

Stepwise Refinement

10. Funktionen II

Stepwise Refinement, Gültigkeitsbereich, Bibliotheken, Standardfunktionen

- Einfache *Programmiertechnik* zum Lösen komplexer Probleme

330

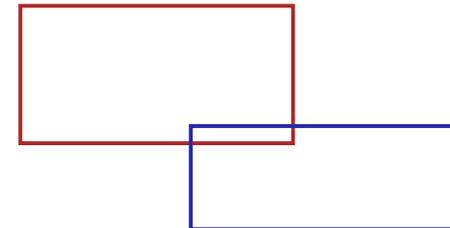
Stepwise Refinement

- 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.

332

Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!



Grobe Lösung

(Include-Direktiven ausgelassen)

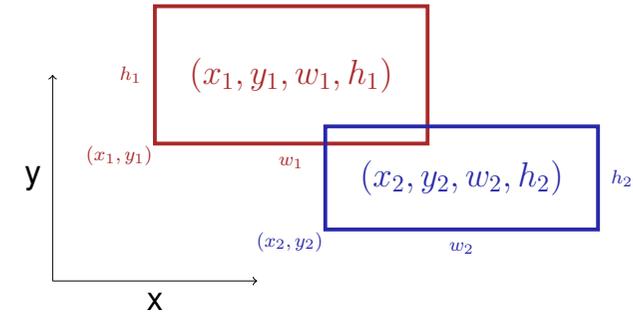
```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe der Loesung

    return 0;
}
```

Verfeinerung 1: Eingabe Rechtecke

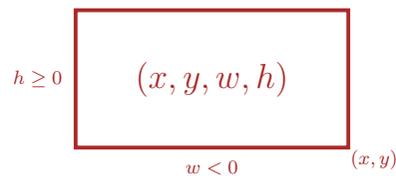


335

336

Verfeinerung 1: Eingabe Rechtecke

Breite w und/oder Höhe h dürfen negativ sein!



337

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;
}
```

338

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;
}
```

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
}

int main() {
    Eingabe Rechtecke ✓

    Schnitt? ✓

    Ausgabe der Loesung ✓

    return 0;
}
```

340

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 ✓

341

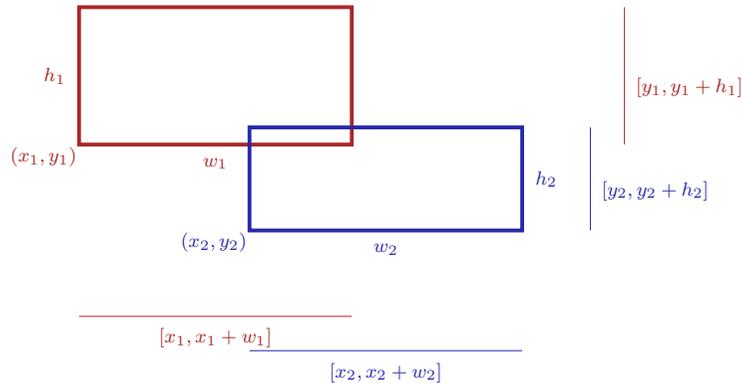
Verfeinerung 3: ...mit PRE und POST!

```
// 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
}
```

342

Verfeinerung 4: Intervallschnitt

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



343

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); ✓
}
```

344

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 ✓

345

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); ✓
}
```

346

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 ✓

347

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); ✓
}
```

348

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);
}

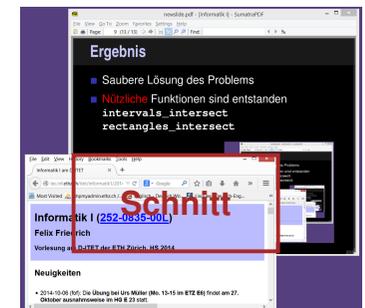
int main ()
{
    std::cout << "Enter two rectangles [x y w h each]\n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;
    bool clash = rectangles_intersect(x1,y1,w1,h1,x2,y2,w2,h2);
    if (clash)
        std::cout << "intersection!\n";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

349

Ergebnis

- Saubere Lösung des Problems
- **Nützliche** Funktionen sind entstanden

intervals_intersect
rectangles_intersect



350

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 ↓

351

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);
```

352

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 ↓

353

...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;
}
```

354

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

355

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.

356

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;
}
```

357

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;
}
```

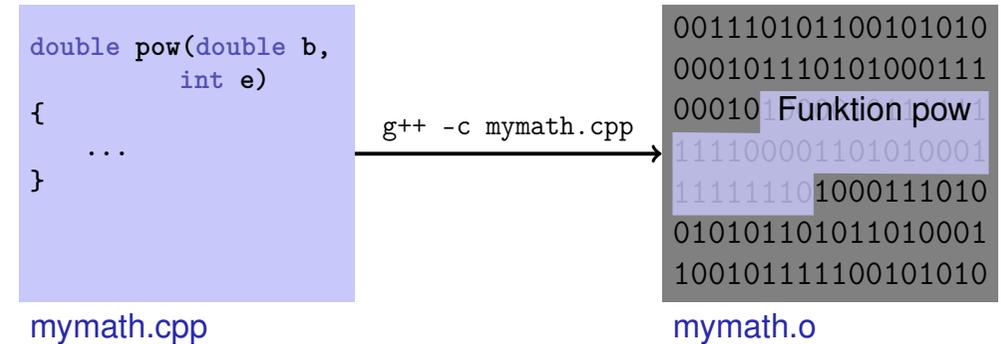
358

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:



359

360

Level 2: Getrennte Übersetzung

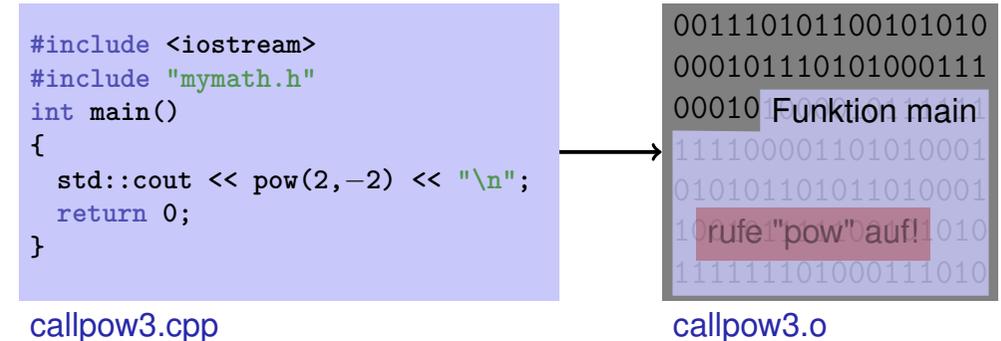
Deklaration aller benötigten Symbole in sog. *Header* Datei.

```
// 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.h` inkludiert wird.



361

362

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! 010
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! 010
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

363

364

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

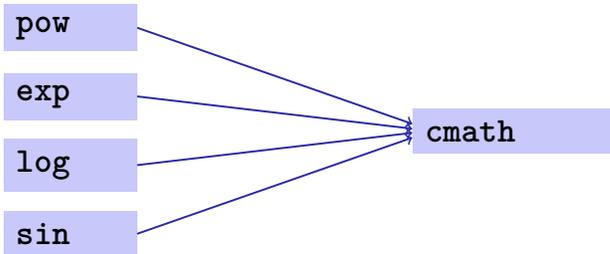
- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte “Hacker”.
- Selbst im kommerziellen Bereich ist „open source“ auf dem Vormarsch.
- Lizenzen erzwingen die Nennung der Quellen und die offene Weiterentwicklung. Beispiel: GPL (GNU General Public License).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

365

366

Bibliotheken

- Logische Gruppierung ähnlicher Funktionen



Namensräume...

```
// cmath
namespace std {

    double pow(double b, int e);

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

367

368

... 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
}
```

369

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.

370

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.

371

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);
```

372

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).
- Andere mathematische Funktionen (`std::pow, ...`) sind in der Praxis fast so genau.

373

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;  
}
```

374

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