KAPITEL 4

Anweisungen

C++ kennt viele verschiedene Arten von Anweisungen. Es gibt zum einen Anweisungen, die Ausdrücke auswerten, und zum anderen Anweisungen, die die Reihenfolge der zukünftigen Anweisungen ändern.

Ausdrucksanweisungen

Eine *Ausdrucksanweisung*(Expression Statement) ist ein Ausdruck, gefolgt von einem einzelnen Semikolon (`;`). Ausdrucksanweisungen führen dazu, dass ein Ausdruck ausgewertet wird. Nebeneffekte, wie die Zuweisung an eine Variable, werden abgeschlossen, bevor die nächste Anweisung ausgeführt wird:

```
a= 10;
```

Null-Anweisungen

Eine *Null-Anweisung* (Null Statement) wird als Semikolon (`;`) geschrieben. Null-Anweisungen sind nützlich, wenn die C++-Syntax eine Anweisung verlangt, aber nichts auszuführen ist:

```
void spin(int n){
  for (int i = 0; i < n; ++i);
}
```

Diese Schleife zählt einfach bis zum angegebenen Wert. Das kann z. B. zum Einfügen einer Verzögerung in einem Echtzeitsystem notwendig sein. Dies gilt natürlich nur unter der Annahme, dass der Compiler die Schleife nicht ohnehin völlig wegoptimiert.

Zusammengesetzte Anweisungen

Eine *zusammengesetzte Anweisung* (Compound Statement) ist eine Gruppe von Anweisungen, die auch leer sein kann. Sie beginnt mit einer linken geschweiften Klammer ({) und endet mit einer rechten geschweiften Klammer (}):

```
while (true){
  // Anfang einer zusammengesetzten Anweisung.
  ...
  if (!done){
      // Noch eine zusammengesetzte Anweisung.
  }

else{
     // Noch eine zusammengesetzte Anweisung.
  }
}
```

Zusammengesetzte Anweisungen werden oft *Blöcke* genannt. Ein Block definiert einen Bereich mit einem eigenen lokalen Geltungs- und Sichtbarkeitsbereich.

Iterationsanweisungen (Schleifen)

Iterationsanweisungen veranlassen die wiederholte Ausführung einer Anweisung oder eines Blocks. C++ bietet drei Arten von Iterationsanweisungen an: `while`, `do while` und `for`. Die `for`-Anweisung gibt es neben der klassischen Form in einer weiteren Variation: als Range-basierte `for`-Anweisung.

while

Eine `while`-Schleife wiederholt eine Anweisung oder einen Block, solange ein am Anfang der Schleife ausgewerteter Ausdruck – der auch eine Deklaration sein kann – `true` ergibt:

```
char ch = 'y';

while (ch == 'y'){
  // Irgendetwas wiederholt tun.
  ...
```

```
  std::cout << "Do it again (y or n)? ";
  std::cin  >> ch;
}
```

Der Block wird so lange wiederholt, wie `ch` gleich `'y'` ist. Wird der Ausdruck am Schleifenanfang schon beim Eintritt in die Schleife zu `false` ausgewertet, wird der Schleifenkörper nie ausgeführt.

do while

Eine `do while`-Schleife wiederholt eine Anweisung oder einen Block so lange, wie der am Ende der Schleife ausgewertete Ausdruck `true` ergibt:

```
char ch;

do{
  // Irgendetwas wiederholt tun.
  ...

  std::cout << "Once more (y or n)? ";
  std::cin  >> ch;
} while (ch == 'y');
```

Der Block wird hier so lange wiederholt, wie `ch` gleich `'y'` ist. Allerdings wird der Schleifenkörper immer mindestens einmal ausgeführt, da die Schleifenbedingung erst am Ende jeder Iteration ausgewertet wird.

for

Eine `for`-Schleife ähnelt einer `while`-Schleife, jedoch gibt es darüber hinaus einen zusätzlichen Mechanismus zum Initialisieren der Schleife und zum Aktualisieren von Zählern am Ende jeder Iteration:

```
typedef std::map<int, std::string> IntStringMap;

IntStringMap m;
char s[4];

for (auto i= 0; i < 10; ++i){
  s[0]= 'a' + i;
```

```
    s[1]= 'b' + i;
    s[2]= 'c' + i;
    s[3]= '\0';

    m.insert(std::make_pair(i,s));
  }
```

Um for-Schleifen zu verstehen, müssen Sie wissen, welche Anweisungen in die Klammern hinter dem Schlüsselwort for gehören. Die Anweisung vor dem ersten Semikolon (auto i= 0) dient zum Initialisieren der Schleife. Der Ausdruck zwischen den beiden Semikola (i < 10) stellt die Bedingung dar, die vor jeder Iteration ausgewertet wird. Der Schleifenkörper kommt nur dann zur Anwendung, wenn der Ausdruck wahr ergibt. Ansonsten wird die Schleife beendet. Nach jeder Iteration wird der ganz rechts stehende Ausdruck (++i) ausgewertet. Nun startet die for-Schleife wieder von Neuem. for-Schleifen können auch kompliziertere Ausdrücke enthalten:

```
void upperString(char* t, const char* s){
  for (; *s != '\0'; *(t++)= std::toupper(*(s++)));
   *(t++) = '\0';
}
```

Diese Funktion benutzt eine for-Schleife, um den C-String s in Großbuchstaben zu konvertieren und nach t zu kopieren. Dabei geht die Funktion davon aus, dass der Speicherplatz für t bereits alloziert wurde. Die Initialisierung ist eine Null-Anweisung, denn s und t sind beim Aufruf der Funktion bereits initialisiert. Auch der Schleifenkörper besteht nur aus einer Null-Anweisung.

Ein in der Initialisierungsanweisung einer for-Schleife deklarierter Name ist bis zum Ende der Schleife sichtbar.

Range-basierte for-Anweisung

Die Range-basierte for-Anweisung besitzt die allgemeine Form:

```
for (Deklaration: Ausdruck){ Anweisungen
}
```

Dabei muss der Ausdruck eine Sequenz `seq` sein, auf der entweder `seq.begin()` und `seq.end()` oder die Funktionen `begin(seq)` und `end(seq)` so aufgerufen werden können, dass sie Iteratoren zurückgeben. Dies trifft auf {}-Initialisierungslisten, auf C-Arrays, auf die C++-Strings und alle Container der Standard Template Library zu. Die Range-basierte `for`-Anweisung bietet die gleiche Funktionalität wie die `for`-Anweisung in sehr kompakter Form. Kombiniert mit der automatischen Typableitung (`auto`), lässt sich damit ein Vektor oder eine Map direkt ausgeben:

```
std::vector<int> vec{1,2,3,4,5};
for (auto v: vec) std::cout << v << " ";   // 1 2 3 4 5

std::map<std::string,int> map{{"Scott",1976},
                              {"Dijkstra",1972}};
for (auto m:map) std::cout << m.first << ": "
                           << m.second << std::endl;
// Scott: 1976
// Dijkstra: 1972
```

Werden die Argumente der Sequenz per Referenz angenommen, lassen sich diese modifizieren:

```
int array[5]={1,2,3,4,5};
for (int& a: array) a *=2 ;
for (auto a: array) std::cout << a << " ";   // 2 4 6 8 10

std::string test{"Only for Testing Purpose."};
for (auto& c: test) c = std::toupper(c);
for (auto c: test) std::cout << c; // ONLY FOR TESTING PURPOSE.
}
```

Verzweigungen

Verzweigungen oder Auswahlanweisungen führen je nach Ergebnis eines Ausdrucks unterschiedliche Anweisungen oder Blöcke aus. Es gibt in C++ zwei verschiedene Auswahlanweisungen: `if` und `switch`.

if

Eine `if`-Anweisung wertet einen Ausdruck aus. Dieser Ausdruck kann auch eine Deklaration sein, die gegebenenfalls im `else`-Zweig

sichtbar ist. Abhängig vom Ergebnis des Ausdrucks wird eine von zwei Anweisungen oder Blöcken anschließend ausgeführt:

```
if (i > 0 && i < 100){
  // Mach irgendetwas, wenn im Bereich.
}
else{
  // Mach irgendetwas, wenn nicht im Bereich.
}
```

Wenn die Auswertung des Ausdrucks `true` ergibt, kommt der Code direkt nach der schließenden Klammer des `if`-Abschnitts zum Einsatz, ansonsten der nach dem `else`. Da die `else`-Klausel optional ist, wird nichts ausgeführt, falls der Ausdruck zu `false` evaluiert. Wenn `if`-Anweisungen ineinander eingebettet sind, gehören `else`-Klauseln immer zum nächstgelegenen `if`.

Seit *C++17* kann eine Variable direkt in der `if`-Anweisung initialisiert werden.

```
std::map<int, std::string> myMap;

if (auto result = myMap.insert(value); result.second){
    useResult(result.first);
}
else{
    // ...
} // result wird automatisch destruiert
```

Die Variable `result` ist nur innerhalb des `if`- und des `else`-Zweigs der `if`-Anweisung gültig. `result` wird nicht in den umgebenden Bereich eingeführt.

constexpr if

In *C++17* erlaubt es `constexpr if`, Sourcecode bedingt zu übersetzen.

```
template <typename T>
auto getAnswer(T){
  static_assert(std::is_arithmetic_v<T>); // arithmetisch
  if constexpr (std::is_integral_v<T>)    // Ganzzahl
    return 42;
  else                                    // Fließkommazahl
    return 42.0;
```

```
}

...

std::cout << getAnswer(5);      // 42
std::cout << getAnswer(5.5);    // 42.0
```

Das Funktions-Template `getAnswer` lässt sich nur mit einem arithmetischen Datentyp aufrufen und gibt zwei Werte von verschiedenen Datentypen zurück. Falls der Typ-Parameter `T` eine Ganzzahl ist, gibt die Funktion einen `int`-Wert zurück, sonst einen `double`-Wert.

Der nicht ausgeführte Zweig einer `constexpr if`-Anweisung muss gültig sein.

switch

Eine `switch`-Anweisung wählt einen Codeabschnitt aus mehreren anhand des Werts eines Steuerausdrucks aus:

```
switch (key){
  case keyDown:
     // Tu etwas, wenn Taste gedrueckt wird.
     ...
     break;

  case keyUp:
     // Tu etwas, wenn Taste losgelassen wird.
     ...
     break;
  ...
  default:
     // Alles, was noch nicht behandelt wurde.
     ...
}
```

Vor jedem Abschnitt steht das Schlüsselwort `case`, gefolgt von einem Ausdruck. Dieser Ausdruck muss zur Compilezeit zu einem eindeutigen, konstanten und integralen Wert ausgewertet werden können. Zur Laufzeit verzweigt die Ausführung zu dem Abschnitt, der dem Steuerausdruck entspricht, und fährt hier fort. Eine `break`-Anweisung am Ende jedes Abschnitts stellt sicher, dass auch der

Code, der zu den verbleibenden Fällen gehört, nicht ausgeführt wird. Darüber hinaus kann ein optionaler `default`-Fall angegeben werden. Dieser wird ausgerufen, wenn der Wert des Steuerausdrucks auf keinen der anderen Fälle zutrifft.

Seit *C++17* kann eine Variable direkt in der `switch`-Anweisung initialisiert werden.

Sprunganweisungen

Sprunganweisungen springen bedingungslos zu einer anderen Anweisung. Es gibt vier verschiedene Arten von Sprunganweisungen in C++: `break`, `continue`, `goto` und `return`.

break

Eine `break`-Anweisung dient dazu, aus der innersten Schleife oder einer `switch`-Anweisung herauszuspringen:

```
for (;;){
  if (done) break;

  // Wenn fertig, setze done auf true, damit
  // der naechste Durchlauf abgebrochen wird.
  ...
}
```

Dies ist eine `for`-Schleife ohne Abbruchbedingung. Eine `break`-Anweisung dient dazu, die Schleife abzubrechen, wenn `done` den Wert `true` hat.

continue

Eine `continue`-Anweisung wird dazu verwendet, an den Anfang der innersten umgebenden Schleife zu springen. Damit wird der restliche Code der aktuellen Schleife übersprungen und der nächste Schleifendurchlauf unmittelbar aufgenommen:

```
while (!done){
  // Wenn an das Ende der Funktion gesprungen
  // werden soll, setze skip auf true.
```

```
    ...
    if (skip) continue;

    // Dies hier wird uebersprungen.
     ...
}
```

Dies ist eine `while`-Schleife, die Anweisungen enthält. Diese können übersprungen werden, wenn `skip` auf `true` gesetzt worden ist. Eine `continue`-Anweisung dient dazu, an den Anfang der Schleife zu springen, um die restlichen Anweisungen zu übergehen.

goto

Eine `goto`-Anweisung springt zu einer expliziten Sprungmarke:

```
if (getLastError() != ERROR_SUCCESS) goto HandleError;

// Dieser Code wird im Fehlerfall uebersprungen.
...

HandleError:
// Setze hier Fehlerbehandlung ein.
```

Da `goto`-Anweisungen zu unstrukturiertem Code führen können, werden sie selten verwendet.

Die Verwendung von `goto`-Anweisungen gilt als sehr schlechter Stil. Daher sollten sie – falls überhaupt – nur sehr gezielt eingesetzt werden. `goto`-Anweisungen werden bisweilen genutzt, um aus sehr tief eingebetteten Iterations- oder Sprunganweisungen direkt herauszuspringen. Natürlich lassen sich solch komplexe Strukturen auch ohne `goto`-Anweisungen auflösen.

return

Eine `return`-Anweisung springt aus einer Funktion heraus und setzt bei Bedarf einen Rückgabewert:

```
double convertToSM(double nm){
  return nm * NMPerSM;
}
```

return-Anweisungen können an beliebiger Stelle in einer Funktion verwendet werden. Der Typ des Rückgabewerts muss dem Rückgabetyp der Funktion entsprechen oder in diesen konvertierbar sein. Funktionen, die void zurückgeben, benötigen keine return-Anweisungen:

```
void sayHello(){
  std::cout << "Hallo" <<std::endl;
}
```

Der Rücksprung aus einer Funktion wird an ihrem Ende automatisch vollzogen, wenn der Funktionskörper abgearbeitet ist. return-Anweisungen ohne Wert können dazu verwendet werden, vorzeitig aus der Funktion herauszuspringen.