1. Funktionen

Funktionsdefinitionen- und Aufrufe, Vor- und Nachbedingungen, Assertions, Gültigkeitsbereich, Bibliotheken, Standardfunktionen

Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar

[1] Prozedur: anderes Wort für Funktion.

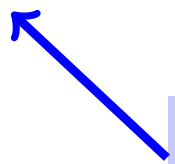
Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar
- strukturieren das Programm: Unterteilung in kleine Teilaufgaben, jede davon durch eine Funktion realisiert

[1] Prozedur: anderes Wort für Funktion.

Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar
- strukturieren das Programm: Unterteilung in kleine Teilaufgaben, jede davon durch eine Funktion realisiert



Prozedurales Programmieren[1]

[1] Prozedur: anderes Wort für Funktion.

Beispiel: Potenzberechnung

```
// 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;
}
```

Beispiel: Potenzberechnung

```
// 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;
}
```

- Aussagekräftige Namen (base, exponent)

Beispiel: Potenzberechnung

```
// 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;
}
```

- Aussagekräftige Namen (base, exponent)
- Vor- und Nachbedingungen – Korrektheit

Programm zur Potenzberechnung

```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>

// 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;
}

int main()
{
  std::cout « pow( 2.0, -2) « "\n"; // outputs 0.25
  std::cout « pow( 1.5,  2) « "\n"; // outputs 2.25
  std::cout « pow( 5.0,  1) « "\n"; // outputs 5
  std::cout « pow( 3.0,  4) « "\n"; // outputs 81
  std::cout « pow(-2.0,  9) « "\n"; // outputs -512

  return 0;
}
```

Vor- und Nachbedingungen

- beschreiben (möglichst vollständig) was die Funktion „macht”
- dokumentieren die Funktion für Benutzer / Programmierer (wir selbst oder andere)
- machen Programme lesbarer: wir müssen nicht verstehen, *wie* die Funktion es macht
- werden vom Compiler ignoriert
- Vor- und Nachbedingungen machen – unter der Annahme ihrer Korrektheit – Aussagen über die Korrektheit eines Programmes möglich.

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

- sind korrekt, wenn immer gilt:
  Wenn die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.

Vor- und Nachbedingungen

- sind korrekt, wenn immer gilt:
  Wenn die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.
- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage
  C++-Standard-Jargon: „Undefined behavior”.

Vor- und Nachbedingungen

- sind korrekt, wenn immer gilt:
  Wenn die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.
- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage
- Vorbedingung sollte so *schwach* wie möglich sein (möglichst grosser Definitionsbereich)
- Nachbedingung sollte so *stark* wie möglich sein (möglichst detaillierte Aussage)

Arithmetische Vor- und Nachbedingungen

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

ist formal inkorrekt:

Arithmetische Vor- und Nachbedingungen

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

ist formal inkorrekt:

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

Arithmetische Vor- und Nachbedingungen

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

Die exakten Vor- und Nachbedingungen sind plattformabhängig und meist sehr kompliziert. Wir abstrahieren und geben die mathematischen Bedingungen an.

Assertions

Vorbedingungen sind nur Kommentare, wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

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

Assertions: Syntax und Semantik

`assert` (*expression*)

- *expression*: Ausdruck, dessen Wert nach **`bool`** konvertierbar ist.
- Semantik:
  - Fall *expression* **`== true`**: kein Effekt
  - Falls *expression* **`== false`**: Programm wird mit entsprechender Fehlermeldung abgebrochen.

Assertions – Empfehlung

- Assertions sind ein wichtiges Werkzeug zur Fehlervermediung während der Programmentwicklung, wenn man sie *konsequent* und *oft* einsetzt.
- Der Laufzeit-Overhead ist vernachlässigbar. Darüber hinaus kann der Compiler angewiesen werden, die Assertions „hinauszukompilieren”, also zu ignorieren.

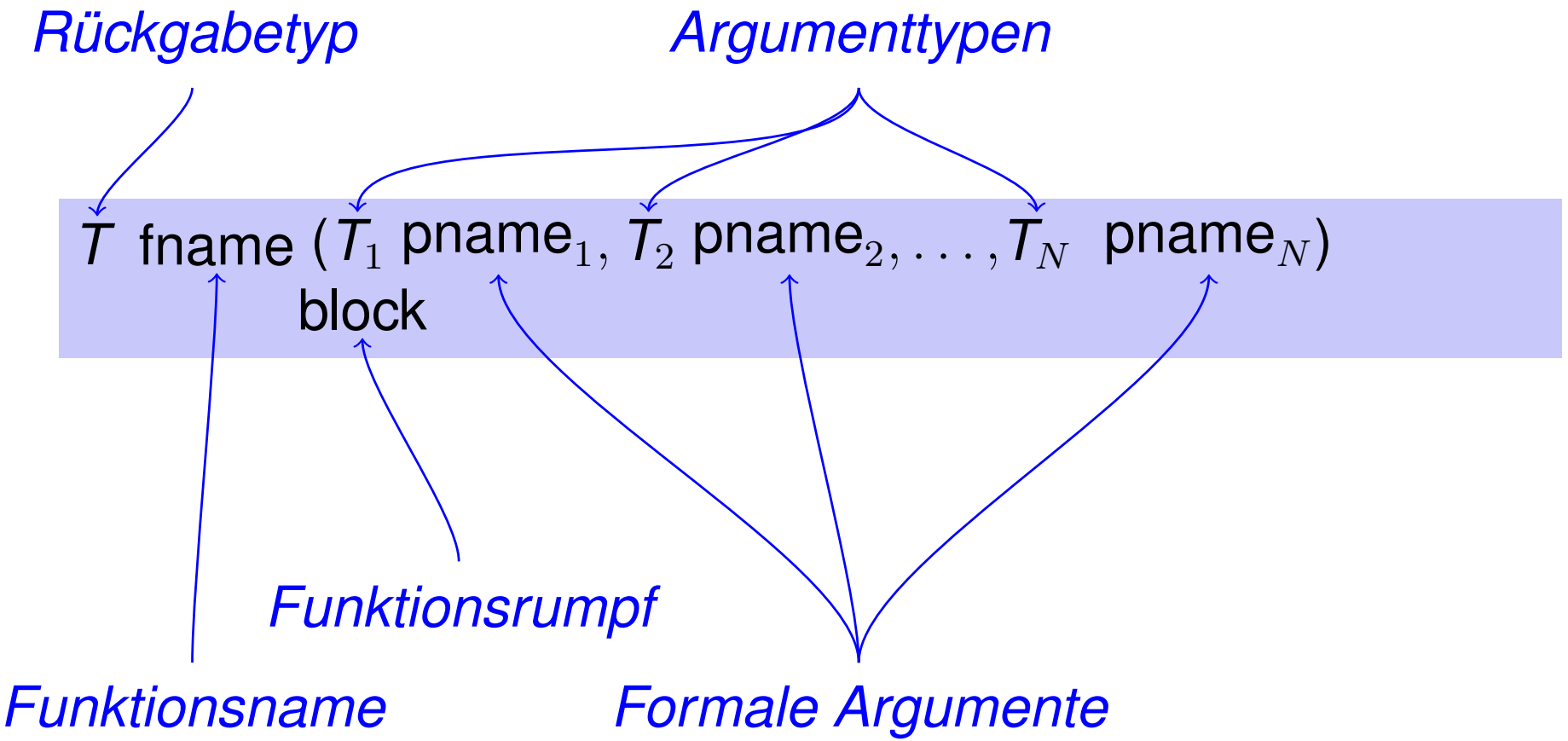
Ausnahmen (Exception Handling)

- Assertions sind ein grober Hammer; falls eine Assertion fehlschlägt, wird das Programm hart abgebrochen.
- C++bietet elegantere Mittel (Exceptions), um auf solche Fehlschläge situationsabhängig (und oft auch ohne Programmabbruch) zu reagieren.
- “Narrensichere” Programmen sollten nur im Notfall abbrechen und deshalb mit Exceptions arbeiten; für diese Vorlesung führt das aber zu weit.

Funktionsdefinitionen

T fname (T_1 pname$_1$, T_2 pname$_2$, $\ldots$, T_N pname$_N$)
block

Funktionsdefinitionen



Funktionsdefinitionen

- dürfen nicht *lokal* auftreten, also nicht in Blocks, nicht in anderen Funktionen und nicht in Kontrollanweisungen
- können im Programm ohne Trennsymbole aufeinander folgen

Funktionsdefinitionen

- dürfen nicht *lokal* auftreten, also nicht in Blocks, nicht in anderen Funktionen und nicht in Kontrollanweisungen
- können im Programm ohne Trennsymbole aufeinander folgen

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

int main ()
{
...
}
```

Funktionsaufrufe

fname (*expression*$_1$, *expression*$_2$, . . . , *expression*$_N$)

- Typen der Aufrufargumente *expression*$_1, \ldots,$ *expression*$_N$ müssen konvertierbar sein in $T_1, \ldots, T_N$.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Typ *T*. Wert und Effekt wie in der Nachbedingung der Funktion *fname* angegeben.

Funktionsaufrufe

fname ($expression_1$, $expression_2$, . . . , $expression_N$)

- Typen der Aufrufargumente $expression_1, \ldots, expression_N$ müssen konvertierbar sein in $T_1, \ldots, T_N$.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Typ T. Wert und Effekt wie in der Nachbedingung der Funktion *fname* angegeben.

Beispiel: `pow(a,b)`

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
- Funktionsaufruf selbst ist R-Wert.

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
- Funktionsaufruf selbst ist R-Wert.

fname: R-Wert $\times$ R-Wert $\times \cdots \times$ R-Wert $\longrightarrow$ R-Wert

Auswertung eines Funktionsaufrufes

- Auswertung der Aufrufargumente
- Initialisierung der formalen Argumente mit den resultierenden Werten
- Ausführung des Funktionsrumpfes: formale Argumente verhalten sich dabei wie lokale Variablen
- Ausführung endet mit
  `return` *expression*;

Auswertung eines Funktionsaufrufes

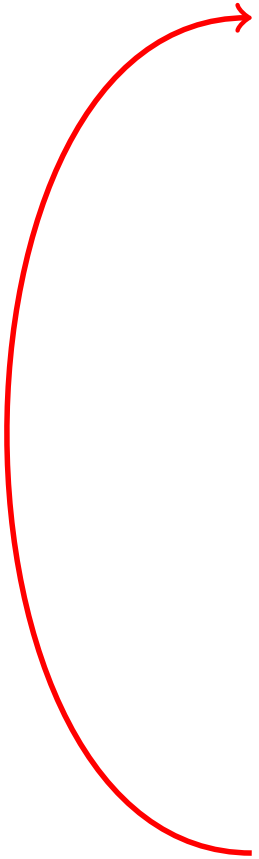
- Auswertung der Aufrufargumente
- Initialisierung der formalen Argumente mit den resultierenden Werten
- Ausführung des Funktionsrumpfes: formale Argumente verhalten sich dabei wie lokale Variablen
- Ausführung endet mit
  `return` *expression*;

Rückgabewert ergibt den Wert des Funktionsaufrufes.

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf



```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)// b = 2.0; e = -2
{
    assert (e >= 0 || b != 0);
    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;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)// b = 2.0; e = -2
{
    assert (e >= 0 || b != 0);  // b = 2.0; e = -2
    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;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    double result = 1.0;  // 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;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;   // b = 0.5;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result *= b;
    return result;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;        // e = 2
    }
    for (int i = 0; i < e ; ++i)
        result *= b;
    return result;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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) // i = 0
        result *= b;
    return result;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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) // i = 0
        result *= b; // result = 0.5
    return result;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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) // i = 1
        result *= b;
    return result;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

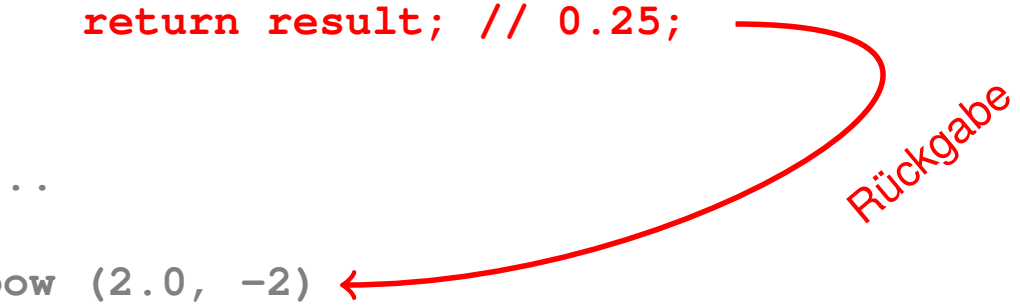
```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i) // i = 1
        result *= b; // result = 0.25
    return result;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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) // i = 2
        result *= b;
    return result;
}

...

pow (2.0, -2)
```

Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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; // 0.25;
}

...

pow (2.0, -2)
```



Beispiel: Auswertung Funktionsaufruf

```
double pow(double b, int e)
{
    assert (e >= 0 || b != 0);
    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;
}

...

pow (2.0, -2) // Wert = 0.25
```

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich
- Verwendung als Rückgabetyp für Funktionen, die *nur* einen Effekt haben

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich
- Verwendung als Rückgabetyp für Funktionen, die *nur* einen Effekt haben

```
// POST: "(i, j)" has been written to standard output
void print_pair (const int i, const int j)
{
    std::cout << "(" << i << ", " << j << ")\n";
}

int main()
{
    print_pair(3,4); // outputs (3, 4)
}
```

`void`-Funktionen

- benötigen kein **`return`**.
- Ausführung endet, wenn Ende des Funktionsrumpfes erreicht wird,

`void`-Funktionen

- benötigen kein **`return`**.
- Ausführung endet, wenn Ende des Funktionsrumpfes erreicht wird, oder
- **`return;`** erreicht wird, oder
- **`return`** *expression*`;` erreicht wird.

Ausdruck vom Typ **`void`** (z.B. Aufruf einer Funktion mit Rückgabetyp **`void`**

Formale Funktionsargumente[2]

- Deklarative Region: Funktionsdefinition
- sind ausserhalb der Funktionsdefinition *nicht* sichtbar
- werden bei jedem Aufruf der Funktion neu angelegt (automatische Speicherdauer)
- Änderungen ihrer Werte haben keinen Einfluss auf die Werte der Aufrufargumente (Aufrufargumente sind R-Werte)

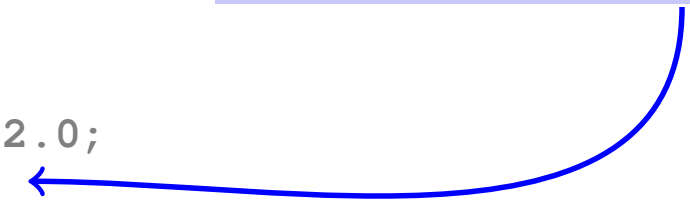
[2] oder „formale Parameter”

Beispiel zur Gültigkeit formaler Argumente

```
int main()
{
    double b = 2.0;
    int e = -2;

    std::cout << pow(b,e); // outputs 0.25
    std::cout << b; // outputs 2
    std::cout << e; // outputs -2
    return 0;
}
```

Beispiel zur Gültigkeit formaler Argumente

Nicht die formalen Argumente **b** und **e** von pow, sondern die hier definierten Variablen lokal zum Rumpf von **main**



```
int main()
{
    double b = 2.0;
    int e = -2;

    std::cout << pow(b,e); // outputs 0.25
    std::cout << b; // outputs 2
    std::cout << e; // outputs -2
    return 0;
}
```

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

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*.

Beispieldeklaration: `double pow (double b, int e)`

Gültigkeitsbereich einer Funktion: Beispiel

```
#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;
}
```

Gültigkeit f

Gültigkeitsbereich einer Funktion: Beispiel

```
#include<iostream>

int f (int i); // Gültigkeitsbereich von f ab hier

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

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

Gültigkeit f

Gültigkeitsbereich einer Funktion: Beispiel

Bei Deklaration: **`const int`** äquivalent zu **`int`**.

```
#include<iostream>

int f (int i); // Gültigkeitsbereich von f ab hier

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

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

Gültigkeit f

Gültigkeitsbereich einer Funktion: Beispiel

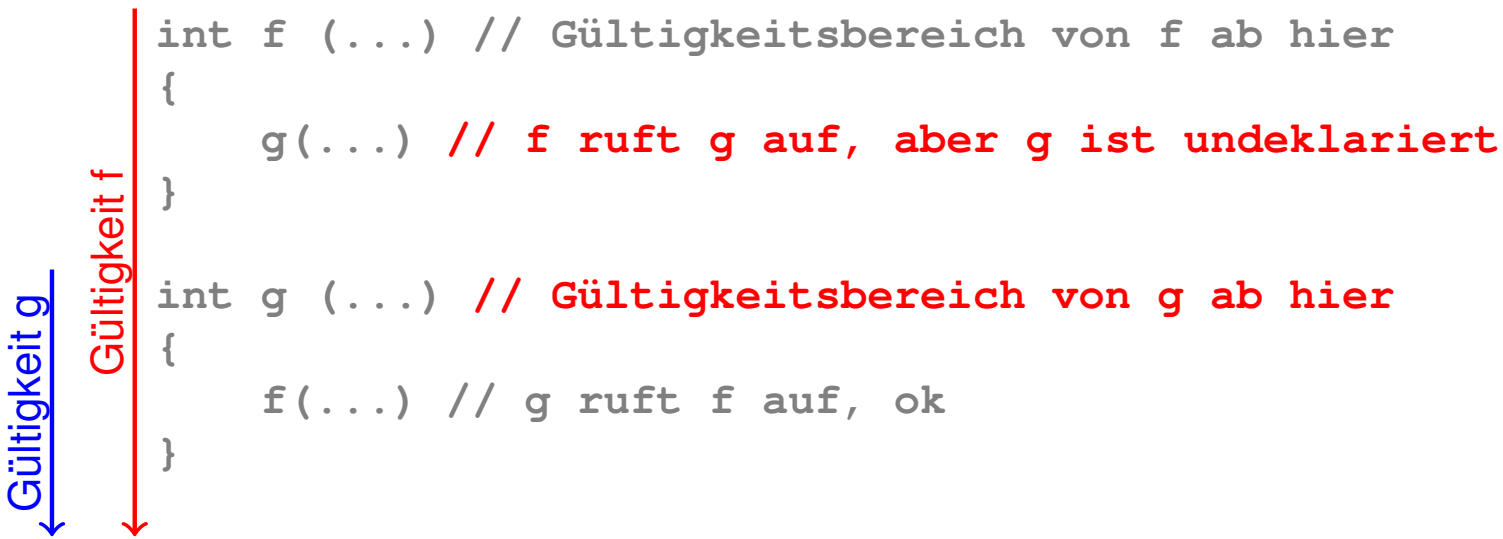
Bei Deklaration: **`const int`** äquivalent zu **`int`**. Grund: Verhalten der Funktion „nach aussen" ist unabhängig davon.

```
#include<iostream>

int f (int i); // Gültigkeitsbereich von f ab hier

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

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

Gültigkeit f

Notwendigkeit der separaten Deklaration

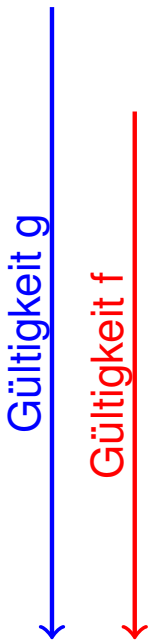
Separate Deklarationen sind manchmal notwendig. Problem:




```
int f (...) // Gültigkeitsbereich von f ab hier
{
    g(...) // f ruft g auf, aber g ist undeklariert
}

int g (...) // Gültigkeitsbereich von g ab hier
{
    f(...) // g ruft f auf, ok
}
```

Notwendigkeit der separaten Deklaration

Separate Deklarationen sind manchmal notwendig. Problem:




```
int g(...); // Gültigkeitsbereich von g ab hier

int f (...) // Gültigkeitsbereich von f ab hier
{
    g(...) // f ruft g auf, ok.
}

int g (...)
{
    f(...) // g ruft f auf, ok
}
```

Diese Lösung nennt sich auch „forward declaration”.

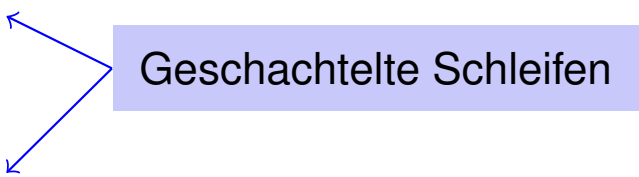
Prozedurales Programmieren

- Funktionen erlauben die Zerlegung der Gesamtaufgabe in klar abgegrenzte Teilaufgaben
- Bei Verwendung „sprechender” Funktionsnamen wird das Gesamtprogramm viel übersichtlicher und verständlicher

Beispiel Prozedurales Programmieren

Berechnung perfekter Zahlen (bisher):

```
// input
std::cout << "Find perfect numbers up to n =? ";
unsigned int n;
std::cin >> n;

// computation and output
std::cout << "The following numbers are perfect.\n";
for (unsigned int i = 1; i <= n ; ++i) {
    // check whether i is perfect
    unsigned int sum = 0;
    for (unsigned int d = 1; d < i; ++d)
        if (i % d == 0) sum += d;
    if (sum == i)
        std::cout << i << " ";
    }
```



Beispiel Prozedurales Programmieren

Berechnung perfekter Zahlen mit Funktionen:

```
// POST: return value is the sum of all divisors of i
// that are smaller than i
unsigned int sum_of_proper_divisors (const unsigned int i)
{
    unsigned int sum = 0;
    for (unsigned int d = 1; d < i; ++d)
        if (i % d == 0) sum += d;
    return sum;
}

// POST: return value is true if and only if i is a
// perfect number
bool is_perfect (const unsigned int i)
{
    return sum_of_proper_divisors (i) == i;
}
```

Beispiel Prozedurales Programmieren

Berechnung perfekter Zahlen mit Funktionen:

```
int main()
{
    // input
    std::cout << "Find perfect numbers up to n =? ";
    unsigned int n;
    std::cin >> n;

    // computation and output
    std::cout << "The following numbers are perfect.\n";

    for (unsigned int i = 1; i <= n ; ++i)
        if (is_perfect (i)) std::cout << i << " ";
    std::cout << "\n";
    return 0;
}
```

Beispiel Prozedurales Programmieren

Berechnung perfekter Zahlen mit Funktionen:

```
int main()
{
    // input
    std::cout << "Find perfect numbers up to n =? ";
    unsigned int n;
    std::cin >> n;

    // computation and output
    std::cout << "The following numbers are perfect.\n";

    for (unsigned int i = 1; i <= n ; ++i)
        if (is_perfect (i)) std::cout << i << " ";
    std::cout << "\n";
    return 0;
}
```

Das Programm ist selbsterklärend!

Keine geschachtelten Schleifen mehr

Prozedurales Programmieren

- Bisher konnten wir nur ohne Funktionen leben, weil die Programme meistens einfach und kurz waren

Prozedurales Programmieren

- Bisher konnten wir nur ohne Funktionen leben, weil die Programme meistens einfach und kurz waren
- Bei komplizierteren Aufgaben schreibt man ohne Funktionen leicht Spaghetti-Code, so wie in der...

Prozedurales Programmieren

- Bisher konnten wir nur ohne Funktionen leben, weil die Programme meistens einfach und kurz waren
- Bei komplizierteren Aufgaben schreibt man ohne Funktionen leicht Spaghetti-Code, so wie in der...
- ...Programmiersprache BASIC (1963+)

Modularisierung

- Funktionen wie `pow` sind in vielen Programmen nützlich.
- Es ist nicht sehr praktisch, die Funktionsdefinition in jedem solchen Programm zu wiederholen.
- Ziel: Auslagern der Funktion.

Modularisierung: Auslagerung

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;
}
```

Modularisierung: Auslagerung

Aufrufendes Programm `callpow2.cpp`

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

#include <iostream>
#include "pow.cpp"

int main()
{
  std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
  std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
  std::cout << pow( 5.0,  1) << "\n"; // outputs 5
  std::cout << pow( 3.0,  4) << "\n"; // outputs 81
  std::cout << pow(-2.0,  9) << "\n"; // outputs -512

  return 0;
}
```

Modularisierung: Auslagerung

Aufrufendes Programm `callpow2.cpp`

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

#include <iostream>
#include "pow.cpp"

int main()
{
  std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
  std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
  std::cout << pow( 5.0,  1) << "\n"; // outputs 5
  std::cout << pow( 3.0,  4) << "\n"; // outputs 81
  std::cout << pow(-2.0,  9) << "\n"; // outputs -512

  return 0;
}
```

Dateiangabe relativ zum Arbeitsverzeichnis.

Modularisierung: getrennte Übersetzung

Problem der Auslagerung

- **`#include`** kopiert den Inhalt der inkludierten Datei (**`pow.cpp`**) in das inkludierende Programm (**`callpow2.cpp`**).
- Compiler muss die Funktionsdefinition für jedes aufrufende Programm neu übersetzen.
- Bei vielen und grossen Funktionen kann das sehr lange dauern.

Modularisierung: getrennte Übersetzung

Lösung:

- **`pow.cpp`** kann getrennt übersetzt werden (z.B. **`g++ -c pow.cpp`**).
- Resultat ist kein ausführbares Programm (**`main`** fehlt ja), sondern *Objekt-Code* **`pow.o`**.

Modularisierung: getrennte Übersetzung

Lösung:

- **`pow.cpp`** kann getrennt übersetzt werden (z.B. **`g++ -c pow.cpp`**).
- Resultat ist kein ausführbares Programm (**`main`** fehlt ja), sondern *Objekt-Code* **`pow.o`**.

Enthält Maschinensprache-Befehle für Funktionsrumpf von **`power`**.

Modularisierung: getrennte Übersetzung

- Auch das aufrufende Programm kann getrennt übersetzt werden, ohne Kenntnis von **`pow.cpp`** oder **`pow.o`**.

Modularisierung: getrennte Übersetzung

- Auch das aufrufende Programm kann getrennt übersetzt werden, ohne Kenntnis von **`pow.cpp`** oder **`pow.o`**.
- Compiler muss nur die *Deklaration* von **`pow`** kennen; diese schreiben wir separat in eine *Header*-Datei **`pow.h`**:

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

Modularisierung: getrennte Übersetzung

Aufrufendes Programm `callpow3.cpp`

```
// Prog: callpow3.cpp
// Call a function for computing powers.

#include <iostream>
#include "pow.h"

int main()

  std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
  std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
  std::cout << pow( 5.0,  1) << "\n"; // outputs 5
  std::cout << pow( 3.0,  4) << "\n"; // outputs 81
  std::cout << pow(-2.0,  9) << "\n"; // outputs -512

  return 0;
```

Modularisierung: getrennte Übersetzung

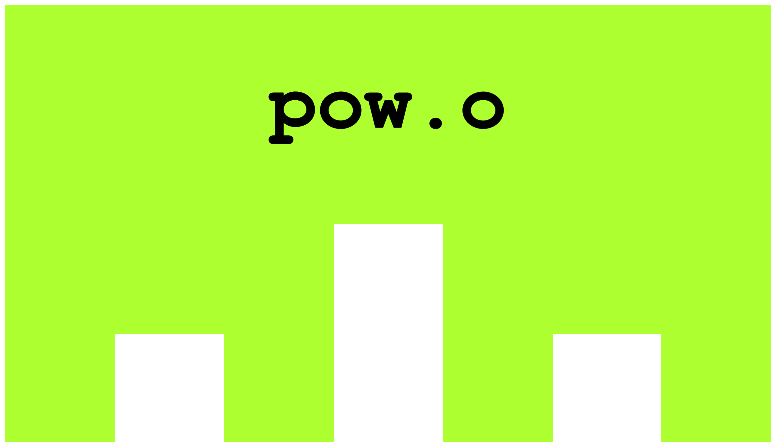
- **`callpow3.cpp`** kann getrennt übersetzt werden. (**`g++ -c callpow3.cpp`**)
- Resultat ist kein ausführbares Programm (**`pow`** fehlt), sondern *Objekt-Code* **`callpow3.o`**

Statt der Maschinensprache-Befehle für Funktionsrumpf von **`pow`** enthält **`callpow3.o`** einen *Platzhalter* für die Speicheradresse, unter der diese zu finden sein werden.

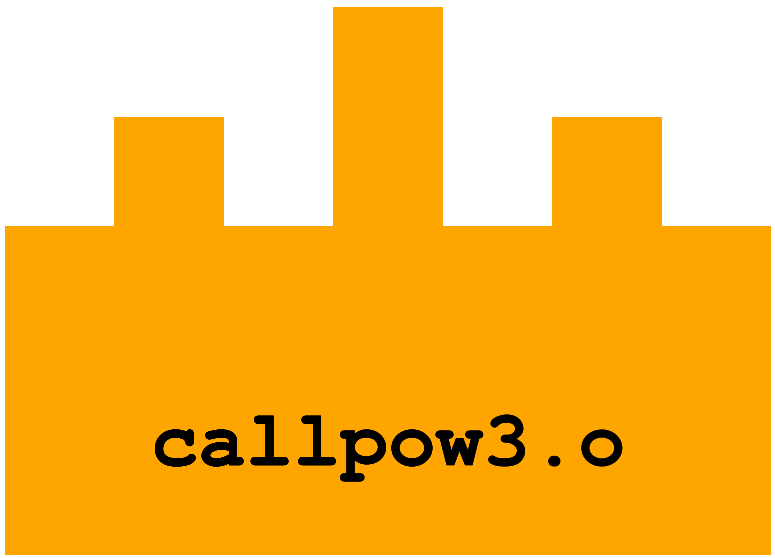
Modularisierung: Der Linker

- baut ausführbares Programm aus den relevanten Objekt-Codes (**`pow.o`** und **`callpow3.o`**) zusammen
- ersetzt dabei die Platzhalter für Funktionsrumpfadressen durch deren wirkliche Adressen im ausführbaren Programm

Modularisierung: Der Linker



Maschinensprache für den Funktionsrumpf von **pow**



Maschinensprache für die Aufrufe von **pow**

Modularisierung: Der Linker

Maschinensprache für den Funktionsrumpf von **`pow`**

Maschinensprache für die Aufrufe von **`pow`**

Modularisierung: Verfügbarkeit von Quellcode

- `pow.cpp` (Quellcode) wird nach dem Erzeugen von `pow.o` (Object Code) nicht mehr gebraucht und könnte gelöscht werden.
- Viele (meist kommerzielle) Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode
- Die Vor-und Nachbedingungen in den Header-Dateien sind die *einzigen* verfügbaren Informationen.

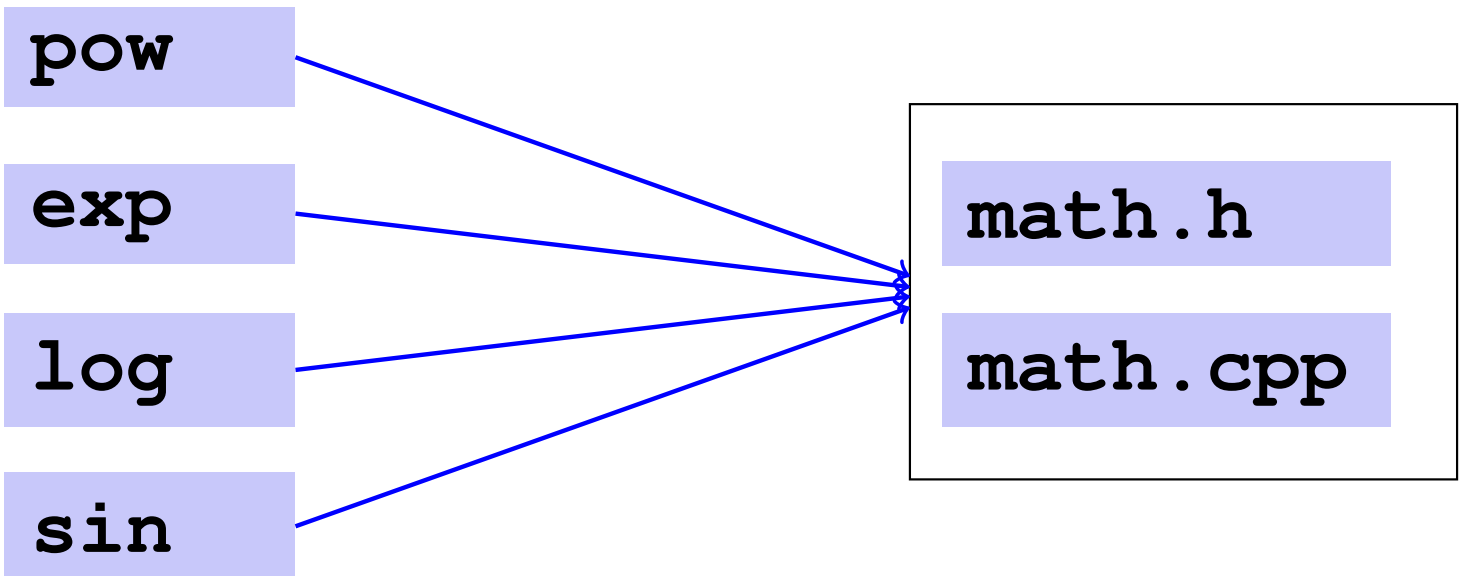
Modularisierung: Verfügbarkeit von Quellcode

„Open Source” Software:

- Alle Quellen sind verfügbar
- Nur das Erlaubt die Weiterentwicklung durch Benutzer und andere interessierte Personen.
- Selbst im kommerziellen Bereich ist „open source” auf dem Vormarsch.
- Lizenzbedingungen forcieren Nennung der Quellen und die offene Weiterentwicklung. Beispiel: GPL.
- Bekannteste „open source” Software: Betriebssystem Linux.

Modularisierung: Bibliotheken

- Gruppierung ähnlicher Funktionen zu Bibliotheken



Modularisierung: Bibliotheken

Eigener Namensraum vermeidet Konflikte mit Benutzer- oder Standard-Funktionalität

```
// math.h
// A small library of mathematical functions.

namespace ifm {
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e);
....
double exp (double x);
...
}
```

Standardbibliothek

Das Benutzen von Funktionen aus der *Standardbibliothek*

- vermeidet die Neuerfindung des Rades (z.B. **`std::pow`**, **`std::sqrt`**)
- führt auf einfache Weise zu interessanten und effizienten Programmen
- garantiert einen gewissen Qualitäts-Standard

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 unsigned int bound
= (unsigned int)(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 $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ ein Teiler von n ist.

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

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

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

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

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

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

Primzahltest mit `sqrt`

```
// Program: prime2.cpp
// Test if a given natural number is prime.
#include <iostream>
#include <cmath>

int main ()
{
    unsigned int n;
    std::cout << "Test if n>1 is prime for n =? ";
    std::cin >> n;

    // Computation: test possible divisors d up to sqrt(n)
    const unsigned int bound = (unsigned int)(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;
}
```

Funktionen aus der Standardbibliothek

`prime2.cpp` könnte inkorrekt sein, falls z.B.
`std::sqrt(121)==10.998`

Funktionen aus der Standardbibliothek

`prime2.cpp` könnte inkorrekt sein, falls z.B. `std::sqrt(121)==10.998`

- **`bound == 10`**
- **`d == 11`** am Ende von for
- Ausgabe: **`121 is prime.`**

Funktionen aus der Standardbibliothek

- Für **`std::sqrt`** garantiert der IEEE Standard 754 noch, dass der exakte Wert auf den nächsten darstellbaren Wert gerundet wird (wie bei `+`, `-`, `*`, `/`)

Funktionen aus der Standardbibliothek

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

Funktionen aus der Standardbibliothek

- Für andere mathematische Funktionen gibt der IEEE Standard 754 keine solchen Garantien, sie können also auch weniger genau sein!
- Gute Programme müssen darauf Rücksicht nehmen und, falls nötig, „Sicherheitsmassnahmen” einbauen.