

Modernes C++

...für Programmierer

Unit 06: Funktionen

by

Dr. Günter Kolousek

Überblick

- ▶ Funktionsdeklarationen
- ▶ auto - Rückgabewert
- ▶ Referenzparameter & Defaultargumente
- ▶ Variable Anzahl an Argumenten
- ▶ Überladen von Funktionen
- ▶ Funktionszeiger, Funktionsobjekte, Lambdaausdrücke
- ▶ inline

Funktionsdeklaration – 1

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

double squared(double); // declarations!
void print(string msg);
int one();

int main() {
    // print("2^2 - 1 = "); // error!
    // cout << squared(2) - one() << endl; // error
}
```

Funktionsdefinition – 2

```
#include <iostream>
using namespace std; // functions2.cpp

double squared(double val) { return val * val; }
void print(string message) { cout << message; }
int one() { return 1; }

int main() {
    print("2^2 - 1 = ");
    cout << squared(2) - one() << endl;
}

2^2 - 1 = 3
```

auto - Rückgabewert

```
#include <iostream> // auto.cpp
using namespace std;
auto sum(int a, int b) -> decltype(a + b) {
    return a + b;
}

int main() {
    cout<< "Summe von 3 und 5: "<< sum(3,5)<< endl;
}
```

Summe von 3 und 5: 8

auto - Rückgabewert – 2

```
#include <iostream> // auto2.cpp
using namespace std;
auto sum(int a, int b) { // C++14
    return a + b;
}

int main() {
    cout<< "Summe von 3 und 5: "<< sum(3,5)<<endl;
}
```

Summe von 3 und 5: 8

auto - Rückgabewert – 3

```
#include <iostream> // auto3.cpp
using namespace std;
auto fac(int n) {
    if (n == 0)
        return 1; // each return: same type!
    else
        return n * fac(n - 1);
}
// do not reverse the logic of if because
// one return with concrete type must be first

int main() {
    cout << "fac(3): " << fac(3) << endl;
}

fac(3): 6
```

Referenzparameter

```
#include <iostream> // lvaluerefpar.cpp
using namespace std;
void incr(int& counter) {
    ++counter;
}

int main() {
    int counter{};
    incr(counter);
    cout << counter << endl;
    // incr(2); // error
}

1
```


Referenzparameter – 2

```
#include <iostream> // rvaluerefpar.cpp
using namespace std;
void incr(int&& counter) {
    ++counter;
}
void incr2(const int& counter) {
    // ++counter; // error
}
int main() {
    int counter{};
    // incr(counter); // error
    incr(2); // does not make sense!
    incr2(2); // does not make sense!
}
```

Defaultargumente

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

```
int getDefault() { return 0; }
```

// will be evaluated at runtime; =0 also possible

```
int incr(int counter=getDefault()) {
    return counter + 1;
}
```

```
int main() {
    cout << incr() << ' ';
    cout << incr(1) << endl;
}
```

1 2

Variable Anzahl von Argumenten

- ▶ mittels ... (nicht typsicher)
- ▶ mittels Funktionstemplates (statisch)
- ▶ mittels `initializer_list`

```
#include <iostream> // vararginitlist.cpp
using namespace std;
void log(initializer_list<string> messages) {
    for (auto msg : messages) {
        cout << msg << ' ';
    }
}
int main() {
    log({"testing", "warning", "error"});
}

testing warning error
```

Überladen von Funktionen

```
#include <iostream> // functionoverloading.cpp
using namespace std;
void say(char c) {
    cout << c << "! ";
}
void say(const char* str) {
    cout << str << "!! ";
}
void say(string str) {
    cout << str << "!!! ";
}
int main() {
    say('x'); say("World"); say(string{"Bob"});
}
```

x! World!! Bob!!!

Überladen von Funktionen – 2

1. exakte Übereinstimmung der Anzahl, Reihenfolge und Typen
2. Durchführung von Promotions
3. Konvertierungen auf gemeinsame Datentypen
4. benutzerdefinierte Konvertierungen
5. ansonsten: variable Anzahl an Argumenten mittels . . .

Überladen von Funktionen – 3

```
#include <iostream> // functionoverloading2.cpp
using namespace std;
void say(int i) { // promotion
    cout << i << "! ";
}
void say(string str) { // user-defined
    cout << str << "!!! ";
}
int main() {
    say('x'); say("World"); say(string{"Bob"});
}
```

120! World!!! Bob!!!

Überladen von Funktionen – 4

```
#include <iostream> // functionoverloading2.cpp
using namespace std;
void say(long long ll) { // conversion
    cout << ll << "!! ";
}
void say(string str) {
    cout << str << "!!! ";
}
int main() {
    say(112); say("World"); say(string{"Bob"});
}
```

112!! World!!! Bob!!!

Überladen von Funktionen – 5

Funktionen aus verschiedenen Scopes werden **nicht** für das Überladen in Betracht gezogen (wichtig bei Klassen,...)!

```
#include <iostream> // functionoverloading3.cpp
using namespace std;
int incr(int counter) {
    cout << "int" << ' '; return counter + 1;
}
int incr(double counter) {
    cout << "double" << ' '; return counter + 1;
}
int main() {
    int incr(double counter); // declaration!
    incr(1); ::incr(1); }
```

double int

Funktionszeiger

```
#include <iostream> // func_ptr.cpp
using namespace std;
int add(int a, int b) {
    return a + b;
}
int mul(int a, int b) {
    return a * b;
}
int main() {
    int (*f)(int, int);
    f = add; cout << f(3, 2) << ' ';
    f = mul; cout << f(3, 2) << endl;
}
```

5 6

Funktionszeiger – 2

```
#include <iostream> // func_ptr2.cpp
using namespace std;
int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }
using func = int (*)(int, int);
using int_list = initializer_list<int>;
int accumulate(int_list list, func f, int init=0) {
    int res{init};
    for (auto elem : list) { res = f(res, elem); }
    return res;
}
int main() {
    cout << accumulate({1, 2, 3, 4}, add) << ' ';
    cout << accumulate({1, 2, 3, 4}, mul, 1); }
```

Funktionsobjekte

```
#include <iostream> // func_obj.cpp
#include <functional>
using namespace std;
int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }
using func = function<int(int, int)>;
using int_list = initializer_list<int>;
int accumulate(int_list list, func f, int init=0) {
    int res{init};
    for (auto elem : list) { res = f(res, elem); }
    return res; }
int main() {    function<int(int, int)> f{add};
    cout << accumulate({1,2,3,4}, f) << endl;
}
```

10

Funktionsobjekte – 2

```
#include <iostream> // func_obj2.cpp
#include <functional>
using namespace std;
using namespace std::placeholders; // _1, _2, ...
int sub(int a, int b) { return a - b; }

int main() {
    auto answer = bind(sub, 43, 1);
    cout << answer() << ' ';
    auto decr = bind(sub, _1, 1); // first of decr
    cout << decr(43) << ' ';
    auto subinv = bind(sub, _2, _1);
    cout << subinv(1, 43) << endl;
}
```

42 42 42

Funktionsobjekte – 3

```
#include <iostream> // func_obj3.cpp
#include <functional>
using namespace std;
```

```
struct Adder {
    int operator()(int a, int b) {
        return a + b;
    }
};
```

```
int main() {
    Adder adder;
    cout << adder(41, 1) << endl;
}
```

42

Lambdaausdrücke

```
#include <iostream> // lambdaexpr.cpp
#include <algorithm>
using namespace std;
int main() {
    auto values = {1, 2, 3, 4};
    int sum{};
    for_each(begin(values),
              end(values),
              [&sum](int val){ sum += val; });
    cout << sum << endl;
}
```

10

Lambdadausdrücke – 2

- ▶ Compiler übersetzt Lambdaausdruck in Funktionsobjekt
- ▶ Lambdaausdruck besteht aus
 1. Capture-Liste
 2. Parameterliste
 3. optionaler Spezifizierer `mutable`
 - ▶ lokale Variable, die als Kopie zur Verfügung stehen können verändert werden
 4. optional `noexcept`
 5. optional der Rückgabetyt nach `->`
 6. Rumpf in geschwungenen Klammern

Lambdaausdrücke – 3

Capture-Liste

- ▶ enthält nur = ... alle lokalen Variablen stehen als Kopie zur Verfügung
- ▶ enthält nur & ... alle lokalen Variablen stehen als Referenz zur Verfügung
- ▶ enthält einzelne Namen, z.B. [a, &b, c]
- ▶ beginnt mit =, z.B. [=, &a, &b, &c] ... als Kopie, aber a, b, c als Referenz;
 - ▶ ab C++ 17: *this Instanzvariablen des umschließenden Objektes als Kopie
- ▶ beginnt mit &, z.B. [&, a, b, c] ... als Referenz, aber a, b, c als Kopie; this ebenfalls, wenn in einem Objekt

Lambdaausdrücke – 4

```
#include <iostream> // lambdaexpr17.cpp
#include <algorithm>
using namespace std;

struct X {
    int i{42};
    int f() {
        // since C++17
        // *this ... captures enclosing X by copy
        return [=, *this]{ return i; }();
    }
};

int main() {
    X x;
    cout << x.f() << endl; // -> 42
}
```

42

Generische Lambdafunktionen

ab C++14 (in C++11 nur mit konkreten Typen)

```
#include <iostream> // lambdageric.cpp
using namespace std;
int main() {
    auto f = [](auto x){ return x; };
    cout << f(1) << ' ';
    cout << f("abc") << ' ';
    cout << f(false) << endl;
}
```

1 abc 0

Capture mit Initialisierung

ab C++14

```
#include <iostream> // lambdacapturemove.cpp
#include <memory>
using namespace std;
int main() {
    // doesn't have to exist; type will be inferred
    cout << [x=1]{ return x; }() << ' ';
    // can now be an rvalue!
    std::unique_ptr<double> pi(new double{3.14});
    cout << [x=move(pi)](){ return *x; }() << ' ';
    int y{1}; auto h = [y=0]{ return y; };
    cout << h() << ' ' << y << endl;
}
```

1 3.14 0 1

inline Funktionen

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

```
inline double square(double x) {  
    return x * x;  
}
```

```
int main() {  
    cout << "2^2 = " << square(2) << endl;  
}
```

2^2 = 4

[[deprecated]]

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

```
[[deprecated]] int f() { return 1; }
```

```
[[deprecated("g() is legacy; use h() instead")]]  
int g() { return 2; }
```

```
int h() { return 3; }
```

```
int main() {  
    cout << g() << endl; // warning!  
}
```

2