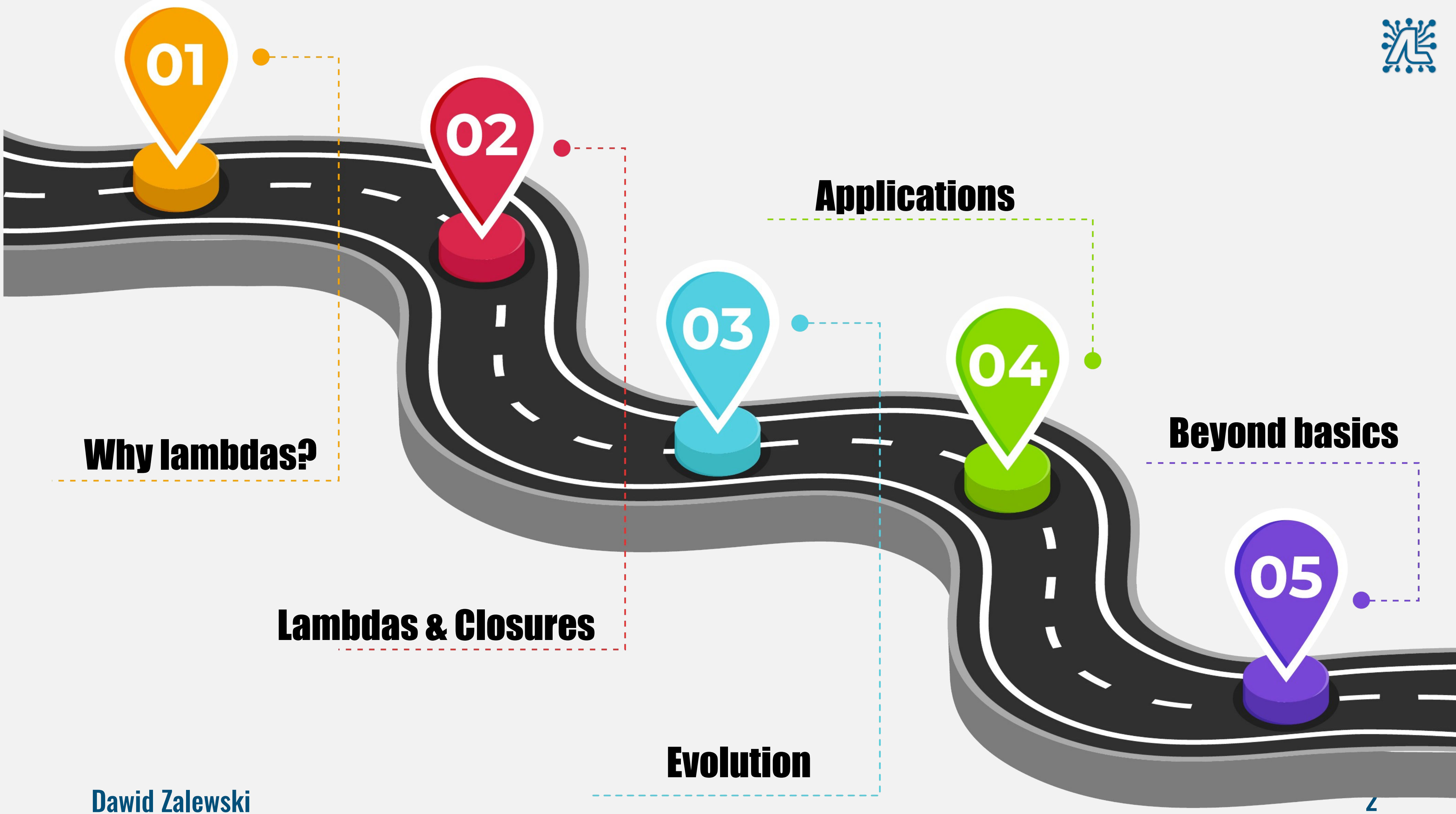
A Kaleidoscope of Lambdas

Dawid Zalewski

2024



Why lambdas?

Lambdas & Closures

Evolution

Applications

Beyond basics

A tale of colors

```
struct Color {
  double h;
  double s;
  double v;
};

std::vector<Color> only_saturated(const std::vector<Color>& v) {
  std::vector<Color> result{};
  for (auto const& c: v)
    if (c.s == 1.0)
      result.push_back(c);
  return result;
}
```

A tale of colors

```
struct FilterCriteria {
  double h_min, h_max;
  double s_min, s_max;
  double v_min, v_max;
};


std::vector<Color> filter_colors(std::vector<Color> const& v,
                                 FilterCriteria const& criteria) {

  std::vector<Color> result{};
  for (auto const& c: v)
    if ( c.h => criteria.h_min && c.h < criteria.h_max && ... )
      result.push_back(c);
  return result;
}
```

A tale of colors

```
using Predicate = bool(*)(Color const&);

bool is_saturated(Color const& c)
{
  return c.s == 1.0;
}

auto fully_saturated = filter_colors(my_colors, is_saturated);

std::vector<Color> filter_colors(std::vector<Color> const& v, Predicate predicate) {
  std::vector<Color> result{};
  for (auto const& c: v)
    if ( predicate(c) )
      result.push_back(c);
  return result;
}
```

A tale of colors

```cpp
struct Predicate {
  double s_min, s_max;
  bool operator()(Color const& c) const {
    return c.s >= s_min && c.s < s_max;
  }
};

auto only_saturated = filter_colors(my_colors,
                                    Predicate{.s_min=1.0, .s_max=2.0});

std::vector<Color> filter_colors(std::vector<Color> const& v, auto predicate) {
  std::vector<Color> result{};
  for (auto const& c: v)
    if ( predicate(c) )
      result.push_back(c);
  return result;
}
```

A tale of colors

```
auto only_saturated = filter_colors(my_colors,
                                    Predicate{.s_min=1.0, .s_max=2.0} );

std::vector<Color> filter_colors(std::vector<Color> const& v, auto const& predicate) {
  std::vector<Color> result{};
  for (auto const& c: v)
    if ( predicate(c) )
      result.push_back(c);
  return result;
}
```

A tale of colors

```
auto only_saturated = filter_colors(my_colors,
                                    [](auto const& color){ return color.s == 1.0; });

std::vector<Color> filter_colors(std::vector<Color> const& v, auto const& predicate) {
  std::vector<Color> result{};
  for (auto const& c: v)
    if ( predicate(c) )
      result.push_back(c);
  return result;
}
```

A tale of colors

```
auto only_saturated = filter_colors(my_colors,
                                    [](auto const& color){ return color.s == 1.0; });

std::vector<Color> filter_colors(std::vector<Color> const& v, auto const& predicate) {
  std::vector<Color> result{};
  std::copy_if(v.begin(), v.end(), std::back_inserter(result), predicate);
  return result;
}
```

Why lambdas?

```
auto only_saturated = my_colors
                     | std::views::filter([](auto const& color)
                                            {
                                              return color.s == 1.0;
                                            })
                     | std::ranges::to<std::vector>();
```

Why lambdas?

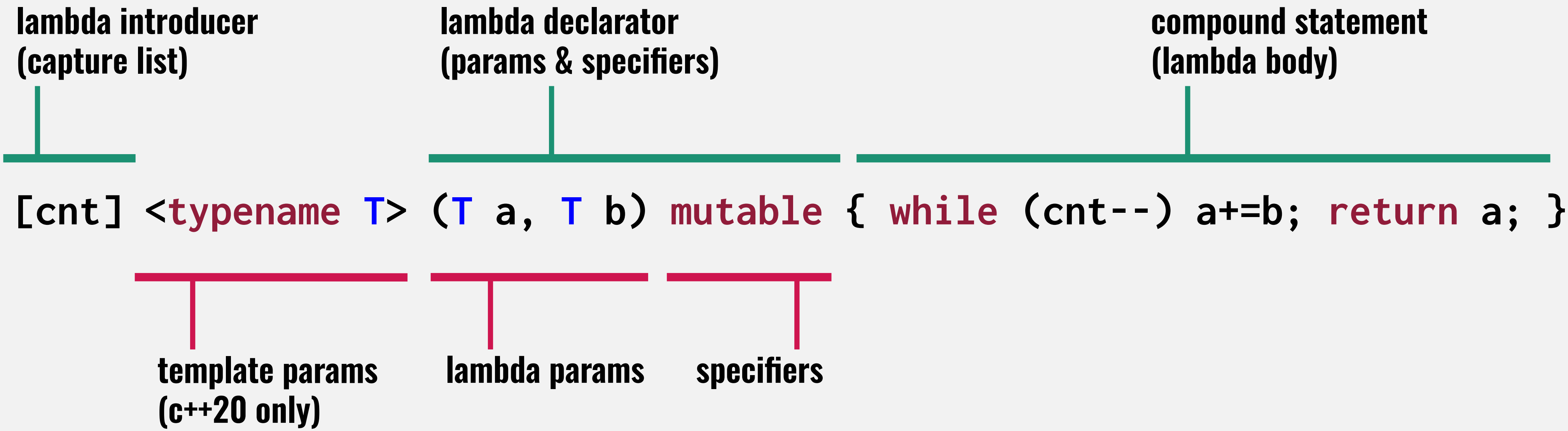
The primary aim is for lambda expressions to serve as "actions" for STL algorithms (...) and similar "callback" mechanisms.

Lambda expressions and closures for C++
wg21.link/n1968

What is a lambda

```
[](){}
 []{}
```

The Anatomy of Lambdas



lambda introducer
(capture list)

lambda declarator
(params & specifiers)

compound statement
(lambda body)

```
[cnt] <typename T> (T a, T b) mutable { while (cnt--) a+=b; return a; }
```

template params
(c++20 only)

lambda params

specifiers

Closures

Lambda expression

```
auto lmb = [](int n) {
  return n == 42;
};

lmb(23 + 19);
```

Closure type

```
class lmb_t {
public:

  inline constexpr bool
  operator()(int n) const {
    return n == 42;
  }

  constexpr lmb_t() = default;
};

auto lmb = lmb_t();
lmb.operator()(23 + 19);
```

Closures

Lambda expression

```
auto lmb = [](int n) {
  return n == 42;
};

lmb(23 + 19);

bool(*func)(int) = lmb;
func(27 + 15);
```

Closure type

```
class lmb_t {
public:
  constexpr bool operator()(int n) const;
  using FuncType = bool(*)(int);
  constexpr operator FuncType() const {
    return call_;
  }
  constexpr lmb_t() = default;
private:
  static constexpr bool call_(int n) {
    return lmb_t{}.operator()(n);
  }
};
```

Closures and captures

Lambda expression

```
auto N = 42;

auto lmb = [ ](int n) {
  return n == N; // Error
};

lmb(23 + 19);

bool(*func)(int) = lmb;
func(27 + 15);
```

Closure type

```
class lmb_t {
public:
  constexpr bool operator()(int n) const;
  using FuncType = bool(*)(int);
  constexpr operator FuncType() const {
    return call_;
  }
  constexpr lmb_t() = default;
private:
  static constexpr bool call_(int n) {
    return lmb_t{}.operator()(n);
  }
};
```

Closures and captures

Lambda expression

```
auto N = 42;

auto lmb = [N](int n) {
  return n == N;
};

lmb(23 + 19);

bool(*func)(int) = lmb;
func(27 + 15);
```

Closure type

```
class lmb_t {
public:
  constexpr bool operator()(int n) const;
  ~~using FuncType = bool(*)(int);~~
  ~~constexpr operator FuncType() const {~~
    ~~return call_;~~
  ~~}~~
  ~~constexpr lmb_t() = default;~~
private:
  ~~static constexpr bool call_(int n) {~~
    ~~return lmb_t{}.operator()(n);~~
  ~~}~~
};
```

Closures and captures

Lambda expression

```cpp
auto N = 42;

auto lmb = [N](int n) {
  return n == N;
};

lmb(23 + 19);
```

Closure type

```cpp
class lmb_t {
public:
  constexpr bool operator()(int n) const {
    return n == N;
  }

  constexpr lmb_t(int N_) : N{N_} {}

private:

  int N;

};
```

Closures and captures

Lambda expression

```
auto ap = AnswerProvider();

auto lmb = [ap](int n) {
  return n == ap;
};

struct AnswerProvider {
  int count_used = 0;
  operator int() {
    count_used++;
    return 42;
  }
};
```

Closure type

```
class lmb_t {
public:
  constexpr bool operator()(int n) const {
    return n == ap.operator int();
  }

  constexpr lmb_t(AnswerProvider ap_) :
            ap{ap_} {}
private:

  AnswerProvider ap;

};
```

Closures and captures

Lambda expression

```cpp
auto ap = AnswerProvider();

auto lmb = [ap](int n) {
  return n == ap;
};

struct AnswerProvider {
  int count_used = 0;
  operator int() {
    count_used++;
    return 42;
  }
};
```

Closure type

```
error: no match for 'operator=='
      |   return n == ap;

note: candidate: 'operator==(int, int)'
      |   return n == ap;

note: conversion of argument 2 would be
      ill-formed
error: passing 'const AnswerProvider' as
'this' argument discards qualifiers
```

Closures and mutable captures

Lambda expression

```
auto ap = AnswerProvider();

auto lmb = [ap](int n) {
  return n == ap;
};

struct AnswerProvider {
  int count_used = 0;
  operator int() {
    count_used++;
    return 42;
  }
};
```

Closure type

```
class lmb_t {
public:
  constexpr bool operator()(int n) const {
    return n == ap.operator int();
  }

  constexpr lmb_t(AnswerProvider ap_) :
            ap{ap_} {}
private:

  AnswerProvider ap;

};
```

Closures and mutable captures

Lambda expression

```
auto ap = AnswerProvider();

auto lmb = [ap](int n) mutable {
  return n == ap;
};

struct AnswerProvider {
  int count_used = 0;
  operator int() {
    count_used++;
    return 42;
  }
};
```

Closure type

```
class lmb_t {
public:
  constexpr bool operator()(int n) ~~const~~ {
    return n == ap.operator int();
  }

  constexpr lmb_t(AnswerProvider ap_) :
            ap{ap_} {}
private:

  AnswerProvider ap;

};
```

Closures and mutable captures

Lambda expression

```
auto ap = AnswerProvider();

auto lmb = [&ap](int n) {
  return n == ap;
};

struct AnswerProvider {
  int count_used = 0;
  operator int() {
    count_used++;
    return 42;
  }
};
```

Closure type

```
class lmb_t {
public:
  constexpr bool operator()(int n) const {
    return n == ap.operator int();
  }

  constexpr lmb_t(AnswerProvider ap_) :
            ap{ap_} {}
private:

  AnswerProvider& ap;

};
```

Closures are unique

Lambda expression

```cpp
auto lambda_1 = [](){};
auto lambda_2 = [](){};

static_assert(

  std::is_same_v<
                 decltype(lambda_1),
                 decltype(lambda_2)
                >

             );
```

Closure type

```cpp
class lambda_1_t {

  constexpr void operator()() { }

  constexpr lambda_1_t() = default;
};

class lambda_2_t {

  constexpr void operator()() { }

  constexpr lambda_2_t() = default;

};
```

Lambdas in C++11

Capture	Parameters	Specifiers	Quirks

`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

Lambdas in C++11

Capturing a copy of the enclosing object

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

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
<C++14	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.*		

C++ generalized captures

```
[         var=initializer](){}
[        &var=initializer](){}
```

C++ generalized captures

```
[auto  var=initializer](){}
[auto &var=initializer](){}
```

C++ generalized captures

```cpp
auto lmb = [answer=42](){
  std::cout << "The answer is" << answer;
};
```

```cpp
auto ans = "forty two"s;

auto lmb = [&ans_str=ans](){
  ans_str = "twenty four"; // OK
};
```

```cpp
auto ans = "forty two"s;

auto lmb = [&ans=std::as_const(ans)](){
  ans = "twenty four"; // ERROR
};
```

```cpp
auto str = "forty two";
auto ans =
         std::make_unique<std::string>(str);

auto lmb = [ans=std::move(ans)](){
  *ans = "twenty four";
};
```

C++14 capturing *this

```
struct AnswerValidator {

  int ans = 42;

  auto get(){
    return [obj_av=*this] (int check) {
      return obj_av.ans == check;
    };
  }
};

auto validator = AnswerValidator{}.get();

validator(24);
```

C++14 generic lambdas

Lambda expression

```
const auto N = 42;

auto lmb = [](auto a, auto b) {
  return (a + b) == N;
};

lmb(19, 23.4);
```

Closure type

Invented types

```
class lmb_t {
public:
  template <typename T1, typename T2>
  bool
  operator()(T1 a, T2 b) const {
    return (a + b) == N;
  }
};
```

C++14 generic lambdas

Lambda expression

```cpp
const auto N = 42;

auto lmb = [](auto a, decltype(a) b) {
  return (a + b) == N;
};

lmb(19, 23.4);
```

Closure type

```cpp
class lmb_t {
public:
  template <typename T1>
  bool
  operator()(T1 a, decltype(a) b) const {
    return (a + b) == N;
  }
};
```

C++14 parameter packs

```
auto add_to_vec = [](auto& vec, auto&&...items)
{
  (vec.push_back( std::forward<decltype(items)>(items) ), ...);
};

std::vector<std::string> names{};
std::string alice = "Alice";
std::string bob = "Bob";
std::string cindy = "Cindy";

add_to_vec(lines, alice, std::move(bob), cindy);
```

Lambdas in C++ 17

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

C++17, capturing *this

```cpp
struct AnswerProvider {
  int n;
  auto get() {
    return [this]() {
      return n;
    };
  }
};

AnswerProvider ap{42};
auto provider = ap.get();

ap.n = 24;
assert(provider() == 42);
```

```cpp
struct AnswerProvider {
  int n;
  auto get() {
    return [*this]() {
      return n;
    };
  }
};

AnswerProvider ap{42};
auto provider = ap.get();

ap.n = 24;
assert(provider() == 42);
```

C++17, finally constexpr

Lambda expression

```
const auto answer = 42;
auto is_answer = [](auto a, auto b){
  return a + b == answer;
};


static_assert(is_answer(19, 23));
```

Closure type

```
class is_answer_t {
public:
  template<typename T1, typename T2>
  constexpr bool
  operator()(T1 a, T2 b) const
  {
    return (a + b) == answer;
  }
};
```

constepxr is implicit if the function call operator (template) satisfies the **constexpr** requirements

Lambdas before C++ 20

	lambda introducer (capture list)	*lambda declarator (params & specifiers)*		*compound statement (lambda body)*
C++11	`[ none ]`	`( none`	`) none`	`{ }`
	`&`	`Type name`	`mutable`	
	`=`		`noexcept`	
	`&(…)var`		`throw`	
	`(…)var`			
	`this`			
C++14	`&var=init`	`auto name (=default)`		
	`var=init`	`Type name (=default)`		
		`auto…name`		
C++17	`*this`		`(throw)`	
			`constexpr`	

C++20 explicit template parameters

```
const auto N = 42;

auto is_answer = [] (auto a, decltype(a) b) {
  return (a + b) == N;
};

is_answer(19, 23.4);
```

C++20 explicit template parameters

```cpp
const auto N = 42;

auto is_answer = [] <typename Same> (Same a, Same b) {
  return (a + b) == N;
};

is_answer(19, 23.4);
```

```
error: no matching function for call to object of type 'is_lambda_t'
note: candidate template ignored: deduced conflicting types for parameter 'Same'
      ('int' vs. 'double')
```

C++20 explicit template parameters

```
const auto N = 42;

auto is_answer = [] < std::integral Same> (Same a, Same b) {
  return (a + b) == N;
};

is_answer(19.1, 23.4);
```

```
error: no matching function for call to object of type 'is_lambda_t'
note: candidate template ignored: constraints not satisfied
```

C++20 explicit template parameters

```
const auto N = 42;

auto is_answer = [] <typename Same>
                    requires std::integral<Same> (Same a, Same b) {
  return a + b == answer;
};

is_answer(19.1, 23.4);
```

```
error: no matching function for call to object of type 'is_lambda_t'
note: candidate template ignored: constraints not satisfied
```

C++20 explicit template parameters

```
template<typename F, typename...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 = [out=get_stream()](auto const& ... s) mutable { ((out << s) ,...); };

auto alice = "alice"s;
auto task = make_task(std::move(f), alice, "bob"s);

task();
```

Unnecessary copy

task closure

C++20 explicit template parameters

```cpp
template<typename F, typename...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 = [out=get_stream()](auto const& ... s) mutable { ((out << s) ,...); };

auto alice = "alice"s;
auto task = make_task(std::move(f), alice, "bob"s);

task();
```

task closure

C++20, the only thing missing

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

>> error: use of 'sum' before deduction of 'auto'

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

>> error: use of undeclared 'operator()'

std::function<int(int)> sum = [&](int n) { return n == 0? 0 : n + sum(n-1); };

>> but really...?
```

C++20, the only thing missing

```
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);
```

Lambdas now

lambda introducer (capture list)

lambda declarator (params & specifiers)

compound statement (lambda body)

	[captures]		(parameters)		->Ret { }
C++11	& = &var(…) var(…) this		Type name	mutable noexcept throw	
C++14	&var=init var=init		auto auto name (=default) Type name (=default) auto…name		
C++17	*this			(throw) constexpr	
C++20	&…var=pack …var=pack	<typename T> requires... <Concept T>	T name (=default) Concept name	~~(throw)~~ consteval	
C++23			this auto&& name this Type name	static	

trailing return type:
- Type
- auto
- decltype(auto)

C++20, the only thing missing

```
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);
```

C++23, explicit object parameter (also) for lambdas

```
auto sum =
                  [](this auto&& self, auto n){
    if (n == 0) return 0;
      return n + self(      n - 1);




};

sum(42);
```

C++23, explicit object parameter (also) for lambdas

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

```
sum(42);
```

Capturing

	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*

Capturing this & member variables

```
template <typename T>
struct CheckerMaker {

  AnswerProvider<T> ap{}; // convertible to T{42}

  auto get(T tolerance) const{
    auto diff = [&](auto candidate){ return candidate - ap; };
    return [&](auto candidate) { return diff(candidate) < tolerance
                                     && diff(candidate) > -tolerance; };
  };
};

CheckerMaker<double> cm{};
auto checker = cm.get(0.42);

std::cout << checker(42.1); // ERROR
```

Capturing this & member variables

```
template <typename T>
struct CheckerMaker {

  AnswerProvider<T> ap{}; // convertible to T{42}

  auto get(T tolerance) const{
    auto diff = [&](auto candidate){ return candidate - ap; };
    return [=](auto candidate) { return diff(candidate) < tolerance
                                     && diff(candidate) > -tolerance; };
  };
};

CheckerMaker<double> cm{};
auto checker = cm.get(0.42);

std::cout << checker(42.1); // FINE
```

Capturing this & member variables

```cpp
template <typename T>
struct CheckerMaker {

  AnswerProvider<T> ap{}; // convertible to T{42}

  auto get(T tolerance) const{
    auto diff = [&](auto candidate){ return candidate - ap; };
    return [=](auto candidate) { return diff(candidate) < tolerance
                                     && diff(candidate) > -tolerance; };
  };
};


auto checker = CheckerMaker<double>{}.get(0.42);

std::cout << checker(42.1); // ERROR AGAIN
```

Capturing this & member variables

```
template <typename T>
struct CheckerMaker {

  AnswerProvider<T> ap{}; // convertible to T{42}

  auto get(T tolerance) const{
    auto diff = [=](auto candidate){ return candidate - ap; };
    return [=](auto candidate) { return diff(candidate) < tolerance
                                     && diff(candidate) > -tolerance; };
  };
};


auto checker = CheckerMaker<double>{}.get(0.42);

std::cout << checker(42.1); // NOPE, STILL WRONG
```

Capturing this & member variables

```
template <typename T>
struct CheckerMaker {

  AnswerProvider<T> ap{}; // convertible to T{42}

  auto get(T tolerance) const{
    auto diff = [=, this](auto candidate){ return candidate - ap; };
    return [=](auto candidate) { return diff(candidate) < tolerance
                                     && diff(candidate) > -tolerance; };
  };
};


auto checker = CheckerMaker<double>{}.get(0.42);

std::cout << checker(42.1); // NOPE, STILL WRONG
```

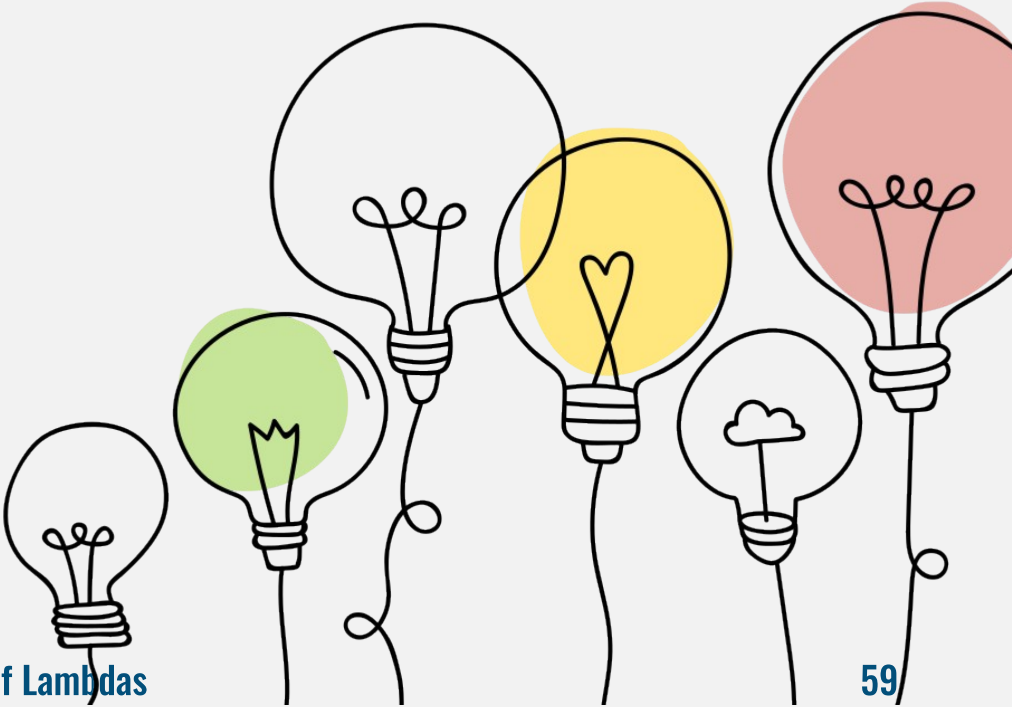
Capturing this & member variables

```
template <typename T>
struct CheckerMaker {

  AnswerProvider<T> ap{}; // convertible to T{42}

  auto get(T tolerance) const{
    auto diff = [=, *this](auto candidate){ return candidate - ap; };
    return [=](auto candidate) { return diff(candidate) < tolerance
                                     && diff(candidate) > -tolerance; };
  };
};


auto checker = CheckerMaker<double>{}.get(0.42);

std::cout << checker(42.1); // OK
```

Capturing this & member variables

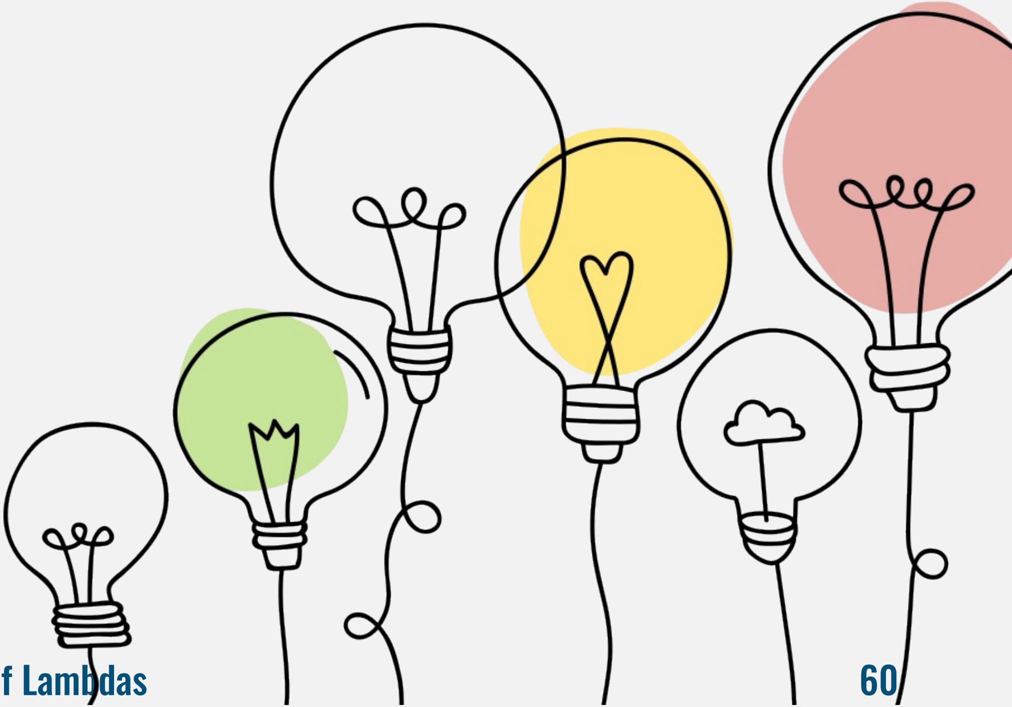
```cpp
template <typename T>
struct CheckerMaker {

  AnswerProvider<T> ap{}; // convertible to T{42}

  auto get(T tolerance) const{
    auto diff = [ap=ap](auto candidate){ return candidate - ap; };
    return [diff, tolerance](auto candidate) { return diff(candidate) < tolerance
                                                   && diff(candidate) > -tolerance; };
  };
};


auto checker = CheckerMaker<double>{}.get(0.42);

std::cout << checker(42.1);
```

The why of lambdas



The why of lambdas, global initialization

```
namespace {

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

}
```



The why of lambdas, global initialization

```cpp
#include <functional>

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

The why of lambdas, default arguments

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

The why of lambdas, constructors

```
struct Loader {

  Loader(AuthToken token, Uri const& uri):
    token_{ token },
    uri_{ uri },
    task_{}
  {}

  void start()
  {
    if (auto conn = Connection{token, uri})
    {
      task_ = Task{ conn };
    }
    else
    {
      throw ...;
    }
  }
};
```

The why of lambdas, constructors

```
struct Loader {

  Loader(AuthToken token, Uri const& uri):
    task_{ std::invoke([&] {
                             Connection conn{token, uri};
                             if (conn) return conn;
                             throw ...;
                           }
         )}
  {}

  void start()
  {
    ...
  }
};
```

The why of lambdas, initialization

```
struct Header{
  std::string id;
  size_t hash;
};

struct Record {
  Header head_;
  Record(Header head)
    : head_{ log_and_init<Header>("head_", std::move(head))} {}

  Record(std::string id, size_t hash)
    : head_{ log_and_init<Header>("head_", std::move(id), hash)} {}
};

Record rec{"data", 4224};
```

The why of lambdas, initialization

```
template <typename Res>
auto log_and_init = [] <typename...Args>(auto&& name, Args&&...args)
{








};
```

The why of lambdas, initialization

```
template <typename Res>
auto log_and_init = [] <typename...Args>(auto&& name, Args&&...args)
{
  auto comma = "";
  std::cout << "Initializing '" << name << "' with: {";

  std::cout << "}\n";

  return Res{
              std::forward<Args>(args)...
            };
};
```

The why of lambdas, initialization

```
template <typename Res>
auto log_and_init = [] <typename...Args>(auto&& name, Args&&...args)
{
  auto comma = "";
  std::cout << "Initializing '" << name << "' with: {";
  ((std::cout << (*comma ? comma : (comma=", ", "")) << args), ...);
  std::cout << "}\n";

  return Res{
              std::forward<Args>(args)...
            };
};
```

The why of lambdas, initialization

```
std::unique_ptr<clock_source> const clck{};

if (monotonic_clock_available){
  clck = std::make_unique<mono_clock>();
}
else if (time_source.is_stationary){
  clck = std::make_unique<mono_wrapper>( standard_clock{} );
}
else {
  clck = std::make_unique<non_stationary_wrapper>( standard_clock{} );
}
```

The why of lambdas, initialization

```
std::unique_ptr<clock_source> const clck{};

if (monotonic_clock_available){
  clck = std::make_unique<mono_clock>(); // WON'T
}
else if (time_source.is_stationary){
  clck = std::make_unique<mono_wrapper>( standard_clock{} ); // EVEN
}
else {
  clck = std::make_unique<non_stationary_wrapper>( standard_clock{} ); // COMPILE...
}
```

The why of lambdas, initialization

```
std::unique_ptr<clock_source> const clck{

  monotonic_clock_available ?
       std::make_unique<mono_clock>() :
       time_source.is_stationary ?
           std::make_unique<mono_wrapper>( standard_clock{} ) :
           std::make_unique<non_stationary_wrapper>( standard_clock{} )

};
```

The why of lambdas, initialization

```
std::unique_ptr<clock_source> const clck
{
  std::invoke(
    []{
        if (monotonic_clock_available) {
          return std::make_unique<mono_clock>();
        }
        else if (time_source.is_stationary){
          return std::make_unique<mono_wrapper>( standard_clock{} );
        }
        else {
          return std::make_unique<non_stationary_wrapper>( standard_clock{} );
        }
      }
  )
};
```

The why of lambdas, off with std::bind

```cpp
struct SerialChannel {
  std::expected<std::size_t, error_code> write(std::span<const std::byte> data);


};

template <typename Writer>
struct DebugProbe {
    Writer writer_;
    DebugProbe(Writer writer) : writer_{std::move(writer)} {}
};

SerialChannel channel{};
auto writer = std::bind(&SerialChannel::write, &channel, _1);
DebugProbe probe{std::move(writer)};
```

The why of lambdas, off with std::bind

```
struct SerialChannel {
  std::expected<std::size_t, error_code> write(std::span<const std::byte> data);
  std::expected<std::size_t, error_code> write(std::span<const std::byte> data,
                                               std::chrono::milliseconds timeout);
};

template <typename Writer>
struct DebugProbe {
    Writer writer_;
    DebugProbe(Writer writer) : writer_{std::move(writer)} {}
};

SerialChannel channel{};
auto writer = std::bind(&SerialChannel::write, &channel, _1);
DebugProbe probe{std::move(writer)};
```

The why of lambdas, off with std::bind

```cpp
struct SerialChannel {
  std::expected<std::size_t, error_code> write(std::span<const std::byte> data);
  std::expected<std::size_t, error_code> write(std::span<const std::byte> data,
                                               std::chrono::milliseconds timeout);
};

template <typename Writer>
struct DebugProbe {
    Writer writer_;
    DebugProbe(Writer writer) : writer_{std::move(writer)} {}
};

SerialChannel channel{};
auto writer = std::bind( static_cast<
       std::expected<std::size_t, error_code>(SerialChannel::*)(std::span<const std::byte>)>
       (&SerialChannel::write), &channel, _1);

DebugProbe probe{std::move(writer)};
```

The why of lambdas, off with std::bind

```
struct SerialChannel {
  std::expected<std::size_t, error_code> write(std::span<const std::byte> data);
  std::expected<std::size_t, error_code> write(std::span<const std::byte> data,
                                               std::chrono::milliseconds timeout);
};

template <typename Writer>
struct DebugProbe {
    Writer writer_;
    DebugProbe(Writer writer) : writer_{std::move(writer)} {}
};

SerialChannel channel{};
auto writer = [&channel](auto data) { return channel.write(data); };

DebugProbe probe{std::move(writer)};
```

The why of lambdas, off with std::bind

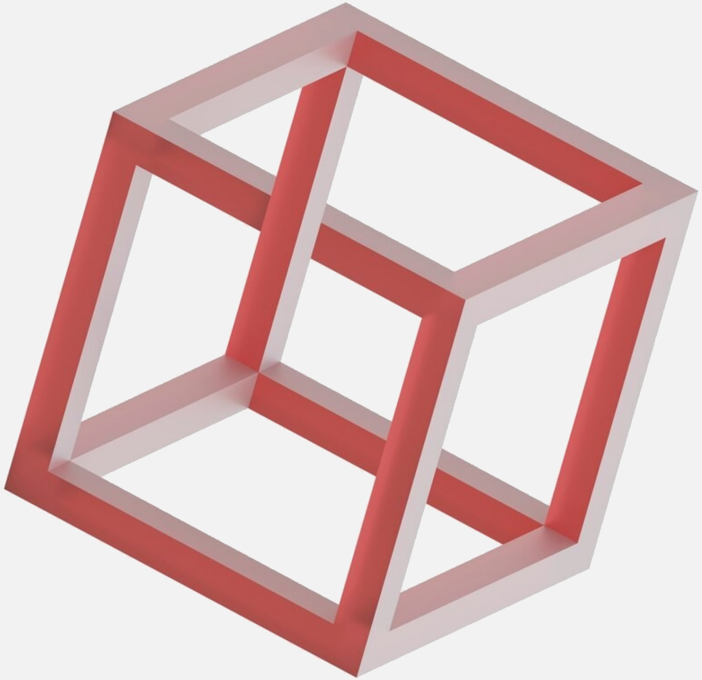
```
struct SerialChannel {

  std::expected<std::size_t, error_code> write(std::span<const std::byte> data,
                                               std::chrono::milliseconds timeout);
};

template <typename Writer>
struct DebugProbe {
    Writer writer_;
    DebugProbe(Writer writer) : writer_{std::move(writer)} {}
};

SerialChannel channel{};
auto writer = std::bind( &SerialChannel::write, &channel, _1, 2 * stats.latency_of(channel) );

DebugProbe probe{std::move(writer)};
```

The why of lambdas, off with std::bind

```cpp
struct SerialChannel {

  std::expected<std::size_t, error_code> write(std::span<const std::byte> data,
                                               std::chrono::milliseconds timeout);
};


SerialChannel channel{};
auto writer = [&channel, &stats](auto data)
              {
                return channel.write( data,
                                      2 * stats.latency_of(channel)
                                    );
              };

DebugProbe probe{std::move(writer)};
```

The why of lambdas, lambdas within lambdas

```
auto get_hello = []() {
  return []() {
    return "Hello, Lambdas!"s;
  };
};

auto hello = get_hello();

std::cout << hello();
```



The why of lambdas, lambdas within lambdas

```
auto validated = []() {
  return []() {

  };
};

using SV = std::string_view;
auto pswd_validator = validated<std::string>(
  [](SV sv){ return sv.size() >= 10; },
  [](SV sv){ return sv.find_first_of("01234567890") != std::string_view::npos; },
  [](SV sv){ return sv.find_first_of("!@#$%^&*()_-+=") != std::string_view::npos; });

std::cout << pswd_validator("abcdefghj$1").value_or("Password error");
```

The why of lambdas, lambdas within lambdas

```cpp
template <typename Ret>
auto validated = []() {
  return []() {

  };
};

using SV = std::string_view;
auto pswd_validator = validated<std::string>(
  [](SV sv){ return sv.size() >= 10; },
  [](SV sv){ return sv.find_first_of("01234567890") != std::string_view::npos; },
  [](SV sv){ return sv.find_first_of("!@#$%^&*()_-+=") != std::string_view::npos; });

std::cout << pswd_validator("abcdefghj$1").value_or("Password error");
```

The why of lambdas, lambdas within lambdas

```cpp
template <typename Ret>
auto validated = []<typename...Vals>(Vals&&...vals) {
  return []() {

  };
};

using SV = std::string_view;
auto pswd_validator = validated<std::string>(
  [](SV sv){ return sv.size() >= 10; },
  [](SV sv){ return sv.find_first_of("01234567890") != std::string_view::npos; },
  [](SV sv){ return sv.find_first_of("!@#$%^&*()_-+=") != std::string_view::npos; });

std::cout << pswd_validator("abcdefghj$1").value_or("Password error");
```

The why of lambdas, lambdas within lambdas

```
template <typename Ret>
auto validated = []<typename...Vals>(Vals&&...vals) {
  return [...validators=std::forward<Vals>(vals)]() {

  };
};

using SV = std::string_view;
auto pswd_validator = validated<std::string>(
  [](SV sv){ return sv.size() >= 10; },
  [](SV sv){ return sv.find_first_of("01234567890") != std::string_view::npos; },
  [](SV sv){ return sv.find_first_of("!@#$%^&*()_-+=") != std::string_view::npos; });

std::cout << pswd_validator("abcdefghj$1").value_or("Password error");
```

The why of lambdas, lambdas within lambdas

```
template <typename Ret>
auto validated = []<typename...Vals>(Vals&&...vals) {
  return [...validators=std::forward<Vals>(vals)]<typename...Args>(Args&&...args) {

  };
};

using SV = std::string_view;
auto pswd_validator = validated<std::string>(
  [](SV sv){ return sv.size() >= 10; },
  [](SV sv){ return sv.find_first_of("01234567890") != std::string_view::npos; },
  [](SV sv){ return sv.find_first_of("!@#$%^&*()_-+=") != std::string_view::npos; });

std::cout << pswd_validator("abcdefghj$1").value_or("Password error");
```

The why of lambdas, lambdas within lambdas

```
template <typename Ret>
auto validated = []<typename...Vals>(Vals&&...vals) {
  return [...validators=std::forward<Vals>(vals)]<typename...Args>(Args&&...args) {







  };
};

std::cout << pswd_validator("abcdefghj$1").value_or("Password error");
```

The why of lambdas, lambdas within lambdas

```cpp
template <typename Ret>
auto validated = []<typename...Vals>(Val&&...vals) {
  return [...validators=std::forward<Vals>(vals)]<typename...Args>(Args&&...args)
                                                           -> std::optional<Ret> {
    if ((validators(args...) && ...)) {
      return std::optional{Ret{std::forward<Args>(args)...}};
    }
    else {
      return std::nullopt;
    }
  };
};

std::cout << pswd_validator("abcdefghj$1").value_or("Password error");
```

Thank you!