Fast and Small

What are the Costs of Language Features

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

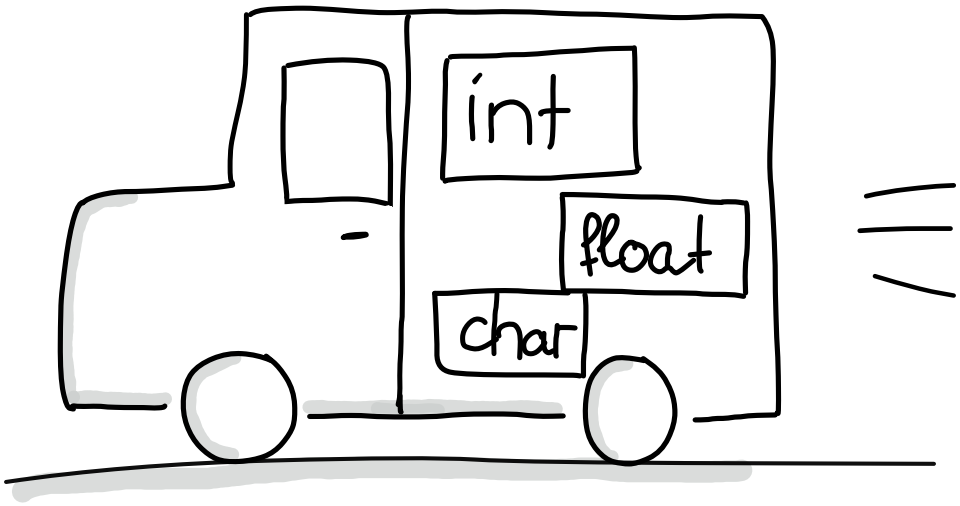
fertig

adjective /ˈfɛrtɪç/

finished
ready
complete
completed

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);
```


return (x);

`return (x);`

> [...] A function returns to its caller by means of the return statement, which has one of the forms
> return ;
> return (expression) ; [...]"
>
> — C-Reference-Manual § 9.10 [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 }
```

```
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   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 [ax, ay] = 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 };
```


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

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>   : 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 };
```

What do we know about `static` ?

`static`

```
1 Singleton& Singleton::Instance()
2 {
3   static Singleton singleton;
4
5   return singleton;
6 }
```

How does this work?

`static` - Block

```
1  Singleton& Singleton::Instance()
2  {
3    static bool __compiler_computed;
4    alignas(Singleton) 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?

`static` - Block

```
 1 Singleton& Singleton::Instance()
 2 {
 3   static  int   __compiler_computed;
 4   alignas(Singleton) static char singleton[sizeof(Singleton)];
 5
 6   if( !__compiler_computed ) {
 7     if(  __cxa_guard_acquire(__compiler_computed)  ) {
 8       if( !__compiler_computed ) {
 9         new (&singleton) Singleton;
10         __compiler_computed = true;
11       }
12       __cxa_guard_release(__compiler_computed);
13     }
14   }
15
16   return *reinterpret_cast<Singleton*>(&singleton);
17 }
```

Conceptual what the compiler generates. See [3] for details.

}

I am Fertig.

https://cppinsights.io

Available online:

https://www.AndreasFertig.Info

Images by Franziska Panter:

https://panther-concepts.de

Used Compilers

- Compilers used to compile (most of) the examples.
  - g++-8-20170910 (GCC) 8.0.0 20170910 (experimental)
  - clang version 9.0.0 (https://github.com/llvm-mirror/clang.git 1fa8b1fbd40d147ef9fa0fe73fbd44977250989f) (https://github.com/llvm-mirror/llvm.git a7866b1030537ff197099251d5ee6b46b35c13e2)

References

[1] Ritchie D. M., “C reference manual”, 1980. https://www.bell-labs.com/usr/dmr/www/cman.pdf

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

[3] “cxa_guard”. www.opensource.apple.com/source/libcppabi/libcppabi-14/src/cxa_guard.cxx

Images:

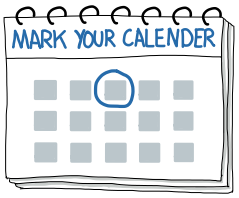
4: Franziska Panter

31: Franziska Panter

Upcoming Events

- *C++: λ Demystified*, Core C++, May 15 2019
- *C++1x für eingebettete Systeme kompakt*, Seminar QA Systems, November 14 2018
- *C++ Templates - die richtige Dosis kompakt*, Seminar QA Systems, November 15 2018

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 focusing on embedded systems. He has a profound knowledge of C++ and is a frequent SG14 member.

He works freelance as a lecturer and trainer. Besides this he develops macOS applications and is the creator of cppinsights.io