Lambdas: From C++11 to C++20, Part 2

lambda, closure, captures, ...
C++
capturing this, constexpr, ...
C++17
templates, capturing [=, this], ...
C++20
Lambdas: From
C++11 to C++20
(part 2)

In the first part of the series we looked at lambdas from the perspective of C++03, C++11 and C++14. In that article, I described the motivation behind this powerful C++ feature, basic usage, syntax and improvements in each of the language standards. I also mentioned several corner cases.

Now it's time to move into C++17 and look a bit into the future (very near future!): C++20.

As a little reminder, the idea for the series comes from one of our recent C++ User Group meeting in Cracow.
We had a live coding session about the "history" of lambda expressions. The talk was lead by a C++ Expert Tomasz Kamiński (see Tomek's profile at Linkedin). See this event: Lambdas: From C++11 to C++20 - C++ User Group Krakow I've decided to take the code from Tomek (with his permission and feedback!), describe it and form the articles. So far, in the first part of the series, I described the following elements of lambda expressions:

- Basic syntax
- The type of a lambda
- The call operator
- Captures (mutable, globals, static variables, class member and this pointer, move-able-only objects, preserving const)
    - Return type
    - IIFE - Immediately Invoked Function Expression
    - Conversion to a function pointer
- Improvements in C++14
    - Return type deduction
    - Captures with an initialiser
    - Capturing a member variable
    - Generic lambdas The above list is just a part of the story of lambdas! Let's now see what changed in C++17 and what we will get in C++20!

The standard (draft before publication) N659 and the lambda section: [expr.prim.lambda]. C++17 added two significant enhancements to lambda expressions:

- constexpr lambdas
- Capture of *this What do those features mean for you? Let’s find out.

Since C++17, if possible, the standard defines operator() for the lambda type implicitly as constexpr: From expr.prim.lambda #4:

> *The function call operator is a constexpr function if either the corresponding lambda-expression’s parameter-declaration-clause is followed by constexpr, or it satisfies the requirements for a constexpr function..*

For example:
```
constexpr auto Square = [] (int n) { return n*n; }; // implicitly constexpr
static_assert(Square(2) == 4);
```

To recall, in C++17 a constexpr function has the following rules:

- *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, or*
        - *a definition of a variable of non-literal type or of static or thread storage duration or for which no initialization is performed.*

How about a more practical example?
```
template<typename Range, typename Func, typename T>
constexpr T SimpleAccumulate(const Range& range, Func func, T init) {
    for (auto &&elem: range) {
        init += func(elem);
    }
    return init;
}

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

    static_assert(SimpleAccumulate(arr, [](int i) {
            return i * i;
        }, 0) == 14);
}
```

Play with code @Wandbox

The code uses a constexpr lambda and then it’s passed to a straightforward algorithm SimpleAccumulate. The algorithm also uses a few C++17 elements: constexpr additions to std::array, std::begin and std::end (used in range-based for loop) are now also constexpr so it means that the whole code might be executed at compile time.

Of course, there's more.

You can also capture variables (assuming they are also constant expressions):
```
constexpr int add(int const& t, int const& u) {
    return t + u;
}

int main() {
    constexpr int x = 0;
    constexpr auto lam = [x](int n) { return add(x, n); };

    static_assert(lam(10) == 10);
}
```

But there's a interesting case where you don't "pass" captured variable any further, like:
```
constexpr int x = 0;
constexpr auto lam = [x](int n) { return n + x };
```

In that case, in Clang, we might get the following warning:
```
warning: lambda capture 'x' is not required to be captured for this use
```

This is probably because x can be replaced in place in every use (unless you pass it further or take the address of this name).

But please let me know if you know the official rules of this behaviour. I've only found (from cppreference) (but I cannot find it in the draft…)

> *A lambda expression can read the value of a variable without capturing it if the variable*
> ** has const non-volatile integral or enumeration type and has been initialised with a constant expression, or*
> ** is constexpr and has no mutable members.*

Be prepared for the future:

In C++20 we'll have constexpr standard algorithms and maybe even some containers, so constexpr lambdas will be very handy in that context. Your code will look the same for the runtime version as well as for constexpr (compile time) version!

In a nutshell:

consexpr lambdas allows you to blend with template programming and possibly have shorter code.

Let's now move to the second important feature available since C++17:

Do you remember our issue when we wanted to capture a class member?

By default, we capture this (as a pointer!), and that's why we might get into troubles when temporary objects go out of scope… We can fix this by using capture with initialiser (see in the first part of the series).

But now, in C++17 we have another way. We can wrap a copy of *this:

```
#include <iostream>

struct Baz {
    auto foo() {
        return [*this] { std::cout << s << std::endl; };
    }

    std::string s;
};

int main() {
   auto f1 = Baz{"ala"}.foo();
   auto f2 = Baz{"ula"}.foo();
   f1();
   f2();
}
```

Play with the code @Wandbox

Capturing a required member variable via init capture guards you from potential errors with temporary values but we cannot do the same when we want to call a method of the type:

For example:

```
struct Baz {
    auto foo() {
        return [this] { print(); };
    }

    void print() const { std::cout << s << '\n'; }

    std::string s;
};
```

In C++14 the only way to make the code safer is init capture this:

```
auto foo() {
    return [self=*this] { self.print(); };
}
```

But in C++17 it’s cleaner, as you can write:

```
auto foo() {
    return [*this] { print(); };
}
```

One more thing:

Please note that if you write [=] in a member function then this is implicitly captured! That might lead to future errors…. and this will be deprecated in C++20.

And this brings us to another section: the future.

Sorry for a little interruption in the flow :)
I’ve prepared a little bonus if you’re interested in C++17, check it out here:

Download a free copy of C++17 Language Ref Card!

With C++20 we'll get the following features:

- Allow [=, this] as a lambda capture - P0409R2 and Deprecate implicit capture of this via [=] - P0806
- Pack expansion in lambda init-capture: ...args = std::move(args)](){} - P0780
- static, thread_local, and lambda capture for structured bindings - P1091
- template lambdas (also with concepts) - P0428R2
- Simplifying implicit lambda capture - P0588R1
- Default constructible and assignable stateless lambdas - P0624R2
- Lambdas in unevaluated contexts - P0315R4

In most of the cases the newly added features "cleanup" lambda use and they allow some advanced use cases.

For example with P1091 you can capture a structured binding.

We have also clarifications related to capturing this. In C++20 you'll get a warning if you capture [=] in a method:

```
struct Baz {
    auto foo() {
        return [=] { std::cout << s << std::endl; };
    }

    std::string s;
};
```

```
GCC 9:
warning: implicit capture of 'this' via '[=]' is deprecated in C++20
```

Play with code @Wandbox

If you really need to capture this you have to write [=, this].

There are also changes related to advanced uses cases like unevaluated contexts and stateless lambdas being default constructible.

With both changes you'll be able to write:

```
std::map<int, int, decltype([](int x, int y) { return x > y; })> map;
```

Read the motivation behind those features in the first version of the proposals: P0315R0and P0624R0

But let's have a look at one interesting feature: template lambdas.

With C++14 we got generic lambdas which means that parameters declared as auto are template parameters.

For a lambda:

```
[](auto x) { x; }
```

The compiler generates a call operator that corresponds to a following template method:

```
template<typename T>
void operator(T x) { x; }
```

But there was no way to change this template parameter and use real template arguments. With C++20 it will be possible.

For example, how can we restrict our lambda to work only with vectors of some type?

We can write a generic lambda:
```
auto foo = []<typename T>(const auto& vec) {
        std::cout<< std::size(vec) << '\n';
        std::cout<< vec.capacity() << '\n';
    };
```

But if you call it with an int parameter (like foo(10);) then you might get some hard-to-read error:
```
prog.cc: In instantiation of 'main()::<lambda(const auto:1&)> [with auto:1 = int]':
prog.cc:16:11:   required from here
prog.cc:11:30: error: no matching function for call to 'size(const int&)'
   11 |         std::cout<< std::size(vec) << '\n';
```

In C++20 we can write:
```
auto foo = []<typename T>(std::vector<T> const& vec) {
        std::cout<< std::size(vec) << '\n';
        std::cout<< vec.capacity() << '\n';
    };
```

The above lambda resolves to a templated call operator:
```
<typename T>
void operator(std::vector<T> const& s) { ... }
```

The template parameter comes after the capture clause [].

If you call it with int (foo(10);) then you get a nicer message:
```
note:   mismatched types 'const std::vector<T>' and 'int'
```

Play with code @Wandbox

In the above example, the compiler can warn us about the mismatch in the interface of a lambda rather than some code inside the body.

Another important aspect is that in generic lambda you only have a variable and not it’s template type. So if you want to access it, you have to use decltype(x) (for a lambda with (auto x) argument). This makes some code more wordy and complicated.

For example (using code from P0428):
```
auto f = [](auto const& x) {
    using T = std::decay_t<decltype(x)>;
    T copy = x;
    T::static_function();
    using Iterator = typename T::iterator;
}
```

Can be now written as:
```
auto f = []<typename T>(T const& x) {
    T::static_function();
    T copy = x;
    using Iterator = typename T::iterator;
}
```

In the above section, we had a glimpse overview of C++20, but I have one more extra use case for you. This technique is possible even in C++14. So read on.

Currently, we have a problem when you have function overloads, and you want to pass them into standard algorithms (or anything that requires some callable object):

```
// two overloads:
void foo(int) {}
void foo(float) {}

int main()
{
  std::vector<int> vi;
  std::for_each(vi.begin(), vi.end(), foo);
}
```

We get the following error from GCC 9 (trunk):

```
error: no matching function for call to
for_each(std::vector<int>::iterator, std::vector<int>::iterator,
 <unresolved overloaded function type>)
   std::for_each(vi.begin(), vi.end(), foo);
                                       ^^^^^
```

However, there's a trick where we can use lambda and then call the desired function overload.

In a basic form, for simple value types, for our two functions, we can write the following code:

```
std::for_each(vi.begin(), vi.end(), [](auto x) { return foo(x); });
```

And in the most generic form we need a bit more typing:

```
#define LIFT(foo) \
  [](auto&&... x) \
    noexcept(noexcept(foo(std::forward<decltype(x)>(x)...))) \
   -> decltype(foo(std::forward<decltype(x)>(x)...)) \
  { return foo(std::forward<decltype(x)>(x)...); }
```

Quite complicated code… right? :)

Let's try to decipher it:

We create a generic lambda and then forward all the arguments we get. To define it correctly we need to specify noexcept and return type. That's why we have to duplicate the calling code - to get the proper types.

Such LIFT macro works in any compiler that supports C++14.

Play with code @Wandbox

In this blog post, you've seen significant changes in C++17, and we have an overview of C++20 features.

We can notice that with each language iteration lambdas blends with other C++ elements. For example, before C++17 we couldn't use them in constexpr context, but now it’s possible. Similarly with generic lambdas since C++14 and their evolution in C++20 in the form of template lambdas.

Have I skipped something?
Maybe you have some exciting example to share?
Please let me know in comments!