

8. Lehrstuhlseminar

Thema: C++11

Gliederung

Einleitung

Delegierte Konstruktoren

Auto, range-based for

std::function, std::bind

λ 's

Initializer lists

enum class

move

std::tuple

std::to_string

Referenzen

Einleitung

- ▶ C++11 enthält zahlreiche Neuerungen
- ▶ Wir stellen hier eine Teilmenge davon vor
- ▶ Weiterführende Quellen: Referenzen
- ▶ Achtung: Nicht alle Erweiterungen von C++11 unterstützen die Compiler bereits (z.B. GCC kein regex)
- ▶ Aber: Alle hier vorgestellten Erweiterungen werden (spätestens) ab GCC \geq Version 4.7 unterstützt

Delegierte Konstruktoren

```
class A {  
    public:  
    A(int x) { cout << x; }  
    A() : A(100) {}  
};  
  
int main() {  
    A a; // 100  
}
```

Auto, range-based for

```
void print_v1(const vector<string>& v) {  
    for (vector<string>::const_iterator it = v.begin(),  
         end = v.end(); it != end; ++it) cout *it;  
}
```

```
void print_v2(const vector<string>& v) {  
    for (auto it = v.begin(), end = v.end();  
         it != end(); ++it) cout *it;  
}
```

```
void print_v3(const vector<string>& v) {  
    for(const auto& s : v) cout << s;  
}
```

std::function, std::bind

```
int multiply(int x, int y) { return x * y; }

int main() {
    function<int (int, int)> f = multiply;
    cout << f(3, 5); // 15
    function<int (int)> f2 =
        std::bind(f, placeholders::_1, 10);
    cout << f2(3); // 30
}
```

std::bind

Ebenfalls möglich: Binden von Memberfunktionen

```
struct A {  
    void say(const string& something) { cout << something; }  
};  
  
int main() {  
    std::function<void (A&)> sayAlwaysHello =  
        std::bind(&A::say, placeholders::_1, "hello");  
    A a;  
    sayAlwaysHello(a); // "hello"  
}
```

λ 's

Syntax: [capture clause] (parameters) \rightarrow return-type {body}

Bsp:

```
int i = 1;
function<void (int)> f = [&i] (int y) { i *= y; };
f(42);
cout << i; // 42
function<int (int)> g = (int n) { n < 2 ? 1 : n*g(n-1); };
auto h = [=] () { ++i; }; // Fehler
```

Typ eines Lambda-Ausdrucks nur Compiler bekannt, "function"
kann als wrapper benutzt werden.

Initializer lists

```
struct A {  
    A(initializer_list<int> l) { for (int i : l) cout << i; }  
    A(int x) { cout << "-" << x; }  
};
```

```
int main() {  
    A a = {1, 2, 3}; // 123  
    A a2{1}; // 1  
    for (int i : {1, 2, 3, 4, 5}) cout << i; // 12345  
}
```

- ▶ initializer list constructor hat höhere Präzedenz als andere Konstruktoren
- ▶ Existiert für `std::vector`, `std::map`, ...

enum class

Jeder enum Wert hängt direkt mit seiner Klasse zusammen.

```
enum class Color = {red, green, blue};
enum class Alarm = {red, green}; // Kein Fehler

int main() {
    int x = red; // Fehler
    int y = Color::red // Fehler
    Color c = Color::red; // Kein Fehler
}
```

move

```
class A {
public:
    A() { cout << "std constructor"; }
    A(const A& rhs) { cout << "copy constructor"; }
    A(A&& rhs) { cout << "move constructor"; }
    A& operator=(const A& rhs) {
        cout << "copy = op"; return *this; }
    A& operator=(A&& rhs) {
        cout << "move = op"; return *this; }
    static A createA() { return A(); }
};

int main() {
    A a; // "std constructor"
    a = A::createA(); // "std constructor" "move = op"
    A a2(std::move(a)); // "move constructor"
}
```

move

```
class A {
public:
    ~A() { cout << "destructor"; delete _data; }
    A() { _data = new vector<int>(10, 42); }
    A& operator=(A&& rhs) {
        delete _data; // Release old resource.
        _data = rhs._data; // Get new resource.
        rhs._data = nullptr; // Remove resource from rhs.
        return *this;
    }
    static A createA() { return A(); }
    vector<int>* _data;
};

int main() {
    A a; // std constructor
    a = A::createA(); // move = operator and destructor
}
```

std::tuple

```
int x = 5;
tuple<int&, int, float> t(x, 2, 3.141);
cout << get<0>(t); // 5
```

```
int y;
tie(ignore, y, ignore) = t; // unpack tuple into variables
cout << y; // 2
```

```
int x1 = 1, x2 = 2, x3 = 3, x4 = 4;
cout << ((tie(x1, x2, x3) < tie(x1, x2, x3))
  ? "T" : "F"); // "F"
cout << ((tie(x1, x2, x3) < tie(x1, x2, x4))
  ? "T" : "F"); // "T"
```

tie() kann fuer lexikographischen Vergleich verwendet werden (z.B. für Comparator)

std::to_string

Simpel, aber nützlich:

```
int main() {  
    string s1 = to_string(123);  
    string s2 = to_string(123.4);  
    cout << s1 << endl; // 123  
    cout << s2 << endl; // 123.4  
}
```

Referenzen

- ▶ <http://isocpp.org/blog/2012/12/c11-a-cheat-sheet-alex-sinyakov>
- ▶ <http://www.nosid.org/cxx11-about-move-semantics.html>
- ▶ <http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html>