C++ für alle

Praktische Neuerungen in C++11

18. September 2012

Änderungen kriegen Klassen. Eine Änderung mit X ist Y:

S syntaktisch

T im Typsystem

L in der Library

for-Schleifen

Alt und bekannt aus C++98/03:

```
std::vector<double> v = // ...;

std::vector<double>::iterator begin, end;
begin = v.begin();
end = v.end();
for (; begin != end; ++begin) {
    double& d = *begin;

    d = std::log(d) / std::log(4.5);
}
```

S for-Schleifen

Range for loops bieten schönere Syntax für Container. Sieht aus wie Java.

```
for (Typ item : container) {
    // ...
}
```

tut (fast) das selbe wie

```
Iter begin = container.begin();
Iter end = container.end();
for (; begin != end; ++begin) {
    Typ item = *begin;
    // ...
}
```

for-Schleifen

Alt und bekannt aus C++98/03:

```
std::vector<double> v = // ...;

std::vector<double>::iterator begin, end;
begin = v.begin();
end = v.end();
for (; begin != end; ++begin) {
    double& d = *begin;

    d = std::log(d) / std::log(4.5);
}
```

S for-Schleifen

Neu und schöner.

```
std::vector<double> v = // ...;

for (double& d : v) {
    d = std::log(d) / std::log(4.5);
}
```

for-Schleifen

```
for (Typ item : container) {
    // ...
}
```

tut (sehr fast) das selbe wie

```
Iter begin = __begin(container);
Iter end = __end(container);
for (; begin != end; ++begin) {
    Typ item = *begin;
    // ...
}
```

`__begin(x)`: `x.begin()`, oder `begin(x)`, oder `(container)`
`__end(x)`: `x.end()`, oder `end(x)`, oder `(container+size)`

S auto-Typen

Bei Variablendeklaration ist **auto** ab jetzt ein Wildcard.

```
std::vector<int>::iterator i = v.begin();
// oder:
auto i = v.begin();
```

auto wird einfach durch fehlende Typnamen ersetzt.

S auto-Typen

Bei Variablendeklaration ist **auto** ab jetzt ein Wildcard.

```
std::vector<int>::iterator i = v.begin();
// oder:
auto i = v.begin();
```

auto wird einfach durch fehlende Typnamen ersetzt.
const auto* ist immer ein konstanter Pointer.

ST Lambdas

Allgemeine Form eines Lambda-Ausdrucks:

```
[ <captures> ] ( <parameter> ) { <beliebiger code> }
```

ST Lambdas

Allgemeine Form eines Lambda-Ausdrucks:

[<captures>] (<**parameter**>) { <**beliebiger code**> }

So wie von normalen Funktionen gewohnt.

ST Lambdas

Allgemeine Form eines Lambda-Ausdrucks:

```
[ <captures> ] ( <parameter> ) { <beliebiger code> }
```

Closures werden *explizit* erzeugt.

- = captured alles by-value
- & captured alles by-reference
- x captured x by-value
- &x captured x by-reference

ST Lambdas

Ein Lambda-Ausdruck wird zu einem Objekt ausgewertet, das man herumreichen kann.

```
std::vector<double> v = // ...;

std::for_each(v.begin(), v.end(),
    [] (double& d) {
        d = std::log(d) / std::log(4.5);
    });
```

ST RValue references

LValue reference T& bindet an einen Namen.

ST RValue references

LValue reference T& bindet an einen Namen.
Neu: RValue reference T&& bindet an einen Wert. Und ist dabei immer noch eine Referenz. Auf variable Daten!

RValue references

Das gibt uns den move constructor.

```
struct M {
    char* riesenLangerCString;

    M(M&& old)
    {
        riesenLangerCString = old.riesenLangerCString;
        old.riesenLangerCString = NULL;
    }
};
```

ST RValue references

Das gibt uns den move constructor. Und std::move.

```
double max(std::vector<double> values) { ... }

std::vector<double> v = ...; // riesig langes ding

auto d1 = max(v); // kopiert v
auto d2 = max(std::move(v)); // zerstört v

if (d1 != d2) throw "cpu on fire";
```

S for-Schleifen

```
for (Typ item : container) {
    // ...
}
```

tut das selbe wie

```
{   auto&& __c = container;
    auto __b = __begin(__c);
    auto __e = __end(__c);
    for (; __b != __e; ++__b) {
        Typ item = *__b;
        // ...
    }
}
```

`__begin(x)`: `x.begin()`, oder `begin(x)`, oder `(container)`
`__end(x)`: `x.end()`, oder `end(x)`, oder `(container+size)`

S Konstruktoren

Konstruktoren können sich gegenseitig aufrufen.

```
struct X {
    X(int i) { ... }

    X(double d) : X(42) { ... }
};
```

`X(4.2)` ruft zuerst den Konstruktor `X(42)` auf, dann den Rest.

Konstruktion

```
struct X {
};

struct Y {
    Y(const X& x) { ... }
};

Y y(X()); // Funktionsprototyp mit Funktionsargument!
Y z{X()}; // konstruiert Y mit neuem X–Objekt
```

ST Und viele kleine Dinge

- Echte Nullpointer-Konstante: `nullptr`

ST Und viele kleine Dinge

- Echte Nullpointer-Konstante: nullptr
- Enumerationen, die keine Integer sind: **enum class**

ST Und viele kleine Dinge

- Echte Nullpointer-Konstante: nullptr
- Enumerationen, die keine Integer sind: **enum class**
- >> ist nicht mehr immer ein Rechtsshift

ST Und viele kleine Dinge

- Echte Nullpointer-Konstante: `nullptr`
- Enumerationen, die keine Integer sind: **enum class**
- `>>` ist nicht mehr immer ein Rechtsshift
- Unicode-Strings mit Präfixen `u8` (UTF-8), `u` (UTF-16), `U` (UTF-32)

ST Und viele kleine Dinge

- Echte Nullpointer-Konstante: `nullptr`
- Enumerationen, die keine Integer sind: **`enum class`**
- `>>` ist nicht mehr immer ein Rechtsshift
- Unicode-Strings mit Präfixen `u8` (UTF-8), `u` (UTF-16), `U` (UTF-32)
- ... und ganz viel mehr.

L ... wohin man auch sieht

Die STL unterstützt alle neuen Sprachfeatures (natürlich) vollständig.

L ... wohin man auch sieht

Die STL unterstützt alle neuen Sprachfeatures (natürlich) vollständig.
Quasi alles andere gibt's auch schon lange in Boost.

Essen bestellen!

L Smart pointers

std::auto_ptr<T> war nie gut.

std::{unique,shared,weak}_ptr<T> sind besser.

L Smart pointers

std::auto_ptr<T> war nie gut.
std::{unique,shared,weak}_ptr<T> sind besser.

- unique_ptr enthält allein einen Pointer. Kopieren verboten.

L Smart pointers

std::auto_ptr<T> war nie gut.
std::{unique,shared,weak}_ptr<T> sind besser.

- unique_ptr enthält allein einen Pointer. Kopieren verboten.
- shared_ptr enthält geteilt mit anderen einen Pointer.

L Smart pointers

std::auto_ptr<T> war nie gut.
std::{unique,shared,weak}_ptr<T> sind besser.

- unique_ptr enthält allein einen Pointer. Kopieren verboten.
- shared_ptr enthält geteilt mit anderen einen Pointer.
- weak_ptr enthält vielleicht einen Pointer. Nützlich zum Kreise brechen.

L Smart pointers

std::auto_ptr<T> war nie gut.
std::{unique,shared,weak}_ptr<T> sind besser.

- unique_ptr enthält allein einen Pointer. Kopieren verboten.
- shared_ptr enthält geteilt mit anderen einen Pointer.
- weak_ptr enthält vielleicht einen Pointer. Nützlich zum Kreise brechen.

Einfach benutzen und nie wieder selbst ein **delete** schreiben.

L Assoziative Container

C++98/03: `{,multi}{set,map}` mit Bäumen.
C++11: `unordered_{,multi}{set,map}` mit Hashtables. Yay!
Funktioniert direkt für STL-Typen.

L Assoziative Container

C++98/03: `{,multi}{set,map}` mit Bäumen.
C++11: `unordered_{,multi}{set,map}` mit Hashtables. Yay!
Funktioniert direkt für STL-Typen.
Alles andere spezialisiert einfach `std::hash<T>`.

L Reguläre Ausdrücke

Alles in `<regex>` enthalten.

```
// simple IPv6
std::regex regex("[0-9a-f]{1,4}(:[0-9a-f]{1,4}){7}");

if (!std::regex_match(addr_from_config, regex))
    throw "config file puking";
```

Zusätzlich `std::regex_search`, `std::regex_replace` und Infrastruktur.

L Reguläre Ausdrücke

Alles in `<regex>` enthalten.

```
// simple IPv6
std::regex regex("[0-9a-f]{1,4}(:[0-9a-f]{1,4}){7}");

if (!std::regex_match(addr_from_config, regex))
    throw "config file puking";
```

Zusätzlich `std::regex_search`, `std::regex_replace` und Infrastruktur.
... Theoretisch. libstdc++ kann das noch nicht, boost aber schon lange.

STL Tupel

Tupel enthalten beliebig viele Elemente.

```
std::tuple<int, int, int> tripel(1, 2, 3);

std::get<0>(tripel) == 1 && std::get<2>(tripel) == 3
```

STL Tupel

Tupel enthalten beliebig viele Elemente.

```
std::tuple<int, int, int> tripel(1, 2, 3);

std::get<0>(tripel) == 1 && std::get<2>(tripel) == 3
```

Zwei Tupel sind kompatibel, wenn sie gleich lang und elementweise kompatibel sind.

```
// wandelt alle ints in doubles um
std::tuple<double, double, double> dt = tripel;
```

STL Tupel

```
std::tuple<int, char, double> foo()
{
    return std::make_tuple(42, 't', 3.1416);
}

// ...

int sinn;
double fastPi;

std::tie(sinn, std::ignore, fastPi) = foo();
// sinn == 42, fastPi == 3.1416
```

L Viel, viel mehr

Vieles davon gibt es schon lange in Boost.

L Viel, viel mehr

Vieles davon gibt es schon lange in Boost.

- Threading

L Viel, viel mehr

Vieles davon gibt es schon lange in Boost.

- Threading
- Generalisierte Funktionspointer

L Viel, viel mehr

Vieles davon gibt es schon lange in Boost.

- Threading
- Generalisierte Funktionspointer
- Viele Zufallsgeneratoren

L Viel, viel mehr

Vieles davon gibt es schon lange in Boost.

- Threading
- Generalisierte Funktionspointer
- Viele Zufallsgeneratoren
- ... Und noch mehr.

EOT

Fragen?

L Threads

Starte einen Thread und warte auf dessen Ende:

```
std::thread thread(zeuch_mit_netzwerk, sock_fd);
// ...
thread.join();
```

Mutexe in der STL:

```
std::mutex m;

m.lock();
// ...
m.unlock();
```

L Threads

Das selbe in "geht nicht kaputt":

```
std::mutex m;

{
    std::lock_guard<std::mutex> lock(m);
    // ...
}
```

`m` wird beim verlassen des Scopes freigegeben. Auch bei Exceptions - vorhin wär das anders.

L Funktionsobjekte

`std::function<...>` bindet an alles aufrufbare.

```
struct X {
    int foo() { return 42; }
};

int bar(X*) { return 23; }

std::function<int (X*)> fn;
fn = &X::foo;
fn = bar;
fn = [] (X*) { return 17 };
```