

C++ Teil 3

Sven Groß



22. Okt 2018

Organisatorisches: Online-Tests

- erster Online-Test während des nächsten Praktikumstermins freigeschaltet
- Passwort teilen Tutoren mit
- 10min Zeit ab Start des Tests, 5 MC-Fragen
- Inhalt: alle C++-Themen bisher, also Folien nochmal anschauen:
Scope, Ein-/Ausgabe, bed. Verzweigung, Schleifen, Funktionen, Unterschied von Vergleich und Zuweisung, ...

- Bewertungsschema: nichts Falsches ankreuzen

☐	⓪ ⓧ ⓧ ⓧ	☒		
☐	ⓧ ⓧ ⓧ ⓧ			☒
☐	⓪ ⓧ ⓧ ⓧ			
☐	ⓧ ⓧ ⓧ ⓧ	☒	☒	☒
		100%	50%	0%

- Kontrollstrukturen:
 - bedingte Verzweigung: `if-else`
 - Schleifen: `while`, `do-while`, `for`
- weitere Operatoren: `+=` `*=` `++`
- Funktionen

1 Funktionen

- Definition und Aufruf
- Wert-/Referenzparameter

2 Namensräume

3 Gleitkommazahlen

- Rundungsfehler
- Auswirkung auf Vergleiche

Funktionsdefinition

Beispiel: Mittelwert zweier reeller Zahlen

Eingabe/Parameter: zwei reelle Zahlen x , y

Ausgabe: eine reelle Zahl

Rechnung: $z = (x + y)/2$,
dann z zurückgeben

```
double MW( double x, double y)    // Funktionskopf
{ // Funktionsrumpf
    double z = (x + y)/2;
    return z;
}
```

- x , y und z sind lokale Variablen des Rumpfblockes
→ werden beim Verlassen des Rumpfes zerstört

Funktionsaufruf

```
1  double MW( double x, double y)    // Funktionskopf
2  { // Funktionsrumpf
3      return (x + y)/2;
4  }
5
6  int main()        // ist auch eine Funktion...
7  {
8      double a= 4.0, b= 8.0, result;
9      result= MW( a, b);
10     return 0;
11 }
```

Was geschieht beim Aufruf `result = MW( a, b);` ?

- Parameter `x`, `y` werden angelegt als **Kopien** der Argumente `a`, `b`. (*call by value*)
- Rumpfblock wird ausgeführt.
- Bei Antreffen von `return` wird der dortige Ausdruck als Ergebnis zurückgegeben und der Rumpf verlassen (`x`, `y` werden zerstört).
- An `result` wird der zurückgegebene Wert **6.0** zugewiesen.

weitere Funktionen

- `int main()` ist das Hauptprogramm, gibt evtl. Fehlercode zurück
- Es gibt auch Funktionen ohne Argumente und/oder ohne Rückgabewert:

```
void SchreibeHallo()  
{  
    cout << "Hallo!" << endl;  
}
```

- Eine Funktion mit Rückgabewert `bool` heißt auch **Prädikat**.

```
bool IstGerade( int n)  
{  
    if ( n%2 == 0)  
        return true;  
    else  
        return false;  
}
```

Funktionen – Beispiele

```
1 double quad( double x)
2 { // berechnet Quadrat von x
3   return x*x;
4 }
5
6 double hypotenuse( double a, double b)
7 { // berechnet Hypotenuse zu Katheten a, b (Pythagoras)
8   return std::sqrt( quad(a) + quad(b) );
9 }
10
11 int main()
12 {
13   double a, b;
14   cout << "Laenge der beiden Katheten: ";   cin >> a >> b;
15
16   double hypoth= hypotenuse( a, b);
17   cout << "Laenge der Hypotenuse=" << hypoth << endl;
18   return 0;
19 }
```

- Variablen `a`, `b` in `main` haben nichts mit Var. `a`, `b` in `hypothenuse` zu tun: Scope ist lokal in der jeweiligen Funktion.
- **Grundsätzlicher Rat:** Variablen nur in Funktionen deklarieren (**lokal**), *nie* ausserhalb (**global**) bis auf wenige Ausnahmen!
- Aufruf einer Funktion nur *nach* deren Deklaration/Definition möglich:

```
double quad( double x); // Deklaration der Funktion quad

// ab hier darf die Funktion quad benutzt werden

double hypothenuse( double a, double b)
{ // berechnet Hypothenuse zu Katheten a, b (Pythagoras)
  return std::sqrt( quad(a) + quad(b) );
}

double quad( double x) // Definition der Funktion quad
{ // berechnet Quadrat von x
  return x*x;
}
```

Wertparameter (call by value)

Diese Funktion tut nicht, was sie soll:

- Funktionsdefinition

```
void tausche( double x, double y)
{
    double tmp= x;
    x= y;
    y= tmp;
}
```

- Aufruf

```
double a=5, b=7;
tausche( a, b);
cout << "a=" << a << ", b=" << b << endl;
// liefert: a=5, b=7
```

- **Grund:** Es werden nur die **Kopien** x, y getauscht!
a, b werden nicht verändert.

Referenzparameter (call by reference)

Abhilfe: Referenzparameter

- Funktionsdefinition mit Referenzparametern (beachte die `&` !)

```
void tausche( double& x, double& y)
{
    double tmp= x;
    x= y;
    y= tmp;
}
```

- Was passiert beim Aufruf `tausche( a, b);` ?
 - Parameter `x`, `y` werden angelegt als **Alias** der Argumente `a`, `b`. (*call by reference*)
 - Beim Tausch von `x`, `y` werden auch `a`, `b` verändert.

~> liefert nun: `a=7`, `b=5`.

- Funktionsdefinition mit Wert- und Referenzparameter

```
void tausche( double x, double& y)
{
    double tmp= x;
    x= y;
    y= tmp;
}
```

- Aufruf

```
double a=5, b=7;
tausche( a, b);
cout << "a=" << a << ", b=" << b << endl;
```

- Was wird ausgegeben? Warum?

- **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!

Namensraum (2)

- Warnendes Beispiel:

```
1 #include <iostream>
2 using namespace std;
3
4 void max( double a, double b)
5 {
6     double maxi= a>b ? a : b;
7     cout << "Das Maximum ist " << maxi << endl;
8 }
9
10 int main()
11 {
12     max( 3, 4);      // ruft std::max auf
13     return 0;
14 }
```

- Besser: `using std::cout; using std::endl;`
statt `using namespace std;`

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

Beispiel: $(11010)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
 $= 16 + 8 + 2 = 26.$

- `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:** ($m = 6$)

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

Gleitkommazahlen: Rundungsfehler

- viele $x \in \mathbb{R}$ nicht exakt darstellbar, z.B. $x = 0.1 \notin \mathbb{M} \leadsto$ **Rundung** zu $\hat{x} \in \mathbb{M}$
`double a = 0.1; // a ist nicht exakt 0.1`
- 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

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

$$\Rightarrow \hat{x} = x \cdot (1 + \varepsilon) \quad \text{mit } |\varepsilon| \leq \text{eps.}$$

- **Maschinengenauigkeit** eps:
 - für `double` ist $\text{eps} = 2^{-53} \approx 10^{-16}$,
 - für `float` ist $\text{eps} = 2^{-24} \approx 6 \cdot 10^{-8}$ (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 $\frac{0}{11}$, $\frac{1}{11}$, ... bis $\frac{11}{11} = 1$ zählen, also 12 Durchläufe, **aber**
 - $\frac{1}{11}$ im Rechner nicht exakt darstellbar → 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 Schleifenvariablen immer ganzzahlig wählen:

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

- 2 Vergleich robust für Rundungsfehler machen:

```
for (double x=0; x <= 1.0 + 1e-8; x+= elftel)
    cout << "x_=" << x << 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 )
{
    ...
}
```