

# 10. Funktionen II

---

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

# Vorbedingungen

Vorbedingung (precondition):

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

# 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.

# Nachbedingungen

Nachbedingung (postcondition):

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

Hier nur Wert, kein Effekt.

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

# Vor- und Nachbedingungen

- sollten korrekt sein:

# Vor- und Nachbedingungen

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

# 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  $b \neq 0$

# Fromme Lügen...

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

ist formal inkorrekt

# Fromme Lügen...

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

ist formal inkorrekt:

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

# Fromme Lügen...sind erlaubt.

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

Mathematische Bedingungen als Kompromiss zwischen formaler Korrektheit und lascher Praxis.

# Prüfen von Vorbedingungen...

- Vorbedingungen sind nur Kommentare.

# Prüfen von Vorbedingungen...

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

## ...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;
    ...
}
```

# Nachbedingungen mit Assertions

- Das Ergebnis „komplizierter“ Berechnungen ist oft einfach zu prüfen.

# Nachbedingungen mit Assertions

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

# 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;
}
```

# Stepwise Refinement

## Einfache Programmier- technik zum Lösen komplexer Probleme

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

Education  
P. Wegner  
Editor

## Program Development by Stepwise Refinement

Niklaus Wirth  
Eidgenössische Technische Hochschule  
Zürich, Switzerland

The creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

Key Words and Phrases: education in programming, programming techniques, stepwise program construction  
CR Categories: 1.50, 4.0

### 1. Introduction

Programming is usually taught by examples. Experience shows that the success of a programming course critically depends on the choice of these examples. Unfortunately, they are too often selected with the prime intent to demonstrate what a computer can do. Instead, a main criterion for selection should be their suitability to exhibit certain widely applicable techniques. Furthermore, examples of programs are commonly presented as finished “products” followed by explanations of their purpose and their linguistic details. But active programming consists of the design of new programs, rather than contemplation of old programs. As a consequence of these teaching methods, the student obtains the impression that programming consists mainly of mastering a language (with all the peculiarities and intricacies so abundant in modern PL’s) and relying on one’s intuition to somehow transform ideas into finished programs. Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual development can be nicely demonstrated.

This paper deals with a single example chosen with

these two purposes in mind. Some well-known techniques are briefly demonstrated and motivated (strategy of preselection, stepwise construction of trial solutions, introduction of auxiliary data, recursion), and the program is gradually developed in a sequence of refinement steps.

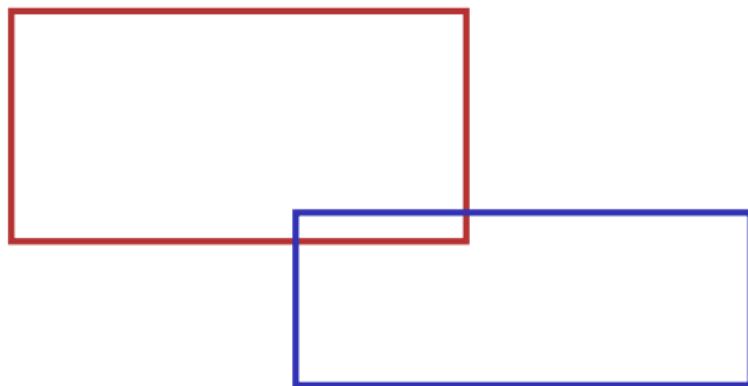
In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and must therefore be guided by the facilities available on that computer or language. The result of the execution of a program is expressed in terms of data, and it may be necessary to introduce further data for communication between the obtained subtasks or instructions. As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel.

Every refinement step implies some design decisions. It is important that these decisions be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions. The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision. Subtrees may be considered as families of solutions with certain common characteristics and structures. The notion of such a tree may be particularly helpful in the situation of changing purpose and environment to which a program may sometime have to be adapted.

A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible. This

# Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!



# Top-Down Ansatz

- Formulierung einer groben Lösung mit Hilfe von
  - Kommentaren
  - fiktiven Funktionen

# Top-Down Ansatz

- Formulierung einer groben Lösung mit Hilfe von
  - Kommentaren
  - fiktiven Funktionen
- Wiederholte Verfeinerung:
  - Kommentare  $\longrightarrow$  Programmtext
  - fiktive Funktionen  $\longrightarrow$  Funktionsdefinitionen

# Grobe Lösung

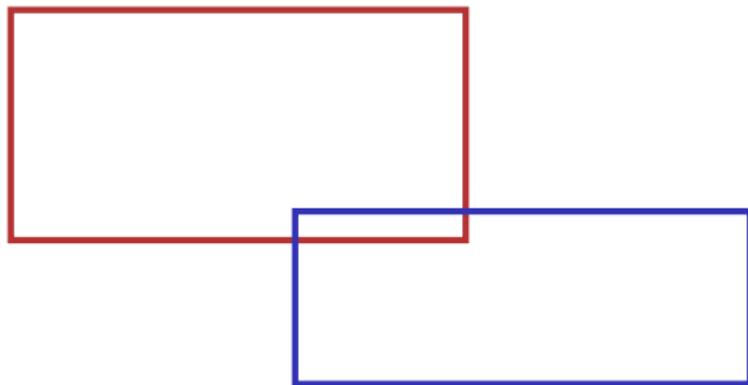
```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

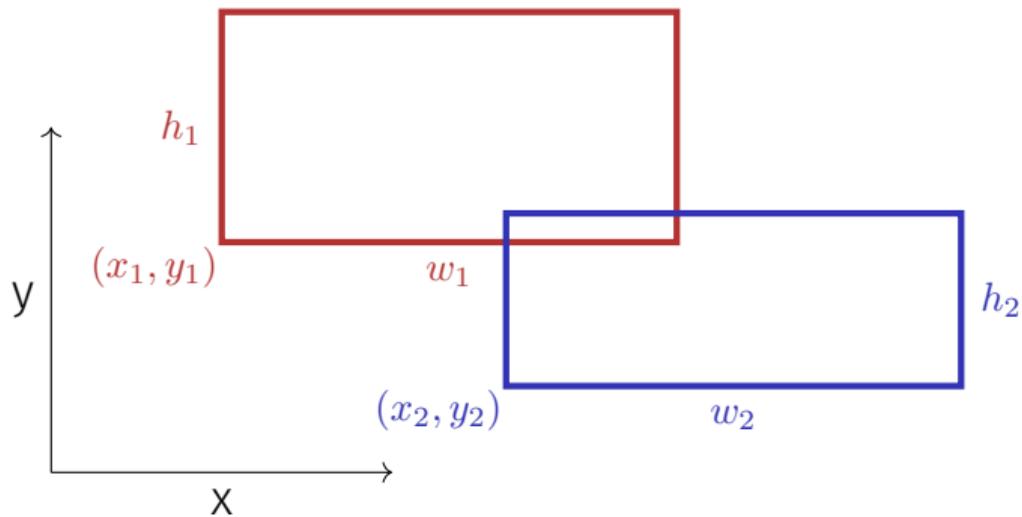
    // Ausgabe der Loesung

    return 0;
}
```

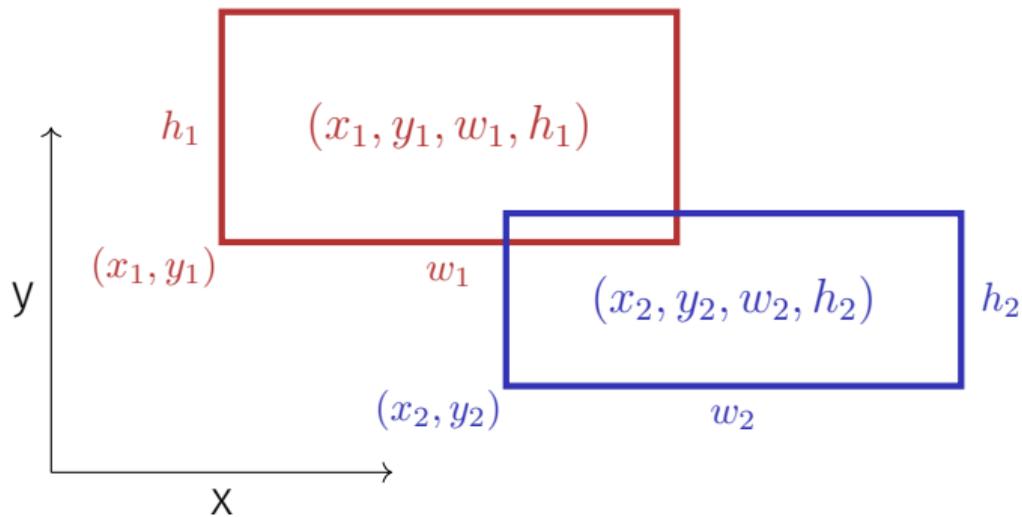
## Verfeinerung 1: Eingabe Rechtecke



# Verfeinerung 1: Eingabe Rechtecke

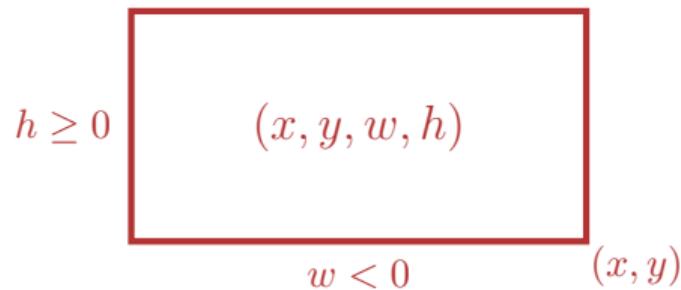


# Verfeinerung 1: Eingabe Rechtecke



# 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 Rectecke ✓

    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 Rechtecke ✓
```

```
    Schnitt? ✓
```

```
    Ausgabe der Loesung ✓
```

```
    return 0;
```

```
}
```

## Verfeinerung 3: Schnittfunktion...

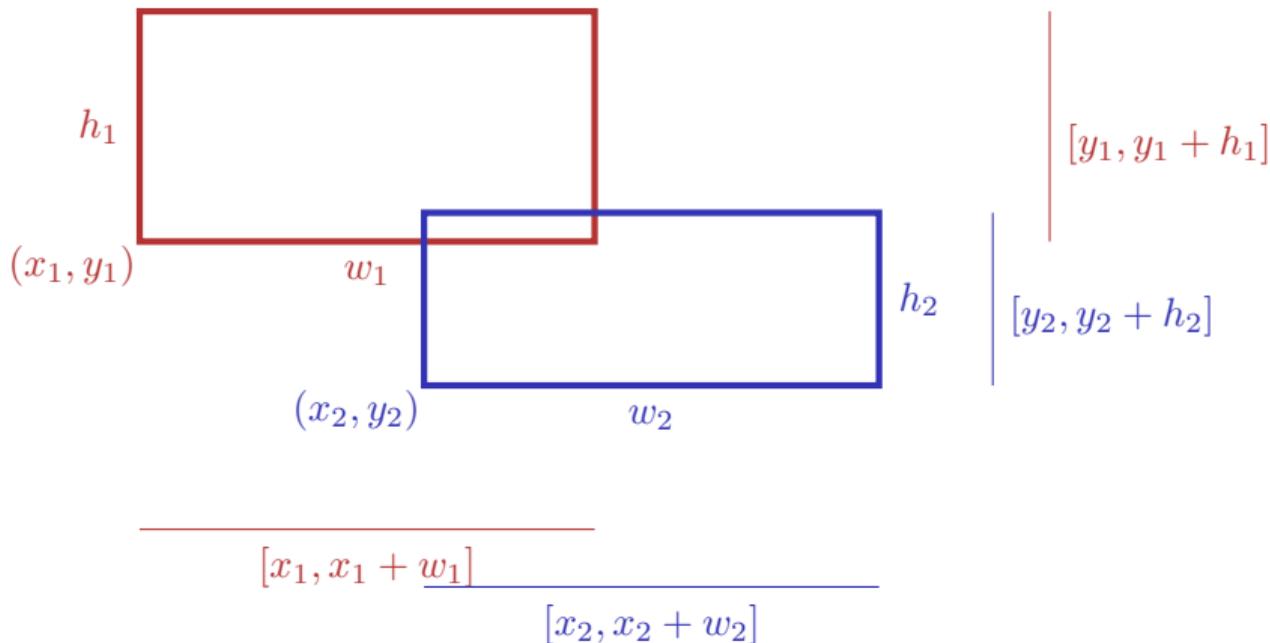
```
bool rectangles_intersect(int x1, int y1, int w1, int h1,  
                          int x2, int y2, int w2, int h2)  
{  
    return false; // todo  
}
```

Funktion main ✓

```
// 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: Intervallschnitt

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: (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

```
// 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;
}
```

```
// 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 ✓

# 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);
}
```

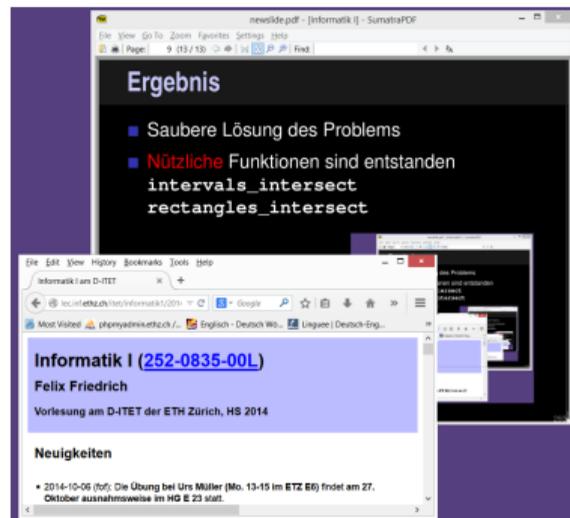
```
int main ()
{
    std::cout << "Enter two rectangles [x y w h each]\n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;
    bool clash = rectangles_intersect(x1,y1,w1,h1,x2,y2,w2,h2);
    if (clash)
        std::cout << "intersection!\n";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

# Ergebnis

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

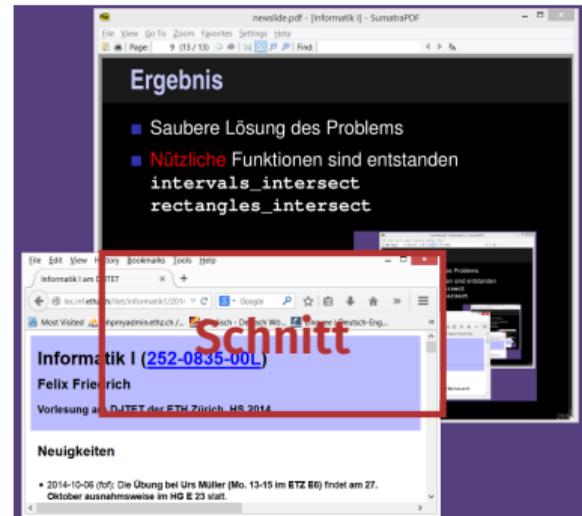
# Ergebnis

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



# 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

# Gültigkeitsbereich einer Funktion

- ist der Teil des Programmes, in dem die Funktion aufgerufen werden kann

Erweiterung durch **Deklaration** einer Funktion: wie Definition aber ohne {...}.

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

# So geht's also nicht...

```
#include <iostream>
```

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

Gültigkeit f

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



## ...aber so!

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

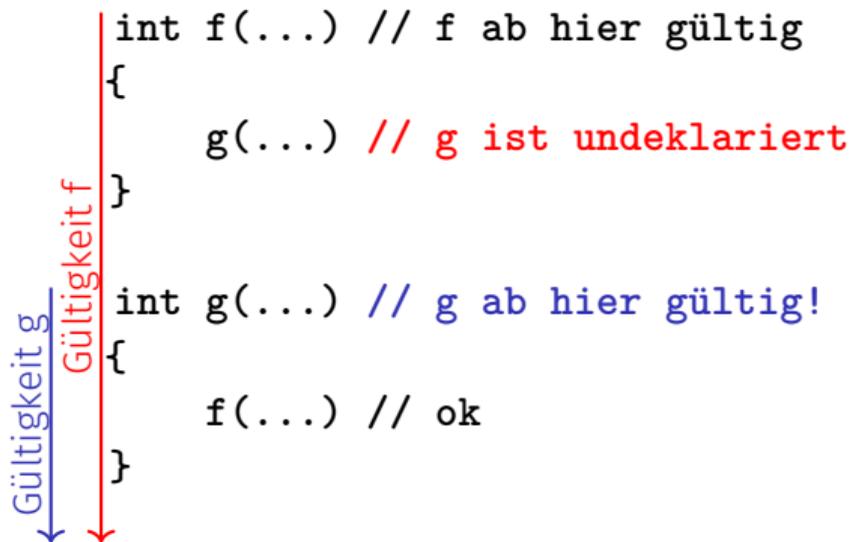
int main()
{
    std::cout << f(1);
    return 0;
}

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

# Forward Declarations, wozu?

Funktionen, die sich gegenseitig aufrufen:

```
int f(...) // f ab hier gültig
{
    g(...) // g ist undeklariert
}
int g(...) // g ab hier gültig!
{
    f(...) // ok
}
```



# Forward Declarations, wozu?

Funktionen, die sich gegenseitig aufrufen:

```
int g(...); // g ab hier gültig

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

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

# Wiederverwendbarkeit

- Funktionen wie `rectangles_intersect` und `pow` sind in vielen Programmen nützlich.

# Wiederverwendbarkeit

- Funktionen wie `rectangles_intersect` und `pow` sind in vielen Programmen nützlich.
- „Lösung:“ Funktion einfach ins Hauptprogramm hineinkopieren, wenn wir sie brauchen!

# 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: Auslagern der Funktion

```
double pow(double b, int e); in  
separater Datei mymath.cpp
```

# Level 1: Inkludieren der Funktion

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

```
#include <iostream>
#include "mymath.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(-2.0, 9) << "\n";

    return 0;
}
```

# Level 1: Inkludieren der Funktion

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

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

# Nachteil des Inkludierens

- `#include` kopiert die Datei (`mymath.cpp`) in das Hauptprogramm (`callpow2.cpp`).

# Nachteil des Inkludierens

- `#include` kopiert die Datei (`mymath.cpp`) in das Hauptprogramm (`callpow2.cpp`).
- Der Compiler muss die Funktionsdefinition für jedes Programm neu übersetzen.



A terminal window titled "Terminal — tcsh8.5 — 80x24" showing the output of the command `sudo port install amarok`. The output lists the steps for installing the `pkgconfig` and `openssl` ports. A cartoon illustration of a person with blonde hair, wearing a blue shirt and brown pants, is shown sleeping in a green office chair. The word "ZZZZ" is written above the person's head.

```
Shabdas-iMac:~ admin$ sudo port install amarok
--> Fetching pkgconfig
--> Attempting to fetch pkg-config-0.25.tar.gz from http://aarnet.au.distfiles
      .macports.org/pub/macports/mpdistfiles/pkgconfig
--> Verifying checksum(s) for pkgconfig
--> Extracting pkgconfig
--> Applying patches to pkgconfig
--> Configuring pkgconfig
--> Building pkgconfig
--> Staging pkgconfig into destroot
--> Installing pkgconfig @0.25_1
--> Deactivating pkgconfig @0.23_1
--> Activating pkgconfig @0.25_1
--> Cleaning pkgconfig
--> Computing dependencies for openssl
--> Fetching openssl
--> Attempting to fetch openssl-1.0.0c.tar.gz from http://aarnet.au.distfiles.
      macports.org/pub/macports/mpdistfiles/openssl
--> Verifying checksum(s) for openssl
--> Extracting openssl
--> Applying patches to openssl
--> Configuring openssl
--> Building openssl
--> Staging openssl into destroot
```

## Level 2: Getrennte Übersetzung

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

mymath.cpp

g++ -c mymath.cpp

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

mymath.o

## Level 2: Getrennte Übersetzung

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

mymath.h

# Level 2: Getrennte Übersetzung

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

callpow3.cpp



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

callpow3.o

# Der Linker vereint...

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
10 rufe pow auf! 1010
111111101000111010
```

callpow3.o

# ... was zusammengehört

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
10 rufe pow auf! 1010
111111101000111010
```

callpow3.o

=

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
10 rufe addr auf! 1010
111111101000111010
```

Ausführbare Datei callpow3

# Verfügbarkeit von Quellcode?

## Beobachtung

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

# 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.

# 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 *einzigsten* lesbaren Informationen.

# Open-Source-Software

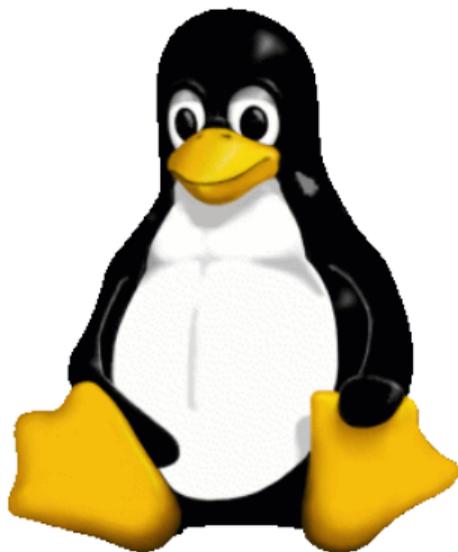
- Alle Quellcodes sind verfügbar.

# Open-Source-Software

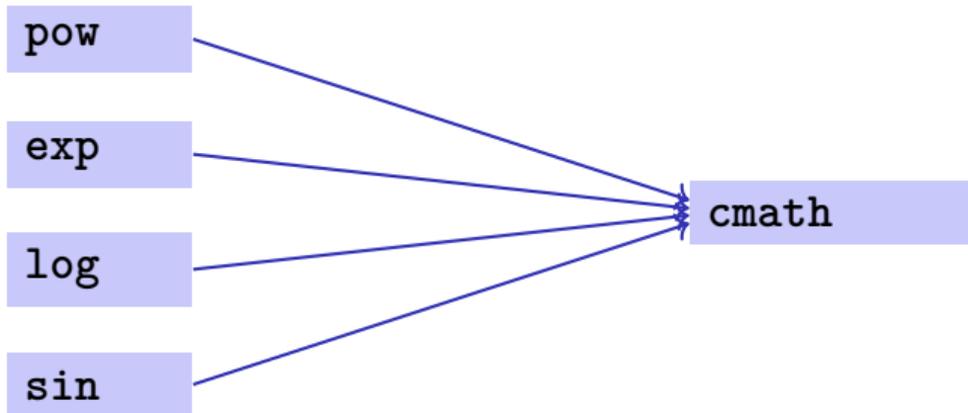
- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte „Hacker“.

# Open-Source-Software

- Alle Quellcodes sind verfügbar.



- Logische Gruppierung ähnlicher Funktionen



# Namensräume...

```
// cmath
namespace std {

    double pow(double b, int e);

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

## ...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
}
```

# Funktionen aus der Standardbibliothek

- vermeiden die Neuerfindung des Rades (wie bei `std::pow`);
- führen auf einfache Weise zu interessanten und effizienten Programmen;

# 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);
```

# 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);
```

# 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).

```
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);  
}
```

```
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);
}
```

```
// 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!

# 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

# 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&`)