Everything you (n)ever wanted to know about C++'s Lambdas

Sharing Good Practice

Dr. Nis Meinert

DLR / KN NAS NZ

DLR Deutsches Zentrum für Luft- und Raumfahrt

Before we start …

- If I manage to get to my last slide **you** made a terrible job asking questions!
- Please, **ask questions** at **any time**!

(slides & exercises: `github.com/avitase/cpp_lambda_talk`)

What is a Lambda Expression in C++?

`cube` is a lambda …

```cpp
int main() {
    auto cube = [](int x) { return x * x * x; };
    return cube(3);
}
```

godbolt.org/z/xWqMss

`is_even` is a lambda …

```cpp
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> xs{1, 2, 3, 4, 5, 6, 7};
    auto is_even = [](int x) { return x % 2 == 0; };
    return std::count_if(xs.begin(), xs.end(), is_even);
}
```

godbolt.org/z/bn5jYT

C++'s Lambda Expression

The simplest (and most boring) lambda

```cpp
auto x = []{};
```

…no capturing, takes no parameters and returns nothing

A slightly more "useful" lambda

```cpp
int main() {
    auto x = [] { return 5; };
    return x();
}
```

godbolt.org/z/1T8zYE

…is equivalent to*

```cpp
int main() {
    struct {
        auto operator()() const {
            return 5;
        }
    } x;
    return x();
}
```

godbolt.org/z/foj6Kq

* check Compiler Explorer links (note `-O0`!)

C++'s Lambda Expression

An even more “useful” lambda

```
int main() {
    auto x = [](int y) { return y; };
    return x(42);
}
```

godbolt.org/z/963TMh

…is equivalent to*

```
int main() {
    struct {
        auto operator()(int y) const {
            return y;
        }
    } x;
    return x(42);
}
```

godbolt.org/z/xfcYM7

Lambda expressions are **pure syntactic sugar** and are equivalent to `struct`s with an appropriate `operator()()` overload!

* check Compiler Explorer links (note `-O0`!)

Q: What is the output of the program?

```python
#!/usr/bin/env python3

if __name__ == '__main__':
    f = {k: lambda x: x + k for k in range(3)}

    for k in range(3):
        print(f[k](2), end='')
```

A: 444

```python
#!/usr/bin/env python3

if __name__ == '__main__':
    f = {k: lambda x: x + k for k in range(3)}

    for k in range(3):
        print(f[k](2), end='')
```

Why? implicit capture by reference

Fix

```python
#!/usr/bin/env python3

from functools import partial

if __name__ == '__main__':
    f = {k: partial(lambda x, k: x + k, k=k) for k in range(3)}
    # f = {k: lambda x, k=k: x + k for k in range(3)}
    # ... would change API

    for k in range(3):
        print(f[k](2), end='')
```

Now prints: 234

Q: What is the output of the program?

```
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    for (int k = 0; k < 3; k++) {
        f.emplace(k, [](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/Ga1rTj

Q: What is the output of the program?

```
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    for (int k = 0; k < 3; k++) {
        f.emplace(k, [](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/Ga1rTj

error: “k” is not captured

Q: What is the output of the program?

```
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    for (int k = 0; k < 3; k++) {
        f.emplace(k, [k](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/59KGYs

A: 234

```cpp
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    for (int k = 0; k < 3; k++) {
        f.emplace(k, [k](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/59KGYs

Q: What is the output of the program?

```
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    int k = 0;
    for (; k < 3; k++) {
        f.emplace(k, [&k](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/hzTPK3

```cpp
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    int k = 0;
    for (; k < 3; k++) {
        f.emplace(k, [&k](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/hzTPK3

C++'s Lambda Expression

Capturing rules

- `[x]`: captures `x` by value
- `[&x]`: captures `x` by reference
- `[=]`: captures all variables (used in the lambda) by value
- `[&]`: captures all variables (used in the lambda) by reference
- `[=, &x]`: captures variables like with `[=]`, but `x` by reference
- `[&, x]`: captures variables like with `[&]`, but `x` by value

Capturing by value

```cpp
int main() {
    int i = 1;
    auto z = [i](int y) {
        return i + y;
    }(3);
    return z;
}
```

godbolt.org/z/3M3cne

…or equivalently

```cpp
class X {
private:
    int i;

public:
    X(int i): i(i) {}

    int operator()(int y) const {
        return i + y;
    }
};

// potentially lots of lines of code

int main() {
    int i = 1;
    auto z = X{i}(3);
    return z;
}
```

godbolt.org/z/oPq3xd

Capturing by reference

```cpp
int main() {
    int i = 1;
    auto z = [&i](int y) {
        return i + y;
    }(3);
    return z;
}
```

godbolt.org/z/8qrx6b

…or equivalently

```cpp
class X {
private:
    int& i;

public:
    X(int& i): i(i) {}

    int operator()(int y) /*const*/ {
        return i + y;
    }
};

// potentially lots of lines of code

int main() {
    int i = 1;
    auto z = X{i}(3);
    return z;
}
```

godbolt.org/z/EP1o1z

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    int i = 1;
    auto x = [i]() { return ++i; };
    std::cout << i << x() << i;
}
```

godbolt.org/z/z16fG9

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    int i = 1;
    auto x = [i]() { return ++i; };
    std::cout << i << x() << i;
}
```

godbolt.org/z/z16fG9

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

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    int i = 1;
    auto x = [i]() mutable { return ++i; };
    std::cout << i << x() << i;
}
```

godbolt.org/z/M8Pjjb

A: 121

```cpp
#include <iostream>

int main() {
    int i = 1;
    auto x = [i]() mutable { return ++i; };
    std::cout << i << x() << i;
}
```

godbolt.org/z/M8Pjjb

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    int i = 1;
    auto x = [&i]() mutable { return ++i; };
    std::cout << i << x() << i;
}
```

godbolt.org/z/shheYa

A: 122

```cpp
#include <iostream>

int main() {
    int i = 1;
    auto x = [&i]() mutable { return ++i; };
    std::cout << i << x() << i;
}
```

godbolt.org/z/shheYa

Q: What is the output of the program?

```
#include <iostream>

int main() {
    auto x = [i=0]() mutable { return ++i; };
    std::cout << x() << x();
}
```

godbolt.org/z/MYaq3s

A: 12

```cpp
#include <iostream>

int main() {
    auto x = [i=0]() mutable { return ++i; };
    std::cout << x() << x();
}
```

godbolt.org/z/MYaq3s

Q: What is the output of the program?

```cpp
#include <iostream>
#include <utility>

int main() {
    auto x = [i=0, j=1]() mutable {
        i = std::exchange(j, j + i);
        return i;
    };

    for (int i = 0; i < 5; ++i) {
        std::cout << x();
    }
}
```

`godbolt.org/z/e95d1x`

`cppreference.com/w/cpp/utility/exchange`

or

Ben Deane, *std::exchange Patterns: Fast, Safe, Expressive, and Probably Underused* on `fluentcpp.com`

A: 11235

```cpp
#include <iostream>
#include <utility>

int main() {
    auto x = [i=0, j=1]() mutable {
        i = std::exchange(j, j + i);
        return i;
    };

    for (int i = 0; i < 5; ++i) {
        std::cout << x();
    }
}
```

`godbolt.org/z/e95d1x`

`cppreference.com/w/cpp/utility/exchange`

or

Ben Deane, *std::exchange Patterns: Fast, Safe, Expressive, and Probably Underused* on `fluentcpp.com`

C++'s Lambda Expression

Remember, lambda expressions are pure syntactic sugar and are equivalent to `struct`s with an appropriate `operator()()` overload …

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    auto x = [] { return 1; };
    auto y = x;
    std::cout << x() << y();
}
```

godbolt.org/z/jsTKEc

A: 11

```cpp
#include <iostream>

int main() {
    auto x = [] { return 1; };
    auto y = x;
    std::cout << x() << y();
}
```

godbolt.org/z/jsTKEc

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    int i = 1;
    int j = 2;
    auto x = [&i, j] { return i + j; };
    i = 4;
    j = 6;
    auto y = x;
    std::cout << x() << y();
}
```

godbolt.org/z/dhTv5o

A: 66

```cpp
#include <iostream>

int main() {
    int i = 1;
    int j = 2;
    auto x = [&i, j] { return i + j; };
    i = 4;
    j = 6;
    auto y = x;
    std::cout << x() << y();
}
```

godbolt.org/z/dhTv5o

Q: What is the output of the program?

```
#include <iostream>
#include <memory>

int main() {
    auto x = [i=std::make_unique<int>(1)] { return *i; };
    auto y = x;
    std::cout << x () << y();
}
```

godbolt.org/z/5ja7ze

Q: What is the output of the program?

```cpp
#include <iostream>
#include <memory>

int main() {
    auto x = [i=std::make_unique<int>(1)] { return *i; };
    auto y = x;
    std::cout << x () << y();
}
```

godbolt.org/z/5ja7ze

error: call to implicitly-deleted copy ctor

Stateful Lambdas

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    auto x = [i=0]() mutable { return ++i; };
    auto y = x;
    x();
    x();
    y();
    y();
    std::cout << x();
}
```

godbolt.org/z/8xfGfz

A: 3

```cpp
#include <iostream>

int main() {
    auto x = [i=0]() mutable { return ++i; };
    auto y = x;
    x();
    x();
    y();
    y();
    std::cout << x();
}
```

godbolt.org/z/8xfGfz

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    auto x = [] { static int i = 0; return ++i; };
    auto y = x;
    x();
    x();
    y();
    y();
    std::cout << x();
}
```

godbolt.org/z/YGEfqq

A: 5

```cpp
#include <iostream>

int main() {
    auto x = [] { static int i = 0; return ++i; };
    auto y = x;
    x();
    x();
    y();
    y();
    std::cout << x();
}
```

godbolt.org/z/YGEfqq

(* undefined in a threaded context, since `static` is not thread-safe!)

Fibonacci (again):

```cpp
#include <utility>

int main() {
    auto fib = [i=0, j=1]() mutable {
        struct Result {
            int &i, &j;

            auto next() {
                i = std::exchange(j, j + i);
                return *this;
            }
        };
        return Result{.i=i, .j=j}.next();
    };

    fib().next().next().next(); // mutate state
    return fib().i;
}
```

godbolt.org/z/nx8vzq

Stateful Lambdas

Let us now try to interact with the state of the Lambda …

```
#include <utility>

int main() {
    auto fib = [i=0, j=1]() mutable {
        struct Result {
            int &i, &j;

            auto next() {
                i = std::exchange(j, j + i);
                return *this;
            }
        };
        return Result{.i=i, .j=j}.next();
    };

    auto r = fib();
    r.i = 2; // mutate state
    r.j = 3; // mutate state
    return fib().j; // 5
}
```

godbolt.org/z/44nM7Y

Stateful Lambdas

…or slightly more conveniently:

```cpp
#include <utility>

int main() {
    auto fib = [i=0, j=1]() mutable {
        struct Result {
            int &i, &j;

            auto next(int n = 1) {
                while (n-- > 0) {
                    i = std::exchange(j, j + i);
                }
                return *this;
            }
        };
        return Result{.i=i, .j=j}.next();
    };

    return fib().next(3).j; // 5
}
```

godbolt.org/z/K93vT4

Stateful Lambdas

```cpp
#include <utility>

int main() {
    auto fib = [i=0, j=1]() mutable {
        struct Result {
            int &i, &j;

            auto next(int n = 1) {
                while (n-- > 0) {
                    i = std::exchange(j, j + i);
                }
                return *this;
            }
        };
        return Result{.i=i, .j=j}.next();
    };

    return fib().next(10).j; // 144
}
```

godbolt.org/z/ePEMEe

```
# g92 -O3
   | main:
19|    mov eax, 144
19|    ret
```

godbolt.org/z/ePEMEe

Best Practices

(partially taken from “Effective Modern C++” by Scott Meyers)

Use Lambdas in STL algorithm

```cpp
#include <algorithm>
#include <vector>

std::vector<int> get_ints();

int main() {
    auto ints = get_ints();
    auto in_range = [](int x) { return x > 0 && x < 10; };
    return *std::find_if(ints.begin(), ints.end(), in_range);
}
```

godbolt.org/z/98ff4e

Use Lambdas in STL algorithm

```cpp
#include <algorithm>
#include <vector>

std::vector<int> get_ints();

int main() {
    auto ints = get_ints();
    return *std::find_if(ints.begin(), ints.end(),
                         [](int x) { return x > 0 && x < 10; });
}
```

godbolt.org/z/z6384d

Stop pollution of namespace with helper variables

```cpp
#include <cmath>
#include <iostream>

int main() {
    auto y = []<typename T>(T x) {
        T mean = 1.;
        T width = 3.;
        auto norm = 1. / std::sqrt(2. * M_PI);
        auto arg = (x - mean) / width;
        return norm * std::exp(-.5 * arg * arg);
    }(.5);

    std::cout << y;
}
```

godbolt.org/z/3jKdM7

Allow variables to be `const`

```cpp
#include <vector>

std::vector<int> get_ints();

int main() {
    auto ints = get_ints();
    const auto sum = [&ints] {
        int acc = 0;
        for (auto& x: ints) acc += x;
        return acc;
    }();

    return sum;
}
```

`godbolt.org/z/oTW3r6`

(cf. ***IIFE***: `bfilipek.com/2016/11/iife-for-complex-initialization.html`
or
`foonathan.net/2020/10/iife-metaprogramming/`)

Avoid default capture modes

Below, there is a dangling pointer lurking in the wings …

```
void add_filter() {
    auto divisor = get_magic_number();
    filters.emplace_back([&](int x) { return x % divisor == 0; });
}
```

This error becomes more obvious, when explicit capturing is used:

```
void add_filter() {
    auto divisor = get_magic_number();
    filters.emplace_back([&divisor](int x) { return x % divisor == 0; });
}
```

Avoid default capture modes

Mitigation of copy & paste bugs:

```
auto divisor = get_magic_number();
std::find_if(container.begin(),
             container.end(),
             [&divisor](int x) { return x % divisor == 0; });
```

`[&divisor]` indicates that there is an *external* dependency and it is not enough to "just copy" the lambda function if needed elsewhere.

(off-topic: check out this interesting article about copy & paste bugs in real world applications: "The Last Line Effect" by the PVS-Studio team, `www.viva64.com/en/b/0260/`)

Avoid default capture modes

Does the following implementation looks fine?

```
struct Widget {
    int divisor = 2;

    void add_filter() const {
        filters.emplace_back([=](int x) { return x % divisor == 0; });
    }
};
```

…given a sufficient implementation of `filters`

Avoid default capture modes

Does the following implementation looks fine?

```cpp
struct Widget {
    int divisor = 2;

    void add_filter() const {
        filters.emplace_back([=](int x) { return x % divisor == 0; });
    }
};
```

…given a sufficient implementation of `filters`

No! Horrible code! Capturing only applies to non-`static` local variables. Why does this work?

Avoid default capture modes

Capturing only applies to non-`static` local variables. Why does this work?

```
Widget::add_filter() const {
    filters.emplace_back([=](int x) { return x % divisor == 0; });
}
```

…but this fails

```
Widget::add_filter() const {
    filters.emplace_back([](int x) { return x % divisor == 0; });
}
```

…and this also

```
Widget::add_filter() const {
    filters.emplace_back([divisor](int x) { return x % divisor == 0; });
}
```

Avoid default capture modes

There is no local variable `divisor`! But what happes is the following

```
Widget::add_filter() const {
    filters.emplace_back([=](int x) {
        return x % divisor == 0;
    });
}
```

copies (implicitly) `this` pointer (until C++17), i.e.

```
Widget::add_filter() const {
    auto copy_of_this = this;
    filters.emplace_back([copy_of_this](int x) {
        return x % copy_of_this->divisor == 0;
    });
}
```

…welcome to the world of undefined behavior, when `Widget` goes out of scope!

Avoid default capture modes

Default capturing by value can be misleading and gives the impression that a lambda is self-contained:

```
static auto divisor = 1;
filters.emplace_back([=](int x) { return x % divisor == 0; });
++divisor;
```

Above, `divisor` is not copied! (as one may have guessed seeing `[=]`)

Stop using `std::bind`

…and prefer lambda expression, since

- → this increases readability,
- → lambdas are much more flexible,
- → `std::bind` can potentially introduce additional overhead at run-time, whereas lambdas are default `constexpr`

Stop using `std::function`

- → `std::function` adds multiple copies of passed object (consider using drop-in replacements such as *delegates**)
- → may cause heap allocation
- → is just a wrapper …

…deduce type of lambda via `auto` or template deduction, **if possible** (cf. exercise)

*`codereview.stackexchange.com/questions/14730/impossibly-fast-delegate-in-c11`

`std::function`: Const-correctness / thread-safety bug

```cpp
#include <functional>
#include <iostream>

struct Callable {
    auto operator()() {
        return 1;
    }

    auto operator()() const {
        return 2;
    }
} c;

int main() {
    std::function<int(void)> f = c;
    const auto cf = f;
    std::cout << f() << cf(); // 11
}
```

godbolt.org/z/f1fE7G

```cpp
#include <iostream>
#include <optional>

int main() {
    using opt_t = std::optional<int>;
    auto o = []() -> opt_t {
        return 42;
    }();
    const auto co = o;
    std::cout << (*o)++;  // O.K.
    std::cout << (*co)++; // error!
}
```

godbolt.org/z/cT7E9j

Compile-time Error: increment of read-only location

(shamelessly stolen from Titus Winters, Pacific++ 2018: *C++ Past vs. Future*)

Inheriting from Lambdas

Inheriting from Lambdas

Consider two lambdas

```
auto f1 = [] { return 1; };
auto f2 = [](int x) { return x; };
```

Is it possible to combine both lambdas (by inheritance) in one common type X?

```
X combined{f1, f2};
auto a = combined();   // should return 1
auto b = combined(42); // should return 42
```

Inheriting from Lambdas

```cpp
struct X: F1, F2 {
    X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}

    using F1::operator();
    using F2::operator();
};
```

…but what is the type of a lambda / what are F1 and F2?

According to the C++17 standard, will this compile?

```
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
    using F1::operator();
    using F2::operator();
};

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/qrex3v

According to the C++17 standard, will this compile?

```
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
    using F1::operator();
    using F2::operator();
};

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/qrex3v

yes!

According to the C++14 standard, will this compile?

```cpp
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
    using F1::operator();
    using F2::operator();
};

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/qrex3v

According to the C++14 standard, will this compile?

```
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
    using F1::operator();
    using F2::operator();
};

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/qrex3v

error: use of class template “X” requires template arguments

Inheriting from Lambdas

What are the deduced types of `auto` / what are the types of `f1` and `f2`?

```
auto f1 = [] { return 1; };
auto f2 = [](int x) { return x; };
```

Use `decltype` to find out!

```
X<decltype(f1), decltype(f2)> combined{f1, f2};
```

Inheriting from Lambdas

…or extract this to a factory function `make_combined`

```cpp
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
    using F1::operator();
    using F2::operator();
};

template <typename F1, typename F2> auto make_combined(F1&& f1, F2&& f2) {
    return X<std::decay_t<F1>, std::decay_t<F2>>{std::forward<F1>(f1),
                                                  std::forward<F2>(f2)};
}

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    auto combined = make_combined(f1, f2);
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/b7vznr

According to the C++17 standard, will this compile?

```cpp
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    using F1::operator();
    using F2::operator();
};

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/sM1jhW

According to the C++17 standard, will this compile?

```
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    using F1::operator();
    using F2::operator();
};

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/sM1jhW

error: cannot deduce template arguments of “`X<F1, F2>`”, as it has no viable deduction guides

According to the C++17 standard, will this compile?

```cpp
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    using F1::operator();
    using F2::operator();
};

template <typename F1, typename F2>
X(F1, F2) -> X<std::decay_t<F1>, std::decay_t<F2>>;

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/6fEGPT

Variadic Templates

```cpp
#include <iostream>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};

template <typename... Fs>
X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    auto f3 = [](double x) { return -x; };
    X combined{f1, f2, f3};
    std::cout << combined() << '\n'     // should print "1"
              << combined(2) << '\n'    // should print "2"
              << combined(3.4) << '\n'; // should print "-3.4"
}
```

godbolt.org/z/jdqWYa

Why?

{

std::variant

An enum class models a **choice between values**:

```cpp
enum class Oven { on, off };
```

`std::variant` models a **choice between types**:

```cpp
struct on { double temperature; };
struct off {};
using Oven = std::variant<on, off>;
```

std::variant

An aggregate type of some simple shapes …

```
struct Shape {
    enum class Type { Circle, Box } type;

    union {
        struct { double radius; } circle;
        struct { double width, height; } box;
    } geometry;
};
```

std::variant

…and an outer function that calculates the respective area

```
auto area(const Shape& shape) {
    switch(shape.type) {
        case Shape::Type::Circle: {
            const auto& g = shape.geometry.circle;
            return M_PI * g.radius * g.radius;
        }
        case Shape::Type::Box: {
            const auto& g = shape.geometry.box;
            return g.width * g.height;
        }
    }

    assert(false);
    __builtin_unreachable();
}
```

```cpp
#include <cassert>
#include <cmath>

struct Shape {
    enum class Type { Circle, Box } type;

    union {
        struct { double radius; } circle;
        struct { double width, height; } box;
    } geometry;
};

auto area(const Shape& shape) {
    switch(shape.type) {
        case Shape::Type::Circle: {
            const auto& g = shape.geometry.circle;
            return M_PI * g.radius * g.radius;
        }
        case Shape::Type::Box: {
[...]
```

godbolt.org/z/zYqTKn

Using `std::variant` instead

```cpp
#include <cmath>
#include <variant>

struct Circle { double radius; };
struct Box { double width, height; };
using Shape = std::variant<Circle, Box>;

auto area(const Shape& shape) {
    struct {
        auto operator()(const Circle& c) const {
            return M_PI * c.radius * c.radius;
        }
        auto operator()(const Box& b) const {
            return b.width * b.height;
        }
    } visitor;

    return std::visit(visitor, shape);
}
```

godbolt.org/z/6eT7qf

}

Q: What is the output of the program?

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};
template <typename... Fs> X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    int a = 0; double b = 0.;
    X visitor{[&a](int x) { a += x; },
              [&b](double x) { b += x; }};
    std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
    std::for_each(v.begin(), v.end(), [&visitor](const auto &x) {
        std::visit(visitor, x);
    });
    std::cout << a << ' ' << b;
}
```

godbolt.org/z/rKdTv5

A: 34

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};
template <typename... Fs> X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    int a = 0; double b = 0.;
    X visitor{[&a](int x) { a += x; },
              [&b](double x) { b += x; }};
    std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
    std::for_each(v.begin(), v.end(), [&visitor](const auto &x) {
        std::visit(visitor, x);
    });
    std::cout << a << ' ' << b;
}
```

godbolt.org/z/rKdTv5

Q: What is the output of the program?

```
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};
template <typename... Fs> X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    int a = 0; double b = 0.;
    X visitor{[&a](int x) { a += x; },
              [&b](double x) { b += x; }};
    std::vector<std::variant<int, double, const char*>> v{1, 1.9, 2, 2.1, "foo"};
    std::for_each(v.begin(), v.end(), [&visitor](const auto& x) {
        std::visit(visitor, x);
    });
    std::cout << a << b;
}
```

godbolt.org/z/9fv85o

A: Compile-time error! (`visitor` is not exhaustive)

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};
template <typename... Fs> X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    int a = 0; double b = 0.;
    X visitor{[&a](int x) { a += x; },
              [&b](double x) { b += x; }};
    std::vector<std::variant<int, double, const char*>> v{1, 1.9, 2, 2.1, "foo"};
    std::for_each(v.begin(), v.end(), [&visitor](const auto& x) {
        std::visit(visitor, x);
    });
    std::cout << a << b;
}
```

godbolt.org/z/9fv85o

Using plain old `structs`

```
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

struct X {
    int &a; double &b;
    auto operator()(int x) { a += x; };
    auto operator()(double x) { b += x; };
};

int main() {
    int a = 0; double b = 0.;
    X visitor{.a=a, .b=b};
    std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
    std::for_each(v.begin(), v.end(), [&visitor](const auto& x) {
        std::visit(visitor, x);
    });
    std::cout << a << b;
}
```

godbolt.org/z/r3a1q4

Nota bene

One could also use a generic lambda…

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

int main() {
    int a = 0; double b = 0.;
    std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
    std::for_each(v.begin(), v.end(), [&a, &b](const auto& x) {
        std::visit([&a, &b](auto x) {
            if constexpr (std::is_same_v<int, decltype(x)>) a += x;
            else b += x;
        }, x);
    });
    std::cout << a << b;
}
```

godbolt.org/z/5T7nTx

…however, no check for exhaustiveness at compile-time here!

Q: What is the output of the program?

```cpp
#include <iostream>
#include <variant>

struct A { auto f() { return 1; }};
struct B { auto g() { return 2; }};

int main() {
    std::visit([](auto x) {
        using X = decltype(x);
        if constexpr (std::is_same_v<X, A>) {
            std::cout << x.f();
        } else if constexpr (std::is_same_v<X, B>) {
            std::cout << x.g();
        } else {
            std::cout << x.palim();
        }
    }, std::variant<A, B>{A{}});
}
```

godbolt.org/z/63zbqd

A: 1

```cpp
#include <iostream>
#include <variant>

struct A { auto f() { return 1; }};
struct B { auto g() { return 2; }};

int main() {
    std::visit([](auto x) {
        using X = decltype(x);
        if constexpr (std::is_same_v<X, A>) {
            std::cout << x.f();
        } else if constexpr (std::is_same_v<X, B>) {
            std::cout << x.g();
        } else {
            std::cout << x.palim();
        }
    }, std::variant<A, B>{A{}});
}
```

godbolt.org/z/63zbqd

`std::variant` evaluation at compile-time

```cpp
#include <variant>

template<typename... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<typename... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main() {
    using T = std::variant<int, double>;
    overloaded visitor = overloaded{[](int x) -> T { return x + 1; },
                                    [](double x) -> T { return x + 2.; }};
    constexpr auto result = std::visit(visitor, T{41});
    static_assert(result.index() == 0 && std::get<0>(result) == 42);
}
```

godbolt.org/z/KM9nej