;
    for (int i = 1; i <= n; ++i) {
        if (n % i == 0) {
            ++counter;
        }
    }
    return counter;
}
```

Das assert prüft einen Teil der PRE (dafür braucht es <cassert>).

# Fragen/Unklarheiten?

# Mal schauen, was ihr gelernt habt

- Geht auf `expert.ethz.ch`
- Logt euch ein
- Geht zu “Code Examples”
- Unter “Week 5: Exercise Session”, öffnet “Math functions”
- 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 (10 Minuten)
- Danach diktiert ihr mir eure Lösung, und wir schauen sie gemeinsam an (10 Minuten)

# Aufgabe

## Aufgabe “Math functions”

Schreibt ein Programm, das für drei ganze Zahlen Folgendes berechnet:

- die Summe, den Durchschnitt (als Fließkommazahl)
- das Maximum und das Minimum

Das Programm liest zuerst eine Zahl `task`, die bestimmt, was berechnet wird, und danach die drei Zahlen. Implementiert alle Funktionen gemäss ihrer PRE und POST.

Beispiel: Eingabe 1 1 2 3, Ausgabe `Sum = 6`

Signaturen nicht ändern, keine zusätzlichen Bibliotheken. Testet jede Funktion einzeln, auch mit negativen und gleichen Zahlen.

# Mögliche Lösung zu *Math functions* (1/2)

```
// POST: Returns the sum of all input values
int sum(int a, int b, int c) {
    return a + b + c;
}

// POST: Returns the arithmetic mean of all input values
float average(int a, int b, int c) {
    return sum(a, b, c) / 3.0;
}
```

- average verwendet sum wieder: dafür sind Funktionen da!
- Warum 3.0 und nicht 3? Für 1 2 4 gäbe / 3 das Resultat 2 statt 2.33333

## Mögliche Lösung zu *Math functions* (2/2)

```
// POST: Returns the largest
//      input value
int maximum(int a, int b, int c) {
    int maxVal = a;
    if (b > maxVal)
        maxVal = b;
    if (c > maxVal)
        maxVal = c;
    return maxVal;
}
```

```
// POST: Returns the smallest
//      input value
int minimum(int a, int b, int c) {
    int minVal = a;
    if (b < minVal)
        minVal = b;
    if (c < minVal)
        minVal = c;
    return minVal;
}
```

Alternative: `return -maximum(-a, -b, -c);`

# Genauer hingeschaut

- Braucht `sum` wirklich keine PRE?

Doch: 1 2147483647 1 0 gibt `sum = -2147483648`. Overflow! Streng genommen: PRE:  $a + b + c$  passt in einen `int`.

- Funktioniert die Alternative für `minimum` immer?

Nicht für `INT_MIN`: `-INT_MIN` passt nicht in einen `int`.

# Fragen/Unklarheiten?

## 10. Headers und Namespaces

---

# Live-Demo: “CPP Headers & Namespaces”

Code Example auf **code expert**

1. Ausführen: warum schlägt es fehl?
2. Mit einer Vorwärtsdeklaration reparieren
3. Header `myAwesomeFunctions.h` anschauen
4. Den Header mit `#include "myAwesomeFunctions.h"` einbinden
5. Beide `even` über ihren Namespace mit `::` aufrufen

# Vorwärtsdeklaration

```
void B(int i); // declaration only

void A(int i) {
    if (i <= 0) return;
    std::cout << "A";
    B(i / 2); // B is defined below
}

void B(int i) {
    if (i <= 0) return;
    std::cout << "B";
    A(i / 2);
}
```

Eine Funktion kann man erst nach ihrer ersten *Deklaration* verwenden.

Die Definition ist auch eine Deklaration, aber man kann die Deklaration allein nach oben ziehen.

A(8) gibt ABAB aus

# Header-Files

perfect.h (Deklaration)

```
#pragma once

bool is_perfect(int number);
```

perfect.cpp (Definition)

```
#include "perfect.h"

bool is_perfect(int number) {
    // ...
}
```

main.cpp

```
#include <iostream>
#include "perfect.h"

int main() {
    // can use is_perfect
}
```

- <...>: Standardbibliothek
- "...": eigene Dateien
- #pragma once: nur einmal einbinden

# Namespaces

```
namespace ifmp {  
    // POST: "Hi" was written  
    void output_func() {  
        std::cout << "Hi";  
    }  
}  
  
int main() {  
    ifmp::output_func();  
    return 0;  
}
```

Namespaces gruppieren Funktionen, Typen usw. und verhindern Namenskonflikte zwischen gleich benannten Funktionen.

Genau wie bei `std::cout`:  
`namespace::name`

# Warum kein using namespace std;?

```
#include <iostream>
#include <algorithm>
using namespace std;

int count = 0;

int main() {
    count++;
    cout << count << "\n";
    return 0;
}
```

error: reference to 'count'  
is ambiguous

<algorithm> hat auch ein  
std::count. Mit  
using namespace std; landen  
alle Namen aus std bei euch, und  
es kracht.

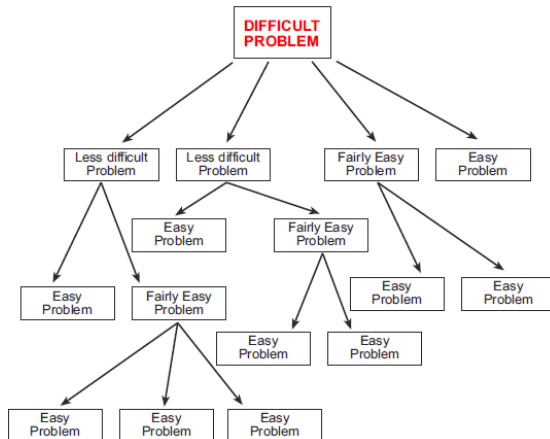
💡 Schreibt lieber std:: aus.

# Fragen/Unklarheiten?

# 11. Stepwise Refinement

---

# Grundidee



- Ein schwieriges Problem in einfachere Teilprobleme zerlegen
- ... so lange, bis jedes Teilproblem einfach ist
- Jedes Teilproblem wird eine Funktion, mit PRE und POST

# Aufgabe

## Aufgabe “Perfect Numbers”

Schreibt ein Programm, das zählt, wie viele vollkommene Zahlen (*perfect numbers*) es im Bereich  $[a, b]$  gibt (mit  $a, b > 0$  und  $b < \text{INT\_MAX}$ ). Verwendet Stepwise Refinement und teilt die Lösung in sinnvolle Funktionen auf. Die Funktion `is_perfect` aus `perfect.h` ist gegeben.

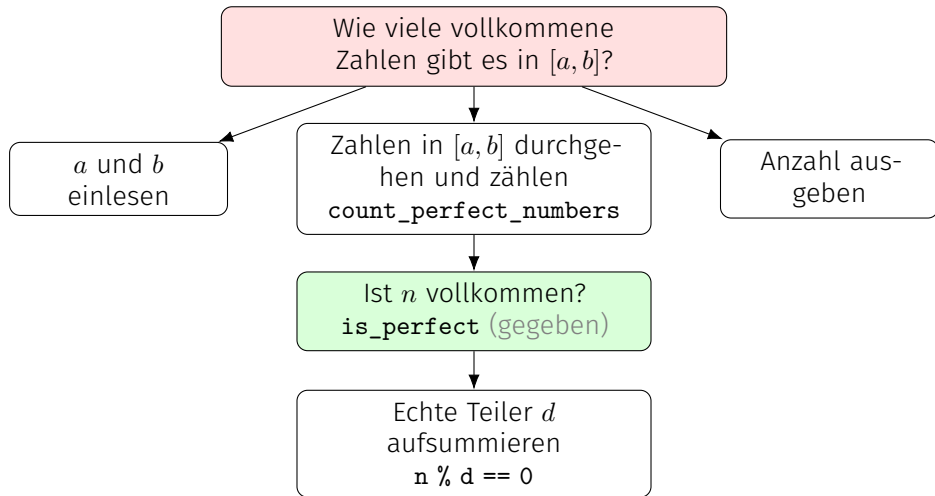
$n$  ist vollkommen, wenn  $n$  gleich der Summe seiner echten Teiler ist:

- $28 = 1 + 2 + 4 + 7 + 14$  ist vollkommen
- $12 < 1 + 2 + 3 + 4 + 6$  ist es nicht

# Mal schauen, was ihr gelernt habt

- Unter “Week 5: Exercise Session” öffnet “Perfect Numbers”
- *Nicht sofort programmieren!*
- Überlegt euch alleine, in welche einfacheren Teilprobleme ihr die Aufgabe zerlegen würdet (5 Minuten)
- Danach bauen wir zusammen den Problembaum und schreiben die Lösung (10 Minuten)

# Problembaum



# Gegeben: is\_perfect

```
#include "perfect.h"

bool is_perfect(int number) {
    int sum = 0;
    for (int d = 1; d < number; ++d) {
        if (number % d == 0) {
            sum += d;
        }
    }
    return sum == number;
}
```

Die zwei unteren Ebenen des Baums: Teiler suchen und aufsummieren.

# Mögliche Lösung zu *Perfect Numbers*

```
#include <iostream>
#include "perfect.h"

// PRE:  0 < a and b < INT_MAX
// POST: returns the number of perfect
//       numbers in [a, b]
int count_perfect_numbers(int a, int b) {
    int count = 0;
    for (int i = a; i <= b; ++i) {
        if (is_perfect(i)) {
            count++;
        }
    }
    return count;
}
```

```
int main() {
    // input
    int a;
    int b;
    std::cin >> a >> b;

    // computation
    int count = count_perfect_numbers(a, b);

    // output
    std::cout << count << std::endl;

    return 0;
}
```

# Genauer hingeschaut

- Warum verlangt die Aufgabe  $b < \text{INT\_MAX}$ ?  
Für  $b = \text{INT\_MAX}$  wäre  $i \leq b$  immer wahr und  $++i$  würde überlaufen.  
Dasselbe wie bei `g` vorhin!
- Jede Funktion löst genau ein Teilproblem und lässt sich einzeln testen.
- Für  $[1, 10\,000]$  gibt es 4 (6, 28, 496, 8128).  $[1, 100\,000]$  dauert schon ein paar Sekunden: `is_perfect` probiert jeden möglichen Teiler aus.

# Fragen/Unklarheiten?

## 12. Extra: Alte Theorieprüfungsfragen

---

# Aus 2022-08 MAVT – Expression Evaluation

`auto` heisst, dass der Compiler den Typ bestimmt. Es steht also für den Typ, den ihr herausfinden sollt.

Typ und Wert von `c`?

```
int a = 5;  
int b = 1;  
auto c = (9 * a + b) % a;
```

int, 1

Typ und Wert von `c`?

```
int a = 5;  
double b = 1;  
auto c = (9.0 * a + b) / a;
```

double, 9.2

# Aus 2022-08 252-08(47/48/56) – Loop Termination

```
int sum = 17;
int i = 1;

do {
    i += sum;
    sum = sum / 2;
} while (i > sum && sum >= 0);

std::cout << sum;
```

Welche Aussage beschreibt die Ausgabe am besten?

- 17
- 8
- Terminiert nie
- 18

Terminiert nie!

- Division zweier positiver `ints` ist nie negativ  $\Rightarrow$  `sum >= 0` ist immer `true`
- `sum` wird kleiner (bis 0), `i` wird grösser  $\Rightarrow$  `i > sum` ist immer `true`

Kein Overflow: ab `sum == 0` bleibt `i` bei 33.

# Fragen/Unklarheiten?

## 13. Outro

---

# Allgemeine Fragen?

Gibt es noch offene Fragen zu dem, was wir bisher in den Übungsstunden gemacht haben?

**Bis nächste Woche!**