9. Funktionen II

Stepwise Refinement, Gültigkeitsbereich,
Bibliotheken, Standardfunktionen

Stepwise Refinement

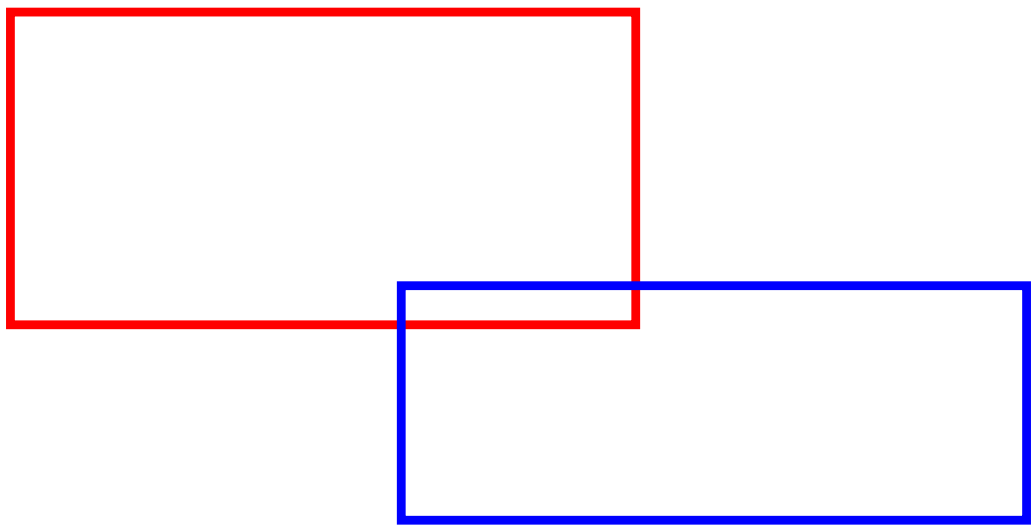
- Einfache *Programmiertechnik* zum Lösen komplexer Probleme

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!

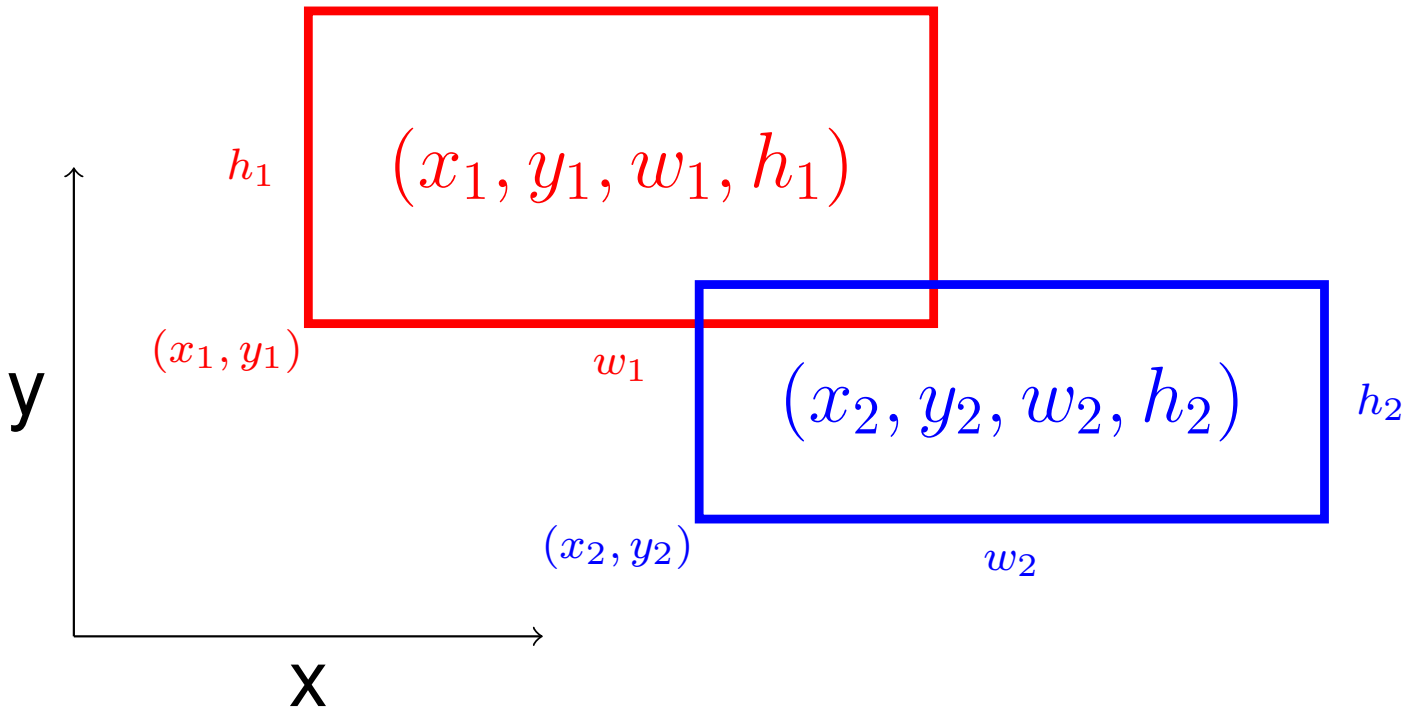


Grobe Lösung

(Include-Direktiven ignoriert)

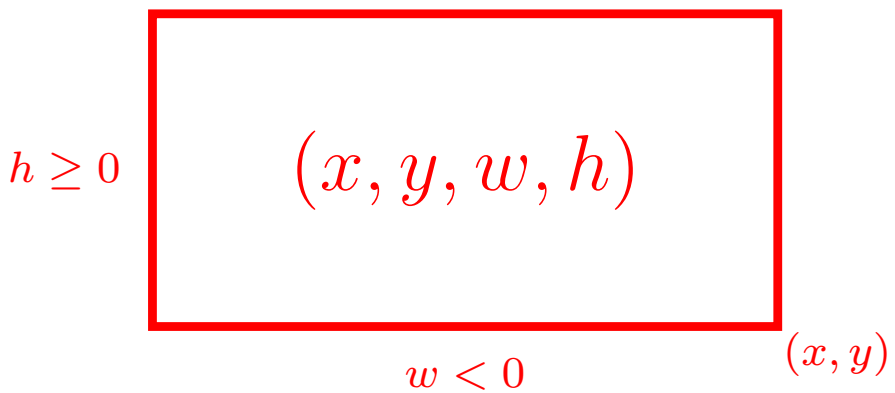
```
int main()
{
    // Eingabe Rechtecke

    // 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 in the format 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()
{
    std::cout << "Enter two rectangles in the format 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;
}
```

Verfeinerung 3: Schnittfunktion...

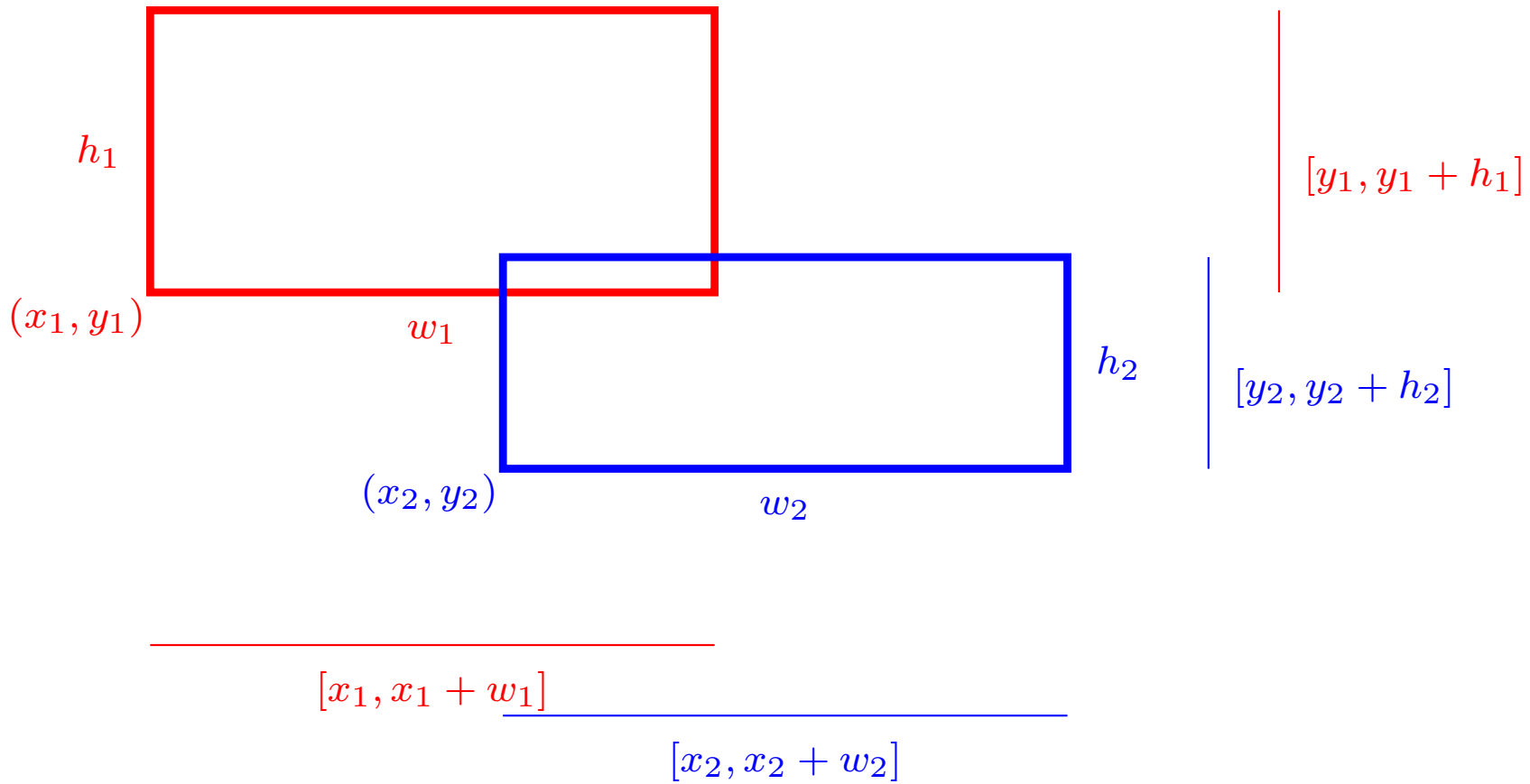
```
bool rectangles_intersect (int x1, int y1, int w1, int h1,
                           int x2, int y2, int w2, int h2)
{
    return false; // todo
}

int main()
{
    std::cout << "Enter two rectangles in the format 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;
}
```

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),(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: [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
}

// 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 5: Min und Max

```
// 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); ✓
}
```

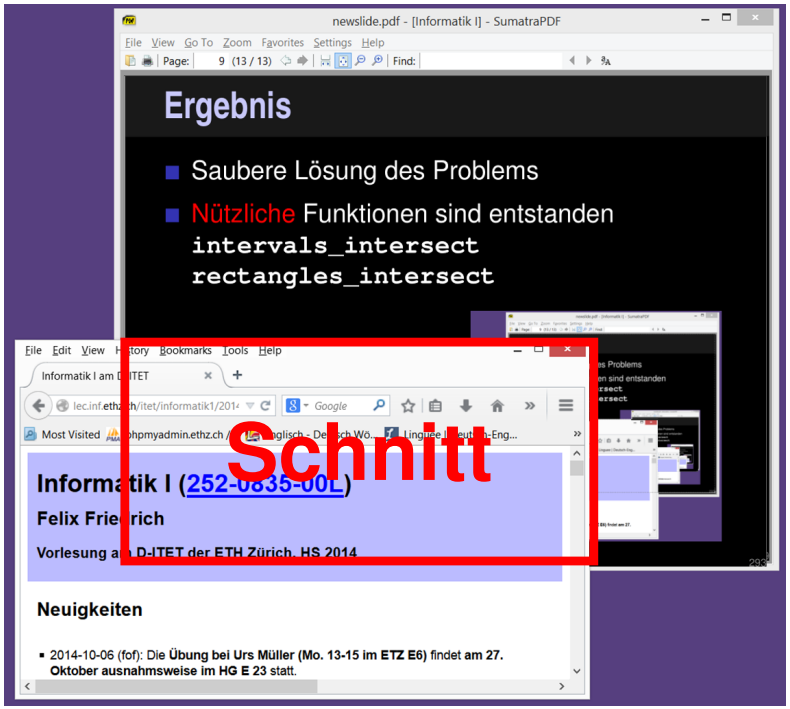
Das haben wir schrittweise erreicht!

```
#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 in the format 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 (const int i) // Gültigkeitsbereich von f ab hier
{
    return i;
}
```

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)
```

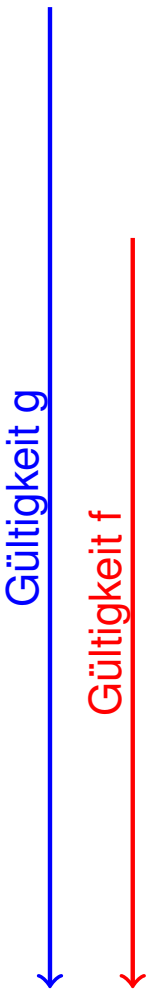
Gültigkeitsbereich: Beispiel


```
#include<iostream>

int f (int i);

int main()
{
    std::cout << f(1); // im Gültigkeitsbereich, ok
    return 0;
}

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

Bei Deklaration: **`const int`** äquivalent zu **`int`**. Grund: Das Verhalten der Funktion „nach aussen" ist unabhängig davon.

Forward Declarations

Funktionen, die sich gegenseitig aufrufen:




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

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

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

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

Separate Datei **`pow.cpp`**

```
// 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 "pow.cpp"

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( 3.0,  4) << "\n";
  std::cout << pow(-2.0,  9) << "\n";

  return 0;
}
```

Dateiangabe relativ zum Arbeitsverzeichnis

Nachteil des Inkludierens

- **`#include`** kopiert die Datei (**`pow.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 **pow.cpp** unabhängig vom Hauptprogramm (z.B. **g++ -c pow.cpp**).

```
// 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;
}
```

Objekt-Code **pow.o** (Maschinensprache)

Level 2: Getrennte Übersetzung...

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

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

// Hauptprogramm callpow3.cpp
#include <iostream>
#include "pow.h"

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( 3.0,  4) << "\n";
  std::cout << pow(-2.0,  9) << "\n";

  return 0;
}
```

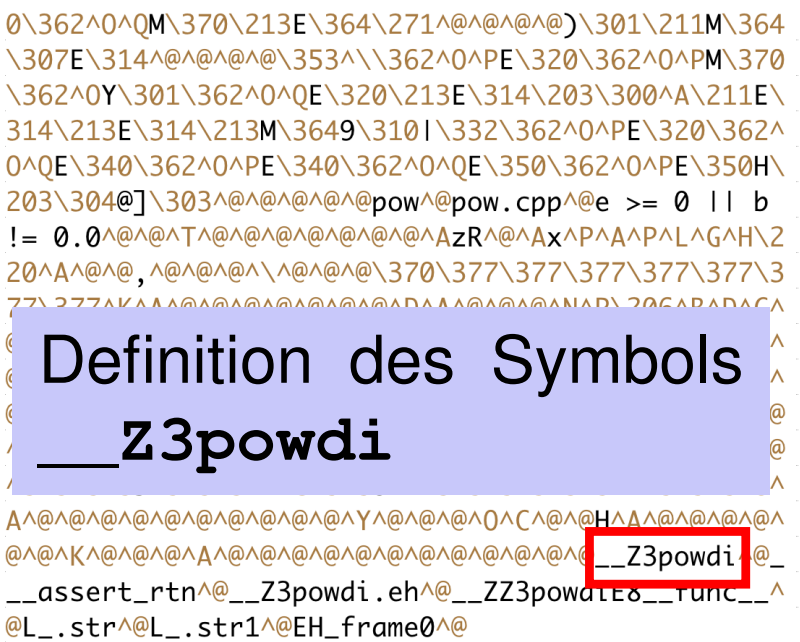
```
@^N^G^@^@\330^A^@^@^@^@^@^@#^A^@^@^N^G^@^@ ^B^@^@
^@^@^@^@7^A^@^@^N^G^@^@P^B^@^@^@^@^@^@k^A^@^@^N^G
^@^@\200^B^@^@^@^@^@^@^A^@^@^@^O^A^@^@^@^@^@^@^@^
@^@^@\256^@^@^@^O^G^@^@\360^A^@^@^@^@^@^@^G^@^@^@
^A^@^@^@^@^@^@^@^@^@^@^@^\^@^@^@^A^@^@^@^@^@^@^@^
@^@^@^@`^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@\225^@^@^@^
A^@^@^@^@^@^@^@^@^@^@^@^Q^@^@^@^A^@^@^@^@^@^@^@^@
^@^@^@'^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@\207^@^@^@^A
^@^@^@^@^@^@^@^@^@^@^@y^@^@^@^A^@^@^@^@^@^@^@^@^@
^@^@^@_main^@__Z3powdi^@__ZSt4cout^@__ZNSolsEd^@_
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_E
S5_PKc^@__ZNSt8ios_base4InitC1Ev^@___dso_handle^@
___cxa_atexit^@__ZNSt8ios_base4InitD1Ev^@_main.eh
^@L_.str^@__GLOBAL__I_main^@__Z41__static_initial
ization_and_destruction_0ii^@__ZStL8__ioinit^@___
tcf_0^@EH_frame0^@__GLOBAL__I_main.eh^@__Z41__sta
tic_initialization_and_destruction_0ii.eh^@___tcf
_0.eh^@^@
```

Objekt-Code **`callpow3.o`** (Maschinensprache)

Das kann doch nicht funktionieren!

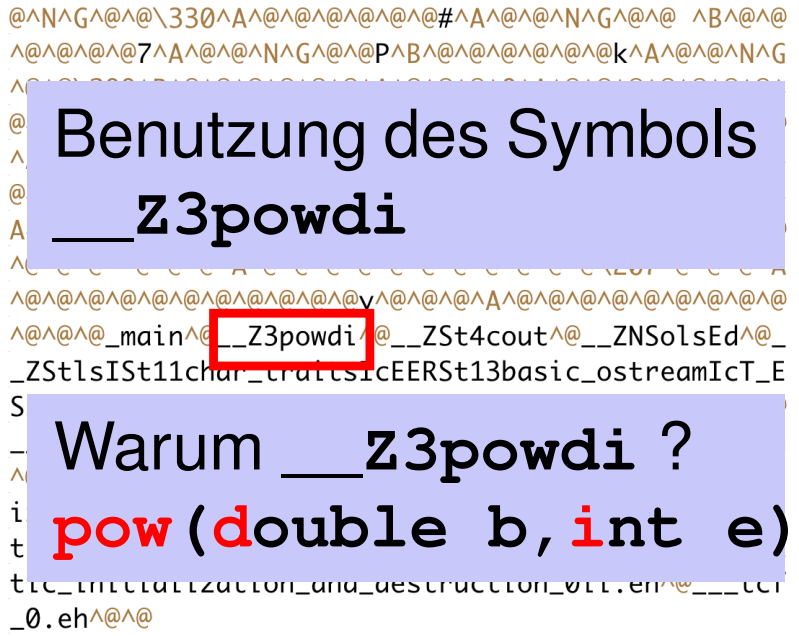
Wie “weiss” **`callpow3.o`**, was **`pow.cpp`** macht?

Erstmal gar nicht, aber...



`pow.o`

(”Maschinensprache”)

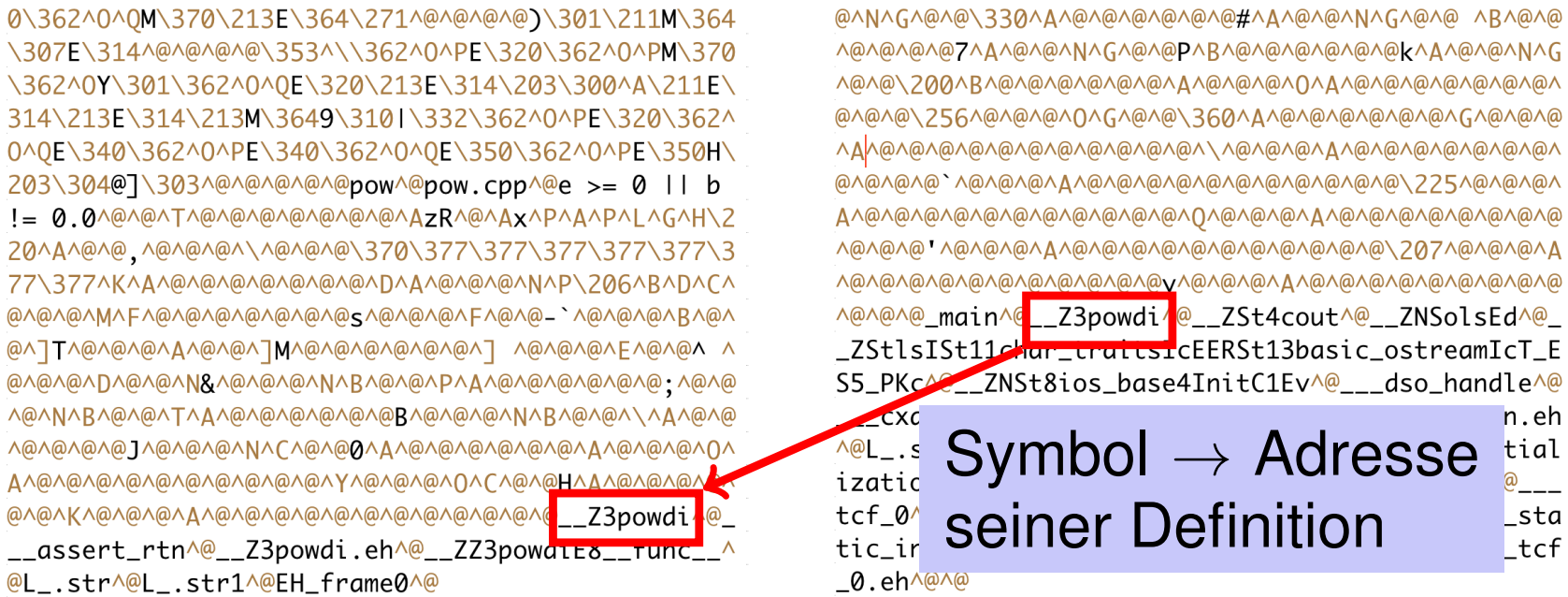


`callpow3.o`

(”Maschinensprache”)

Es funktioniert doch! Der Linker

Zusammensetzung von **pow.o** und **callpow3.o**
(z.B. **g++ pow.o callpow3.o -o callpow3**)



Verfügbarkeit von Quellcode?

Beobachtung

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

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

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

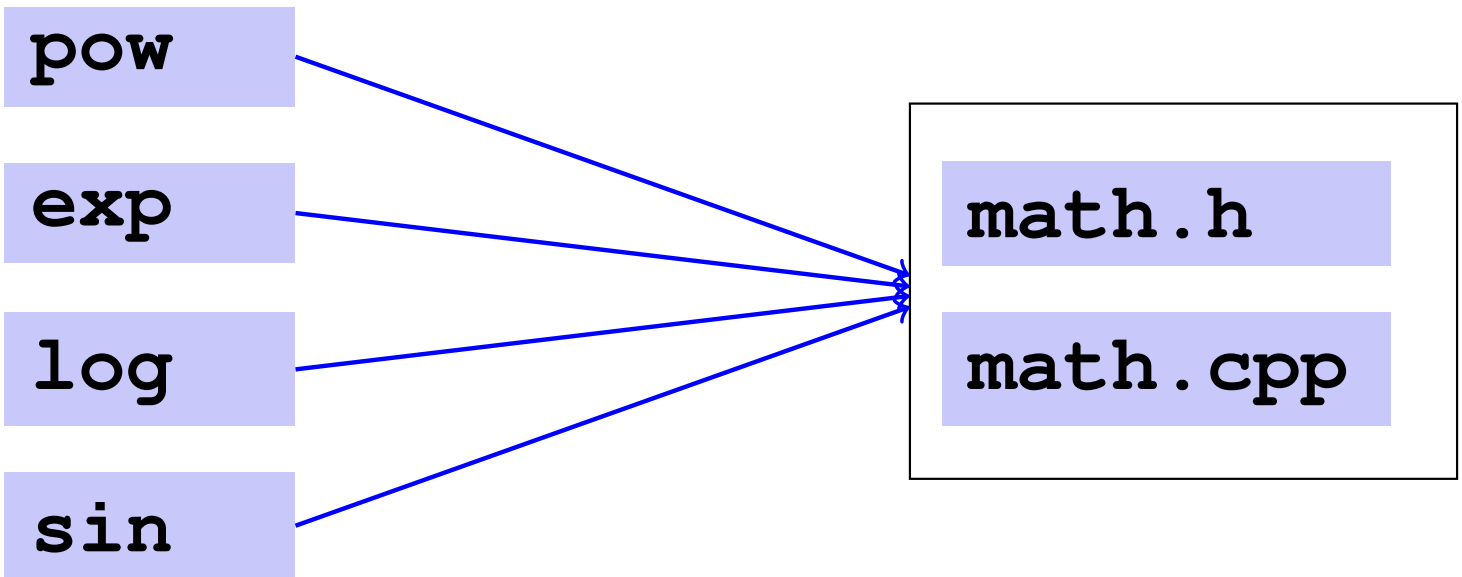
```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e);
```

„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...

...vermeiden Namenskonflikte.

```
// 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);
  ...
}

// Benutzung:
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;
- garantierten 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 $\{2, \ldots, n-1\}$ 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 $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ ein Teiler von n ist.

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

Fall 1: Nach `for`-Anweisung gilt `d <= bound`, d.h. $d \leq \lfloor\sqrt{n}\rfloor$

`n % d == 0`, d echter Teiler, n keine Primzahl.

Fall 2: Nach `for`-Anweisung gilt `d > bound`, d.h. $d > \lfloor\sqrt{n}\rfloor$

Kein Teiler in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$, n ist Primzahl.

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 double bound = 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;
}
```

Garantierte Qualität?

Unser schneller Primzahltest könnte inkorrekt sein, falls z.B.

```
sqrt(121) == 10.9999...
```

```
const double bound = 10.9999...;
unsigned int d;
for (d=2; d <= 10.9... && 121%d != 0; ++d);
```

Fall 2: Nach `for`-Anweisung gilt `d > 10`, d.h. $d > \lfloor$`std::sqrt(121)`$\rfloor$

Kein Teiler in $\{2, \ldots, 10\}$, 121 ist “Primzahl.”

`std::sqrt` hat die Qualität!

- Für `std::sqrt` garantiert der IEEE Standard 754 noch, dass auf den nächsten darstellbaren Wert gerundet wird (wie bei `+`, `-`, `*`, `/`)
- Also: `std::sqrt(121) == 11`
- `prime2.cpp` mit `std::sqrt` ist korrekt!

Auch bei `std::` heisst es aufpassen!

- Für andere mathematische Funktionen gibt der IEEE Standard 754 keine solchen Garantien, sie können also auch weniger genau sein!

```
std::pow (2026 * 2026, 0.25) != std::sqrt (2026)
```

- Gute Programme müssen darauf Rücksicht nehmen und evtl. „Sicherheitsmassnahmen” einbauen.