

10. Funktionen II

Stepwise Refinement, Gültigkeitsbereich, Bibliotheken,
Standardfunktionen

Stepwise Refinement

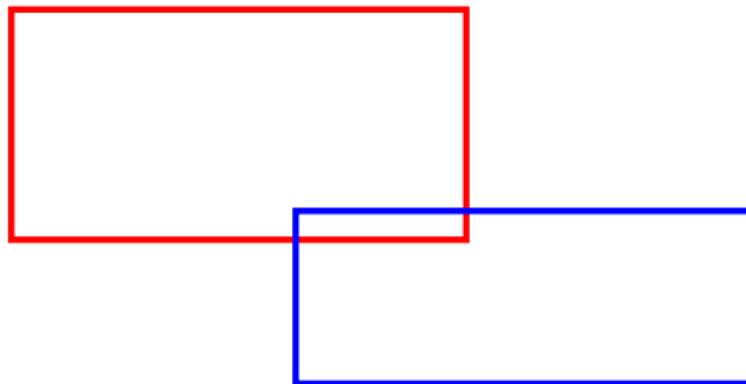
- Einfache *Programmiertechnik* zum Lösen komplexer Probleme

Stepwise Refinement

- Problem wird schrittweise gelöst. Man beginnt mit einer groben Lösung auf sehr hohem Abstraktionsniveau (nur Kommentare und fiktive Funktionen).
- In jedem Schritt werden Kommentare durch Programmtext ersetzt und Funktionen unterteilt.
- Die Verfeinerung bezieht sich auch auf die Entwicklung der Datenrepräsentation (mehr dazu später).
- Wird die Verfeinerung so weit wie möglich durch Funktionen realisiert, entstehen Teillösungen, die auch bei anderen Problemen eingesetzt werden können.
- Stepwise Refinement fördert (aber ersetzt nicht) das strukturelle Verständnis des Problems.

Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!



Grobe Lösung

(Include-Direktiven ignoriert)

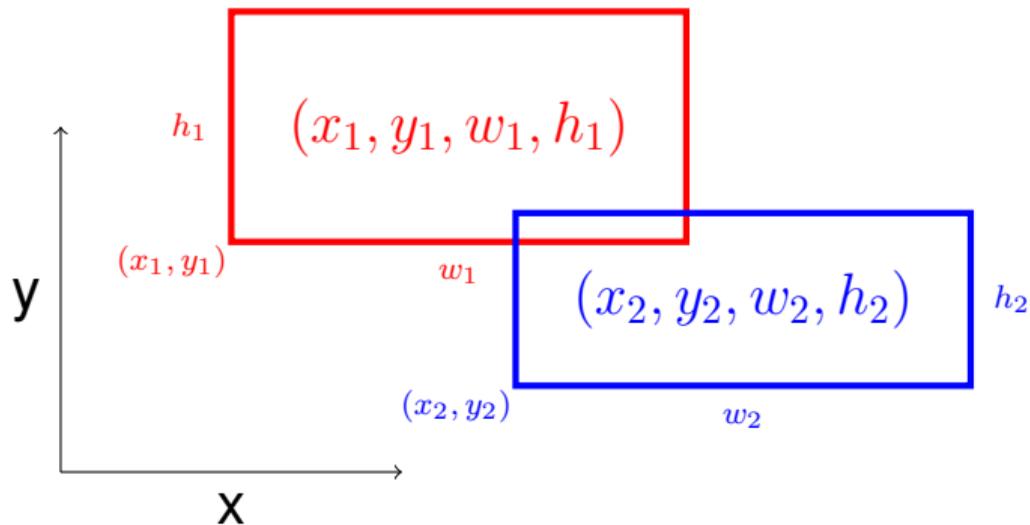
```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe

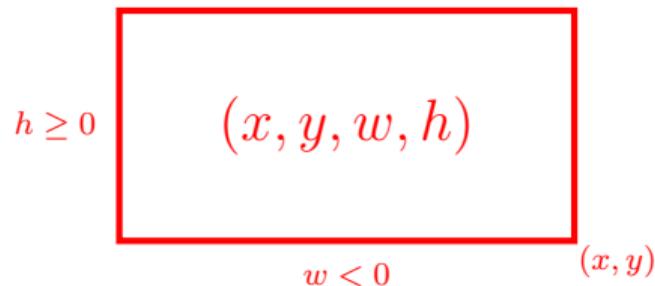
    return 0;
}
```

Verfeinerung 1: Eingabe Rechtecke



Verfeinerung 1: Eingabe Rechtecke

Breite w und/oder Höhe h dürfen negativ sein!



Verfeinerung 1: Eingabe Rechtecke

```
int main()
{
    std::cout << "Enter two rectangles [x y w h each] \n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;

    // Schnitt?

    // Ausgabe der Loesung

    return 0;
}
```

Verfeinerung 2: Schnitt? und Ausgabe

```
int main()
{
```

Eingabe ✓

```
    bool clash = rectangles_intersect (x1,y1,w1,h1,x2,y2,w2,h2);
```

```
    if (clash)
```

```
        std::cout << "intersection!\n";
```

```
    else
```

```
        std::cout << "no intersection!\n";
```

```
    return 0;
```

```
}
```

Verfeinerung 3: Schnittfunktion...

```
bool rectangles_intersect (int x1, int y1, int w1, int h1,  
                           int x2, int y2, int w2, int h2)  
{  
    return false; // todo  
}
```

```
int main() {
```

Eingabe ✓

Schnitt ✓

Ausgabe ✓

```
    return 0;
```

```
}
```

Verfeinerung 3: Schnittfunktion...

```
bool rectangles_intersect (int x1, int y1, int w1, int h1,  
                           int x2, int y2, int w2, int h2)  
{  
    return false; // todo  
}
```

Funktion main ✓

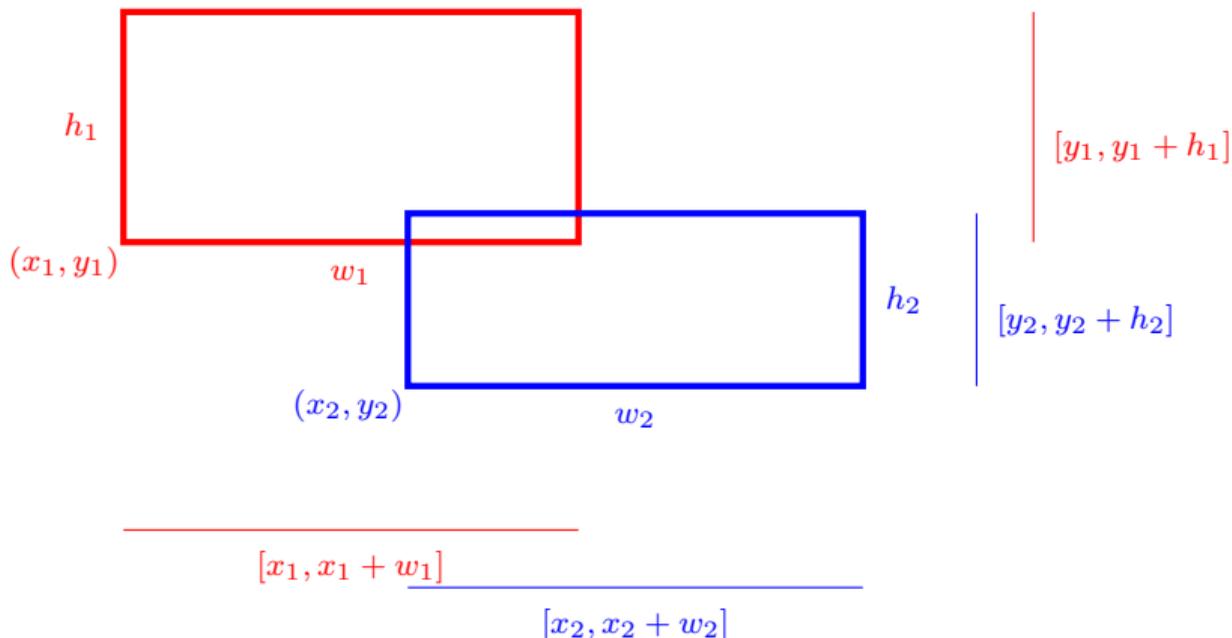
Verfeinerung 3:

...mit PRE und POST!

```
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles,  
//       where w1, h1, w2, h2 may be negative.  
// POST: returns true if (x1, y1, w1, h1) and  
//       (x2, y2, w2, h2) intersect  
bool rectangles_intersect (int x1, int y1, int w1, int h1,  
                           int x2, int y2, int w2, int h2)  
{  
    return false; // todo  
}
```

Verfeinerung 4: Intervallschnitte

Zwei Rechtecke schneiden sich genau dann, wenn sich ihre x - und y -Intervalle schneiden.



Verfeinerung 4: Intervallschnitte

```
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
//       w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1), (x2, y2, w2, h2) intersect
bool rectangles_intersect (int x1, int y1, int w1, int h1,
                           int x2, int y2, int w2, int h2)
{
    return intervals_intersect (x1, x1 + w1, x2, x2 + w2)
        && intervals_intersect (y1, y1 + h1, y2, y2 + h2); ✓
}
```

Verfeinerung 4: Intervallschnitte

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,  
//       with [a,b] := [b,a] if a>b  
// POST: returns true if [a1, b1],[a2, b2] intersect  
bool intervals_intersect (int a1, int b1, int a2, int b2)  
{  
    return false; // todo  
}
```

Funktion rectangles_intersect ✓

Funktion main ✓

Verfeinerung 5: Min und Max

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,  
//       with [a,b] := [b,a] if a>b  
// POST: returns true if [a1, b1],[a2, b2] intersect  
bool intervals_intersect (int a1, int b1, int a2, int b2)  
{  
    return max(a1, b1) >= min(a2, b2)  
        && min(a1, b1) <= max(a2, b2); ✓  
}
```

Verfeinerung 5: Min und Max

```
// POST: the maximum of x and y is returned
int max (int x, int y){
    if (x>y) return x; else return y;
}
```

gibt es schon in der Standardbibliothek

```
// POST: the minimum of x and y is returned
int min (int x, int y){
    if (x<y) return x; else return y;
}
```

Funktion `intervals_intersect` ✓

Funktion `rectangles_intersect` ✓

Funktion `main` ✓

Nochmal zurück zu Intervallen

```
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,  
//       with [a,b] := [b,a] if a>b  
// POST: returns true if [a1, b1],[a2, b2] intersect  
bool intervals_intersect (int a1, int b1, int a2, int b2)  
{  
    return std::max(a1, b1) >= std::min(a2, b2)  
        && std::min(a1, b1) <= std::max(a2, b2); ✓  
}
```

Das haben wir schrittweise erreicht!

```
#include<iostream>
#include<algorithm>

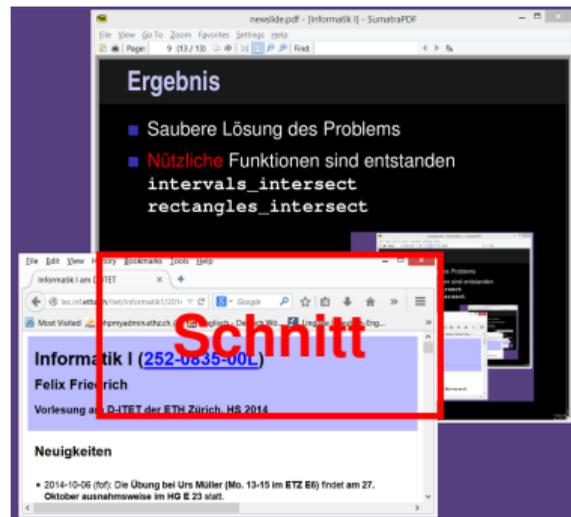
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect (int a1, int b1, int a2, int b2)
{
    return std::max(a1, b1) >= std::min(a2, b2)
        && std::min(a1, b1) <= std::max(a2, b2);
}

// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
//      w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1),(x2, y2, w2, h2) intersect
bool rectangles_intersect (int x1, int y1, int w1, int h1,
                          int x2, int y2, int w2, int h2)
{
    return intervals_intersect (x1, x1 + w1, x2, x2 + w2)
        && intervals_intersect (y1, y1 + h1, y2, y2 + h2);
}

int main ()
{
    std::cout << "Enter two rectangles [x y w h each]\n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;
    bool clash = rectangles_intersect (x1,y1,w1,h1,x2,y2,w2,h2);
    if (clash)
        std::cout << "intersection!\n";
    else
        std::cout << "no intersection!\n";
    return 0;
}
```

Ergebnis

- Saubere Lösung des Problems
- **Nützliche** Funktionen sind entstanden
`intervals_intersect`
`rectangles_intersect`



Wo darf man eine Funktion benutzen?

```
#include<iostream>
```

```
int main()
```

```
{
```

```
    std::cout << f(1); // Fehler: f undeklariert
```

```
    return 0;
```

```
}
```

```
int f (int i) // Gültigkeitsbereich von f ab hier
```

```
{
```

```
    return i;
```

```
}
```

Gültigkeit f
↓

Gültigkeitsbereich einer Funktion

- ist der Teil des Programmes, in dem die Funktion aufgerufen werden kann
- ist definiert als die Vereinigung der Gültigkeitsbereiche aller ihrer Deklarationen (es kann mehrere geben)

Deklaration einer Funktion: wie Definition aber ohne *block*.

```
double pow (double b, int e)
```

So geht's also nicht...

```
#include<iostream>
```

```
int main()
```

```
{
```

```
    std::cout << f(1); // Fehler: f undeklariert
```

```
    return 0;
```

```
}
```

```
int f (int i) // Gültigkeitsbereich von f ab hier
```

```
{
```

```
    return i;
```

```
}
```

Gültigkeit f



...aber so!

```
#include<iostream>
int f (int i); // Gueltigkeitsbereich von f ab hier

int main()
{
    std::cout << f(1);
    return 0;
}

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

Forward Declarations, wozu?

Funktionen, die sich gegenseitig aufrufen:

```
int g(...); // forward declaration

int f (...) // f ab hier gültig
{
    g(...) // ok
}

int g (...)
{
    f(...) // ok
}
```

Gültigkeit g

Gültigkeit f

Wiederverwendbarkeit

- Funktionen wie `rectangles_intersect` und `pow` sind in vielen Programmen nützlich.
- “Lösung:” Funktion einfach ins Hauptprogramm hineinkopieren, wenn wir sie brauchen!
- Hauptnachteil: wenn wir die Funktionsdefinition ändern wollen, müssen wir *alle* Programme ändern, in denen sie vorkommt.

Level 1: Auslagern der Funktion

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Level 1: Inkludieren der Funktion

```
// Prog: callpow2.cpp
// Call a function for computing powers.
```

```
#include <iostream>
```

```
#include "math.cpp" ← Dateiangebe relativ zum Arbeitsverzeichnis
```

```
int main()
{
    std::cout << pow( 2.0, -2) << "\n";
    std::cout << pow( 1.5, 2) << "\n";
    std::cout << pow( 5.0, 1) << "\n";
    std::cout << pow(-2.0, 9) << "\n";

    return 0;
}
```

Nachteil des Inkludierens

- `#include` kopiert die Datei (`math.cpp`) in das Hauptprogramm (`callpow2.cpp`).
- Der Compiler muss die Funktionsdefinition für jedes Programm neu übersetzen.
- Das kann bei sehr vielen und grossen Funktionen sehr lange dauern.

Level 2: Getrennte Übersetzung...

von `math.cpp` unabhängig vom Hauptprogramm:

```
double pow(double b,  
           int e)  
{  
    ...  
}
```

`math.cpp`

`g++ -c math.cpp`

```
001110101100101010  
000101110101000111  
000101100011000111  
111100001101010001  
111111101000111010  
010101101011010001  
100101111100101010
```

`math.o`

Level 2: Getrennte Übersetzung...

Deklaration aller benötigten Symbole in sog. *Header* Datei.

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is be  
double pow (double b, int e);
```

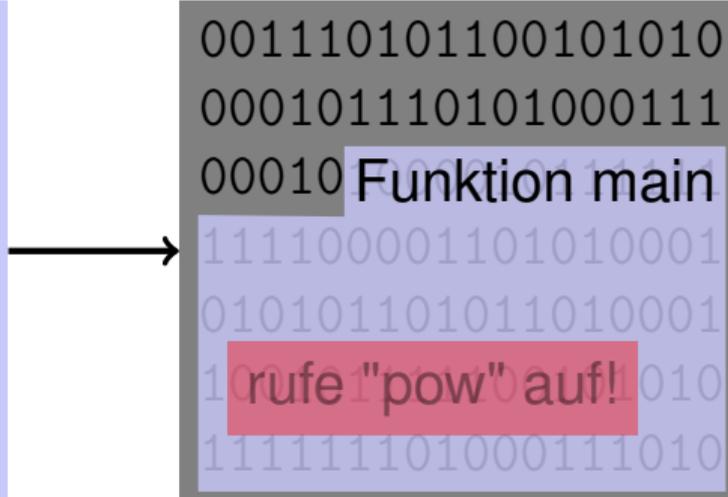
math.h

Level 2: Getrennte Übersetzung...

des Hauptprogramms unabhängig von `math.cpp`, wenn eine *Deklaration* von `math` inkludiert wird.

```
#include <iostream>
#include "math.h"
int main()
{
    std::cout << pow(2,-2) << "\n";
    return 0;
}
```

callpow3.cpp



```
001110101100101010
000101110101000111
00010 Funktion main
111100001101010001
010101101011010001
1 rufe "pow" auf! 1010
111111101000111010
```

callpow3.o

Der Linker vereint...

```
001110101100101010
000101110101000111
000101110101000111
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

math.o

+

```
001110101100101010
000101110101000111
000101110101000111
111100001101010001
010101101011010001
100101111100101010
111111101000111010
```

callpow3.o

... was zusammengehört

```
001110101100101010
000101110101000111
00010 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

math.o

+

```
001110101100101010
000101110101000111
00010 Funktion main
111100001101010001
010101101011010001
10 rufe "pow" auf! 010
111111101000111010
```

callpow3.o

=

```
001110101100101010
000101110101000111
00010 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
001110101100101010
000101110101000111
00010 Funktion main
111100001101010001
010101101011010001
10 rufe addr auf! 010
111111101000111010
```

Ausführbare Datei callpow

Verfügbarkeit von Quellcode?

Beobachtung

`math.cpp` (Quellcode) wird nach dem Erzeugen von `math.o` (Object Code) nicht mehr gebraucht.

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

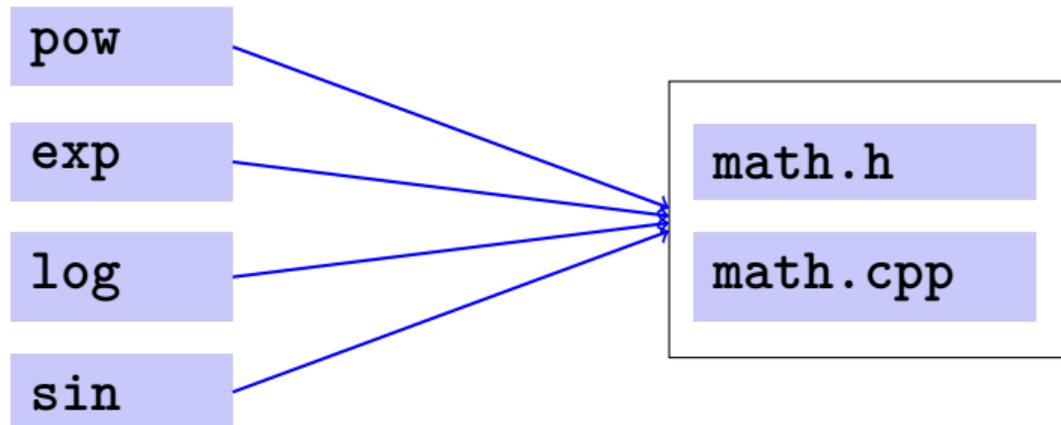
Header-Dateien sind dann die *einzigsten* lesbaren Informationen.

„Open Source“ Software

- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte “Hacker”.
- Selbst im kommerziellen Bereich ist „open source“ auf dem Vormarsch.
- Lizenzen erzwingen die Nennung der Quellen und die offene Weiterentwicklung. Beispiel: GPL (GNU General Public License).
- Bekannte „Open Source“ Softwares: Linux (Betriebssystem), Firefox (Browser), Thunderbird (Email-Programm)

Bibliotheken

- Logische Gruppierung ähnlicher Funktionen



Namensräume...

```
// ifmpmath.h
// A small library of mathematical functions
namespace ifmp {

    // PRE: e >= 0 || b != 0.0
    // POST: return value is b^e
    double pow (double b, int e);

    ....
    double exp (double x);
    ...
}
```

... vermeiden Namenskonflikte

```
#include <cmath>
#include "ifmpmath.h"

int main()
{
    double x = std::pow (2.0, -2); // <cmath>
    double y = ifmp::pow (2.0, -2); // ifmpmath.h
}
```

Funktionen aus der Standardbibliothek

- vermeiden die Neuerfindung des Rades (wie bei `ifmp::pow`);
- führen auf einfache Weise zu interessanten und effizienten Programmen;
- garantieren einen Qualitäts-Standard, der mit selbstgeschriebenen Funktionen kaum erreicht werden kann.

Primzahltest mit sqrt

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, n - 1\}$ ein Teiler von n ist.

```
unsigned int d;  
for (d=2; n % d != 0; ++d);
```

Primzahltest mit sqrt

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, \lfloor \sqrt{n} \rfloor\}$ ein Teiler von n ist.

```
const unsigned int bound = std::sqrt(n);  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```

- Das funktioniert, weil `std::sqrt` auf die nächste darstellbare `double`-Zahl rundet (IEEE Standard 754).
- Andere mathematische Funktionen (`std::pow, ...`) sind in der Praxis fast so genau.

Primzahltest mit sqrt

```
// Test if a given natural number is prime.
#include <iostream>
#include <cassert>
#include <cmath>
@
@

int main ()
{
    // Input
    unsigned int n;
    std::cout << "Test if n>1 is prime for n =? ";
    std::cin >> n;
    assert (n > 1);

    // Computation: test possible divisors d up to sqrt(n)
    @const unsigned int bound = std::sqrt(n);
    unsigned int d;
    for (d = 2; d <= bound && n % d != 0; ++d);@

    // Output
    if (d <= bound)
        // d is a divisor of n in {2,...,[sqrt(n)]}
        std::cout << n << " = " << d << " * " << n / d << ".\n";
    else
        // no proper divisor found
        std::cout << n << " is prime.\n";

    return 0;
}
```

```
void swap (int x, int y) {  
    int t = x;  
    x = y;  
    y = t;  
}  
  
int main(){  
    int a = 2;  
    int b = 1;  
    swap (a, b);  
    assert (a==1 && b == 2); // fail! 😞  
}
```

```
// POST: values of x and y are exchanged
void swap (int& x, int& y) {
    int t = x;
    x = y;
    y = t;
}

int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a==1 && b == 2); // ok! 😊
}
```

Sneak Preview: Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen

Referenztypen (z.B. `int&`)

