Lambda in C++

C++98/03

- Lambda was not born yet.
- What could you use with high-order function such as std::for_each?

```
void CallablePrint(std::string const& name) {
    std::cout << "Hello " << name << ", what's up?" << "\n";
}

struct AnotherCallablePrint
{
    void operator()(std::string const& name) {
        std::cout << "Eyy " << name << ", does it work?" << "\n";
    }
};
```

```
std::for_each(
        developers.begin(),
        developers.end(),
        CallablePrint
);

AnotherCallablePrint  anotherCallableObj;
std::for_each(
        developers.begin(),
        developers.end(),
        anotherCallableObj
);
```

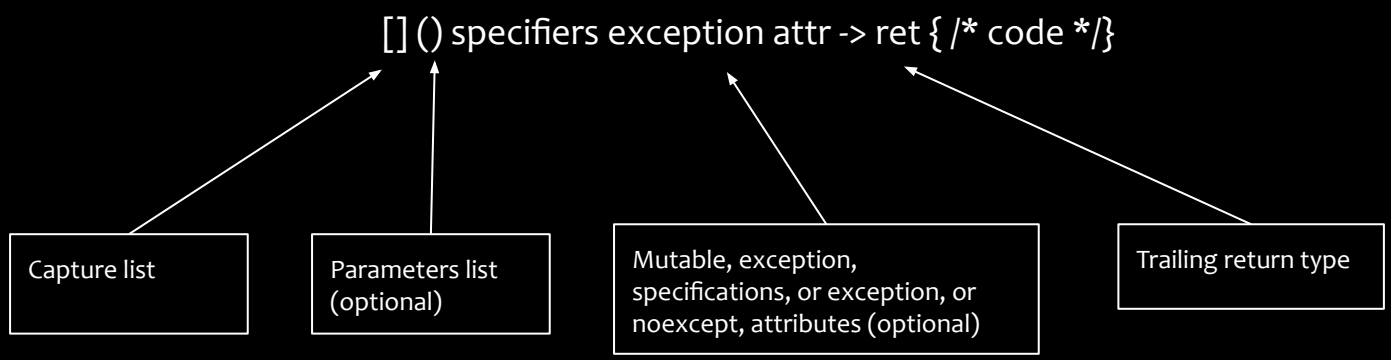
Problems

- Callable objects need to be defined in a different place than the invocation place.
- There is helper class for common callable objects, but does not improve readability.

```
    std::vector<int> goodNumbers;
    goodNumbers.push_back(1);
    goodNumbers.push_back(2);
    goodNumbers.push_back(3);
    goodNumbers.push_back(4);
    goodNumbers.push_back(5);

auto x = std::count_if(
        goodNumbers.begin(),
        goodNumbers.end(),
        std::bind(std::less<int>(), std::placeholders::_1, 2));
```

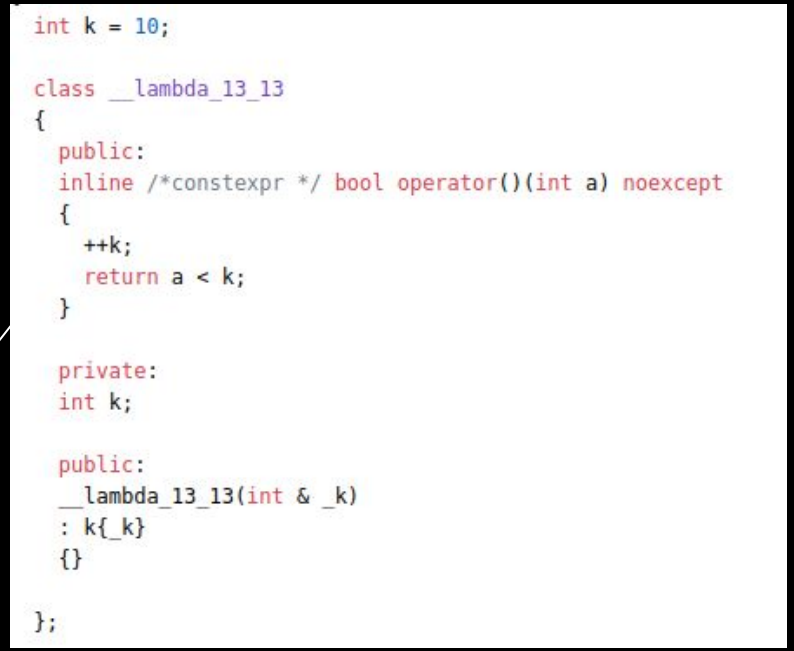
C++11

- Syntax


[] () specifiers exception attr -> ret { /* code */}
Capture list
Parameters list (optional)
Mutable, exception, specifications, or exception, or noexcept, attributes (optional)
Trailing return type

Examples

```
// empty lambda
auto x = [](){};

// with parameters list
auto y = [](int x, int ratio) {return x * ratio;};

// trailing return type
auto z = [](double x) -> int {return x;};

// additional specifiers
int k = 10;
auto a = [k](int a) mutable noexcept {++k; return a < k;};

// optional parameters list
auto b = [k]{std::cout << k;}; // no () need
//auto c = [k] mutable {++k;};
// [] noexcept {};
```



```
int k = 10;

class __lambda_13_13
{
  public:
  inline /*constexpr */ bool operator()(int a) noexcept
  {
    ++k;
    return a < k;
  }

  private:
  int k;

  public:
  __lambda_13_13(int & _k)
  : k{_k}
  {}

};
```

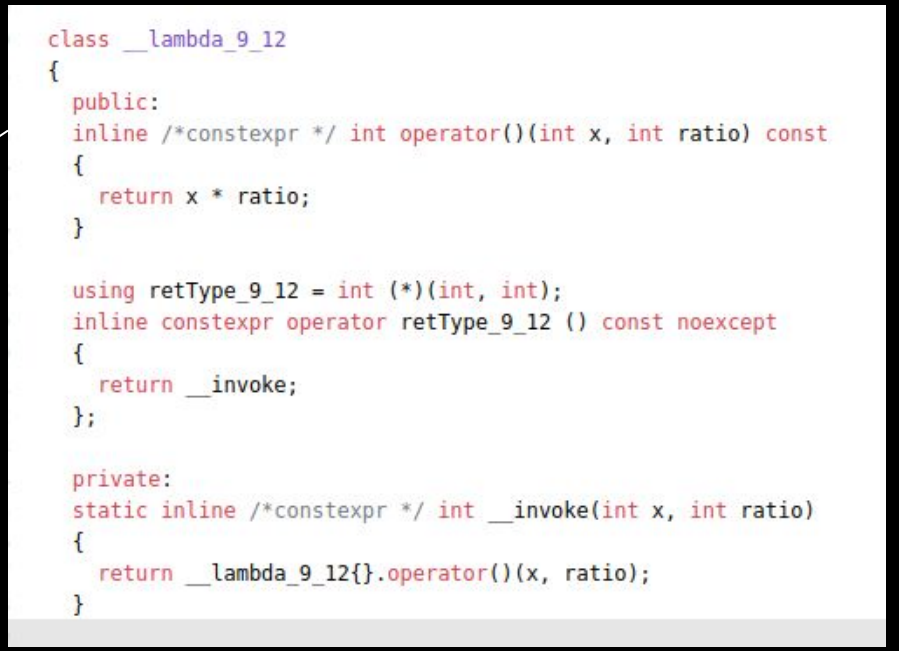
Size of Lambda

```
struct BigBigChicken
{
 uint64_t value;
 uint64_t key;
};

 // empty lambda
 auto x = [] {};

 // with parameters list
 auto y = [](int x, int ratio) { return x * ratio; };
 // additional specifiers
 int k = 10;
 auto a = [k](int a) mutable noexcept {
    ++k;
    return a < k;
 };
 std::cout << "sizeof(x) = " << sizeof(x) << "\n";
 std::cout << "sizeof(y) = " << sizeof(y) << "\n";
 std::cout << "sizeof(a) = " << sizeof(a) << "\n";

 BigBigChicken bigChicken;
 auto c = [&bigChicken]() { std::cout << "I'm a big chicken\n"; };
 std::cout << "sizeof(bigChicken) = "<< sizeof(bigChicken) << "\n";
```

```
1  #include <iostream>
2
3  int main()
4  {
5
6    class __lambda_6_12
7    {
8      public:
9      inline /*constexpr */ void operator()() const
10     {
11     }
12
13     using retType_6_12 = auto (*)() -> void;
14     inline constexpr operator retType_6_12 () const noexcept
15     {
16       return __invoke;
17     };
18
19     private:
20     static inline /*constexpr */ void __invoke()
21     {
22       __lambda_6_12{}.operator()();
23     }
24
```

Size of Lambda

```
struct BigBigChicken
{
 uint64_t value;
 uint64_t key;
};

 // empty lambda
 auto x = [] {};

 // with parameters list
 auto y = [](int x, int ratio) { return x * ratio; };
 // additional specifiers
 int k = 10;
 auto a = [k](int a) mutable noexcept {
    ++k;
    return a < k;
 };
 std::cout << "sizeof(x) = " << sizeof(x) << "\n";
 std::cout << "sizeof(y) = " << sizeof(y) << "\n";
 std::cout << "sizeof(a) = " << sizeof(a) << "\n";

 BigBigChicken bigChicken;
 auto c = [&bigChicken] () { std::cout << "I'm a big chicken\n"; };
 std::cout << "sizeof(bigChicken) = "<< sizeof(bigChicken) << "\n";
```



```
class __lambda_9_12
{
  public:
  inline /*constexpr */ int operator()(int x, int ratio) const
  {
    return x * ratio;
  }

  using retType_9_12 = int (*)(int, int);
  inline constexpr operator retType_9_12 () const noexcept
  {
    return __invoke;
  };

  private:
  static inline /*constexpr */ int __invoke(int x, int ratio)
  {
    return __lambda_9_12{}.operator()(x, ratio);
  }
```

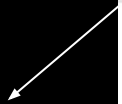
Size of Lambda

```
struct BigBigChicken
{
 uint64_t value;
 uint64_t key;
};

 // empty lambda
 auto x = [] {};

 // with parameters list
 auto y = [](int x, int ratio) { return x * ratio; };
 // additional specifiers
 int k = 10;
 auto a = [k](int a) mutable noexcept {
    ++k;
    return a < k;
 };
 std::cout << "sizeof(x) = " << sizeof(x) << "\n";
 std::cout << "sizeof(y) = " << sizeof(y) << "\n";
 std::cout << "sizeof(a) = " << sizeof(a) << "\n";

 BigBigChicken bigChicken;
 auto c = [&bigChicken]() { std::cout << "I'm a big chicken\n"; };
 std::cout << "sizeof(bigChicken) = "<< sizeof(bigChicken) << "\n";
```

```
int k = 10;

class __lambda_6_11
{
  public:
  inline /*constexpr */ bool operator()(int a) noexcept
  {
    ++k;
    return a < k;
  }

  private:
  int k;

  public:
  __lambda_6_11(int & _k)
  : k{_k}
  {}

};

__lambda_6_11 a = __lambda_6_11{k};
```

Size of Lambda

```cpp
struct BigBigChicken
{
 uint64_t value;
 uint64_t key;
};

 // empty lambda
 auto x = [] {};

 // with parameters list
 auto y = [](int x, int ratio) { return x * ratio; };
 // additional specifiers
 int k = 10;
 auto a = [k](int a) mutable noexcept {
    ++k;
   return a < k;
 };
 std::cout << "sizeof(x) = " << sizeof(x) << "\n";
 std::cout << "sizeof(y) = " << sizeof(y) << "\n";
 std::cout << "sizeof(a) = " << sizeof(a) << "\n";

 BigBigChicken bigChicken;
 auto c = [&bigChicken] () { std::cout << "I'm a big chicken\n"; };
 std::cout << "sizeof(bigChicken) = "<< sizeof(bigChicken) << "\n";
```



```cpp
BigBigChicken bigChicken;

class __lambda_13_12
{
  public:
  inline /*constexpr */ void operator()() const
  {
    std::operator<<(std::cout, "I'm a big chicken\n");
  }

  private:
  BigBigChicken & bigChicken;

  public:
  __lambda_13_12(BigBigChicken & _bigChicken)
  : bigChicken{_bigChicken}
  {}

};

__lambda_13_12 c = __lambda_13_12{bigChicken};
```

Size of Lambda

```
auto d =
   static_cast<int (*)(int, int)>([](int x, int ratio) {
return x * ratio; });

std::cout << "sizeof(d) = " << sizeof(d)
          << "\n"; // 8 bytes (size of a pointer of my
64-bit pc)

// another way of converting to function pointer
// unary + acts as a no-op, therefore the only legal result
here is a function
// pointer
auto e = +[](int x, int ratio) { return x * ratio; };
```

```
class __lambda_13_36
{
  public:
  inline /*constexpr */ int operator()(int x, int ratio) const
  {
    return x * ratio;
  }

  using retType_13_36 = int (*)(int, int);
  inline constexpr operator retType_13_36 () const noexcept
  {
    return __invoke;
  }

  private:
  static inline /*constexpr */ int __invoke(int x, int ratio)
  {
    return __lambda_13_36{}.operator()(x, ratio);
  }


};

using FuncPtr_12 = int (*)(int, int);
FuncPtr_12 d = static_cast<int (*)(int, int)>(__lambda_13_36{}.operator __lambda_13_36::retType_13_36());
std::operator<<(std::cout, "sizeof(d) = ").operator<<(sizeof(d));
```

Preserving Constaness

```
int const f = 10;
auto foo = [f]() mutable {
   std::cout << std::is_const<decltype(f)>::value <<
"\n";
};
foo();
```

```
const int f = 10;

class __lambda_13_14
{
  public:
  inline /*constexpr */ void operator()()
  {
    std::operator<<(std::cout.operator<<(std::integral_constant<bool, true>::value), "\n");
  }

  private:
  const int f;

  public:
  __lambda_13_14(const int & _f)
  : f{_f}
  {}

};
```

Immediately Invoked Functional Expression (IIFE)

- Initialize const variable/object when the initialization logic is complex

```
// iife
int xx = 10, yy = 11;
auto const zz = [xx, yy]() mutable noexcept {
  ++xx;
  --yy;

  return xx + yy;
}();
std::cout << "zz = " << zz << "\n";
```

C++14

- Default parameters
- Return type as auto
- Capture with an initializer
- Generic lambdas

C++14

- Default parameters
- Return type as auto

```
// default parameters
auto f = [](int a, int b = 10) { std::cout << "a + b = " << a + b << "\n";
};
f(20);

// return type deduction, before C++14, the return type can’t be deduced to
// float
auto x = [](int a, float b) { return a + b; };
```

Capture with Initializer list

```
// capture with an initializer
int a = 30;
int b = 12;

auto const foo = [z = a + b]() {
std::cout << z << "\n"; };
foo();
```

```
int a = 30;
int b = 12;

class __lambda_16_20
{
  public:
  inline /*constexpr */ void operator()() const
  {
    std::operator<<(std::cout.operator<<(z), "\n");
  }

  private:
  int z;

  public:
  __lambda_16_20(const int & _z)
  : z{_z}
  {}

};

const __lambda_16_20 foo = __lambda_16_20{a + b};
```

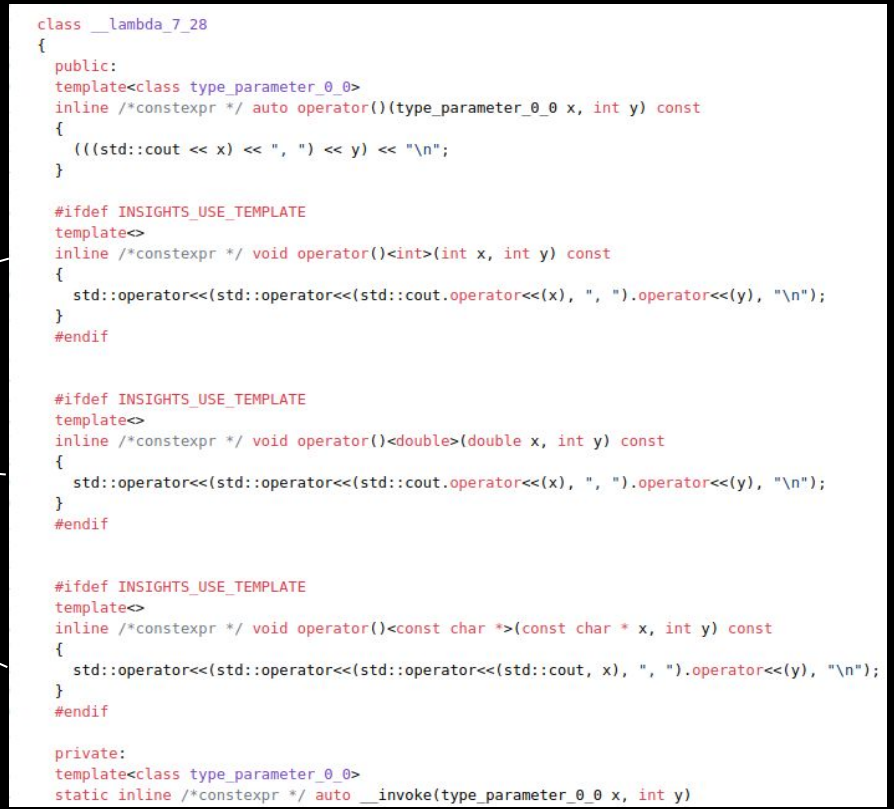
Capture with Initializer list

```
std::unique_ptr<int> p(new int{10});
auto const bar = [ptr = std::move(p)] {
  std::cout << "pointer in lambda: " << ptr.get() <<
"\n";
};
std::cout << "pointer in main(): " << p.get() << "\n";
bar();

// remember that having a only-movable variable makes
bar not copyable object
// Error since std::function only accepts copyable
object
// std::function<void(void)> fx = [ptr = std::move(p)] {
//  std::cout << "no way to compile";
// };
```

```
std::unique_ptr<int, std::default_delete<int> > p = std::unique_ptr<int, std::default_delete<int> >(new int{10});

class __lambda_14_20
{
  public:
  inline /*constexpr */ void operator()() const
  {
    std::operator<<(std::operator<<(std::cout, "pointer in lambda: ").operator<<(reinterpret_cast<const void *>(ptr.get
  }

  private:
  std::unique_ptr<int, std::default_delete<int> > ptr;
  public:
  // inline __lambda_14_20(const __lambda_14_20 &) /* noexcept */ = delete;
  // inline __lambda_14_20 & operator=(const __lambda_14_20 &) /* noexcept */ = delete;
  __lambda_14_20(std::unique_ptr<int, std::default_delete<int> > && _ptr)
  : ptr{std::move(_ptr)}
  {}

};

const __lambda_14_20 bar = __lambda_14_20{std::unique_ptr<int, std::default_delete<int> >(std::move(p))};
```

Generic Lambda

```
// Generic lambda
 auto const generic_foo = [](auto x, int
y) {
   std::cout << x << ", " << y << "\n";
 };
 generic_foo(42, 1);
 generic_foo(4.2, 10);
 generic_foo("hello", 42);
```



```
class __lambda_7_28
{
  public:
  template<class type_parameter_0_0>
  inline /*constexpr */ auto operator()(type_parameter_0_0 x, int y) const
  {
    (((std::cout << x) << ", ") << y) << "\n";
  }

  #ifdef INSIGHTS_USE_TEMPLATE
  template<>
  inline /*constexpr */ void operator()<int>(int x, int y) const
  {
    std::operator<<(std::operator<<(std::cout.operator<<(x), ", ").operator<<(y), "\n");
  }
  #endif


  #ifdef INSIGHTS_USE_TEMPLATE
  template<>
  inline /*constexpr */ void operator()<double>(double x, int y) const
  {
    std::operator<<(std::operator<<(std::cout.operator<<(x), ", ").operator<<(y), "\n");
  }
  #endif


  #ifdef INSIGHTS_USE_TEMPLATE
  template<>
  inline /*constexpr */ void operator()<const char *>(const char * x, int y) const
  {
    std::operator<<(std::operator<<(std::operator<<(std::cout, x), ", ").operator<<(y), "\n");
  }
  #endif

  private:
  template<class type_parameter_0_0>
  static inline /*constexpr */ auto __invoke(type_parameter_0_0 x, int y)
```

Generic Lambda with Variadic Template Parameters

```
// variadic generic arguments with lambda
auto const sumAll = [](auto... args) {
   std::cout << "sum of: " <<
sizeof...(args) << " numbers\n";
   return sum(args...);
};
std::cout << "sumAll = " << sumAll(1, 2,
3.0, 4.2) << "\n";
```

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
double sum<int, int, double, double>(int s, int __ts1, double __ts2, double __ts3)
{
  return static_cast<double>(s) + sum(__ts1, __ts2, __ts3);
}
#endif


#ifdef INSIGHTS_USE_TEMPLATE
template<>
double sum<int, double, double>(int s, double __ts1, double __ts2)
{
  return static_cast<double>(s) + sum(__ts1, __ts2);
}
#endif


#ifdef INSIGHTS_USE_TEMPLATE
template<>
double sum<double, double>(double s, double __ts1)
{
  return s + sum(__ts1);
}
#endif
```

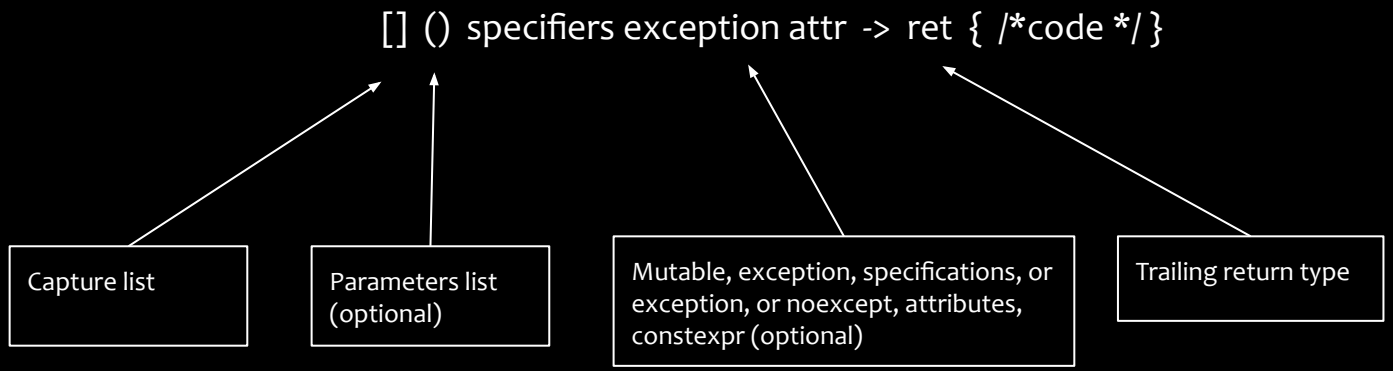
Recursive Lambda

```
auto const recursive_fact = [](int n) noexcept {
  auto const f_impl =
      [](int n, auto const& impl) noexcept -> int
      {
        return n > 1 ? n * impl(n - 1, impl) : 1;
      };
  return f_impl(n, f_impl);
};
```

```
class __lambda_7_31
{
  public:
  inline /*constexpr */ int operator()(int n) const noexcept
  {

    class __lambda_8_25
    {
      public:
      template<class type_parameter_0_0>
      inline /*constexpr */ int operator()(int n, const type_parameter_0_0 & impl) const noexcept
      {
        return n > 1 ? n * impl(n - 1, impl) : 1;
      }

      /* First instantiated from: insights.cpp:11 */
      #ifdef INSIGHTS_USE_TEMPLATE
      template<>
      inline /*constexpr */ int operator()<__lambda_8_25>(int n, const __lambda_8_25 & impl) const noexcept
      {
        return n > 1 ? n * impl.operator()(n - 1, impl) : 1;
      }
      #endif

      private:
      template<class type_parameter_0_0>
      static inline /*constexpr */ int __invoke(int n, const type_parameter_0_0 & impl) noexcept
      {
        return __lambda_8_25{}.operator()<type_parameter_0_0>(n, impl);
      }

    };

    const __lambda_8_25 f_impl = __lambda_8_25{};
    return f_impl.operator()(n, f_impl);
```

C++17

- Constexpr lambda
- Capture *this



Constexpr Lambda

Conditions for constexpr function

- It shall not be virtual
- Its return type shall be a literal type
- Each of its parameter types shall be a literal type
- Its function body shall be =delete, =default, or a compound statement that does not contain
  - An asm definition
  - A goto statement
  - An identifier label
  - A try block
  - A definition of a variable of non-literal type or of static or thread storage duration for which no initialisation is performed.

Constexpr Lambda

```cpp
#include <array>
#include <iostream>

template<typename Range, typename Func, typename T>
constexpr T
SumAll(Range&& range, Func func, T init)
{
 for(auto&& elem : range) {
   init += func(elem);
 }
 return init;
}

int
main()
{
 constexpr std::array arr{1, 2, 3, 4};

 constexpr auto sum = SumAll(
   arr, [](auto i) { return i * i; }, 0
 );
 static_assert(sum == 30);
}
```

```cpp
template<typename Range, typename Func, typename T>
inline constexpr T SumAll(Range && range, Func func, T init)
{
  {
    auto && __range1 = range;
    for(; ; ) {
      auto && elem;
      init = static_cast<T>(static_cast<dependent type>>(init) + func(elem));
    }

  }
  return init;
}


/* First instantiated from: insights.cpp:19 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int SumAll<const std::array<int, 4> &, __lambda_20_9, int>(const std::array<int, 4> & range, __lambda_20
{
  {
    const std::array<int, 4> & __range1 = range;
    const int * __begin0 = __range1.begin();
    const int * __end0 = __range1.end();
    for(; __begin0 != __end0; ++__begin0) {
      const std::array<int, 4>::value_type & elem = *__begin0;
      init = static_cast<int>(init + func.operator()(elem));
    }
```

Constexpr Lambda

```cpp
#include <iostream>

template<size_t N>
constexpr auto
Factorial() { return N * Factorial<N - 1>();}

template<>
constexpr auto
Factorial<1>(){ return 1; }

template<size_t N, typename Func>
constexpr auto
WeirdFactorial(Func func) { return Factorial<N>() * func();}

int
main()
{
 constexpr auto fact = WeirdFactorial<5>([]() { return 2; });
 static_assert(fact == 240);
}
```

Capture *this

Problem

- Capture of an object via this pointer, the object's lifetime may end before the lambda function has a chance to run. When it runs, it's undefined behavior.

```
/// Capture *this
struct LongLife
{
 int value;
 void print() const { std::cout << "value = " << value << "\n"; }
 auto execute()
 {
   return [*this] {
     std::this_thread::sleep_for(std::chrono::seconds(1));
     print();
   };
 }
 ~LongLife()
 {
   std::cout << "Long live, the King!"<< "\n";
 }
};

int main() {
 std::thread t;
 {
   LongLife longLifeObj{42};
   t = std::thread([&longLifeObj] { longLifeObj.execute()(); });
 }
 t.join();
}
```

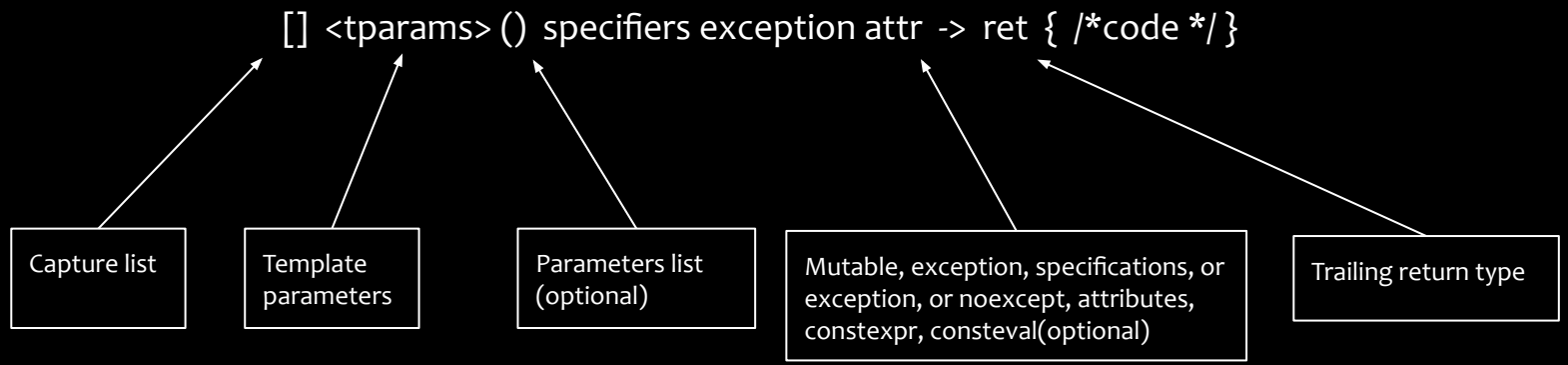
std::invoke

- It is used to invoke a function.

```cpp
// IIFE
[](auto x) { std::cout << "x = " << x << "\n"; }(42);

// std::invoke
 std::invoke([](auto x) { std::cout << "x = " << x << "\n"; }, 42);

// std::apply
std::tuple x{1, 2, 3, "Hello"};
std::apply([](auto... args) { ((std::cout << args << " "), ...); }, x);
```

C++20

- Template lambda
- New options to capture this
- Concepts and constraints



Overloading Pattern

- Inheritance form lambda!

```
// overloading pattern
template<typename... Ts>
struct Overloading : Ts...
{
 using Ts::operator()...;
};
template<typename... Ts>
Overloading(Ts...) -> Overloading<Ts...>;
int main()
{
 // Overloading
 std::vector<std::variant<int, std::string, double>> values{1, "Hello", 4.2};

 for(auto const& v : values) {
   std::visit(
     Overloading(
       [](int v) { std::cout << "v (int) = " << v << "\n"; },
       [](std::string const& v) { std::cout << "v (string) = " << v << "\n"; },
       [](double v) { std::cout << "v (double) = " << v << "\n"; }
     ),
     v
   );
 }
}
```

Template Lambda

- With generic lambda, the generated functor also uses template, but we have no control over the type.
- If we want to access the type from auto, we need decltype

```
// template lambda
auto foo = [](auto x, auto y) { std::cout << x << ", "
<< y << "\n"; };
```

```
auto bar = []<typename T, typename U>(T x, U y) {
  std::cout << x << ", " << y << "\n";
};
```

```
class __lambda_7_14
{
  public:
  template<class type_parameter_0_0, class type_parameter_0_1>
  inline /*constexpr */ auto operator()(type_parameter_0_0 x, type_parameter_0_1 y) const
  {
    (((std::cout << x) << ", ") << y) << "\n";
  }
  private:
  template<class type_parameter_0_0, class type_parameter_0_1>
  static inline /*constexpr */ auto __invoke(type_parameter_0_0 x, type_parameter_0_1 y)
  {
    return __lambda_7_14{}.operator()<type_parameter_0_0, type_parameter_0_1>(x, y);
  }

  public:
  // /*constexpr */ __lambda_7_14() = default;

};

__lambda_7_14 foo = __lambda_7_14{};
```

```
class __lambda_8_14
{
  public:
  template<typename T, typename U>
  inline /*constexpr */ auto operator()(T x, U y) const
  {
    (((std::cout << x) << ", ") << y) << "\n";
  }
  private:
  template<typename T, typename U>
  static inline /*constexpr */ auto __invoke(T x, U y)
  {
    return __lambda_8_14{}.operator()<T, U>(x, y);
  }

  public:
  // /*constexpr */ __lambda_8_14() = default;

};
```

Default Constructible Lambda

- Before C++20, lambda is not default constructible.

```
// default constructible lambda
auto soFoo = [](int x, int y) { std::cout << x << ", "
<< y << "\n"; };

// no error
decltype(soFoo) soBar;
```

```
#include <iostream>

int main()
{

  class __lambda_7_16
  {
    public:
    inline /*constexpr */ void operator()(int x, int y) const
    {
      std::operator<<(std::operator<<(std::cout.operator<<(x), ", ").operator<<(y), "\n");
    }

    using retType_7_16 = void (*)(int, int);
    inline constexpr operator retType_7_16 () const noexcept
    {
      return __invoke;
    };

    private:
    static inline /*constexpr */ void __invoke(int x, int y)
    {
      __lambda_7_16{}.operator()(x, y);
    }

    public:
    // inline /*constexpr */ __lambda_7_16() noexcept = default;
    // /*constexpr */ __lambda_7_16() = default;

  };

  __lambda_7_16 soFoo = __lambda_7_16{};
  __lambda_7_16 soBar;
  return 0;
}
```

Exam

```cpp
#include <iostream>

int
main()
{
 auto x = -(-(!(![]() {}))) - (-(!(![]() {})));

 std::cout << "x = " << x << "\n";
}
```

Test

```cpp
#include <iostream>

int
main()
{
 auto x = -(-(!(![]){})));

 std::cout << "x = " << x << "\n";
}
```

```cpp
class __lambda_5_22
{
  public:
  inline /*constexpr */ void operator()() const
  {
  }

  using retType_5_22 = auto (*)() -> void;
  inline constexpr operator retType_5_22 () const noexcept
  {
    return __invoke;
  }

  private:
  static inline /*constexpr */ void __invoke()
  {
    __lambda_5_22{}.operator()();
  }


};

int x = -(-static_cast<int>((!(!+static_cast<void (*)()>(__lambda_5_22{}.operator __lambda_5_22::retType_5_22())))));
std::operator<<(std::operator<<(std::cout, "x = ").operator<<(x), "\n");
```

References

- Lambda Story. Everything You Need to Know about Lambda in Modern Cpp - Bartlomiej Filipek
- C++ Lambda Idioms - Timur Doumler (CppCon 2022)
- Lambda Calculus in C++ Lambdas - David Stone (Lighting Talk - CppCon 2022)
- Back to Basics: Lambdas - Nicolai Josuttis (CppCon 2021)