Modernes C++

...für Programmierer

Unit 03: Datentypen & Deklarationen

by

Dr. Günter Kolousek

16. September 2018

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 → `<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
```

Deklaration vs. Definition

- Deklaration
  - Zuordnung von Name zu Typ
  - ist eine Anweisung
  - $\rightarrow$ Gültigkeitsbereich
  - $\rightarrow$ Lebensdauer
- Definition
  - ist eine Deklaration
  - enthält alle Angaben um Namen zu benutzen
    - d.h. alles was der Linker benötigt!
    - bei Variable wird Speicher reserviert
    - bei Funktion ist Funktionsrumpf vorhanden
    - Klasse (Struktur) vollständig vorhanden

Deklaration vs. Definition – 2

```cpp
#include <iostream>  // declarations.cpp
using namespace std;
constexpr double get_r() {
  return 3;
}

struct User;  // no definition
extern int err_nr;  // no definition

int main() {
    char ch;
    auto cnt{1};
    const double e{2.7182};
    constexpr double pi{3.1415};
    constexpr double U{2 * get_r() * pi};
}
```

Ausdruck vs. Anweisung

- Ausdruck hat Wert
  - z.B.: `1 + 2`
  - z.B.: `a = 3`
  - z.B.: `if (a = 0) cout << a;`
- Anweisung hat keinen Wert
  - einfache Anweisungen
    - Ausdruck + `;` $\equiv$ Anweisung, z.B.: `2 + 3;`
  - zusammengesetzte Anweisungen (`if`, `while`, `switch`,...)
    - teilweise mit `;` (z.B. `class` oder `struct`)

Gültigkeitsbereich (engl. scope)

- in der Regel gültig ab Deklaration
- verschiedene Arten
  - lokal: innerhalb von `{ }`
  - Klasse: gültig in der gesamten Klasse
  - Namespace: innerhalb eines Namenraumes
  - global: bis Ende der Datei
  - Anweisung: innerhalb von `( )` einer `for`, `while`, `if`, `switch`, bis Ende der Anweisung
  - Funktion: gültig in der gesamten Funktion; nur Labels

Gültigkeitsbereich – 2

```
#include <iostream>  // scope.cpp
using namespace std;
int x;  // global

int main() {
    cout << x << endl;  // 0
    int x;  // local (global x shadowed)
    x = 1;  // local x
    {
        int x=x;  // de facto uninitialized!
        cout << x << endl;  // e.g.: -1081928100
        x = 2;
    }
    x = 3;  ::x = 1;
    cout << x << " " << ::x << endl;  // 3 1
}
```

Initialisierung

```cpp
#include <initializer_list>
using namespace std;
int main() { // init.cpp
    // direct-list-initialization
    // explicit and non-explicit constructors
    int i1{1};  // recommended!!!
    // copy-list-initialization
    // only non-explicit constructors
    int i2={2};
    int i3=3;  // don't do it!
    int i4(4);  // also: no!
    auto i5{5};  // be careful of "old" compilers
    auto i6={6}; // not the same: see next slide!
    auto i7=7;  // yes but not needed any more
    auto i8(8);  // almost no...
}
```

Initialisierung – C++17

In ”neuen” Compilerversionen auch bei C++ 11 und C++ 14!

→ auf Empfehlung des Standardkomitees!!

```
#include <iostream>
#include <initializer_list>

using namespace std;

int main() {
    auto a={1, 2, 3};  // initializer_list<int>
    for (auto e : a) cout << e << ' ';
    auto b={4};
    for (auto e : b) cout << e << ' ';
    auto c{42};
    cout << c << endl;
    // auto d{1, 2, 3};  // error!
}
```

```
1 2 3 4 42
```

Initialisierung – 2

```
#include <initializer_list>
class X {};  // init2.cpp

int main() {
    // int i1{1.5};  // compile error: narrowing...
    // int i2={2.5};  // compile error...
    int i3=3.5;  // i3 == 3 → narrowing
    int i4(4.5);  // i4 == 4
    int i5();  // function declaration!!
    X x(X());  // ditto!
}
```

Initialisierung – 3

```cpp
#include <iostream>  // init3.cpp
#include <vector>
using namespace std;

int main() {
    vector<int> v1(10);
    cout << v1.size() << " " << v1[0] << endl;
    vector<int> v2(1, 10);
    cout << v2.size() << " " << v2[0] << endl;
    //vector<int> v3{1, 10};  // until C++14
    vector v3{1, 10};  // since C++17
    cout << v3.size() << " " << v3[0] << endl;
}
```

```
10 0
1 10
2 1
```

Initialisierung – 4

- wenn keine Initialisierungsspezifierer vorhanden, dann:
  - wenn global, Namespace, `static`, dann: initialisiert mit `{}`
    - bei benutzerdefinierten Typ: Default-Konstruktor
  - wenn lokal oder am Heap, dann:
    - benutzerdefinierter Typ und Default-Konstruktor: initialisiert
    - anderenfalls: nicht initialisiert

Initialisierung – 5

```cpp
#include <iostream>  // init4.cpp
#include <vector>
using namespace std;
int x;  // initialized with {}

int main() {
    int x;  // not initialized
    char buf[1024];  // not initialized

    int* p{new int};  // *p not initialized
    string s;  // s == ""
    vector<int> v;  // v == {}
    string* ps{new string};  // *ps == ""
}
```

Initialisierung – 6

```
#include <complex>  // init5.cpp
#include <vector>
using namespace std;

int main() {
    int a[]{1, 2, 3};  // array-initializer
    struct S {
        int i;
        string s;
    };
    S s{1, ”hello”};  // struct-initializer
    complex<double> z{0, 1};  // use constructor
    vector<int> v{1, 2, 3};  // list-initializer
}
```

Objekte und Werte

- Objekt: zusammenhängender Speicherbereich
- L-Wert (lvalue): Ausdruck der auf Objekt verweist
  - linke Seite einer Zuweisung, z.B. `i = 5;`
  - Faustregel: kann & angewandt werden → lvalue
  - aber: Konstanten sind lvalues, aber nicht auf linker Seite
- R-Wert (rvalue):
  - ”kein lvalue”, z.B. ein Wert, der von Funktion zurückgegeben wird, z.B. `int i; i = f();`
  - kann aber auf auf linker Seite stehen: `g() = 3;`

Objekte und Werte – 2

```cpp
#include <iostream>  // lrvalues.cpp
using namespace std;

int x{0};

int f() { return 0; }
int& g() { return x; }

int main() {
    // f() = 2;  // error: lvalue required...
    g() = 1;
    cout << x << endl;
}
```

```
1
```

Objekte und Werte – 3

```cpp
#include <iostream>  // lrvalues2.cpp
using namespace std;

int main() {
    int i;
    i = 4;
    // 4 = i;  // error: lvalue required...
    (i + 1) = 5;  // error!
    const int j{6};  // j is an lvalue
    // j = 7;  // error!
}

int& h() {
  return 2; //error: invalid init...from an rvalue
}
```

Lebensdauer von Objekten

Gibt an, wann ein Objekt ”zerstört” wird

- automatisch: wenn es Gültigkeitsbereich verlässt (lokal)
- statisch: enden mit Programmende (global, Namensraum, `static`)
- Freispeicher (engl. free store, heap): bei `delete`
- temporäre Objekte: z.B. Zwischenergebnisse in einer Berechnung `a * (b + c * d)`
  - enden mit Ende des vollständigen Ausdruckes (nicht Teil eines anderen Ausdruckes)
  - außer wenn an Referenz gebunden
- threadlokal: Objekte, die `thread_local` deklariert sind, enden mit Threadende

Implizite Konvertierungen

- Aufweitung der integralen Datentypen (engl. integral promotions, kurz: promotions):
  - `char`, `signed char`, `unsigned char`, `short`, `unsigned short` **zu** `int`, `unsigned`
  - `char16_t`, `char32_t`, `wchar_t` bzw. `enum` **zu** `int`, `unsigned`, `long`, `unsigned long`, `unsigned long long`
  - `bool` zu `int`
- Konvertierungen auf gemeinsamen Typ

Implizite Konvertierungen – 2

```cpp
#include <iostream>  // conv.cpp
using namespace std;

int main() {
    char a{'0'};
    char b{'0'};  // ASCII decimal: 48
    cout << a << ' ' << sizeof(a) << endl;
    cout << a + b << ' ' << sizeof(a + b) << endl;
}
```

```
0 1
96 4
```

Implizite Konvertierungen – 3

```cpp
#include <iostream>  // conv2.cpp
using namespace std;

int main() {
    long long int ll{};
    char c{};

    cout << "size(ll) = " << sizeof(ll) << endl;
    cout << "size(c) = " << sizeof(c) << endl;
    cout << "size(ll+c) = "<<sizeof(ll + c)<<endl;
}
```

```
sizeof(ll) = 8
sizeof(c) = 1
sizeof(ll+c) = 8
```

Implizite Konvertierungen – 4

```cpp
#include <iostream>  // conv3.cpp
using namespace std;

int main() {
    int i{};
    i = 3.5;
    cout << i << endl;  // ok, it's expected
    char c;
    c = 128;  // undef behaviour if 8bits signed
    cout << static_cast<int>(c) << endl;  // explicit
}
```

```
3
-128
```

Explizite Konvertierungen

- Regel: ”don’t cast at all!”
- `static_cast` → das Mittel der Wahl
  - liefert Wert des neuen Typs
  - nicht bei Downcasts verwenden (da keine Überprüfung)
  - kein Overhead zur Laufzeit
- `dynamic_cast`
  - konvertiert Pointer und Referenzen innerhalb von Vererbungshierarchien
  - liefert `nullptr` zurück, wenn nicht konvertierbar
  - außer bei Referenzen → `std::bad_cast` Exception
- `const_cast`
  - zum ”Wegcasten” von `const`
  - kein Overhead zur Laufzeit
- `reinterpret_cast`
  - Bitpattern des Werts wird als neuer Typ interpretiert
  - kein Overhead zur Laufzeit

`using`

- `using`-Direktive
  - alle Bezeichner des angegebenen Namensraumes im aktuellen Gültigkeitsbereich
  - z.B. `using namespace std;`
- Typalias (engl. type alias declaration)
  - neuer Name für bestehenden Typ
- `using`-Deklaration
  - Verwendung eines bestehenden Namens aus anderem Namensraum

using-Typalias

```cpp
#include <iostream>  // typealias.cpp
#include <vector>
using namespace std;

int main() {
    using IntStack = std::vector<int>;
    IntStack stack{};
    stack.push_back(1); stack.push_back(2);
    cout << stack.back() << endl;
    stack.pop_back();
    cout << stack.back() << endl;
    stack.pop_back();
}
```

using-Deklaration

```cpp
#include <iostream>  // usingdecl.cpp
#include <vector>

int main() {
    // equiv to: using vector = std::vector;
    using std::vector;

    using std::cout;  // cout is no type!
    vector<int> vec{1, 2, 3};
    cout << vec.size() << std::endl;
}
```

```
3
```