

10. Functions II

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

Preconditions

precondition:

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

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.

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

- 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

- 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 $b \neq 0$

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...

```
// 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
```

Mathematical conditions as a compromise between formal correctness and lax practice

Checking Preconditions...

- Preconditions are only comments.

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.

Postconditions with Asserts

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

Postconditions with Asserts

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

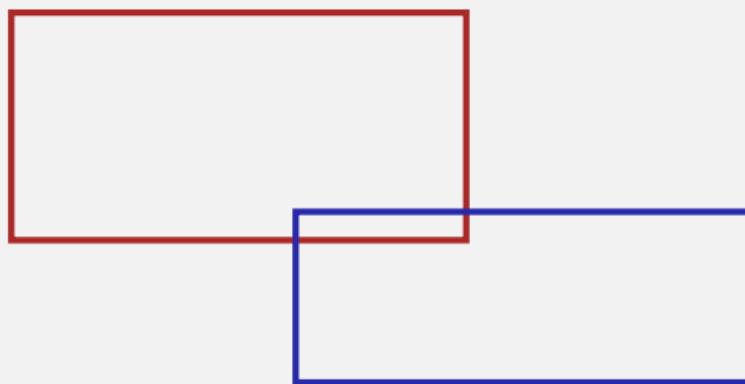
```
// PRE: the discriminant p*p/4 - q is nonnegative
// POST: returns larger root of the polynomial x^2 + p x + q
double root(double p, double q)
{
    assert(p*p/4 >= q); // precondition
    double x1 = - p/2 + sqrt(p*p/4 - q);
    assert>equals(x1*x1+p*x1+q,0)); // postcondition
    return x1;
}
```

Stepwise Refinement

- A simple *technique* to solve complex problems

Example Problem

Find out if two rectangles intersect!



Top-Down Approach

- Formulate a coarse solution using
 - comments
 - fictitious functions
- Repeated refinement:
 - comments → program text
 - fictitious functions → function definitions

Top-Down Approach

- Formulate a coarse solution using
 - comments
 - fictitious functions
- Repeated refinement:
 - comments → program text
 - fictitious functions → function definitions

Coarse Solution

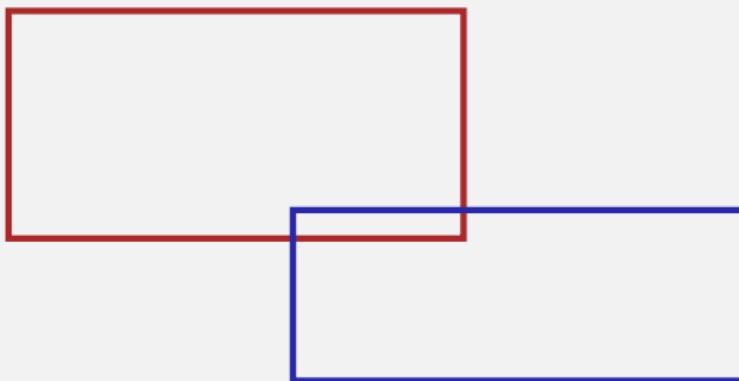
```
int main()
{
    // input rectangles

    // intersection?

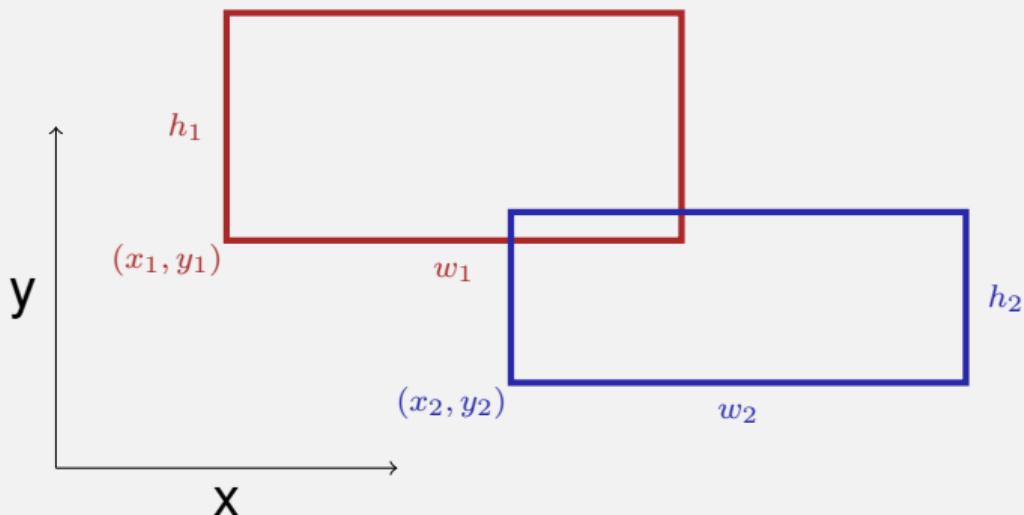
    // output solution

    return 0;
}
```

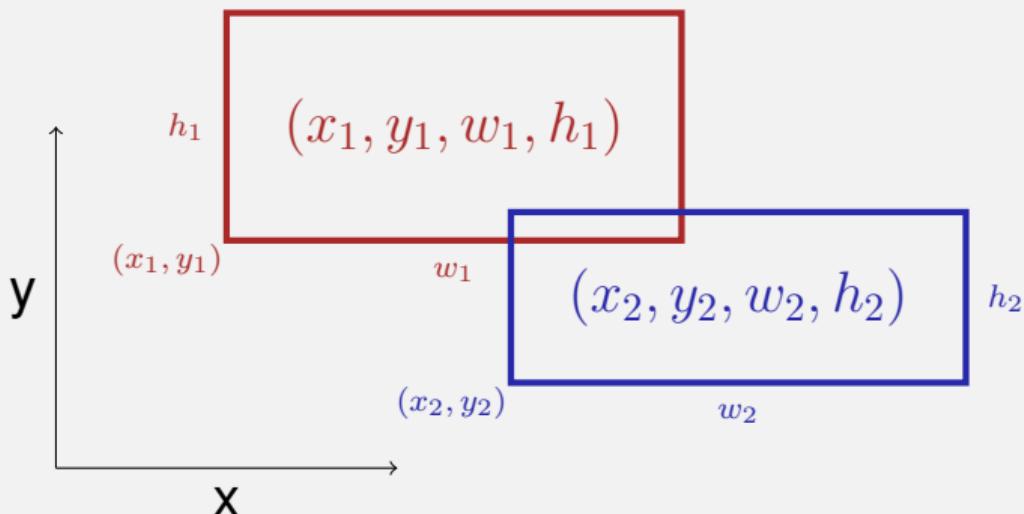
Refinement 1: Input Rectangles



Refinement 1: Input Rectangles

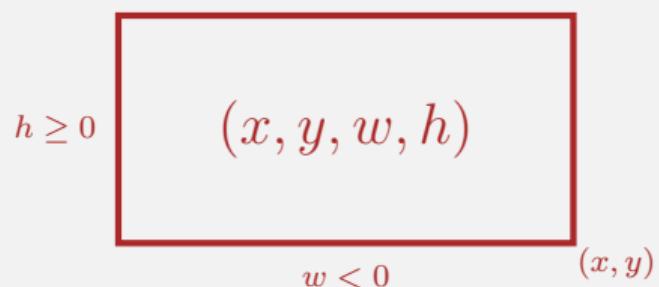


Refinement 1: Input Rectangles



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 ✓

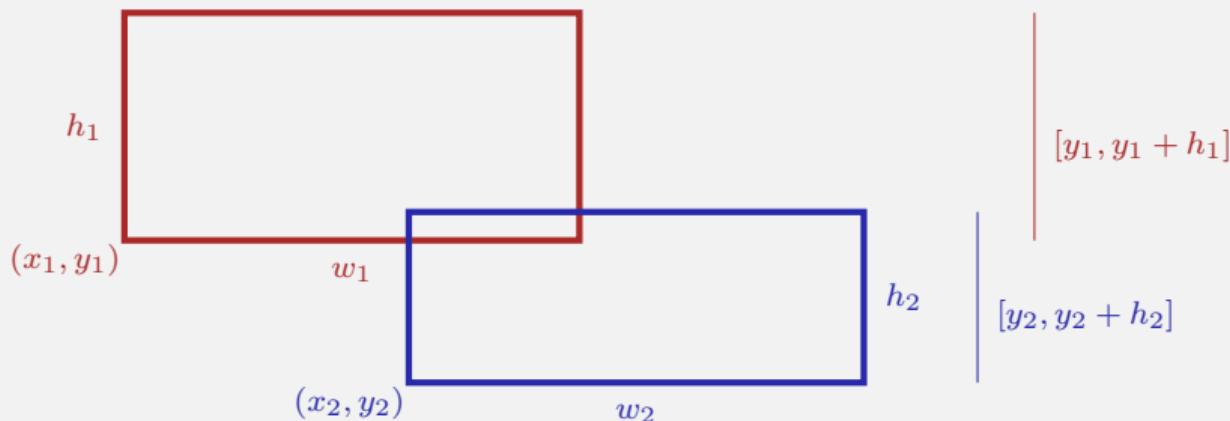
Refinement 3:

...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.



$$[x_1, x_1 + w_1]$$

$$[x_2, x_2 + w_2]$$

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: (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

```
// 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;
}
```

Function intervals_intersect ✓

Function rectangles_intersect ✓

Function main ✓

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;  
}
```

already exists in the standard library

```
// POST: the minimum of x and y is returned  
int min(int x, int y){  
    if (x<y) return x; else return y;  
}
```

Function intervals_intersect ✓

Function rectangles_intersect ✓

Function main ✓

Back to Intervals

```
// 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 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

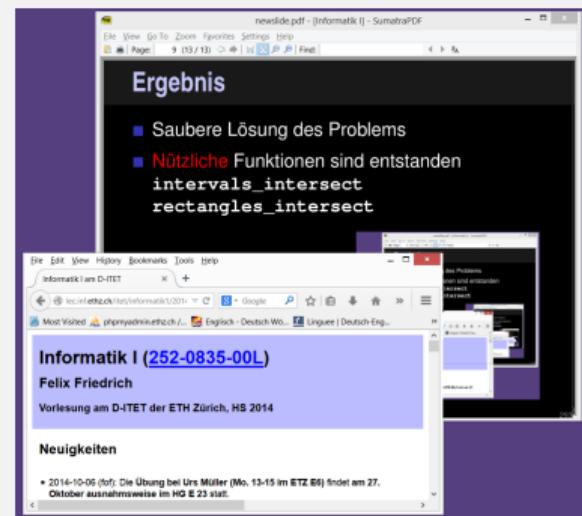
`intervals_intersect`

`rectangles_intersect`

Result

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

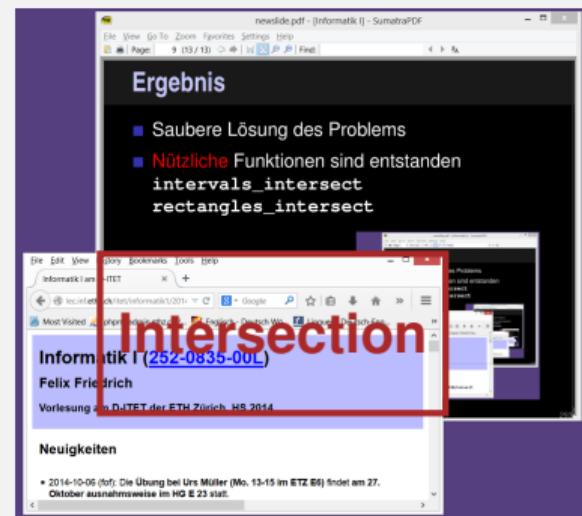
intervals_intersect
rectangles_intersect



Result

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

intervals_intersect
rectangles_intersect



Where can a Function be Used?

```
#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

Scope of a Function

- is the part of the program where a function can be called

Extension by *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:

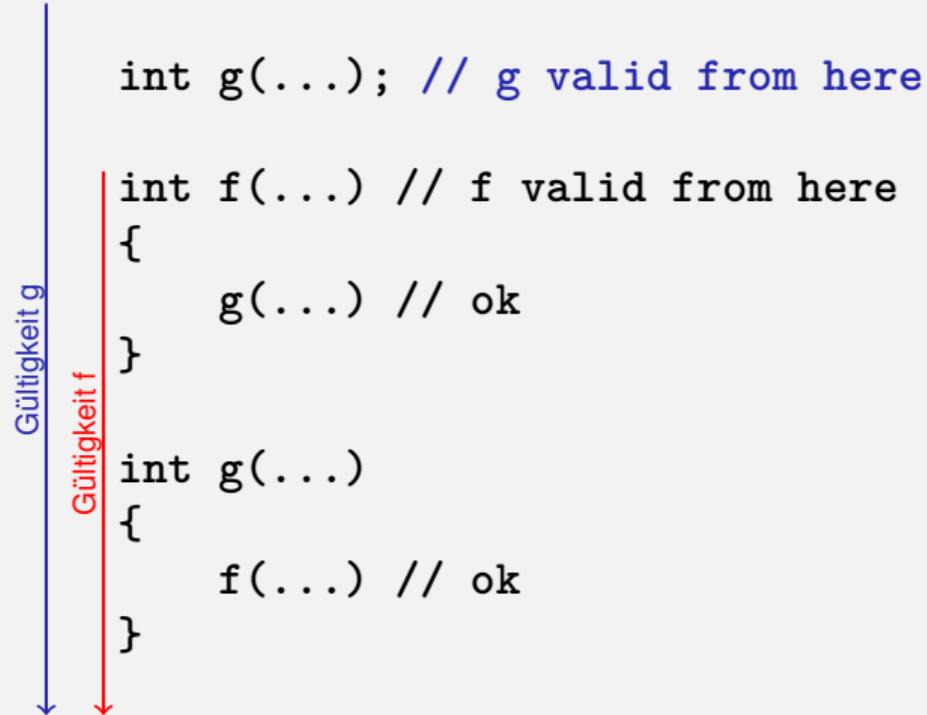
```
int f(...) // f valid from here
{
    g(...) // g undeclared
}

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

The diagram illustrates the scope of the functions f and g. A blue vertical line, labeled "Gültigkeit g" (Scope of g), starts at the declaration of g and extends downwards. A red vertical line, labeled "Gültigkeit f" (Scope of f), starts at the declaration of f and extends upwards. The two lines intersect at the point where g is called from f, indicating that g is valid within the scope of f at that specific call site.

Forward Declarations, why?

Functions that mutually call each other:



Reusability

- Functions such as `rectangles_intersect` and `pow` are useful in many programs.

Reusability

- Functions such as `rectangles_intersect` and `pow` are useful in many programs.
- “Solution”: copy-and-paste the source code

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: Outsource the Function

```
double pow(double b, int e); in  
separate file mymath.cpp
```

Level 1: Include the Function

```
// 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: Include the Function

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

```
#include <iostream>  
#include "mymath.cpp" ← 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`).

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



```
Terminal — tcsh8.5 — 80x24
Shabdaz-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: Separate Compilation

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

g++ -c mymath.cpp

mymath.cpp



The image shows a sequence of binary digits (0s and 1s) representing the assembly or machine code generated from the C++ code. A specific section of the binary code is highlighted in a light blue box, containing the text "Funktion pow".

001110101100101010
000101110101000111
000101Funktion pow1
111100001101010001
111111101000111010
010101101011010001
10010111100101010

mymath.o

Level 2: Separate Compilation

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

mymath.h

Level 2: Separate Compilation

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

callpow3.cpp



The image shows a grid of binary digits (0s and 1s). Overlaid on this grid are several text labels in different colors:

- "Funktion main" is in blue and is positioned above the first few columns of binary digits.
- "rufe pow auf!" is in red and is positioned below the last few columns of binary digits.

callpow3.o

The linker unites...

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



mymath.o

```
001110101100101010  
000101110101000111  
000101 Funktion main  
111100001101010001  
010101101011010001  
100rufe pow auf!01010  
111111101000111010
```

callpow3.o

... what belongs together

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

mymath.o

+

```
001110101100101010  
000101110101000111  
000101 Funktion main  
111100001101010001  
010101101011010001  
100 rufe pow auf! 01010  
111111101000111010
```

callpow3.o

=

```
001110101100101010  
000101110101000111  
000101 Funktion pow  
111100001101010001  
111111101000111010  
1000111010  
010101101011010001  
100101111100101010  
001110101100101010  
000101110101000111  
000101 Funktion main  
111100001101010001  
010101101011010001  
10 rufe 1addr auf! 01010  
111111101000111010
```

Executable callpow3

Availability of Source Code?

Observation

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

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.

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

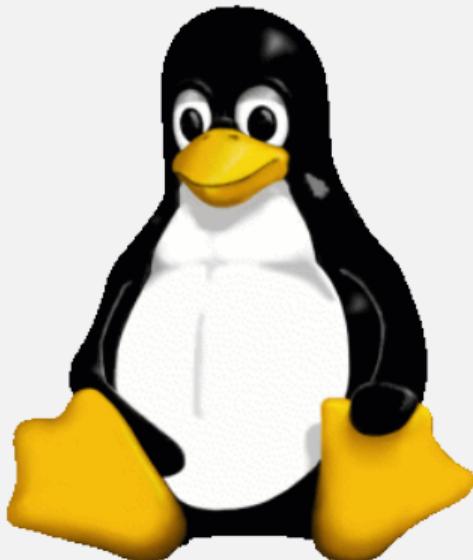
- Source code is generally available.

Open-Source Software

- Source code is generally available.
- Only this allows the continued development of code by users and dedicated “hackers”.

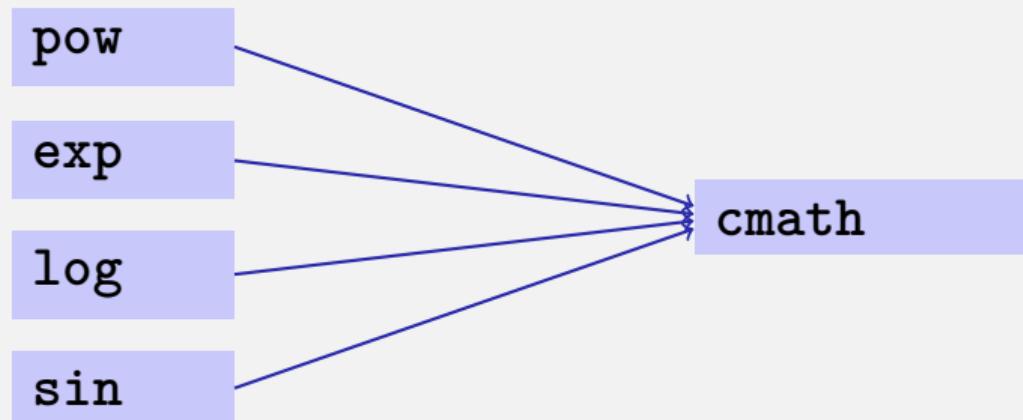
Open-Source Software

- Source code is generally available.
- Only this allows the continued development of code by users and dedicated “hackers”.



Libraries

- Logical grouping of similar functions



Name Spaces...

```
// cmath
namespace std {

    double pow(double b, int e);

    ...
    double exp(double x);
    ...

}
```

...Avoid Name Conflicts

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

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

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;

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 $\{2, \dots, n - 1\}$ 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 $\{2, \dots, \lfloor \sqrt{n} \rfloor\}$ dividing n .

```
unsigned int bound = std::sqrt(n);
unsigned int d;
for (d = 2; d <= bound && 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 $\{2, \dots, \lfloor \sqrt{n} \rfloor\}$ 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).

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);  
}
```

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);
}
```

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.

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

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



Reference types (e.g. int&)