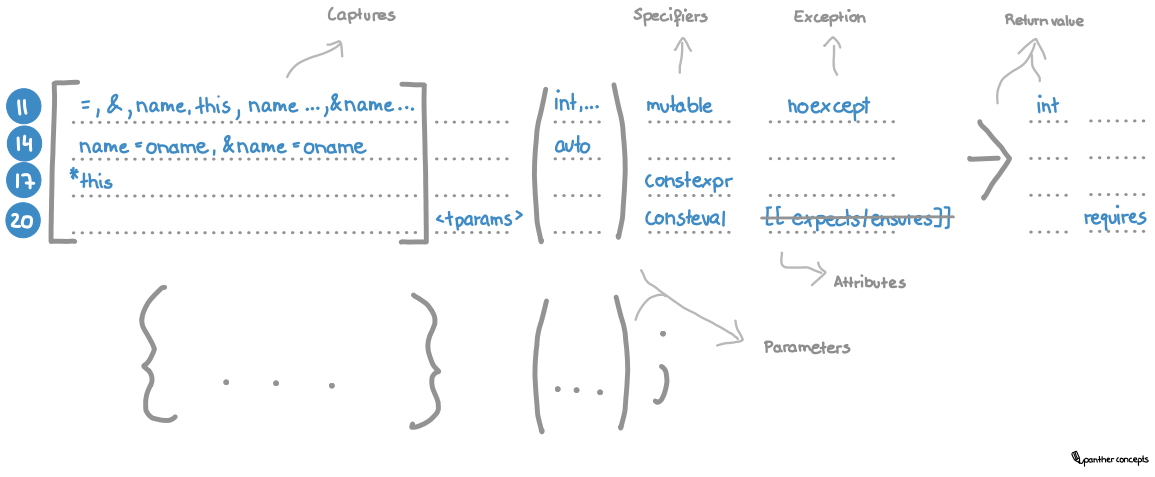
C++: λ Demystified

Ein Blick hinter die Kulissen

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

Lambda Evolution



Lambda Interna

```
1 int main()
2 {
3   const char hello[]{"Hallo ESE Kongress"};
4
5   [&] { printf("%s\n", hello); }();
6 }
```


Einfangen von globalen Variablen

```
1 int x{2};
2
3 int main()
4 {
5   [] { ++x; }();
6 }
```

Generische Lambdas

C++14

- Verfügen über einen Call-Operator der ein Template ist, mit einem `auto` Rückgabetyp.
- Alle `auto`-Parameter sind Template-Parameter.

```
1 auto l = []( auto  v) { return v * 2; };
2
3 auto d = l(2.0);
4 auto i = l(2);
```


Generische Lambdas

C++14

- In Kombination mit C++17's `constexpr if` können wir Lambdas mit verschiedenen Rückgabetypen erzeugen.

```
1  auto l = [](auto v) {
2    if constexpr(std::is_same_v<decltype(v), double>) {
3      return v * 2.0;
4    } else {
5      return v * 2;
6    }
7  };
8
9  auto d = l(2.0);
10 auto i = l(2);
```

Die ungültige Referenz-Falle

- Ist dieses unschuldig aussehene Lambda okay?

```
1  auto Func()
2  {
3    int x{22};
4
5    auto l = [&] { return x * x; };
6
7    // a bunch of code follows.
8
9    return l;
10 }
```


Die ungültige Referenz-Falle

- Ist dieses unschuldig aussehene Lambda okay?

> [...] If a non-reference entity is implicitly or explicitly captured by reference, invoking the function call operator of the corresponding lambda-expression after the lifetime of the entity has ended is likely to result in undefined behavior. [...]"
>
> — N4800 § 7.5.5.2 p16 [1]

- Als Regel: Nur Variablen als Referenz einfangen, wenn das Lambda an eine Funktion weitergereicht oder lokal genutzt wird.

```
1  auto Func()
2  {
3    int x{22};
4
5    auto l = [ = ] { return x * x; };
6
7    // a bunch of code follows.
8
9    return l;
10 }
```

Die ungültige Referenz-Falle

- Ist dieses unschuldig aussehene Lambda okay?
- Als Regel: Nur Variablen als Referenz einfangen, wenn das Lambda an eine Funktion weitergereicht oder lokal genutzt wird.
- Eine kleine Änderung, immer noch Einfangen als Kopie.

```
1  auto Func()
2  {
3    int* x = new int{22};
4
5    auto l = [=] { return (*x) * (*x); };
6
7    // a bunch of code follows.
8
9    return l;
10 }
```


Die ungültige Referenz-Falle

- Ist dieses unschuldig aussehene Lambda okay?
- Als Regel: Nur Variablen als Referenz einfangen, wenn das Lambda an eine Funktion weitergereicht oder lokal genutzt wird.
- Eine kleine Änderung, immer noch Einfangen als Kopie.
- Autsch...

```
1  auto Func()
2  {
3    int* x = new int{22};
4
5    auto l = [=] { return (*x) * (*x); };
6
7    // a bunch of code follows.
8    // and in the middle of that code:
9    delete x;
10
11   return l;
12 }
```

Die ungültige Referenz-Falle

- Ist dieses unschuldig aussehene Lambda okay?
- Als Regel: Nur Variablen als Referenz einfangen, wenn das Lambda an eine Funktion weitergereicht oder lokal genutzt wird.
- Eine kleine Änderung, immer noch Einfangen als Kopie.
- Autsch...
- Kein Problem mit schlauen Zeigern (*smart pointers*).

```
1  auto Func()
2  {
3    auto x = make_shared<int>(22);
4
5    auto l = [=] { return (*x) * (*x); };
6
7    // a bunch of code follows.
8
9    return l;
10 }
```


Captures

```
1  class Test
2  {
3  public:
4    Test(int x)
5    : a{x}
6    {
7      auto l1 = [=] { return a + 2; };
8
9      printf("l1: %d\n", l1());
10
11     ++a;
12
13     printf("l1: %d\n", l1());
14   }
15
16   int a;
17 };
18
19 int main()
20 {
21   Test t{2};
22 }
```

Captures

```
1  class Test
2  {
3  public:
4    Test(int x)
5    : a{x}
6    {
7      auto l1 = [=] { return a + 2; };
8
9      printf("l1: %d\n", l1());
10
11     ++a;
12
13     printf("l1: %d\n", l1());
14   }
15
16   int a;
17 };
18
19 int main()
20 {
21   Test t{2};
22 }
```

```
$ ./a.out
l1: 4
l1: 5
```


Captures

C++17

```
1  class Test
2  {
3  public:
4    Test(int x)
5    : a{x}
6    {
7      auto l1 = [ * this ] { return a + 2; };
8
9      printf("l1: %d\n", l1());
10
11     ++a;
12
13     printf("l1: %d\n", l1());
14   }
15
16   int a;
17 };
18
19 int main()
20 {
21   Test t{2};
22 }
```

```
$ ./a.out
l1: 4
l1: 4
```

Captures

C++17

```
1  class Test
2  {
3  public:
4    Test(int x)
5    : a{x}
6    {
7      auto l2 = [*this] { return a + 2; };
8    }
9
10   int a;
11   int b;
12 };
```

Captures

C++14

```
1  class Test
2  {
3  public:
4    Test(int x)
5    : a{x}
6    {
7      auto l2 = [al = a] { return al + 2; };
8    }
9
10   int a;
11   int b;
12 };
```

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

constexpr Lambdas

C++17

- Mit C++17 können Lambdas in `constexpr`-Kontexten genutzt werden. Algorithmusfunktionen von [2].

```
 1  template<class InputIt, class UnaryPredicate>
 2 constexpr InputIt find_if_not(InputIt first, InputIt last, UnaryPredicate q)
 3 {
 4   for(; first != last; ++first) {
 5     if(!q(*first)) {
 6       return first;
 7     }
 8   }
 9   return last;
10 }
11
12 template<class InputIt, class UnaryPredicate>
13 constexpr bool all_of(InputIt first, InputIt last, UnaryPredicate p)
14 {
15   return find_if_not(first, last, p) == last;
16 }
17
18 int main()
19 {
20   constexpr int  ar[5]{1, 3, 5, 7, 9};
21   constexpr bool allEven =
22     all_of(&ar[0], &ar[5], [](int i) { return (i % 2) == 0; });
23
24   return allEven;
25 }
```


constexpr Lambdas

C++20

- Mit C++17 können Lambdas in `constexpr`-Kontexten genutzt werden.
- Dank P0202 [3] werden mehr Algorithmen in C++20 funktionieren.

```
 1 #include <algorithm>
 2 #include <array>
 3
 4 int main()
 5 {
 6   constexpr std::array<int, 5> ar{1, 3, 5, 7, 9};
 7   constexpr bool               allEven =
 8     std::all_of(ar.begin(), ar.end(), [](int i) { return (i % 2) == 0; });
 9
10   return allEven;
11 }
```

Lambdas Angewandt

- Wo und wie sind Lambdas hilfreich?
  - Wenn zusätzliche Funktionalität vor oder nach einer Aktion benötigt wird.

```
1  template<typename T>
2  void CodeGenerator::WrapInParensOrCurlys(const BraceKind        braceKind,
3                                           T&&                    lambda,
4                                           const AddSpaceAtTheEnd addSpaceAtTheEnd)
5  {
6    if(BraceKind::Curlys == braceKind) {
7      mOutputFormatHelper.Append('{');
8    } else {
9      mOutputFormatHelper.Append('(');
10   }
11
12   lambda();
13
14   if(BraceKind::Curlys == braceKind)
15   {
16     mOutputFormatHelper.Append('}');
17   }
18   else { mOutputFormatHelper.Append(')'); }
19
20   if(AddSpaceAtTheEnd::Yes == addSpaceAtTheEnd) {
21     mOutputFormatHelper.Append(' ');
22   }
23 }
```

Von C++ Insights.

Lambdas Angewandt

- Wo und wie sind Lambdas hilfreich?
  - Aufräumen / freigeben von Ressourcen.

```
1  size_t ReadData(span<char> buffer)
2  {
3    int fd = Open(/*some well known file*/);
4
5    if-(1 == fd) {
6      return 0;
7    }
8
9    const auto len =
10     read(fd, buffer.data(), buffer.size());
11
12   if-(1 == len) {
13     return 0;
14   }
15
16   ftruncate(fd, len);
17
18   close(fd);
19
20   return gsl::narrow_cast<size_t>(len);
21 }
```

Lambdas Angewandt

- Wo und wie sind Lambdas hilfreich?
  - Aufräumen / freigeben von Ressourcen.

```cpp
size_t ReadData(span<char> buffer)
{
  int  fd = Open(/* some well known file*/);
  FinalAction cleanup{[&] {
    if(-1 != fd) {
      close(fd);
    }
  }};

  if(-1 == fd) {
    return 0;
  }

  const auto len =
    read(fd, buffer.data(), buffer.size());

  if(-1 == len) {
    return 0;
  }

  ftruncate(fd, len);

  return gsl::narrow_cast<size_t>(len);
}
```


Lambdas Angewandt

- Wo und wie sind Lambdas hilfreich?
  - Aufräumen / freigeben von Ressourcen.

```cpp
template<typename T>
class FinalAction
{
public:
  explicit FinalAction(T&& action)
  : mAction{std::forward<T>(action)}
  {
  }

  ~FinalAction() { mAction(); }

private:
  T mAction;
};
```

```cpp
size_t ReadData(span<char> buffer)
{
  int  fd = Open(/* some well known file*/);
  FinalAction cleanup{[&] {
    if(-1 != fd) {
      close(fd);
    }
  }};

  if(-1 == fd) {
    return 0;
  }

  const auto len =
    read(fd, buffer.data(), buffer.size());

  if(-1 == len) {
    return 0;
  }

  ftruncate(fd, len);

  return gsl::narrow_cast<size_t>(len);
}
```

Template Lambdas

C++20

```
1 int main()
2 {
3   auto max = [](auto x, auto y) {
4     return (x > y) ? x : y;
5   };
6
7   max(2, 3);    // ok
8   max(2, 3.0);  // not wanted
9 }
```


Template Lambdas

C++20

```
1 int main()
2 {
3   auto max = []<typename T>(T x, T y)
4   {
5     return (x > y) ? x : y;
6   };
7
8   max(2, 3);  // ok
9   // max(2, 3.0);  // does not compile anymore
10 }
```

Template Lambdas

C++20

```
1 auto lambda = []<typename T>(std::vector<T> t){};
2 std::vector<int> v{};
3
4 lambda(v);
5 // lambda(20);
```

```
1 #include <array>
2
3 int main()
4 {
5   auto l = []<size_t N>(std::array<int, N> x) {};
6
7   std::array<int, 2> a{};
8
9   l(a);
10 }
```


Default erstellbare Lambdas & `decltype`

C++20

C++14 version:

```
1 auto compare = [](auto x, auto y) { return x > y; };
2 std::map<std::string, int, decltype(compare)> map{{"a", 1}, {"b", 2}};
3
4 for(const auto& [v, k] : map) {
5   printf("%s\n", v.c_str());
6 }
```

Aktuell nicht in Clang und C++ Insights verfügbar.

Default erstellbare Lambdas & `decltype`

C++20

```
1 std::map<std::string, int, decltype([](auto x, auto y) { return x > y; })> map{
2   {"a", 1},
3   {"b", 2}};
4
5 for(const auto& [v, k] : map) {
6   printf("%s\n", v.c_str());
7 }
```

Aktuell nicht in Clang und C++ Insights verfügbar.

Lambdas übermäßig angewendet

- Können wir Lambdas zu oft nutzen?

```
1 const bool isListInitialization{
2   [&]() { return stmt->getLParenLoc().isInvalid(); }()};
```

Lambdas übermäßig angewendet

- Können wir Lambdas zu oft nutzen?

```
1 const bool isListInitialization{stmt->getLParenLoc().isInvalid()};
```


}

Ich bin Fertig.

−15% für "Programmieren mit C++11 bis C++17" mit **ESE2019**

18. - 20.3.2020 Ludwigsburg

https://www.eventbrite.de/e/programmieren-mit-c11-bis-c17-tickets-72078874855

https://www.AndreasFertig.Info

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

Verwendete Compiler

- Compiler welche zum Übersetzen des (meisten) Codes verwendet wurden.
  - g++ (Homebrew GCC 9.2.0_1) 9.2.0
  - clang version 9.0.0 (https://github.com/llvm/llvm-project.git 0399d5a9682b3cef71c653373e38890c63c4c365)

Quellen

[1] Smith R., “Working Draft, Standard for Programming Language C++”, *N4800*, Mai 2019. http://wg21.link/n4800

[2] “cppreference: std::find, std::find_if, std::find_if_not”. https://en.cppreference.com/w/cpp/algorithm/find

[3] Polukhin A., “Add constexpr modifiers to functions in <algorithm> and <utility> headers”. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0202r3.html

Bilder:

2: Franziska Panter

33: Franziska Panter

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

Nächste Events

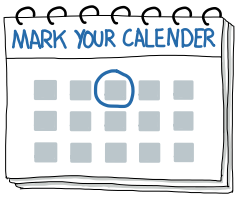
Talks

- *C++ Insights: How stuff works, Lambdas and more!*, MUC++, February 05, 2020
- *C++ Insights: How stuff works, Lambdas and more!*, OOP, February 06, 2020

Training Classes

- *Programmieren mit C++11 bis C++17*, Andreas Fertig, March 18, 2020

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



Über Andreas Fertig

Andreas ist freiberuflicher Trainer und Berater für C++ mit dem Spezialgebiet eingebettete Systeme. Bereits seit seinem Informatik Studium in Karlsruhe befasst er sich mit eingebetteten Systemen und den damit einher gehenden Anforderungen und Besonderheiten. Er arbeitete 10 Jahre für die Philips Medizin Systeme GmbH als C++ Softwareentwickler und Architekt mit dem Schwerpunkt eingebettete Systeme.
Andreas engagiert sich im C++ Standardisierungskommitee, speziell in SG14 die sich mit dem Bereich eingebettete Systeme befasst.
Zudem entwickelt er macOS Anwendungen und ist der Autor von cppinsights.io.