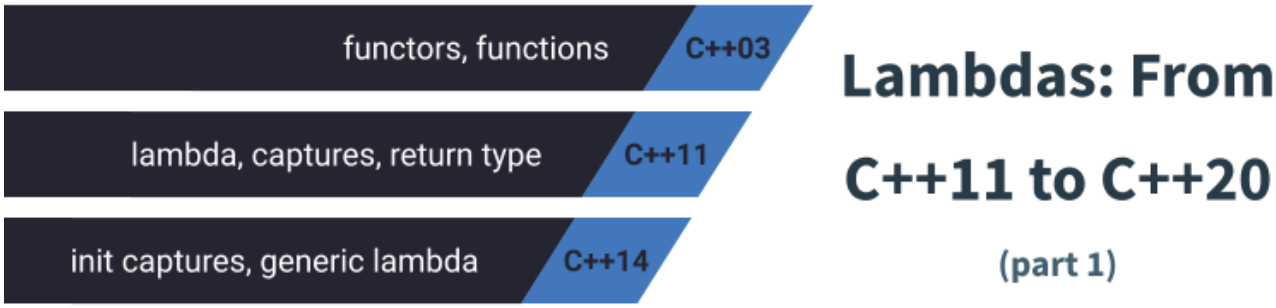
Lambdas: From C++11 to C++20, Part 1



Lambda expressions are one of the most powerful additions to C++11, and they continue to evolve with each new C++ language standard. In this article, we'll go through history and see the evolution of this crucial part of modern C++.

The second part is available:
Lambdas: From C++11 to C++20, Part 2

At one of our local C++ User Group meeting, 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!), describe it and form a separate article.

We'll start by learning about C++03 and the need of having compact, local functional expressions. Then we'll move on to C++11 and C++14. In the second part of the series, we'll see changes from C++17, and we'll even have a peek of what will happen in C++20.

Since the early days of STL, std::algorithms - like std::sort could take any callable object and call it on elements of the container. However, in C++03 it meant only function pointers and functors.

For example:
```
#include <iostream>
#include <algorithm>
#include <vector>

struct PrintFunctor {
    void operator()(int x) const {
        std::cout << x << std::endl;
    }
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), PrintFunctor());
}
```

Runnable code: @Wandbox

But the problem was that you had to write a separate function or a functor in a different scope than the invocation of the algorithm.

As a potential solution, you could think about writing a local functor class - since C++ always has support for that syntax. But that didn't work…

See this code:

```
int main() {
    struct PrintFunctor {
        void operator()(int x) const {
            std::cout << x << std::endl;
        }
    };

    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), PrintFunctor());
}
```

Try to compile it with -std=c++98 and you'll see the following error on GCC:

```
error: template argument for
'template<class _IIter, class _Funct> _Funct
std::for_each(_IIter, _IIter, _Funct)'
uses local type 'main()::PrintFunctor'
```

Basically, in C++98/03 you couldn't instantiate a template with a local type.

Because all of those limitations the Committee started to design a new feature, something that we can create and call "in place"… "lambda expressions"!

If we look at N3337 - the final draft of C++11, we can see a separate section for lambdas: [expr.prim.lambda].

Lambdas were added into the language in a smart way I think. They use some new syntax, but then the compiler "expands" it into a real class. This way we have all advantages (and disadvantages sometimes) of the real strongly typed language.

Here's a basic code example that also shows the corresponding local functor object:

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

int main() {
    struct {
        void operator()(int x) const {
            std::cout << x << '\n';
        }
    } someInstance;

    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), someInstance);
    std::for_each(v.begin(), v.end(), [] (int x) {
            std::cout << x << '\n';
        }
    );
}
```

Live example @WandBox

You can also check out CppInsights that shows how the compiler expands the code:

See this sample:
CppInsighs: lambda test

In the example the compiler transforms:

```
[] (int x) { std::cout << x << '\n'; }
```

Into something like that (simplified form):

```
struct {
    void operator()(int x) const {
        std::cout << x << '\n';
    }
} someInstance;
```

The syntax of the lambda expression:

```
[] ()   { code; }
^  ^  ^
|  |  |
|  |  optional: mutable, exception, trailing return, ...
|  |
|  parameter list
|
lambda introducer with capture list
```

Some definitions before we start:

From [expr.prim.lambda#2]:

> *The evaluation of a lambda-expression results in a prvalue temporary . This temporary is called the* ***closure object****.*

And from [expr.prim.lambda#3]:

> *The type of the lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type — called the* ***closure type****.*

A few examples of lambda expressions:

For example:

```
[](float f, int a) { return a*f; }
[](MyClass t) -> int { auto a = t.compute(); return a; }
[](int a, int b) { return a < b; }
```

Since the compiler generates some unique name for each lambda, there’s no way to know it upfront.

That’s why you have to use auto (or decltype)) to deduce the type.

```
auto myLambda = [](int a) -> double { return 2.0 * a; }
```

What’s more [expr.prim.lambda]:

> *The closure type associated with a lambda-expression has a deleted ([dcl.fct.def.delete]) default constructor and a deleted copy assignment operator.*

That’s why you cannot write:

```
auto foo = [&x, &y]() { ++x; ++y; };
decltype(foo) fooCopy;
```

This gives the following error on GCC:

```
error: use of deleted function 'main()::<lambda()>::<lambda>()'
       decltype(foo) fooCopy;
                     ^~~~~~~
note: a lambda closure type has a deleted default constructor
```

The code that you put into the lambda body is “translated” to the code in the operator() of the corresponding closure type.

By default it’s a const inline method. You can change it by specifying mutable after the parameter declaration clause:

```
auto myLambda = [](int a) mutable { std::cout << a; }
```

While a const method is not an “issue” for a lambda without an empty capture list… it makes a difference when you want to capture.

The [] does not only introduce the lambda but also holds a list of captured variables. It’s called “capture clause”.

By capturing a variable, you create a member copy of that variable in the closure type. Then, inside the lambda body, you can access it.

The basic syntax:

- [&] - capture by reference, all automatic storage duration variable declared in the reaching scope
- [=] - capture by value, a value is copied
- [x, &y] - capture x by value and y by a reference explicitly

For example:
```
int x = 1, y = 1;
{
    std::cout << x << " " << y << std::endl;
    auto foo = [&x, &y]() { ++x; ++y; };
    foo();
    std::cout << x << " " << y << std::endl;
}
```

You can play with the full example @Wandbox

While specifying [=] or [&] might be handy - as it captures all automatic storage duration variable, it’s clearer to capture a variable explicitly. That way the compiler can warn you about unwanted effects (see notes about global and static variable for example)

You can also read more in item 31 in “Effective Modern C++” by Scott Meyers: “Avoid default capture modes.”

And an important quote:

> *The C++ closures do not extend the lifetimes of the captured references.*

Mutable

By default operator() of the closure type is const, and you cannot modify captured variables inside the body of the lambda.

If you want to change this behaviour you need to add mutable keyword after the parameter list:
```
int x = 1, y = 1;
std::cout << x << " " << y << std::endl;
auto foo = [x, y]() mutable { ++x; ++y; };
foo();
std::cout << x << " " << y << std::endl;
```

In the above example, we can change the values of x and y… but those are only copies of x and y from the enclosing scope.

Capturing Globals

If you have a global value and then you use [=] in your lambda you might think that also a global is captured by value… but it’s not.
```
int global = 10;

int main()
{
    std::cout << global << std::endl;
    auto foo = [=] () mutable { ++global; };
    foo();
    std::cout << global << std::endl;
```

```
    [] { ++global; } ();
    std::cout << global << std::endl;
    [global] { ++global; } ();
}
```

Play with code @Wandbox

Only variables with automatic storage duration are captured. GCC can even report the following warning:
warning: capture of variable 'global' with non-automatic storage duration

This warning will appear only if you explicitly capture a global variable, so if you use [=] the compiler won't help you.

The Clang compiler is even more helpful, as it generates an error:
error: 'global' cannot be captured because it does not have automatic storage duration

See @Wandbox

Capturing Statics

Similarly to capturing a global variable, you'll get the same with a static variable:

```
#include <iostream>

void bar()
{
    static int static_int = 10;
    std::cout << static_int << std::endl;
    auto foo = [=] () mutable { ++static_int; };
    foo();
    std::cout << static_int << std::endl;
    [] { ++static_int; } ();
    std::cout << static_int << std::endl;
    [static_int] { ++static_int; } ();
}

int main()
{
   bar();
}
```

Play with code @Wandbox

The output:
10
11
12

And again, this warning will appear only if you explicitly capture a global variable, so if you use [=] the compiler won't help you.

Do you know what will happen with the following code:

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

struct Baz
{
    std::function<void()> foo()
    {
        return [=] { std::cout << s << std::endl; };
    }

    std::string s;
};
```

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

The code declares a Baz object and then invokes foo(). Please note that foo() returns a lambda (stored in std::function) that captures a member of the class.

Since we use temporary objects, we cannot be sure what will happen when you call f1 and f2. This is a dangling reference problem and generates Undefined Behaviour.

Similarly to:

```
struct Bar {
    std::string const& foo() const { return s; };
    std::string s;
};
auto&& f1 = Bar{"ala"}.foo(); // dangling reference
```

Play with code @Wandbox

Again, if you state the capture explicitly ([s]):

```
std::function<void()> foo()
{
    return [s] { std::cout << s << std::endl; };
}
```

The compiler will prevent you for making this mistake, by emitting errors:

```
In member function 'std::function<void()> Baz::foo()':
error: capture of non-variable 'Baz::s'
error: 'this' was not captured for this lambda function
...
```

See in this example @Wandbox

If you have an object that is movable only (for example unique_ptr), then you cannot move it to lambda as a captured variable. Capturing by value does not work, so you can only capture by reference… however this won't transfer the ownership, and it's probably not what you wanted.

```
std::unique_ptr<int> p(new int{10});
auto foo = [p] () {}; // does not compile....
```

If you capture a const variable, then the constness is preserved:

```
int const x = 10;
auto foo = [x] () mutable {
    std::cout << std::is_const<decltype(x)>::value << std::endl;
    x = 11;
};
foo();
```

Test code @Wandbox

In C++11 you could skip the trailing return type of the lambda and then the compiler would deduce the type for you.

Initially, return type deduction was restricted to lambdas with bodies containing a single return statement, but this restriction was quickly lifted as there were no issues with implementing a more convenient version.

See C++ Standard Core Language Defect Reports and Accepted Issues (thanks Tomek for finding the correct link!)

So since C++11, the compiler could deduce the return type as long as all of your return statements are convertible to the same type.

> *If all return statements return an expression and the types of the returned expressions after lvalue-to-rvalue conversion (7.1 [conv.lval]), array-to-pointer conversion (7.2 [conv.array]), and function-to-pointer conversion (7.3 [conv.func]) are the same, that common type;*

```
auto baz = [] () {
    int x = 10;
    if ( x < 20)
        return x * 1.1;
    else
        return x * 2.1;
};
```

Play with the code @Wandbox

In the above lambda, we have two returns statements, but they all point to double so the compiler can deduce the type.

IIFE - Immediately Invoked Function Expression

In our examples I defined a lambda and then invoked it by using a closure object… but you can also invoke it immediately:

```
int x = 1, y = 1;
[&]() { ++x; ++y; }(); // <-- call ()
std::cout << x << " " << y << std::endl;
```

Such expression might be useful when you have a complex initialisation of a const object.

```
const auto val = []() { /* several lines of code... */ }();
```

I wrote more about it in the following blog post: IIFE for Complex Initialization.

Conversion to function pointer

> *The closure type for a lambda-expression with no lambda-capture has a public non-virtual non-explicit const conversion function to pointer to function having the same parameter and return types as the closure type's function call operator. The value returned by this conversion function shall be the address of a function that, when invoked, has the same effect as invoking the closure type's function call operator.*

In other words, you can convert a lambda without captures to a function pointer.

For example

```
#include <iostream>

void callWith10(void(* bar)(int))
{
    bar(10);
}

int main()
{
    struct
    {
        using f_ptr = void(*)(int);

        void operator()(int s) const { return call(s); }
        operator f_ptr() const { return &call; }

    private:
        static void call(int s) { std::cout << s << std::endl; };
    } baz;

    callWith10(baz);
```

```
    callWith10([](int x) { std::cout << x << std::endl; });
}
```

Play with the code @Wandbox

The standard N4140 and lambdas: [expr.prim.lambda].

C++14 added two significant enhancements to lambda expressions:

- Captures with an initialiser
- Generic lambdas

The features can solve several issues that were visible in C++11.

Lambda return type deduction was updated to conform to the rules of auto deduction rules for functions.

[expr.prim.lambda#4]

> *The lambda return type is auto, which is replaced by the trailing-return-type if provided and/or deduced from return statements as described in [dcl.spec.auto].*

In short, we can create a new member variable of the closure type and then use it inside the lambda.

For example:

```
int main() {
    int x = 10;
    int y = 11;
    auto foo = [z = x+y]() { std::cout << z << '\n'; };
    foo();
}
```

It can solve a few problems, for example with movable only types.

Move

Now, we can move an object into a member of the closure type:

```
#include <memory>

int main()
{
    std::unique_ptr<int> p(new int{10});
    auto foo = [x=10] () mutable { ++x; };
    auto bar = [ptr=std::move(p)] {};
    auto baz = [p=std::move(p)] {};
}
```

Optimisation

Another idea is to use it as a potential optimisation technique. Rather than computing some value every time we invoke a lambda, we can compute it once in the initialiser:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>
#include <iostream>
#include <string>

int main()
{
    using namespace std::string_literals;
    std::vector<std::string> vs;
    std::find_if(vs.begin(), vs.end(), [](std::string const& s) {
```

```
     return s == "foo"s + "bar"s; });
    std::find_if(vs.begin(), vs.end(), [p="foo"s + "bar"s](std::string const& s) { return s == p; });
}
```

Initialiser can also be used to capture a member variable. We can then capture a copy of a member variable and don’t bother with dangling references.

For example

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

    std::string s;
};

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

Play with code @Wandbox

In foo() we capture a member variable by copying it into the closure type. Additionally, we use auto for the deduction of the whole method (previously, in C++11 we could use std::function).

Another significant improvement to Lambdas is a generic lambda.

Since C++14 you can now write:

```
auto foo = [](auto x) { std::cout << x << '\n'; };
foo(10);
foo(10.1234);
foo("hello world");
```

This is equivalent to using a template declaration in the call operator of the closure type:

```
struct {
    template<typename T>
    void operator()(T x) const {
        std::cout << x << '\n';
    }
} someInstance;
```

Such generic lambda might be very helpful when deducting type is hard.

For example:

```
std::map<std::string, int> numbers {
    { "one", 1 }, {"two", 2 }, { "three", 3 }
};

// each time entry is copied from pair<const string, int>!
std::for_each(std::begin(numbers), std::end(numbers),
    [](const std::pair<std::string, int>& entry) {
        std::cout << entry.first << " = " << entry.second << '\n';
    }
);
```

Did I make any mistake here? Does entry have the correct type?

.
.

.

Probably not, as the value type for std::map is std::pair<const Key, T>. So my code will perform additional string copies…

This can be fixed by using auto:

```
std::for_each(std::begin(numbers), std::end(numbers),
    [](auto& entry) {
        std::cout << entry.first << " = " << entry.second << '\n';
    }
);
```

You can play with code @Wandbox

What a story!

In this article, we started from the early days of lambda expression in C++03 and C++11, and we moved into an improved version in C++14.
You saw how to create a lambda, what's the basic structure of this expression, what's capture clause and many more.

In the next part of the article, we'll move to C++17, and we'll also have a glimpse of the future C++20 features.

The second part is available:
Lambdas: From C++11 to C++20, Part 2

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