C++ Teil 2

Sven Groß

2. Nov 2020

Themen der letzten Vorlesung

- Hallo Welt
- Elementare Datentypen
- Ein-/Ausgabe
- Operatoren
- Kontrollstrukturen
  - bedingte Verzweigung: `if ... else`

Heutige Themen

1. Anmerkungen zu `cin`
2. Kontrollstrukturen
   - Verzweigungen
   - Schleifen
3. Weitere Operatoren
4. Funktionen

Anmerkungen zu `cin`

- Kennen wir schon bei `cout`:

```
int k= 42;
cout << "Die␣Antwort␣ist";
cout << k;
cout << endl;
```

ist gleichbedeutend mit

```
int k= 42;
cout << "Die␣Antwort␣ist" << k << endl;
```

- Ähnlich bei `cin`:

```
int a, b;
cin >> a;
cin >> b;
```

ist gleichbedeutend mit

```
int a, b;
cin >> a >> b;
```

Anmerkungen zu cin (2)

- `cin` liest aus dem **Eingabestrom** (i.d.R. Tastatureingaben)
- `cin` liest per Eingabeoperator `>>` höchstens bis zum nächsten **Whitespace** (Leerzeichen, Tabulator, Zeilenumbruch)
- je nach einzulesendem Datentyp werden Eingaben unterschiedlich interpretiert

```
int k;
string s;
// Eingabe:  5  11erRaus
cin >> k >> s;  // liest k = 5,   s = "11erRaus"

// Eingabe:  5  11erRaus
cin >> s >> k;  // liest s = "5",   k = 11
cin >> s;       // liest s = "erRaus"
```

- **Quiz:** Was passiert bei Eingabe von `1,82` ?

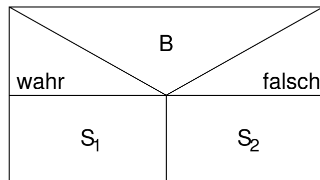
```
double laenge;
cin >> laenge;
```

Kontrollstrukturen: bedingte Verzweigung

- `if-else`-Verzweigung:

```
if (i < 0)
{ // Bedingung erfuellt
    cout << "i␣ist␣negativ"
         << endl;
}
else
{ // Bedingung nicht erfuellt
    cout << "i␣ist␣nicht␣negativ"
         << endl;
}
```

Nassi-Shneiderman-Diagramm:



- kein `;` nach `if` oder `else` !
- `else`-Teil kann auch entfallen

if-else-Verzweigung: Quiz

Quiz für menschliche Compiler:

- Wer findet den Fehler?
- Was wäre passiert, wenn der `else`-Teil nicht da wäre?

(Verwirrender) Hinweis: Der `g++`-Compiler meckert über das `else`.

```
1  if (i < 0);
2  { // Bedingung erfuellt
3      cout << "i ist negativ"
4           << endl;
5  }
6  else
7  { // Bedingung nicht erfuellt
8      cout << "i ist nicht negativ"
9           << endl;
10 }
```

Bedingte Verzweigung (2)

- Fallunterscheidung mit `if-else`-Verzweigung:

```
if (i==1)
    cout << "Eins\n";
else if (i==2)
    cout << "Zwei\n";
else if (i==3)
    cout << "Drei\n";
else
    cout << "???\n";
```

- Alternativ mit `switch-case`-Verzweigung (`break` nicht vergessen!):

```
switch (i)
{
  case 1:  cout << "Eins\n"; break;
  case 2:  cout << "Zwei\n"; break;
  case 3:  cout << "Drei\n"; break;
  default: cout << "???!\n";
}
```

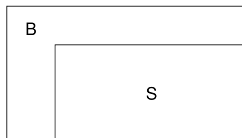
A			
w_1	w_2	...	sonst
S_1	S_2	...	S

Schleifen: `while`-Schleife

Schleifen werden gebraucht, um Anweisungsblöcke mehrfach zu wiederholen.

- Solange Bedingung B erfüllt ist, wiederhole Block S.

```
while (B)
{
  // Anweisungen aus S
}
```



- abweisende Schleife: erst B testen, dann (evtl.) S ausführen.
- kein `;` nach `while (B)` !
  Sonst wird nur `;` (leere Anweisung) wiederholt!

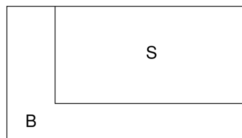
```
int i=1;

while ( i<10 )
{
    cout << i << "␣";
    ++i;  // kurz fuer i= i+1;
}
```

Schleifen: do...while-Schleife

- Variante: **nicht abweisende Schleife**

```
do
{
  // Anweisungen aus S
} while (B);
```



- Erst S ausführen, dann B testen.
- Schleife wird *mindestens einmal* durchlaufen.
- kein `;` nach `do` !

```
int i=1;

do
{
    cout << i << "␣";
    i= i*3;
} while ( i<100 );
```

wichtigste Schleife: `for`-Schleife

- Oft braucht man Schleifen, die genau `n`-mal durchlaufen werden.

```
int i= 1;
while (i <= n)
{
  // Anweisungen...

  i++;   // kurz fuer i= i+1;
}
```

- `i` heißt Zählvariable.
- Kurzform (sehr praktisch): **`for`-Schleife**

`for` (*Initialisierung; Bedingung; Zähleränderung*)

```
for (int i= 1; i <= n; i++)
{
  // Anweisungen...
}
```

for-Schleife – Beispiel

```
// wir berechnen die Summe der Zahlen 1, 2, ..., 100

int sum= 0;

for (int i=1; i<=100; ++i)
{
    sum= sum + i;
    cout << sum << "␣";
}

// cout << i;    // Fehler: i nicht definiert!

cout << "\nSumme␣=␣" << sum << endl;
```

- Ausgabe:

```
1 3 6 10 15 21 28 ...
Summe = 5050
```

- Scope der Zählvariable ist der Schleifenblock.

Schleifenabbruch mit break und continue

- `break` bricht die Schleife ab,
  Code nach der Schleife wird als Nächstes bearbeitet.
- `continue` bricht nur den aktuellen Schleifendurchlauf ab,
  es wird mit nächstem Schleifendurchlauf fortgefahren.

Beispiel: Es wird wiederholt der Kehrwert von eingegebenen Zahlen berechnet.
Null als Eingabe wird ignoriert.
Durch Eingabe von 99 wird die Berechnung abgebrochen.

```
1   while (true) {  // Endlosschleife
2       int x;
3       cout << "Wert: ";  cin >> x;
4
5       if (x==0)   // ignorieren: 1/x nicht def.
6           continue;
7       cout << "Kehrwert: " << ( 1.0/x ) << endl;
8       if (x==99)  // Abbruch
9           break;
10  }
11  cout << "Tschoe wa!" << endl;  // Aachener Abschiedsgruss
```

Weitere Operatoren

Operatoren in C++

Zuweisung	`=`	`j = j + 3`	
	`+= -= *= /=`	`j += 3`	
In-/Dekrement	`++ --`	`j++ ++j`	
Auswahl	`?:`	`cond ? val1 : val2`	(3 Argumente)

- **Zuweisung**: `j*= 5;` als kürzere Schreibweise für `j= j*5;`
- **Inkrement**: `++j;` als kürzere Schreibweise für `j= j+1;` bzw. `j+= 1;`
- **Dekrement**: `--j;` als kürzere Schreibweise für `j= j-1;` bzw. `j-= 1;`
- **Auswahl**: `betrag= (j >= 0) ? j : -j;` als kürzere Schreibweise für

```
if (j >= 0)
    betrag= j;
else
    betrag= -j;
```

Weitere Operatoren – Beispiele

- **Zuweisung**:

```
double x= 1;
x+= 5;   // wie x= x+5, also x ist 6
x*= 5;   // wie x= x*5, also x ist 30
x/= 4;   // wie x= x/4, also x ist 7.5
```

- **Inkrement**: unterschiedliche Rückgabe der Prä-/Postfix-Variante

```
int a= 0, b= 0;
cout << "Rueckgabe: Postfix " << a++
     <<            ", Praefix " << ++b << endl;
cout << "Nachher: a = " << a
     <<          ", b = " << b << endl;
```

liefert Ausgabe

```
Rueckgabe: Postfix 0, Praefix 1
Nachher: a = 1, b = 1
```

Noch mehr Beispiele

- Diese Schleife zählt in Fünferschritten:

```
for (int i= 1; i <= 100; i+= 5)
{
    cout << i << endl;
}
```

- Diese Schleife zählt rückwärts:

```
for (int i= 10; i >= 1; --i)
{
    cout << i << endl;
}
```

- Quiz: Was macht diese Schleife?

```
for (int i= 1; i <= 10; ++i)
{
    cout << i << endl;
    i*= 2;
}
```

Funktionen

- Beispiele von Funktionen:

```
double sqrt( double x)

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

  Rückgabetyp Name (formale Parameterliste)

- Aufruf:

`sqrt( 4.0);`	→ liefert `2.0`
`pow( x, 2.0);`	→ 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

  → werden beim Verlassen des Rumpfes zerstört

Funktionsaufruf

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

int main()      // ist auch eine Funktion...
{
    double a= 4.0, b= 8.0, result;
    result= MW( a, b);
    return 0;
}
```

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

```cpp
double quad( double x)
{ // berechnet Quadrat von x
  return x*x;
}

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

int main()
{
  double a, b;
  cout << "Laenge der beiden Katheten: ";  cin >> a >> b;

  double hypoth= hypothenuse( a, b);
  return 0;
}
```

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;
}
```