C++ Lambda Story

The evolution of a powerful **modern C++** feature

From **C++03** to **C++20**

(BF) C++ STORIES

Bartłomiej Filipek

C++ Lambda Story

The evolution of a powerful modern C++ feature: from C++03 to C++20

Bartłomiej Filipek

This book is for sale at http://leanpub.com/cpplambda

This version was published on 2019-03-24

Contents

About the Book

This short book contains updated versions of two articles that appeared at bfilipek.com:

- Lambdas: From C++11 to C++20, Part 1[1]
- Lambdas: From C++11 to C++20, Part 2[2]

I updated the blog posts and added more examples and better descriptions.

The articles also are based on a live coding presentation given by Tomasz Kamiński at our local Cracow C++ User Group.

The book shows the story of lambda expressions, so we'll start with C++03, and then we'll move into the latest C++ standards.

- C++11 - early days. You'll learn about all the elements of a lambda expression and even some tricks. This is the longest chapter as we need to cover a lot of this.
- C++14 - updates. Once lambdas were adopted, we saw some options to improve them.
- C++17 - more improvements, especially by handling `this` pointer and allowing `constexpr`.
- C++20 - in this section we'll have a glimpse overview of the future.

Who This Book is For

This book is intended for all C++ developers who like to learn all about a modern C++ feature: lambda expressions.

[1] https://www.bfilipek.com/2019/02/lambdas-story-part1.html
[2] https://www.bfilipek.com/2019/03/lambdas-story-part2.html

Reader Feedback

If you spot an error, typo, a grammar mistake... or anything else (especially some logical issues!) that should be corrected, then please let us know!

Write your feedback to bartlomiej.filipek AT bfilipek.com.

You can also use this place:

- Leanpub Book's Feedback Page[3]

Code License

The code for the book is available under the Creative Commons License.

Formatting

The code is presented in a monospace font, similarly to the following example:

For longer examples:

title

```
#include <iostream>

int main() {
    std::string text = "Hello World";
    std::cout << text << '\n';
}
```

Or shorter snippets:

```
int foo() {
    return std::clamp(100, 1000, 1001);
}
```

[3] https://leanpub.com/cpplambda/feedback

Snippets of longer programs were usually shortened to present only the core mechanics.

Usually, source code uses full type names with namespaces, like `std::string`, `std::filesystem::`. However, to make code compact and present it nicely on a book page the namespaces sometimes might be removed, so they don't use space. Also, to avoid line wrapping longer lines might be manually split into two. In some cases, the code in the book might skip `include` statements.

Syntax Highlighting Limitations

The current version of the book might show some limitations regarding syntax highlighting.

For example:

- `if constexpr` - Link to Pygments issue: #1432 - C++ if constexpr not recognized (C++17)[4]
- The first menthod of a class is not highlighted - #1084 - First method of class not highlighted in C++[5]
- Teplate method is not highlighted #1434 - C++ lexer doesn't recognize function if return type is templated[6]
- Modern C++ attributes are sometimes not recognised properly

Other issues for C++ and Pygments: issues C++[7].

Online Compilers

Instead of creating local projects you can also leverage some online compilers. They offer a basic text editor and usually allow you to compile only one source file (the code that you edit). They are very handy if you want to play with a simple code example.

For example, many of the code samples for this book were created in Wandbox Online compiler and then adapted adequately for the book content.

Here's a list of some of the useful services:

[4] https://bitbucket.org/birkenfeld/pygments-main/issues/1432/c-if-constexpr-not-recognized-c-17
[5] https://bitbucket.org/birkenfeld/pygments-main/issues/1084/first-method-of-class-not-highlighted-in-c
[6] https://bitbucket.org/birkenfeld/pygments-main/issues/1434/c-lexer-doesnt-recognize-function-if
[7] https://bitbucket.org/birkenfeld/pygments-main/issues?q=c%2B%2B

- Coliru[8] - uses GCC 8.1.0 (as of July 2018), offers link sharing and a basic text editor, it's simple but very effective.
- Wandbox[9] - it offers a lot of compilers - for example, most of Clang and GCC versions - and also you can use boost libraries. Also offers link sharing.
- Compiler Explorer[10] - shows the compiler output from your code! Has many compilers to pick from.
- CppBench[11] - run a simple C++ performance tests (using google benchmark library).
- C++ Insights[12] - it's a Clang-based tool which does a source to source transformation. It shows how the compiler sees the code, for example by expanding lambdas, auto, structured bindings or range-based for loops.

There's also a nice list of online compilers gathered on this website: List of Online C++ Compilers[13].

[8] http://coliru.stacked-crooked.com/
[9] https://wandbox.org/
[10] https://gcc.godbolt.org/
[11] http://quick-bench.com/
[12] https://cppinsights.io/
[13] https://arnemertz.github.io/online-compilers/

About the Author

Bartłomiej Filipek is a C++ software developer with more than 11 years of professional experience. In 2010 he graduated from Jagiellonian University in Cracow with a Masters Degree in Computer Science.

Bartek currently works at Xara, where he develops features for advanced document editors. He also has experience with desktop graphics applications, game development, large-scale systems for aviation, writing graphics drivers and even biofeedback. In the past, Bartek has also taught programming (mostly game and graphics programming courses) at local universities in Cracow.

Since 2011 Bartek has been regularly blogging at his website: bfilipek.com. In the early days the topics revolved around graphics programming, and now the blog focuses on Core C++. He also helps as co-organizer at C++ User Group in Krakow. You can hear Bartek in one @CppCast episode where he talks about C++17, blogging and text processing.

Since October 2018, Bartek has been a C++ Expert for Polish National Body that works directly with ISO/IEC JTC 1/SC 22 (C++ Standard Committee). In the same month, Bartek was also awarded by Microsoft and got his first MVP title for years 2019/2020.

In his spare time, he loves assembling trains and Lego with his little son. And he's a collector of large Lego models.

Bartek is the author of C++17 In Detail

Acknowledgement

This short book wouldn't be possible without valuable input from C++ Expert Tomasz Kamiński (see Tomek's profile at Linkedin).

Tomek led a live coding presentation about "history" of lambdas at our local C++ User Group in Cracow:

Lambdas: From C++11 to C++20 - C++ User Group Krakow

A lot of examples used in this book comes from that session.

Also, I'd like to thank Dawid Pilarski (panicsoftware.com/about-me) and JFT for helpful feedback on many details of lambdas.

Last but not least, many updates to the book was possible because of the feedback and comments I got under the initial English articles. So I'd like to express gratitude to all readers of my blog!

Revision History

- 25th March 2019 - First Edition is live!

1. Lambdas in C++03

Since the early days of the Standard Library - 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:

A basic print functor

```
#include <algorithm>
#include <iostream>
#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[1]

The example defines a simple functor with `operator()`.

While function pointers were stateless, functors could do much more work and contain some state. One example is to count the number of invocations:

[1] https://wandbox.org/permlink/7OGJzJlfg40SSQUG

Functor with a state

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

struct PrintFunctor {
    PrintFunctor(): numCalls(0) { }

    void operator()(int x) const {
        std::cout << x << '\n';
        ++numCalls;
    }

    mutable int numCalls;
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    PrintFunctor visitor = std::for_each(v.begin(), v.end(), PrintFunctor());
    std::cout << "num calls: " << visitor.numCalls << '\n';
}
```

Runnable code: @Wandbox[2]

In the above example, we used a member variable to count the number of invocations of the call operator. Since the call operator is `const`, then I had to use a mutable variable.

We can also "capture" variables from the calling scope. To do that we have to create a member variable in our functor and initialise it in the constructor.

[2] https://wandbox.org/permlink/14xW15TQ7K0G0nxv

Functor with a captured variable

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

struct PrintFunctor {
    PrintFunctor(const std::string& str):
        strText(str), numCalls(0) { }

    void operator()(int x) const {
        std::cout << strText << x << '\n';
        ++numCalls;
    }

    const std::string strText;
    mutable int numCalls;
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    const std::string introText("Elem: ");
    PrintFunctor visitor = std::for_each(v.begin(), v.end(),
                                         PrintFunctor(introText));
    std::cout << "num calls: " << visitor.numCalls << '\n';
}
```

Runnable code: @Wandbox[3]

In this iteration, `PrintFunctor` takes an extra parameter to initialise a member variable. Then this variable is used in the call operator.

Issues

As you see functors are quite powerful. It's a separate class so you can design them any way you like.

[3] https://wandbox.org/permlink/Ogi8rPQbVGeCtYER

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:

Local Functor

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

    std::vector<int> v;
    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.

Motivation for a New Feature

In C++11 the Committee lifted the limitation of the template instantiation with a local type. So you can write your functor that is close to the use of it.

But C++11 also brought another idea to life: what if the compiler could "write" such small functor for developers? That would mean that with some new syntax we could create functors "in place" and open the doors for cleaner and more compact syntax.

And that was the start of "lambda expressions"!.

If we look at N3337[4] - the final draft of C++11, we can see a separate section for lambdas: [expr.prim.lambda][5].

Let's have a look at this new feature in the next chapter.

[4] https://timsong-cpp.github.io/cppwp/n3337/
[5] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda

2. Lambdas in C++11

Hooray! The C++ Committee heard Voices of C++03 developers, and since C++11 we got lambda expression!

Lambdas quickly become one of the most recognisable features of modern C++.

We can read the spec located under: N3337[1] - the final draft of C++11,

And the separate section for lambdas: [expr.prim.lambda][2].

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.

In this chapter you'll learn:

- the basic syntax of lambdas
- how to capture variables
- how to capture member variables
- what's the return type of a lambda
- what is a closure
- some edge cases
- conversion to a function pointer
- IIFE

Let's go!

[1] https://timsong-cpp.github.io/cppwp/n3337/
[2] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda

The Syntax

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

First Lambda and a Corresponding Functor

```
#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[3]

In the example the compiler transforms:

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

Into something like that (simplified form):

[3] https://wandbox.org/permlink/86wzD14LVEnMiO2Y

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

The syntax of the lambda expression:

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

Some definitions before we start:

From [expr.prim.lambda#2][4]:

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

And from [expr.prim.lambda#3][5]:

> 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:

[4] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#2
[5] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#3

```
[]{} // the simplest lambda
[](float f, int a) { return a*f; }
[](MyClass t) -> int { auto a = t.compute(); return a; }
[](int a, int b) { return a < b; }
[x](int a, int b) mutable { return a < b; ++x; }
```

The Type of a Lambda

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][6]:

> 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
```

Another aspect is that if you have two lambdas:

[6] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#19

```
auto firstLam = [](int x) { return x*2; };
auto secondLam = [](int x) { return x*2; };
```

Then their types are different! Even if the "code behind" is the same... after all the compiler is required to declare two unique unnamed types for each lambda.

You can, however copy lambdas:

Copying lambdas

```
#include <type_traits>

int main() {
    auto firstLam = [](int x) { return x*2; };
    auto secondLam = firstLam;
    static_assert(std::is_same_v<decltype(firstLam), decltype(secondLam)>);
}
```

If you copy a lambda, then you also copy its state. This is important when we'll talk about capture variables. Then a closure type will store such variable as a member field.

A peek into the future

In C++20 a stateless lambda will be default constructible and assignable.

The Call Operator

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 variables from the local scope.

And the capture clause is a topic of the next section:

Captures

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.

We did a similar thing for `PrintFunctor` in the C++03 Chapter. In that class, we added a member variable `std::string strText;` which was initialised in the constructor.

The basic syntax for captures:

- `[&]` - 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:

Capturing a Variable

```
std::string s {"Hello World"};
auto foo = [str]() { std::cout << str << '\n'; };
foo();
```

For the above lambda, the compiler might generate the following local functor:

A Possible Compiler Generated Functor, Single Variable

```
struct _unnamedLambda {
    _unnamedLambda(std::string s) : str(s) { }

    void operator()() const {
        std::cout << str << '\n';
    }

    std::string str;
};
```

A variable is passed into the constructor that is conceptually called in a place of lambda declaration.

To be precise the standard mentions in [expr.prim.lambda#21][7]:

> When the lambda-expression is evaluated, the entities that are captured by copy are used to direct-initialize each corresponding non-static data member of the resulting closure object.

A possible constructor that I showed above (`_unnamedLambda`) is only for demonstration purpose, as the compiler might implement it differently and won't expose it.

Capturing Two Variables by Reference

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

For the above lambda, the compiler might generate the following local functor:

A Possible Compiler Generated Functor, Two References

```
struct _unnamedLambda {
    _unnamedLambda(int& a, int& b) : x(a), y(b) { }

    void operator()() const {
        ++x; ++y;
    }

    int& x;
    int& y;
} someInstance;
```

Since we capture `x` and `y` by reference, the closure type will contain member variables that are also references.

[7] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#21

You can play with the full example @Wandbox[8]

The value of the value-captured variable is at the time the lambda is defined - not when it is used! The value of a ref-captured variable is the value when the lambda is used - not when it is defined.

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."

The C++ closures do not extend the lifetimes of the captured references. Be sure that the capture variable still lives when lambda is invoked.

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:

Capturing Two Variables by Copy

```
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`. Of course, since those are only copies of `x` and `y` from the enclosing scope, we don't see their new values after `foo` is invoked.

On the other hand, if you capture by reference then, in a non-mutable lambda, you cannot rebind a reference, but you can change the referenced variable.

[8] https://wandbox.org/permlink/da9ltcv53ECxnoEk

Capturing a Variable by Reference

```
int x = 1;
std::cout << x << '\n';
auto foo = [&x]() { ++x; };
foo();
std::cout << x << '\n';
```

In the above example, the lambda is not `mutable`, but it can change the referenced value.

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.

Capturing Globals

```
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[9]

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:

[9] https://wandbox.org/permlink/hsS8K0I6PrRyX45Z

```
error: 'global' cannot be captured because it does not have automatic storage d\
uration
```

See @Wandbox[10]

Capturing Statics

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

Capturing Static Variables

```
#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[11]

The output:

[10] https://wandbox.org/permlink/p5Ro10V3l0tLcYkk
[11] https://wandbox.org/permlink/YSF2px6Sjy7z5GqF

```
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.

Capturing a Class Member And `this`

Things get a bit more complicated where you're in a class method:

Error when capturing a member variable

```
#include <iostream>

struct Baz {
    void foo() {
        auto lam = [s]() { std::cout << s; };
        lam();
    }

    std::string s;
};

int main() {
    Baz b;
    b.foo();
}
```

Runnable code @Wandbox[12]

The code tries to capture `s` which is a member variable. But the compiler will emit an error message:

[12] https://wandbox.org/permlink/mp5VgqIyu5LWLn0f

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

To solve this issue, you have to capture the `this` pointer. Then you'll have access to member variables.

We can update the code to:

```
struct Baz {
    void foo() {
        auto lam = [this]() { std::cout << s; };
        lam();
    }

    std::string s;
};
```

No compiler errors are generated now.

You can also use `[=]` or `[&]` to capture `this` (they both have the same effect!)

But please notice that we captured `this` by value... to a pointer. So you have access to the member variable, not its copy.

In C++11 (and even in C++14) you cannot write:

```
auto lam = [*this]() { std::cout << s; };`
```

To capture a copy of the object.

If you use your lambdas in the context of a single method then capturing `this` will be fine. But how about more complicated cases?

Do you know what will happen with the following code?

Returning a Lambda From a Method

```
#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[13]

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

[13] https://wandbox.org/permlink/ntaWn7p4MVVT6fZj

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

All in all capturing `this` might get tricky when a lambda can outlive the object itself. This might happen when you use async calls or multithreading.

We'll return to that topic in the C++17 chapter.

Move-able-only Objects

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....
```

Preserving Const

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[14]

[14] https://wandbox.org/permlink/pbnGo223HNdOoNLQ

Return Type

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[15] [16]

So since C++11, the compiler could deduce the return type as long as all of your return statements are of 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[17]

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

In C++14 return type of lambda will be updated to adapted to the rules of `auto` type deduction for regular functions.

[15] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#975
[16] Thanks Tomek for finding the correct link!
[17] https://wandbox.org/permlink/kVKjlBObC9futJNV

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[18].

Conversion to a Function Pointer

If your lambda doesn't capture then:

> 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

[18] https://www.bfilipek.com/2016/11/iife-for-complex-initialization.html

Conversion to a Function Pointer

```
#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[19]

Summary

In this chapter, you learned how to create and use lambda expressions. I described the syntax, capture clause, type of the lambda, and more.

Lambda Expressions become one of the significant marks of Modern C++. With more use cases developers also saw possibilities to improve lambdas. And that's why you can now move to the next chapter and see updates that the Committee added in C++14.

[19] https://wandbox.org/permlink/tSZDkOpqQl4EdTp6

3. Lambdas in C++14

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

- Captures with an initialiser
- Generic lambdas

Plus, the Standard also updated some rules, for example:

- Default parameters for lambdas
- Return type as `auto`

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

You can see the specification in N4140[1] and lambdas: [expr.prim.lambda][2].

[1] https://timsong-cpp.github.io/cppwp/n4140/
[2] https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda

Default Parameters for Lambdas

In C++14 you can use default paramaters in a function call. This is a small feature, but makes lambda more like a regular function.

Lambda with Default Paramter

```
#include <iostream>

int main() {
    auto lam = [](int x = 10) { std::cout << x << '\n'; };
    lam();
    lam(100);

    return 0;
}
```

What's interesting is that GCC and Clang supported this feature since C++11.

Return Type

In C++14 Lambda return type deduction was updated to conform to the rules of `auto` deduction rules for functions.

[expr.prim.lambda#4][3]:

> 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][4]

If you have multiple return statements they all have to deduce the same type:

[3] https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda#4
[4] https://timsong-cpp.github.io/cppwp/n4140/dcl.spec.auto

```
auto foo = [] (int x) {
    if (x < 0)
        return x * 1.1f; // float!
    else
        return x * 2.1;  // double!
};
```

The above code won't compile as the first return statement retuns `float` while the second deduced `double`.

Another important concept related to the return type is that we can stop using `std::function` to return a lambda!

The compiler can deduce the proper closure type:

```
auto CreateMulLambda(int x) {
    return [x](int param) { return x * param; };
}

auto lam = CreateMulLambda(10);
```

Captures With an Initializer

Now some bigger updates!.

In lambda expression you can capture variables. The compiler expands that capture syntax and creates member variables of the closure type.

Now, in C++14, you can create new member variables and initialise them in the capture clause. Then you can use those variables inside the lambda.

For example:

Simple Capture With an Initialiser

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

In the example above the compiler will generate a new member variable and initialise it with `x+y`.

So conceptually it will resolve into:

```
struct _unnamedLambda {
    void operator()() const {
        std::cout << z << '\n';
    }

    int z;
} someInstance;
```

And `z` will be directly initialised (with `x+y`) when the lambda expresion is evaluated.

This new feature can solve a few problems, for example with movable only types.

Let's review them now.

Move

Previously, in C++11, you couldn't capture a unique pointer by value.

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

Capturing a movable only 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)] {};
}
```

Thanks to the initialiser you can assign the proper value even for `unique_ptr`.

Optimisation

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

Creating a string for lambda

```
#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;
        }
    );
}
```

The code above shows two calls to `std::find_if`. In the first scenario we don't capture anything and just compare the input value against `"foo"s + "bar"s`. Every time the lambda is invoked a temporary value that will store the sum of those strings will be created.

The second call to `find_if` shows an optimisation: we create a capture variable `p` that computes the sum of strings once. Then we can safely refer to it in the lambda body.

Capturing a Member Variable

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:

Capturing a member variable

```
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[5]

[5] https://wandbox.org/permlink/zIzWARqYIGnFz3jN

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`).

Generic Lambdas

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");
```

Please notice `auto x` as a parameter to the lambda.

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;
```

With generic lambdas you're not restricted to using `auto x`, you can add any qualifiers as with other `auto` variables.

Such generic lambda might be very helpful when type deduction is tricky.

For example:

Correct type for map iteration

```
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[6]

Bonus - LIFTing with lambdas

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):

[6] https://wandbox.org/permlink/jUxlrWasTCBDVEYr

Calling function overloads

```
// 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[7]

Summary

As you saw in this chapter C++14 brought several key improvements to lambda expressions. Since C++14 you can now declare new variables to use inside a lambda scope, and you can also use them effieicnely in template code. In the next chapter we'll dive into C++17 which brings more updates!

[7] https://wandbox.org/permlink/r81jASiPPmYXTOmx

4. Lambdas in C++17

The standard (draft before publication) N659[1] and the lambda section: [expr.prim.lambda][2].

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.

[1] https://timsong-cpp.github.io/cppwp/n4659/
[2] https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda

`constexpr` Lambda Expressions

Since C++17, if possible, the standard defines `operator()` for the lambda type implicitly as `constexpr`:

From expr.prim.lambda #4[3]:

> 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?

[3] https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda#closure-4

constexpr lambda

```
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[4]

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 lambda, capture

```
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);
}
```

[4] https://wandbox.org/permlink/5fr5NCQAvvEKsWKq

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[5]) (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:

Capture of `*this`

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 as I described in the C++14 chapter.

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

[5] https://en.cppreference.com/w/cpp/language/lambda

Capturing `*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[6]

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:

Capturing `this` to call a method

```
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 to use init capture of `this`:

[6] https://wandbox.org/permlink/i8m9UeAHa2YsqkgL

```
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!

Some Guides

Ok, so should we capture `[this]` or `[*this]` why is this important?

In most cases, when you work inside the scope of a class, then `[this]` (or `[&]`) is perfectly fine. There's no extra copy which is essential when your objects are large.

You might consider `[*this]` when you really want a copy, and when there's a chance a lambda will outlive the object.

This might be crucial for avoiding data races in async or parallel execution. Also, in the async/multithreading execution mode, the lambda might outlive the object, and then `this` pointer might no longer be alive.

Summary

In this chapter you've seen that C++17 joined two important elements of C++: `constexpr` with lambdas. Now you can use lambdas in `constexpr` context! What's more the C++17 Standard also addressed the capturing `this` problem.

In the next chapter, we'll have a glimpse overview of the future that comes with C++20.

5. Future with C++20

Let's have a glimpse overview of the changes that will get in C++20.

In this chapter you'll see:

- What will change in C++20
- What are the new options to capture `this`
- What are template lambdas

Quick Overview of the Changes

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

- Allow `[=, this]` as a lambda capture - P0409R2[1] and Deprecate implicit capture of this via `[=]` - P0806[2]
- Pack expansion in lambda init-capture: `...args = std::move(args)](){}` - P0780[3]
- `static`, `thread_local`, and lambda capture for structured bindings - P1091[4]
- template lambdas (also with concepts) - P0428R2[5]
- Simplifying implicit lambda capture - P0588R1[6]
- Default constructible and assignable stateless lambdas - P0624R2[7]
- Lambdas in unevaluated contexts - P0315R4[8]

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

For example with P1091[9] 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:

[1] https://wg21.link/p0409r2
[2] https://wg21.link/P0806
[3] https://wg21.link/P0780
[4] https://wg21.link/P1091
[5] https://wg21.link/P0428R2
[6] https://wg21.link/P0588R1
[7] https://wg21.link/P0624R2
[8] https://wg21.link/P0315R4
[9] https://wg21.link/P1091

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

Play with code @Wandbox[10]

The warning appears, because even with `[=]` you'll capture `this` as a pointer. So it's better to write what you want explicitly: `[=, this]`, or `[=, *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;
```

C++20 Standard is feature complete, so we shouldn't expect any new features that will relate to lambdas. But even the elements already voted in might slightly change, so don't treat the above list as obsolete, but rather as work in progress.

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

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:

[10] https://wandbox.org/permlink/yRosU85B0Q9LnwOv

```
auto foo = [](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 = in\
t]':
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[11]

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.

[11] https://wandbox.org/permlink/gupbJfUfHHQ2y48q

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;
}
```

Summary

In this chapter, you saw more changes to lambdas. Lambas are quite a stable feature of modern C++, so most of the new elements relate to quite advanced uses. For example unevaluated contexts or capturing structured bindings. There are also "extensions" - for instance in template lambdas. In most of the cases using generic lambda will do the work, but for advanced scenarios, you might explicitly want to declare a template argument.

References

- C++11 - [expr.prim.lambda][12]
- C++14 - [expr.prim.lambda][13]
- C++17 - [expr.prim.lambda][14]
- Lambda Expressions in C++ | Microsoft Docs[15]
- Demystifying C++ lambdas - Sticky Bits - Powered by FeabhasSticky Bits – Powered by Feabhas[16]
- The View from Aristeia: Lambdas vs. Closures[17]
- Simon Brand - Passing overload sets to functions[18]
- Jason Turner - C++ Weekly - Ep 128 - C++20's Template Syntax For Lambdas[19]
- Jason Turner - C++ Weekly - Ep 41 - C++17's constexpr Lambda Support[20]

[12] https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda
[13] https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda
[14] https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda
[15] https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2017
[16] https://blog.feabhas.com/2014/03/demystifying-c-lambdas/
[17] http://scottmeyers.blogspot.com/2013/05/lambdas-vs-closures.html
[18] https://blog.tartanllama.xyz/passing-overload-sets/
[19] https://www.youtube.com/watch?v=ixGiE4-1GA8&
[20] https://www.youtube.com/watch?v=kmza9U_niq4