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

iCSC 2020

Nis Meinert

Rostock University

Introduction

What is a Lambda Expression in C++?

`cube` is a lambda …

```
1  int main() {
2      auto cube = [](int x) { return x * x * x; };
3      return cube(3);
4  }
```

godbolt.org/z/zBE2_n

`is_even` is a lambda …

```
1  #include <algorithm>
2  #include <vector>
3
4  int main() {
5      std::vector<int> xs{1, 2, 3, 4, 5, 6, 7};
6      auto is_even = [](int x) { return x % 2 == 0; };
7      return std::count_if(xs.begin(), xs.end(), is_even);
8  }
```

godbolt.org/z/kWx7qu

Syntax

C++'s Lambda Expression

The simplest (and most boring) lambda

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

…no capturing, takes no parameters and returns nothing

A slightly more “useful” lambda

```
1 int main() {
2     auto x = [] { return 5; };
3     return x();
4 }
```

godbolt.org/z/DrnSSE

…is equivalent to

```
1 int main() {
2     struct {
3         auto operator()() const {
4             return 5;
5         }
6     } x;
7     return x();
8 }
```

godbolt.org/z/R8qx3Q

Q: What is the output of the program?

```
1  #!/usr/bin/env python3
2
3  if __name__ == '__main__':
4      f = {k: lambda x: x + k for k in range(3)}
5
6      for k in range(3):
7          print(f[k](2), end='')
```

Fix

```
1  #!/usr/bin/env python3
2
3  from functools import partial
4
5  if __name__ == '__main__':
6      f = {k: partial(lambda x, k: x + k, k=k) for k in range(3)}
7      # f = {k: lambda x, k=k: x + k for k in range(3)}
8      # ... would change API
9
10     for k in range(3):
11         print(f[k](2), end='')
```

Now prints: 234

Q: What is the output of the program?

```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, [](int x) { return x + k; });
    }

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

godbolt.org/z/QssJXN

Q: What is the output of the program?

```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/qHFY32

Q: What is the output of the program?

```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/-FTWqI

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

```
1  int main() {
2      int i = 1;
3      auto z = [i](int y) {
4          return i + y;
5      }(3);
6      return z;
7  }
```

godbolt.org/z/bHveG8

…or equivalently

```
1  class X {
2  private:
3      int i;
4
5  public:
6      X(int i): i(i) {}
7
8      int operator()(int y) const {
9          return i + y;
10     }
11 };
12
13 // potentially lots of lines of code
14
15 int main() {
16     int i = 1;
17     auto z = X{i}(3);
18     return z;
19 }
```

godbolt.org/z/8tiwby

Capturing by reference

```
1  int main() {
2      int i = 1;
3      auto z = [&i](int y) {
4          return i + y;
5      }(3);
6      return z;
7  }
```

godbolt.org/z/xazquF

…or equivalently

```
1  class X {
2  private:
3      int& i;
4
5  public:
6      X(int& i): i(i) {}
7
8      int operator()(int y) /*const*/ {
9          return i + y;
10     }
11 };
12
13 // potentially lots of lines of code
14
15 int main() {
16     int i = 1;
17     auto z = X{i}(3);
18     return z;
19 }
```

godbolt.org/z/3ycaAW

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      int i = 1;
5      auto x = [i]() { return ++i; };
6      std::cout << i << x() << i;
7  }
```

godbolt.org/z/nv83nh

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      int i = 1;
5      auto x = [i]() mutable { return ++i; };
6      std::cout << i << x() << i;
7  }
```

godbolt.org/z/Gs995r

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/gEhwLt

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      auto x = [i=0]() mutable { return ++i; };
5      std::cout << x() << x();
6  }
```

godbolt.org/z/iLqPrn

Q: What is the output of the program?

```
1  #include <iostream>
2  #include <utility>
3
4  int main() {
5      auto x = [i=0, j=1]() mutable {
6          i = std::exchange(j, j + i);
7          return i;
8      };
9
10     for (int i = 0; i < 5; ++i) {
11         std::cout << x();
12     }
13 }
```

godbolt.org/z/eTdadM

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 `structs` with an appropriate `operator()()` overload …

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      auto x = [] { return 1; };
5      auto y = x;
6      std::cout << x() << y();
7  }
```

godbolt.org/z/i_AnMx

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/35Q3uR

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/u-6mxM

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/U-CLpA

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      auto x = [] { static int i = 0; return ++i; };
5      auto y = x;
6      x();
7      x();
8      y();
9      y();
10     std::cout << x();
11 }
```

godbolt.org/z/_8QjoA

Stateful Lambdas

Fibonacci (again):

```
1  #include <utility>
2
3  int main() {
4      auto fib = [i=0, j=1]() mutable {
5          struct Result {
6              int &i, &j;
7
8              auto next() {
9                  i = std::exchange(j, j + i);
10                 return *this;
11             }
12         };
13         return Result{.i=i, .j=j}.next();
14     };
15
16     fib().next().next().next(); // mutate state
17     return fib().i;
18 }
```

godbolt.org/z/m9s7ei

Stateful Lambdas

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

```
1  #include <utility>
2
3  int main() {
4      auto fib = [i=0, j=1]() mutable {
5          struct Result {
6              int &i, &j;
7
8              auto next() {
9                  i = std::exchange(j, j + i);
10                 return *this;
11             }
12         };
13         return Result{.i=i, .j=j}.next();
14     };
15
16     auto r = fib();
17     r.i = 2; // mutate state
18     r.j = 3; // mutate state
19     return fib().j; // 5
20 }
```

godbolt.org/z/xpLDpb

Stateful Lambdas

…or slightly more conveniently:

```
1  #include <utility>
2
3  int main() {
4      auto fib = [i=0, j=1]() mutable {
5          struct Result {
6              int &i, &j;
7
8              auto next(int n = 1) {
9                  while (n-- > 0) {
10                     i = std::exchange(j, j + i);
11                 }
12                 return *this;
13             }
14         };
15         return Result{.i=i, .j=j}.next();
16     };
17
18     return fib().next(3).j; // 5
19 }
```

godbolt.org/z/aN3sNi

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/ok7Za-

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

godbolt.org/z/ok7Za-

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/y-343Z

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/J7cccJ

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/NC6DKj

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/p_I8hF

(cf. IIFE: bfilipek.com/2016/11/iife-for-complex-initialization.html)

Avoid default capture modes

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

```
1 void add_filter() {
2     auto divisor = get_magic_number();
3     filters.emplace_back([&](int x) { return x % divisor == 0; });
4 }
```

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

```
1 void add_filter() {
2     auto divisor = get_magic_number();
3     filters.emplace_back([&divisor](int x) { return x % divisor == 0; });
4 }
```

Avoid default capture modes

Mitigation of copy & paste bugs:

```
1 auto divisor = get_magic_number();
2 std::find_if(container.begin(),
3              container.end(),
4              [&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?

```
1  struct Widget {
2      int divisor = 2;
3
4      void add_filter() const {
5          filters.emplace_back([=](int x) { return x % divisor == 0; });
6      }
7  };
```

…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?

```
1 Widget::add_filter() const {
2     filters.emplace_back([=](int x) { return x % divisor == 0; });
3 }
```

…but this fails

```
1 Widget::add_filter() const {
2     filters.emplace_back([](int x) { return x % divisor == 0; });
3 }
```

…and this also

```
1 Widget::add_filter() const {
2     filters.emplace_back([divisor](int x) { return x % divisor == 0; });
3 }
```

Avoid default capture modes

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

```
1  Widget::add_filter() const {
2      filters.emplace_back([=](int x) {
3          return x % divisor == 0;
4      });
5  }
```

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

```
1  Widget::add_filter() const {
2      auto copy_of_this = this;
3      filters.emplace_back([copy_of_this](int x) {
4          return x % copy_of_this->divisor == 0;
5      });
6  }
```

…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:

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

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

Stop using `std::bind`

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`

Stop using `std::function`

- → `std::function` add multiple copies of passed object (consider using drop-in replacements such as *delegate*s*)
- → 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`

Inheriting from Lambdas

Inheriting from Lambdas

Consider two lambdas

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

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

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

Inheriting from Lambdas

```
1  struct X: F1, F2 {
2      X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
3
4      using F1::operator();
5      using F2::operator();
6  };
```

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

According to the C++17 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/nMNbMZ

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/nMNbMZ

Inheriting from Lambdas

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

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

Use `decltype` to find out!

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

Inheriting from Lambdas

…or extract this to a factory function `make_combined`

```
1  #include <iostream>
2
3  template <typename F1, typename F2> struct X: F1, F2 {
4      X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
5      using F1::operator();
6      using F2::operator();
7  };
8
9  template <typename F1, typename F2> auto make_combined(F1&& f1, F2&& f2) {
10     return X<std::decay_t<F1>, std::decay_t<F2>>{std::forward<F1>(f1),
11                                                  std::forward<F2>(f2)};
12 }
13
14 int main() {
15     auto f1 = [] { return 1; };
16     auto f2 = [](int x) { return x; };
17     auto combined = make_combined(f1, f2);
18     std::cout << combined() << combined(2); // should print "12"
19 }
```

godbolt.org/z/dmBP8E

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/MhrL87

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/qDYu3G

Variadic Templates

```
1  #include <iostream>
2
3  template <typename... Fs> struct X: Fs... {
4      using Fs::operator()...;
5  };
6
7  template <typename... Fs>
8  X(Fs...) -> X<std::decay_t<Fs>...>;
9
10 int main() {
11     auto f1 = [] { return 1; };
12     auto f2 = [](int x) { return x; };
13     auto f3 = [](double x) { return -x; };
14     X combined{f1, f2, f3};
15     std::cout << combined() << '\n'     // should print "1"
16               << combined(2) << '\n'    // should print "2"
17               << combined(3.4) << '\n'; // should print "-3.4"
18 }
```

godbolt.org/z/T8wYP2

Why?

{

std::variant

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

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

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

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

std::variant

An aggregate type of some simple shapes …

```
1  struct Shape {
2      enum class Type { Circle, Box } type;
3
4      union {
5          struct { double radius; } circle;
6          struct { double width, height; } box;
7      } geometry;
8  };
```

std::variant

…and an outer function that calculates the respective area

```
1  auto area(const Shape& shape) {
2      switch(shape.type) {
3          case Shape::Type::Circle: {
4              const auto& g = shape.geometry.circle;
5              return M_PI * g.radius * g.radius;
6          }
7          case Shape::Type::Box: {
8              const auto& g = shape.geometry.box;
9              return g.width * g.height;
10         }
11     }
12
13     assert(false);
14     __builtin_unreachable();
15 }
```

```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/U6Uaip

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/nkvKi2

}

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/8j3M7v

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, 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/qYwPjF

Using plain old `structs`

```cpp
#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/E6SNXT

Nota bene

One could also use a generic lambda…

```
1  #include <algorithm>
2  #include <iostream>
3  #include <variant>
4  #include <vector>
5
6  int main() {
7      int a = 0; double b = 0.;
8      std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
9      std::for_each(v.begin(), v.end(), [&a, &b](const auto& x) {
10         std::visit([&a, &b](auto x) {
11             if constexpr (std::is_same_v<int, decltype(x)>) a += x;
12             else b += x;
13         }, x);
14     });
15     std::cout << a << b;
16 }
```

godbolt.org/z/Dcdmoi

…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/Dyy9mg

`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/b988jT