Let’s Look at Lambdas

Roger Orr
Let's Look at Lambdas

Roger Orr

OR/2 Limited

auto λ = []((){});

What is this weird syntax? Where might we use it? Why?
What's the problem?

• There are times when it is useful to create a C++ function with minimal syntax, and lambdas in C++ provide a relatively terse way to do this

• A typical use case is a short bespoke function being supplied as an argument to a standard library method

• The concept is present in some other languages too, but there are some details that are primarily C++ specific

• Lambdas were added in C++11, so let's start by seeing what we used to do before we had lambdas...
One of the exciting things when C++98 first came out was the STL, which provided (and still provides) a very rich paradigm using iterators and algorithms.

The STL at its best enabled terse, powerful, and efficient code:

```cpp
void basic_sort(std::vector<int> &v)
{
    ...
    std::sort(v.begin(), v.end());
    ...
}
```
Example: C++98 use of algorithms

- This code instantiates a specialization of the `sort` algorithm for `iterators` for a `vector` of `int` (based on the types of the arguments supplied)

```cpp
void basic_sort(std::vector<int> &v)
{
    ...
    std::sort<std::vector<int>::iterator>(v.begin(), v.end());
    ...
}
```

- It was typesafe (the `right` comparison is automatically used)
- Compile time template instantiation meant this was `faster` than using C's `qsort`
Example: C++98 use of algorithms

- Some customisation was easy
  
  ```cpp
  void reverse_sort(std::vector<int> &v)
  {
    std::sort(v.begin(), v.end(), std::greater<int>());
  }
  ```

- Adding a function object as the comparator in this example reverses the direction of the sort

- It's pretty clear what the code is doing and it's hard to get wrong
Example: C++98 use of algorithms

- Some customisation was easy
  
  ```cpp
  void reverse_sort(std::vector<int> &v)
  {
      std::sort(v.begin(), v.end(), std::greater<int>());
  }
  ```

- Adding a function object as the *comparator* in this example *reverses* the direction of the sort

- It's pretty clear what the code is doing and it's hard to get wrong – or is it?

  ```cpp
  std::sort(v.begin(), v.end(), std::greater()); // error
  std::sort(v.begin(), v.end(), std::greater<int>()); // error
  std::sort(v.begin(), v.end(), std::greater()); // error (up to C++17)
  std::sort(v.begin(), v.end(), std::greater<bool>()); // whoops
  ```

- Ok, maybe I lied...
Example: C++98 use of algorithms

- **C++14** did make this *specific* example a little simpler:

  ```cpp
  void reverse_sort(std::vector<int> &v)
  {
      std::sort(v.begin(), v.end(), std::greater<>());
  }
  ```

- **C++17** made it simpler still, because of Class Template Argument Deduction (CTAD):

  ```cpp
  void reverse_sort(std::vector<int> &v)
  {
      std::sort(v.begin(), v.end(), std::greater());
  }
  ```
Example: C++98 use of algorithms

- Some customisation was harder – suppose we want to sort **absolute** values?

```
void absolute_sort(std::vector<int> &v)
{
    std::sort(v.begin(), v.end(), /* What goes here? */ );
}
```

- We need to supply a **function** or a **function object** that compares the absolute value of two int values.
Example: C++98 use of algorithms

- Using a function may be simplest:

```cpp
bool abs_less_fn(int a, int b)
{
    return std::abs(a) < std::abs(b);
}

void absolute_sort(std::vector<int> &v)
{
    std::sort(v.begin(), v.end(), abs_less_fn);
}
```
Example: C++98 use of algorithms

- We now have a **function** at **module** scope and have **leaked** implementation details out of the original function's scope.
- Incidentally, we have also lost some of the benefits of the STL/C++ as
  - the **type** of the instantiation has a comparator that's simply **pointer-to-function** so the comparison will **not** be inlined*
  - functions are stateless

(* I tried different compilers and various optimization settings without success)
Example: C++98 use of algorithms

• Using a function object is more flexible:

```cpp
struct abs_less
{
    bool operator()(int a, int b)
    {
        return std::abs(a) < std::abs(b);
    }
};

void absolute_sort(std::vector<int> &v)
{
    std::sort(v.begin(), v.end(), abs_less());
}
```
Example: C++98 use of algorithms

- We now have created a **class** at **module** scope and have still **leaked** implementation details out of the original function's scope.
- However, C++98 did not allow you to instantiate templates using a **local** class.
  - Proposal N1353 (Anthony Williams) attempted to resolve this.
  - It is dated **2001-10-15** – this morphed along the way but since C++11 local classes are valid arguments for template instantiations.
  - (Amusingly, perhaps, Visual C++ used to allow this prior to their support for C++98, whereas at least one other vendor had to significantly change their name-mangling implementation to support this for C++11)
- It is much more likely to be optimised (as there is no type erasure).
In C++98/03 you could resolve the scope issue with a `static` function member in a local class.

```cpp
void absolute_sort2(std::vector<int> &v)
{
    struct abs_
    {
        static bool less(int a, int b)
        {
            return std::abs(a) < std::abs(b);
        }
    };

    std::sort(v.begin(), v.end(), abs_::less);
}
```

But that's a lot of 'scaffolding' to support a basically simple piece of code.
Boost to the rescue?

- Some library solutions were provided to try and make the process less painful, with varying degrees of success.
- Here is an example using boost bind:

```cpp
void boost_absolute_sort(std::vector<int> &v)
{
    int (*abs)(int) = std::abs; // (Needed as std::abs is an overload set)

    std::sort(v.begin(), v.end(),
              boost::bind(std::less<int>(),
                          boost::bind(abs, _1), boost::bind(abs, _2)));
}
```
- You may feel this cure is worse than the disease.
In C++98 we had some ways to provide code to algorithms, but with explicit scaffolding:

- Free functions
- Member functions
- Static member functions of local classes
- Template solutions, such as boost::bind

Of course, all these are still available in C++20
A C++11 Lambda solution

- With C++11 we gained support for lambda expressions (which are syntactic sugar for writing an anonymous local class*)

```cpp
void lambda_absolute_sort(std::vector<int> &v)
{
    std::sort(v.begin(), v.end(), [](int a, int b)
    {
        return std::abs(a) < std::abs(b);
    });
}
```

(*Incidentally, the desire for lambda expressions gave support for Anthony's local classes proposal, mentioned earlier)
A C++11 Lambda solution

- With C++11 we now have support for lambda expressions (which are syntactic sugar for writing an anonymous local class)

```cpp
void lambda_absolute_sort(std::vector<int> &v)
{
    std::sort(v.begin(), v.end(), [](int a, int b)
    {
        return std::abs(a) < std::abs(b);
    });
}
```
A C++11 Lambda solution

Pseudo-code roughly equivalent to the previous slide

```cpp
class unnamable {
public:
    auto operator()(int a, int b) {
        return std::abs(a) < std::abs(b);
    }
};

std::sort(v.begin(), v.end(), unnamable());
```

- The lambda `declarator` defines the **arguments** to `operator()`, and the lambda `compound expression` provides the **body** of `operator()`.
A C++11 Lambda solution

Pseudo-code more equivalent to the previous slide

```cpp
void lambda_absolute_sort(std::vector<int> &v)
{
    struct unnamable
    {
        auto operator()(int a, int b) const
        {
            return std::abs(a) < std::abs(b);
        }
    };

    std::sort(v.begin(), v.end(), unnamable());
}
```

- The function call operator in the lambda is implicitly **const**. (This doesn't really matter in this example, but stay tuned ...)
A C++11 Lambda solution

- The type of the lambda is unnamable*, but you can give the expression a name for clarity, or to allow it to be re-used, by using a variable

```cpp
void lambda_absolute_sort(std::vector<int> &v)
{
    auto abs_less = [](int a, int b)
    {
        return std::abs(a) < std::abs(b);
    };
    std::sort(v.begin(), v.end(), abs_less);
}
```

*Hence the variable must be declared auto as the type cannot be named. This was a strong motivation for adding auto to C++11
A C++11 Lambda solution

As far as I can tell there is no single winner for how to format a lambda

auto alternative1 = [](int a, int b) {
    return std::abs(a) < std::abs(b);
};

auto alternative2 = [](int a, int b) {
    return std::abs(a) < std::abs(b);
};

auto alternative3 = []
    (int a, int b)
    {
        return std::abs(a) < std::abs(b);
    };

...
Lambda decay

• For a simple lambda like the one in the previous example C++ provides a conversion to a function pointer. Hence you can pass a lambda to an existing function that takes a function pointer

```
// in some header
using callback = void(int);
void set_callback(callback *cb);

// code that uses this header
void setup()
{
    set_callback([](int val)
    {
        std::cout << "callback with " << val << '\n';
    });
}

• This is known as lambda decay
Lambda decay

- Lambda decay calls a conversion function that returns a pointer to a function that invokes the lambda (typically optimised away)
- **Use** of the pointer *in a template* is likely to prevent optimization

```cpp
void lambda_absolute_sort(std::vector<int> &v)
{
    using bin_func = bool(int,int);
    bin_func *abs_less = [](int a, int b)
    {
        return std::abs(a) < std::abs(b);
    };
    std::sort(v.begin(), v.end(), abs_less); // << Don't do this
}
```
Lambda capture

- A lambda can refer to **local variables** in scope at the point of definition
  - The local variables can be *named* in the lambda introducer

```c
struct type
{
  bool method(int value)
  {
    auto lambda = [value](int test)
    {
      return test < value;
    };
    return lambda(42);
  }
};
```
Lambda capture

- A lambda can refer to local variables in scope at the point of definition and can also refer to members of the enclosing class (if any)
  - The local variables can be named in the lambda introducer
  - Members of the enclosing class can be referred to if this is captured

```cpp
struct type {
    int field;
    bool method(int value) {
        auto lambda = [value, this]() {
            return field < value;
        };
        return lambda();
    }
};
```
Lambda capture – by value or by reference?

- Of course, it's never quite that simple.
- There is a choice with capturing variables between copying the object and forming a reference to the object
- Use an '& to form a reference
  ```cpp
  void test(int value, int reference)
  {
    auto lambda = [value, &reference]()
    {
      std::cout << value << " , " << reference << " \n";
    };
    ++value; ++reference;
    lambda();
  }
  
  test(10, 10) → 10, 11
  ```
Lambda capture – by value or by reference?

- Some pseudo-code may help here

```cpp
void test(int value, int reference)
{
    class __anon
    {
        int __value;
        int& __reference;
    public:
        __anon(int value, int& reference)
            : __value(value), __reference(reference) {} 
        void operator()() const 
        {
            std::cout << __value << ", " << __reference << '\n';
        }
    };
    auto lambda = __anon(value, reference);
    ...
```
Lambda capture by reference

- When you use reference capture you need to think about the *lifetime* of the captured object to avoid dangling references.

```cpp
auto make_adder(int value)
{
    return [&value](int n){ return n + value; };
}

int main()
{
    auto add_ten = make_adder(10);
    int result = add_ten(9);

    std::cout << "9 + 10 = " << result << '\n';
} // 9 + 10 = 18*

(*possibly; or 9, or -13319, or ...*)
What isn't a capture

- A lambda can refer **directly** to global and block scope static variables and static members of the containing class – note these are **not** captures:

```cpp
int global = 1;

struct X {
    static int class_static;
    int method()
    {
        static int block_static = 2;

        auto l = []() {
            return global + block_static + class_static;
        };
        return l();
    }
};
```
You can also define a default capture mode for a lambda, and this allows you to implicitly capture variables.

The options are “=” (for copy) and “&” (for reference)

Any captures not using the default can be specified explicitly

```cpp
void test(int value, int reference)
{
    auto lambda = [=, &reference]()
    {
        std::cout << value << "", " << reference << "\n";
    };
}
```

value is captured implicitly by value, and reference explicitly by ref
Default lambda capture

- Use with care! It's can be quite easy to *accidentally* capture something and end up with either two separate objects with the same name, or a 'dangling' reference to something that has gone out of scope

- There is also a confusion about member variables if you capture this implicitly: members of the class are still accessed by *reference*

```cpp
bool type::method(int value) {
    auto lambda = [=]()
    {
        return field < value; // field (by ref), value (copy)
    };
    return lambda();
}
```

- *(Capturing *this* implicitly, as shown here, is deprecated in C++20)*
Default lambda capture

- Pseudo-code to help to show why members of the captured class are accessed by reference

```cpp
bool type::method(int value)
{
    class __anon {
        type *__this; // copy of the 'this' pointer
        int __value;
        public:
            auto operator()() const
            {
                return __this->field < __value;
            }
            lambda{this, value};
    }
    ... 
}
```
Lambda functions are `const` by default; you can change this by adding the `mutable` keyword.

```cpp
int main()
{
    int counter = 0;
    auto lambda = [counter]() mutable
    {
        std::cout << ++counter << '
';
    };

    Lambda(); // output: 1
    Lambda(); // output: 2
    std::cout << counter << '
'; // local variable counter is still 0
}
```
Warning: const is shallow (just like in regular classes) so variables captured by reference can be modified even without the mutable keyword

```cpp
int main()
{
    int counter = 0;
    auto lambda = [&counter]() {
        std::cout << ++counter << '
';
    };

    Lambda(); // output: 1
    Lambda(); // output: 2
    std::cout << counter << '
'; // counter is now 2
}
```
Can you cheat the anonymity?

- The type of the lambda is *unnamable* - but not *unusable*

```cpp
int main()
{
    int counter = 0;
    auto lambda = [&counter]()
    {
        std::cout << ++counter << '\n';
    };

    int size = sizeof(lambda);
    std::cout << typeid(lambda).name() << '\n';
    decltype(lambda) l2 = lambda;
}

- You can't however create a **different** lambda object as the constructors you would need are deleted
• Lambda syntax, which removes some of the scaffolding
• The lambda behaves very like an anonymous local class
• Capturing variables: by value or by reference
• Mutable lambdas
C++14 added the ability to make the lambda function generic on the type of the arguments, using auto.

This means we can make code using lambdas more generic (!)

```cpp
template <typename T>
void lambda_absolute_sort(std::vector<T> &v)
{
    std::sort(v.begin(), v.end(), [](auto a, auto b)
    {
        return std::abs(a) < std::abs(b);
    });
}
```
We can also, of course, use this to invoke the same lambda with different argument types (hence the alternative name of *polymorphic* lambdas)

```cpp
void fred(int idx, double* ptr)
{
    auto increment = [](auto& v) { ++v; };
    increment(idx);
    increment(ptr);
}
```
Generic lambdas (C++14) - aside

- Note that, with both C++14 and C++17, this means you could create a lambda with deduced argument types but you could not do the same for a regular function

  ```
  void increment(auto& v) { ++v; } // error in C++14/17
  ```

- However, support for using `auto` like this was added to C++20 as part of adding `concepts` to the language
Generic lambdas (C++14)

- The previous example is syntactic sugar for:

  ```cpp
  void fred(int idx, double* ptr)
  {
    class __anon
    {
      template <typename T>
      void operator()(T& v) const { ++v; }
    };
    auto increment = __anon();
    ...
  }
  ```

- except … you are not allowed template members of inner classes!
- This is one of the few places where a lambda is not just syntactic sugar
Generic lambdas (C++14)

- The previous example is syntactic sugar for:

```cpp
void fred(int idx, double value)
{
    class __anon
    {
        template <typename T>
        void operator()(T& v) const { ++v; }
    }
    auto increment = __anon();
    ...
}
```

- **except** ... you are not allowed template members of inner classes!

- There are **two** papers, P1988 (“Allow templates in local classes”) and P2044 (“Member Templates for Local Classes”) trying to fix this. A single merged proposal (P2044) is going forward
Sometimes it is useful to be able to initialise the capture variable from an expression.

```cpp
int main()
{
    auto lambda = [counter = 0]() mutable
    {
        std::cout << ++counter << '
';
    };

    lambda();
    lambda();
    lambda();
}
```

The type of the variable is deduced from the type of the expression.
Sometimes it is useful to be able to rename the capture variable.

```cpp
int main()
{
  int value = ...;

  auto lambda = [counter = value]() mutable
  {
    std::cout << ++counter << '\n';
  };

  lambda();
  lambda();
  lambda();
}
```
Init capture (C++14)

- Init capture also allows you to explicitly capture a **single** field from the containing class, rather than via `this`, and/or to move construct

```cpp
bool type::method(std::string value)
{
    auto lambda = [field{field}, value{std::move(value)}]() {
        return field < value;
    };
    return lambda();
}
```

- Here I've kept the same name for the captured variables – this is fine, at least syntactically, just as it is in a constructor initializer list
In C++17 capture of *this was added; this captures a copy of the current object (or a slice if a class hierarchy)

This is useful when you wish to pass the lambda on and it will outlive the current object. From the proposal paper, P0018:

```cpp
// C++14, using init-capture
std::future spawn() {
    return std::async( [=, tmp=*this]() -> int { return tmp.value; });
}

// Can now be written as

std::future spawn() {
    return std::async( [=, *this]() -> int { return value; });
}
```
Use case: delayed initialising

- Lambdas can be useful for complex variable initialisation to enable, for instance, a variable to be declared const

```c
double delta;
if (high_precision) {
    double h1 = 1e-10;
    h1 = some_complex_calculation(h1, other_args);
    delta = h1;
} else if (something_else) {
    delta = 1e-8;
} else {
    delta = 1e-6;
}
```

- Can delta change later in the function?
- Do I always initialise delta?
Use case: delayed initialising

- Lambdas can be useful for complex variable initialisation to enable, for instance, a variable to be declared const

```cpp
const double delta = [&]() {
    if (high_precision) {
        double h1 = 1e-10;
        return some_complex_calculation(h1, other_args);
    } else if (something_else) {
        return 1e-8;
    }
    return 1e-6;
}();
```

- delta is const so cannot change later in the function
- delta is always initialised (you get a warning about not returning a value)
Use case: as a refactoring tool

• Lambdas can be useful during refactoring
• You've a large method in which you identify a block of code that performs a single function
  • This is traditionally a candidate for 'extract method'
  • A lambda could be used as a more local alternative
  • The **name** and the **scope** help partition the method
• A very similar sequence of operations may be repeated within a method, you may be able to turn this into a lambda which is used repeatedly and reduce the code duplication
  • The **body** of the lambda is the commonality
  • The **arguments** supplied to the lambda differentiate the uses
In previous versions of C++ there was an asymmetry between lambdas and the regular operator() syntax:

```cpp
auto lambda = [](auto x) { return x; }; // Ok
struct {
    template<typename T>
    auto operator()(T x) { return x; }
} functor;
```

The usage is the same:

```cpp
double d = lambda(1.0);
int i = functor(1);
```
C++20: lambda and template arguments

- In the current version of C++ you can now use `auto` in the regular operator() syntax, so you can now write:

```cpp
struct {
    auto operator()(auto x) { return x; }
} functor;
```

- Conversely, you can now use `typename` in the lambda syntax:

```cpp
auto lambda = []<typename T>(T x) { return x; };
```

- There can be advantages in having the name `T` available
The changes also include being able to constrain arguments using concepts, in the same way that regular functions can be

As an example, a lambda with a “numeric” constraint:

```cpp
auto square1 = [](numeric auto x) { return x * x; };
auto square2 = [](numeric T)(T x) { return x * x; };
auto square3 = [](typename T>(T x) requires numeric<T> {
    return x * x;
}
```

The last syntax allows more complex constraints to be expressed, such as a constraint between two argument types
Recap ....

- Generic lambdas
- Init capture
- *this copy capture
- A couple of use cases
- Harmonisation of lambda and template arguments, and concepts
Performance

• Lambdas are very useful – but what's the cost?
• As the title of Chandler Carruth's 2019 CppCon presentation provocatively puts it: “There are no Zero-cost Abstractions”
• While lambdas look simple there is a reasonable amount of 'supporting machinery' being written by the compiler
• Fortunately there is little evidence that this impacts compilation time
• However, might we get “code bloat”? 
Performance – code bloat

- Lambdas are classes and so can potentially introduce code bloat when polymorphic and also when used inside templates.

```cpp
template<class T, class U, class V>
void f() {
    ...;
    [](T t) { ... }(t);
    ...;
}
```

- Each instantiation of f() will include a separate instantiation of the lambda, as it is scoped within f(), even if all the 'T' arguments are the same.

- This can be resolved, where it becomes a problem, using the technique popularised by SCARY iterators (where the type of the iterator only depends on some of the template arguments types and not all of them - N2911).
Performance – code bloat

- The lambda in the previous example could be hoisted outside the function, perhaps like this:

```cpp
constexpr auto the_lambda = [] (auto t) {
    ...
};

template<class T, class U, class V>
void f() {
    ...
    the_lambda(t);
    ...
}
```
Performance

• Lambdas are very useful – but what's the cost?
• As the title of Chandler Carruth's 2019 CppCon presentation provocatively puts it: “There are no Zero-cost Abstractions”
• The additional cost at *compile* time appears to be relatively low
• But I'm sure most people are more concerned about runtime cost...
Performance

- Lambdas are very useful – but what's the cost?
- One hint is that you can add `constexpr` to a lambda*. So the compiler must be able to deconstruct it at compile time.
- In C++20 you can also mark a lambda as `consteval` meaning it can only be evaluated in an immediate context (and back-end likely never sees it)

*If you don't mark the lambda `constexpr` the function call operator will be `implicitly` `constexpr` anyway, if it happens to satisfy all the `constexpr` function requirements.
Here is a simple example of a constexpr lambda.

```cpp
template <int dims>
void function() {
...
    auto square = [](auto v) constexpr {
        return v * v;
    };

double ret[square(dims)];
...
```
Performance

- In an unoptimised build you would expect to see the call to the lambda operator() and so you will see a runtime cost. This is not a great concern to most people.
- Optimisers in many cases will inline the whole lambda class and the calls to its operator().

```cpp
int raw_func(int val)
{
    return val * val;
}

int lambda_func(int val)
{
    auto sq = [val](){
        return val * val;
    };

    int ret = sq();
    
    return ret;
}
```
Performance

- In an **unoptimised** build on x86_64
  - raw_func:
    - 4 instructions (msvc) or 7 (gcc and clang)
  - lambda_func:
    - 11 instructions (msvc) or 13 (gcc and clang)
    - includes 2 (msvc) or 1 (gcc and clang) methods calls
    - the methods total 14 (msvc) or 10 (gcc amd clang) instructions
- In a **minimally optimised** build (-O1) on x86_64
  - raw_func and lambda_func are **identical**
  - Both are 3 instructions in all three compilers
Performance

- In less trivial source code you would need to confirm whether the compiler does indeed optimise away the lambda.
- As you build up more complex code, for example passing lambdas as arguments to another function templates, you will eventually hit the limit of the optimiser. However, you may be surprised how far you can get.
Performance

- Where a lambda is not totally erased, one thing to bear in mind is that the implicit class layout of the lambda class is decided early in compilation, and can't be reduced later on during optimisation even if the compiler is able to eliminate some of the captured variables.
- It's not clear (to me anyway) how often this is a significant problem.
- As always, I suggest getting the code working solidly first and worrying about this issue later if it is identified by things such as profiling.
Down with ()!

- When there are no arguments to the lambda the () can be omitted:
  ```cpp
  auto sq = [val] () { return val * val; };
  ```
- can also be written as:
  ```cpp
  auto sq = [val] { return val * val; };
  ```
- Of course, some coding styles prefer retaining the ().
- Unfortunately the () is currently *required* if you wish to use `mutable`
  ```cpp
  auto seq = [val = 0] () mutable { return ++val; };
  ```
- This will be fixed in C++23 (See P1102R2, approved 2021-02-22)
  ```cpp
  auto seq = [val = 0] mutable { return ++val; };
  ```
- This may not change your world greatly
Possible future shortening?

- Lambdas in C++ require more syntax than in some other languages
- There is disagreement about how much of this is necessary
  - C++ has value and reference semantics, and const
  - Substitution failure is not an error (SFINAE) imposes some constraints
  - C++ syntax makes parsing complicated
- There have been attempts to allow abbreviated syntax. In particular, some would like to remove the necessity of providing the 'scaffolding' of a function call (braces and a return statement). For example, P0583R2 (2017) proposed the syntax => *expression*. While this paper was rejected by EWG they did not reject the principle of exploring shorter syntax in the future.
Conclusion

- Lambdas are a very useful feature of C++
- They are primarily “syntactic sugar” to remove supporting code
- Capture by value or by reference as appropriate, and consider lifetime
- Prefer to avoid implicit capture
- Using a named variable for the lambda expression can aid readability