Modernes C++

...für Programmierer

Unit 03: Datentypen & Deklarationen

by

Dr. Günter Kolousek

12. September 2019

Überblick

- Überblick Datentypen
- Fundamentale Datentypen
- Deklarationen vs. Definitionen
- Ausdruck vs. Anweisung
- Gültigkeitsbereich
- Initialisierung
- Objekte, Werte, Lebenszeit
- Konvertierungen
- `using`

Überblick über Datentypen

- eingebauten Datentypen (engl. built-in)
  - fundamentale Datentypen
  - Typen auf Basis von Deklarationsoperatoren
    - Zeigertypen: `int*`,...
    - Array-Typen: `char[]`,...
    - Referenztypen: `double&`,...
- benutzerdefinierte Datentypen
  - Datenstrukturen und Klassen: `struct`, `class`
  - Aufzählungstypen: `enum` und `enum class`

Fundamentale Datentypen

- arithmetische Typen
  - integrale Typen: → rechnen & bitweise logische Operationen
    - `bool`
    - Zeichentypen: `char`, `wchar_t`,...
    - Ganzzahltypen: `int`, `long long`,...
  - Gleitkommazahltypen:
    - `float`
    - `double`
    - `long double`
- `void`

bool

- `true`, `false`
- in arithmetischen & bitweisen Ausdrücken → Konvertierung zu `int`
  - `true` → `1`
  - `false` → `0`
- Konvertierung zu `bool`
  - ”alles ungleich 0” wird als `true` betrachtet (implizit konvertiert)
  - ”alles gleich 0” wird als `false` betrachtet (implizit konvertiert)

bool - 2

```cpp
#include <iostream>  // bool.cpp
using namespace std;
int main() {
    cout << true << endl; // 1
    cout << false << endl;  // 0
    cout << boolalpha;  // yet another I/O manip.
    cout << true << endl; // true
    cout << false << endl;  // false
    cout << noboolalpha << true << endl;  // 1
    cout << false << endl;  // 0
    cout << true + 1 << endl;  // 2
    cout << (true & 3) << endl;  // 1
}
```

bool - 3

```cpp
#include <iostream>  // bool2.cpp
using namespace std;
int main() {
    bool b1=42;  // b1 == true !!
    //bool b2{42};  // Fehler!

    int i=3;
    while (i) {
        cout << i << ' ';  // 3 2 1
        i--;
    }
}
```

Zeichentypen

- `char` ... mit oder ohne Vorzeichen (implementierungsabhängig)
  - *meist* 8 Bit
  - `sizeof(char) == 1`
- `unsigned char` ... ohne Vorzeichen
- `signed char` ... mit Vorzeichen
  - nicht spezifiziert (z.B. 1er oder 2er Komplement)
    - seit C++14 bijektive Abbildung zu `unsigned char`!
  - **meist:** $[-128, 127]$
- `wchar_t` ... implementierungsabhängig
- `char16_t` ... 16-Bit-Zeichensätze
- `char32_t` ... 32-Bit-Zeichensätze

Zeichenliterale

- einfache Hochkommas, z.B. `'a'`
- Escape-Zeichen ist `\`:
  - `\n`, `\t`, `\\`, `\'`, `\"`
  - `\0` (Nullzeichen),...
- Unicode-Zeichen
  - `U'\UCAFEDEAD` ... `char32_t` (UTF-32)
  - `u'\uDEAD'` ≡ `U'\U0000DEAD'` ... `char16_t` (UTF-16)
  - `u8'a'` ... `char` (ab C++17)

Ganzzahltypen

- Einteilung in vorzeichenbehaftet und vorzeichenlos
  - `int` … vorzeichenbehaftet; Synonym: `signed int`
  - `unsigned int` … Synonym: `unsigned`
- Einteilung nach Größen
  - `short int` … Synonym: `short`
  - `int`
  - `long int` … Synonym: `long`
  - `long long int` … Synonym: `long long`

Ganzzahltypen – 2

- `++i` vs. `i++` … preinkrement vs. postinkrement

```
int a{0};
int b{0};
b = ++a;  // a == 1, b == 1
b = a++;  // a == 2, b == 1
```

- `~`, `|`, `&`, `^`, `>>`, `<<` … bitweise

```
int a{1};
a = a | 1 << 2;  // a == 0b101
```

- `+=`, `-=`, usw. … zusammengesetzte Zuweisungen

Zahlenliterale

- dezimal: `123`, `123'456'789`
- binär: `0b1101`, `0b1111'0000'0000'0000`
- oktal: `0123`
- hexadezimal: `0xCAFE`
- Suffix `l` oder `L`: `123L`
- Suffix `ul`, `lu`, `Lu`,...: `123UL`
- Suffix `ll`, `LL`: `123LL`
- Suffix `llu`, `llU`,...: `123LLU`

Zahlenliterale – 2

```cpp
#include <iostream>  // numbers.cpp
using namespace std;
int main() {
    cout << 123'456'789 << endl;
    cout << hex << 0xFF << endl;
    cout << 0777 << ' ' << oct << 0777 << endl;
    cout << showbase << hex << 0xCAFE << endl;
    cout << dec << 0xff << endl;
}
```

```
123456789
ff
1ff 777
0xcafe
255
```

Formatierung der Ausgabe

```cpp
#include <iostream>  // outnums.cpp
#include <iomanip>  // setw, setfill,...
using namespace std;
int main() {
    cout << left << setw(5) << 3 << 'm' << endl;
    cout << 3 << 'm' << endl;  // reset!
    cout << internal << setw(5)<< -3<< 'm'<< endl;
    cout << right << setw(5) << -3 << 'm' << endl;
    cout<< setfill('*')<< setw(5)<< 3<< 'm'<< endl;
}
```

```
3     m
3m
-    3m
    -3m
****3m
```

Formatierung der Ausgabe – 2

```cpp
#include <iostream>  // outnums2.cpp
#include <iomanip>  // setw, setfill,...
using namespace std;
int main() {
    cout << uppercase << hex << 0xcafe << endl;
    double pi = 3.1415926;
    cout << pi << ' ';
    cout << setprecision(3) << pi << ' ';
    cout << showpos << pi << endl;
    cout << showpoint << setprecision(10) << 2.78
         << endl << pi << endl;
}
```

```
CAFE
3.14159 3.14 +3.14
+2.780000000
+3.141592600
```

Formatierung der Ausgabe – 3

- Alle Manipulatoren mit Argumenten $\rightarrow$ `<iomanip>`
- `setw`
  - nur für nächste Ausgabe!
  - minimale Breite wird angegeben
- Ausrichtung
  - Default ist `right`
  - `intern` nur für numerische Werte
- Groß/Kleinbuchstaben bei Hexadezimalzahlen: `uppercase` und `nouppercase`
- `setprecision`
- Anzeige des Vorzeichens: `showpos` und `noshowpos`

Gleitkommazahlen

- Größen
  - `float`
  - `double`
  - `long double`
- Literale
  - `10.0` … `double`
  - `10.0f` oder `10.0F` … `float`
  - `3.14l` oder `3.14L` … `long double`
  - `-2.78e-3` … $-2.78 \cdot 10^{-3}$

Größen

- sind implementierungsabhängig!
- `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
- `1 <= sizeof(bool) <= sizeof(long)`
- `sizeof(char) <= sizeof(wchar_t) <= size(long)`
- `sizeof(float) <= sizeof(double) <= sizeof(long double)`

Größen – 2

```
#include <iostream>  // sizes.cpp
#include <limits>
using namespace std;
int main() {
    static_assert(sizeof(int) >= 4, "size(int)<4");
    cout << "1: " << sizeof(1) << endl;
    cout << "1L: " << sizeof(1L) << endl;
    cout << "1LL: " << sizeof(1LL) << endl;
    cout << "max. float: " <<
      numeric_limits<float>::max() << endl;
    cout << "max. double: " <<
      numeric_limits<double>::max() << endl;
    cout << "char signed? " <<
      numeric_limits<char>::is_signed << endl;
}
```

Größen – 3

Mögliche Ausgabe:

```
1: 4
1L: 4
1LL: 8
max. float: 3.40282e+38
max. double: 1.79769e+308
char signed? 1
```

decltype

auto ignoriert sowohl const als auch Referenzen → decltype(auto)

```
int i{1};
int& r{i};
auto ar{r};                    // int, nicht: int&
decltype(r) dr{r};             // int& C++11/14
decltype(auto) dra{r};         // int& C++14
const int k{42};
auto ak{k};                    // int, nicht: const int
decltype(k) dk{k};             // const int, C++11/14
decltype(auto) dka{k};         // const int, C++14
```

Template-Konstanten ab C++ 14

```
template<typename T> constexpr T
pi{3.14159265358979323846264338328L};
```

- `pi<float>` π mit `float` Genauigkeit
- `pi<double>` π mit `double` Genauigkeit