

Warum C++11?

Nur weil GCC6 jetzt `-std=gnu++14` default macht?!

Andreas Neiser

8. Februar 2017

```
!ErrorHasOccured() ??!?! HandleError();
```

AAA – Almost always **auto** – Fast immer **auto**

Statt Typ hinzuschreiben, einfach **auto**.

Der Compiler findet's schon raus, hoffentlich...

```
1 typedef vector<int> nums_t;  
2 static const int t[] = {1,2,3};  
3 nums_t a(t,t+sizeof(t)/sizeof(t[0]));  
4 for(nums_t::iterator i=a.begin(); i!=a.end(); ++i)  
5     *i *= 2;
```

Wo weiß der Compiler eigentlich eh schon,
was für ein Typ die Variable hat?

AAA – Almost always **auto** – Fast immer **auto**

Statt Typ hinschreiben, einfach **auto**.

Der Compiler findet's schon raus, hoffentlich...

```
1 typedef vector<int> nums_t;  
2 static const int t[] = {1,2,3};  
3 nums_t a(t,t+sizeof(t)/sizeof(t[0]));  
4 for(nums_t::iterator i=a.begin(); i!=a.end(); ++i)  
5     *i *= 2;
```

Wo weiß der Compiler eigentlich eh schon,
was für ein Typ die Variable hat?

```
6 static const int t[] = {1,2,3};  
7 vector<int> a(t,t+sizeof(t)/sizeof(t[0]));  
8 for(auto i=a.begin(); i!= a.end(); ++i)  
9     *i *= 2;
```

AAA – Almost always **auto** – Fast immer **auto**

Statt Typ hinschreiben, einfach **auto**.

Der Compiler findet's schon raus, hoffentlich...

```
1 typedef vector<int> nums_t;  
2 static const int t[] = {1,2,3};  
3 nums_t a(t,t+sizeof(t)/sizeof(t[0]));  
4 for(nums_t::iterator i=a.begin(); i!=a.end(); ++i)  
5     *i *= 2;
```

Wo weiß der Compiler eigentlich eh schon,
was für ein Typ die Variable hat?

```
6 static const int t[] = {1,2,3};  
7 vector<int> a(t,t+sizeof(t)/sizeof(t[0]));  
8 for(auto i=a.begin(); i!=a.end(); ++i)  
9     *i *= 2;
```

auto macht Code schlanker und allgemeiner
(und man kann über wichtigere Dinge nachdenken als **typedefs**)

Geschweifte Klammern und `initializer_list`

```
1 vector<int> a{1,2,3}; // oh, wie einfach!  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

Geschweifte Klammern und initializer_list

```
1 vector<int> a{1,2,3}; // oh, wie einfach!  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

Genauso geht:

```
4 struct my_t {  
5     int A; int B;  
6     my_t(int a, int b) : A(a), B(b) {}  
7 };  
8 vector<my_t> a{{1,2},{3,4}};
```

Geschweifte Klammern und initializer_list

```
1 vector<int> a{1,2,3}; // oh, wie einfach!  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

Genauso geht:

```
4 struct my_t {  
5     int A; int B;  
6     my_t(int a, int b) : A(a), B(b) {}  
7 };  
8 vector<my_t> a{{1,2},{3,4}};
```

Aber **Vorsicht** bei:

```
9 vector<int> a(5);  
10 vector<int> b{5};
```

Geschweifte Klammern und initializer_list

```
1 vector<int> a{1,2,3}; // oh, wie einfach!  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

Genauso geht:

```
4 struct my_t {  
5     int A; int B;  
6     my_t(int a, int b) : A(a), B(b) {}  
7 };  
8 vector<my_t> a{{1,2},{3,4}};
```

Aber **Vorsicht** bei:

```
9 vector<int> a(5);  
10 vector<int> b{5};
```

{...} macht das Initialisieren „natürlich hübsch“

for-ranged loops und nochmal **auto**

```
1 vector<int> a{1,2,3};  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

for-ranged loops und nochmal **auto**

```
1 vector<int> a{1,2,3};  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

...kann man auch so schreiben:

```
4 vector<int> a{1,2,3};  
5 for(auto& v : a)  
6     v *= 2;
```

for-ranged loops und nochmal **auto**

```
1 vector<int> a{1,2,3};  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

...kann man auch so schreiben:

```
4 vector<int> a{1,2,3};  
5 for(auto& v : a)  
6     v *= 2;
```

Hm, aber was ist mit:

```
7 const vector<int> a{1,2,3};  
8 for(auto& v : a)  
9     v *= 2;
```

for-ranged loops und nochmal **auto**

```
1 vector<int> a{1,2,3};  
2 for(auto i=a.begin(); i != a.end(); ++i)  
3     *i *= 2;
```

...kann man auch so schreiben:

```
4 vector<int> a{1,2,3};  
5 for(auto& v : a)  
6     v *= 2;
```

Hm, aber was ist mit:

```
7 const vector<int> a{1,2,3};  
8 for(auto& v : a)  
9     v *= 2;
```

for-ranged loops machen STL container endlich benutzbar,
und **auto** ist schlau genug für **const**-correctness (und mehr)

Zero-overhead byte-packing

```
1 auto packed = Make(0,1,1,0,  
2                   1,0,0,1,  
3                   1,0,0,1,  
4                   0,1,1,0); // = {0x69, 0x96}
```

Zero-overhead byte-packing

```
1 auto packed = Make(0,1,1,0,  
2                    1,0,0,1,  
3                    1,0,0,1,  
4                    0,1,1,0); // = {0x69, 0x96}
```

Eine erste Idee, schonmal mit variadic templates:

```
5 using byte_t = unsigned char;  
6 template<typename... Bools>  
7 auto Make(Bools... bools) {  
8     const vector<bool> b{static_cast<bool>(bools)...};  
9     vector<byte_t> a(b.size()/8);  
10    for(auto i=0u; i<a.size(); ++i)  
11        a[a.size()-i-1] = (b[8*i+7]<<7) | (b[8*i+6]<<6)  
12                        | (b[8*i+5]<<5) | (b[8*i+4]<<4)  
13                        | (b[8*i+3]<<3) | (b[8*i+2]<<2)  
14                        | (b[8*i+1]<<1) | b[8*i+0];  
15    return a;  
16 }
```

Zero-overhead byte-packing II

Erstmal:

```
1 template<typename ... Bools>
2 auto Make(Bools... bools) {
3     constexpr auto nBools = sizeof...(bools);
4     static_assert(nBools % 8 == 0, "nBools_wrong");
5     auto a = array<byte_t, nBools/8>();
6     Fill(a, nBools/8, bools...);
7     return a;
8 }
```

Fill(...) wird dem Compiler nun sagen,
wie a zu füllen ist...

Zero-overhead byte-packing III

...und zwar mit rekursiven function calls:

```
1 template<size_t N, typename... Bools>
2 void Fill(array<byte_t, N>& a,
3           size_t i,
4           bool b7, bool b6, bool b5, bool b4,
5           bool b3, bool b2, bool b1, bool b0,
6           Bools... bools) {
7     a[N-i] = (b7<<7) | (b6<<6) /* | ... */ | b0;
8     Fill(a, i-1, bools...);
9 }
10
11 template<size_t N>
12 void Fill(array<byte_t, N>&, size_t) {}
```


Zero-overhead byte-packing III

...und zwar mit rekursiven function calls:

```
1 template<size_t N, typename... Bools>
2 void Fill(array<byte_t, N>& a,
3         size_t i,
4         bool b7, bool b6, bool b5, bool b4,
5         bool b3, bool b2, bool b1, bool b0,
6         Bools... bools) {
7     a[N-i] = (b7<<7) | (b6<<6) /* | ... */ | b0;
8     Fill(a, i-1, bools...);
9 }
10
11 template<size_t N>
12 void Fill(array<byte_t, N>&, size_t) {}
```

Auf zur Livedemo mit <https://godbolt.org/g/57y21s>
mehr bei Jason Turner's CppCon2016 talk:
<https://youtu.be/zBkNBP00wJE>

Nie wieder schreckliches raw pointer management

Warum will man sowas nicht:

```
1 auto p = new int;  
2 DoSomething(p);  
3 delete p;
```

Nie wieder schreckliches raw pointer management

Warum will man sowas nicht:

```
1 auto p = new int;  
2 DoSomething(p);  
3 delete p;
```

Besser, aber noch nicht gut:

```
4 struct owner_t {  
5     int* p;  
6     owner_t(int* p_) : p(p_) {}  
7     ~owner_t() { delete p; }  
8 };  
9  
10 { // some scope  
11     owner_t o(new int);  
12     DoSomething(o.p);  
13 }
```

Nie wieder schreckliches raw pointer management II

Was passiert bei Kopien von `owner_t`:

```
1 { // some scope
2   owner_t o(new int);
3   auto o_copy = o;
4   DoSomething(o.p);
5   // uh uh, double-free *zonk*
6 }
```

Nie wieder schreckliches raw pointer management II

Was passiert bei Kopien von `owner_t`:

```
1 { // some scope
2   owner_t o(new int);
3   auto o_copy = o;
4   DoSomething(o.p);
5   // uh uh, double-free *zonk*
6 }
```

Ownership sollte nicht kopiert werden können!

Nie wieder schreckliches raw pointer management II

Was passiert bei Kopien von `owner_t`:

```
1 { // some scope
2   owner_t o(new int);
3   auto o_copy = o;
4   DoSomething(o.p);
5   // uh uh, double-free *zonk*
6 }
```

Ownership sollte nicht kopiert werden können!

Was ist dann aber mit:

```
7 vector<owner_t> ptrs{new int()}; // oh no, again
```

Nie wieder schreckliches raw pointer management II

Was passiert bei Kopien von `owner_t`:

```
1 { // some scope
2   owner_t o(new int);
3   auto o_copy = o;
4   DoSomething(o.p);
5   // uh uh, double-free *zonk*
6 }
```

Ownership sollte nicht kopiert werden können!

Was ist dann aber mit:

```
7 vector<owner_t> ptrs{new int()}; // oh no, again
```

Man kann STL Container so nicht mehr verwenden!

Ähm, nein...

Move semantics mit `unique_ptr<T>`

Die Rettung ist `unique_ptr<int>` statt `owner_t`

```
1 { // some scope
2   auto o = make_unique<int>(); // factory
3   DoSomethingWithRef(*o);
4   DoSomething(move(o));
5   // o invalid here, as DoSomething took ownership
6 }
```


Move semantics mit `unique_ptr<T>`

Die Rettung ist `unique_ptr<int>` statt `owner_t`

```
1 { // some scope
2   auto o = make_unique<int>(); // factory
3   DoSomethingWithRef(*o);
4   DoSomething(move(o));
5   // o invalid here, as DoSomething took ownership
6 }
```

Aber es ist nicht alles Sonnenschein:

```
7 using p_t = unique_ptr<int>;
8 p_t in[] = {make_unique<int>(1),
9             make_unique<int>(2)};
10 const list<p_t> ptrs(make_move_iterator(begin(in)),
11                    make_move_iterator(end(in)));
12 for(const auto& p : ptrs)
13   *p *= 2; // constness?!
```

Endlich Ende

Was war:

- **auto** geht einfach immer
- { ... } als Initialisierungsliste
- Variadic templates mit zero-overhead dank -O3
- Object ownership management

Was ist:

- Scott Meyers, Effective Modern C++, 2014
- Bjarne Stroustrup, The C++ Programming Language, 2013
- CppCon auf YouTube

Was wird (vielleicht):

- Concepts für **template**
- Structured bindings für tuple

```

1 template<int p, int i = p - 1>
2 struct is_prime {
3     constexpr static bool v = p % i &&
4         is_prime<p, i - 1>::v;
5 };
6
7 template<int p>
8 struct is_prime<p, 1>{constexpr static bool v = 1;};
9
10 template<int p>
11 struct is_prime<p, 0>{constexpr static bool v = 1;};
12
13 template<int max>
14 void prime_print() {
15     cout << max << " _"
16         << (max==1 ? 0 : is_prime<max>::v) << endl;
17     prime_print<max-1>();
18 };
19
20 template<> void prime_print<0>() {};

```