Warum C++11?

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

Andreas Neiser

30. Januar 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 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?

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

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?

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

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

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:

```
1 struct my_t {
2   int A; int B;
3   my_t(int a, int b) : A(a), B(b) {}
4 };
5 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:

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

Aber Vorsicht bei:

```
1 vector<int> a(5);
2 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:

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

Aber Vorsicht bei:

```
1 vector<int> a(5);
2 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:

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

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

Hm, aber was ist mit:

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

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

Hm, aber was ist mit:

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

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

Eine erste Idee, schonmal mit variadic templates:

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

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:

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

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:

```
1 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:

```
1 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:

```
1 using p_t = unique_ptr<int>;
2 p_t in[] = {make_unique<int>(1),
3              make_unique<int>(2)};
4 const list<p_t> ptrs(make_move_iterator(begin(in)),
5                      make_move_iterator(end(in)));
6 for(const auto& p : ptrs)
7   *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`