Vorlesung

Objektorientiertes Programmieren in C++

Teil 2 / 2020

Detlef Wilkening
www.wilkening-online.de

5 Typen & Variablen

5.1 Typen

Ein Datentyp (auch kurz nur Typ genannt) beschreibt eine bestimmte Art von Objekten (z.B. Ganzzahlen), und definiert für den Compiler, wie sie zu behandeln sind, wieviel Speicher sie benötigen, und welche Operationen auf ihnen ausführbar sind.

Es wird unterschieden zwischen:

- Elementare Datentypen – auch eingebaute Datentypen genannt, wie z.B, „bool“, „int“ oder „double“. Sie sind Bestandteil der Sprache.
- Basis-Datentpyen – ein Oberbegriff für alle elementaren Datentypen, Aufzählungen (auch Enums genannt) und Zeigern.
- Benutzerdefinierten Datentypen – diese werden vom „Benutzer“ mit den C++ Sprachmitteln definiert (z.B. mit „enum“ oder „class“, wie z.B. die Klasse „std::string“.

5.1.1 Elementare Datentypen

Es gibt in C++20 insgesamt 19 elementare Datentypen. Wir beschränken uns bei den

elementaren Typen erstmal auf die vier gebräuchlichsten Typen ‚bool', ‚int', ‚double' und ‚char'. Natürlich sind in der Praxis auch alle anderen Typen wichtig – aber wir wollen uns hier für den Einstieg beschränken. Natürlich sollten Sie auch dieses Thema auf Dauer vertiefen.

Typ	**Beschreibung**	**Allgemeines**
bool	Wahrheitswert	Erlaubte Werte sind true und false (beides Schlüsselworte)
int	vorzeichenbehaftete Ganzzahlen	Typischerweise auf heutigen Rechnern 2, 4 oder 8 Byte groß. 4 Byte große int's umfassen dann meist den Wertebereich von: -2147483648 .. +2147483647 Aber Achtung – die genaue Größe, und der exakte Wertebereich ist Compiler- und Plattformspezifisch – der Standard legt nur einen gewissen Rahmen fest.
double	Fließkomma-Zahl	Typischerweise auf heutigen Rechnern 8 Byte groß und nach IEEE754 Standard codiert. Aber Achtung – auch hier gilt: die genaue Größe, der exakte Wertebereich, und die Codierung sind compiler- und plattformspezifisch – der Standard legt nur einen gewissen Rahmen fest.
char	Zeichen	Genau 1 Byte groß (per C++ Definition). Achtung: • In C++ ist ein Byte nicht zwingend 8 Bit groß, dass ist plattform-spezifisch. • In C++ ist der Name ‚char' KEIN Kurzform-Name des Daten-Typs ‚signed char' wie in C. In C++ sind ‚char', ‚signed char' und ‚unsigned char' drei unterschiedliche Datentypen.

Alle elementaren Datentypen in C++20 (ohne nähere Erklärung ihrer Bedeutung, Größe):

- „bool"
  => Typ für die Wahrheitswerte „true" und „false"
- „char", „wchar_t", „char8_t", „char16_t" und „char32_t"
  => Zeichen-Typen und gleichzeitig auch integrale Typen
- „signed char", „unsigned char", „short", „unsigned short", „int", „unsigned int", „long", „unsigned long", „long long" und „unsigned long long"
  => Integrale Typen
- float, double, long double
  => Fließkomma-Typen

Achtung – die genaue Größe und interne Bit-Repräsentation aller elementaren Datentypen ist in C++ nicht festgelegt, sondern vom Compiler, Prozessor und dem Betriebssytem abhängig. So existieren in der Realität mehrere unterschiedliche Bit-Repräsentation für integrale Typen oder auch Fließkomma-Typen. Schreiben Sie Programme, die unabhängig von der genauen Bit-Repräsentation sind.

5.2 Literale

- Literale sind feste Werte im Quelltext, die auch immer implizit einen Typ haben. Es gibt:
  - Ganzzahlen
  - Gleitkommazahlen
  - Zeichen-Konstanten
  - Zeichenketten-Konstanten

- Ganzzahlen werden einfach dezimal hingeschrieben, z.B.: 123 0 89 34562 -897
  Ihr Typ ist int, wenn nicht anders definiert (Es gibt eine spezielle Postfix-Notation für z.B. long Ganzzahlen.) Sie können auch oktal (mit führender Null), hexadezimal (mit führendem „0x“) und seit C++14 auch binär (mit führendem „0b“) geschrieben werden.

- Gleitkommazahlen enthalten einen Dezimalpunkt, z.B.: 0.1 .4 31.459
  Ihr Typ ist „double“, wenn nicht anders definiert (Postfix-Notation mit z.B. „f“ für „float“).
  Es gibt auch eine wissenschaftliche Schreibweise für die 10^ Notation, z.B. „1.2E+12“

- Zeichen-Konstanten stehen in einfachen Hochkommata und sind vom Typ „char“.
  Sie dürfen Backslash-Sequenzen (s.o.) enthalten, z.B.: 'A' '. ' '\\' '\n'
  Bzgl. der Kodierung wird einfach der der Zeichensatz der Maschine genommen, oder Sie nutzen explizit Unicode.

- Zeichenketten-Konstanten stehen normalerweise in doppelten Hochkommata und dürfen auch Backslash-Sequenzen enthalten, z.B.: "Hallo" "Dies ist ein Text". Der Typ von Zeichenketten-Konstanten ist ziemlich kompliziert, und wird hier daher noch nicht näher besprochen. Wichtig ist – eine Zeichenketten-Konstante ist kein „std::string“.

Hiermit können wir ein Programm schreiben, das feste Werte ausgeben kann.

```
#include <iostream>
using namespace std;

int main()
{
   cout << 123 << ' ' << 'H' << "allo\\ \"" << 3.1415 << "\"\n";
}
```

Ausgabe
```
123 Hallo\ "3.1415"
```

Aber auch zur Variablen-Initialisierung werden Literale häufig benutzt:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   int v = 42;
   string s("C++");           // (*)
   cout << v << s;
}
```

Ausgabe
```
42C++
```

5.3 Variablen

5.3.1 Variablen Definitionen

In C++03 gab es *„nur“* 4 Syntaxen um Variablen zu definieren – in C++11 sind einige weitere Syntaxen hinzugekommen. Sie unterscheiden sich durch die Art der Typ- und der Initialisierungs-Angabe.

5.3.1.1 Alte Defintions-Syntaxen

Diese vier Syntaxen gab es schon in C++03, und sind daher oft die Gebräuchlichsten – aber das könnte sich in Zukunft ändern.

```
typ name;
typ name = typ();
typ name = initial-wert;
typ name(initial-wert-oder-werte);
```

Sie beginnen alle mit dem Typ, dann folgt der Name und am Schluß die optionale Initialisierungs-Angabe mit Gleichheitszeichen oder Klammern – hier einige Beispiele:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   int n0 ;            // Erzeugt eine int Variable mit rein zufaelligem Startwert (*)
   cout << n0 << '\n';

   int n1 = int();                  // Erzeugt eine int Variable mit dem Wert '0' (**)
   cout << n1 << '\n';

   int n2 = 42;                     // Erzeugt eine int Variable mit dem Wert '42'
   cout << n2 << '\n';

   bool b = true;
   cout << b << '\n';                // Der bool-Wert "true" erzeugt die Ausgabe '1'

   double d(3.14);
   cout << d << '\n';

   char c(' ');
   cout << "->" << c << "<-" << '\n';

   string empty;                    // Leerstring
   cout << "->" << empty << "<-" << '\n';

   string name("C++");              // String mit Inhalt "C++"
   cout << name << '\n';

   string s(5, '*');                // String mit Inhalt "*****" - 5 x '*'
   cout << s << '\n';
}
```

```
Mögliche Ausgabe (da die erste Ausgabe rein zufällig ist)
865226374
0
42
1
3.14
-> <-
-><-
C++
*****
```

Hinweise:

- Der Header „string“ muß eingebunden werden, damit der Compiler den Typ „std::string“ kennt.
- Wird eine lokale Int-Variable ohne Startwert definiert – wie z.B. in Zeile (*) – so ist der Startwert der Variablen rein zufällig.
- Bei der Nutzung von „typ()“ zur Initialisierung – wie z.B. in Zeile (**) – ist bei den elementaren Datentypen wohldefiniert, dass sie mit „false“, „‘\0‘“, „0“ bzw. „0.0“ initialisiert werden.
- Im Normalfall wird der Bool-Wert „true“ als „1“ ausgegeben (siehe obiges Beispiel), und „false“ als „0“. Dies entspricht der aus C stammenden Konvention, dass „0“ „false“ und „1“ „true“ entspricht. Mit dem Manipulator „std::boolalpha“ kann die Ausgabe zu „false“ und „true“ geändert werden – siehe folgendes Beispiel.

```
#include <iostream>
using namespace std;

int main()
{
   cout << false << ' ' << true << '\n';                    // => 0 1
   cout << boolalpha << false << ' ' << true << '\n';       // => false true
}
```

```
Ausgabe
0 1
false true
```

5.3.1.2 Lokale Variablen-Definition ohne Initialwert

Die Definition einer lokalen Variable ohne Initialwert verhält sich bei manchen Typen unerwartet – z.B. bei allen elementaren Datentypen, Enums oder Zeigern. Sie erzeugt dann nämlich eine Variable mit rein zufälligem Startwert. Bei den meisten benutzerdefinierten Datentypen ist der Inital-Wert in einem solchen Fall aber wohldefiniert (z.B. bei “std::string”) – dies muß aber nicht so sein. Im Zweifelsfall befragen Sie die entsprechende Dokumentation.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   int n;                          // Achtung - der Startwert der Variable ist zufaellig
   cout << "->" << n << '\n';

   string empty;                   // Leerstring - Startwert bei "string" wohldefiniert
   cout << "->" << empty << "<-" << '\n';

   double d;                       // Achtung - der Startwert der Variable ist zufaellig
   if (empty=="")                  // Ist hier aber nicht schlimm, da die Variable "d"
   {                               // vor der Nutzung in jedem Fall initialisiert wird
      d = 0.5;
   }
   else
   {
      d = 2.5;
   }
   cout << d << '\n';
}
```

```
Mögliche Ausgabe (da Wert von n zufällig ist)
456223184
```

```
->0<-
0.5
```

Hinweis – bei statischen und globalen Variablen ist der Startwert aller Variablen genau definiert. Bei den elementaren Daten-Typen ist er dann „false“, „‘\0‘“, „0“ bzw. „0.0“.

Bei der Definition von lokalen Variablen ohne Initial-Wert sollte man genau wissen, was man macht, d.h. ob der Typ eine saubere Initialisierung vornimmt oder nicht, bzw. ob eine Initialisierung für den Programm-Ablauf notwendig ist oder nicht. Die Nutzung von nicht initialisierten Variablen führt immer Fehlern, die leider manchmal nicht sofort auffallen.

```
#include <iostream>
using namespace std;

int main()
{
   bool b = bool();                          // Wohldefiniert "false"
   cout << "bool " << b << '\n';             // -> 0

   char c = char();                          // Wohldefiniert "\0"
   cout << "char " << c << '\n';             // Zeichen mit Code "\0" ist nicht zu sehen

   short s = short();                        // Wohldefiniert "0"
   cout << "short " << s << '\n';

   unsigned long ul = unsigned long();       // Wohldefiniert "0"
   cout << "unsigned long " << ul << '\n';

   double d = double();                      // Wohldefiniert "0.0"
   cout << "double " << d << '\n';

   enum E { A, B } ;                         // Enums kommen erst spaeter
   E e = E();                                // Enum-Konstante mit dem Wert "0"
   cout << "enum " << e << '\n';

   cout << "Zeiger " << p << '\n';
}
```

Ausgabe

```
bool 0
char
short 0
unsigned long 0
double 0
enum 0
Zeiger 00000000
```

5.3.1.3 Variablen-Definition mit dem Zuweisungs-Operator

Die Syntax mit dem Zuweisungs-Operator „=“ ist nur bedingt zu empfehlen, da sie:
a) nicht mit mehreren Initialisierungswerten umgehen kann, und
b) nicht in allen Fällen das macht, was man meint.

Aber z.B. bei elementaren Datentypen ist sie problemlos nutzbar, weshalb sie dort aufgrund der besseren Lesbarkeit (oder ist es einfach nur Gewohnheit?) häufig eingesetzt wird.

```
#include <string>

int main()
{
   int n = 23;                    // Kein Problem, da elementarer Datentyp

   std::string s1 = "C++";        // Achtung, hier passiert mehr als man denkt
                                  // Details siehe spaeter
   std::string s2("C++");         // So ist es darum besser
}
```

Hinweis – es ist sehr sehr sehr empfehlenswert, für Variablen sprechende Namen zu wählen, die den Sinn und die Semantik der Variablen ausdrücken, In diesem Sinne sind im ersten Beispiel „name“ und „empty“ gute Variablen-Namen, während z.B. „s“, „s1“ und „s2“ nur als Beispiel-Namen aktzeptabel sind.

5.3.1.4 Vereinheitlichte Initialisierungs-Syntax in C++11

Um einige Fallen der alten C++03 Initialisierungs-Syntaxen zu umgehen, und neue Anwendungsfälle zu ermöglichen, hat man in C++11 eine neue vereinheitlichte Initialisierungs-Syntax mit geschweiften Klammern eingeführt.

Im Prinzip war die Idee, dass man einfach die alten Syntaxen jetzt auch mit geschweiften statt runden Klammern schreiben kann. Durch die neue Syntaxen mit den geschweiften Klammern fallen die alten Fallen weg, und man konnte neue Anwendungsfälle definieren (die vorher nicht möglich waren, da die Syntax schon belegt war).

So gut das an vielen Stellen funktioniert – und gerade die neuen Anwendungsfälle sind sehr hilfreich – leider hat es nicht 100% geklappt. Die neue Syntax mit den geschweiften Klammern hat zwei neue Fallen, eine wurde mit C++17 berichtigt.

Aber hier nun erstmal die neue vereinheitlichte Initialisierungs-Syntax von C++11:

```
typ name{};
typ name = {};
typ name{ initial-wert };
typ name({ initial-wert });
typ name = { initial-wert };
typ name{ typ() };
typ name({ typ() });
typ name = { typ() };
```

Und hier ein Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
   int n1{};                // Definiert "n1" mit dem Initial-Wert von "0"
   int n2 = {};             // Definiert "n2" mit dem Initial-Wert von "0"
   int n3{ int() };         // Definiert "n3" mit dem Initial-Wert von "0"
   int n4({ int() });       // Definiert "n4" mit dem Initial-Wert von "0"
   int n5 = { int() };      // Definiert "n5" mit dem Initial-Wert von "0"
   int n6{ 6 };
   int n7 = { 7 };

   cout << n1 << " - " << n2 << '\n';
   cout << n3 << " - " << n4 << " - " << n5 << '\n';
   cout << n6 << " - " << n7 << '\n';
}
```

```
Ausgabe
0 - 0
0 - 0 - 0
6 - 7
```

Das funktioniert nicht nur mit einem elementaren Daten-Typ wie „int“, sondern auch mit Klassen wie z.B. „std::string“.

```
#include <iostream>
```

```
using namespace std;

int main()
{
   string s1{};                 // Leerstring
   string s2 = {};              // Leerstring
   string s3{ string() };       // Leerstring
   string s4({ string() });     // Leerstring
   string s5 = { string() };    // Leerstring
   string s6{ "C++" };
   string s7 = { "C++" };

   cout << "->" << s1 << "<-" << '\n';
   cout << "->" << s2 << "<-" << '\n';
   cout << "->" << s3 << "<-" << '\n';
   cout << "->" << s4 << "<-" << '\n';
   cout << "->" << s5 << "<-" << '\n';
   cout << "->" << s6 << "<-" << '\n';
   cout << "->" << s7 << "<-" << '\n';
}
```

```
Ausgabe
-><-
-><-
-><-
-><-
-><-
->C++<-
->C++<-
```

Aber Achtung, wenn Sie den String mit mehreren Werten initialisieren wollen, wie z.B. bei:

```
string s(5, '*');            // Erzeugt einen String mit 5 Sternen
```

Dann funktioniert das nicht mit den geschweiften Klammern. Denn die geschweiften Klammern stellen dann in C++11 eine sogenannte Initialisierungs-Liste da, und die würde in diesem Fall einen String mit 2 Zeichen erzeugen:

- Zeichen 1 ist das Zeichen mit dem Zeichen-Code „5“
- Zeichen 2 ist der Stern

Da auf den meisten Plattformen ASCII basierte Zeichensätze vorkommen und dort das große „A“ den Zeichen-Code „65“ hat, gibt folgendes Programm auf den meisten Rechnern „AB“ aus:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s{65, 'B'};           // Erzeugt NICHT einen String mit 65 B's
                                // Sondern meist einen String mit Inhalt "AB"
   cout << s << '\n';
}
```

```
Mögliche Ausgabe (nur auf Plattformen mit ASCII-basiertem Zeichensatz)
AB
```

5.3.1.5 Automatische Typ-Deduktion mit „auto“ und „decltype“

Mit C++11 sind Variablen-Definitionen eingeführt worden, bei denen der Programmierer den Typ nicht selber angeben muß, sondern der Compiler ihn ermittelt. Hierfür gibt es in C++11 nun zwei Formen:

- Mit „auto“ – hier wird der Typ anhand des Typs des Initial-Wertes bestimmt. Für uns ist dies lange Zeit die einzige Form der automatischen Typ-Deduktion, die wir nutzen

werden.

- Mit „decltype“ – hier wird der Typ anhand eines Beispiel-Ausdrucks bestimmt. Diese Form ist für spezielle Anwendungen, die wir lange Zeit nicht benötigen. Darum werden wir diese Form im Skript quasi fast nicht nutzen, und auch nicht detailierter erklären.

Automatische Typ-Deduktion mit "auto"

Bei der Nutzung von „auto“ wird die Variablen-Definition nicht mit dem Typ sondern stattdessen mit dem Schlüsselwort „auto“ eingeleitet. Der Compiler wird hierbei aufgefordert den Typen selber anhand des Initialwertes zu bestimmen. Im folgenden Beispiel sehen wir die Definition vierer Int-Variablen „n1“ bis „n4“ und einer Double-Variablen „d“ ohne Angabe des Typen – der Compiler ermittelt die Typen automatisch aus den Initalwerten. Zur besseren Rückmeldung wird hier der Typ mit Hilfe von RTTI (typeinfo, typeid) ausgegeben – auch wenn wir RTTI in der Vorlesung nicht kennen lernen werden.

```
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
   // Literal "42" hat den Typ "int" -> Variable "n1" hat den Typ "int"
   auto n1 = 42;
   cout << typeid(n1).name() << " -> " << n1 << '\n';

   // Literal "44" hat den Typ "int" -> Variable "n2" hat den Typ "int"
   auto n2(44);
   cout << typeid(n2).name() << " -> " << n2 << '\n';

   // Variable "n1" hat den Typ "int" -> Variable "n3" hat den Typ "int"
   auto n3 = n1;
   cout << typeid(n3).name() << " -> " << n3 << '\n';

   // Variable "n2" hat den Typ "int" -> Variable "n4" hat den Typ "int"
   auto n4 = n2;
   cout << typeid(n4).name() << " -> " << n4 << '\n';

   // Var. "n2" hat den Typ "int", "2*n2" dann auch -> Variable "n4" hat den Typ "int"
   auto n5 = 2*n2;
   cout << typeid(n5).name() << " -> " << n5 << '\n';

   // Literal "3.14" hat den Typ "double" -> Variable "d" hat den Typ "double"
   auto d = 3.14;
   cout << typeid(d).name() << " -> " << d << '\n';
}
```

```
Mögliche Ausgabe (nicht exakt, da die genaue Ausgabe von RTTI undefiniert ist)
int -> 42
int -> 44
int -> 42
int -> 44
int -> 88
double -> 3.14
```

Damit der Compiler den Typ aus dem Initalwert selber bestimmen kann, muß es genau einen Intialwert geben. Die Syntax ohne Initialwert oder die Syntax mit runden Klammern und mehreren Initialswerten funktionert nicht. Außerdem gibt es im Zusammenspiel mit Zeichenketten-Literalen bzw. den geschweiften Klammern einige Probleme – siehe weiter unten im Kapitel.

Die Definition mit „auto“ ist also eingeschränkt – aber gerade bei komplexen Typen und

generischem Code oft sehr hilfreich.

```
#include <vector>
using namespace std;

int main()
{
   vector<int> v;
   vector<int>::iterator iter1 = begin(v);     // Viel "sinnlose" Schreibarbeit
   auto                  iter2 = begin(v);     // Viel kuerzer und einfacher
}
```

Hinweis – die automatische Typ-Deduktion mit „auto“ kann auch mit „const“ und Referenzen kombiniert werden.

Probleme mit „auto“

Zeichenketten-Literale sind **nicht** vom Typ „std::string“ – d.h. „auto“ mit einem Initial-Zeichenketten-Literal erzeugt **keine** String-Variable. Die automatische Typ-Bestimmung läßt sich zwar auch mit Zeichenketten-Literalen nutzen – erzeugt aber nicht unbedingt das vom C++ Anfänger erwartete Ergebnis:

```
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
   auto x = "C++";
   cout << typeid(x).name() << " -> " << x << '\n';
}
```

```
Mögliche Ausgabe (nicht exakt, da die genaue Ausgabe von RTTI undefiniert ist)
char const * -> C++
```

Automatische Typ-Deduktion mit "decltype"

Während die automatische Typ-Deduktion mit „auto“ den Typ über den Initialwert ermittelt, arbeitet „decltype“ mit einem Beispiel-Ausdruck, der in runden Klammern hinter „decltype“ als Typ angegeben werden muß:

```
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

string fct();                   // Deklaration einer Funktion "fct", die einen
                                // "string" zurueckgibt - siehe spaeter
int main()
{
   int n = 6;
   decltype(n) x1 = n;   // Da "n" ein "int", wird nun "x1" ein "int"
   decltype(1+2) x2;     // Da "1+2" ein "int", wird nun "x2" ein "int"
   decltype(x1<x2) x3;   // Da "x1<x2" ein "bool", wird nun "x3" ein "bool"
   decltype(fct()) x4;   // Da "fct()" einen String zurueckgibt, wird "x4" ein "string"

   cout << typeid(x1).name() << '\n';
   cout << typeid(x2).name() << '\n';
   cout << typeid(x3).name() << '\n';
   cout << typeid(x4).name() << '\n';
}
```

```
Mögliche Ausgabe (nicht exakt, da die genaue Ausgabe von RTTI undefiniert ist)
class std::initializer_list<int>
int
int
```

```
bool
std::string
```

5.3.2 Lokale Variablen

- Alle Variablen, die in Funktionen definiert werden, sind lokale Variablen.
- Lokale Variablen können an fast jeder Stelle in Funktionen definiert werden.
- Lokale Variablen sind lokal zu dem sie umgebenden scope (Bereich), der durch die geschweiften Klammern gebildet wird – d.h. auch der Name.
- Im Programm kann auf sie nur ab der Definition bis zum Ende des sie umgebenden Scopes zugegriffen werden – ihre Sichtbarkeit ist also sehr begrenzt.
- Lokale Variablen werden bei der Programmausführung an der Stelle ihrer Definition erzeugt, und beim Verlassen des sie umgebenden Scopes zerstört. Ihre Lebensdauer korrospondiert also mit ihrer Sichtbarkeit und ist ganz exakt festgelegt.

```
int main()
{
   int n1;        // Hier wird n1 erzeugt => n1 ist nun sichtbar und kann genutzt werden
   n1 = 2;        // Okay, denn n1 exisitiert und ist sichtbar
   n2 = 3;        // Compiler-Fehler - n2 existiert nicht
   int n2;        // Hier wird n2 erzeugt => n2 ist nun sichtbar und kann genutzt werden
   n2 = 4;        // Okay, denn nun existiert n2

   {              // Neuer Scope
      int n3;     // Hier wird n3 erzeugt
      n3 = 5;     // Okay, denn n3 exisitiert und ist sichtbar
      n1 = 6;     // Okay, n1 ist auch hier sichtbar und existent
      int n2;     // Hier wird ein neues n2 erzeugt, und verdeckt das obere n2
      n2 = 7;     // Hier wird das innere n2 genommen, denn das obere n2 ist verdeckt
   }              // Hier wird n3 und das innere n2 zerstoert

   n3 = 8;          // Compiler-Fehler - n3 ist zerstoert und auch nicht mehr sichtbar
   n2 = 9;          // Hier wird wieder das aeussere n2 angesprochen
   n1 = 10;         // Okay, n1 ist hier natuerlich bekannt
}                   // Hier werden erst n2 und dann n1 zerstoert
                    // und sind auch nicht mehr sichtbar
```

Hinweis – wie oben erwähnt ist ein Block „{ }“ in einer Funktion ein Äquivalent zu einer Anweisung, und kann daher jederzeit zur Schachtelung benutzt werden.

Es ist C++ ein empfehlenswerter Stil lokale Variablen erst dort zu definieren, wo sie benötigt werden. Dies hat mehrere Vorteile:

- Ihre Sichtbarkeit wird so eingeschränkt, und damit der Bereich indem die Variable falsch benutzt werden kann.
- Die Definition steht dort wo man die Variable benötigt, und nicht viele Zeilen darüber.
- Die Variablen können so direkt mit dem richtigen Initalwert erzeugt werden, was vorher oft nicht möglich war.
- Variablenerzeugung und –zerstörung, bzw. sagen wir schon mal lieber Objekterzeugung und –zerstörung, sind in C++ nicht immer *kostenlos*, d.h. können Performance und Speicherplatz kosten. Und damit Sie nur für Dinge *zahlen,* die sie auch benötigen, vermeiden Sie so unnötige Konstruktionen und Destruktionen.

5.3.3 Globale Variablen

Globale Variablen sind Variablen, auf die – wie der Name schon sagt – von überall zugegriffen werden kann. Damit scheinen sie im ersten Augenblick sehr *sexy* zu sein, da sie scheinbar problemlos nutzbar sind. In der Praxis ist es aber so, dass globale Variablen häufig ein Quell von Fehlern und Problemen sind. Uneingeschränkter Zugriff heißt nämlich auch, dass jeder die Variable verwenden kann, und sie damit auch leicht fehlerhaft oder konkurierend verwendet wird.

Globale Variablen werden außerhalb von Funktionen und Klassen definiert. Sie werden vor dem Betreten der Main-Funktion erzeugt, und nach dem Verlassen von Main zerstört. Obwohl wir noch keine Funktionen kennen, hier ein kleines Beispiel, das zeigt, dass globale Variablen nicht an den Kontext eines Scopes (z.B. einer Funktion) gebunden sind.

```
#include <iostream>
using namespace std;

int g = 42;

void fct()
{
   cout << "fct - g: " << g << '\n';
   g = 32;
}

int main()
{
   cout << "g: " << g << '\n';
   ++g;
   cout << "g: " << g << '\n';

   fct();

   cout << "g: " << g << '\n';
}
```

```
Ausgabe
g: 42
g: 43
fct - g: 43
g: 32
```

Praxis – in der Praxis sollten globale Variablen vermieden werden, da sie häufig Probleme verursachen. Häufig werden sie durch die Verwendung von Klassen überflüssig.

5.4 Konstanten

In der Praxis sollten möglichst wenig Literale fest im Quelltext stehen.

```
// Sehr sehr schlechter Code
double betrag = sum * 1.19;          // MwSt dazurechnen
```

Stattdessen sollten Konstanten verwandt werden. Um statt Variablen Konstanten zu definieren, muss einfach der Modifier „const“ (ist ein Schlüsselwort) bei einer Variablen-Definition vor oder hinter den Typ gesetzt werden.

```
const double mwst = 19.0;
const double mwst_mult = (1.0 + mwst/100.0);

...

euro betrag = sum * mwst_mult;
```

Konstanten sind in C++ konstante Variablen, d.h. sie haben noch vollständigen Variablen-Charakter (z.B. benötigen sie Speicherplatz, haben eine Adresse, uvm). Aber der Compiler sorgt dafür das sie nicht verändert werden, und er darf beliebig mit ihnen optimieren – z.B. kann er die Konstanten-Werte direkt in den Maschinencode einsetzen.

```
const int ci = 18;
ci = 20;            // Compiler-Fehler - ci ist const
++ci;               // Compiler-Fehler - ci ist const
```

Da Konstanten „konstante Variablen“ sind gilt für sie alles, was auch für Variablen gilt, d.h. in Funktionen definierte Konstanten sind lokale Konstanten und leben nur innerhalb der Funktion und sind nur innerhalb dieser benutzbar. Dies ist bei Konstanten selten gewünscht - meist möchte man sie global für das gesamte Programm definieren.

In diesem Fall definiert man sie ausserhalb von Funktionen. Solche globalen Konstanten existieren die gesamte Programmlaufzeit, und sind von allen später definierten Funktionen nutzbar.

```
const int max_size = 42;

void f()
{
   //  max size ist hier nutzbar
}

...

int main()
{
   //  max_size ist auch hier nutzbar
}
```

Achtung – sobald mehrere Quelltexte ins Spiel kommen, kann es etwas komplizierter werden. Aus Zeitmangel kann dieses Thema hier nicht näher betrachtet werden – letztlich entstehen die Probleme, da die Konstanten im gesamten Programm natürlich nur einmal vorhanden sein dürfen – und das ist in manchen Situationen nicht ganz trivial. Für weitere Informationen muß ich hier daher auf die Literatur verweisen – Stichpunkte: Konstante Objekte müssen als „extern“ deklariert werden und nur einmal definiert werden, und integrale Konstanten werden vom Compiler automatisch als „static“ betrachtet.

5.4.1 Kombination von „const“, „auto“ und „decltype“

Achtung – im Normallfall wird bei der automatischen Typ-Deduktion mit „auto“ das „const“ am Initial-Typ ignoriert, d.h. eine Auto Variable ist von sich aus niemals „const“.

```
#include <iostream>
using namespace std;

int main()
{
   const int n = 11;
   auto a = n;
   a = 20;                  // Okay, denn "a" ist nur eine normale Variable
}                           // "auto" hat "const" ignoriert
```

Soll der automatische Typ auch das „const“ erkennen und berücksichtigen, so muss die

Variablen-Definitions-Syntax mit „decltype“ (siehe Kapitel 5.3.1.5) gewählt werden:

```
#include <iostream>
using namespace std;

int main()
{
   const int n = 11;
   decltype(n) a = n;
   a = 20;                  // Compiler-Fehler - "a" ist auch eine Konstante
}                           // "decltype" ignoriert "const" NICHT
```

Aber selbst wenn „auto“ im Normallfall „const“ ignoriert – man kann „const“ und die automatische Typ-Deduktion mit „auto“ (siehe Kapitel 5.3.1.5) manuell kombinieren. Mit „const auto“ wird an den deduzierten Typ immer ein „const“ angefügt:

```
#include <iostream>
using namespace std;

int main()
{
   int n = 11;
   const auto a = n;
   a = 20;                  // Compiler-Fehler, da "a" eine Konstante ist
}
```

5.4.2 Linksbindend, außer...

Leider ist in C++ nichts wirklich einfach – dies gilt selbst für „const“...

Im Normallfall ist „const“ linksbindend. Damit ist gemeint, dass sich const immer auf den links vom const stehenden Typ bezieht. Bei:

```
int const gap = 4;
```

sorgt das const dafür, dass das int-Objekt „gap“ konstant ist.

Steht links vom const aber kein Typ, so bindet „const“ nach rechts. Das hat zur Folge, dass Sie auch folgendes schreiben dürfen:

```
const int gap = 4;
```

Beide Varianten sind absolut gleichwertig. Lassen Sie sich nicht verwirren, wenn Sie mal die eine und mal die andere Variante finden. Und auch nicht, wenn Sie die eine Variante in Ihren Quelltexten nutzen, und der Compiler in einer Meldung die andere benutzt. Beide Varianten sind absolut gleichwertig.

5.4.3 constexpr

Mit „const“ definierte Variablen können Laufzeit- oder Compile-Zeit Konstanten sein, je nach Kontext. Werden Variablen stattdessen mit „constexpr“ definiert, so müssen es Compile-Zeit Konstanten sein.

```
constexpr int gap = 4;      // Zwingend Compile-Zeit Konstante
```

6 Operatoren

- Der Typ einer Variablen legt fest, welche Operationen auf sie durchführbar sind.
- Operationen können Funktionen oder Operatoren sein.

6.1 Operatoren

Operatoren sind quasi Funktionen, bei denen der Name aus fest definierten Zeichen besteht, der Aufruf implizit ohne „()“ erfolgt, und bei denen im Normallfall die Anzahl der Parameter festliegt.

- Beispiele: = + - * / ?: ->* >>= ==

Zusätzlich haben Operatoren einen Vorrang (auch Priorität genannt – z.B. Punktrechnung vor Strichrechnung bei mathematischen Typen) und eine Assozivität, d.h. eine Auswertungsrichtung:

- Denn was ist das Ergebnis von: „8-4-2“ ?
  - (8-4)-2 => 2 richtig
  - 8-(4-2) => 6 falsch

Das richtige Ergebnis ist „2“, denn der Minus-Operator ist linksassoziativ (er wird von links nach rechts ausgewertet). Daher zuerst wird „8-4“ gerechnet, und dann wird vom Zwischen-Ergebnis „4“ wieder „2“ abgezogen.

In der Praxis sind viele Operationen nur als Operatoren verfügbar, wie z.B. die Addition von Integer-Variablen.

6.2 Operatoren-Liste

Hier eine vollständige Liste aller Operatoren in C++ mit ihrem Vorrang und ihrer Assozivität:

Vorr.	Operator	Bedeutung	Assozivität
1.	::	Bereichszuordnung	---
2.	.-> ->	Elementzugriff	L -> R
	[ ]	Array-Zugriff (Index-Operator)	L -> R
	( )	Funktionsaufruf	L -> R
	++ --	Post-Increment bzw. –Decrement	L -> R
	typeid	Typ	L -> R
	const_cast static_cast reinterpret_cast dynamic_cast	4 Konvertierungs-Operatoren	L -> R
3.	sizeof	Größe	R -> L
	++ --	Prä-Increment bzw –Decrement	R -> L

	~	1er Komplement	R -> L
	!	Logisches not	R -> L
	+ -	Vorzeichen (unäres + bzw. -)	R -> L
	&	Adresse	R -> L
	*	Dereferenzierung	R -> L
	new delete	Dynamische Speicherverwaltung	R -> L
	( )	Klassischer C Cast	R -> L
4.	.* ->*	Zeiger auf Element	L -> R
5.	*	Multiplikation	L -> R
	/	Division	L -> R
	%	Modulo	L -> R
6.	+	Addition	L -> R
	-	Subtraktion	L -> R
7.	<< >>	Bit-Links- bzw. Bit-Rechts-Schieben oder auch Ein- bzw. Ausgabe-Operator	L -> R
8.	<	Kleiner	L -> R
	>	Größer	L -> R
	<=	Kleiner gleich	L -> R
	>=	Größer gleich	L -> R
9.	==	Gleich	L -> R
	!=	Ungleich	L -> R
10.	&	Bitweises Und	L -> R
11.	^	Bitweises XOR	L -> R
12.	|	Bitweises Oder	L -> R
13.	&&	Logisches Und	L -> R
14.	||	Logisches Oder	L -> R
15.	? :	Bedingung	R -> L
16.	=	11 Zuweisungen	R -> L
	*=		
	/=		
	%=		

	+= -= <<= >>= &= |= ^=		
17.	,	Komma	L -> R

- Viele dieser Operatoren sollten intuitiv klar sein, da sie normal sind und Sie Programmier-Erfahrung mitbringen – z.B. die Addition oder die Vergleichs-Operatoren.
- Ein paar Operatoren sind aber nicht zwingend intuitiv in ihrer Benutzung und Semantik, so dass sie im Weiteren besprochen werden.
- Und ein Rest von Operatoren wird in diesem Tutorial nicht besprochen.

6.3 Zuweisungs-Operatoren

Der einfache Zuweisungs-Operator '=' weist den Wert des rechten Ausdrucks der Variablen auf der linken Seite zu.

```
#include <iostream>
using namespace std;

int main()
{
   int n;
   int m = 4;                        // Achtung – keine Zuweisung, sondern Initialsierung

   n = m+2;
   cout << "n: " << n << '\n';
}
```

Ausgabe
```
n: 6
```

Zusätzlich gibt es noch 10 operative Zuweisungs-Operatoren, die die jeweilige Operation auf rechtem und linkem Ausdruck der Anweisung durchführen. Die Anweisung „erg #= arg;" entspricht hierbei immer der Anweisung „erg = erg # (arg);" – wobei das Hash „#" hier als Platzhalter für einen der in der Tabelle bei Prio 16 aufgeführten 10 Operatoren ist. Der folgende Code enthält jeweils ein Beispiel für die multiplikative Zuweisung „*=" und die subtraktive Zuweisung „-=":

```
#include <iostream>
using namespace std;

int main()
{
   int n;                  // Uninitialisiert ist okay, da n zuerst geschrieben wird
   int m = 4;

   n *= 3+m;                          // entspricht: n = n * (3+m);    => 35
   cout << "n: " << n << '\n';

   n -= m + 2;                        // entspricht: n = n - (m+2);    => 29
   cout << "n: " << n << '\n';
}
```

```
Ausgabe
n: 35
n: 29
```

Hinweis – für die korrekte Darstellung des entsprechenden Ausdrucks muss der zugewiesene Ausdruck in Klammern gesetzt werden. Die Prioritäts-Regeln wie „Punkt-Rechnung vor Strich-Rechnung“ gelten nicht für den Operator der Zuweisung. Vergleiche hierfür auch das erste Beispiel mit „n *= 3+m“, wo die Addition „3+m“ vor der Multiplikation ausgewertet wird.

Alle Zuweisungen können direkt in einem umfassenden Ausdruck genutzt werden, da in C++ eine Zuweisung einen Wert hat – die Variable, der etwas zugewiesen wurde.

```
#include <iostream>
using namespace std;

int main()
{
   int n, m;            // Uninitialisiert ist okay, da n und m zuerst geschrieben werden
   int o = 4;

   n = 3 + (m = 4 * (o +=2));                // (*)

   cout << "n: " << n << '\n';
   cout << "m: " << m << '\n';
   cout << "o: " << o << '\n';
}
```

```
Ausgabe
n: 27
m: 24
o: 6
```

In der Zeile (*) wird

- Zuerst „o“ um „2“ erhöht – „o“ ist dann also „6“.
- Dann wird der neue Wert „6“ von „o“ mit „4“ mulitpliziert und dann „m“ zugewiesen – „m“ wird also zu „24“.
- Zum Schluß wird der neue Wert „24“ von „m“ zu „3“ addiert und dann „n“ zugewiesen – „n“ wird also zu „27“.

Hinweis – aufgrund der Operator-Prioritäten mußte der Ausdruck in (*) geklammert werden.

Achtung – selbst wenn es mit den Zuweisungs-Operatoren möglich ist: verändern und nutzen Sie in C++ eine Variable niemals mehrmals im gleichen Ausdruck. In vielen Fällen ist das Ergebnis undefiniert, da die genaue interne Abarbeitungs-Reihenfolge nicht festgelegt ist, damit der Compiler diese Freiheit für Performance-Optimierungen nutzen kann.

6.4 Geteilt- und Modulo-Operator

Während die normalen mathematischen Operatoren (plus, minus und mal) in ihrem Verhalten ziemlich intuitiv sind, ist dies beim Geteilt- und Modulo-Operator vielleicht nicht so.

6.4.1 Geteilt-Operator

Für Fließkomma-Typen (wie z.B. „float“ oder „double“) arbeitet der Geteilt-Operator ganz intuitiv. Anders wird dies, wenn man ihn auf integrale Typen (wie z.B. „short“, „unsigned int“, „long“, …) anwendet. Denn dann ist das Ergebnis kein Fließkomma-Ergebnis, sondern ein integraler Typ.

Teilen Sie Integer durch Integer, so ist das Ergebnis wieder ein Integer – selbst wenn das Ergebnis damit nicht exakt ist. Das Ergebnis ist nur der ganz-zahlige Anteil der Division, den Rest kann man mit dem Modulo-Operator bestimmen – siehe Kapitel 6.4.3.

```
#include <iostream>
using namespace std;

int main()
{
   cout << "7/1 => " << 7/1 << '\n';            // => 7
   cout << "7/2 => " << 7/2 << '\n';            // => 3
   cout << "7/3 => " << 7/3 << '\n';            // => 2
   cout << "7/4 => " << 7/4 << '\n';            // => 1
   cout << "7/5 => " << 7/5 << '\n';            // => 1
   cout << "7/6 => " << 7/6 << '\n';            // => 1
   cout << "7/7 => " << 7/7 << '\n';            // => 1
   cout << "7/8 => " << 7/8 << '\n';            // => 0
   cout << "7/9 => " << 7/9 << '\n';            // => 0
}
```

```
Ausgabe
7/1 => 7
7/2 => 3
7/3 => 2
7/4 => 1
7/5 => 1
7/6 => 1
7/7 => 1
7/8 => 0
7/9 => 0
```

Hinweis – es sind keine Klammern um die Divisions-Ausdrücke (z.B. „7/2“) notwendig, da die Priorität des Geteilt-Operators „/“ (Prio 5) höher ist als die des Ausgabe-Operator „<<“ (Prio 7). Wenn Sie mit den Operator-Prioritäten nicht so vertraut sind, oder die Auswertung-Reihenfolge expliziter machen woll, können Sie die Rechen-Ausdrücke natürlich auch zusätzlich klammern, z.B.:

```
   cout << "7/1 => " << (7/1) << '\n';
```

6.4.2 Fließkomma-Ergebnis

Wie dividiert man aber nun zwei Integer-Zahlen so, daß man das korrekte Fließkomma-Ergebnis erhält? Ganz einfach: sobald einer der beiden Operanden ein Fließkomma-Typ ist, wird der Andere in einen Fließkomma-Typ gewandelt und man erhält eine „normale“ Fließkomma-Division mit Fließkomma-Typ Ergebnis.

Hierbei ist es egal, ob man eine Fließkomma-Variable, ein Fließkomma-Literal oder eine explizite Typ-Konvertierung mit „static_cast“ benutzt (explizite Typ-Konvertierungen mit „static_cast“ werden spaeter vorgestellt).

```
#include <iostream>
```

```
using namespace std;

int main()
{
   double d = 2;

   cout << 5./2 << '\n';                              // Fliesskomma-Literal
   cout << 5/2. << '\n';                              // Fliesskomma-Literal
   cout << 7/d << '\n';                               // Fliesskomma-Variable
   cout << 9/static_cast<double>(2) << '\n';          // Explizite Typ-Konvertierung
}
```

Ausgabe

```
2.5
2.5
3.5
5.5
```

6.4.3 Modulo-Operator

Der Modulo-Operator kann nur auf integrale Typen angewendet werden, und liefert dann den Rest der Integer-Division – vergleiche auch Kapitel 6.4.1.

```
#include <iostream>
using namespace std;

int main()
{
   cout << "7/1 => " << 7/1 << " Rest " << 7%1 << '\n';        // => 7 Rest 0
   cout << "7/2 => " << 7/2 << " Rest " << 7%2 << '\n';        // => 3 Rest 1
   cout << "7/3 => " << 7/3 << " Rest " << 7%3 << '\n';        // => 2 Rest 1
   cout << "7/4 => " << 7/4 << " Rest " << 7%4 << '\n';        // => 1 Rest 3
   cout << "7/5 => " << 7/5 << " Rest " << 7%5 << '\n';        // => 1 Rest 2
   cout << "7/6 => " << 7/6 << " Rest " << 7%6 << '\n';        // => 1 Rest 1
   cout << "7/7 => " << 7/7 << " Rest " << 7%7 << '\n';        // => 1 Rest 0
   cout << "7/8 => " << 7/8 << " Rest " << 7%8 << '\n';        // => 0 Rest 7
   cout << "7/9 => " << 7/9 << " Rest " << 7%9 << '\n';        // => 0 Rest 7
}
```

Ausgabe

```
7/1 => 7 Rest 0
7/2 => 3 Rest 1
7/3 => 2 Rest 1
7/4 => 1 Rest 3
7/5 => 1 Rest 2
7/6 => 1 Rest 1
7/7 => 1 Rest 0
7/8 => 0 Rest 7
7/9 => 0 Rest 7
```

6.5 Logische „Und“ und „Oder“ Operatoren

Bei den boolschen Operanden „Und“ („&&“) und „Oder“ („||“) werden die beiden Operanden logisch miteinander verknüpft, und das Ergebnis ist wieder ein „bool“, daher entweder „false“ oder „true“. Die beiden folgenden Tabellen zeigen das Ergebnis der boolschen Operation in Abhängigkeit von den Werten der Operanden:

Op. 1	Op. 2	&& (Und)
false	false	**false**
true	false	**false**
false	true	**false**
true	true	**true**

Op. 1	Op. 2	|| (Oder)
false	false	**false**
true	false	**true**
false	true	**true**
true	true	**true**

```
#include <iostream>
using namespace std;

int main()
{
   cout << boolalpha;

   cout << "false && false => " << (false && false) << '\n';       // => false
   cout << "false && true  => " << (false && true)  << '\n';       // => false
   cout << "true  && false => " << (true  && false) << '\n';       // => false
   cout << "true  && true  => " << (true  && true)  << '\n';       // => true

   bool b1 = false;
   bool b2 = true;

   bool or1 = b1 || b2;
   bool or2 = b1 || false;

   cout << "or1: " << or1 << '\n';           // => true
   cout << "or2: " << or2 << '\n';           // => false
}
```

Ausgabe

```
false && false => false
false && true  => false
true  && false => false
true  && true  => true
or1: true
or2: false
```

Hinweis – im Gegensatz zu den Divisions- bzw. Modulo-Operationem in Kapitel 6.4 muß hier der logische Ausdruck in der Ausgabe geklammert werden, da die Priorität des Ausgabe-Operators „<<“(Prio 7) höher ist als die des logischen Und-Operators „&&“ (Prio 13).

6.5.1 Kurzschluß-Auswertung

Eine Besonderheit der logischen Operatoren ist die sogenannte Kurzschluß-Auswertung, die im obigen Beispiel nicht auffällt. Sobald nämlich das Ergebnis des boolschen Ausdrucks feststeht, wird der restliche Ausdruck nicht mehr weiter ausgewertet. Nehmen wir z.B. an, dass wir eine Variable „b“ haben, die den Wert „false“ hat, und „b“ nun und-mäßig mit einer anderen Variablen „x“ verknüpft wird: „b && x“, dann hat dieser Ausdruck immer den Wert „false“ – unabhängig vom Wert von „x“. Die Variable „x“ wird also gar nicht mehr berücksichtigt.

Unser Problem ist, das es uns bei Variablen gar nicht auffällt, wenn sie im Ausdruck nicht ausgewertet werden. Die Kurzschluß-Auswertung scheint also keinen Einfluß auf unseren Programm-Ablauf zu haben. Dieser Eindruck täuscht aber.

Sobald z.B. Funktionen ins Spiel kommen, ist die Situation eine andere. Selbst wenn wir aktuell noch keine Funktionen kennen, will ich hier ein entsprechendes Beispiel anbringen, um die Kurzschluß-Auswertung „in Aktion“ zu zeigen. Kommen Sie später nochmal hierhin zurück.

```
#include <iostream>
using namespace std;
```

```
bool f_true()
{
   cout << "- f_true()" << '\n';
   return true;
}

bool f_false()
{
   cout << "- f_false()" << '\n';
   return false;
}

int main()
{
   cout << boolalpha;

   cout << "f_true() && f_false()" << '\n';
   bool b1 = f_true() && f_false();            // Ruft beide Funktionen auf
   cout << "> " << b1 << '\n';

   cout << '\n';

   cout << "f_false() && f_true()" << '\n';
   bool b2 = f_false() && f_true();            // Ruft nur "f_false" auf
   cout << " => " << b2 << '\n';
}
```

Ausgabe

```
f_true() && f_false()
- f_true()
- f_false()
> false

f_false() && f_true()
- f_false()
 => false
```

In beiden Fällen ist das Ergebnis in „b1“ bzw. „b2“ „false“. Aber während bei der ersten Berechnung beide Funktionen aufgerufen werden, wird bei der Zweiten nur die Funktion „f_false“ aufgerufen. Da sie „false“ zurückliefert, steht fest dass der Gesamt-Ausdruck „false“ wird, und die zweite Funktion „f_true“ gar nicht mehr aufgerufen werden muß.

6.6 Prä- und Post-Inkrement und -Dekrement Operatoren

Der Increment-Operator „++“ ist für alle arithmetischen Typen und Zeiger definiert und entspricht einer Erhöhung um „1“. Analog entspricht der Decrement-Operator „--“ einer Erniedrigung um „1“ bei arithmetischen Typen und Zeigern. Beide Operatoren gibt es in Prä- und Postfix-Notation, d.h. der Operator kann vor bzw. nach der Variablen stehen. Der Unterschied zwischen Prä- und Postfix-Notation wird weiter unten erklärt.

```
#include <iostream>
using namespace std;

int main()
{
   int n = 4;
   cout << n << " ++ => ";
   ++n;
   cout << n << " ++ => ";
   n++;
   cout << n << '\n';

   n = 4;
   cout << n << " -- => ";
   --n;
   cout << n << " -- => ";
   n--;
   cout << n << '\n';
```

```
}
```

Ausgabe
```
4 ++ => 5 ++ => 6
4 -- => 3 -- => 2
```

Hinweise:

- Im obigen Beispiel gibt es keinen Unterschied zwischen den Prä- und Postfix-Notationen, aber bei anderen Nutzungsweisen – siehe unten – ist dies anders.
- Zu den mathematischen Typen gehören sämtliche Zeichen-Typen, alle integralen Typen (außer „bool“) und die Fließkomma-Typen.

Bevorzugen Sie in C++ immer die Präfix-Notation (daher der Operator steht vor der Variablen) – wenn Sie nicht den semantischen Unterschied zwischen beiden explizit nutzen wollen (siehe unten). Bei den elementaren Daten-Typen und Zeigern macht dies zwar keinen Unterschied – anders wird dies aber z.B. bei Iteratoren, wo die Präfix-Notation performanter ist. Im Sinne eines möglichst einheitlichen Stils sollten Sie daher in C++ immer die Präfix-Notation verwenden.

Die Prä- und Postfix-Notation verhalten sich unterschiedlich, wenn der Ausdruck Teil eines größeren Ausdrucks ist.

- Präfix-Notation „++x“ bzw. „--x"
  Zuerst wird die Variable verändert und dann als Ausdrucks-Ergebnis ausgewertet
  1. Verändern von „x“
  2. Auswerten von „x“
- Postfix-Notation „x++“ bzw. „x--"
  Zuerst wird die Variable als Ausdrucks-Ergebnis ausgewertet und dann verändert
  1. Auswerten von „x“
  2. Verändern von „x“

```
#include <iostream>
using namespace std;

int main()
{
   int n = 4;

   cout << "n: " << n << '\n';

   cout << "++n => " << ++n << '\n';        // Liefert schon den neuen Wert "5"
   cout << "n: " << n << '\n';

   cout << "n++ => " << n++ << '\n';        // liefert noch den alten Wert "5"
   cout << "n: " << n << '\n';              // Hier ist "n" dann auch "6"
}
```

Ausgabe
```
n: 4
++n => 5
n: 5
n++ => 5
n: 6
```

Achtung – selbst wenn es mit den Increment- und Decrement-Operatoren möglich ist: verändern und nutzen Sie in C++ eine Variable niemals smehrmals im gleichen Ausdruck. In vielen Fällen ist das Ergebnis undefiniert, da die genaue interne Abarbeitungs-Reihenfolge

nicht festgelegt ist, damit der Compiler diese Freiheit für Performance-Optimierungen nutzen kann.

6.7 Fragezeichen-Operator

Der Fragezeichen-Operator „?:“ erlaubt eine Fallunterscheidung innerhalb eines Ausdrucks auf Basis eines boolschen Wertes. Hiermit können z.B. Variablen in Abhängigkeit von einer Bedingung initialisiert werden.

Syntax: Boolscher-Ausdruck ? True-Wert : False-Wert

Das Ergebnis des Ausdrucks ist der „True-Wert“ wenn der boolsche Ausdruck „true“ ist, und der „False-Wert“ wenn der boolsche Ausdruck „false“ ist.

```
#include <iostream>
using namespace std;

int main()
{
   cout << boolalpha;

   bool flag = true;
   int n = flag ? 3 : 4;
   cout << "flag: " << flag << "  =>  n: " << n << '\n';

   flag = false;
   n = flag ? 3 : 4;
   cout << "flag: " << flag << "  =>  n: " << n << '\n';

   int m = n*(flag ? 1 : -1) + 5;
   cout << "flag: " << flag << "  =>  m: " << m << '\n';
}
```

Ausgabe

```
flag: true  =>  n: 3
flag: false  =>  n: 4
flag: false  =>  m: 1
```

Hinweise:

- Der Fragezeichen-Operator wird häufig auch Bedingungs-Operator genannt.
- Er ist der einzige trinäre Operator (mit drei Operanden) von C++

7 Kontrollstrukturen

7.1 Bedingter-Kontrollfluss – „if“ & „else“

Die wichtigste Kontroll-Struktur ist die If-Anweisung. Sie ermöglicht einen bedingten Sprung, d.h. in Abhängigkeit von einem Boolschen-Ausdruck unterschiedliche Pfade im Programm zu durchlaufen.

7.1.1 Einfachste Form

Die einfachste Form der If-Anweisung sieht folgendermaßen aus:

Syntax:

```
if (ausdruck)
   anweisung
```

Sie beginnt mit dem Schlüsselwort „if“ – dann folgt in runden Klammern ein Ausdruck, der nach „bool“ auswertbar sein muss. Abgeschlossen wird sie durch eine Anweisung. Diese Anweisung wird nur ausgeführt, wenn der Ausdruck zu „true“ ausgewertet wurde. Im Falle von „false“ wird die Anweisung übersprungen.

```
#include <iostream>
using namespace std;

int main()
{
   int n = 5;

   if (n<7)
      cout << "n<7" << '\n';

   if (n>20)
      cout << "n>20" << '\n';

   if (n!=6)
      cout << "n!=6" << '\n';
}
```

```
Ausgabe
n<7
n!=6
```

7.1.2 Blöcke für mehrere Anweisungen

Soll im If-Zweig mehr als eine Anweisung ausgeführt werden, so muß mit den geschweiften Klammern ein Block gebildet werden. Ein solcher Block steht dabei syntaktisch für eine einzelne Anweisung.

```
#include <iostream>
using namespace std;

int main()
{
   int n = 5;

   if (n<7)
   {
      cout << "n<7" << '\n';
      cout << "n ist " << n << '\n';
      cout << "Und weiter geht es..." << '\n';
   }
}
```

```
Ausgabe
n<7
n ist 5
Und weiter geht es...
```

Hinweis – dies gilt genauso für Schleifen oder andere Konstrukte, die normalerweise nur auf eine Anweisung wirken.

7.1.3 If-Else Anweisung

Für den Fall, dass neben dem True-Fall auch beim False-Fall Aktionen ausgeführt werden

sollen, kann auf die Anweisung (bzw. dem Block) des True-Falls das Schlüsselwort „else“ mit einer Anweisung für den False-Fall folgen.

Syntax:

```
if (ausdruck)
   anweisung
else
   anweisung
```

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
   int n = 5;

   if (n<7)
      cout << "n<7" << '\n';
   else
      cout << "n>=7" << '\n';

   if (n>20)
      cout << "n>20" << '\n';
   else
      cout << "n<=20" << '\n';
}
```

```
Ausgabe
n<7
n<=20
```

Hinweis – auch für den False-Zweig gilt natürlich: wenn mehr als eine Anweisung notwendig ist, dann muß ein Block genutzt werden – siehe Kapitel 7.1.2.

7.1.4 If-Anweisung ohne True-Zweig

Es gibt keine Syntax, um eine If-Anweisung nur mit False-Zweig ohne True-Zweig zu implementieren. Ist dies gewünscht, so gibt es zwei Implementierungs-Möglichkeiten:

- True-Zweig mit leerer Anweisung
- Bool-Ausdruck in der If-Anweisung umformulieren

True-Zweig mit leerer Anweisung

In C++ sind leere Anweisungen erlaubt – sie bestehen aus einem einzelnen Semikolon. Damit läßt sich der True-Zweig einer If-Anweisung sehr schnell abhandeln.

```
#include <iostream>
using namespace std;

int main()
{
   int n = 10;

   if (n<7);                        // <= dieses Semikolen ist der leere True-Zweig
   else
      cout << "n>=7" << '\n';
}
```

```
Ausgabe
n>=7
```

Achtung – so ein leerer True-Zweig ist nicht besonders gut lesbar. Und so ein einzelnes Semikolon ist schnell überlesen bzw. verwirrend – von daher ist die Lösung nicht zu empfehlen. Formulieren Sie besser den Bool-Ausdruck um.

Bool-Ausdruck in der If-Anweisung umformulieren

Formulieren Sie den boolschen Ausdruck so um, dass er statt „true“ „false“ liefert bzw. umgekehrt. Wenn sich der Ausdruck nicht umformulieren läßt, bzw. die Umformulierung aufwändig und fehlerträchtig ist – dann nutzen Sie einfach den boolschen Not-Operator „!“ um den boolschen Ausdruck zu negieren.

```
#include <iostream>
using namespace std;

int main()
{
   int n = 10;

   if (n>=7)                          // Ausdruck umformuliert
      cout << "n>=7" << '\n';

   if (!(n<7))                        // Ausdruck negiert mit !
      cout << "n>=7" << '\n';
}
```

```
Ausgabe
n>=7
n>=7
```

7.1.5 Mehrfach-Verzweigungen mit If-ElseIf

Nicht immer gibt es nur einen True-Fall bzw. einen True- und einen False-Fall. Häufig hat man eine Art Auflistung von verschiedenen Fällen, d.h. eine Mehrfach-Verzweigung. In diesem Fällen nimmt man verschachtelte If-Anweisungen, die auch als If-ElseIf-Anweisung bezeichnet werden.

Im Prinzip ist eine If-ElseIf-Anweisung nichts neues – hier wird nur der False-Zweig durch eine If-Anweisung dargestellt. Der Unterschied ist eine eigenständige Einrückungs-Konvention, bei der das „if“ direkt auf das „else“ folgt.

Syntax:

```
if (ausdruck)
   anweisung
else if (ausdruck)
   anweisung
else
   anweisung
```

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
   int age = 27;
   if (age<10)
      cout << "Kind im Alter von " << age << '\n';
   else if (age<18)
      cout << "Jug'\n'icher im Alter von " << age << '\n';
```

```
   else if (age<30)
      cout << "Junger Erwachsener im Alter von " << age << '\n';
   else
      cout << "Erwachsener im Alter von " << age << '\n';
}
```

Ausgabe
```
Junger Erwachsener im Alter von 27
```

Hinweis – auch hier ist der Else-Zweig natürlich optional.

7.1.6 Variablen-Definition im Bedingungs-Ausdruck

Es ist möglich, innerhalb des If-Bedingungs-Ausdrucks

7.1.7 Initialisierte Bedingung

Mit C++17 wird ein neue Variante einer If-Anweisung eingeführt werden

7.2 Mehrfach-Verzweigung – „switch“

Für Mehrfachverzweigungen in Abhängigkeit von einer integralen Variablen gibt es in C++ die ‚switch‘ Anweisung. Außerdem kann der integrale Ausdruck nur auf Gleichheit zu Compile-Zeit Konstanten des entsprechenden Typs getestet werden. Wenn man das jeweilige Problem mit diesen Einschränkungen lösen kann, dann ist eine Switch-Anweisung einer If-Else-If Mehrfach-Verzweigung aus Gründen der Lesbarkeit vorziehen. Ansonsten muß man eine If-Else-If Mehrfach-Verzweigung nutzen – siehe Kapitel 7.1.5.

Syntax

```
switch (ausdruck)
{
case constant:             // <- diese Konstante muss compile-zeit-konstant sein
   anweisung
   ...
   break;
case constant:             // <- diese Konstante muss compile-zeit-konstant sein
   anweisung
   ...
   break;
...
default:
   anweisung
   ...
   break;
}
```

Der Default-Zweig ist hierbei optional.

Erst die „break“ Anweisung beendet einen Block. Ist kein „break“ vorhanden, so läuft der Programmfluss in den nächsten Case-Block hinein - auch in den Default-Block. Dieses Verhalten mag etwas ungewöhnlich wirken, sorgt aber dafür dass man einen Case-Block für mehrere Werte nutzen kann – siehe den Case-Block im folgenden Beispiel für die Werte „2“ und „3“.

```
#include <iostream>
using namespace std;

int main()
{
   for (int i=0; i<7; ++i)
   {
      switch (i)
      {
      case 1:
         cout << "i ist 1" << '\n';
         break;
      case 2:
      case 3:
         cout << "i ist 2 oder 3" << '\n';
      case 4:
         cout << "Fluss kommt aus 2/3 oder i ist 4" << '\n';
         break;
      default:
         cout << "i ist weder 1,2,3 oder 4" << '\n';
         break;
      }
   }
}
```

Ausgabe

```
i ist weder 1,2,3 oder 4
i ist 1
i ist 2 oder 3
Fluss kommt aus 2/3 oder i ist 4
i ist 2 oder 3
Fluss kommt aus 2/3 oder i ist 4
Fluss kommt aus 2/3 oder i ist 4
i ist weder 1,2,3 oder 4
i ist weder 1,2,3 oder 4
```

Hinweis – Switch-Anweisungen werden häufig mit Aufzählungen (Enums) kombiniert.

7.3 For-Schleife

Syntax

```
for (init; test; update)
   anweisung
```

Zählschleife
Kopfgesteuert

```
for (int i=0; i<3; ++i)
{
   std::cout << i << ' ';
}
```

Ausgabe

```
0 1 2
```

Es sind leere Init-, Test- und Update-Anweisungen erlaubt.
Eine leere Test-Anweisung entspricht „true“.

Die typische C++ Art für eine Endlos-Schleife ist daher:

```
for (;;)
{
   // Die leere For-Schleife, Endlose Weiten....
}
```

Hinweis – Endlos-Schleifen werden häufig in Verbindung mit „break“ (siehe Kapitel 7.6.1) für Schleifen mit einem Ausgang in der Mitte benutzt (siehe Kapitel 7.7).

7.3.1 Range-Basierte For-Schleife

Später werden wir Container kennenlernen, die es uns ermöglichen, Objekt-Mengen zu speichern. Um über diese Objekt-Mengen zu laufen, kann man zum Teil eine For-Schleife mit index-basiertem Zugriff nutzen, z.B. beim Vektor. Besser ist aber die Nutzung von Iteratoren, die u.a. den Vorteil hat mit allen Containern zu funktionieren. Eine häufig noch bessere Variante ist die Nutzung von Algorithmen oder Ranges.

Leider kann man mit Algorithmen nicht alle Anforderungen abdecken, so dass man immer mal wieder auf eine For-Schleife mit Iteratoren zurückfallen muß – nur leider sind Iterator-Schleifen für den Anfänger nicht ganz trivial. Ist man in so einer Situation, und will man über den gesamten Container iterieren, so gibt es in C++11 eine neue Form der For-Schleife, die diesen Anwendungs-Fall viel einfacher abdeckt – sie wird „Range-Basierte For-Schleife“ genannt.

Bei der neuen C++11 For-Schleife definiert man zuerst im Schleifen-Kopf eine Variable (das Schleifen-Objekt), die die einzelnen Objekte im Container während der Durchläufe aufnimmt – und danach gibt man nur noch nach einem Doppelpunkt den Container an:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v { 2, 4, 8, 1 };

   for (int value : v)
   {
      cout << value << ' ';
   }
   cout << '\n';
}
```

```
Ausgabe
2 4 8 1
```

Hinweise:

- Im Prinzip verstehen wir dieses Kapitel noch nicht, da erst in späteren Kapiteln Container und Iteratoren eingeführt werden – dann wird diese Schleife auch nochmal vorgestellt. Trotzdem wollte ich die neue C++11 For-Schleife auch schon mit in diesem Kapitel einführen, um alle Schleifen an einer Stelle stehen zu haben. Kommen Sie später nochmal hierher zurück und lesen Sie das Kapitel erneut.
- Das Schleifen-Objekt kann auch mit „const“ (Kapitel 5.4) und/oder Referenzen definiert werden. Außerdem kann auch hier die automatische Typ-Deduktion mit „auto“ genutzt werden (Kapitel 5.3.1.5).

7.4 While-Schleife

Syntax

```
while (ausdruck)
   anweisung
```

Die While-Schleife wird solange wiederholt, wie der Ausdruck „ausdruck“ true ist.

Beispiel

```
int i = 0;
while (i<3)
{
   std::cout << i << ' ';
   ++i;
}
```

```
Ausgabe
0 1 2
```

Kopfgesteuert, d.h. kann 0-mal durchlaufen werden.

7.5 Do-Schleife

Syntax

```
do
   anweisung
while (ausdruck);
```

Die Do-Schleife wird solange wiederholt, wie der Ausdruck „ausdruck“ „true“ ist.

Beispiel

```
int i = 0;
do
{
   std::cout << i << ' ';
   ++i;
}
while (i<3);
```

```
Ausgabe
0 1 2
```

Im Gegensatz zur while Schleife wird die do Schleife immer mindestens einmal durchlaufen – sie ist fußgesteuert.

7.6 Break- und Continue-Anweisungen

7.6.1 Schlüsselwort „break“

Das Schlüsselwort „break“ in einer Schleife sorgt für einen sofortigen Abbruch der Schleife.

```
for (int i=0; i<10; ++i)
{
   if (i==5) break;
   cout << i;
```

```
}
```

Ausgabe
```
01234
```

Break verläßt nur die innerste Schleife – sollen mehrere Schleifen verlassen werden, so benutzt man in C++ typischerweise ein Label und „goto“, siehe Kapitel 7.8.

Hinweis – häufig wird „break“ in verbindung mit quasi *Endlos-Schleifen* (siehe Kapitel 7.3) für Schleifen mit einem Ausgang in der Mitte benutzt – siehe Kapitel 7.7.

7.6.2 Schlüsselwort „continue“

Das Schlüsselwort „continue“ in einer Schleife springt direkt an das Schleifenende – auch dies gilt nur für die innerste Schleife. Bei allen Schleifen wird daher nach dem „continue“ die Abbruchbedingung getestet, bevor mit dem nächsten Schleifendurchlauf fortgefahren wird. Bei der For-Schleife wird vorher noch der Update-Teil der For-Schleife ausgeführt.

Im folgenden Beispiel wird so z.B. die „5“ ausgelassen, d.h. nicht mit ausgegeben.

```
for (int i=0; i<10; ++i)
{
   if (i==5) continue;
   cout << i;
}
```

Ausgabe
```
012346789
```

Die folgende Do-Schleife lässt die Ausgabe der Zahlen „4“ bis „6“ aus.

```
#include <iostream>
using namespace std;

int main()
{
   int i = 1;
   do
   {
      ++i;
      if (i>3 && i<7) continue;
      cout << i;
   } while (i<9);
}
```

Ausgabe
```
23789
```

7.7 Schleife mit Ausgang in der Mitte

In der Praxis werden häufig Schleifen mit einem Ausgang in der Mitte benötigt. Die typische Aufgabe, bei der sowas notwendig ist, sieht folgendermassen aus:

1. (Nächsten) Wert holen, berechnen, initialisieren oder so
2. Hat es geklappt, oder ist Wert okay?

Wenn nein -> Schleifen-Ende
3. Wert verarbeiten
4. Sprung zu 1.

Wollte man dies z.B. mit einer normalen While-Schleife implementieren, müßte man z.B. folgendes machen:

```
get-value
while (value-okay?)
{
   value-verarbeiten
   get-value
}
```

Da hier meistens die beiden „get-value“ Code-Stücke identisch sind, würde dies zu einer Code-Verdopplung führen. Dies ist sehr unschön, denn z.B. bei einer Änderung an dem „get-value“ Code muß diese synchron an zwei Stellen durchgeführt werden – und das ist fehleranfällig. Außerdem ist dieser Code nicht gut zu lesen und zu verstehen. Darum gibt es das Software-Prinzip DRY – daher der Versuch jede Code-Verdopplung zu vermeiden.

Hier führt das DRY-Prinzip zu einer Schleife mit Ausgang in der Mitte:

```
end-los-schleife
{
   get-value
   verlasse-schleife wenn value-not-okay?
   value-verarbeiten
}
```

Folgendes Beispiel soll das verdeutlichen, ohne dabei sonderlich sinnvoll zu sein. Reale Beispiele werden wir in den späteren Kapiteln noch mehrfach kennenlernen.

```
#include <iostream>
using namespace std;

int main()
{
   int i = 1;
   for (;;)
   {
      i += 2;
      if (i>8) break;
      cout << i;
   }
}
```

Ausgabe
357

7.8 Labels und „goto“

Es existiert ein „goto“ in C++, das man aber nicht benutzen sollte, da die Verwendung von Goto‘s zu sogenanntem Spaghetti-Code führt.

Es gibt aber ein Einsatzgebiet, für das manche C++ Programmierer den Einsatz von Goto’s für gerechtfertigt halten: das Springen aus mehreren Schleifen, da dies mit „break“ nicht funktioniert, denn „break“ verläßt nur die innerste Schleife.

Um ein Goto zu benutzen, muss ein Sprungziel in Form eines Labels definiert werden. Dieses Label ist ein Symbol (d.h. ein erlaubter C++ Name) mit dem Postfix „:“. Zum Springen zum Label muss dann hinter dem „goto“ nur der Label-Name angegeben werden.

Vergleichen Sie die folgenden drei Beispiele, die im Prinzip alle zwei verschachtelte Schleifen darstellen, wobei bei den ersten beiden Beispielen das Label „label“ nicht genutzt wird:

1. Einfach zwei verschachtelte Schleifen
2. Innere Schleife mit „break“
   Sie können sehen, dass das Break nur die innerste Schleife beendet.
3. Innere Schleife mit „goto“ aus beiden Schleifen heraus

Einfach zwei verschachtelte Schleifen

```
#include <iostream>
using namespace std;

int main()
{
   const int limit = 8;

   cout << "Vor Schleife\n";
   for (int i=0; i<limit; ++i)
   {
      cout << "> ";
      for (int j=0; j<limit-i; ++j)
      {
         cout << i*j << ' ';
      }
      cout << " <\n";
   }

   cout << "Vor Label\n";
   label:
   cout << "Nach Label\n";
}
```

Ausgabe

```
Vor Schleife
> 0 0 0 0 0 0 0 0  <
> 0 1 2 3 4 5 6  <
> 0 2 4 6 8 10  <
> 0 3 6 9 12  <
> 0 4 8 12  <
> 0 5 10  <
> 0 6  <
> 0  <
Vor Label
Nach Label
```

Innere Schleife mit „break“

```
#include <iostream>
using namespace std;

int main()
{
   const int limit = 8;

   cout << "Vor Schleife\n";
   for (int i=0; i<limit; ++i)
   {
      cout << "> ";
      for (int j=0; j<limit-i; ++j)
      {
         if (i*j > 10) break;
         cout << i*j << ' ';
```

```
        }
        cout << " <\n";
    }

    cout << "Vor Label\n";
    label:
    cout << "Nach Label\n";
}
```

Ausgabe

```
Vor Schleife
> 0 0 0 0 0 0 0 0  <
> 0 1 2 3 4 5 6  <
> 0 2 4 6 8 10  <
> 0 3 6 9  <
> 0 4 8  <
> 0 5 10  <
> 0 6  <
> 0  <
Vor Label
Nach Label
```

Innere Schleife mit „goto“

```
#include <iostream>
using namespace std;

int main()
{
    const int limit = 8;

    cout << "Vor Schleife\n";
    for (int i=0; i<limit; ++i)
    {
        cout << "> ";
        for (int j=0; j<limit-i; ++j)
        {
            if (i*j > 10) goto label;
            cout << i*j << ' ';
        }
        cout << " <\n";
    }

    cout << "Vor Label\n";
    label:
    cout << "Nach Label\n";
}
```

Ausgabe

```
Vor Schleife
> 0 0 0 0 0 0 0 0  <
> 0 1 2 3 4 5 6  <
> 0 2 4 6 8 10  <
> 0 3 6 9 Nach Label
```