10. Functions II

Pre- and Postconditions Stepwise Refinement, Scope, Libraries and Standard Functions

Pre- and Postconditions

- characterize (as complete as possible) what a function does
- document the function for users and programmers (we or other people)
- make programs more readable: we do not have to understand *how* the function works
- are ignored by the compiler
- Pre and postconditions render statements about the correctness of a program possible – provided they are correct.

Preconditions

precondition:

- what is required to hold when the function is called?
- defines the *domain* of the function

0^e is undefined for $e < 0$

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

Postconditions

postcondition:

- What is guaranteed to hold after the function call?
- Specifies *value* and *effect* of the function call.

Here only value, no effect.

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

Pre- and Postconditions

- should be correct:
- *if* the precondition holds when the function is called *then* also the postcondition holds after the call.

Funktion **`pow`**: works for all numbers $b \neq 0$

Pre- and Postconditions

- We do not make a statement about what happens if the precondition does not hold.
- C++-standard-slang: “Undefined behavior”.

Function **`pow`**: division by 0

Pre- and Postconditions

- pre-condition should be as **weak** as possible (largest possible domain)
- post-condition should be as **strong** as possible (most detailed information)

White Lies…

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

is formally incorrect:

- Overflow if e or b are too large
- b^e potentially not representable as a double (holes in the value range!)

White Lies are Allowed

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

The exact pre- and postconditions are platform-dependent and often complicated. We abstract away and provide the mathematical conditions. $\Rightarrow$ compromise between formal correctness and lax practice.

Checking Preconditions...

- Preconditions are only comments.
- How can we ensure that they hold when the function is called?

…with 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;
    ...
}
```

Postconditions with Asserts

- The result of “complex” computations is often easy to check.
- Then the use of asserts for the postcondition is worthwhile.

```
// 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;
}
```

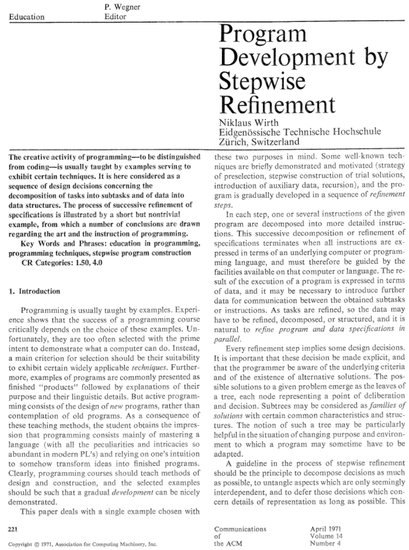
Exceptions

- Assertions are a rough tool; if an assertions fails, the program is halted in a unrecoverable way.
- C++provides more elegant means (exceptions) in order to deal with such failures depending on the situation and potentially without halting the program
- Failsafe programs should only halt in emergency situations and therefore should work with exceptions. For this course, however, this goes too far.

Stepwise Refinement

A simple *technique* to solve complex problems

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 decision 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

221

Copyright © 1971, Association for Computing Machinery, Inc.

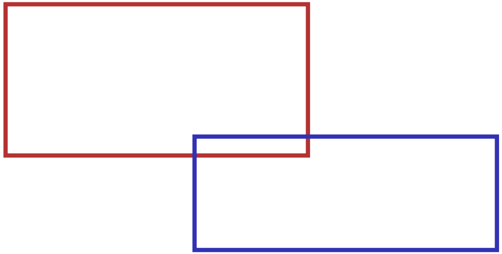
Communications of the ACM April 1971 Volume 14 Number 4

Stepwise Refinement

- Solve the problem step by step. Start with a coarse solution on a high level of abstraction (only comments and abstract function calls)
- At each step, comments are replaced by program text, and functions are implemented (using the same principle again)
- The refinement also refers to the development of data representation (more about this later).
- If the refinement is realized as far as possible by functions, then partial solutions emerge that might be used for other problems.
- Stepwise refinement supports (but does not replace) the structural understanding of a problem.

Example Problem

Find out if two rectangles intersect!

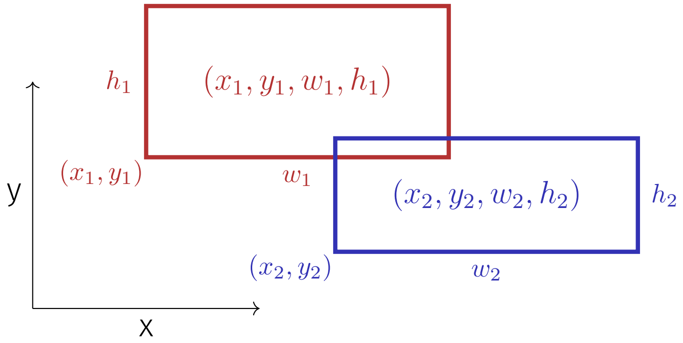


Coarse Solution

(include directives omitted)

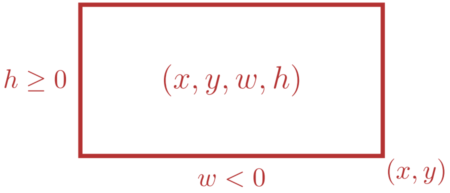
```
int main()
{
    // input rectangles

    // intersection?

    // output solution

    return 0;
}
```

Refinement 1: Input Rectangles



Refinement 1: Input Rectangles

Width w and height h may be negative.



Refinement 1: Input Rectangles

```
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;

    // intersection?

    // output solution

    return 0;
}
```

Refinement 2: Intersection? and Output

```
int main()
{
    input rectangles ✓

    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;
}
```

Refinement 3: Intersection Function...

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

int main() {
    input rectangles ✓

    intersection? ✓

    output solution ✓

    return 0;
}
```

Refinement 3: Intersection Function...

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

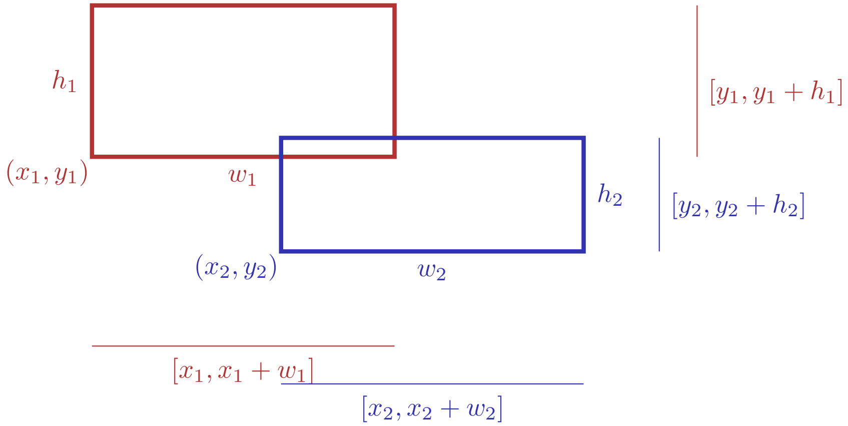
Function main ✓

Refinement 3: ...with PRE and 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
}
```

Refinement 4: Interval Intersection

Two rectangles intersect if and only if their x and y-intervals intersect.



Refinement 4: Interval Intersections

```
// 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); ✓
}
```

Refinement 4: Interval Intersections

```
// 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
}
```

Function rectangles_intersect ✓

Function main ✓

Refinement 5: Min and 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); ✓
}
```

Refinement 5: Min and 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;
}
```

already exists in the standard library

`Function intervals_intersect` ✓

`Function rectangles_intersect` ✓

`Function main` ✓

Back to Intervals

```
// 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); ✓
}
```

Look what we have achieved step by step!

```
#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);
}
```

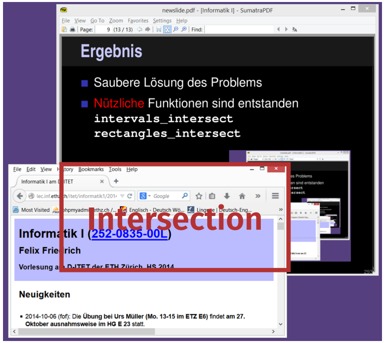
```
int main ()
{
  std::cout << "Enter two rectangles [x y w h each]\n";
  int x1, y1, w1, h1;
  std::cin >> x1 >> y1 >> w1 >> h1;
  int x2, y2, w2, h2;
  std::cin >> x2 >> y2 >> w2 >> h2;
  bool clash = rectangles_intersect(x1,y1,w1,h1,x2,y2,w2,h2);
  if (clash)
    std::cout << "intersection!\n";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

Result

- Clean solution of the problem
- Useful functions have been implemented

  `intervals_intersect`

  `rectangles_intersect`


newslide.pdf - [Informatik I] - SumatraPDF
File View Go To Zoom Favorites Settings Help
Page: 9 (13 / 13) Find:

Ergebnis

- Saubere Lösung des Problems
- Nützliche Funktionen sind entstanden

  `intervals_intersect`

  `rectangles_intersect`

File Edit View History Bookmarks Tools Help
Informatik I am D-ITET
Most Visited
Englisch - Deutsch Wö...
Linguee | Deutsch-Eng...
Intersection

Informatik I (252-0835-00L)

Felix Friedrich

Vorlesung am D-ITET der ETH Zürich, HS 2014

Neuigkeiten

- 2014-10-06 (fof): Die **Übung bei Urs Müller (Mo. 13-15 im ETZ E6)** findet **am 27. Oktober ausnahmsweise im HG E 23** statt.

Where can a Function be Used?

```cpp
#include <iostream>

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

int f(int i) // Scope of f starts here
{
    return i;
}
```

Gültigkeit f

Scope of a Function

- is the part of the program where a function can be called
- is defined as the union of all scopes of its declarations (there can be more than one)

declaration of a function: like the definition but without `{...}`.

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

This does not work...

```
#include <iostream>


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

int f(int i) // Scope of f starts here
{
    return i;
}
```

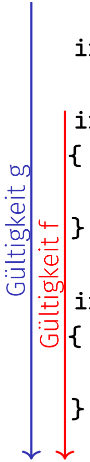
Gültigkeit f

. . . but this works!

```
#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, why?

Functions that mutually call each other:



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

int f(...) // f valid from here
{
    g(...) // ok
}

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

Reusability

- Functions such as **`rectangles_intersect`** and **`pow`** are useful in many programs.
- “Solution”: copy-and-paste the source code
- Main disadvantage: when the function definition needs to be adapted, we have to change **all** programs that make use of the function

Level 1: Outsource the Function

```
// 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: Include the Function

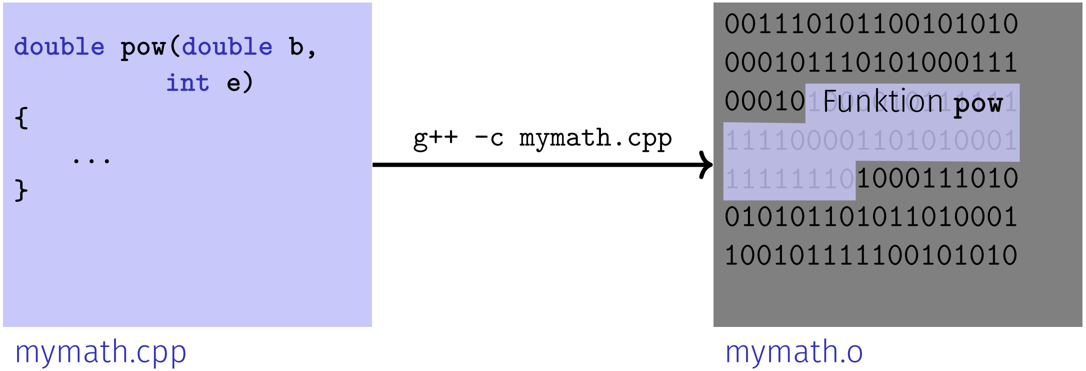
```
// Prog: callpow2.cpp
// Call a function for computing powers.

#include <iostream>
#include "mymath.cpp"  ⟵  file in working directory

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;
}
```

Disadvantage of Including

- **`#include`** copies the file (**`mymath.cpp`**) into the main program (**`callpow2.cpp`**).
- The compiler has to (re)compile the function definition for each program
- This can take long for many and large functions.

Level 2: Separate Compilation

of **`mymath.cpp`** independent of the main program:



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

```
g++ -c mymath.cpp
```

Level 2: Separate Compilation

Declaration of all used symbols in so-called **header** file.

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

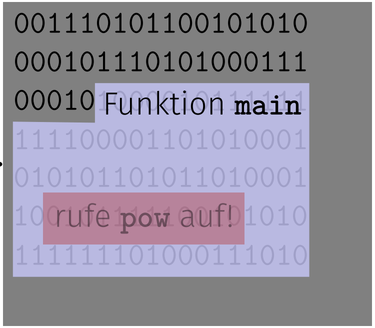
mymath.h

Level 2: Separate Compilation

of the main program, independent of **`mymath.cpp`**, if a *declaration* from **`mymath`** is included.

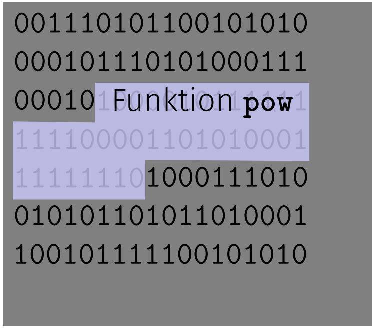
```
#include <iostream>
#include "mymath.h"
int main()
{
  std::cout << pow(2,-2) << "\n";
  return 0;
}
```

callpow3.cpp



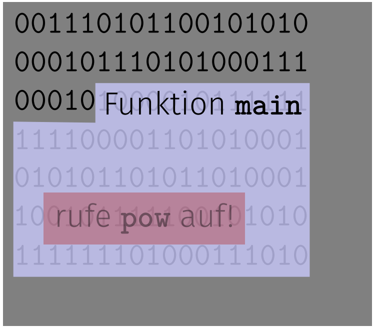
callpow3.o

The linker unites...



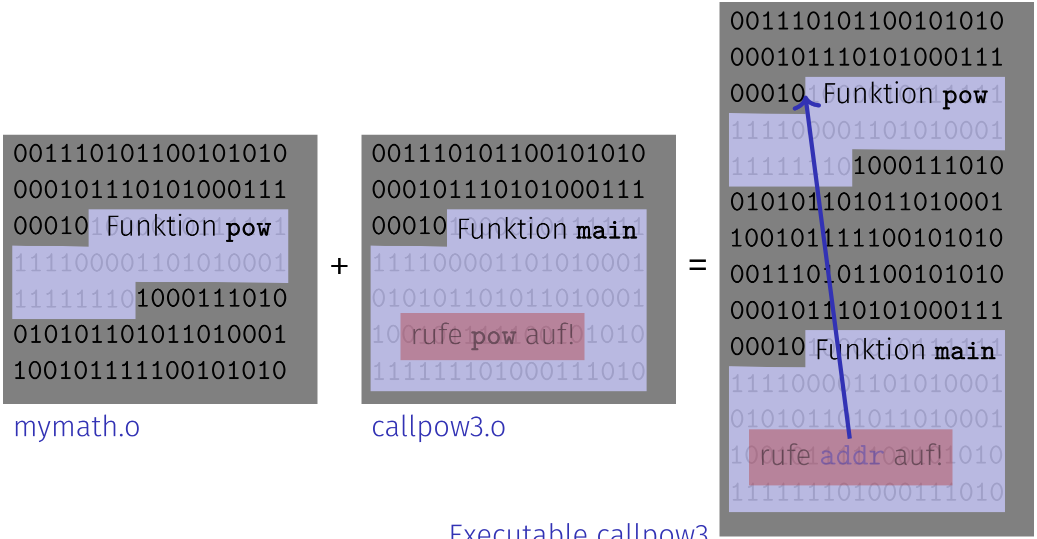
mymath.o

+



callpow3.o

… what belongs together



Availability of Source Code?

Observation

`mymath.cpp` (source code) is not required any more when the **`mymath.o`** (object code) is available.

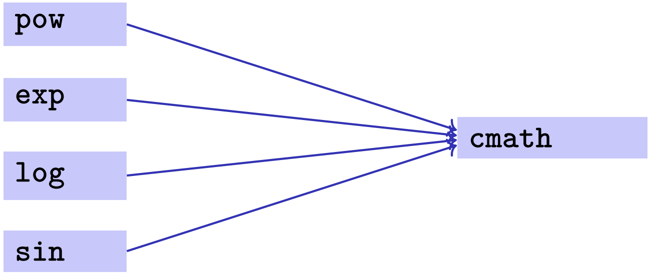
Many vendors of libraries do not provide source code.
Header files then provide the *only* readable informations.

Open-Source Software

- Source code is generally available.
- Only this allows the continued development of code by users and dedicated "hackers".
- Even in commercial domains, open-source software gains ground.
- Certain licenses force naming sources and open development. Example GPL (GNU Genereal Public License)
- Known open-source software: Linux (operating system), Firefox (browser), Thunderbird (email program)...

Libraries

- Logical grouping of similar functions



Name Spaces...

```
// cmath
namespace std {

  double pow(double b, int e);

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

…Avoid Name Conflicts

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

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

Name Spaces / Compilation Units

In C++ the concept of separate compilation is *independent* of the concept of name spaces
In some other languages,e.g. Modula / Oberon (partially also for Java) the compilation unit can define a name space.

Functions from the Standard Library

- help to avoid re-inventing the wheel (such as with **`std::pow`**);
- lead to interesting and efficient programs in a simple way;
- guarantee a quality standard that cannot easily be achieved with code written from scratch.

Example: Prime Number Test with `sqrt`

$n \geq 2$ is a prime number if and only if there is no d in $\{2, \ldots, n-1\}$ dividing n .

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

Prime Number test with `sqrt`

$n \geq 2$ is a prime number if and only if there is no d in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ dividing n .

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

- This works because `std::sqrt` rounds to the next representable `double` number (IEEE Standard 754).

Prime Number test with `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);
```

Functions Should be More Capable!

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!  ☹
}
```

Functions Should be More Capable!

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!
}
```

Sneak Preview: Reference Types

- We can enable functions to change the value of call arguments.
- Not a new concept specific to functions, but rather a new class of types

F