

## 10. Funktionen II

Vor- und Nachbedingungen Stepwise Refinement,  
Gültigkeitsbereich, Bibliotheken, Standardfunktionen

### Vor- und Nachbedingungen

- beschreiben (möglichst vollständig) was die Funktion „macht“
- dokumentieren die Funktion für Benutzer / Programmierer (wir selbst oder andere)
- machen Programme lesbarer: wir müssen nicht verstehen, *wie* die Funktion es macht
- werden vom Compiler ignoriert
- Vor- und Nachbedingungen machen – unter der Annahme ihrer Korrektheit – Aussagen über die Korrektheit eines Programmes möglich.

315

316

### Vorbedingungen

Vorbedingung (precondition):

- Was muss bei Funktionsaufruf gelten?
- Spezifiziert *Definitionsbereich* der Funktion.

$0^e$  ist für  $e < 0$  undefiniert

```
// PRE: e >= 0 || b != 0.0
```

### Nachbedingungen

Nachbedingung (postcondition):

- Was gilt nach Funktionsaufruf?
- Spezifiziert *Wert* und *Effekt* des Funktionsaufrufes.

Hier nur Wert, kein Effekt.

```
// POST: return value is b^e
```

317

318

## Vor- und Nachbedingungen

- sollten korrekt sein:
- *Wenn* die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.

Funktion `pow`: funktioniert für alle Basen  $b \neq 0$

319

## Vor- und Nachbedingungen

- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage.
- C++-Standard-Jargon: „Undefined behavior“.

Funktion `pow`: Division durch 0

320

## Vor- und Nachbedingungen

- Vorbedingung sollte so *schwach* wie möglich sein (möglichst grosser Definitionsbereich)
- Nachbedingung sollte so *stark* wie möglich sein (möglichst detaillierte Aussage)

321

## Fromme Lügen...

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is b^e
```

ist formal inkorrekt:

- Überlauf, falls  $e$  oder  $b$  zu gross sind
- $b^e$  vielleicht nicht als `double` Wert darstellbar (Löcher im Wertebereich)

322

## Fromme Lügen... sind erlaubt.

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

Die exakten Vor- und Nachbedingungen sind plattformabhängig und meist sehr kompliziert. Wir abstrahieren und geben die mathematischen Bedingungen an.  $\Rightarrow$  Kompromiss zwischen formaler Korrektheit und lascher Praxis.

323

## Prüfen von Vorbedingungen...

- Vorbedingungen sind nur Kommentare.
- Wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

324

## ...mit Assertions

```
#include <cassert>
...
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e) {
    assert (e >= 0 || b != 0);
    double result = 1.0;
    ...
}
```

325

## Nachbedingungen mit Assertions

- Das Ergebnis „komplizierter“ Berechnungen ist oft einfach zu prüfen.
- Dann lohnt sich der Einsatz von `assert` für die Nachbedingung

```
// PRE: the discriminant p*p/4 - q is nonnegative
// POST: returns larger root of the polynomial x^2 + p x + q
double root(double p, double q)
{
    assert(p*p/4 >= q); // precondition
    double x1 = - p/2 + sqrt(p*p/4 - q);
    assert(equals(x1*x1+p*x1+q,0)); // postcondition
    return x1;
}
```

327

## Ausnahmen (Exception Handling)

- Assertions sind ein grober Hammer; falls eine Assertion fehlschlägt, wird das Programm hart abgebrochen.
- C++ bietet elegantere Mittel (Exceptions), um auf solche Fehlschläge situationsabhängig (und oft auch ohne Programmabbruch) zu reagieren.
- „Narrensichere“ Programmen sollten nur im Notfall abbrechen und deshalb mit Exceptions arbeiten; für diese Vorlesung führt das aber zu weit.

328

## Stepwise Refinement

- Einfache *Programmiertechnik* zum Lösen komplexer Probleme

32 Nikolaus Wirth. Program development by stepwise refinement. Commun. ACM 14, 4, 1971

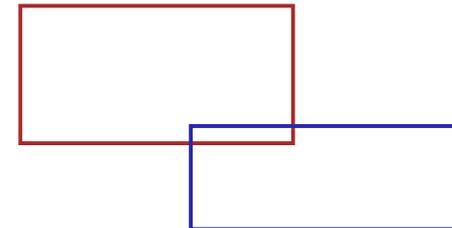
## Stepwise Refinement

- 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 implementiert unterteilt (demselben Prinzip folgend).
- 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.

330

## Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!



331

## Grobe Lösung

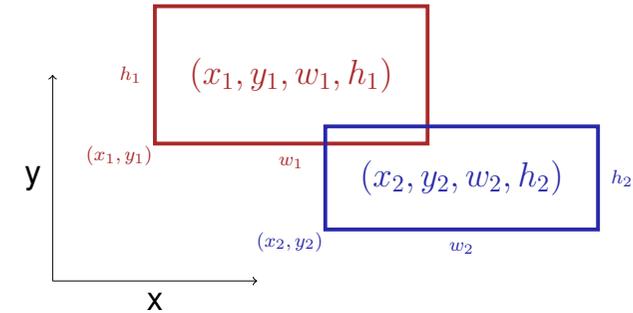
```
(Include-Direktiven ausgelassen)
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe der Loesung

    return 0;
}
```

## Verfeinerung 1: Eingabe Rechtecke

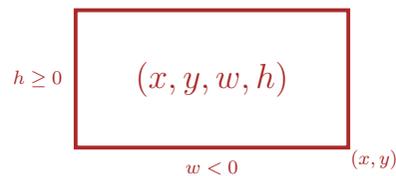


334

335

## Verfeinerung 1: Eingabe Rechtecke

Breite  $w$  und/oder Höhe  $h$  dürfen negativ sein!



336

## 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;
}
```

337

## Verfeinerung 2: Schnitt? und Ausgabe

```
int main()
{
    Eingabe Rechtecke ✓

    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;
}
```

338

## 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 Rechtecke ✓

    Schnitt? ✓

    Ausgabe der Loesung ✓

    return 0;
}
```

339

## 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 ✓

340

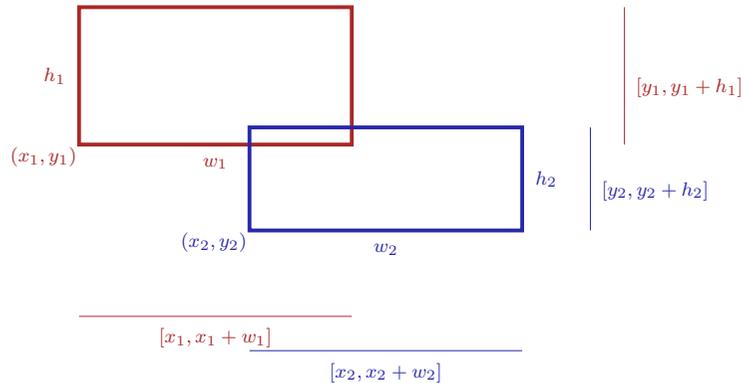
## Verfeinerung 3: ...mit PRE und POST!

```
// 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
}
```

341

## Verfeinerung 4: Intervallschnitt

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



342

## 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); ✓
}
```

343

## 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 ✓

344

## 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); ✓
}
```

345

## 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 ✓

346

## 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); ✓
}
```

347

## 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);
}

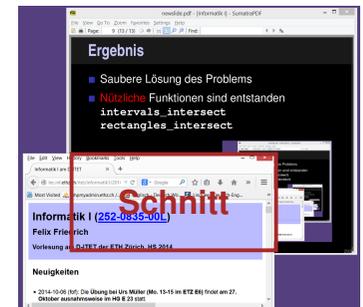
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
// w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1),(x2, y2, w2, h2) intersect
bool rectangles_intersect(int x1, int y1, int w1, int h1,
                          int x2, int y2, int w2, int h2)
{
    return intervals_intersect(x1,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

348

## Ergebnis

- Saubere Lösung des Problems
- **Nützliche** Funktionen sind entstanden  
intervals\_intersect  
rectangles\_intersect



349

## Wo darf man eine Funktion benutzen?

```
#include <iostream>

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

int f(int i) // Gueltigkeitsbereich von f ab hier
{
    return i;
}
```

Gültigkeit f  
↓

350

## 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 {...}.

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

351

## So geht's also nicht...

```
#include <iostream>

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

int f(int i) // Gueltigkeitsbereich von f ab hier
{
    return i;
}
```

Gültigkeit f  
↓

352

## ...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;
}
```

353

## 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

354

## 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.

355

## 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;
}
```

356

## Level 1: Inkludieren der Funktion

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

#include <iostream>
#include "mymath.cpp" ← Datei im 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;
}
```

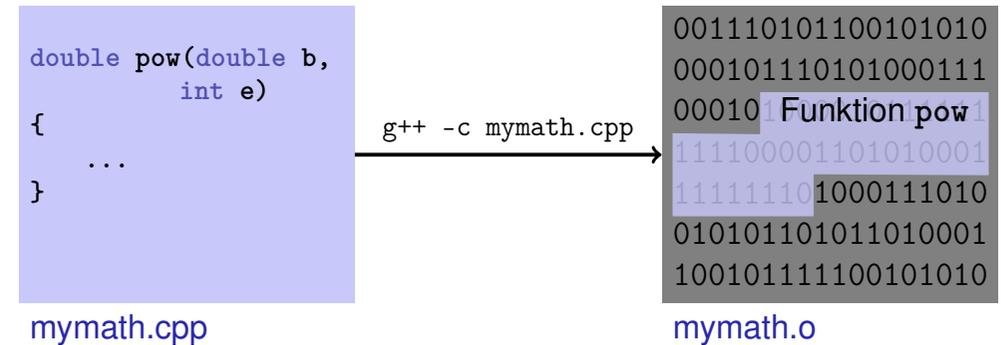
357

## Nachteil des Inkludierens

- `#include` kopiert die Datei (`mymath.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 `mymath.cpp` unabhängig vom Hauptprogramm:



358

359

## Level 2: Getrennte Übersetzung

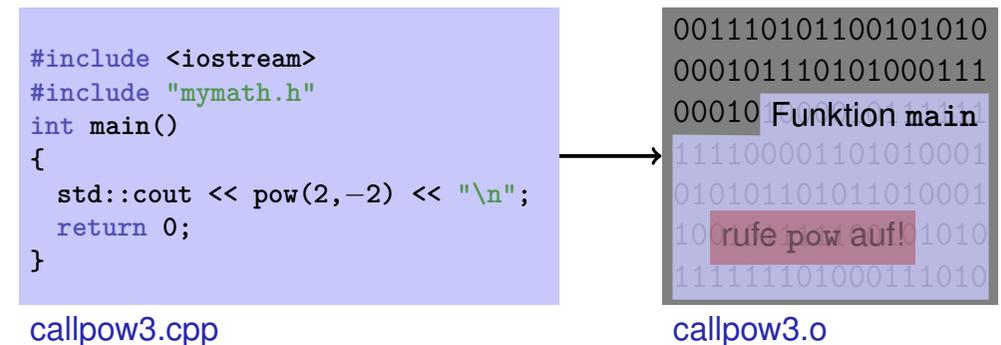
Deklaration aller benötigten Symbole in sog. *Header* Datei.

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

mymath.h

## Level 2: Getrennte Übersetzung

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



360

361

## Der Linker vereint...

```
001110101100101010
000101110101000111
00010 Funktion pow
111100001101010001
11111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
00010 Funktion main
111100001101010001
010101101011010001
10 rufe pow auf! 1010
11111101000111010
```

callpow3.o

## ... was zusammengehört

```
001110101100101010
000101110101000111
00010 Funktion pow
111100001101010001
11111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
00010 Funktion main
111100001101010001
010101101011010001
10 rufe pow auf! 1010
11111101000111010
```

callpow3.o

=

```
001110101100101010
000101110101000111
00010 Funktion pow
111100001101010001
11111101000111010
010101101011010001
100101111100101010
001110101100101010
000101110101000111
00010 Funktion main
111100001101010001
010101101011010001
10 rufe addr auf! 010
11111101000111010
```

Ausführbare Datei callpow3

362

363

## Verfügbarkeit von Quellcode?

### Beobachtung

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

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

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

## Open-Source-Software

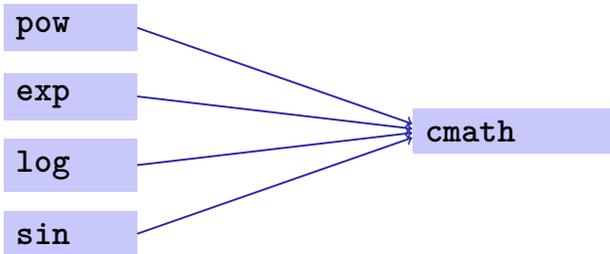
- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte „Hacker“.
- Selbst im kommerziellen Bereich ist Open-Source-Software auf dem Vormarsch.
- Lizenzen erzwingen die Nennung der Quellen und die offene Weiterentwicklung. Beispiel: GPL (GNU General Public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

364

365

## Bibliotheken

- Logische Gruppierung ähnlicher Funktionen



## Namensräume...

```
// cmath
namespace std {

    double pow(double b, int e);

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

366

367

## ... vermeiden Namenskonflikte

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

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

368

## Namensräume / Kompilationseinheiten

In C++ ist das Konzept der separaten Kompilation *unabhängig* vom Konzept der Namensräume.

In manchen anderen Sprachen, z.B. Modula / Oberon (zum Teil auch bei Java) definiert die Kompilationseinheit gerade einen Namensraum.

369

## Funktionen aus der Standardbibliothek

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

## Beispiel: 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);
```

370

371

## 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.

```
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).

## 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)  
    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;  
}
```

372

373

## Funktionen sollten mehr können!

## Swap ?

```
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! ☹️
}
```

375

## Funktionen sollten mehr können!

## Swap ?

```
// 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! 😊
}
```

376

## 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&)

377