Variablen

Will man Datenobjekte speichern, sie manipulieren und/oder später wiederverwenden, muss man dafür Variable mit Namen definieren. Bei statisch stark typisierten Programmiersprachen wie C, C++ muss man für jede Variable den Datentyp explizit angeben, den sie speichern soll. Das geschieht z.B. durch eine Variablen**definition**:

```
int x, y;       // x, y sind 2 int-Variable
double e;       // e ist eine double-Variable
```

Bei Funktionsdefinitionen muss in C++ der Rückgabetyp und die Typen aller Argumente angegeben werden:

```
int square(int x)    { return x*x; }
// square ist eine Funktion mit int-Argument und int-Ergebnis
```

Eine Variable, die so definiert wird, hat u.U. noch keinen definierten Startwert. Greift man auf eine uninitialisierte Variable zu, so ist das in C++ **Undefined Behavior**, d.h. das Programm darf in diesem Fall alles machen (abstürzen, falsch rechnen…). **Es wird daher dringend empfohlen, alle Variablen bei ihrer Definition sofort zu initialisieren!** Folgende Schreibweisen sind möglich:

```
int x = 1, y = -5;          // 1) in C/C++ seit Beginn
int x{1}, y{-5};            // 2) in C++ seit C++11
int x = {1}, y = {-5};      // 3) in C++ seit C++11
int x(1), y(-5);            // 4) in C++ seit C++20
```

Es gibt in C++ überhaupt keine Einigkeit, welche der verschiedenen Schreibweisen man für die Initialisierung verwenden soll. Ich verwende für die Initialisierung meistens die geschwungenen Klammern ohne Gleichheitszeichen (Variante 2), weil das die offiziell empfohlene Schreibweise ist, obwohl sich niemand daran zu halten scheint.

Bei der Initialisierung mit geschwungenen Klammern (2, 3) ist eine erforderliche Typumwandlung nur in einen größeren Typ erlaubt und nicht in einen kleineren Typ (narrowing Conversion):

```
double x{1};        // ok,   int -> double
int x{1.5};         //Fehler: double -> int ist narrowing Conversion
```

Bei der Initialisierung mit runden Klammern (4) sind alle Umwandlungen erlaubt:

```
double x{1};        // ok,   int -> double
int x{1.5};         //ok    double -> int ist narrowing Conversion
```

Auch wenn man die Initialisierung oft mit einem Gleichheitszeichen schreibt, ist es dennoch **keine Zuweisung**, sondern eben eine **Initialisierung**. Dabei sind Dinge erlaubt, die bei einer Zuweisung nicht erlaubt sind:

```
const double pi = 3.14159;      // initialisiere eine const double Konstante

const double pi;                // Konstante ohne definierten Inhalt ist sinnlos!
pi = 3.14159;                   // Fehler: pi ist konstant. Deshalb!
```

`auto` als Typangabe in Initialisierungen

Seit C++11 kann man als Typangabe **bei einer Initialisierung** auch `auto` schreiben. Dann versucht C++ den Typ aus dem Initialisierer zu erraten:

```
auto a = 1, b{2}, c(2);         // a,b,c sind dadurch int
auto x = 1.0, y{x}, z(x+y);     // x,y,z sind dadurch double (seit C++20)
auto z;                         // Fehler: C++ hat keinen Anhaltspunkt
auto x = 1, y = 1.0;
      // Fehler: Alle Variablen einer Definition müssen denselben Typ haben!
```

Gleich nach Einführung dieses `auto`-Typs wurde dieser fast überall verwendet („aaa“ „almost always auto“). Die Euphorie ist dann ins Gegenteil umgeschlagen („always avoid auto“). Es gibt aber sehr oft gute Gründe für das `auto`:

1) Eine Variable soll genau denselben Typ haben wie eine andere, ohne dass man weiß, welcher Typ es wirklich ist: `auto x = y;`
2) Man müsste den Typ herausfinden und ist zu faul: `auto x = f(…);`
3) Niemand kennt den Typ (z.B. von Lambdas): `auto f = [](…){…};`
4) Das Schreiben des Typs erzeugt starke Schmerzen:
   ```
   std::list<std::complex<double>>::const_reverse_iterator
        it = v.crbegin();
   auto it{v.crbegin()};
   ```
5) Man schreibt Range-based for-Loops: `for (auto x: v)`
6) Man schreibt generische Lambdas, die mit vielen Typen funktionieren sollen:
   `[](auto x) {…}`
7) Man schreibt Templates und verwendet C++20-Features.

   …

Gültigkeit von Variablen

Variablen sind nur innerhalb des Scopes bekannt, in dem sie definiert sind. Sind Variablen **außerhalb aller Scopes** definiert, so sind Sie **global** und gelten im gesamten Programm. Die Lebensdauer von globalen Objekten beginnt vor `main()` und endet nach `main()`. Die Lebensdauer von nichtglobalen Objekten beginnt an der Stelle ihrer Definition und endet bei der schließenden Klammer `}` des enthaltenden Scopes.

Wie sucht C++ nach Namen

Wird ein Name mit Namespace angegeben (`std::cout`), so beginnt die Suche in diesem Namespace. Bei Namen ohne Namespace beginnt die Suche an der Stelle, wo dieser Name steht. Wird im umgebenden Scope dieser Name nicht gefunden, so wird die Suche im nächstäußeren Scope fortgesetzt. Ist auch dort Name nicht vorhanden, wird im nächstäußeren Scope weitergesucht usw.

Wir bei der Suche in einem Scope dieser Name gefunden, so werden alle gleichlautenden Namen dieses Scopes in Betracht gezogen.

Wird der Name einer Funktion gesucht, so sucht C++ zuerst nach dem Namen wie eben geschildert und macht dann weitere Suchläufe in den Typdefinitionen aller Argumente (ADL = Argument Dependend Lookup). Alle gefundenen richtigen Namen werden in Betracht gezogen:

```
std::cout << Bruch{1, 2} => operator<<(std::cout, Bruch{1,2})
```

C++ sucht zuerst im Namespace `std` und dann auch in den Typdefinitionen von `std::ostream` und `Bruch` nach der Funktion `operator<<()`.