C++ Teil 3

Sven Groß

2. Nov 2015

Themen der letzten Vorlesung

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

Heutige Themen

1. Funktionen
   - Definition und Aufruf
   - Wert-/Referenzparameter
2. Lesbarkeit von Code
3. Namensräume

Funktionen

- Beispiele von Funktionen:

```
double sqrt( double x)
```

```
double pow( double basis, double exp)
```

Rückgabetyp Name (formale Parameterliste)

- Aufruf:

```
sqrt( 4.0);
```
$\rightarrow$ liefert `2.0`

```
pow( x, 2.0);
```
$\rightarrow$ liefert x^2

Name (Argumentliste)

- Funktionen sind Sinneinheiten für Teilprobleme (Übersichtlichkeit, Struktur, Wiederverwendbarkeit)

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

  $\rightarrow$ 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 hypothenuse( double a, double b)
7  { // berechnet Hypothenuse 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= hypothenuse( a, b);
17   cout << "Laenge␣der␣Hypothenuse␣=␣" << hypoth << endl;
18   return 0;
19 }
```

Funktionen – Hinweise

- 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.

$\rightsquigarrow$ liefert nun: a=7, b=5.

Wert- und Referenzparameter – Quiz

- 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?

Lesbarkeit von Code – gute Beispiele

Fünfmal derselbe Code:

1. gut lesbar durch **Einrückung**, **Kommentare** und genug **Leerzeichen**:

```
if ( a*a - 2*a == 0 ) // nur erfuellt, falls a 0 oder 2
{
    cout << "Null␣oder␣Zwei:␣" << a << endl;
}
else
{
    cout << "weder␣Null␣noch␣Zwei:␣" << a << endl;
}
cout << "Bye!\n";      // Abschied
```

2. etwas kompakter, aber immer noch gut lesbar:

```
if ( a*a - 2*a == 0 ) { // nur erfuellt, falls a 0 oder 2
    cout << "Null␣oder␣Zwei:␣" << a << endl;
}
else {
    cout << "weder␣Null␣noch␣Zwei:␣" << a << endl;
}
cout << "Bye!\n";      // Abschied
```

Lesbarkeit von Code – so bitte *nicht!*

Fünfmal derselbe Code:

3. Falsche Einrückung kann auch verwirren:

```
if ( a*a - 2*a == 0 ) {
    cout << "Null oder Zwei: " << a << endl;
}
else
    cout << "weder Null noch Zwei: " << a << endl;
    cout << "Bye!\n";   // missverstaendlich!!!
```

4. Das ist schwer zu verstehen und fehleranfällig: $\rightsquigarrow$ `{ }`, **mehr einrücken!**

```
if ( a*a - 2*a == 0 )
cout << "Null oder Zwei: " << a << endl;
 else
 cout << "weder Null noch Zwei: " << a << endl;
  cout << "Bye!\n";
```

5. Der Compiler versteht auch das, du auch?

```
if(a*a-2*a==0){cout<<"Null oder Zwei: "<<a<<endl;}else
cout<<"weder Null noch Zwei: "<<a<<endl;cout<<"Bye!\n";
```

Lesbarkeit von Code – Empfehlungen

Lesbarkeit extrem wichtig, insbesondere für Fehlersuche!

- Code **einrücken** (Empfehlung: 4 Leerzeichen)
  $\rightsquigarrow$ **Tab**-Taste in `Code::Blocks` benutzen
- öffnende und schließende **Blockklammern** `{ }` untereinander
- genug **Leerzeichen** als Trenner zwischen Operatoren,
  z.B. `a= 17 + 2*3;` statt `a=17+2*3;`
- **Kommentare** (lieber zuviel als zuwenig),
  um eigenen Code auch noch in 2 Wochen zu verstehen

```
if ( a*a - 2*a == 0 ) // nur erfuellt, falls a 0 oder 2
{
    cout << "Null␣oder␣Zwei:␣" << a << endl;
}
else
{
    cout << "weder␣Null␣noch␣Zwei:␣" << a << endl;
}
cout << "Bye!\n";      // Abschied
```

Namensraum

- **Namensraum** kapselt Bezeichner (z.B. Variablen- und Funktionsnamen)
  $\rightsquigarrow$ 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;`