Modernes C++

...für Programmierer

Unit 08: Klassen

by

Dr. Günter Kolousek

6. November 2019

Überblick

- Strukturen und einfache Klassen
- Initialisierung
- Direct vs. Copy Initialisierung
- RVO
- Move und Löschen von Methoden
- `const` und `constexpr` Methoden
- Methoden für lvalues und rvalues

Strukturen

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

struct Circle {
    double r{1};
    double x{};
    double y;
};

int main() {
    Circle c;  // it's a ”POD” (plain old data type)!
    // therefore, no initialization!
    cout << c.r <<  ' ' << c.x << ' ' << c.y << endl;
    c = Circle{3, 2, 1};
    cout << c.r <<  ' ' << c.x << ' ' << c.y << endl;
}
```

```
1 0 6.9531e-310
3 2 1
```

Strukturen - 2

```cpp
#include <iostream>  // structs2.cpp
using namespace std;

// convention: use struct for simple aggregates
struct Point {
    double x;
    double y;
};

int main() {
    Point p{};// also a POD but initialized explicitly
    cout << p.x <<  ' ' << p.y << endl;
    Point points[]{{}, {1, 1}, {2, {}}};
    cout << points[0].x <<  ' ' << points[1].x << ' '
      << points[2].x << endl;
}
```

```
0 0
0 1 2
```

Strukturen - 3

```cpp
#include <iostream>  // structs3.cpp
using namespace std;

// should be inside .h file
struct Counter {
    static int cnt;
};

// should be inside .cpp file
int Counter::cnt{};
// without -> undefined reference to `Counter::cnt'

int main() {
    cout << ++Counter::cnt << endl;
}
```

```
1
```

Strukturen - 4

```cpp
#include <iostream>  // structs4.cpp
using namespace std;

struct Counter {
    inline static int cnt;
    // implicitly inline:
    constexpr static double pi{3.1415};
};

// Neither definition of cnt nor pi!

int main() {
    cout << ++Counter::cnt << endl;
    cout << Counter::pi << endl;
}
```

```
1
3.1415
```

Einfache Klasse

```
#include <iostream>  // simpleclass.cpp
#include <tuple>
using namespace std;
class Circle {
    double r{1};  double x;  double y;
  public:
    Circle()=default;  // no POD!
    Circle(double r, double x, double y)
      : r{r}, x{x}, y{y} {}
    auto data() const { return make_tuple(r, x, y); } };
int main() {
    auto [r, x, y] = Circle{}.data();
    cout << r <<  ' ' << x << ' ' << y << ", ";
    tie(r, x, y) = Circle{3, 2, 1}.data();
    cout << r <<  ' ' << x << ' ' << y << ", ";
    auto circ = Circle{3, 2, 1}.data();
    cout << get<0>(circ) << ' ' << get<1>(circ) << ' '
      << get<2>(circ)<<endl;
}
```

```
1 0 0, 3 2 1, 3 2 1
```

Initialisierung

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

struct Distance {
    int len;
    Distance() {}  // user-declared and defined!
};
struct Distance2 {
    int len;
    Distance2()=default;  // you are free to choose...
};
int main() {
    Distance d1;
    Distance d2{};
    cout << d1.len << ", " << d2.len << endl;
    Distance2 d3;
    Distance2 d4{};
    cout << d3.len << ", " << d4.len << endl;
}
-1294638960, 21924
-737770272, 0
```

Initialisierung – 2

```
#include <iostream>  // classinit2.cpp
using namespace std;

class Distance {
    double len;
  public:
    Distance() : Distance{0} {} // delegating cons!
    Distance(double len) : len{len} {
        cout << this->len;
    } };
int main() {
    Distance d1;
    Distance d2{1};  // direct init
    Distance d3={2};  // copy init (same as: d3=2)
}

012
```

Initialisierung – 3

direct Konstruktor wird direkt aufgerufen

- bester Konstruktor (überladen) wird gesucht
- u.U. implizite Konvertierung
- sowohl `explicit` als auch nicht-`explicit` Konstrukturoen werden verwendet!

copy

- **vor** C++17: Kopierkonstruktor wird aufgerufen
  - temporäres Objekt wird mittels Initialisierungswert angelegt (implizite Konvertierung eingeschlossen)
  - benanntes Objekt wird mittels Kopierkonstruktor initialisiert
  - nicht-`explicit` deklarierte Konstruktoren werden verwendet!
- **ab** C++17
  - nicht-`explicit` deklarierte Konstruktoren werden verwendet!

Direct vs. Copy

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

int main() {
    int x1={0};  // copy-initialization
    int x2{0};  // direct-initialization
    auto x3={0};  // initializer_list<int>
    auto x4{0};  // int, since C++17
                 // recommended since C++14:
    cout << x1 << endl;
    cout << x2 << endl;
    //cout << x3 << endl;  // does not compile!
    cout << x4 << endl;
}
```

Direct vs. Copy – 2

```cpp
#include <iostream>  // directcopy2.cpp
using namespace std;
class Distance {  double len;
  public:
    Distance()=default;
    explicit Distance(double len)
      : len{len} { cout << "ctor" << endl; }
    Distance(const Distance& other) : len{other.len} {
        cout << "copy ctor" << endl;  }
    ~Distance() { cout << "dstor" << endl; }  };
int main() {
    Distance d1;
    Distance d2{1};  // direct init
    // copy init -> compiler error because of explicit
    // Distance d3 = 2;
} // C++17: *no* output of "copy ctor"
```

```
ctor
dstor
dstor
```

Direct vs. Copy – 3

```
#include <iostream>  // directcopy3.cpp
#include <initializer_list>
using namespace std;
class Distance {
    double len{};
  public:
    Distance(double len)
      : len{len} { cout << "ctor" << endl; }
    Distance(initializer_list<double> l)
      : len{*l.begin()} {
        cout << "initializer ctor" << endl;  }
    ~Distance() { cout << "dtor" << endl; }  };
int main() {
    Distance d1{1};  Distance d2 = {1};  }
```

```
initializer ctor
initializer ctor
dtor
dtor
```

RVO

$\rightarrow$ RVO (return value optimization)

- seit C++ 11 im Standard (aka copy elision)!
- seit C++ 17 in vielen Fällen verpflichtend!

4 grundlegende Arten

- RVO
- Named RVO
- Passing a temporary by value
- Throwing and catching exceptions per value

RVO – 2

```
#include <iostream>  // rvo1.cpp
using namespace std;
class Distance {
    double len{};
  public:
    Distance(double len)
      : len{len} { cout << "ctor" << endl; }
    Distance(const Distance& other) : len{other.len} {
        cout << "copy ctor" << endl;  }
    ~Distance() {  cout << "dtor" << endl;  }  };
Distance one() { return Distance{1}; }
int main() {
    Distance d2{one()};  // direct init
    Distance d3=one();  // copy init
}
```

```
ctor
ctor
dtor
dtor
```

RVO – 3

```
#include <iostream>  // rvo2.cpp
using namespace std;
class Distance {
    double len{};
  public:
    Distance(double len)
      : len{len} { cout << "ctor" << endl; }
    Distance(const Distance& other) : len{other.len} {
        cout << "copy ctor" << endl;  }
    ~Distance() {  cout << "dtor" << endl;  }  };
Distance one() {
    Distance res{1};
    return res; }
int main() { Distance d2{one()};
             Distance d3=one();  }

ctor
ctor
dtor
dtor
```

RVO – 4

```
#include <iostream>  // rvo3.cpp
using namespace std;
class Distance {
    double len{};
  public:
    Distance(double len)
      : len{len} { cout << ”ctor” << endl; }
    Distance(const Distance& other) : len{other.len} {
        cout << ”copy ctor” << endl;  }
};
void one(Distance d) {
    cout << ”inside one” << endl; }
int main() {
    one(Distance{1});
}
```

```
ctor
inside one
```

Move

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

class Distance {
    double len;
  public:
    Distance()=default;
    Distance(double len)
      : len{len} { cout << "ctor: " << len << endl; }
    Distance(const Distance& other) : len{other.len} {
        cout << "copy ctor: " << len << endl;
    }
    Distance(Distance&& other) : len(other.len) {
        cout << "move ctor: " << len << endl;
    }
    ~Distance() {
        cout << "dstor" << endl;
    }
};
```

Move – 2

```
Distance one(int n) {
    Distance tmp1{1};  Distance tmp2{2};
    if (n > 2) return tmp1;  else return tmp2;  }
int main() {  Distance d1{one(1)};
              Distance d2=one(3);  }
```

```
ctor: 1
ctor: 2
move ctor: 2
dstor
dstor
ctor: 1
ctor: 2
move ctor: 1
dstor
dstor
dstor
dstor
```

No Move

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

class Distance {
    double len;
  public:
    Distance()=default;
    Distance(double len)
      : len{len} { cout << "ctor: " << len << endl; }
    Distance(const Distance& other) : len{other.len} {
        cout << "copy ctor: " << len << endl;
    }
    Distance(Distance&& other) = delete;
    ~Distance() { cout << "dstor" << endl;  }
};
```

No Move – 2

```
Distance one(int n) {
    Distance tmp1{1};  Distance tmp2{2};
    if (n > 2) return tmp1;  else return tmp2;  }
int main() {  Distance d1{one(1)};
              Distance d2=one(3);  }
```

```
ctor: 1
ctor: 2
copy ctor: 2
dstor
dstor
ctor: 1
ctor: 2
copy ctor: 1
dstor
dstor
dstor
dstor
```

const - Methoden

```
#include <iostream>
using namespace std;
class Data {  // const.cpp
    string data;
  public:
    Data(string data) : data{data} {}
    char* get_raw() { return data.data(); }
    // overloaded method:
    char const* get_raw() const { return data.data(); }
};
int main() {
    Data d1{"abc"};
    char* cstr{d1.get_raw()};
    cstr[1] = 'x';  cout << cstr << endl;  // -> axc
    const Data d2{"abc"};
    // invalid conversion from ‘const char*’ to ‘char*’:
    //cstr = d2.get_raw();
    const char* cstr2{d2.get_raw()};
}
```

constexpr - Methoden

- ab C++ 14 sind diese nicht mehr implizit `const`!
- Beachte die folgende globale Funktion:

```
constexpr Circle scale(const Circle& circ,
                       double factor) {
   Circle tmp{circ};
   tmp.set_radius(circ.get_radius() * factor);
   return tmp;
}
```

  `set_radius` kann nicht `const` sein, aber
  - `set_radius` kann `constexpr` sein und
  - Compiler kann `scale` zur Übersetzungszeit berechnen!
- Methoden können natürlich zusätzlich zu `constexpr` als `const` markiert werden.

lvalue vs rvalue

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
class Data {  // lvalue_vs_rvalue.cpp
    char* data;
    size_t size;
  public:
    // be aware of dangling pointer!
    Data(char* data, size_t size) : data{data}, size{size} {}
    char* get_data() const& {  // will be called on lvalue
        char* result{new char[size]};
        copy_n(data, size, result);
        return result;
    }
    char* get_data() && {  // will be called on rvalue
        char* tmp{data};
        data = nullptr;
        size = 0;
        return tmp;
    }
};
```

lvalue vs rvalue – 2

```
int main() {
    Data d1{const_cast<char*>(”abc”), 4};
    char* cstr{d1.get_data()};
    cstr[1] = 'x';  cout << cstr << endl;  // -> axc
    delete[] cstr;
    char* cstr2{Data{const_cast<char*>(”abc”), 4}.get_data()};
    //cstr2[1] = 'x';  does not work any more (it's read-only)
    cout << cstr2 << endl;  // -> abc
    // no delete here!!!
}
```

```
axc
abc
```