Die erste C++ 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 (später mehr dazu)

Funktion zur 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;
}
```

Vorbedingung

Nachbedingung

Programm zur Potenzberechnung

```
// Prog: callpow.C
// 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 (wir selbst oder andere)
- machen Programme lesbarer: wir müssen nicht verstehen, *wie* die Funktion es macht
- werden vom Compiler ignoriert

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 dem Funktionsaufruf ?
- spezifiziert *Wert* und *Effekt* des Funktionsaufrufs

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.
- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage!

Vor-und Nachbedingungen

- 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
```

sind 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 hässlich. Wir abstrahieren und geben die mathematischen Bedingungen an.

Assertions

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

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
  assert (e >= 0 || b != 0.0);
  double result = 1.0;
  // der Rest ist wie vorher...
  ...
}
```

Falls *expression* Wert *true* hat: kein Effekt.

`assert` (*expression*)

Ausdruck, dessen Wert nach `bool` konvertierbar ist

Assertions

Wenn das fertige Programm einmal alle Assertions "überlebt", so können wir den Compiler anweisen, sie "hinauszukompilieren" (kein Laufzeitverlust!).

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
  assert (e >= 0 || b != 0.0);
  double result = 1.0;
  // der Rest ist wie vorher...
  ...
}
```

Falls *expression* Wert *false* hat: Programm wird mit enstprechender Fehlermeldung abgebrochen.

`assert` (*expression*)

Ausdruck, dessen Wert nach `bool` konvertierbar ist

Assertions

`#include <cassert>`

Assertions sind ein wichtiges Werkzeug zur Fehlervermeidung während der Programmentwicklung, wenn man sie *konsequent* und *oft* einsetzt.

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
  assert (e >= 0 || b != 0.0);
  double result = 1.0;
  // der Rest ist wie vorher...
  ...
}
```

Falls *expression* Wert *false* hat: Programm wird mit enstprechender Fehlermeldung abgebrochen.

`assert` (*expression*)

Ausdruck, dessen Wert nach `bool` konvertierbar ist

Funktionsdefinitionen

T fname (*T1* pname1, *T2* pname2, …, *TN* pnameN)
block

formale Argumente

Funktionsname

Funktionsrumpf

Argumenttypen

Rückgabetyp

Funktionsdefinitionen

- dürfen nicht *lokal* (in Blocks, anderen Funktionen oder Kontrollanweisungen) auftreten
- können im Programm ohne Trennsymbole aufeinander folgen

Funktionsdefinitionen

- können im Programm ohne Trennsymbole aufeinander folgen

```
int foo (int x)
{
   return x;
}
int bar (int x)
{
   return x + 1;
}
```

Diese Namen haben sich für inhaltsleere Funktionen eingebürgert, die lediglich C++ Aspekte demonstrieren.

Funktionsdefinitionen

- können ohne Trennsymbole aufeinander folgen

```
int foo (int x)
{
   return x;
}
int bar (int x)
{
   return x + 1;
}
```

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

Funktionsaufrufe

(z.B. `pow (2.0, -2)`)

Aufrufargumente (Typen konvertierbar in *T1,…,TN*)

fname (*expression1, expression2,…, expressionN*)

Ausdruck vom Typ *T* (Wert und Effekt wie in der Nachbedingung von *fname*)

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufparameter sind R-Werte
- Funktionsaufruf selbst ist R-Wert

fname : R-Wert × R-Wert × … × R-Wert ⟶ R-Wert

Auswertung eines Funktionsaufrufs

- 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: Wert des Funktionsaufrufs

Auswertung eines Funktionsaufrufs: Beispiel

```
pow (2.0, -2)
```

```
double pow (double b, int e)
{ // 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;
}
```

Auswertung eines Funktionsaufrufs: Beispiel

```
pow (2.0, -2)
```

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
     result *= b;
  return result;
}
```

Auswertung eines Funktionsaufrufs: Beispiel

```
pow (2.0, -2)
```

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
     result *= b; // 0.5
  return result;
}
```

i==0

Auswertung eines Funktionsaufrufs: Beispiel

```
pow (2.0, -2)
```

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
    result *= b; // 0.25
  return result;
}
```

i==1

Auswertung eines Funktionsaufrufs: Beispiel

```
pow (2.0, -2)
```

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
     result *= b; // 0.25
  return result;  // 0.25
}
```

Auswertung eines Funktionsaufrufs: Beispiel

```
pow (2.0, -2)
```

Auswertung komplett

```
double pow (double b, int e)
{ // b = 2.0; e = -2;
  double result = 1.0;
  if (e < 0) {
    // b^e = (1/b)^(-e)
    b = 1.0/b; // b = 0.5;
    e = -e;    // e = 2;
  }
  for (int i = 0; i < e; ++i)
     result *= b; // 0.25
  return result;  // 0.25
}
```

Rückgabe

0.25

Der Typ void

- fundamentaler Typ (Wertebereich leer)
- Verwendung als Rückgabetyp für Funktionen, die *nur* Effekt haben

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

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

Void-Funktion

Void-Funktionen

- brauchen 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

- 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)

Gültigkeitsbereich formaler Funktionsargumente: Beispiel 1

```
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;
}
```

nicht die `b` und `e` in der Definition von pow, sondern die hier definierten Variablen lokal zum Rumpf von `main`

Gültigkeitsbereich formaler Funktionsargumente: Beispiel 2

Lokale Variablen im Funktionsrumpf dürfen nicht so heissen wie ein formales Argument:

```
int f (int i)
{
  int i = 5; // ungültig; i versteckt Argument
  return i;
}
```

Gültigkeitsbereich formaler Funktionsargumente: Beispiel 2

Lokale Variablen im Funktionsrumpf dürfen nicht so heissen wie ein formales Argument, ausser sie sind in einem geschachtelten Block deklariert:

```
int f (int i)
{
  {
    int i = 5; // ok; i ist lokal zum Block
  }
  return i; // das formale Argument
}
```

Gültigkeitsbereich formaler Funktionsargumente: Beispiel 2

Lokale Variablen im Funktionsrumpf dürfen nicht so heissen wie ein formales Argument, ausser sie sind in einem geschachtelten Block deklariert:

```
int f (int i)
{
  {
    int i = 5; // syntaktisch ok, aber schlechter
  }            // Stil
  return i; // das formale Argument
}
```

Gültigkeitsbereich einer Funktion

- ist der Teil des Programms, in dem die Funktion aufgerufen werden kann
- ist definiert als die Vereinigung der Gültigkeitsbereiche aller ihrer *Deklarationen* (es kann mehrere geben)

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

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

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
}
```

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
}
```

Prozedurales Programmieren

- Funktionen erlauben die Zerlegung der Gesamtaufgabe in klar abgegrenzte Teilaufgaben
- Bei Verwendung "sprechender" Funktionsnamen wird das Gesamt-programm viel übersichtlicher und verständlicher

Prozedurales Programmieren: Berechnung perfekter Zahlen

Bisher (Musterlösung)

```
// 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 << " ";
}
```

Was ist nochmal eine perfekte Zahl???

Geschachtelte Schleifen

Prozedurales Programmieren: Berechnung perfekter Zahlen

Neu mit Funktionen

```
// POST: return value is the sum of all divisors of i
//       that are smaller than i
unsigned int sum_of_proper_divisors (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 (unsigned int i)
{
  return sum_of_proper_divisors (i) == i;
}
```

Prozedurales Programmieren: Berechnung perfekter Zahlen

Neu 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

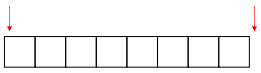
keine geschachtelten Schleifen mehr

Prozedurales Programmieren

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

Felder als Funktionsargumente

- Zu Erinnerung: wir können Felder nicht mit anderen Feldern initialisieren
- Genau das bräuchten wir aber, wenn wir Felder als Funktionsargumente benutzen wollten!
- Lösung: wir übergeben das Feld mittels zwei Zeigern!



Felder als Funktionsargumente

- Beispiel: Füllen aller Feldelemente mit einem festen Wert

Hier "leben" wirklich Elemente eines Feldes

Zeiger auf erstes Element

Zeiger *hinter* das letzte Element (past-the-end)

```
// PRE:  [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, int value) {
  // iteration by pointer
  for (int* p = first; p != last; ++p)
    *p = value;
}
```

Durchlaufe den Bereich und setze jedes Element auf `value`

Felder als Funktionsargumente

- Anwendungsbeispiel:

```
int main()
{
  int a[5];
  fill (a, a+5, 1); // a == {1, 1, 1, 1, 1}
  return 0;
}
```

Felder als Funktionsargumente

- Anwendungsbeispiel (unter Verwendung der Standardbibliothek):

```
#include <algorithm>
int main()
{
  int a[5];
  std::fill (a, a+5, 1); // a == {1, 1, 1, 1, 1}
  return 0;
}
```

Modularisierung

- Funktionen wie **`pow, fill,...`** 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.C`**:

```
// 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.C`**:

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

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

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.C`**) in das inkludierende Programm (**`callpow2.C`**)
- Compiler muss die Funktionsdefinition für jedes aufrufende Programm neu übersetzen.
- bei vielen und grossen Funktionen kann das jeweils sehr lange dauern.

Modularisierung: getrennte Übersetzung

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

Maschinensprache-Befehle für Funktionsrumpf von **pow**

Modularisierung: getrennte Übersetzung

- Auch das aufrufende Programm kann getrennt übersetzt werden, ohne Kenntnis von **pow.C** oder **pow.o** !
- Compiler muss nur die *Deklaration* von **pow** kennen; diese schreiben wir 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.C:**

```
// Prog: callpow3.C
// 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

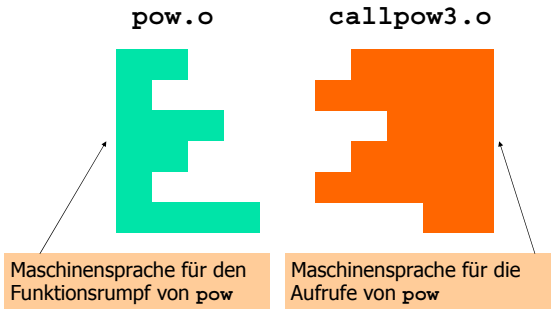
- **callpow3.C** kann getrennt übersetzt werden (z.B. **g++ -c callpow3.C**)
- Resultat ist kein ausführbares Programm (**pow** fehlt), sondern Objekt-Code **callpow3.o**

Anstatt 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

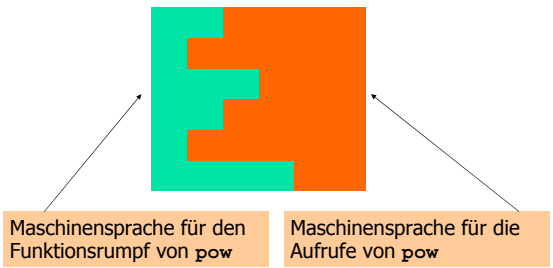
- baut ausführbares Programm aus den relevanten Objekt-Codes (`pow.o` und `callpow3.o`) zusammen
- ersetzt dabei die Platzhalter für Funktionsrumpfadressen durch deren wirkliche Adressen im ausführbaren Programm

Modularisierung: Der Linker



Modularisierung: Der Linker

`callpow3` (ausführbares Programm)



Modularisierung: Verfügbarkeit von Quellcode

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

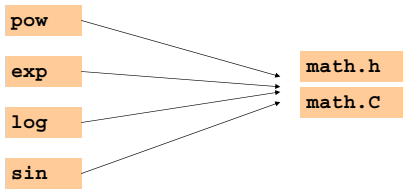
Modularisierung: Verfügbarkeit von Quellcode

"Open source" Software:

- Alle Quellen sind verfügbar
- nur das erlaubt die Weiterentwicklung durch Benutzer und andere interessierte Personen
- im kommerziellen Bereich ist "open source" auf dem Vormarsch (trotz Microsoft…)
- Bekannteste "open source" Software: Das Betriebssystem **Linux**

Modularisierung: Bibliotheken

- Gruppierung ähnlicher Funktionen zu Bibliotheken



Modularisierung: Bibliotheken

- Eigener Namensraum vemeidet Konflikte mit Benutzer-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);
  ...
}
```

Benutzen von Funktionen aus der Standardbibliothek

- vermeidet die Neuerfindung des Rades (z.B. gibt es `std::pow, std::fill`)
- führt auf einfache Weise zu interessanten und effizienten Programmen

Primzahltest mit `std::sqrt`

n ≥ *2* ist Primzahl genau dann, wenn kein *d* in {*2,…,n-1* } ein Teiler von *n* ist.

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

Primzahltest mit `std::sqrt`

n ≥ *2* ist Primzahl genau dann, wenn kein *d* in {*2,…,√n* } ein Teiler von *n* ist.

```
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` (das heisst d < n):

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

Primzahltest mit `std::sqrt`

n ≥ *2* ist Primzahl genau dann, wenn kein *d* in {*2,…,√n* } ein Teiler von *n* ist.

```
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` (das heisst d > √n):

Dann: kein Teiler in {2,…,√n}, n ist Primzahl

Primzahltest mit `std::sqrt`

```
// Program: prime2.C
// Test if a given natural number is prime.

#include <iostream>
#include <cmath>

int main ()
{
  // Input
  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)
  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;
}
```

Benutzen von Funktionen aus der Standardbibliothek

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

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

Benutzen von 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.C` ist korrekt unter dem IEEE Standard 754

Benutzen von Funktionen aus der Standardbibliothek

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