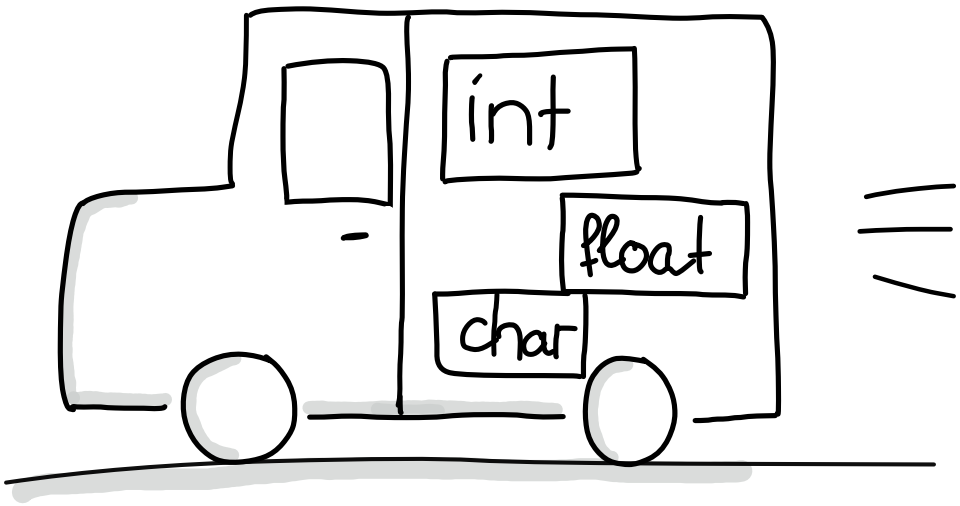
Fast and Small

What are the Costs of Language Features

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

pay only for what you use

auto



decltype(auto)

```
 1
 2
 3 int foo = 1;
 4
 5          auto  a = foo;
 6 decltype(auto) b = foo;
 7
 8          auto  c = (foo);
 9 decltype(auto) d = (foo);
10
11 ++foo;
12
13 printf("a: %d b: %d c: %d d: %d\n", a, b, c, d);
```

decltype(auto)

```
 1
 2
 3 int foo = 1;
 4
 5          auto  a = foo;
 6 decltype(auto) b = foo;
 7
 8          auto  c = (foo);
 9 decltype(auto) d = (foo);
10
11 ++foo;
12
13 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)

```
 1 #define MAX(x,y) (((x) > (y)) ? (x) : (y))
 2
 3 int foo = 1;
 4
 5          auto  a = foo;
 6 decltype(auto) b = foo;
 7
 8          auto  c = MAX(a, b);
 9 decltype(auto) d = MAX(a, b);
10
11 ++foo;
12
13 printf("a: %d b: %d c: %d d: %d\n", a, b, c, d);
```

decltype(auto)

```
 1 #define RANDOM_MACRO(x) (x++)
 2
 3 int foo = 1;
 4
 5          auto  a = foo;
 6 decltype(auto) b = foo;
 7
 8          auto  c = RANDOM_MACRO(foo);
 9 decltype(auto) d = RANDOM_MACRO(foo);
10
11 ++foo;
12
13 printf("a: %d b: %d c: %d d: %d\n", a, b, c, d);
```

decltype(auto)

```
 1 #define RANDOM_MACRO(x) (x++)
 2
 3 int foo = 1;
 4
 5          auto  a = foo;
 6 decltype(auto) b = foo;
 7
 8          auto  c = RANDOM_MACRO(foo);
 9 decltype(auto) d = RANDOM_MACRO(foo);
10
11 ++foo;
12
13 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)

```
 1
 2 #define RANDOM_MACRO(x) (++x)
 3 int foo = 1;
 4
 5          auto  a = foo;
 6 decltype(auto) b = foo;
 7
 8          auto  c = RANDOM_MACRO(foo);
 9 decltype(auto) d = RANDOM_MACRO(foo);
10
11 ++foo;
12
13 printf("a: %d b: %d c: %d d: %d\n", a, b, c, d);
```


decltype(auto)

```
 1
 2 #define RANDOM_MACRO(x) (++x)
 3 int foo = 1;
 4
 5          auto  a = foo;
 6 decltype(auto) b = foo;
 7
 8          auto  c = RANDOM_MACRO(foo);
 9 decltype(auto) d = RANDOM_MACRO(foo);
10
11 ++foo;
12
13 printf("a: %d b: %d c: %d d: %d\n", a, b, c, d);
```

```
$ ./a.out
a: 1 b: 1 c: 2 d: 4
```

decltype(auto)

> [...]
>
> a) if e is an unparenthesized id-expression naming an lvalue or reference introduced from the identifier-list of a decomposition declaration, decltype(e) is the referenced type as given in the specification of the decomposition declaration
>
> b) otherwise, if e is an unparenthesized id-expression or an unparenthesized class member access (5.2.5), decltype(e) is the type of the entity named by e. If there is no such entity, or if e names a set of overloaded functions, the program is ill-formed;
>
> c) otherwise, if e is an xvalue, decltype(e) is T&&, where T is the type of e;
>
> d) otherwise, if e is an lvalue, decltype(e) is T& where T is the type of e;
>
> e) otherwise, decltype(e) is the type of e.
>
> [...]"
>
> — N4640 § 7.1.7.2 p 4 [1]


```
1 std::vector<int> numbers{1, 2, 3, 5};
2
3 for(auto it = numbers.begin(); it != numbers.end(); ++it)
4 {
5   printf("%d\n", *it);
6 }
```

range-based for

```
1 std::vector<int> numbers{1, 2, 3, 5};
2
3 for(auto & it : numbers)
4 {
5   printf("%d\n", it);
6 }
```


range-based for - Behind The Scenes

```
1  {
2    auto && __range = for-range-initializer;
3
4    for ( auto __begin = begin-expr,
5               __end   = end-expr;
6          __begin != __end;
7          ++__begin ) {
8      for-range-declaration = *__begin;
9      statement
10   }
11 }
```

range-based for - Behind The Scenes

```
1  std::vector<int> numbers{1, 2, 3, 5};
2
3  {
4    auto && __range = numbers;
5
6    for ( auto __begin = __range.begin(),
7               __end   = __range.end();
8          __begin != __end;
9          ++__begin ) {
10     auto& it = *__begin;
11     printf("%d\n", it);
12   }
13 }
```


range-based for - Behind The Scenes

```
1  {
2    auto && __range = for-range-initializer;
3    auto __begin = begin-expr;
4    auto __end   = end-expr;
5    for ( ;
6
7          __begin != __end;
8          ++__begin ) {
9      for-range-declaration = *__begin;
10     statement
11   }
12 }
```

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


Lambdas

```
1  int main()
2  {
3    int x = 1;
4
5    auto lambda = [&]() { ++x; };
6
7    lambda();
8
9    return x;
10 }
```

Lambdas

```
1  int main()
2  {
3    int x = 1;
4
5    auto lambda = [&]() { ++x; };
6
7    lambda();
8
9    return x;
10 }
```

```
1  int main()
2  {
3    int x = 1;
4
5    class anon {
6      public:
7      int& _x;
8
9      auto operator()() const
10     { ++_x; }
11   };
12
13   anon lambda{x};
14
15   lambda();
16
17   return x;
18 }
```


Lambdas

```
1  int main()
2  {
3    std::string foo;
4
5    auto a = [=]    () { printf( "%s\n", foo.c_str()); };
6
7    auto b = [=]    () { };
8
9    auto c = [foo]  () { printf( "%s\n", foo.c_str()); };
10
11   auto d = [foo]  () { };
12
13   auto e = [&foo] () { printf( "%s\n", foo.c_str()); };
14
15   auto f = [&foo] () { };
16 }
```

Structured Bindings

```
1 struct Point
2 {
3   int x;
4   int y;
5 };
6
7 Point pt{1,2};
8 auto [x, y] = pt;
```


Structured Bindings

```
1 struct Point
2 {
3   int x;
4   int y;
5 };
6
7 Point pt{1,2};
8 auto [x, y] = pt;
```

```
1 struct Point
2 {
3   int x;
4   int y;
5 };
6
7 Point pt{1,2};
8 auto  __tmp = pt;
9 auto& x = get<0>(pt);
10 auto& y = get<1>(pt);
```

Structured Bindings

```
1 struct Point
2 {
3   int x;
4   int y;
5 };
6
7 Point pt{1,2};
8 auto & [x, y] = pt;
```

```
1 struct Point
2 {
3   int x;
4   int y;
5 };
6
7 Point pt{1,2};
8 auto & __tmp = pt;
9 auto& x = get<0>(pt);
10 auto& y = get<1>(pt);
```


Structured Bindings - Lookup-Order

- The compiler takes several steps to find a possible decomposition:
  a) Array
  b) `tuple_size`
  c) Class with only `public` members.

Structured Bindings - User Class

```
1  class Point {
2  public:
3    constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}
4
5    constexpr double GetX() const noexcept { return mX; }
6    constexpr double GetY() const noexcept { return mY; }
7
8    constexpr void SetX(double x) noexcept { mX = x; }
9    constexpr void SetY(double y) noexcept { mY = y; }
10 private:
11   double mX, mY;
12 };
```


Structured Bindings - User Class

- We can enable decomposition of any class.
  - The compiler searches for `std::tuple_size` of the class.
  - `std::tuple_size<T>` number of decomposable elements in the class.
  - `std::tuple_element<I, T>` type of the element at index `I`.
  - `T::get<I>` class method template to access element `I` of the class.

```
1 template<> struct std::tuple_size<Point>       { constexpr static size_t value = 2; };
```

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

Structured Bindings - User Class

```
1  template<> struct std::tuple_size<Point>  : std::integral_constant<size_t, 2> {};
2  template<> struct std::tuple_element<0, Point> { using type = double; };
3  template<> struct std::tuple_element<1, Point> { using type = double; };
4
5  class Point {
6  public:
7    constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}
8
9    constexpr double GetX() const noexcept { return mX; }
10   constexpr double GetY() const noexcept { return mY; }
11
12   constexpr void SetX(double x) noexcept { mX = x; }
13   constexpr void SetY(double y) noexcept { mY = y; }
14 private:
15   double mX, mY;
16 };
```


Structured Bindings - User Class

```
1  template<> struct std::tuple_size<Point>  : std::integral_constant<size_t, 2> {};
2  template<> struct std::tuple_element<0, Point> { using type = double; };
3  template<> struct std::tuple_element<1, Point> { using type = double; };
4
5  class Point {
6  public:
7    constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}
8
9    constexpr double GetX() const noexcept { return mX; }
10   constexpr double GetY() const noexcept { return mY; }
11
12   constexpr void SetX(double x) noexcept { mX = x; }
13   constexpr void SetY(double y) noexcept { mY = y; }
14 private:
15   double mX, mY;
16
17 public:
18
19   template<size_t N>
20   constexpr decltype(auto) get() const noexcept {
21     if      constexpr(N == 1) { return GetX(); }
22     else if constexpr(N == 0) { return mY;     }
23   }
24 };
```

Structured Bindings - User Class

```
1  template<> struct std::tuple_size<Point>  : std::integral_constant<size_t, 2> {};
2  template<> struct std::tuple_element<0, Point> { using type = double; };
3  template<> struct std::tuple_element<1, Point> { using type = double; };
4
5  class Point {
6  public:
7    constexpr Point(double x, double y) noexcept : mX(x), mY(y) {}
8
9    constexpr double GetX() const noexcept { return mX; }
10   constexpr double GetY() const noexcept { return mY; }
11
12   constexpr void SetX(double x) noexcept { mX = x; }
13   constexpr void SetY(double y) noexcept { mY = y; }
14 private:
15   double mX, mY;
16
17 public:
18
19   template<size_t N>
20   constexpr decltype(auto) get()       noexcept {
21     if      constexpr(N == 1) { return GetX(); }
22     else if constexpr(N == 0) { return ( mY );   }
23   }
24 };
```

What do we know about `static` ?

static

```
1 static size_t counter = 0;
2
3 Singleton& Singleton::Instance()
4 {
5   static Singleton singleton;
6
7   return singleton;
8 }
```


static

> [...] Dynamic initialization of a block-scope variable with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. [...]"
>
> — N4640 § 6.7 p4 [1]

How does this work?

`static` - Block

```
1  Singleton& Singleton::Instance()
2  {
3    static bool __compiler_computed;
4    static char singleton[sizeof(Singleton)];
5
6    if( !__compiler_computed ) {
7      new (&singleton) Singleton;
8      __compiler_computed = true;
9    }
10
11   return *reinterpret_cast<Singleton*>(&singleton);
12 }
```

Conceptual what the compiler generates.

`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 [2]

Thread-safe?

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

static - Block

```
1  Singleton& Singleton::Instance()
2  {
3    static  int  __compiler_computed;
4    static char singleton[sizeof(Singleton)];
5
6    if( !__compiler_computed ) {
7      if(  __cxa_guard_acquire(__compiler_computed)  ) {
8        new (&singleton) Singleton;
9        __compiler_computed = true;
10       __cxa_guard_release(__compiler_computed);
11     }
12   }
13
14   return *reinterpret_cast<Singleton*>(&singleton);
15 }
```

Conceptual what the compiler generates.

static - Block

```
1  class Singleton {
2  public:
3     static Singleton& Instance() {
4         static Singleton singleton;
5         return singleton;
6     }
7
8     int  Get()      const { return mX; }
9     void Set(int x)       { mX = x; }
10
11 private:
12    int mX;
13
14
15 };
16
17 int main(int argc, char* argv[]) {
18    Singleton& s = Singleton::Instance();
19
20    s.Set(argc);
21
22    return s.Get();
23 }
```

static - Block

```
 1  class Singleton {
 2  public:
 3     static Singleton& Instance() {
 4         static Singleton singleton;
 5         return singleton;
 6     }
 7
 8     int  Get()      const { return mX; }
 9     void Set(int x)       { mX = x; }
10
11  private:
12     int mX;
13
14     Singleton() : mX{0} {}
15  };
16
17  int main(int argc, char* argv[]) {
18     Singleton& s = Singleton::Instance();
19
20     s.Set(argc);
21
22     return s.Get();
23  }
```

static - Block

```
 1  class Singleton {
 2  public:
 3     static Singleton& Instance() {
 4         static Singleton singleton;
 5         return singleton;
 6     }
 7
 8     int  Get()      const { return mX; }
 9     void Set(int x)       { mX = x; }
10
11  private:
12     int mX;
13
14     Singleton() = default;
15  };
16
17  int main(int argc, char* argv[]) {
18     Singleton& s = Singleton::Instance();
19
20     s.Set(argc);
21
22     return s.Get();
23  }
```

static - Block

```
 1  class Singleton {
 2  public:
 3     static Singleton& Instance() {
 4         static Singleton singleton;
 5         return singleton;
 6     }
 7
 8     int  Get()      const { return mX; }
 9     void Set(int x)       { mX = x; }
10
11  private:
12     int mX;
13
14     ~Singleton() {}
15  };
16
17  int main(int argc, char* argv[]) {
18     Singleton& s = Singleton::Instance();
19
20     s.Set(argc);
21
22     return s.Get();
23  }
```

static - Block

```
 1  class Singleton {
 2  public:
 3     static Singleton& Instance() {
 4         static Singleton singleton;
 5         return singleton;
 6     }
 7
 8     int  Get()      const { return mX; }
 9     void Set(int x)       { mX = x; }
10
11  private:
12     int mX;
13
14     virtual ~Singleton() = default;
15  };
16
17  int main(int argc, char* argv[]) {
18     Singleton& s = Singleton::Instance();
19
20     s.Set(argc);
21
22     return s.Get();
23  }
```

}

Ich bin Fertig.

Available online:

https://www.AndreasFertig.Info

Images by Franziska Panter:

https://panther-concepts.de

References

[1] Smith R., “Working Draft, Standard for Programming Language C++”, *N4640*, Feb. 2016. http://wg21.link/n4640

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

Images:

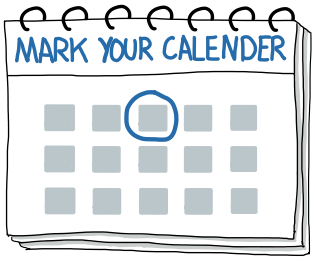
3: Franziska Panter

45: Franziska Panter

Upcoming Events

- *Use the power of the language*, Keynote Address, Clean Code Days, June 21 2017
- *C++1x für eingebettete Systeme kompakt*, Seminar QA Systems, November 21 2017

To keep in the loop, periodically check my *Talks and Training* (`https://andreasfertig.info/talks.html`) page.



About Andreas Fertig

Andreas holds an M.S. in Computer Science from Karlsruhe University of Applied Sciences. Since 2010 he has been a software developer and architect for Philips Medical Systems focus on embedded systems.

He has profound practical and theoretical knowledge of C++ at various operating systems.

He works freelance as a lecturer and trainer. Besides this he develops various Mac OS X applications.