Warum C++11?

Andreas Neiser

27. Januar 2017

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

Statt Typ hinzuschreiben, einfach **auto** stattdessen.
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** stattdessen.
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** stattdessen.
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 `std::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 `std::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 `std::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 `std::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

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); // should be {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); // should be {0x69, 0x96}
```

Eine Idee:

```
template<typename... Bools>
auto Make(Bools... bools) {
  const vector<bool> b{(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+0];
  return a;
}
```

Zero-overhead byte-packing II

Erstmal:

```
template<typename... Bools>
auto Make(Bools... bools) {
  constexpr auto nBools = sizeof...(bools);
  static_assert(nBools % 8 == 0, "Bools not packable");
  auto a = array<byte_t, nBools/8>();
  Fill(a, nBools/8, bools...);
  return a;
}
```

Zero-overhead byte-packing III

Dann:

```
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) /* | ... */ |  b0;
8    Fill(a, i-1, bools...);
9  }
10
11 template<size_t N>
12 void Fill(array<byte_t, N>&, size_t) {}
```