Lambdas – the good, the bad and the tricky…

Dawid Zalewski

11/18/2019

github.com/zaldawid
zaldawid@gmail.com
saxion.edu

Who is he?

- ~25 year on & off playing with computers
- + some microfluidics, thermodynamics, real-time, cryo-cooling, embedded, Bayesian methods, …
- C, C#, Python, C++, Java, …
- teaching (mostly) programming @ SAXION UNIVERSITY OF APPLIED SCIENCES

Outline

- **Lambdas 101**
- Evolution of lambdas
- Lambdas in C++20
- The tricky parts

Pop quiz #1

```
void func() {

  const auto n = 42;
  auto k = 5;

  auto l1 = [=]  (int a) { return k + a; };

  auto l2 = [] <typename T> (T&& a, T&& b) { return a + b; };

  auto l3 = [] (int a=2) { return n + a; };

  constexpr auto r = [] () {return n + 11; } ();

  auto l4 = [k=k] (auto a) { return k + a; };
}
```

✓ C++ 11

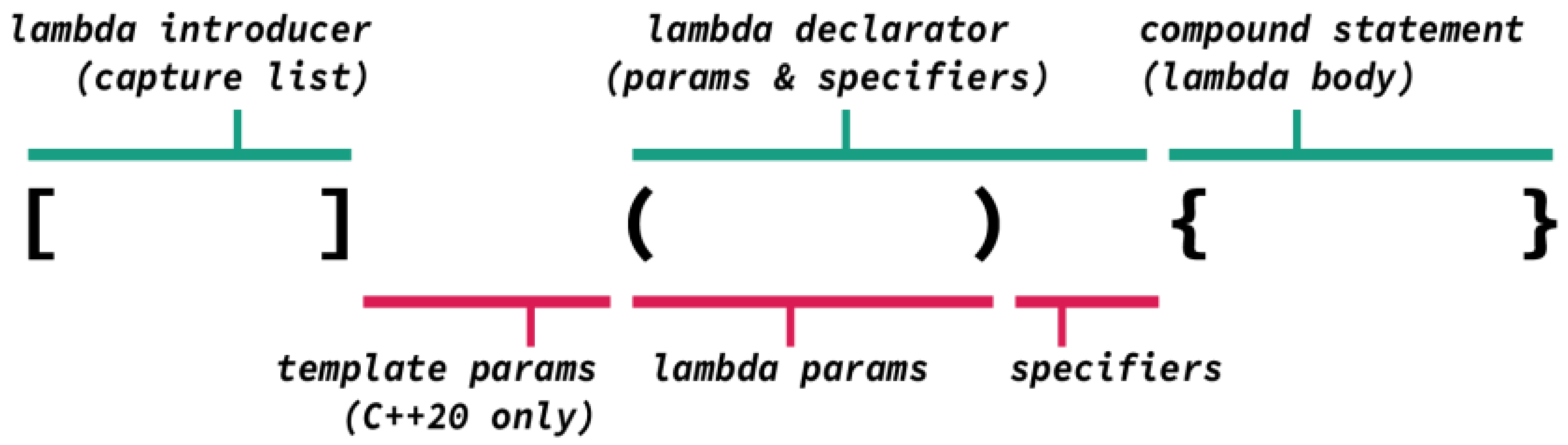
✓ C++ 20

✓ C++ 14

✓ C++ 17

✓ C++ 14

Lambdas' anatomy



Lambdas 101 (1): closures

Lambda expression

```
auto lambda = [](){};

lambda();
lambda.operator()();

void (*func) () = lambda;
func();
```

Closure type

```
class lambda_class{
public:

  void operator()() const {};

  using fp_t = void (*) ();
  operator fp_t() const {return call;}

private:
  static void call() {};

} lambda;
```

Lambdas 101 (2): captures

```
void func(){
  auto n = 42;
  auto k = 11;

  auto l1 = [=] () { return k + n; };

  auto l2 = [n] () { return n; };

  auto l3 = [&] () { return n; };

  auto l4 = [=, &k] () { return n + k; };

}
```

← captures n & k by copy (**implicit**)

← captures n by copy (**explicit**)

← captures n by reference (**implicit**)

← captures k by reference (**explicit**) & n by copy (**implicit**)

Lambdas 101 (3): closures

Lambda expression

```
auto n = 42;

auto lambda = [n](){
  return n;
};
```

Closure type

```
class lambda_class{
public:
  int operator()() const {
    return cap_n_;
  };

  using fp_t = void (*) ();
  operator fp_t() const {return call;}

private:
  static void call() {};
  int cap_n_;

} lambda(n);
```

(The lines `using fp_t = void (*) ();`, `operator fp_t() const {return call;}` and `static void call() {};` are struck through.)

This constructor is not really there.

Lambdas 101 (4): const vs. mutable

```
void func(){
  auto k = 42;

  auto l1 = [=] () {
    return k += 11;
  };



  auto l2 = [=] () mutable {
    return k += 11;
  };

}
```

```
struct l1_class{
  int operator()() const {
    return k += 11;
  }
private:
  int k=42;
} l1;

struct l2_class{
  int operator()() {
    return k += 11;
  }
private:
    int k=42;
} l2;
```

Lambdas 101 (4): closure's uniqness

Lambda expression

```
auto lambda0 = [](){};

auto lambda1 = [](){};

auto copy = lambda0;
```

Closure types

```
class lambda0_class{
public:
  void operator()() const {};
} lambda0;

class lambda1_class{
public:
  void operator()() const {};
} lambda1;

lambda0_class copy = lambda0;
```

Each lambda expression gives rise to its own closure type.

Outline

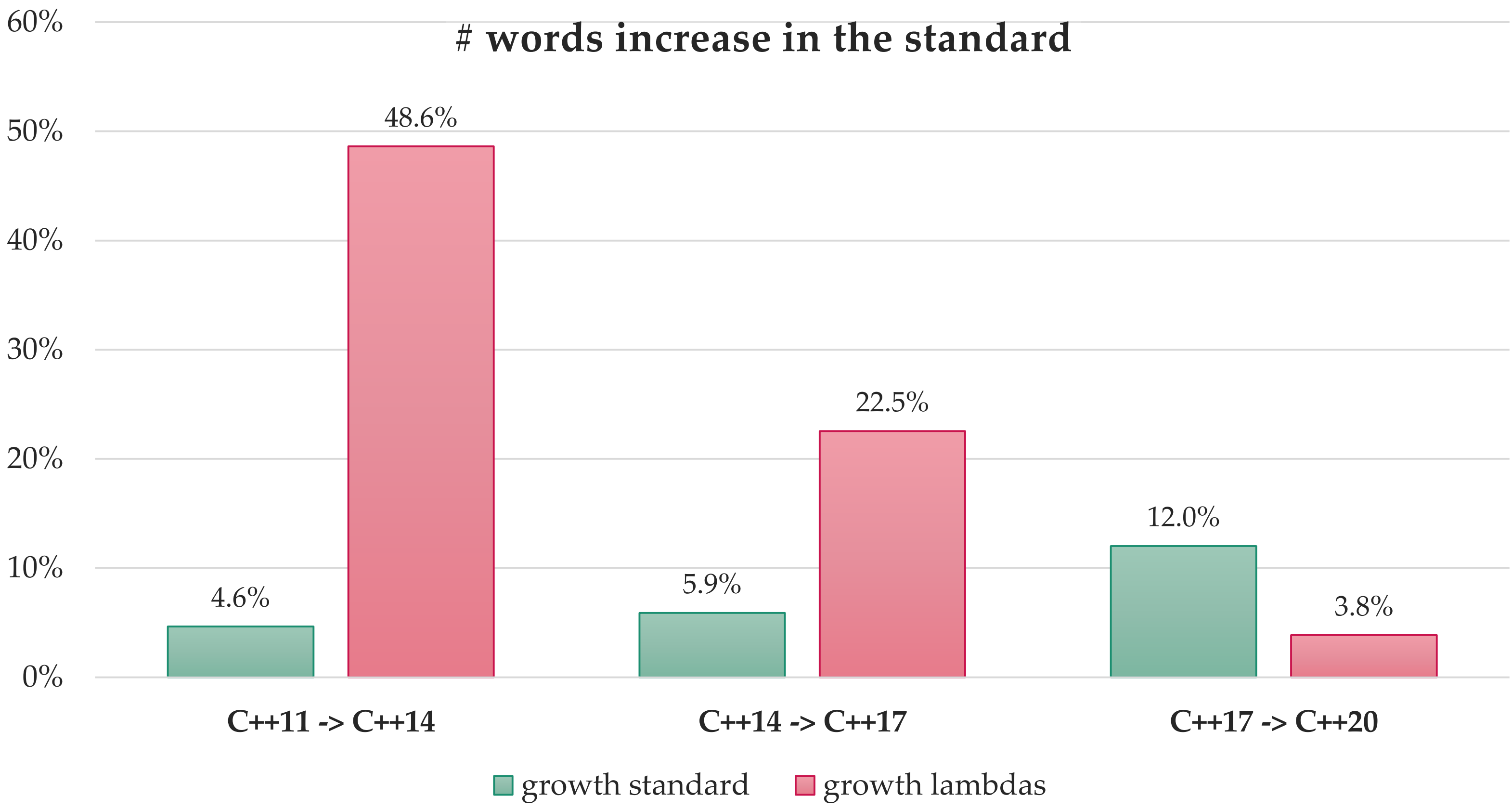
- Lambdas 101
- **Evolution of lambdas**
- Lambdas in C++20
- The tricky parts

How big are the lambdas

- What part of the standard (exclusive library) is about lambdas?

	C++11	C++14	C++17	C++20
% words of the standard	1.2%	1.7%	1.9%	1.8%
mentions outside [expr.prim.lambda]	~5	~15	~25	~50

How big are the lambdas?



Lambdas in C++11

Capture	Parameters	Specifiers	Quirks
none	`Type name`	`mutable`	• no default params
`&`		`noexcept`	• no generic types (templates)
`=`		`throw`	• no capture by **move**
`&var`			• so-so return type deduction
`var`			• no capture of enclosing object by copy
`&var…`			• no **constexpr**
`var…`			
`this`			

Comes later

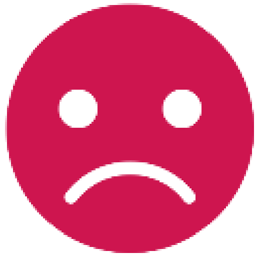
Lambda's in C++11

Capturing a copy of the enclosing object

```
struct A {
  void func(){
    auto lambda = [*this](){
      ...
  }
};
```

Capturing a move-only object

```
std::unique_ptr<int> num = ...;
auto lambda = [num]() {
  ...
};
```



Neither will work in C++11

Lambdas in C++14

	Capture	Parameters	Specifiers	Quirks

inherited	*none*
	`&`
	`=`
	`&var`
	`var`
	`&var…`
	`var…`
	`this`
		`Type name`	`mutable`
			`noexcept`
			`throw`
				• ~~no default params~~
				• ~~no generic types (templates)~~
				• ~~no capture by **move**~~
				• ~~so-so return type deduction~~
				• ~~no capture of enclosing object by copy~~
				• no **constexpr**

C++14	`&var=`*init*
	`var=`*init*
		`auto name`
		`auto…name`
		`Type name=`*def.*
		`auto name=`*def.*		

Lambdas in C++14

Lambda

Type0 Type1

```
auto lambda = [](auto a, auto b){
  return a + b;
};
```

Closure type

```
class lambda_class{
public:

  template <typename Type0, typename Type1>
  auto operator()(Type0 a, Type1 b) const{
    return a + b;
  }

};
```

Generic lambdas produce closures with a function call operator template.
One *invented* template type per **`auto`**.

Lambdas in C++14

Lambda

```
auto lambda = [](auto&&...a){
  return sum(
     std::forward<decltype(a)>(a)...);
};
```

Closure type

```
class lambda_class{
public:

  template <typename...Type0>
  auto operator()(Type1&&...a) const{
     return sum(
     std::forward<decltype(a)>(a)...);
  }

};
```

Works with forwarding references, pack expansion…

Lambdas in C++14: init captures

Capturing this

```
struct A {
  int a;
  void func(){

    auto lambda = [this](){
      a = 5;
    };
  }
};
```

a refers to the member of the enclosing A instance

Capturing a copy of *this

```
struct A {
  int a;
  void func(){

    auto lambda = [self=*this](){
      self.a = 5;
    };
  }
};
```

self is a copy of the enclosing A instance

Lambdas in C++14: init captures

Capturing a **move**–only object

```
std::unique_ptr<std::string> pstr = ...;

auto lambda = [ s = std::move(pstr) ](){
  std::cout << *s;
};
```

Closure

```
class lambda_class{

  std::unique_ptr<std::string> pstr_cap_;

  void operator()() const {…}

} lambda{std::move(pstr)};
```

Objects can be moved into the closures (or forwarded into them)

Lambdas in C++14: init captures

Capturing a **`const`** reference:

```
int num;
auto lambda = [ &n=static_cast<const int&>(num) ](){
  n = 42; //error: n is a const reference to num
};
```

Pro-tip: clang emits wrong diagnostics here:

```
cannot assign to a variable captured by copy in a non-mutable lambda
```

and so does msvc:

```
'n': a by copy capture cannot be modified in a non-mutable lambda
```

gcc does the right thing:

```
error: increment of read-only reference 'n'
```

Lambdas in C++17

	Capture	Parameters	Specifiers	Quirks
inherited	*none*	`Type name`	`mutable`	• ~~no *easy* capture of enclosing object by copy~~
	`&`	`auto name`	`noexcept`	• ~~no constexpr~~
	`=`	`auto…name`		• limited generic types
	`&var`	`Type name=`*def.*		• no init capture with pack expansion
	`var`	`auto name=`*def.*		• … and a few more
	`&var…`			
	`var…`			
	`this`			
	`&var=`*init*			
	`var=`*init*			
C++17	`*this`		`constexpr`	
			`(throw)`	

Lambdas in C++17: *this

more comes later

Capturing the enclosing object

```
struct A{
  int n;
  void func(){
    return [this](){
      return n;
    };
  }
};
A a{0};
auto lambda = a.func();
a.n = 42;
assert(lambda() == 42);
```

`n` belongs to the original `A` instance

Capturing a copy of the enclosing object

```
struct A{
  int n;
  void func(){
    return [*this](){
      return n;
    };
  }
};
A a{0};
auto lambda = a.func();
a.n = 42;
assert(lambda() == 0);
```

`n` belongs a copy of the original `A` instance

Lambdas in C++17: constexpr

Lambda

```
auto lambda = [](auto a){
  return a + a;
};

static_assert(10==lambda(5));
```

Closure type

```
class lambda_class{
public:
  template <typename Type0>
  auto constexpr operator()(Type0 a) const{
    return a + a;
  }
} lambda;
```

`constepxr` is implicit if the function call operator (template) satisfies the **`constexpr`** requirements [*dcl.constexpr*].

Lambdas in C++17: constexpr

Lambda

```
auto lambda = [](auto a){
  return a + a;
};

static_assert(10==lambda(5));

using fp_t = int(*)(int);
constexpr fp_t func = lambda;

static_assert(10==func());
```

Closure type

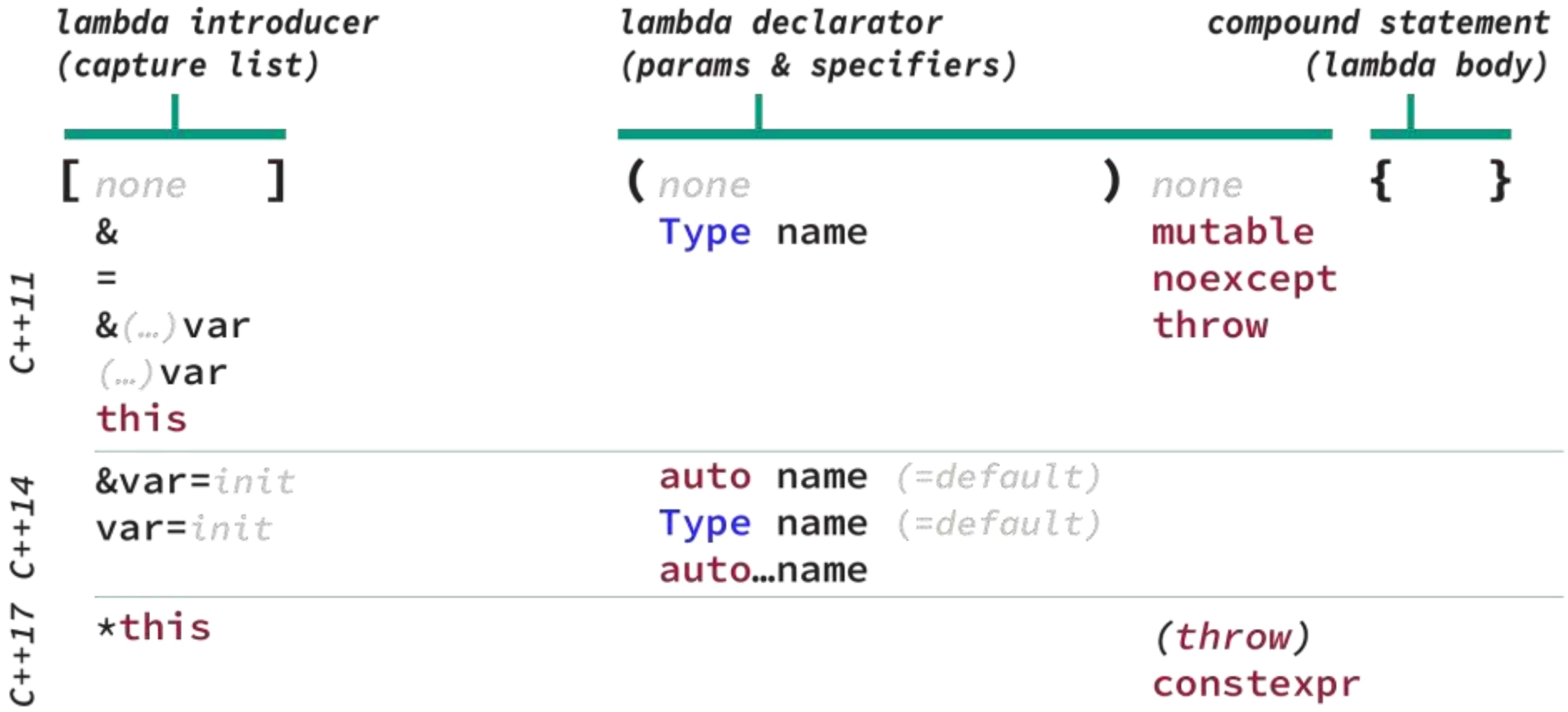
```
class lambda_class{
public:
  template <typename Type0>
  auto constexpr operator()(Type0 a) const{
    return a + a;
  }
} lambda;
```

constepxr is implicit if the function call operator (template) satisfies the **constexpr** requirements [*dcl.constexpr*].

Outline

- Lambdas 101
- Evolution of lambdas
- **Lambdas in C++20**
- The tricky parts

Lambdas before C++20



Lambdas before C++20

- Limited generic types (no `template <typename…>`) [p0428]
- Lambdas are not default-constructible [p0624]
- Lambdas cannot appear in unevaluated context [p0315]
- No pack expansion in init capture [p0780]
- No capturing of structured bindings [p1091]
- Weirdness around captures in member functions [p0806, p0409]
- No self-referencing lambdas [p0839]

Lambdas in C++20: templates

- Generic lambdas with implicit invented types are no fun:

```
auto push_one = [](auto& v){
    using T = typename std::remove_reference_t<decltype(v)>::value_type;
    value_factory<T> f;
    v.push_back( f.get() );
};

std::vector<double> v;
push_one(v);
```

Lambdas in C++20: templates

- Explicit templates remove the boilerplate code:

```
auto push_one = [] <typename T> (std::vector<T>& v){

    value_factory<T> f;
    v.push_back( f.get() );
};

std::vector<double> v;
push_one(v);
```

Lambdas in C++20: templates

- Explicit templates remove the boilerplate code:

```
auto push_one = [] <typename C, typename T=typename C::value_type> (C& v){

    value_factory<T> f;
    v.push_back( f.get() );
};

std::vector<double> v;
push_one(v);
```

Lambdas in C++20: templates

< C++20

```
auto sum = [] (auto a, auto b){
    return a + b;
};

sum(21, 21.0); // → 42.0
```

≥ C++20

```
auto  sum  = []<typename T>(T a, T b){
    return a + b;
};

sum(21, 21.0);
```

Look: same T!

```
error: no match for call to
'(::<lambda(T, T)>) (int, double)'
```

One template type can be used for multiple arguments.

Lambdas in C++20: templates++

```
template <typename T>
concept Integral = std::is_integral<T>::value;

auto sum = [] <Integral T> (T a, T b) { return a + b; };


auto sum = [] (T a, T b) requires Integral<T> { return a + b; };


sum(21.0, 21.0);
```

```
error: (…) candidate template ignored: constraints not satisfied
           with T = double] because 'double' does not satisfy 'Integral'
```

Lambdas in C++20: finally unevaluated

```
struct Process { int priority; };

using Container = std::vector<Process>;

struct ProcessOrdering {
  bool operator()(const Process& lhs, const Process& rhs) const {
    return lhs.priority > rhs.priority;
  }
};

std::priority_queue< Process, Container, ProcessOrdering > queue;
```

Lambdas in C++20: finally unevaluated

```
struct Process { int priority; };

using Container = std::vector<Process>;

using ProcessOrdering =

  decltype( [](auto&& lhs, auto&& rhs){ return lhs.priority > rhs.priority; } );
```

Only for capture-less lambdas!

```
std::priority_queue< Process, Container, ProcessOrdering > queue;
```

Lambdas in C++20: default constructible

```
template <typename T, typename Scaler>
struct Vector{
    T* data;
    std::size_t length;

    void scale(){
        auto ps = new Scaler();
        for (auto d = data; d < data + length; ++d )
            *d = ps->operator()(*d);
        delete ps;
    }
};
Vector<double, decltype([](const auto& p){return std::log(p); })> v;
v.scale();
```

Only for capture-less lambdas!

Detour: Lambdas special functions

	C++11	C++14	C++17	C++20 (with captures)	C++20 (no captures)
`lambda()`	`=delete`	`=delete`	none	`=delete`	**`=default`**
`~lambda()`	declared	declared	declared	declared	declared
`lambda(const&)`	declared	declared	`=default`	`=default`	`=default`
`operator=(const&)`	`=delete`	`=delete`	`=delete`	`=delete`	**`=default`**
`lambda(&&)`	*declared*	*declared*	`=default`	`=default`	`=default`
`operator=(&&)`	not declared	not declared	not declared	not declared	**`=default`**
`decltype([](){})`	no	no	no	no	yes

Revolution

Lambdas in C++20: structured bindings

```
auto tuple = std::make_tuple(42, "alice"s);
auto& [n, s] = tuple;

auto by_val = [=]() { return n == 42; };
auto by_ref = [&]() { s = "bob"; };
```

← Totally illegal!

A structured binding declaration introduces the identifiers (…) as names
[C++17, dcl.struct.bind]

A bit-field , a structured binding, (…) shall not be captured by reference
[C++20, expr.prim.lambda.capture]

Pro tip: GCC & MSVC will happily accept both even in the C++17 mode.

Lambdas in C++20: init capture pack expansion

```
template<class F, class... Args>
auto make_task(F&& f, Args&&... args) {

  return [f = std::forward<F>(f), args...]() mutable {

    return std::forward<F>(f)(std::forward<Args>(args)...);

  };
}

auto f = [](auto&& ... s) {((std::cout << std::forward<decltype(s)>(s)) ,...);};

auto task = make_task(f, std::string("bob"));
```

Unnecessary copy

task closure

Lambdas in C++20: init capture pack expansion

```
template<class F, class... Args>
auto make_task(F&& f, Args&&... args) {

  return [f = std::forward<F>(f), ...args=std::forward<Args>(args)]() mutable {

    return std::forward<F>(f)(std::forward<Args>(args)...);

  };
}

auto f = [](auto&& ... s) {((std::cout << std::forward<decltype(s)>(s)) ,...);};

auto task = make_task(f, std::string("bob"));
```

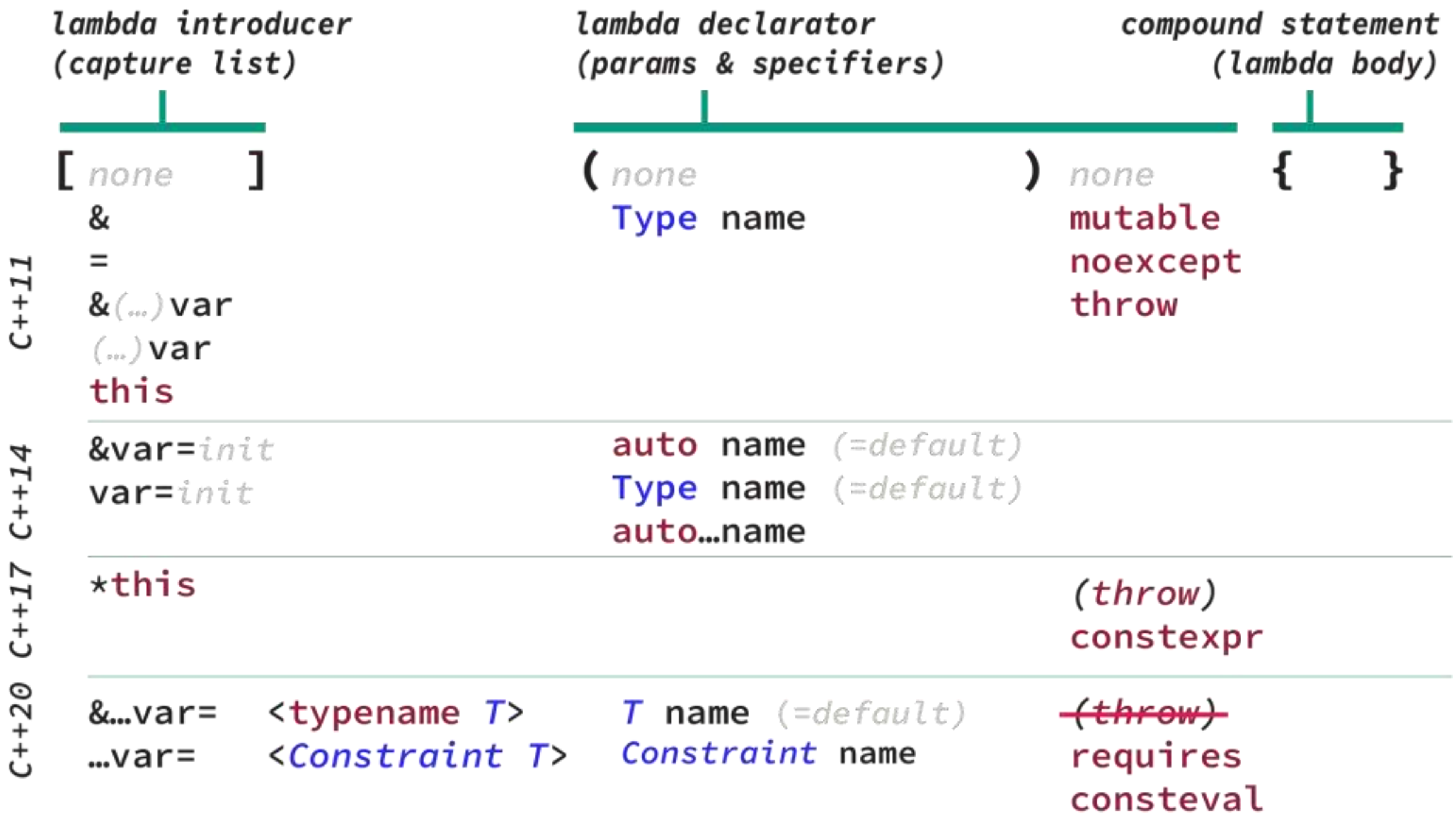
task closure

Init-captures with pack expansions help avoiding copies.

Outline

- Lambdas 101
- Evolution of lambdas
- Lambdas in C++20
- **The tricky parts**

Lambdas now


lambda introducer
(capture list)
lambda declarator
(params & specifiers)
compound statement
(lambda body)
[none]
(none)
none
{ }
C++11
&
=
&(...)var
(...)var
this
Type name
mutable
noexcept
throw
C++14
&var=init
var=init
auto name (=default)
Type name (=default)
auto...name
C++17
*this
(throw)
constexpr
C++20
&...var=
...var=
<typename T>
<Constraint T>
T name (=default)
Constraint name
(throw)
requires
consteval

Pop quiz #2 (captures)

```
struct A{
  int n;

  void func(){
    auto l1 = [=] () { return n; };
    auto l2 = [&] () { return n; };
    auto l3 = [&n] () { return n; };
    auto l4 = [n] () { return n; };
    auto l5 = [this] () { return n; };
    auto l6 = [n=n] () { return n; };
  }
};
```

- ← captures a reference to the A instance (*this)
- ← captures a reference to the A instance (*this)
- ← illegal – compilation error
- ← illegal – compilation error
- ← captures a reference to the A instance (*this)
- ← captures a copy of `this->n`

Lambda captures (I)

	Capture	Automatic Variables	Enclosing Object (*this)	Enclosing Object's Member Variables
	`&`	by reference	by reference	---
	`=`	copied	**by reference***	---
	`&var`	by reference	---	illegal
	`var`	copied	---	illegal
	`this`	---	by reference	---
C++17	`*this`	---	**copied**	---
	`&, this`	by reference	by reference	---
C++20	`=, this`	copied	by reference	---
C++17	`&, *this`	by reference	**copied**	---
C++17	`=, *this`	copied	**copied**	---

* *– deprecated in C++20*

Lambda captures (2)

```
struct A{
  int n = 0;

  void func(){
    auto m = 55;

    auto l1 = [&](){ ++n; };
    auto l2 = [=](){ ++n; };
    auto l3 = [&](){ ++m; };
    auto l4 = [=](){ ++m; };
    auto l5 = [this](){ ++n; };
    auto l6 = [*this](){ ++n; };
  }
};
```

← OK: captures a reference to the A instance (*this)
← OK: captures a reference to the A instance (*this)
← OK: captures a reference to the local m
← NOK: copies the local m (needs `mutable`)
← OK: captures a reference to the A instance (*this)
← NOK: copies the A instance (needs `mutable`)

Lambda captures (3)

Lambda within a class

```
struct A{
  int n;

  void func(){
    auto l1 = [&](){ return ++(this->n); };
  }

};
```

Magic this

Closure type

```
struct A{
  int n;
  void func(){
    struct l1_class{
      A*& this_;
      int operator()() const {
          return ++(this_>n);
      }
    } l1{this};
  }
};
```

this automagically refers to the enclosing object

Lambda captures (4)

Lambda within a class

```
struct A{
  int n;

  void func(){
    auto l2 = [*this]() mutable {
                  return ++(this->n);
                };
  }
};
```

Even more magic this

Closure type

```
struct A{
  int n;
  void func(){
    struct l2_class{
      A a;
      int operator()() {
        return ++(a.n);
      }
    } l2{*this};
  }
};
```

this automagically refers to a closure's copy of the enclosing object

Lambda captures (5)

Lambda within a class

```
struct A{
  int n;

  void func(){
    auto l3 = [&n=n](){ return ++n; };
  }

};
```

Closure type

```
struct A{
  int n;
  void func(){
    struct l3_class{
      int& n;
      int operator()() const {
        return ++n;
      }
    } l3{n};
  }
};
```

No automagical **this** (we don't capture it).

Lambda captures (6)

What	Examples	Capture?	How?

variables with automatic storage	local variables	always	`=`
			`&`
			`var`
			`&var`

variables with static storage	namespace scope & **`static`** variables	never	---

member variables	variables belonging to classes	always	**`this`**`, *`**`this`**
			`&var=var`
			`var=var`

constants	**`const`** & **`constexpr`** variables	not needed	---

Lambda captures (7)

```
int G = 55;
struct A{
  inline static int MS = 5;
  void func(){
    const int K = 42;
    static int S = 0;

    auto l1 = [=](){ ++G; };              ← OK: captures nothing
    auto l2 = [=](){ G = K; };            ← OK: captures nothing
    auto l3 = [&](){ ++S; MS = S; };      ← OK: captures nothing
    auto l4 = [](){ ++G; };               ← OK: works exactly like l1
    auto l5 = [](){ G = K; };             ← OK: works exactly like l2
    auto l6 = [](){ ++S; MS = S;};        ← OK: works exactly like l3
  }
};
```

Outline

- Lambdas 101
- Evolution of lambdas
- Lambdas in C++20
- The tricky parts
- **Secret part: *Fun with lambdas aka Lambdas applied***

Lambdas applied: initializers

- *Global* initialization:

```
static const auto faster = []{
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  return nullptr;
}();
```

Static variables are initialized before the program starts.

- Default function arguments:

```
void print_number(int number =
           [](auto n){ auto sum = n; while(n--) sum += n; return sum;}(42) ){
  std::cout << number;
}
```

Lambdas applied: initializers

- Logging initializer with a lambda:

```
auto init_with = [] < typename...Args>(Args&&...args){
    ((std::cout << args), ...);
    return (std::forward<Args>(args),...);
};

struct A {
  double number;
  A(double num):
    number( init_with("Initializing number with: ", num))
  {}
};
```

Lambdas applied: wrappers

- Let's say we have an idea:

```
[](auto& s){ std::cout << s; }  << "alice" << 42;
```

Lambdas applied: wrappers

- Let's say we have an idea:

```
streamer{[](auto& s){ std::cout << s; }} << "alice" << 42;

template <typename F>
struct streamer{
  F f;
  template <typename T>
  streamer& operator<<(const T& arg){
    f.operator()(arg);
    return *this;
  }
};
template <typename T> streamer(T) -> streamer<T>;
```

Lambdas applied: inheritance

- Inheriting from lambdas using aggregate initialization:

```
streamer{[](auto& s){ std::cout << s; }} << "alice" << 42;

template <typename F>
struct streamer: F{

  template <typename T>
  streamer& operator<<(const T& arg){
    F::operator()(arg);
    return *this;
  }
};
template <typename T> streamer(T) -> streamer<T>;
```

Aggregate initialization:
Each direct public base (F) is copy initialized from the corresponding clause (lambda) in the list.

Lamdas applied: multiple inheritance

- With pack expansion multiple inheritance is possible:

```
streamer{[](auto& s){std::cout << s;}, [](auto& s){log(s);}} << "alice" << 42;

template <typename...Fs>
struct streamer: Fs...{

  template <typename T>
  streamer& operator<<(const T& arg){
    (Fs::operator()(arg),...);
    return *this;
  }
};
template <typename...T> streamer(T...) -> streamer<T...>;
```

Lambdas applied: resource clean-up

- Using lambdas + wrappers to do resource clean-up

```
void func(){
  FILE* fp = std::fopen("test.txt", "r");
  WHEN_DONE([&](){ std::fclose(fp); });

  ...

  auto pstr = new std::string("alice");
  WHEN_DONE([&](){ delete pstr; });

  ...

}
```

Lambdas applied: resource clean-up

- With pack expansion multiple inheritance is possible:

```
template <typename...Ts>
struct when_done : Ts...{
  ~when_done() noexcept {
    (Ts::operator()(),...);
  }
};

template <typename...Ts> when_done(Ts...) -> when_done<Ts...>;

#define CONCAT_IMPL(x, y) x ## y
#define CONCAT(x, y) CONCAT_IMPL(x, y)
#define WHEN_DONE(...) auto CONCAT(wd__, __LINE__) = when_done{ __VA_ARGS__ }
```

C-macros, yack!

Lambdas applied: recursive lambdas (I)

- Lambdas do not support recursion directly:

Not valid C++ (unless using msvc):

```
auto sum = [](auto n) -> int { return n == 0? 0 : n + sum(n-1); };
```

Not valid C++ (unless using gcc):

```
auto sum = [](auto n) -> int { return n == 0? 0 : n + operator()(n-1); };
```

Not valid C++ (but see proposal P0839):

```
auto sum = [] sum_n(auto n) -> int { return n == 0? 0 : n + sum_n(n-1); };
```

Lambdas applied: recursive lambdas (2)

- Add another lever of indirection:

```
auto sum = [](auto n){

    auto sum_impl = [](auto& self, auto n){
        if (n == 0) return 0;
        return n + self(self, n - 1);
    };

    return sum_impl(sum_impl, n);

};

sum(42); //903
```

Lambdas applied: recursive lambdas (3)

- Use a higher-order function (Y-combinator):

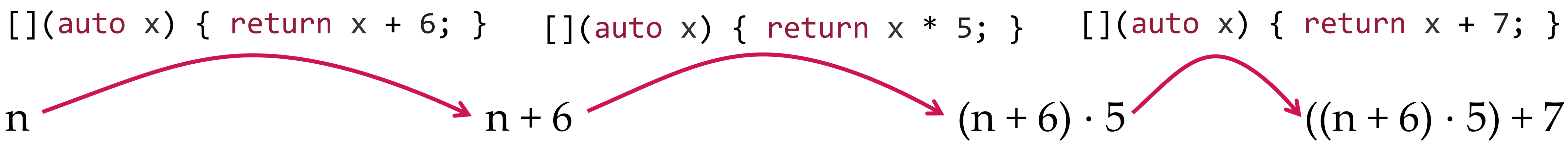
```
template<class F>
struct recurse : F{
  template <typename...Arg>
  auto operator()(Arg&&...arg) -> decltype(auto){
    return F::operator()(*this, std::forward<Arg>(arg)...);
  }
};
template <typename F> recurse(F) -> recurse<F>;

auto sum = recurse{[](auto& self, auto n)->int { return n == 0? 0 : n+self(n-1);}};
```

Lambdas ideas: function composition

- Chained application of lambdas (function composition)

```
auto fourty_two = compose{
                           [](auto x) { return x + 6; }, // -> 7
                           [](auto x) { return x * 5; }, // -> 35
                           [](auto x) { return x + 7; }  // -> 42
                          }(1);
```



Lambdas ideas: function composition

- Chained application of lambdas (function composition)

Recursive Functor Template Definition

```
template<class T, class... Ts>
struct compose: T, compose<Ts...>{
  template <typename...Args>
  decltype(auto) operator()(Args&&...args){
    return compose<Ts...>::operator()(
                            T::operator()(std::forward<Args>(args)...));
  }
};

template<class T>
struct compose<T> : T { using T::operator(); };

template <typename...Ts> compose(Ts...) -> compose<Ts...>;
```

Lambdas – the good,
the bad
and the tricky…

TIME FOR ANSWERS

Dawid Zalewski
github.com/zaldawid
zaldawid@gmail.com
saxion.edu