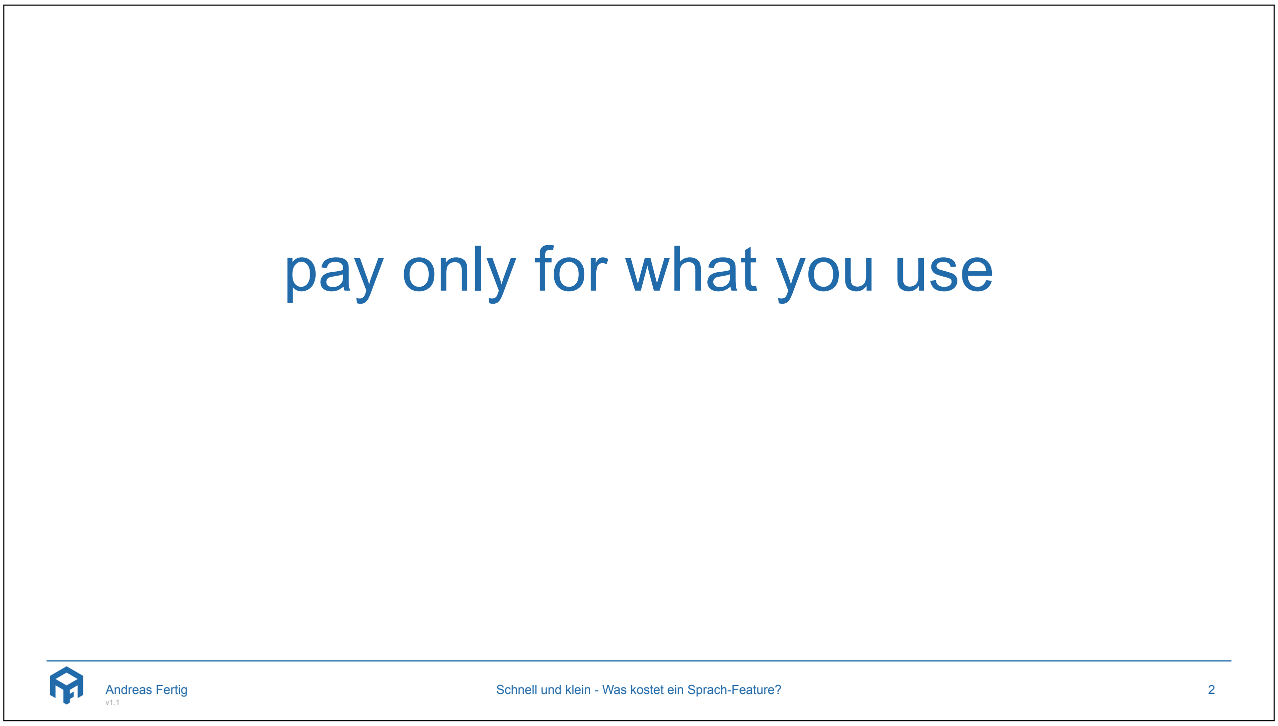
Schnell und klein

Was kostet ein Sprach-Feature?

Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

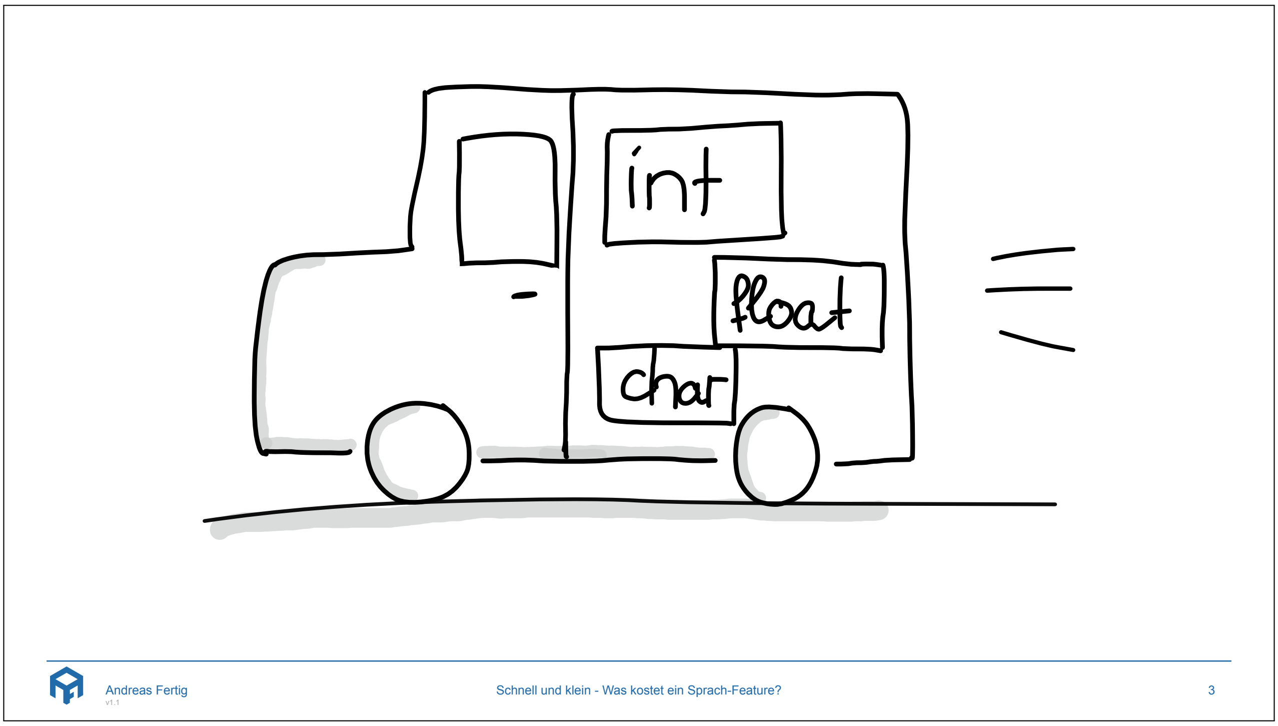


pay only for what you use

Andreas Fertig
v1.1

Schnell und klein - Was kostet ein Sprach-Feature?

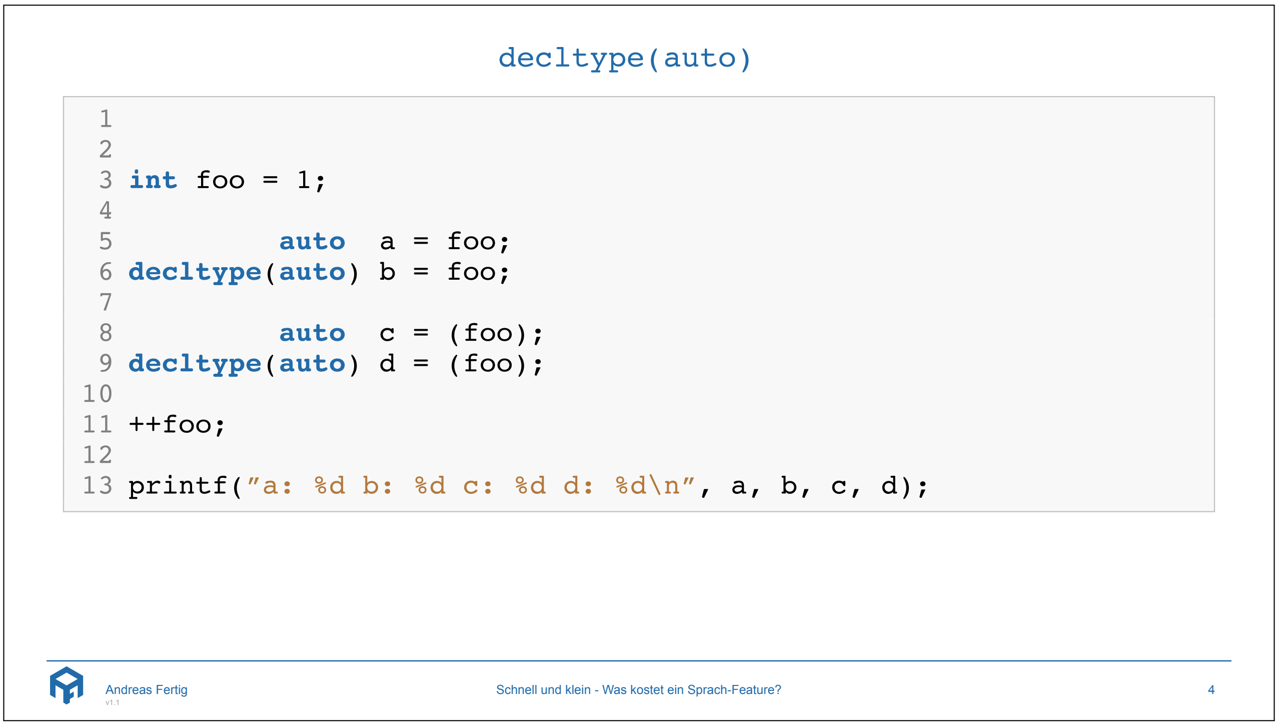
2



Andreas Fertig
v1.1

Schnell und klein - Was kostet ein Sprach-Feature?

3



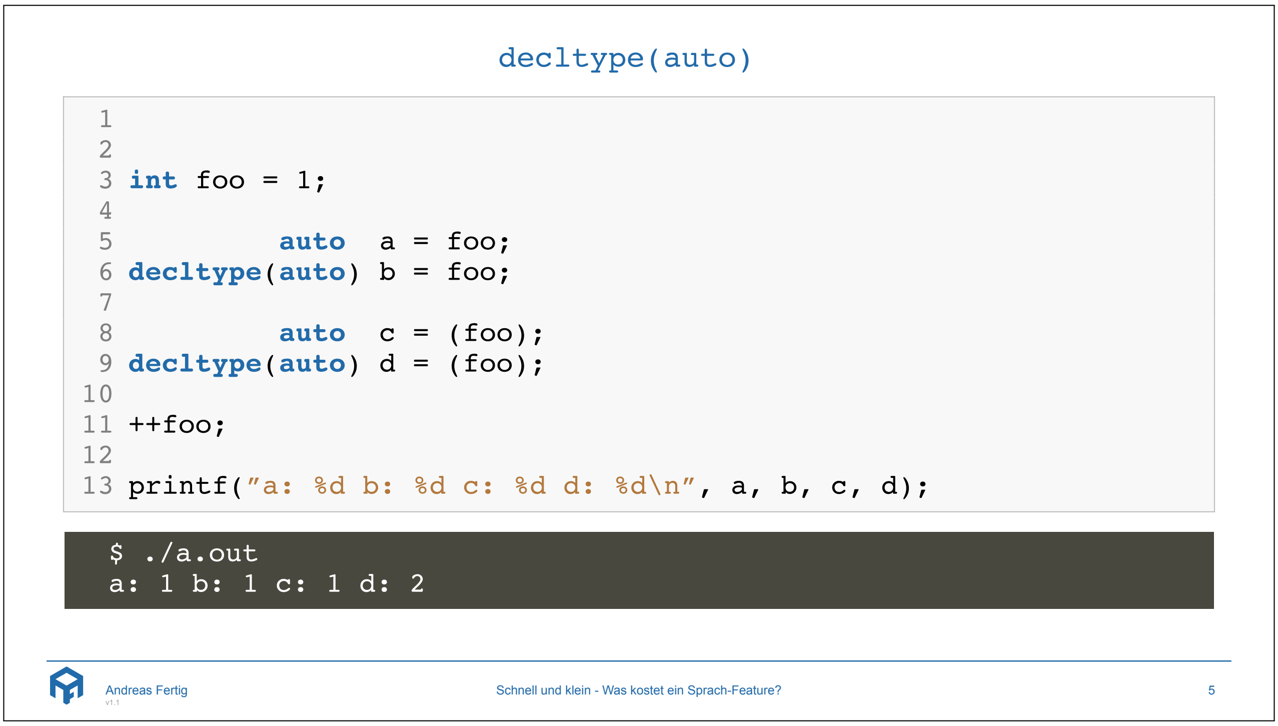
decltype(auto)

```
int foo = 1;

         auto  a = foo;
decltype(auto) b = foo;

         auto  c = (foo);
decltype(auto) d = (foo);

++foo;

printf("a: %d b: %d c: %d d: %d\n", a, b, c, d);
```

Andreas Fertig
v1.1

Schnell und klein - Was kostet ein Sprach-Feature?

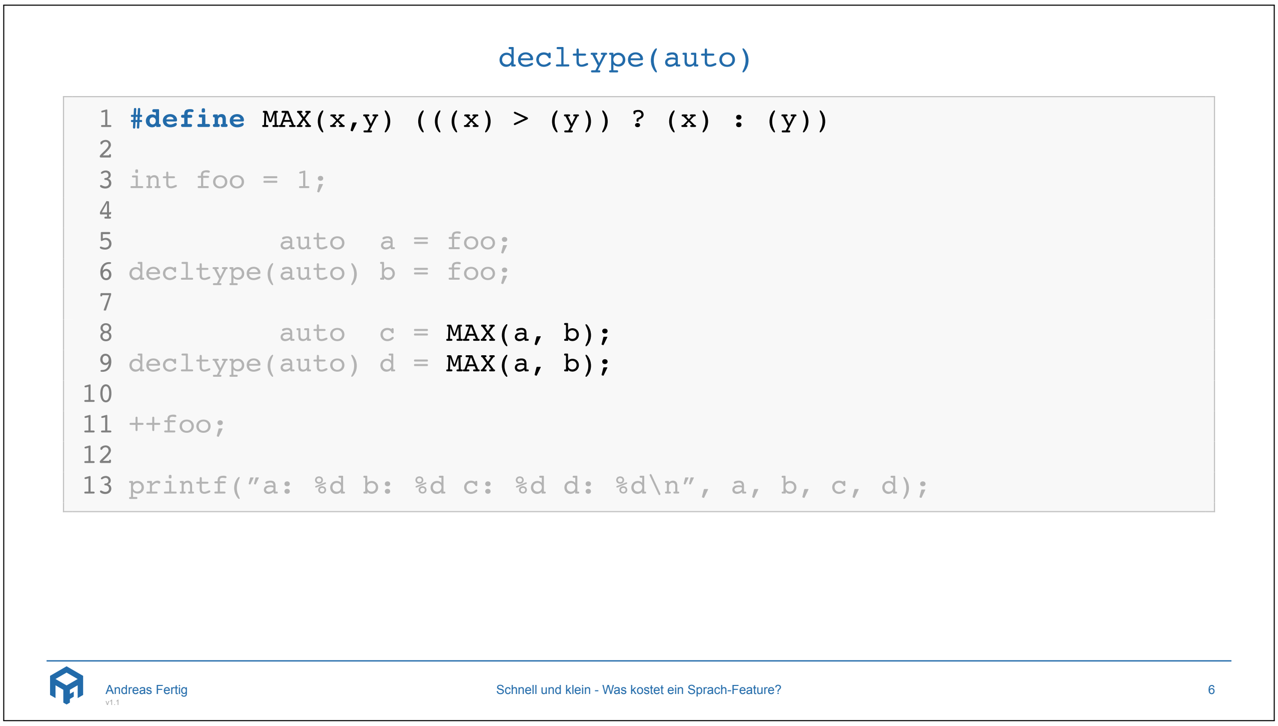
4



decltype(auto)

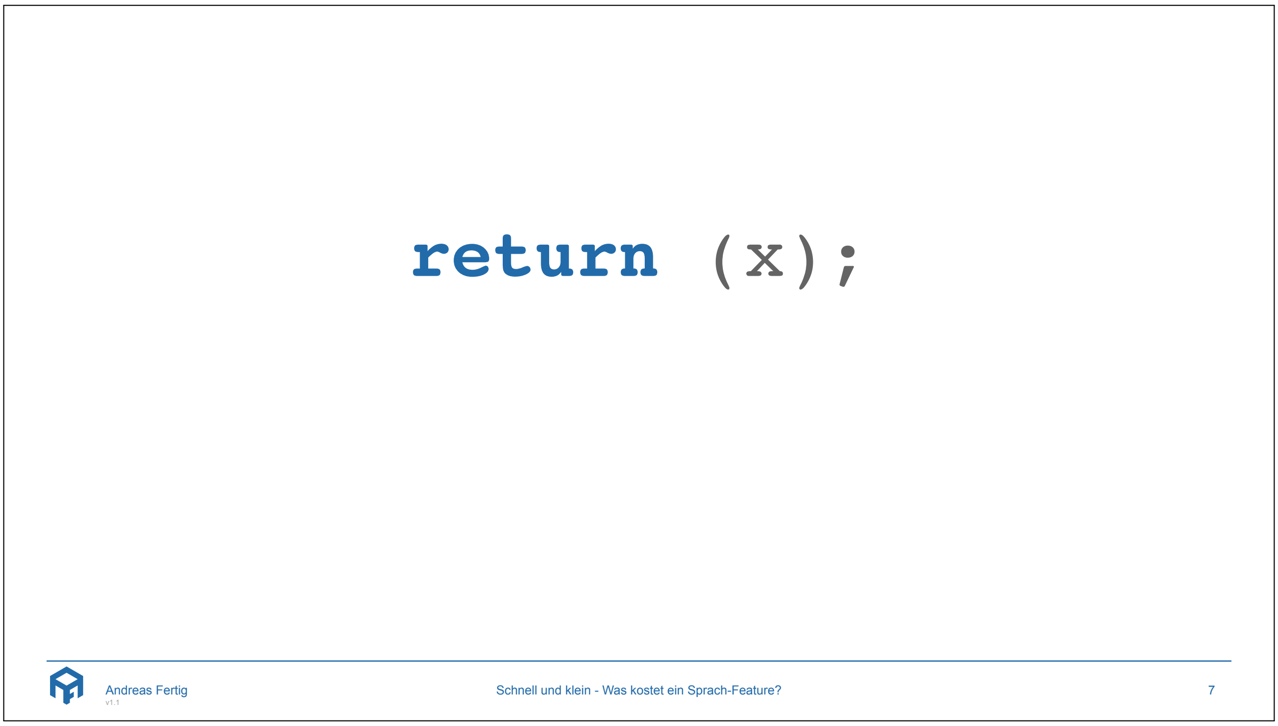
```


int foo = 1;

         auto  a = foo;
decltype(auto) b = foo;

         auto  c = (foo);
decltype(auto) d = (foo);

++foo;

printf(”a: %d b: %d c: %d d: %d\n”, a, b, c, d);
```

```
$ ./a.out
a: 1 b: 1 c: 1 d: 2
```



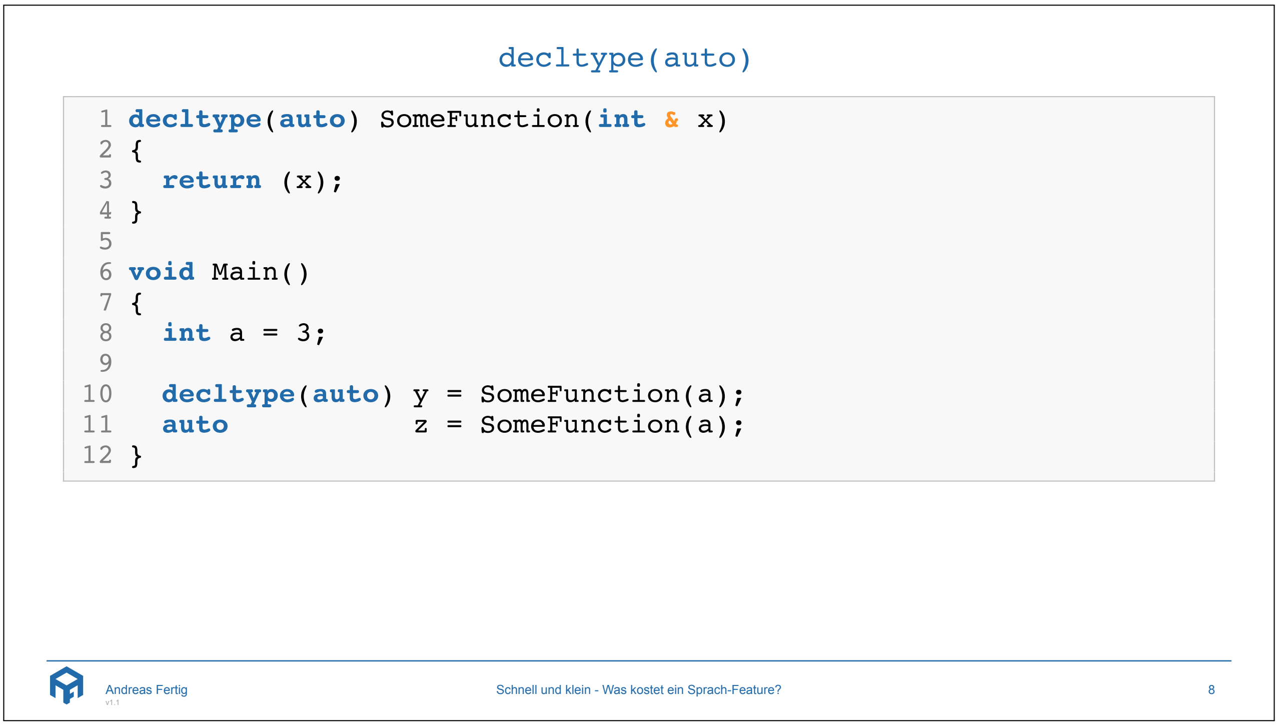
decltype(auto)

```
#define MAX(x,y) (((x) > (y)) ? (x) : (y))

int foo = 1;

         auto  a = foo;
decltype(auto) b = foo;

         auto  c = MAX(a, b);
decltype(auto) d = MAX(a, b);

++foo;

printf(”a: %d b: %d c: %d d: %d\n”, a, b, c, d);
```



```
return (x);
```

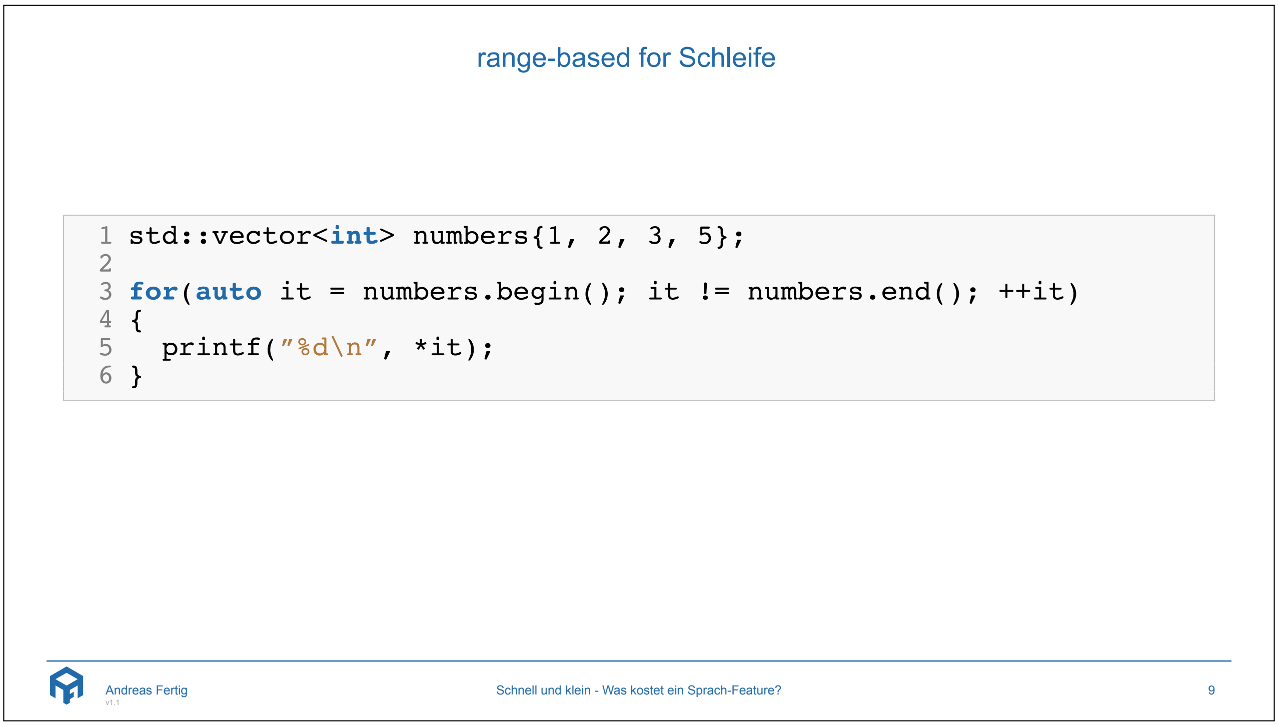
Andreas Fertig v1.1 Schnell und klein - Was kostet ein Sprach-Feature? 7



decltype(auto)

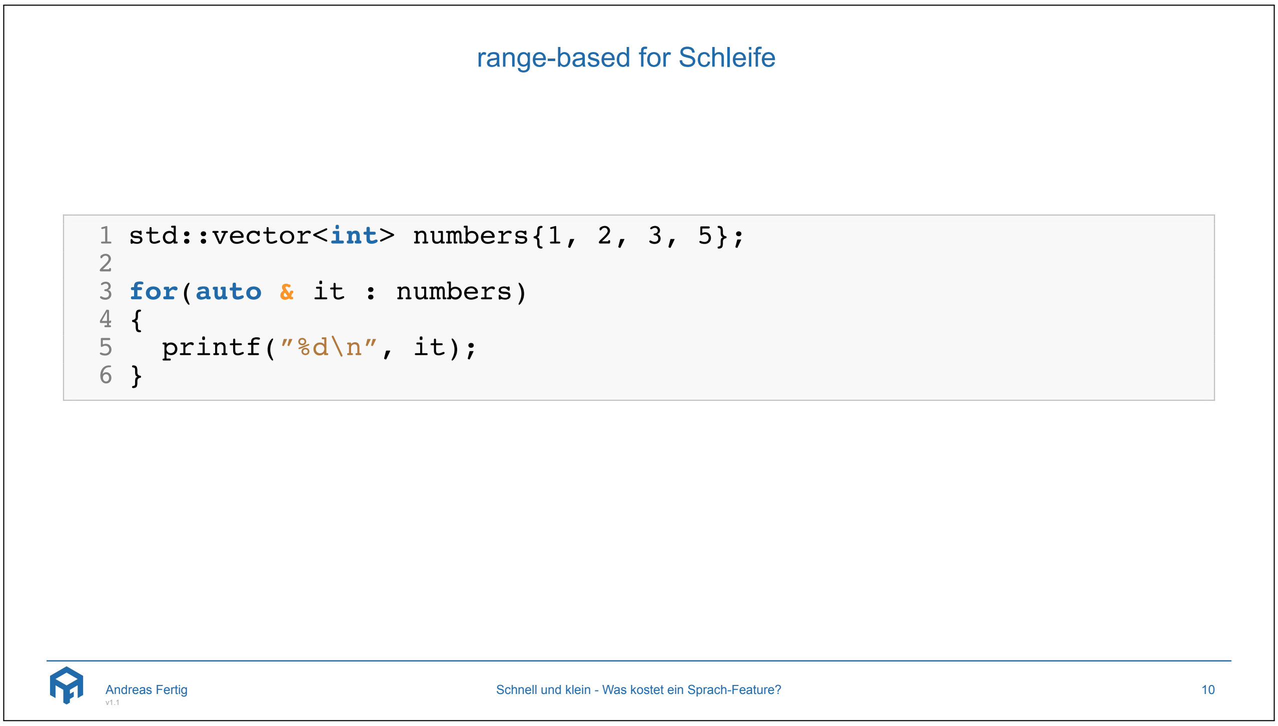
```
decltype(auto) SomeFunction(int & x)
{
  return (x);
}

void Main()
{
  int a = 3;

  decltype(auto) y = SomeFunction(a);
  auto           z = SomeFunction(a);
}
```

Andreas Fertig v1.1 Schnell und klein - Was kostet ein Sprach-Feature? 8



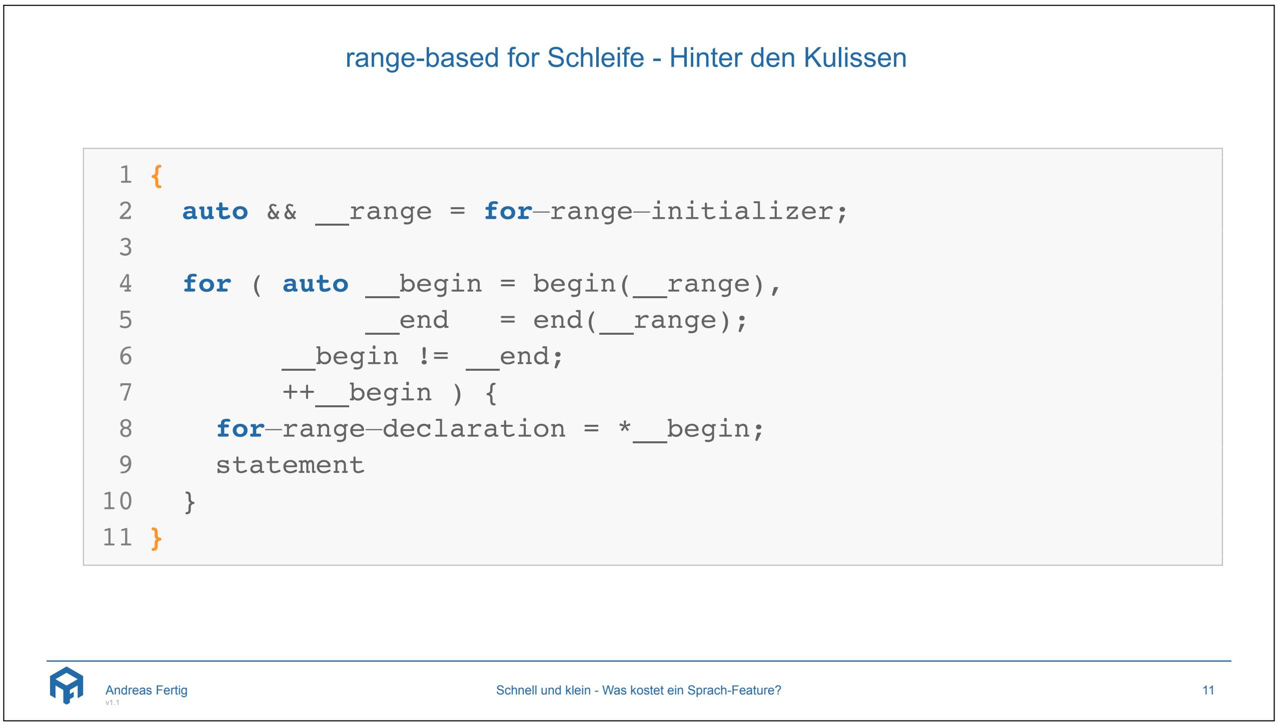
range-based for Schleife

```
std::vector<int> numbers{1, 2, 3, 5};

for(auto it = numbers.begin(); it != numbers.end(); ++it)
{
  printf("%d\n", *it);
}
```



range-based for Schleife

```
std::vector<int> numbers{1, 2, 3, 5};

for(auto & it : numbers)
{
  printf("%d\n", it);
}
```



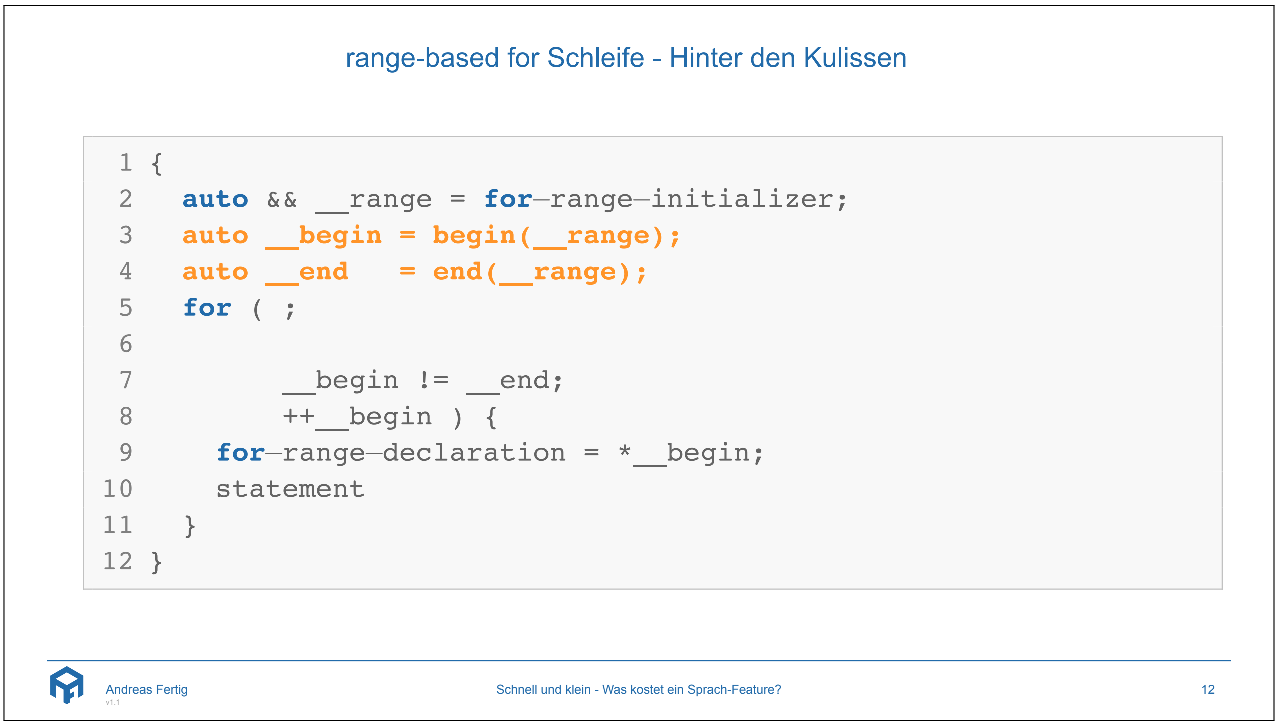
range-based for Schleife - Hinter den Kulissen

```
{
  auto && __range = for-range-initializer;

  for ( auto __begin = begin(__range),
             __end   = end(__range);
        __begin != __end;
        ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

Andreas Fertig
v1.1

Schnell und klein - Was kostet ein Sprach-Feature?

11



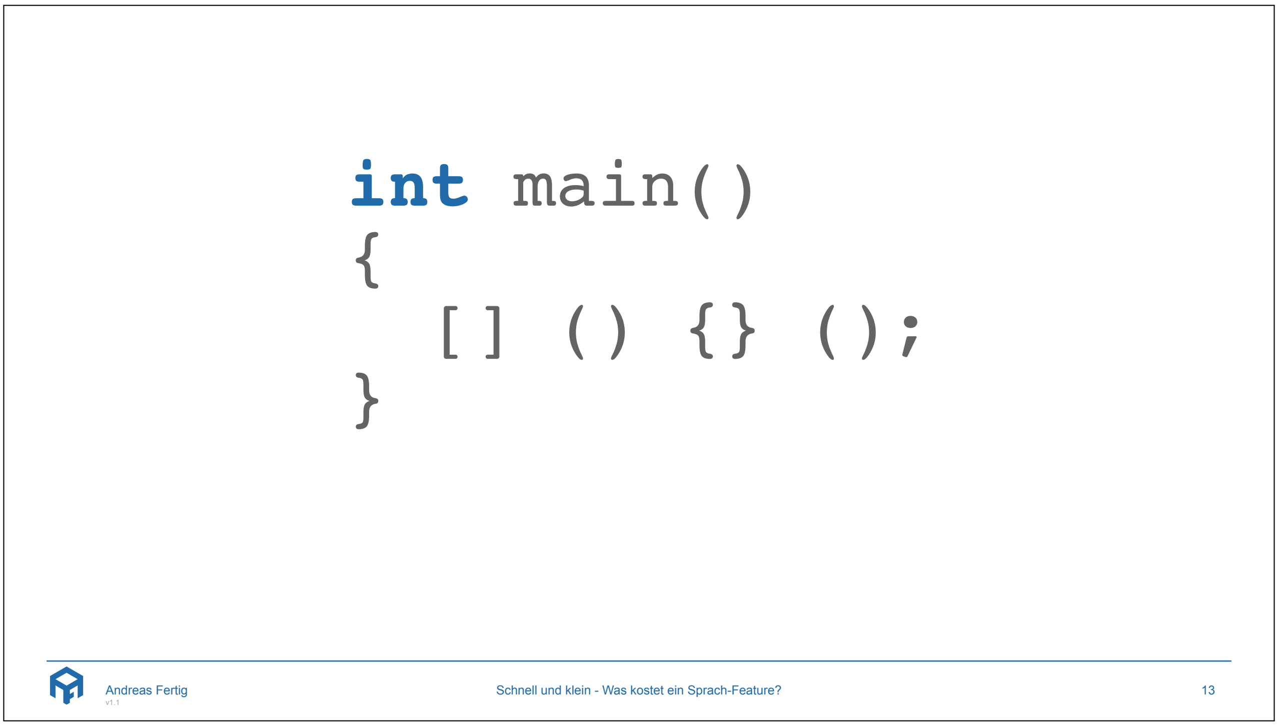
range-based for Schleife - Hinter den Kulissen

```
{
  auto && __range = for-range-initializer;
  auto __begin = begin(__range);
  auto __end   = end(__range);
  for ( ;

        __begin != __end;
        ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

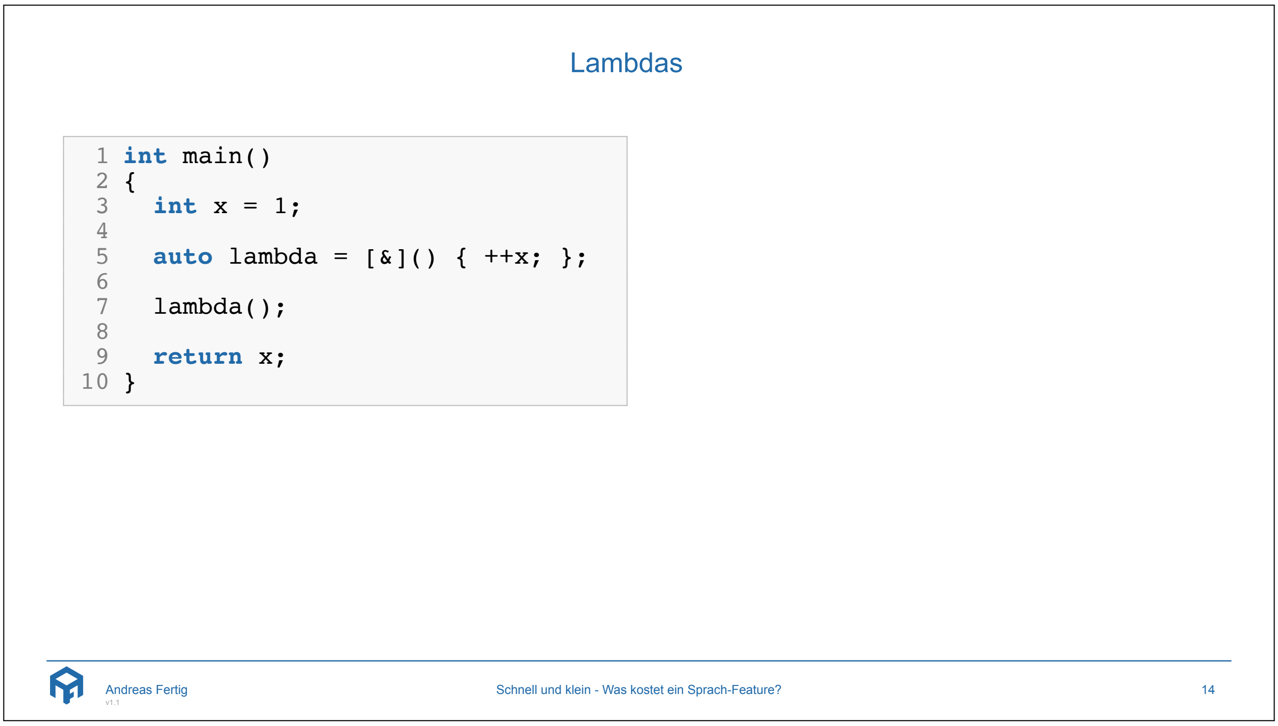
Andreas Fertig
v1.1

Schnell und klein - Was kostet ein Sprach-Feature?

12

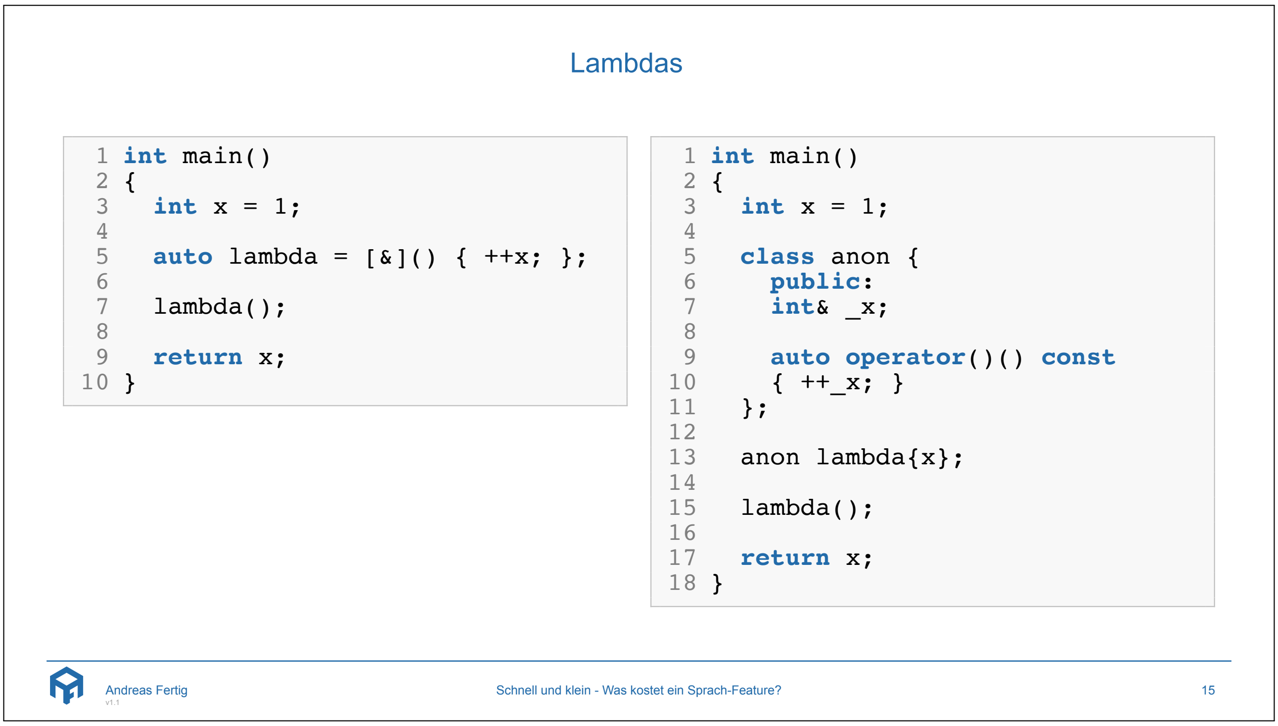


```
int main()
{
   [] () {} ();
}
```



Lambdas

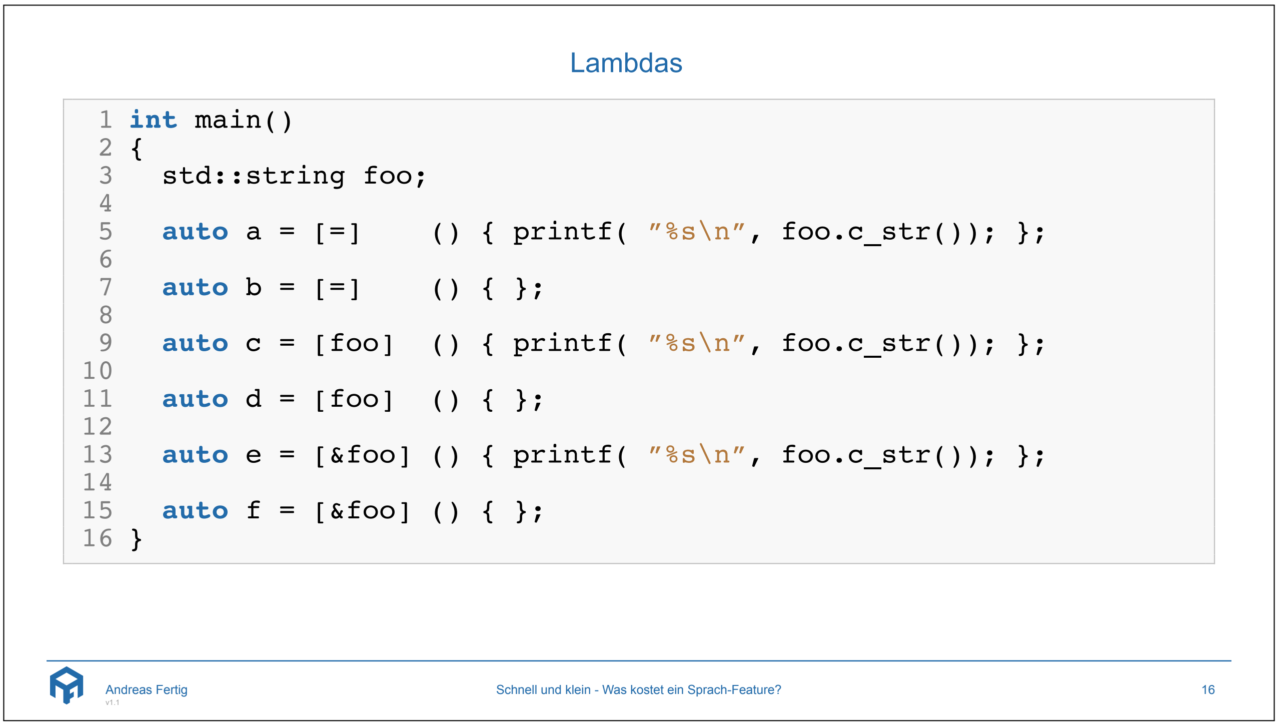
```
int main()
{
  int x = 1;

  auto lambda = [&]() { ++x; };

  lambda();

  return x;
}
```



Lambdas

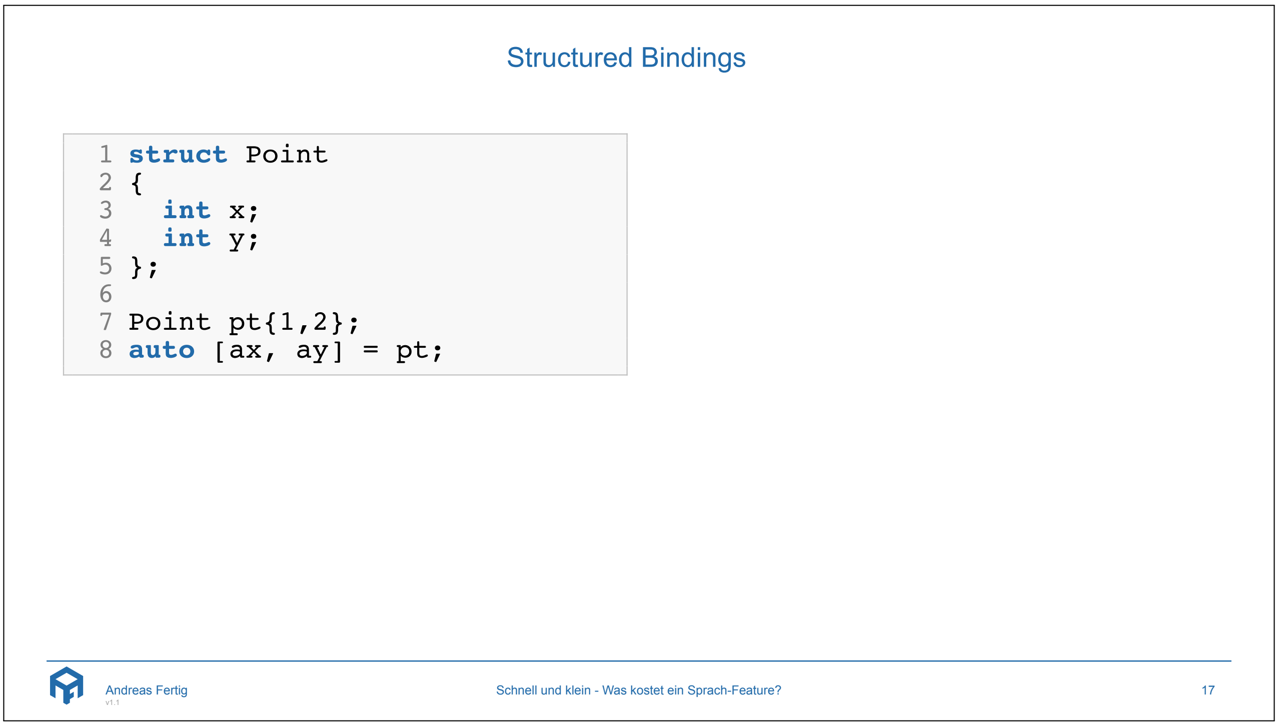
```cpp
int main()
{
  int x = 1;

  auto lambda = [&]() { ++x; };

  lambda();

  return x;
}
```

```cpp
int main()
{
  int x = 1;

  class anon {
    public:
    int& _x;

    auto operator()() const
    { ++_x; }
  };

  anon lambda{x};

  lambda();

  return x;
}
```



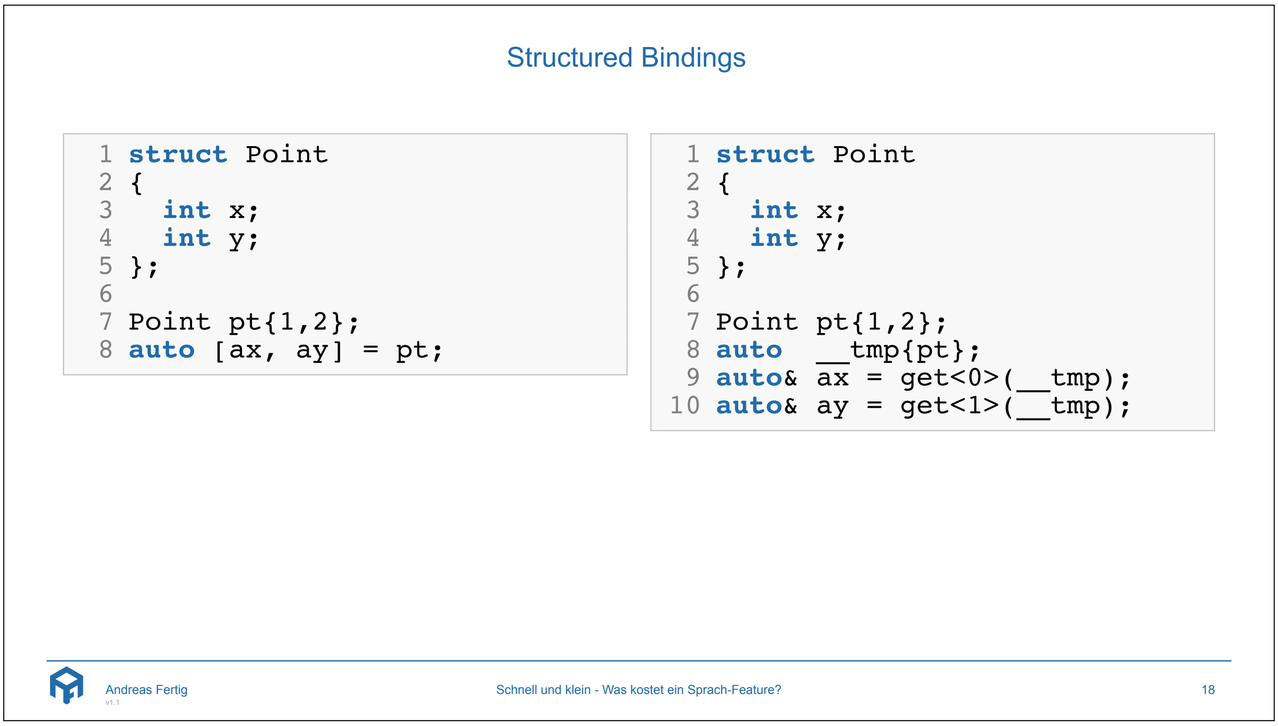
Lambdas

```cpp
int main()
{
  std::string foo;

  auto a = [=]    () { printf( ”%s\n”, foo.c_str()); };

  auto b = [=]    () { };

  auto c = [foo]  () { printf( ”%s\n”, foo.c_str()); };

  auto d = [foo]  () { };

  auto e = [&foo] () { printf( ”%s\n”, foo.c_str()); };

  auto f = [&foo] () { };
}
```



Structured Bindings

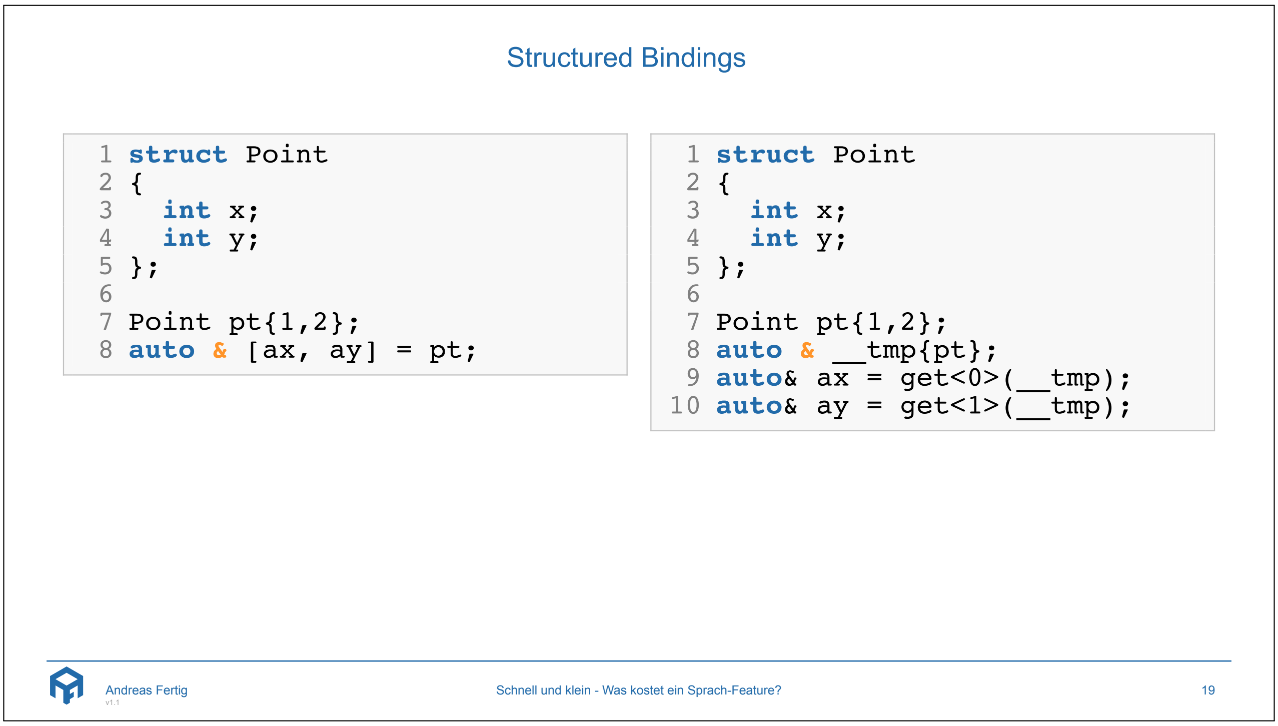
```cpp
struct Point
{
  int x;
  int y;
};

Point pt{1,2};
auto [ax, ay] = pt;
```



Structured Bindings

```cpp
struct Point
{
  int x;
  int y;
};

Point pt{1,2};
auto [ax, ay] = pt;
```

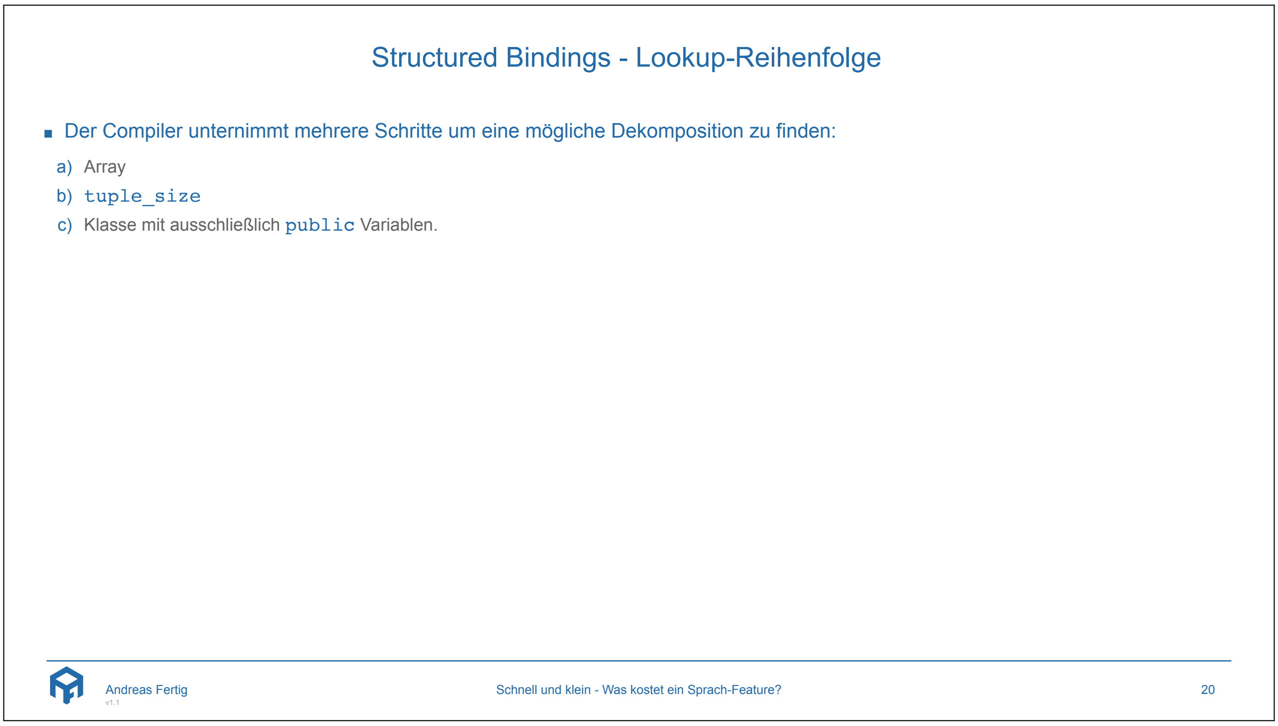
```cpp
struct Point
{
  int x;
  int y;
};

Point pt{1,2};
auto  __tmp{pt};
auto& ax = get<0>(__tmp);
auto& ay = get<1>(__tmp);
```



Structured Bindings

```cpp
struct Point
{
  int x;
  int y;
};

Point pt{1,2};
auto & [ax, ay] = pt;
```

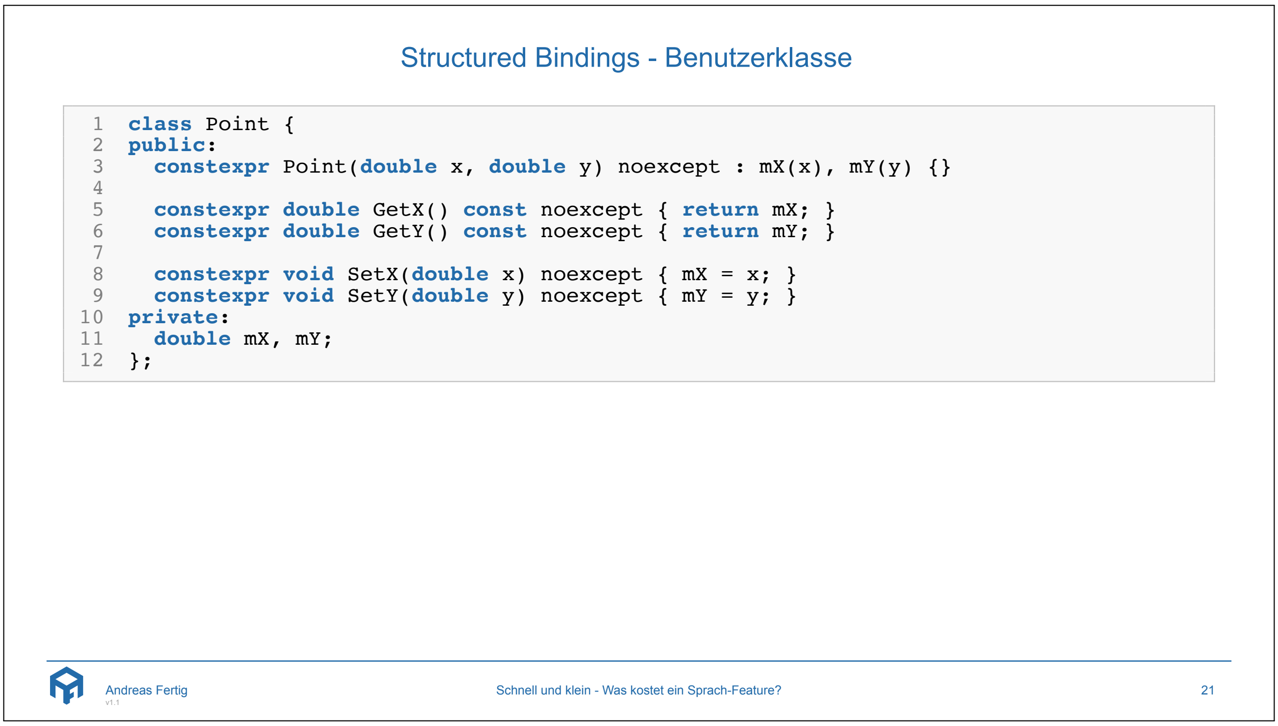
```cpp
struct Point
{
  int x;
  int y;
};

Point pt{1,2};
auto & __tmp{pt};
auto& ax = get<0>(__tmp);
auto& ay = get<1>(__tmp);
```



Structured Bindings - Lookup-Reihenfolge

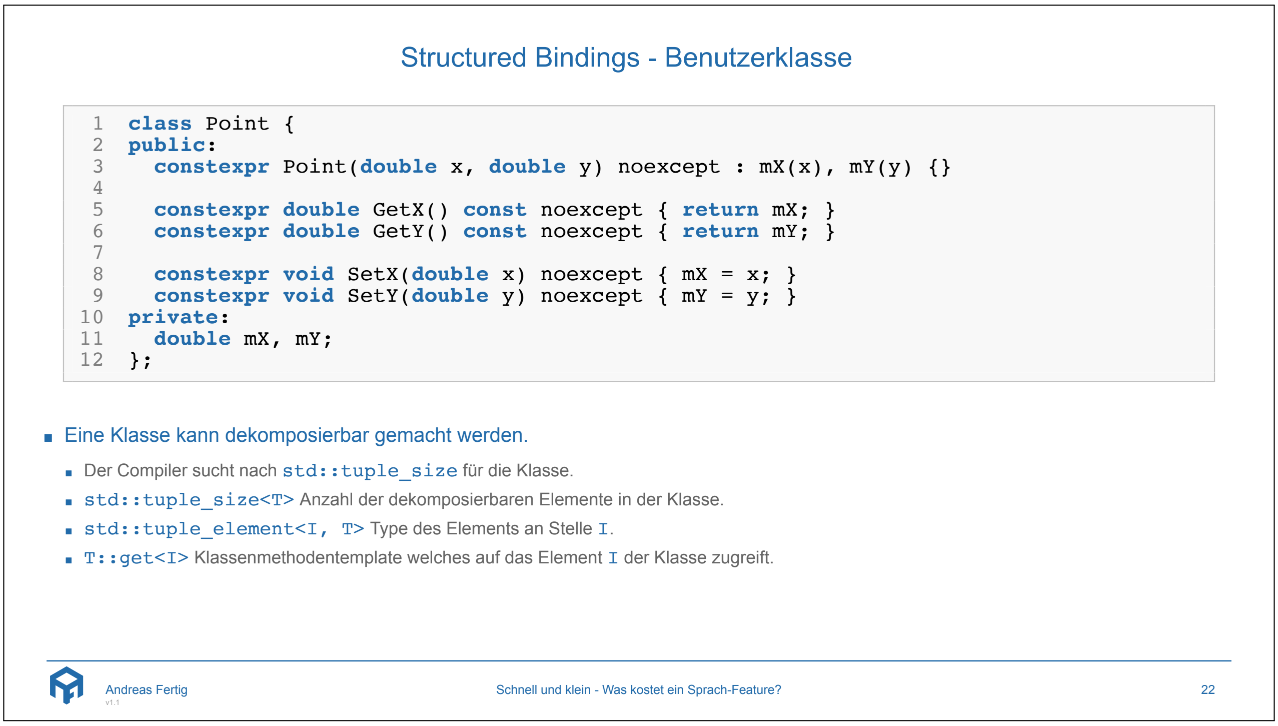
- Der Compiler unternimmt mehrere Schritte um eine mögliche Dekomposition zu finden:

a) Array
b) `tuple_size`
c) Klasse mit ausschließlich `public` Variablen.



Structured Bindings - Benutzerklasse

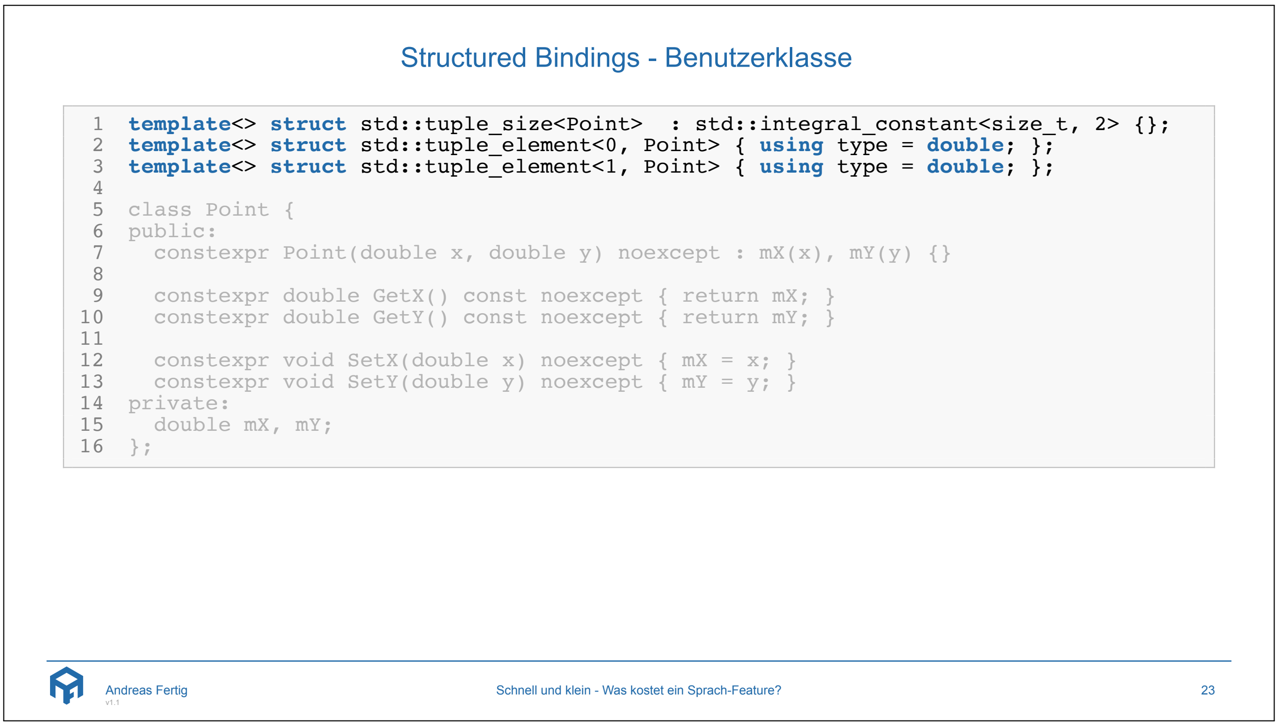
```cpp
class Point {
public:
  constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}

  constexpr double GetX() const noexcept { return mX; }
  constexpr double GetY() const noexcept { return mY; }

  constexpr void SetX(double x) noexcept { mX = x; }
  constexpr void SetY(double y) noexcept { mY = y; }
private:
  double mX, mY;
};
```



Structured Bindings - Benutzerklasse

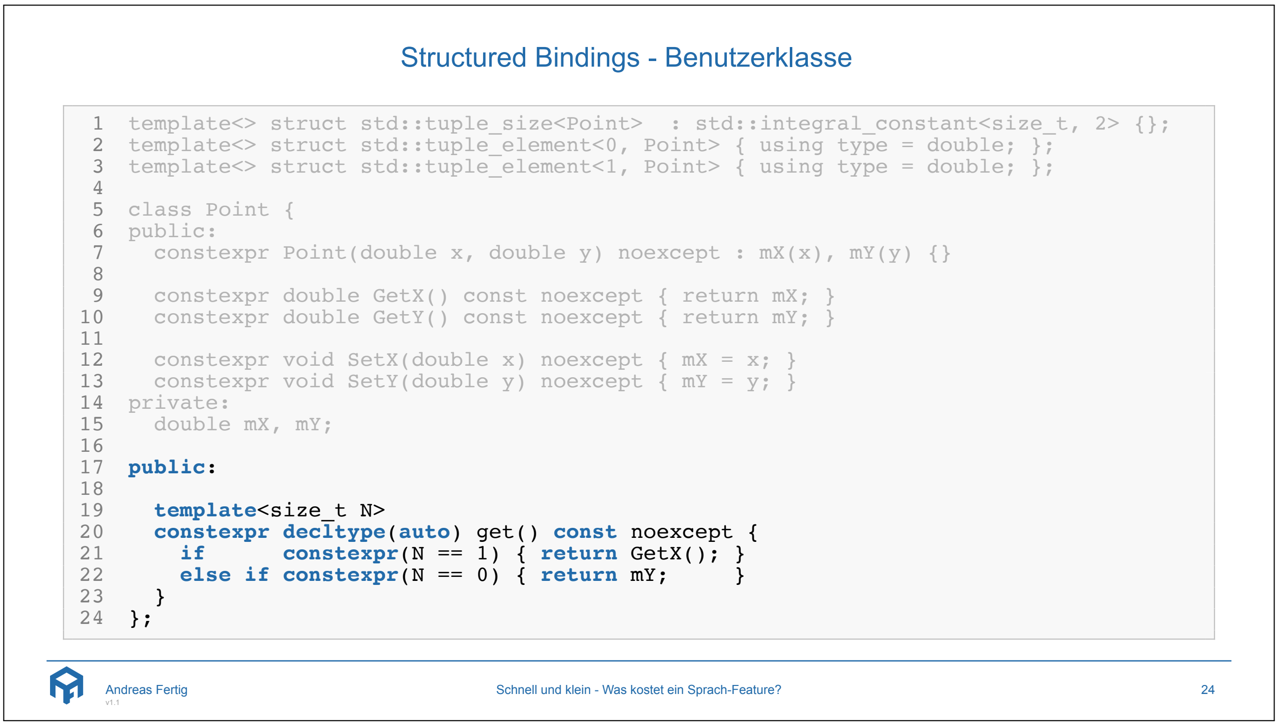
```cpp
class Point {
public:
  constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}

  constexpr double GetX() const noexcept { return mX; }
  constexpr double GetY() const noexcept { return mY; }

  constexpr void SetX(double x) noexcept { mX = x; }
  constexpr void SetY(double y) noexcept { mY = y; }
private:
  double mX, mY;
};
```

- Eine Klasse kann dekomposierbar gemacht werden.
  - Der Compiler sucht nach `std::tuple_size` für die Klasse.
  - `std::tuple_size<T>` Anzahl der dekomposierbaren Elemente in der Klasse.
  - `std::tuple_element<I, T>` Type des Elements an Stelle `I`.
  - `T::get<I>` Klassenmethodentemplate welches auf das Element `I` der Klasse zugreift.



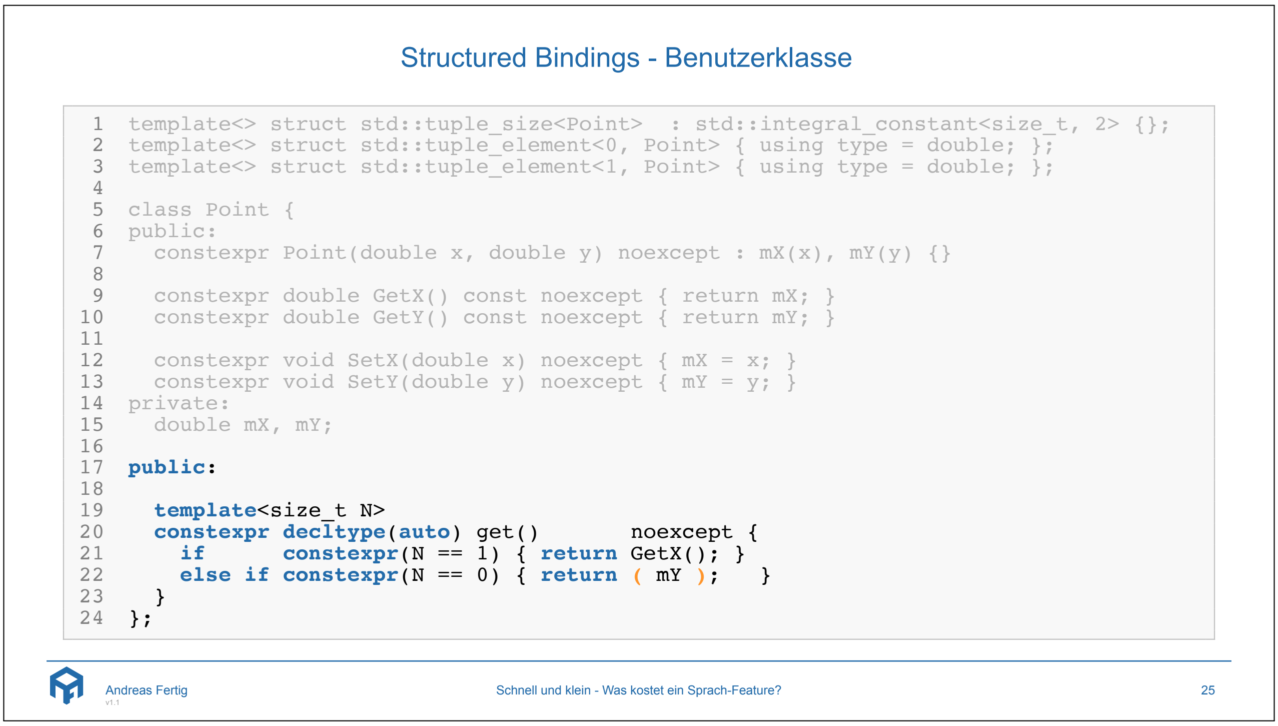
Structured Bindings - Benutzerklasse

```cpp
template<> struct std::tuple_size<Point>  : std::integral_constant<size_t, 2> {};
template<> struct std::tuple_element<0, Point> { using type = double; };
template<> struct std::tuple_element<1, Point> { using type = double; };

class Point {
public:
  constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}

  constexpr double GetX() const noexcept { return mX; }
  constexpr double GetY() const noexcept { return mY; }

  constexpr void SetX(double x) noexcept { mX = x; }
  constexpr void SetY(double y) noexcept { mY = y; }
private:
  double mX, mY;
};
```



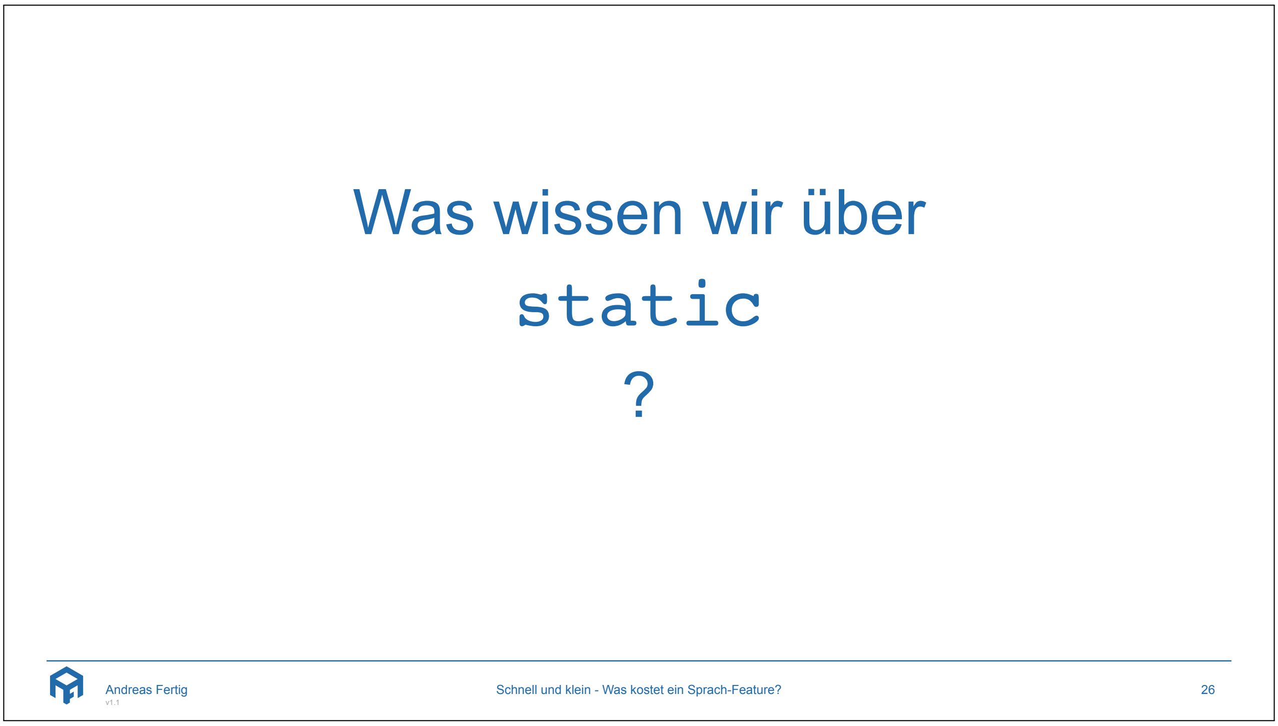
Structured Bindings - Benutzerklasse

```cpp
template<> struct std::tuple_size<Point>  : std::integral_constant<size_t, 2> {};
template<> struct std::tuple_element<0, Point> { using type = double; };
template<> struct std::tuple_element<1, Point> { using type = double; };

class Point {
public:
  constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}

  constexpr double GetX() const noexcept { return mX; }
  constexpr double GetY() const noexcept { return mY; }

  constexpr void SetX(double x) noexcept { mX = x; }
  constexpr void SetY(double y) noexcept { mY = y; }
private:
  double mX, mY;

public:

  template<size_t N>
  constexpr decltype(auto) get() const noexcept {
    if       constexpr(N == 1) { return GetX(); }
    else if constexpr(N == 0) { return mY;     }
  }
};
```

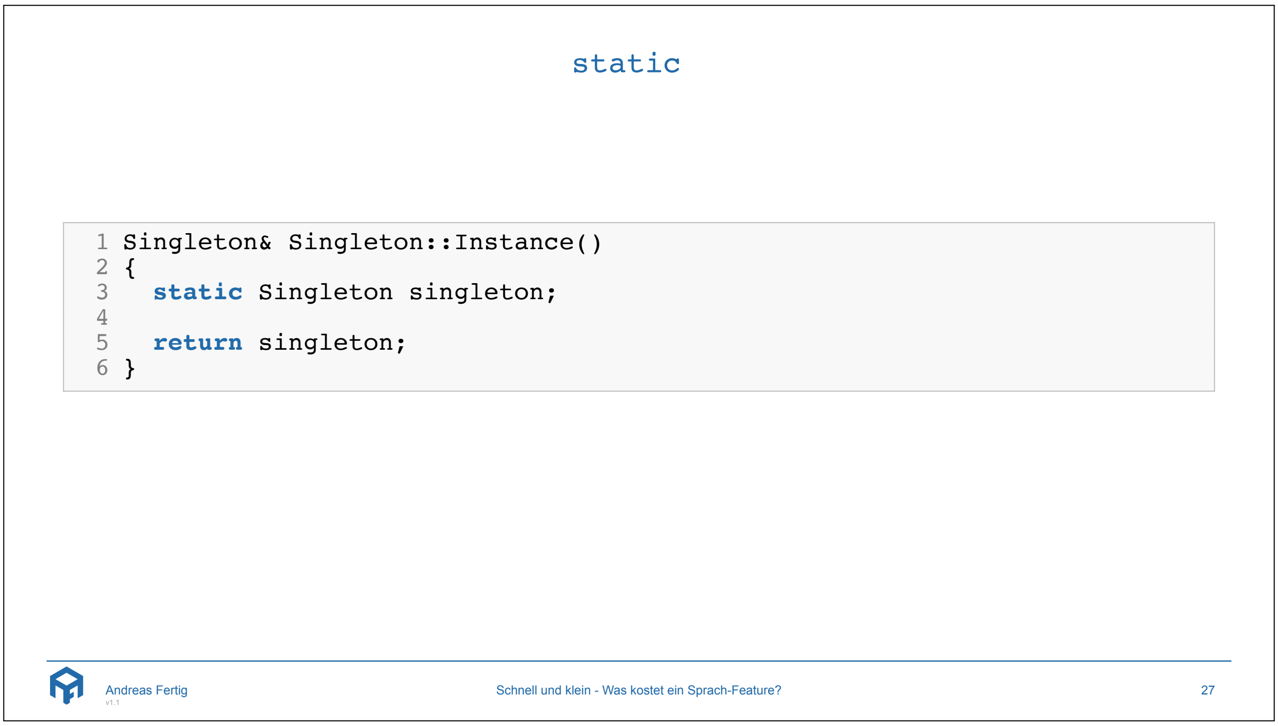


Structured Bindings - Benutzerklasse

```
template<> struct std::tuple_size<Point>  : std::integral_constant<size_t, 2> {};
template<> struct std::tuple_element<0, Point> { using type = double; };
template<> struct std::tuple_element<1, Point> { using type = double; };

class Point {
public:
  constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}

  constexpr double GetX() const noexcept { return mX; }
  constexpr double GetY() const noexcept { return mY; }

  constexpr void SetX(double x) noexcept { mX = x; }
  constexpr void SetY(double y) noexcept { mY = y; }
private:
  double mX, mY;

public:

  template<size_t N>
  constexpr decltype(auto) get()       noexcept {
    if       constexpr(N == 1) { return GetX(); }
    else if constexpr(N == 0) { return ( mY );    }
  }
};
```

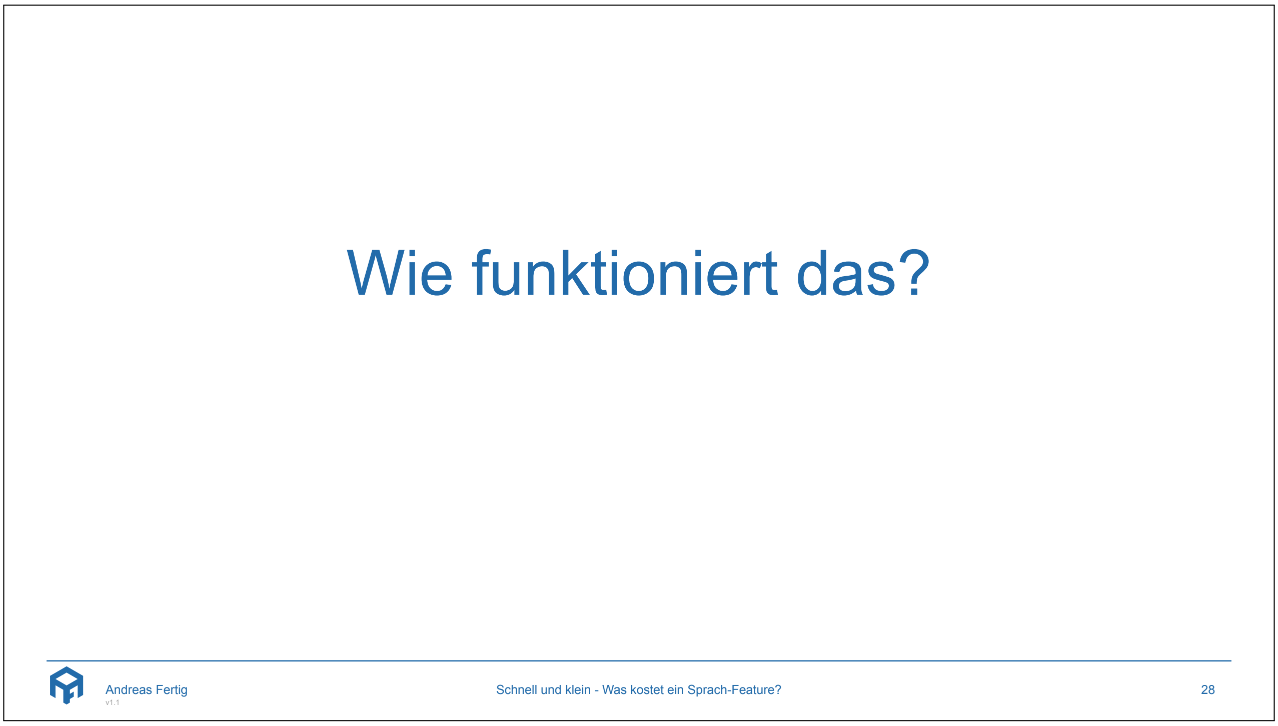


Was wissen wir über `static` ?

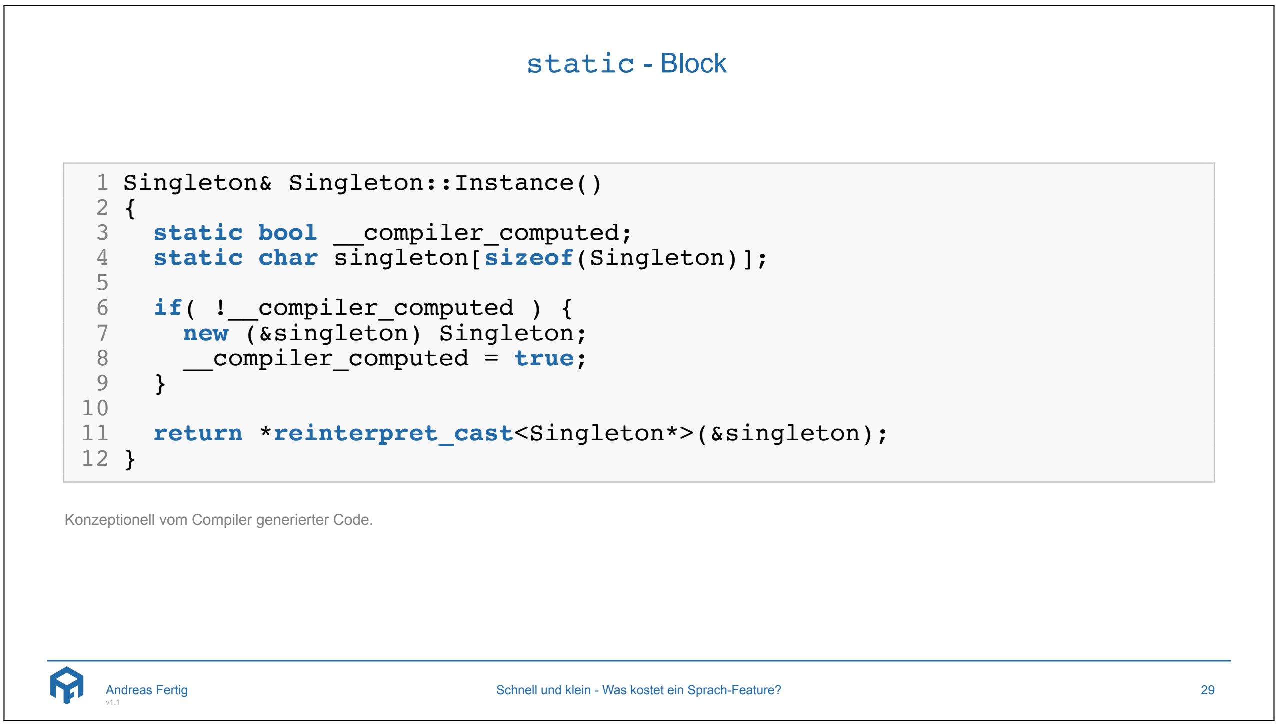


static

```cpp
Singleton& Singleton::Instance()
{
  static Singleton singleton;

  return singleton;
}
```



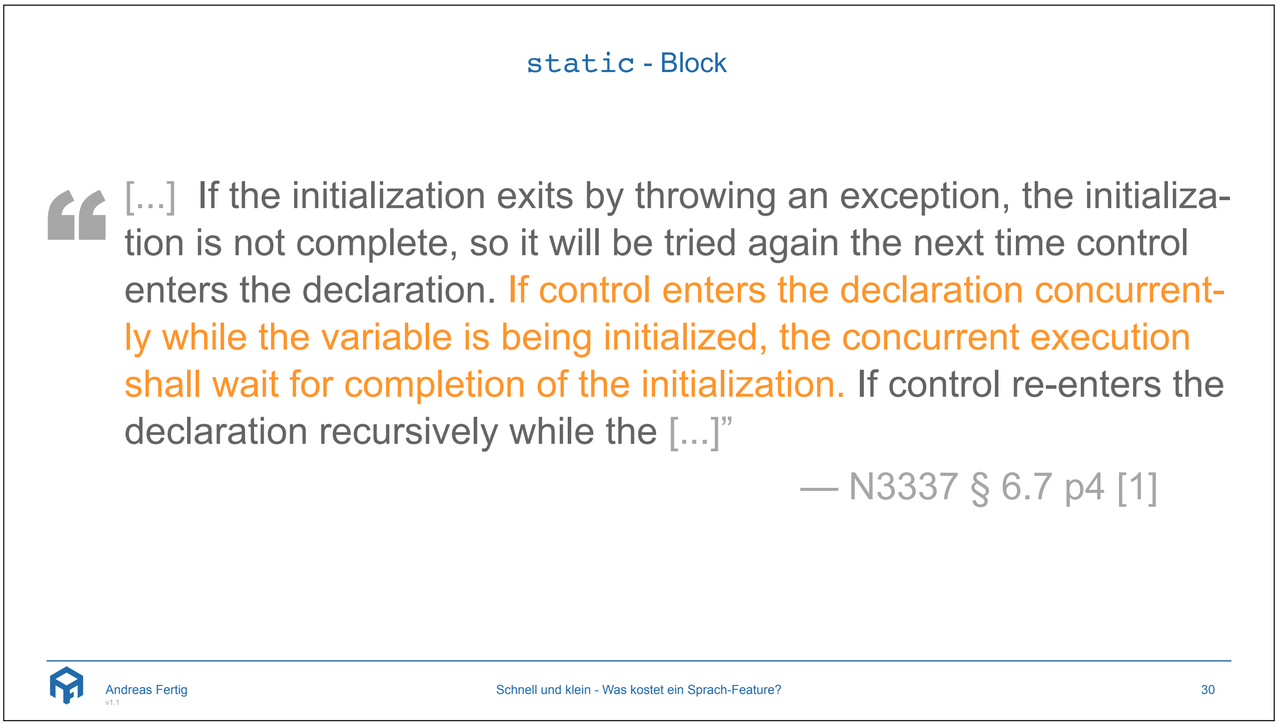
Wie funktioniert das?



`static` - Block

```cpp
Singleton& Singleton::Instance()
{
  static bool __compiler_computed;
  static char singleton[sizeof(Singleton)];

  if( !__compiler_computed ) {
    new (&singleton) Singleton;
    __compiler_computed = true;
  }

  return *reinterpret_cast<Singleton*>(&singleton);
}
```

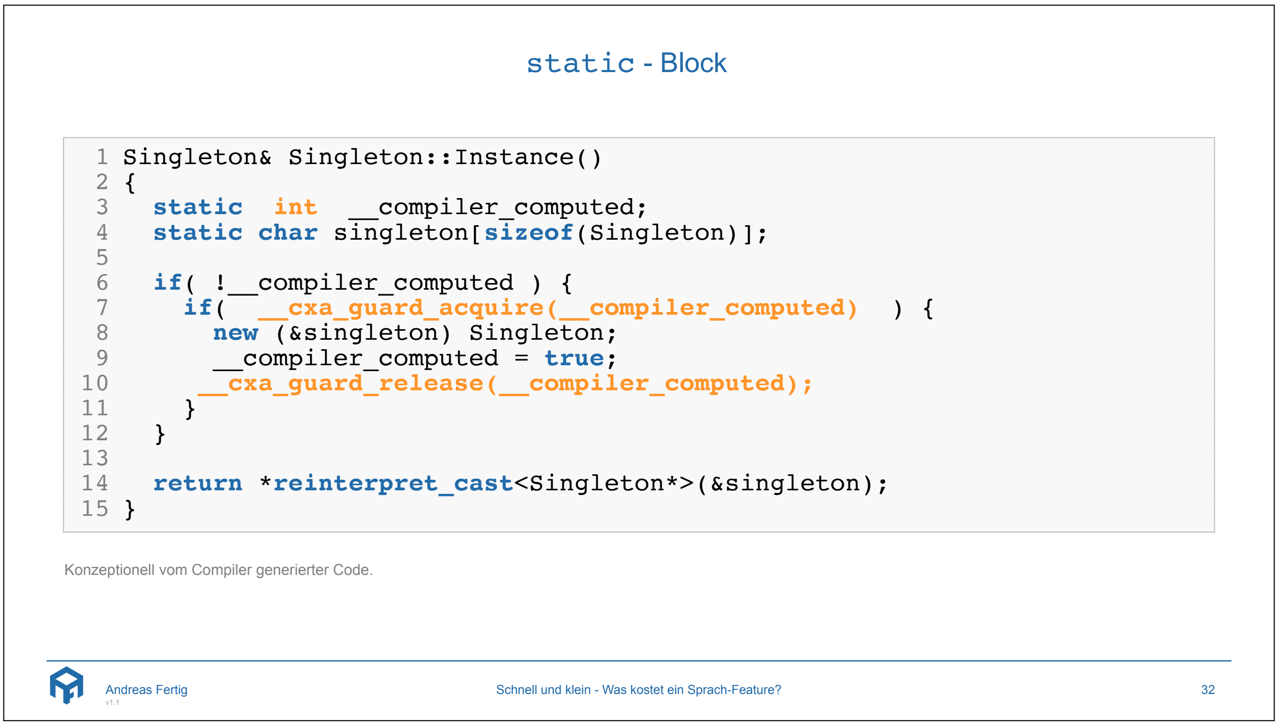
Konzeptionell vom Compiler generierter Code.



`static` - Block

> “[...] If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization. If control re-enters the declaration recursively while the [...]”
>
> — N3337 § 6.7 p4 [1]

Thread-safe?



static - Block

```
Singleton& Singleton::Instance()
{
  static  int   __compiler_computed;
  static char singleton[sizeof(Singleton)];

  if( !__compiler_computed ) {
    if(  __cxa_guard_acquire(__compiler_computed)   ) {
      new (&singleton) Singleton;
      __compiler_computed = true;
     __cxa_guard_release(__compiler_computed);
    }
  }

  return *reinterpret_cast<Singleton*>(&singleton);
}
```

Konzeptionell vom Compiler generierter Code.

}

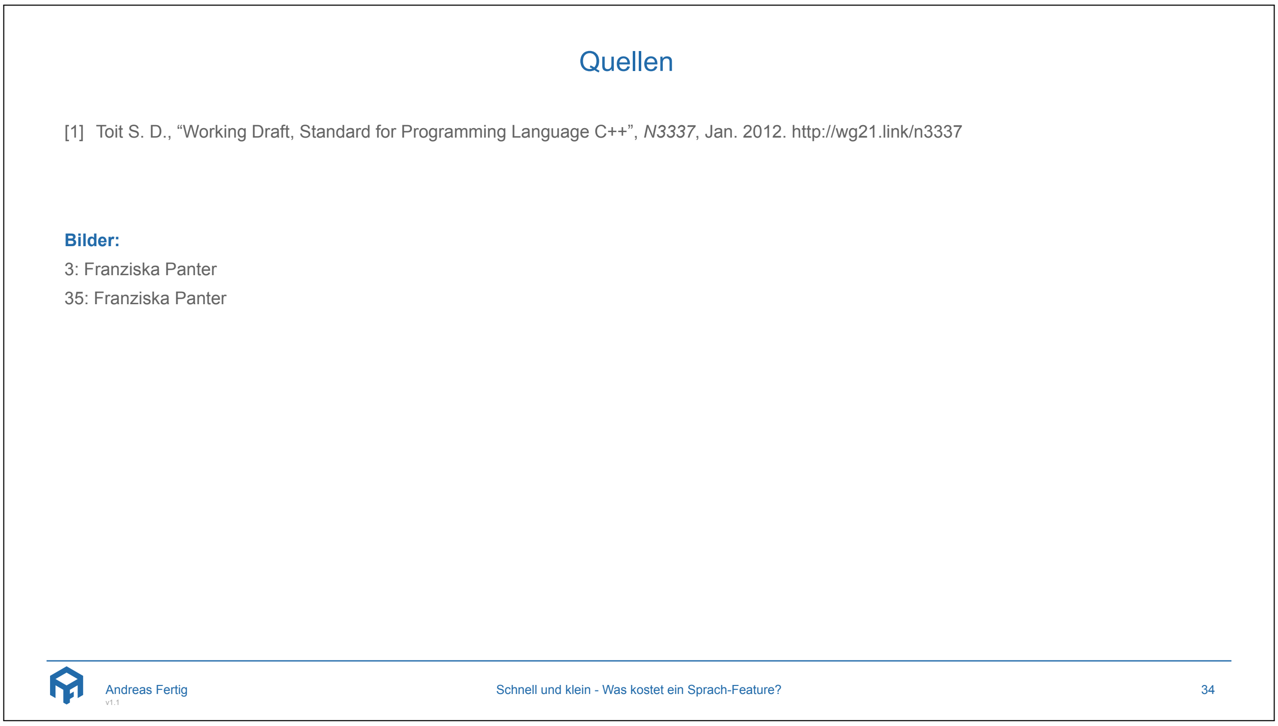
Ich bin Fertig.

Available online:

https://www.AndreasFertig.Info

Images by Franziska Panter:

https://panther-concepts.de



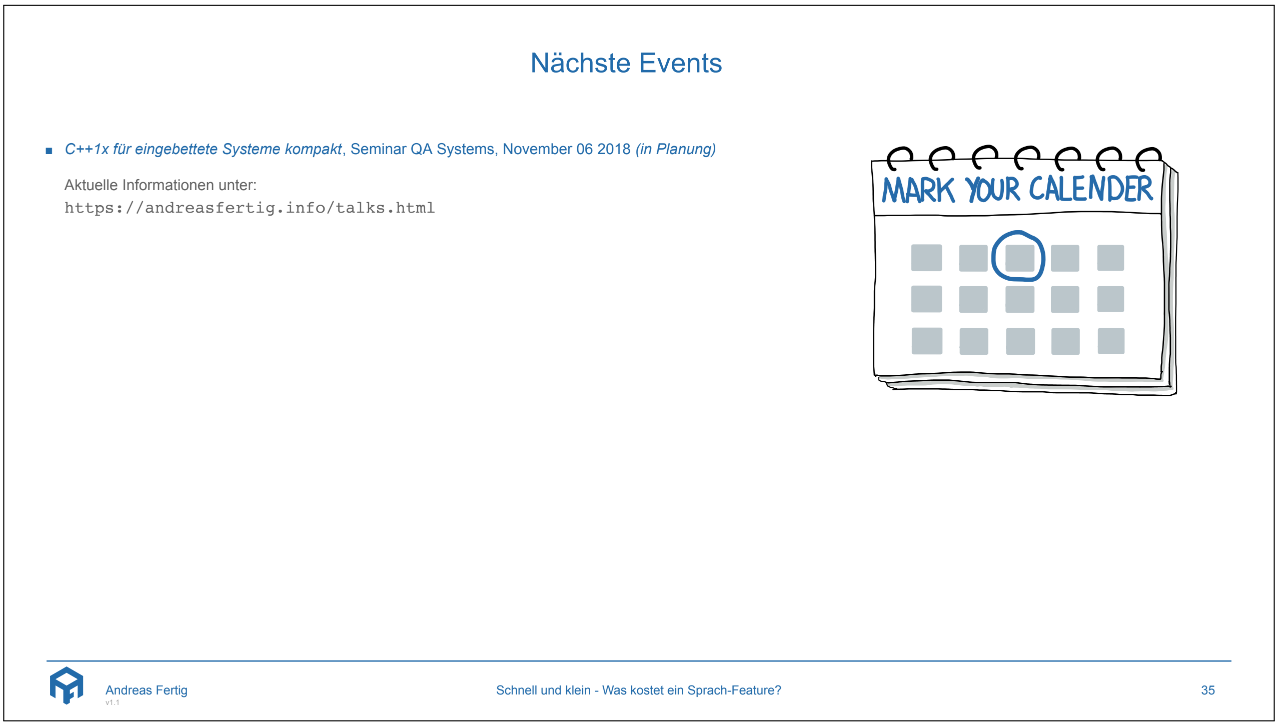
Quellen

[1] Toit S. D., “Working Draft, Standard for Programming Language C++”, *N3337*, Jan. 2012. http://wg21.link/n3337

Bilder:

3: Franziska Panter

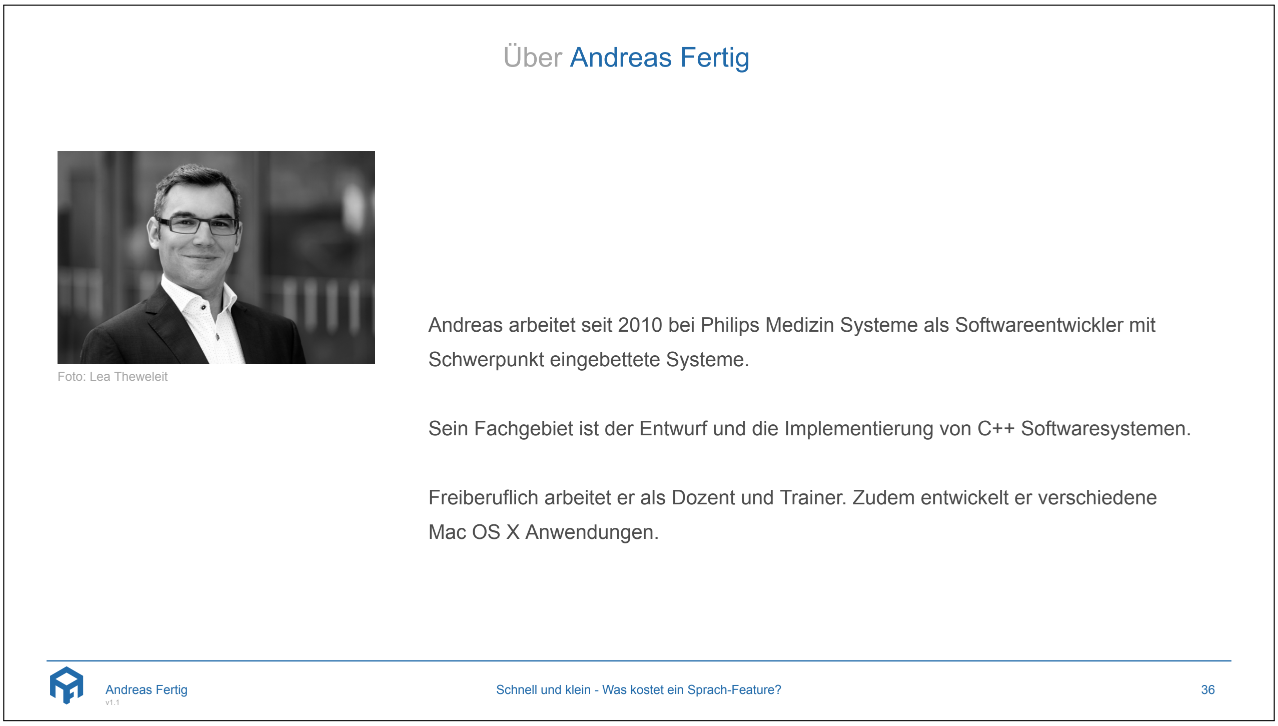
35: Franziska Panter



Nächste Events

- *C++1x für eingebettete Systeme kompakt*, Seminar QA Systems, November 06 2018 *(in Planung)*

Aktuelle Informationen unter:
`https://andreasfertig.info/talks.html`



Über Andreas Fertig

Foto: Lea Theweleit

Andreas arbeitet seit 2010 bei Philips Medizin Systeme als Softwareentwickler mit Schwerpunkt eingebettete Systeme.

Sein Fachgebiet ist der Entwurf und die Implementierung von C++ Softwaresystemen.

Freiberuflich arbeitet er als Dozent und Trainer. Zudem entwickelt er verschiedene Mac OS X Anwendungen.