C++ Teil 4

Sven Groß

9. Nov 2015

Themen der letzten Vorlesung

- Funktionen:
  - Definition und Aufruf
  - Wert- und Referenzparameter,
    call by value / call by reference
- Lesbarkeit von Code:
  Einrückung, Kommentare

Heutige Themen

1. Namensräume
2. Hands-on Live Programming zu A2
3. Gleitkommazahlen
   - Rundungsfehler
   - Auswirkung auf Vergleiche
4. Funktionen
   - Überladung
   - Rekursion

Namensraum

- **Namensraum** kapselt Bezeichner (z.B. Variablen- und Funktionsnamen)
  ⇝ vermeidet Konflikte z.B. mit Funktionen aus eingebundenen Bibliotheken

```
namespace Mein {
  double sqrt( double x);
  ...
} // end of namespace "Mein"
```

- **Qualifizierung** *außerhalb* des Namensraum jeweils mit Scope-Operator `::` oder einmal zu Anfang mit `using`-Anweisung (z.B. `using std::endl;`)

```
std::cout << "Standard-Wurzel: " <<  std::sqrt(2.0)
          << ",  meine Wurzel: " << Mein::sqrt(2.0)
          << endl;
```

- Qualifizierung aller Elemente eines Namensraum mit `using namespace ...` möglich, aber nicht zu empfehlen (Unklarheiten vorprogrammiert).

  Vorsicht: `using namespace std;` zwar bequem, macht aber alle Vorteile des Namensraum zunichte!

Binäre Zahlendarstellung

- `char`, `int`, `unsigned int`, `long`, ... werden als **Binärzahlen** dargestellt.

  Beispiel:
  $$\begin{aligned}(11010)_2 &= \mathbf{1}\cdot 2^4 + \mathbf{1}\cdot 2^3 + \mathbf{0}\cdot 2^2 + \mathbf{1}\cdot 2^1 + \mathbf{0}\cdot 2^0 \\ &= 16 + 8 + 2 = 26.\end{aligned}$$

- `double`, `float` werden als binäre **Gleitkommazahlen** dargestellt:
  $$\hat{x} = \pm 0.d_1 \dots d_m \cdot 2^e \in \mathbb{M}$$
  mit **Binärziffern** $d_j \in \{0, 1\}$, **Mantissenlänge** m und **Exponent** e.
  $\mathbb{M}$ ist die Menge der **Maschinenzahlen**.

- **Beispiel:** $(m = 6)$
  $$\begin{aligned}\hat{x} &= (0.101110)_2 \cdot 2^{(11)_2} \in \mathbb{M} \\ &= (\mathbf{1}\cdot 2^{-1} + \mathbf{0}\cdot 2^{-2} + \mathbf{1}\cdot 2^{-3} + \mathbf{1}\cdot 2^{-4} + \mathbf{1}\cdot 2^{-5}) \cdot 2^3 \\ &= 2^2 + 2^0 + 2^{-1} + 2^{-2} = 5.75.\end{aligned}$$

Gleitkommazahlen: Rundungsfehler

- viele $x \in \mathbb{R}$ nicht exakt darstellbar, z.B. $x = 0.3 \notin \mathbb{M} \rightsquigarrow$ **Rundung** zu $\hat{x} \in \mathbb{M}$

```
double a= 0.3;   //  a ist nicht exakt 0.3
```

- arithmetische Operationen mit Gleitkommazahlen (*floating point operations, FLOPs*): Assoziativ-, Distributivgesetz gelten nicht!
- Grund sind **Rundungsfehler**: nach Operation wird Ergebnis $x \in \mathbb{R}$ auf nächste darstellbare Gleitkommazahl $\hat{x} \in \mathbb{M}$ gerundet.

relativer Fehler durch Rundung

$$\left| \frac{\hat{x} - x}{x} \right| \le \frac{1}{2^m} =: \mathrm{eps}$$

$$\Rightarrow \qquad \hat{x} = x \cdot (1 + \varepsilon) \qquad \text{mit } |\varepsilon| \le \mathrm{eps}.$$

- **Maschinengenauigkeit** eps:
  - für `double` ist $\mathrm{eps} = 2^{-53} \approx 10^{-16}$,
  - für `float` ist $\mathrm{eps} = 2^{-24} \approx 6 \cdot 10^{-8}$ (nicht mal TR-Genauigkeit)

Rundungsfehler: Auswirkung auf Vergleiche

```
int main()
{
    const double elftel= 1./11.;

    for (double x=0; x <= 1.0; x+= elftel)
        cout << "x␣=␣" << x << endl;
    return 0;
}
```

- Schleife sollte (mathematisch) eigentlich von $\frac{0}{11}, \frac{1}{11}, \ldots$ bis $\frac{11}{11} = 1$ zählen, also 12 Durchläufe, aber
  - $\frac{1}{11}$ im Rechner nicht exakt darstellbar $\quad \rightarrow$ Rundungsfehler
  - Rundungsfehler nach jeder Addition durch `+=`
  - Rundungsfehler häufen sich an (akkumulieren)
- je nach Prozessortyp / Compiler / Optimierungsstufe wird die Schleife 11 oder 12 mal durchlaufen: letzter Durchgang wird evtl. weggelassen, da dann `x` evtl. geringfügig größer als 1 ist

Sinnvolle Vergleiche mit double

Bessere Alternativen?

1. Vergleich robust für Rundungsfehler machen:

```
    for (double x=0; x <= 1.0 + 1e-8; x+= elftel)
        cout << "x␣=␣" << x << endl;
```

2. Schleifenvariablen immer ganzzahlig wählen:

```
    for (int i=0; i <= 11; ++i)
        cout << "x␣=␣" << i*elftel << endl;
```

Vorsicht bei `==` :

- Vergleiche wie `x == 0.1` testen Gleichheit der Binärdarstellung, wegen Rundungsfehlern sinnlos, bitte immer so:

```
  if ( std::abs( x - 0.1) <= 1e-9 )
  {
      ...
  }
```

Signatur einer Funktion

- **Signatur** einer Funktion = Name + Parameterliste
- Funktionen dürfen selben Namen besitzen,
  aber Signatur muss eindeutig sein.
- Drei verschiedene Funktionen mit verschiedener Signatur:

```
    void PrintSum( int x, int y);
    void PrintSum( double x, double y);
    void PrintSum( double x, double y, double z);
```

- *Nicht* erlaubt, da beide Funktionen selbe Signatur besitzen:

```
    int    Summe( double x, double y);
    double Summe( double a, double b);
```

- Eindeutige Signatur wichtig, damit der Compiler beim Funktionsaufruf die richtige Funktion auswählen kann, z.B. bei `PrintSum( 1.0, 3.0);`

Funktionsüberladung

```
void PrintSum( int x, int y)
{
    cout << "i-sum␣=␣" << x + y << endl;
}

void PrintSum( double x, double y)
{
    cout << "d-sum␣=␣" << x + y << endl;
}

int main()
{
    PrintSum( 1, 3);      // Ausgabe "i-sum = 4"
    PrintSum( 1.3, 2.7); // Ausgabe "d-sum = 4"
    return 0;
}
```

- **Überladen**: Funktionen mit gleichem Namen und gleicher Parameterzahl, aber verschiedenen Typen $\rightsquigarrow$ unterscheidbar durch Signatur
- Eindeutige Signatur wichtig, damit richtige Funktion ausgewählt wird

Funktionsüberladung (2)

```
void PrintSum( double x, double y)
{
    cout << "d-sum␣=␣" << x + y << endl;
}

int main()
{
    PrintSum( 1, 3);     // Ausgabe "d-sum = 4"
    PrintSum( 1.3, 2.7); // Ausgabe "d-sum = 4"
    return 0;
}
```

- `PrintSum( 1, 3);` bewirkt keinen Compilerfehler, obwohl keine `int`-Funktion deklariert ist.
- Grund: automatische, implizite Typumwandlung (**Cast**) für `int` $\to$ `double`

$\leadsto$ oft nützlich, aber manchmal auch gefährlich.

Funktionsüberladung (3)

```
void PrintSum( int x, int y)
{
    cout << "i-sum␣=␣" << x + y << endl;
}

int main()
{
    PrintSum( 1, 3);      // Ausgabe "i-sum = 4"
    PrintSum( 1.3, 2.7); // Ausgabe "i-sum = 3"
    return 0;
}
```

- `PrintSum( 1.3, 2.7);` bewirkt keinen Compilerfehler, obwohl keine `double`-Funktion deklariert ist.
- Grund: automatische, implizite Typumwandlung (**Cast**) für `double` → `int`

⇝ Informationsverlust!

Funktionsüberladung – warnendes Beispiel

```cpp
#include <iostream>
using namespace std;  // gefaehrlich !

void max( double a, double b)
{
    double maxi= a>b ? a : b;
    cout << "Das␣Maximum␣ist␣" << maxi << endl;
}

int main()
{
    max( 3, 4);      // ruft std::max fuer int auf
    return 0;
}
```

- am besten passende Funktion wird aufgerufen,
  hier `int std::max( int, int)`
- **Besser:** `using std::cout; using std::endl;` statt `using namespace std;`, um Vorteil des Namensraums (Kapselung, Eindeutigkeit) zu erhalten

Rekursive Funktionen

- **Rekursive Funktionen** rufen sich selber auf
- sicherstellen, dass Rekursion abbricht!
- einfache Rekursion kann auch durch Iteration (Schleife) ersetzt werden

```
int fakultaet_rekursiv( int n)
{
    if (n>1)
        return n*fakultaet_rekursiv(n-1);
    else
        return 1;     // Abbruch der Rekursion
}

int fakultaet_iterativ( int n)
{
    int fak= 1;
    for (int i=1; i<=n; ++i)
        fak*= i;

    return fak;
}
```

Rekursive Funktionen (2)

Beispiel: Fibonacci-Zahlen: $a_0 := a_1 := 1, \quad a_n := a_{n-2} + a_{n-1}$ für $n \geq 2$

```
int fib_rekursiv( int n)
{
    if (n >= 2)
      return fib_rekursiv( n-2) + fib_rekursiv( n-1);
    else
      return 1;     // Abbruch der Rekursion
}
```

- Rekursion hier sehr ineffizient!
  - `fib_rekursiv( n )` erzeugt $\mathcal{O}(2^{n-1})$ Aufrufe von sich selbst.
  - Bei Berechnung von a_n wird z.B. a_2 sehr oft neu berechnet.

$\rightsquigarrow$ **besser:** Rekursion durch Iteration ersetzen

Rekursive Funktionen (3)

Beispiel: Fibonacci-Zahlen: $a_0 := a_1 := 1, \quad a_n := a_{n-2} + a_{n-1}$ für $n \geq 2$

```
int fib_iterativ( int n)
{
    if (n < 2)
        return 1;

    int ak, ak1= 1, ak2= 1;  // a_k, a_{k-1}, a_{k-2}

    for (int k= 2; k <= n; ++k)
    {
        ak= ak2 + ak1;        // nach Definition
        ak2= ak1;  ak1= ak; // Variablen shiften
    }
    return ak;
}
```

Iterativer Ansatz ist deutlich effizienter:

- Aufwand zur Berechnung von a_n ist linear in n (nicht exponentiell)
- a_2 wird genau einmal berechnet (für `k=2`)