

# 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 **typedefs**)

## 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